The Dissertation Committee for Kostas Ferles
certifies that this is the approved version of the following dissertation:

Practical Formal Methods for Software Analysis and Development

Committee:

Işıl Dillig, Supervisor

Swarat Chaudhuri

James Bornholt

Alex Aiken

Kenneth McMillan

Practical Formal Methods for Software Analysis and Development

by

Kostas Ferles, M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2020

Practical Formal Methods for Software Analysis and Development

Publication No. ________________

Kostas Ferles, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Işıl Dillig

Formal methods techniques for improving software correctness and reliability fall into two categories, namely, program analysis and program synthesis. Program analysis techniques automatically find defects (or prove the absence thereof) in existing software. In a dual way, program synthesis techniques generate correct-by-construction code from high-level specifications. In this thesis, we propose an array of formal method techniques that further improve the state-of-the-art of program analysis techniques, while also applying program synthesis techniques in previously unexplored domains.

Broadly speaking, the long history of program analysis can be summarized as the battle between precision and scalability. As a first step in this thesis, we propose a technique called *program trimming* that helps arbitrary safety-checking tools to achieve a better balance between precision and scalability. In a nutshell, program trimming semantically simplifies the program by eliminating provably correct execution paths. Because the number of execution paths is a typical scalability bottleneck for some techniques (e.g., symbolic execution) and a source of imprecision for others (e.g., abstract interpretation), program trimming can be used to improve both precision and scalability of program analysis tools. We have implemented our technique in a tool called TRIMMER and showed that Trimmer significantly improves the behavior of two different program analyzers over a set of challenging verification benchmarks.

Program synthesis, on the other hand, has only recently started to appear in more practical aspects of software development. Formal method techniques in this area aim to ease programming for several domains while targeting a broad range of programmers, from novices to experts. In this thesis, we propose a novel program synthesis technique, implemented in a tool called EXPRESSO, that aids experts in writing correct and efficient concurrent programs. Specifically, EXPRESSO allows programmers to implement concurrent programs using the implicit signaling paradigm, where the programmer specifies the condition under which a thread should block but she does not need to worry about explicitly signaling other threads when this condition becomes true. Given such an implicit signaling program, EXPRESSO generates an efficient and correct-by-construction program that does not contain deadlocks caused by improper signaling. Our evaluation shows that EXPRESSO is able to synthesize efficient implementations of real-world monitors with performance comparable to the one written by programming experts.

Finally, we observe that certain monitors require their clients to use their API in a manner that conforms to a context-free specification. Statically verifying that a client conforms to a context-free API protocol cannot be handled by any prior technique. To rectify this and ensure that client applications properly use such protocols, we propose CFPChecker, a tool that verifies the correct usage of context-free API protocols. Given a program, $\mathcal{P}$, and an API protocol expressed as a parameterized context-free grammar, $\mathcal{G}_S$, CFPChecker either proves that $\mathcal{P}$ conforms to $\mathcal{G}_S$ or provides a genuine program trace that demonstrates an API misuse. We have evaluated our proposed technique on a wide variety of popular context-free API protocols and several clients drawn from popular open-source applications. Our experiments show that CFPChecker is effective in both verifying the correct usage and finding counterexamples of context-free API protocols.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With software becoming a necessity in our daily lives, ensuring its correctness has become of paramount importance. Application development has evolved to an extremely complex process that requires developers to build on top of existing complicated layers of software libraries and frameworks, and thus, perpetually increasing the number of these already complex software components. Therefore, ensuring application correctness is a daunting endeavor that requires proving the overall composition of the application's sub-components is correct. In this thesis, we propose a suite of formal method techniques that aim to help developers in proving the correctness of their code or writing correct-by-construction code in the first place. Specifically, the techniques proposed in this thesis achieve the following:

1. Increase the precision and scalability of existing state-of-the-art program analysis techniques.

2. Introduce novel program synthesis techniques in the context of concurrent programming.

3. Extend the class of properties that safety-checking tools are able to handle.

In what follows, we provide a high-level overview of the techniques developed for each of the above directions.

Scalable & Precise Program Analysis. As a first step, we describe a new program simplification technique called *program trimming* that aims to improve the scalability and precision of safety checking tools. Given a program $\mathcal{P}$, program trimming generates a new program $\mathcal{P}'$ such that $\mathcal{P}$ and $\mathcal{P}'$ are *equi-safe* (i.e., $\mathcal{P}'$ has a bug if and only if $\mathcal{P}$ has a bug), but $\mathcal{P}'$ has fewer execution paths than $\mathcal{P}$. Since many program analyzers are sensitive to the number of execution paths, program trimming has the potential to improve the effectiveness of safety checking tools.

In addition to introducing the concept of program trimming, we also present a lightweight static analysis that can be used as a pre-processing step to remove program paths while retaining equi-safety. We have implemented the proposed technique in a tool called TRIMMER and evaluate it in the context of two program analysis techniques, namely abstract interpretation and dynamic symbolic execution. Our experiments show that program trimming significantly improves the effectiveness of both techniques.

Program Synthesis for Concurrent Programming. Explicit signaling between threads is a perennial cause of bugs in concurrent programs. While there are several run-time techniques to *automatically* notify threads upon the availability of some shared resource, such techniques are not widely-adopted due to their run-time overhead. This thesis proposes a new solution based on static analysis for automatically generating a performant *explicit-*

signal program from its corresponding *implicit-signal* implementation. The key idea is to generate verification conditions that allow us to minimize the number of required signals and unnecessary context switches, while guaranteeing semantic equivalence between the source and target programs. We have implemented our method in a tool called Expresso and evaluate it on challenging benchmarks from prior work and open-source software. Expresso-generated code significantly outperforms past automatic signaling mechanisms (avg. 1.56x speedup) and closely matches the performance of hand-optimized explicit-signal code.

Verifying Context-Free API Protocols. There exists extensive prior work that automatically verify the correct usage of API protocols. All prior work focuses on API protocols that can only be expressed as a regular language. However, 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 thesis 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 CF-PChecker and evaluate it on 10 popular Java applications that use at least one API with a context-free specification. Our evaluation shows that CF-PChecker 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 CF-PChecker enables verification of safety properties that are beyond the reach of existing tools.

The rest of the thesis is described as follows. Chapter 2 presents program trimming. Chapter 3 describes our synthesis technique for automatic signal-placement. Chapter 4 describes our technique for verifying correct usage of context-free API protocols. Chapter 6 list some potential future directions. Last, Chapter 5 presents related work to the techniques proposed in this thesis.

Chapter 2

Failure-Directed Program Trimming

Due to its potential to dramatically simplify programs with respect to a certain criterion (e.g., the value of a program variable at a given location), program slicing [210] has been the focus of decades of research in the program analysis community [204]. In addition to being useful for program understanding, slicing also has the potential to improve the scalability of bug-finding and verification tools by removing irrelevant code snippets with respect to some property of interest. Yet, despite this potential, relatively few bug-finding and verification tools use slicing as a pre-processing step.

In this thesis, we argue that existing notions of a "program slice" do not adequately capture the kinds of program simplification that are beneficial to safety checking tools. Instead, we propose a new semantic program simplification technique called *program trimming*, which removes *program paths* that are irrelevant to the safety property of interest. Given a program $\mathcal{P}$, program trimming generates a simplified program $\mathcal{P}'$ such that $\mathcal{P}'$ violates a safety property if and only if the original program $\mathcal{P}$ does (i.e., $\mathcal{P}$ and $\mathcal{P}'$ are *equi-safe*). However, $\mathcal{P}'$ has the advantage of containing fewer execution paths than $\mathcal{P}$. Since the scalability and precision of many program analyzers depend on the number of program paths, program trimming can have a positive impact on many kinds of program analyses, particularly those that are *not* property directed.

To illustrate the difference between the standard notion of program slicing and our proposed notion of program trimming, consider the following very simple program, where $\star$ indicates a non-deterministic value (e.g., user input):

```
1 x := ⋆; y := ⋆;
2 if (y > 0) { while (x < 10) { x := x + y; } }
3 else { x := x - 1; }
4 assert x > 0;
```

Suppose that our goal is to prove the assertion; so, we are interested in the value of `x` at line 4. Now, every single statement in this program is relevant to determining the value of `x`; hence, there is nothing that can be removed using program slicing. However, observe that the `then` branch of the `if` statement is actually irrelevant to the assertion. Since this part of the program can never result in a program state where the value of `x` is less than 10, lines 2 and 3 can be simplified without affecting whether or not the assertion can fail. Hence, for the purposes of safety checking, the above program is equivalent to the following much simpler trimmed program $\mathcal{P}'$:

```
1 x := ⋆; y := ⋆;
2 assume y <= 0;
3 x := x - 1;
4 assert x > 0;
```

Observe that $\mathcal{P}'$ contains far fewer paths compared to the original program $\mathcal{P}$. In fact, while $\mathcal{P}$ contains infinitely many execution paths, the trimmed program $\mathcal{P}'$ contains only two, one through the successful and one through the failing branch of the assertion. Consequently, program analyzers that eagerly explore all program paths, such as bounded model checkers [30, 64] and symbolic execution engines [154], can greatly benefit from program trimming in terms of scalability. Furthermore, since many static analyzers (e.g., abstract interpreters [67]) typically lose precision at join points of the control flow graph, program trimming can improve their precision by removing paths that are irrelevant to a safety property.

Motivated by these observations, this thesis introduces the notion of failure-directed program trimming and presents a lightweight algorithm to remove execution paths in a way that guarantees equi-safety. The key idea underlying our approach is to statically infer *safety conditions*, which are sufficient conditions for correctness and can be computed in a lightweight way. Our technique then negates these safety conditions to obtain *trimming conditions*, which are necessary conditions for the program to fail. The trimming conditions are used to instrument the program with assumptions such that program paths that violate an assumption are pruned.

Program trimming is meant as a lightweight but effective pre-processing step for program analyzers that check safety. We have implemented our proposed trimming algorithm in a tool called TRIMMER and used it to pre-process hundreds of programs, most of which are taken from the software verification competition (SV-COMP) [27]. We have also evaluated the impact of trimming in the context of two widely-used program analysis techniques, namely abstract interpretation [67] and dynamic symbolic execution [43, 106]. Our experiments with Crab [102, 103] (an abstract interpreter) show that program trimming can considerably improve the precision of static analyzers. Furthermore, our experiments with Klee [42] (a dynamic symbolic execution tool) show that program trimming allows the dynamic symbolic execution engine to find more bugs and verify more programs within a given resource limit.

To summarize, we make the following key contributions:

- We introduce the notion of program trimming as a new kind of program simplification technique.
- We propose an effective and lightweight inference engine for computing safety conditions.
- We describe a modular technique for instrumenting the program with trimming conditions.
- We demonstrate empirically that program trimming has a significant positive impact on the effectiveness of program analyzers. For instance, the cheapest configuration of Crab (an abstract interpreter) with trimming proves 21% more programs safe than the most expensive configuration of Crab without trimming in *less than 70% of the time*. In the context of a dynamic symbolic execution engine (Klee), trimming increases both the number of uncovered bugs by up to 30% and the number of verified programs by up to 18% while reducing the running time by up to 30%.

2.1 Guided Tour

The running example, shown in Figure 2.1, is written in C extended with `assume` and `assert` statements. Note that the example is intentionally quite artificial to illustrate the main ideas behind our technique. Procedure `main` assigns a non-deterministic integer value to variable `m` and computes its factorial using the recursive `fact` procedure. The (light and dark) gray boxes are discussed below and *should be ignored for now.* We examine two variations of this example: one for dynamic symbolic execution (DSE) engines and another for abstract interpreters (AI).

Motivation #1: scalability. First, let us ignore the assertion on line 7 and only consider the one on line 15. Clearly, this assertion cannot fail unless `m` is equal to 123. Observe that procedure `main` contains infinitely many execution paths because the number of recursive calls to `fact` depends on the value of `m`, which is unconstrained. Consequently, a dynamic symbolic execution engine, like KLEE, would have to explore (a number of) these paths until it finds the bug or exceeds its resource limit. However, there is only one buggy execution path in this program, meaning that the dynamic symbolic execution engine is wasting its resources exploring paths that cannot possibly fail.

Our approach. Now, let us see how program trimming can help a symbolic execution tool in the context of this example. As mentioned in Section 2, our program trimming technique first computes *safety conditions*, which are sufficient conditions for the rest of the program to be correct. In

```
int fact(int n) {
  assume 0 <= n;
  assume n != 0;                // AI
  int r = 1;
  if (n != 0)
    r = n * fact(n - 1);
  assert n != 0   r == 1;     // AI
  return r;
}

void main() {
  int m = ⋆
  assume m == 123;              // DSE
  int f = fact(m);
  assert m != 123   f == 0;   // DSE
}
```

Figure 2.1: Running example illustrating program trimming.

this sense, standard *weakest preconditions* [79] are instances of safety conditions. However, automatically computing safety conditions precisely, for instance via weakest precondition calculi [79, 163], abstract interpretation [67], or predicate abstraction [16, 109], can become very expensive (especially in the presence of loops or recursion), making such an approach unsuitable as a preprocessing step for program analyzers that already check safety. Instead, we use lightweight techniques to infer safety conditions that describe a subset of the safe executions in the program. That is, the safety conditions inferred by our approach can be stronger than necessary, but they are still useful for ruling out many program paths that "obviously" cannot violate a safety property.

In contrast to a safety condition, a *trimming condition* at a given program point reflects a *necessary* condition for the rest of the program execution to fail. Since a necessary condition for a property $\neg Q$ can be obtained using the negation of a sufficient condition for Q, we can compute a valid trimming condition for a program point π as the negation of the safety condition at π. Thus, our approach trims the program by instrumenting it with assumptions of the form `assume` ϕ, where ϕ is the negation of the safety condition for that program point. Since condition ϕ is, by construction, necessary for the program to fail, the trimmed program preserves the safety of the original program. Moreover, since execution terminates as soon as we encounter an assumption violation, instrumenting the program with trimming conditions prunes program paths in a semantic way.

Program trimming on this example. Revisiting our running example from Figure 2.1, the safety condition right after line 14 is `m != 123 || f == 0`. Since procedure `fact` called at line 14 neither contains any assertions nor modifies the value of `m`, a valid safety condition right before line 14 is `m != 123`. Indeed, in executions that reach line 14 and satisfy this safety condition, the assertion does not fail. We can now obtain a sound trimming condition by negating the safety condition. This allows us to instrument the program with the `assume` statement shown in the dark gray box of line 13. Any execution that does not satisfy this condition is correct and is effectively removed by the `assume` statement in a way that preserves safety. As a result, a dynamic symbolic execution tool running on the instrumented program will only explore the single execution path containing the bug and will not waste any resources on provably correct paths. Observe that a bounded model checker would similarly benefit from this kind of instrumentation.

Motivation #2: precision. To see how our approach might improve the precision of program analysis, let us ignore the assertion on line 15 and only consider the one on line 7. Since `n = 0` implies `r = 1` on line 7, this assertion can clearly never fail. However, an abstract interpreter, like CRAB, using intervals [67] cannot prove this assertion due to the inherent imprecision of the underlying abstract domain. In particular, the abstract interpreter knows that `n` is non-negative at the point of the assertion but has no information about `r` (i.e., its abstract state is $\top$). Hence, it does not have sufficient information to discharge the assertion at line 7.

Suppose, however, that our technique can infer the safety condition `n = 0` on line 3. Using this condition, we can now instrument this line with the trimming condition `n != 0`, which corresponds to the assumption in the light gray box. If we run the same abstract interpreter on the instrumented program, it now knows that `n` is strictly greater than 0 and can therefore prove the assertion even though it is using the same interval abstract domain. Hence, as this example illustrates, program trimming can also be useful for improving the precision of static analyzers in verification tasks.

2.2 Program Trimming

In this section, we formally present the key insight behind failure-directed program trimming using a simple imperative language in the style of IMP [212], augmented with `assert` and `assume` statements. This lays the foundation for understanding the safety condition inference, which is described in the next section and is defined for a more expressive language. Here, we present the semantics of the IMP language using big-step operational semantics, specifically using judgments of the form $\langle \sigma, s \rangle \Downarrow_{\varphi} \sigma'$ where:

- s is a program statement,
- σ, σ' are *valuations* mapping program variables to values,
- $\varphi \in \{\lightning, \Diamond, \checkmark\}$ indicates whether an assertion violation occurred ($\lightning$), an assumption was violated ($\Diamond$), or neither assertion nor assumption violations were encountered (denoted $\checkmark$).

We assume that the program terminates as soon as an assertion or assumption violation is encountered. We also ignore non-determinism to simplify the presentation.

Definition 1. (Failing execution) *We say that an execution of s under σ is* failing *iff $\langle s, \sigma \rangle \Downarrow_{\lightning} \sigma'$, and* successful *otherwise.*

In other words, a failing execution exhibits an assertion violation. Executions with *assumption* violations also terminate immediately but are not considered failing.

Definition 2. (Equi-safety) *We say that two programs s, s' are* equi-safe *iff, for all valuations σ, we have:*

$$\langle s, \sigma \rangle \Downarrow_{\lightning} \sigma' \iff \langle s', \sigma \rangle \Downarrow_{\lightning} \sigma'$$

In other words, two programs are equi-safe if they exhibit the same set of failing executions starting from the same state σ. Thus, program s' has a bug if and only if s has a bug.

As mentioned in Section 2, the goal of program trimming is to obtain a program s' that (a) is equi-safe to s and (b) can terminate early in successful executions of s:

Definition 3. (Trimmed program) *A program s' is a* trimmed version *of s iff s, s' are equi-safe and*

$$\begin{array}{ll} (1) & \langle s, \sigma \rangle \Downarrow_{\checkmark} \sigma' \implies \langle s', \sigma \rangle \Downarrow_{\checkmark} \sigma' \vee \langle s', \sigma \rangle \Downarrow_{\diamondsuit} \sigma'' \\ (2) & \langle s, \sigma \rangle \Downarrow_{\diamondsuit} \sigma' \implies \langle s', \sigma \rangle \Downarrow_{\diamondsuit} \sigma'' \end{array}$$

Here, the first condition says that the trimmed program s' either exhibits the same successful execution as the original program or terminates early with an assumption violation. The second condition says that, if the original program terminates with an assumption violation, then the trimmed program also violates an assumption but can terminate in a different state σ''. In the latter case, we allow the trimmed program to end in a different state σ'' than the original program because the assumption violation could occur earlier in the trimmed program. Intuitively, from a program analysis perspective, we can think of trimming as a program simplification technique that prunes execution paths that are guaranteed not to result in an assertion violation.

Observe that program trimming preserves all terminating executions of program s. In other words, if s terminates under valuation σ, then the trimmed version s' is also guaranteed to terminate. However, program trimming does not give any guarantees about non-terminating executions. Hence, even though this technique is suitable as a pre-processing technique for safety checking, it does not necessarily need to preserve liveness properties. For example, non-terminating executions of s can become terminating in s'.

The definition of program trimming presented above does not impose any *syntactic* restrictions on the trimmed program. For instance, it allows program trimming to add and remove arbitrary statements as long as the resulting program satisfies the properties of Definition 3. However, in practice, it is desirable to make some syntactic restrictions on how trimming can be performed. In this thesis, we perform program trimming by adding assumptions to the original program rather than removing statements. Even though this transformation does not "simplify" the program from a program understanding point of view, it is very useful to subsequent program analyzers because the introduction of `assume` statements prunes program paths in a *semantic* way.

2.3 Static Analysis for Trimming

As mentioned in Section 2, our trimming algorithm consists of two phases, where we infer safety conditions using a lightweight static analysis in the first phase and instrument the program with trimming conditions in the next phase. In this section, we describe the safety condition inference.

2.3.1 Programming Language

In order to precisely describe our trimming algorithm, we first introduce a small, but realistic, call-by-value imperative language with pointers and procedure calls. As shown in Figure 2.2, a program in this language consists of one or more procedure definitions. Statements include sequencing, assignments, heap reads and writes, memory allocation, procedure calls, assertions, assumptions, and conditionals. Since loops can be expressed as tail-recursive procedures, we do not introduce an additional loop construct. Also, observe that we only allow conditionals with non-deterministic predicates, denoted $\star$. However, a conditional of the form $\texttt{if}\ (p)\ \{s_1\}\ \texttt{else}\ \{s_2\}$ can be expressed as follows in this language:

$$\texttt{if}\ (\star)\ \{\texttt{assume}\ p; s_1\}\ \texttt{else}\ \{\texttt{assume}\ \neg p; s_2\}$$

Since the language is quite standard, we do not present its operational semantics in detail. However, as explained in Section 2.2, we assume that the execution of a program terminates as soon as we encounter an assertion or assumption violation (i.e., the predicate evaluates to false). As in Section 2.2, we use the term *failing execution* to indicate a program run with an assertion violation.

2.3.2 Safety Condition Inference

Recall from Section 2, that a *safety condition* at a given program point π is a sufficient condition for any execution starting at π to be error free. More

$$\begin{array}{lll}
\text{Program } \mathcal{P} & ::= & \overline{prc} \\[1em]
\text{Procedure } prc & ::= & \texttt{proc}\ \ prc(\overline{v_{in}}) : v_{out}\ \ \{s\} \\[1em]
\text{Statement } s & ::= & s_1; s_2 \mid v := e \mid v_1 := *v_2 \mid *v := e \\
 & \mid & v := \texttt{malloc}(e) \mid v := \texttt{call}\ \ prc(\bar{v}) \\
 & \mid & \texttt{assert}\ p \mid \texttt{assume}\ p \\
 & \mid & \texttt{if}\ (\star)\ \{s_1\}\ \texttt{else}\ \{s_2\} \\[1em]
\text{Expression } e & ::= & v \mid c \mid e_1 \oplus e_2 \ \ (\oplus \in \{+, -, \times\}) \\[1em]
\text{Predicate } p & ::= & e_1 \oslash e_2 \ \ (\oslash \in \{<, >, =\}) \\
 & \mid & p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p
\end{array}$$

Figure 2.2: Programming language used for formalization. The notation $\overline{s}$ denotes a sequence $s_1, \ldots, s_n$.

precisely, a safety condition for a (terminating) statement s is a formula φ such that $\varphi \Rightarrow wp(s, true)$, where $wp(s, \phi)$ denotes the *weakest precondition* of s with respect to postcondition ϕ [79]. While the most precise safety condition is $wp(s, true)$, our analysis intentionally infers stronger safety conditions so that trimming can be used as a pre-processing technique for safety checkers.

Our safety condition inference engine is formalized using the rules shown in Figure 2.3. Our formalization makes use of an "oracle" Λ for resolving queries about pointer aliasing and procedure side effects. For instance, this oracle can be implemented using a scalable pointer analysis, such as the Data Structure Analysis (DSA) method of Lattner et al. [161]. In the rest of this section, we assume that the oracle for resolving aliasing queries is *flow-insensitive.*

Figure 2.3 includes two types of inference rules, one for statements and one for procedures. Both forms of judgments utilize a *summary environment* Υ that maps each procedure *prc* to its corresponding safety condition (or "summary"). Since our language contains recursive procedures, we would, in general, need to perform a fixed-point computation to obtain sound and precise summaries. However, because our analysis initializes summaries conservatively, the analysis can terminate at any point to produce sound results.

With the exception of rule (10), all rules in Figure 2.3 derive judgments of the form $\Lambda, \Upsilon, \Phi \vdash s : \Phi'$. The meaning of this judgment is that, using environments Λ and Υ, it is provable that $\{\Phi'\}s\{\Phi\}$ is a valid Hoare triple (i.e., $\Phi' \Rightarrow wp(s, \Phi)$ if s terminates). Similarly to the computation of standard weakest preconditions [79], our analysis propagates safety conditions backward but sacrifices precision to improve scalability. Next, we only focus on those rules where our inference engine differs from standard precondition computation.

Heap reads and writes. An innovation underlying our safety condition inference is the handling of the heap. Given a store operation $*v := e$, this statement can modify the value of all expressions $*x$, where x is an alias of v. Hence, a sound way to model the heap is to rewrite $*v := e$ as

$$*v := e; \texttt{if}\ (v = v_1)\ *v_1 := e; \ldots; \texttt{if}\ (v = v_k)\ *v_k := e;$$

where $v_1, \ldots, v_k$ are potential aliases of v. Effectively, this strategy accounts for the "side effects" of statement $*v := e$ to other heap locations by explicitly introducing additional statements. These statements are of the form `if` $(v =$

$$(1)\qquad \frac{\begin{array}{c}\Lambda, \Upsilon, \Phi \vdash s_2 : \Phi_2 \\ \Lambda, \Upsilon, \Phi_2 \vdash s_1 : \Phi_1\end{array}}{\Lambda, \Upsilon, \Phi \vdash s_1; s_2 : \Phi_1}$$

$$(2)\qquad \frac{\Phi' \equiv \Phi[e/v]}{\Lambda, \Upsilon, \Phi \vdash v := e : \Phi'}$$

$$(3)\qquad \frac{\Phi' \equiv \Phi[\mathit{drf}(v_2)/v_1]}{\Lambda, \Upsilon, \Phi \vdash v_1 := *v_2 : \Phi'}$$

$$(4)\qquad \frac{\Phi' \equiv \mathit{store}(\mathit{drf}(v), e, \Lambda, \Phi)}{\Lambda, \Upsilon, \Phi \vdash *v := e : \Phi'}$$

$$(5)\qquad \frac{\Phi' \equiv \forall v.\Phi}{\Lambda, \Upsilon, \Phi \vdash v := \texttt{malloc}(e) : \Phi'}$$

$$(6)\qquad \frac{\begin{array}{c}\overline{\alpha} \equiv \mathit{modLocs}(\mathit{prc}, \Lambda) \\ \Phi_s \equiv \forall v.\ \mathit{havoc}(\overline{\alpha}, \Lambda, \Phi) \\ \Phi' \equiv \Phi_s \wedge \mathit{summary}(\mathit{prc}, \Upsilon, \overline{v_{act}})\end{array}}{\Lambda, \Upsilon, \Phi \vdash v := \texttt{call}\ \ \mathit{prc}(\overline{v_{act}}) : \Phi'}$$

$$(7)\qquad \frac{\Phi' \equiv p \wedge \Phi}{\Lambda, \Upsilon, \Phi \vdash \texttt{assert}\ \ p : \Phi'}$$

$$(8)\qquad \frac{\Phi' \equiv p \Rightarrow \Phi}{\Lambda, \Upsilon, \Phi \vdash \texttt{assume}\ \ p : \Phi'}$$

$$(9)\qquad \frac{\begin{array}{c}\Lambda, \Upsilon, \Phi \vdash s_1 : \Phi_1 \\ \Lambda, \Upsilon, \Phi \vdash s_2 : \Phi_2 \\ \Phi' \equiv \Phi_1 \wedge \Phi_2\end{array}}{\Lambda, \Upsilon, \Phi \vdash \texttt{if}\ \ (\star)\ \ \{s_1\}\ \ \texttt{else}\ \ \{s_2\} : \Phi'}$$

$$(10)\qquad \frac{\begin{array}{c}\Lambda, \Upsilon, \mathit{true} \vdash s : \Phi \\ \Upsilon' \equiv \Upsilon[\mathit{prc} \mapsto \Phi]\end{array}}{\Lambda, \Upsilon \vdash \texttt{proc}\ \ \mathit{prc}(\overline{v_{in}}) : v_{out}\ \ \{s\} : \Upsilon'}$$

Figure 2.3: Inference rules for computing safety conditions.

v_i) $*v_i := e$, i.e., if v and v_i are indeed aliases, then change the value of expression $*v_i$ to e.

While the strategy outlined above is sound, it unfortunately conflicts with our goal of computing safety conditions using lightweight analysis. In particular, since we use a coarse, but scalable alias analysis, most pointers have a large number of possible aliases in practice. Hence, introducing a linear number of conditionals causes a huge blow-up in the size of the safety conditions computed by our technique. To prevent this blow-up, our inference engine computes a safety precondition that is stronger than necessary by using the following conservative *store* operation.

Definition 4. (Memory location) *We represent memory locations using terms that belong to the following grammar:*

$$\text{Memory location } \alpha := v \mid \mathit{drf}(\alpha)$$

Here, v represents any program variable, and drf is an uninterpreted function representing the dereference of a memory location.

To define our conservative store operation, we make use of a function $\mathit{aliases}(v, \Lambda)$ that uses oracle Λ to retrieve all memory locations α that may alias v.

Definition 5. (Store operation) *Let $\mathit{derefs}(\Phi)$ denote all α' for which a* *sub-term $drf(\alpha')$ occurs in formula Φ. Then,*

$$store(drf(\alpha), e, \Lambda, \Phi) := \Phi[e/drf(\alpha)] \wedge \bigwedge_{\alpha_i \in A \setminus \{\alpha\}} \alpha_i \neq \alpha$$
$$\text{where } A \equiv aliases(\alpha, \Lambda) \cap derefs(\Phi)$$

In other words, we compute the precondition for statement $*v := e$ as though the store operation was a regular assignment, but we also "assert" that v is distinct from every memory location α_i that can potentially alias v. To see why this is correct, observe that $\Phi[e/drf(v)]$ gives the weakest precondition of $*v := e$ when v does not have any aliases. If v does have aliases that are relevant to the safety condition, then the conjunct $\bigwedge_{\alpha_i \in A \setminus \{v\}} \alpha_i \neq v$ evaluates to *false*, meaning that we can never guarantee the safety of the program. Thus, $store(drf(v), e, \Lambda, \Phi)$ logically implies $wp(*v := e, \Phi)$.

Example 1. *Consider the following code snippet:*

```
if (⋆) {assume $x = y; a := 3;$}
else   {assume $x \neq y; *y := 3;$}
$*x := a; t := *y;$
assert $t = 3;$
```

Right before the heap write $*x := a$, our analysis infers the safety condition $drf(y) = 3 \wedge x \neq y$. Before the heap write $*y := 3$, the safety condition is $x \neq y$, which causes the condition before the assumption `assume` $x \neq y$ to be true. This means that executions through the `else` branch are verified and may be trimmed because x and y are not aliases for these executions.

Interprocedural analysis. We now turn our attention to the handling of procedure calls. As mentioned earlier, we perform interprocedural analysis in a modular way, computing summaries for each procedure. Specifically, a summary $\Upsilon(f)$ for procedure f is a sufficient condition for any execution of f to be error free.

With this intuition in mind, let us consider rule (6) for analyzing procedure calls of the form $v := \texttt{call}\ \mathit{prc}(\bar{e})$. Suppose that $\bar{\alpha}$ is the set of memory locations modified by the callee *prc* but expressed in terms of the memory locations in the *caller*. Then, similarly to other modular interprocedural analyses [22, 23], we conservatively model the effect of the statement $v := \texttt{call}\ \mathit{prc}(\overline{v_{act}})$ as follows:

$$\begin{array}{l}\texttt{assert}\ \mathit{summary}(\mathit{prc});\\ \texttt{havoc}\ v; \texttt{havoc}\ \bar{\alpha};\end{array}$$

Here, `havoc` α denotes a statement that assigns an unknown value to memory location α. Hence, our treatment of procedure calls asserts that the safety condition for *prc* holds before the call and that the values of all memory locations modified in *prc* are "destroyed".

While our general approach is similar to prior techniques on modular analysis [22, 23], there are some subtleties in our context to which we would like to draw the reader's attention. First, since our procedure summaries (i.e., safety conditions) are not provided by the user, but instead inferred by our algorithm (see rule (10)), we must be conservative about how summaries are "initialized". In particular, because our analysis aims to be lightweight, we do not want to perform an expensive fixed-point computation in the presence of recursive procedures. Therefore, we use the following *summary* function to yield a conservative summary for each procedure.

Definition 6. (Procedure summary) *Let $hasAsrts(f)$ be a predicate that yields true iff procedure f or any of its (transitive) callees contain an assertion. Then,*

$$summary(f, \Upsilon, \bar{v}) = \begin{cases} \Upsilon(f)[\bar{v}/\overline{v_{in}}] & \text{if } f \in dom(\Upsilon) \\ \mathit{false} & \text{if } hasAsrts(f) \\ \mathit{true} & \text{otherwise} \end{cases}$$

In other words, if procedure f is in the domain of Υ (meaning that it has previously been analyzed), we use the safety condition given by $\Upsilon(f)$, substituting formals by the actuals. However, if f has not yet been analyzed, we then use the conservative summary *false* if f or any of its callees have assertions, and *true* otherwise. Observe that, if f is not part of a strongly connected component (SCC) in the call graph, we can always obtain the precise summary for f by analyzing the program bottom-up. However, if f is part of an SCC, we can still soundly analyze the caller by using the conservative summaries given by $summary(f, \Upsilon, \bar{v})$.

The other subtlety about our interprocedural analysis is the particular way in which havocking is performed. Since the callee may modify heap locations accessible in the caller, we define a *havoc* operation that uses the *store* function from earlier to conservatively deal with memory locations.

Definition 7. (Havoc operation)

$$\begin{aligned}
havoc(drf(\alpha), \Lambda, \Phi) &:= \forall v_{new}.\ store(drf(\alpha), v_{new}, \Lambda, \Phi) \\
&\qquad where\ v_{new} \notin freeVars(\Phi) \\
havoc(\overline{\alpha}, \Lambda, \Phi) &:= havoc(tail(\overline{\alpha}), \Lambda, havoc(head(\overline{\alpha}), \Lambda, \Phi))
\end{aligned}$$

Observe that the above definition differs from the standard way this operation is typically defined [22]. In particular, given a scalar variable v, the assignment $v := \star$, and its postcondition ϕ, the standard way to compute a conservative precondition for the assignment is $\forall v.\phi$ (i.e., ϕ must hold for *any* value of v). Note that an alternative way of computing the precondition is $\forall x.\phi[x/v]$, where x is not a free variable in ϕ. In the context of scalars, these two definitions are essentially identical, but the latter view allows us to naturally extend our definition to heap locations by using the previously defined *store* function. Specifically, given a heap location $drf(\alpha)$ modified by the callee, we model the effect of this modification as $\forall v_{new}.\ store(drf(\alpha), v_{new}, \Lambda, \Phi)$.

Theorem 1. *Suppose that* $\Lambda, \Upsilon, \Phi \vdash s : \Phi'$*, and assume that* Λ *provides sound information about aliasing and procedure side effects. Then, under the condition that* s *terminates and that the summaries provided by* Υ *are sound, we have* $\Phi' \Rightarrow wp(s, \Phi)$.[1]

[1]Proofs or proof sketches for all theorems can be found in the appendix.

2.4 Program Instrumentation

In the previous section, we discussed how to infer safety conditions for each program point. Recall that program trimming annotates the code with *trimming conditions*, which are necessary conditions for failure. Here, we describe how we instrument the program with suitable assumptions that preserve safety of the original program.

Intraprocedural instrumentation. First, let us ignore procedure calls and consider instrumenting a single procedure in isolation. Specifically, consider a procedure with body $s_1; \ldots; s_n$ and let:

$$\Lambda, \Upsilon, \mathit{true} \vdash s_i; \ldots; s_n : \Phi$$

We instrument the program with the statement `assume` $\neg\Phi$ right before statement s_i if s_i complies with the instrumentation strategy specified by the user (see Section 2.5). Note that we do not instrument at every single instruction because subsequent safety checkers must also analyze the assumptions, which adds overhead to their analysis.

Theorem 2. *Suppose that our technique adds a statement* `assume` Φ *before* $s_i; \ldots; s_n$*. Then,* Φ *is a necessary condition for* $s_i; \ldots; s_n$ *to have an assertion violation.*

Interprocedural instrumentation. One of the key challenges in performing program instrumentation is how to handle procedure calls. In particular, we cannot simply annotate a procedure f using the safety conditions computed for f. The following example illustrates why such a strategy would be unsound.

Example 2. *Consider procedures* `foo`*,* `bar`*, and* `baz`*:*

```
proc foo(x) {*x := 2; }
proc bar(a) {x := malloc(a); foo(x); assert a < 100; }
proc baz(b) {x := malloc(b); foo(x); assert b > 10; }
```

Here, the safety condition for procedure `foo` *is just true since* `foo` *does not contain assertions or have callees with assertions. However, observe that we cannot simply instrument* `foo` *with* `assume` *false because there are assertions after the call to* `foo` *in* `bar` *and* `baz`*. One possible solution to this challenge is to only instrument the* `main` *method, which would be very ineffective. Another possible strategy might be to propagate safety conditions top-down from callers to callees in a separate pass. However, this latter strategy also has some drawbacks. For instance, in this example, variables* `a` *and* `b` *are not in scope in* `foo`*; hence, there is no meaningful instrumentation we could add to* `foo` *short of* `assume` *true, which is the same as having no instrumentation at all.*

We solve this challenge by performing a program transformation inspired by previous work [115, 157]. The key idea underlying this transformation is to create, for each procedure *prc*, a new procedure *prc*$'$ that can never fail. In particular, we create *prc*$'$ by (a) changing all assertions `assert` ϕ in *prc* to `assume` ϕ, and (b) replacing all calls to f (including recursive ones) with f'. Now, given a call site of *prc*, $v :=$ `call` $prc(\bar{e})$, we replace it with the following conditional:

```
if (⋆) {$v :=$ call $prc'(\bar{e})$;}
else   {$v :=$ call $prc(\bar{e})$; assume $false$;}
```

This transformation is semantics preserving since it is merely a case analysis: Either *prc* succeeds, in which case it is safe to replace the call to *prc* with *prc′*, or it fails, in which case we can call original *prc* but add `assume` *false* afterward since *prc* has failed. The following example illustrates this transformation.

Example 3. *Consider the following procedures:*

```
proc foo($x, y$)  {assert $x > 0$; bar($y$);}
proc bar($z$)     {assert $z > 0$;}
proc main($x, y$) {foo($x, y$);}
```

Our transformation yields the following new program:

```
proc foo'($x, y$)  {assume $x > 0$; bar'($y$);}
proc foo($x, y$)   {
   assert $x > 0$;
   if (⋆) {bar'($y$);}
   else   {bar($y$); assume $false$;}
}
proc bar'($z$)     {assume $z > 0$;}
proc bar($z$)      {assert $z > 0$;}
proc main($x, y$)  {
   if (⋆) {foo'($x, y$);}
   else   {foo($x, y$); assume $false$;}
}
```

The main advantage of this transformation is that it allows us to perform program instrumentation in a modular and conceptually simple way. In particular, we do not need to instrument the "safe" version *prc′* of a procedure *prc* since *prc'* never fails. On the other hand, it is safe to instrument *prc* with the negation of the local safety conditions since every call site of *prc* is followed by the statement `assume` *false* (i.e., execution terminates immediately after the call).

Example 4. *Consider the following procedures* `foo` *and* `bar`*:*

```
proc foo(x)    {assert x > 10;}
proc bar(a, x) {foo(x); assert a < 100;}
```

Our instrumentation yields the following new program:

```
proc foo'(x)  {assume x > 10;}
proc foo(x)    {assume x ≤ 10; assert x > 10;}
proc bar(a, x) {
   assume a ≥ 100 ∨ x ≤ 10;
   if (⋆) {foo'(x);}
   else   {foo(x); assume false;}
   assert a < 100;
}
```

Discussion. The reader may notice that our program transformation introduces additional branches that did not exist in the original program. Since the goal of program trimming is to reduce the number of execution paths while retaining equi-safety, this transformation may seem counter-intuitive. However, because one of the branches is always followed by `assume` *false*, our transformation does not lead to a blow-up in the number of paths and allows us to perform the instrumentation modularly.

2.5 Implementation

We have implemented our program trimming algorithm as a tool called Trimmer, meant as a lightweight pre-processor for program analyzers that check safety. Our implementation is based on the LLVM infrastructure [160] and performs instrumentation at the LLVM bit-code level. Hence, Trimmer can be conveniently integrated into any safety checking tool that is built on top of the LLVM infrastructure and is capable of analyzing `assume` statements.

Recall from Section 2.3 that Trimmer's safety inference engine requires alias and side effect information to soundly analyze heap stores and procedure calls. For this purpose, Trimmer leverages LLVM's DSA pointer analysis [161], a highly-scalable, summary-based, flow-insensitive analysis.

Since Trimmer can be useful to a variety of program analysis tools (including both static and dynamic analyzers), Trimmer can be customized in different ways depending on the assumptions made by subsequent safety checkers. In what follows, we describe the different configurations that Trimmer provides.

Reasoning about integer arithmetic. Trimmer provides the option of treating integral-type expressions either as mathematical (unbounded) or fixed-width integers. Since some safety checkers ignore integer over- and under-flows but others do not, Trimmer supports both encodings.[2] Analyzers treat-

[2]For the fixed-width integer encoding, Trimmer strengthens safety conditions by requiring that there are no integer over- or under-flows. Specifically, Trimmer utilizes arithmetic operations in the LLVM instruction set that return both the result of the operation and a

ing values as mathematical integers can therefore use the configuration of TRIMMER that also makes this same unsound assumption.

Eliminating quantifiers. Recall from Section 2.3 that the safety conditions generated by our inference engine contain universal quantifiers. Hence, when negating the safety conditions, the resulting trimming conditions contain existentially-quantified variables. TRIMMER provides two alternatives for eliminating quantifiers. First, TRIMMER can remove quantifiers using Z3's quantifier elimination (QE) capabilities [74] after simplifying and pre-processing the formula. Second, TRIMMER also allows replacing quantified variables by calls to non-deterministic functions. Since quantified variables at the formula level correspond to program variables with unknown values, this strategy has the same effect as quantifier elimination.

Bounding the instrumentation. After TRIMMER instruments the program with trimming conditions, subsequent safety checkers need to analyze the assumptions. Hence, the number of additional `assume` statements as well as the *size* of the predicates can affect the running time of program analyzers. For this reason, TRIMMER allows users to customize where to add assumptions in the code. For example, sensible strategies include adding instrumentation right before loops and procedure calls, or before every conditional.

In a similar vein, TRIMMER also provides different options for bounding the size of the formulas used in `assume` statements. For example, the user

flag indicating whether an over-flow occurred. Note that TRIMMER does not use bit-vectors for encoding fixed-width integers.

can bound the number of conjuncts in the formula to be at most k, where k is a value chosen by the user. This strategy is sound because Trimmer guarantees that the "simplified" formulas are weaker than the original trimming conditions.

2.6 Experiments

To evaluate the effectiveness of program trimming, we have used Trimmer to pre-process hundreds of programs by instrumenting them with `assume` statements. Since these assumptions are not useful on their own, we evaluate the effect of program trimming in the context of two different LLVM-based program analyzers for safety checking. In particular, we use Crab, an abstract interpreter that supports several abstract domains, and Klee, a widely-used dynamic symbolic execution engine.

We ran our experiments on 439 programs[3], most of which (92%) are taken from the software verification competition (SV-COMP) benchmarks, which have clearly defined outcomes and are handled by numerous tools. Since the errors in many of the buggy programs in this benchmark set are very shallow[4], we also augment these benchmarks with additional buggy programs, either taken from other sources or obtained by injecting deeper bugs into safe SV-COMP benchmarks. The benchmarks taken from SV-COMP span

[3]Available at: `https://mariachris.github.io/FSE2017/benchmarks.zip`

[4]For example, in the existing SV-COMP benchmarks, Klee can find the bug with a very low resource limit for 85% of the buggy programs.

a broad range of categories, including ControlFlow, Loops, Recursive, and ArrayReach, but exclude categories that are not handled by Klee or Crab, e.g., BitVectorsReach, Concurrency.

In what follows, we describe the effects of program trimming on the results of Crab and Klee. We ran all of our experiments on an Intel Xeon CPU E5-2640 v3 @ 2.60GHz machine with 132 GB of memory running the Ubuntu 14.04.1 operating system. We used the latest available version of Crab and the latest version of Klee that was compatible with LLVM 3.6, which Crab requires.

2.6.1 Impact of Program Trimming on Crab

To demonstrate that program trimming increases precision across a range of abstract domains, we compare the performance of Crab (with and without trimming) on three different domains with varying levels of precision:

- **Int** denotes the (non-relational) interval domain [67], which infers invariants of the form $c_1 \leq x \leq c_2$;
- **Zones** is the (relational) zones abstract domain [180], which infers difference constraints of the form $x - y \leq c$;
- **RTZ** is Crab's most precise (native) abstract domain and corresponds to the reduced product of disjunctive intervals (i.e., disjunctions of constraints of the form $c_1 \leq x \leq c_2$) [95] and the zones abstract domains.

Table 2.1: Overview of trimming configurations (incl. total number of added `assume` statements and time for pre-processing all benchmarks in the two right-most columns).

Configuration	MC	QE	L/P	C	A	Time (s)
$\textbf{Trim}_{L+B}$	4	✓	✓	✗	143	5.31
$\textbf{Trim}_{B}$	4	✓	✓	✓	1638	4.97
$\textbf{Trim}_{ND+B}$	4	✗	✓	✓	2801	7.34
$\textbf{Trim}_{L}$	∞	✓	✓	✗	156	6.05
Trim	∞	✓	✓	✓	1735	5.74
$\textbf{Trim}_{ND}$	∞	✗	✓	✓	2852	8.62

As mentioned in Section 2.5, Trimmer can be customized using a variety of different configurations. To understand the precision vs. performance trade-off, we evaluate Crab using the configurations of Trimmer shown in Table 2.1. Here, the column labeled MC indicates the maximum number of conjuncts used in an `assume` statement. The third column labeled QE indicates whether we use quantifier elimination or whether we model quantified variables using calls to non-deterministic functions (recall Section 2.5). Finally, the columns labeled L/P and C denote the instrumentation strategy. In configurations where there is a checkmark under L/P, we add `assume` statements right before loops (L) and before procedure (P) calls. In configurations where there is a checkmark under C, we also add instrumentation before every conditional. The two right-most columns show the total number of added `assume` statements (not trivially *true*) and the pre-processing time for all benchmarks. Since average trimming time is 11–20 milliseconds per benchmark, we see that program trimming is indeed very lightweight.

The results of our evaluation are summarized in Table 2.2. As we can see from this table, all configurations of program trimming improve the precision of Crab, and these improvements range from 23% to 54%. For instance, for the interval domain, the most precise configuration of Trimmer allows the verification of 68 benchmarks instead of only 49 when using Crab without trimming.

Another observation based on Table 2.2 is the precision vs. performance trade-offs between different configurations of Trimmer. Versions of Crab that use Trimmer with QE seem to be faster and more precise than those configurations of Trimmer without QE. In particular, the version of Trimmer with QE performs better because there are fewer variables for the abstract domain to track. We also conjecture that Trimmer using QE is more precise because the abstract domain can introduce imprecision when reasoning about logical connectives. For instance, consider the formula $\exists x.(x = 1 \land x \neq 1)$, which is logically equivalent to *false*, so Trimmer with QE would instrument the code with `assume` *false*. However, if we do not use QE, we would instrument the code as follows:

$$x := \texttt{nondet}(); \texttt{assume}\ \ x = 1 \land x \neq 1;$$

When reasoning about the `assume` statement, an abstract interpreter using the interval domain takes the meet of the intervals $[1, 1]$ and $\top$, which yields $[1, 1]$. Hence, using Trimmer without QE, Crab cannot prove that the subsequent code is unreachable.

Table 2.2: Increased precision of an abstract interpreter due to trimming. Since CRAB treats integers as unbounded, our instrumentation also makes this assumption.

CONFIGURATION	SAFE	TIME (S)
Int	49 (+0%)	129 (+0%)
Trim$_{L+B}$ + **Int**	63 (+29%)	149 (+16%)
Trim$_{B}$ + **Int**	65 (+33%)	173 (+34%)
Trim$_{ND+B}$ + **Int**	61 (+24%)	198 (+53%)
Trim$_{L}$ + **Int**	64 (+31%)	151 (+17%)
Trim + Int	**68 (+39%)**	191 (+48%)
Trim$_{ND}$ + **Int**	62 (+27%)	227 (+76%)
Zones	52 (+0%)	130 (+0%)
Trim$_{L+B}$ + **Zones**	66 (+27%)	148 (+14%)
Trim$_{B}$ + **Zones**	68 (+31%)	195 (+50%)
Trim$_{ND+B}$ + **Zones**	64 (+23%)	222 (+71%)
Trim$_{L}$ + **Zones**	67 (+29%)	150 (+15%)
Trim + Zones	**73 (+40%)**	281 (+116%)
Trim$_{ND}$ + **Zones**	66 (+27%)	320 (+146%)
RTZ	52 (+0%)	215 (+0%)
Trim$_{L+B}$ + **RTZ**	67 (+29%)	231 (+7%)
Trim$_{B}$ + **RTZ**	76 (+46%)	535 (+149%)
Trim$_{ND+B}$ + **RTZ**	66 (+27%)	582 (+171%)
Trim$_{L}$ + **RTZ**	68 (+31%)	237 (+10%)
Trim + RTZ	**80 (+54%)**	1620 (+653%)
Trim$_{ND}$ + **RTZ**	67 (+29%)	3330 (+1449%)

Summary. Table 2.2 shows that trimming significantly improves the precision of an abstract interpreter with reasonable overhead. Our cheapest trimming configuration (**Trim**$_{L+B}$ + **Int**) proves 21% more programs safe than the most expensive configuration of CRAB without trimming (**RTZ**) in *less than 70% of the time.*

Table 2.3: Summary of comparison with KLEE. Since KLEE treats integers in a sound way, we also use the variant of TRIMMER that reasons about integer over- and under-flows.

CONFIGURATION	SAFE	UNSAFE	PATHS	TIMEOUT	MAX-FORKS	TIME (S)
KLEE$_{BFS}$	126 (+0%)	118 (+0%)	9231 (+0%)	73 (+0%)	73 (+0%)	21679
Trim$_{L+B}$ + KLEE$_{BFS}$	146 (+16%)	145 (+23%)	5978 (-35%)	52 (-40%)	46 (-51%)	15558
Trim$_{L}$ + KLEE$_{BFS}$	**146 (+16%)**	**153 (+30%)**	**5678 (-38%)**	**50 (-32%)**	**40 (-45%)**	**15264**
KLEE$_{DFS}$	126 (+0%)	99 (+0%)	10024 (+0%)	91 (+0%)	75 (+0%)	26185
Trim$_{L+B}$ + KLEE$_{DFS}$	146 (+16%)	124 (+25%)	6939 (-31%)	72 (-21%)	48 (-36%)	20797
Trim$_{L}$ + KLEE$_{DFS}$	**146 (+16%)**	**129 (+30%)**	**6695 (-33%)**	**72 (-21%)**	**43 (-43%)**	**21164**
KLEE$_{R}$	126 (+0%)	121 (+0%)	9227 (+0%)	71 (+0%)	72 (+0%)	21077
Trim$_{L+B}$ + KLEE$_{R}$	149 (+18%)	146 (+21%)	5967 (-35%)	49 (-31%)	44 (-39%)	14844
Trim$_{L}$ + KLEE$_{R}$	**149 (+18%)**	**152 (+26%)**	**5699 (-38%)**	**48 (-32%)**	**40 (-44%)**	**14850**

2.6.2 Impact of Program Trimming on KLEE

In our second experiment, we evaluate the impact of program trimming on KLEE, a state-of-the-art dynamic symbolic execution tool. We use a subset[5] of the variants of TRIMMER (see Table 2.1) and evaluate trimming on KLEE with three search strategies: breadth-first search (BFS), depth-first search (DFS), and random search (R).

Since programs usually have infinitely many execution paths, it is necessary to enforce some resource bounds when running KLEE. In particular, we run KLEE with a timeout of 300 seconds and a limit of 64 on the number of forks (i.e., symbolic branches).

The results of our evaluation are presented in Table 2.3. Here, the column labeled SAFE shows the number of programs for which KLEE explores all execution paths without reporting any errors or warnings.[6] Hence, these pro-

[5]In particular, since KLEE's analysis is already path-sensitive we do not consider variants that instrument before conditionals here.

[6]By warning, we mean any internal KLEE warning that designates an incompleteness in

grams can be considered verified. The second column, labeled Unsafe, shows the number of programs reported as buggy by each variant of Klee. In this context, a bug corresponds to an explicit assertion violation in the program. Next, the third column, labeled Paths, shows the number of program paths that Klee explored for each variant. Note that fewer paths is better—this means that Klee needs to explore fewer executions before it finds the bug or proves the absence of an assertion violation. The next two columns measure the number of programs for which each Klee variant reaches a resource limit. In particular, the column labeled Timeout shows the number of programs for which Klee fails to terminate within the 5-minute time limit. Similarly, the column Max-Forks indicates the number of programs for which each Klee variant reaches the limit that we impose on the number of forks. Finally, the last column, labeled Time, shows the total running time of each Klee variant on all benchmarks.

As shown in Table 2.3, program trimming increases the number of programs that can be proved safe by 16–18%. Furthermore, program trimming allows Klee to find up to 30% more bugs within the given resource limit. In addition, Klee with program trimming needs to explore significantly fewer paths (up to 38%) and reaches the resource bound on significantly fewer programs. Finally, observe that the overall running time of Klee decreases by up to 30%.

Klee's execution (e.g., solver timeouts and concretizing symbolic values).

Figure 2.4 compares the number of benchmarks solved by the original version of KLEE (using BFS) with its variants using program trimming. Specifically, the x-axis shows how many benchmarks were solved (i.e., identified as safe or unsafe) by each variant (sorted by running time), and the y-axis shows the corresponding running time per benchmark. For instance, we can see that **Trim**$_L$ + KLEE$_{BFS}$ solves 246 benchmarks within less than one second each, whereas the original version of KLEE only solves 203 benchmarks.

Summary. Overall, the results shown in Table 2.3 and Figure 2.4 demonstrate that program trimming significantly improves the effectiveness and performance of a mature, state-of-the-art symbolic execution tool. In particular, program trimming allows KLEE to find more bugs and prove more programs correct within a given resource limit independently of its search strategy.

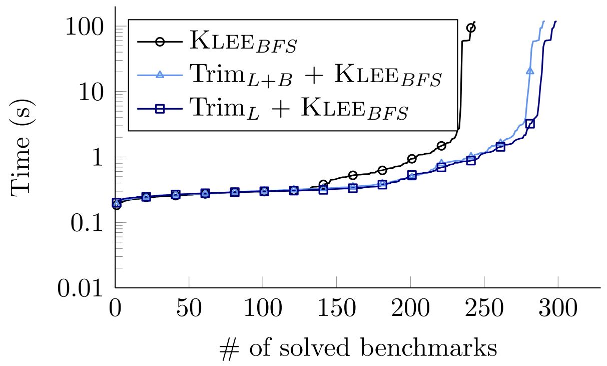


Figure 2.4: Quantile plot of time and solved benchmarks for selected KLEE variants.

Chapter 3

Symbolic Reasoning for Automatic Signal Placement

A common challenge in concurrent programming is to coordinate access to shared resources and achieve correct synchronization between different threads. While there are many different language constructs that can be used to perform synchronization, a widely-established programming pattern is to encapsulate inter-thread coordination using *monitors* [116, 136, 162]. At a high level, a monitor encapsulates all shared state between threads and guarantees mutual exclusion. In addition, monitors perform synchronization between threads by blocking and unblocking them depending on the availability of some shared resource.

Broadly speaking, monitors can be classifed into two categories, depending on the burden they impose on the system vs. the programmer [40]. In particular, *explicit-signal monitors* typically employ *condition variables* to perform synchronization between threads and use an explicit "`signal`" construct to notify other threads when some shared resource becomes available. In contrast, implicit-signal (automatic) monitors provide a `waituntil(P)` construct such that any thread executing this statement blocks until predicate `P` becomes true. In implicit-signal monitors, there is no explicit `signal` construct, and it is the responsibility of the system to notify threads that are currently blocked on a predicate. To give the reader some intuition, Figure 3.1 shows the implementation of an implicit-signal monitor for the well-known readers-writers problem, and Figure 3.2 shows its corresponding implementation as an explicit-signal monitor.

As illustrated by the example from Figures 3.1 and 3.2, programming with implicit monitors is considerably easier because the programmer does not need to reason about when and which threads should be notified. In fact, it is well-known that many concurrency bugs are caused by erroneous signal placement in explicit-signal implementations [111, 146]. However, despite their easier programmability, implicit-signal monitors are not widely-used due to performance considerations. In particular, because the system needs to notify threads that are blocked on a predicate, run-time support for implicit-signal monitors may result in considerable overhead. For example, according to Buhr et al., automatic monitors can be 10-50 times slower than explicit signals [40]. Even though recent work by Hung and Garg proposes a more efficient implementation of automatic monitors [141], explicit-signal monitors still remain the de-facto synchronization mechanism in real-world concurrent programs.

In this thesis, we propose a new solution —based on static analysis— to programming with implicit-signal monitors. Given the implementation of an implicit-signal monitor, our method automatically synthesizes an efficient and semantically equivalent explicit-signal implementation. We believe this approach has two advantages compared to prior run-time techniques: First, because our method does not require additional run-time book-keeping, it has the potential to be as efficient as a performant hand-written explicit-signal implementation. Second, because the code generated by our system can be inspected and further refined by the programmer, it is more transparent compared to automatic-signaling systems that provide run-time instrumentation.

While it is straightforward to generate *any* semantically equivalent explicit-signal implementation of an automatic-signal monitor, a key consideration is the *efficiency* of the synthesized code. In particular, the synthesized code should *not* spuriously wake up threads that are blocked on a predicate that evaluates to false.[1] In practice, this means that the generated code should not notify threads blocked on a predicate P, if P is guaranteed to be false at the time of notification. Furthermore, whenever possible, the generated code should notify a single –rather than *all*– threads blocked on a predicate in order to avoid unnecessary context switches.

In addition to avoiding spurious wake-ups, another important efficiency consideration is to minimize the use of *conditional signals*, which notify other threads only if some condition evaluates to true. Because conditional signals require evaluating the truth value of (potentially complex) predicates at run-time, it is desirable to use unconditional signals whenever possible. In fact,

[1]While a thread that is spuriously woken up will "go back to sleep", this introduces significant overhead due to an unnecessary context-switch.

while some run-time solutions, such as AutoSynch [141] avoid spurious wake-ups altogether, they may still incur significant overhead due to the frequent evaluation of predicates at run-time.

The solution that we adopt in this thesis tries to minimize both the use of spurious wake-ups as well as conditional signals by performing precise static analysis of the monitor code. In particular, our method automatically generates Hoare triples, that, if valid, allow us to establish that a program fragment does *not* need to signal other threads waiting on a predicate. For program fragments where signaling may be necessary, our method generates additional Hoare triples whose validity allows us to minimize the use of conditional signals as well as *broadcast* operations that notify all threads.

In order to successfully discharge the generated Hoare triples, our method uses so-called *monitor invariants*, which are assertions that hold every time a thread enters or leaves the monitor. Our approach automatically infers these monitor invariants by combining abductive reasoning and predicate abstraction, allowing the synthesis of non-trivial invariants that involve disjunctions. Monitor invariants allow us to discharge verification conditions that could not be proven otherwise (e.g., by strengthening the precondition of the generated Hoare triples) and are therefore crucial for generating efficient explicit-signaling code.

We have implemented our proposed ideas in a tool called Expresso and evaluate the efficiency of the code generated by Expresso by comparing it against manually written explicit-signal monitors as well as the state-of-the-art

AutoSynch tool that provides run-time support for implicit-signal monitors. Our evaluation shows that the performance of the code synthesized by Expresso is an average of 1.56x faster than AutoSynch and comparable to that of hand-written code.

In all, we make the following key contributions:

- We propose a novel technique, based on static analysis, for generating efficient explicit-signal implementations of implicit-signal monitors.
- We show how the automatic signal placement problem can be reduced to proving the validity of certain kinds of Hoare triples in concurrent programs.
- We introduce the notion of *monitor invariants* and show how to automatically infer them using abductive reasoning and monomial predicate abstraction.
- We implement the proposed techniques in a tool called Expresso and evaluate it by comparing against AutoSynch, a state-of-the art runtime system for implicit-signal monitors, as well as hand-written code.

3.1 Overview of Technique

In this section, we give a high-level overview of our approach with the aid of the reference Readers-Writers example, shown in Figure 3.1. In particular, we explain the reasoning performed by Expresso to automatically

```
class RWLock {
  unsigned int readers = 0;
  boolean writerIn = false;

  atomic void enterReader() {
    waituntil(!writerIn);
    readers++;
  }
  atomic void exitReader() {
    if(readers > 0) readers--;
  }
  atomic void enterWriter() {
    waituntil(readers == 0 && !writerIn);
    writerIn = true;
  }
  atomic void exitWriter() {
    writerIn = false;
  }  }
```

Figure 3.1: Implicit-signal monitor for readers-writers lock.

generate the code shown in Figure 3.2 by analyzing the implicit-signal monitor of Figure 3.1.

Expresso starts its analysis by inferring a *monitor invariant*, which is an assertion that holds every time a thread enters or exits the monitor. For the code in Figure 3.1, Expresso successfully infers the invariant $readers \geq 0$. Then, Expresso uses this invariant to determine for each conditional critical section in Figure 3.1 (a) if signaling is necessary, (b) whether to signal or broadcast, and (c) whether to do so conditionally or unconditionally.

EnterReader. Consider a reader thread t_r executing the method `enterReader`. To generate explicit signaling code, we need to determine whether t_r needs to notify any writer threads blocked on predicate $P_w = (readers = 0 \land \neg writerIn)$

at line 13. Towards this goal, we ask the following question: "Assuming that a writer thread t_w is blocked on P_w, is it possible that P_w becomes true after t_r executes the code in `enterReader`?". If the answer to this question is "no", we have established that t_r does not need to signal. Thus, to prove that no signals are necessary, Expresso generates and checks the validity of the following Hoare triple:

$$\{\mathit{readers} \geq 0 \land \neg \mathit{writerIn} \land \neg P_w\} \;\; \texttt{readers++} \;\; \{\neg P_w\}$$

Here, the precondition states that (a) the monitor invariant holds when t_r enters the monitor, (b) `!writerIn` must hold if t_r executes `readers++`, and (c) P_w is false, meaning that some writer thread may be blocked at line 13. The post-condition says that P_w continues to stay false after t_r exits the monitor. Since this Hoare triple is indeed valid, Expresso establishes that no signaling is necessary. Observe that dropping the conjunct $\mathit{readers} \geq 0$ from the precondition would result in a Hoare triple that is *not* valid; thus, the monitor invariant is crucial for avoiding the signal operation in this example.

ExitReader. For the `exitReader` method, Expresso needs to determine whether we should signal any reader threads blocked at line 6 or writer threads blocked at line 13. Using similar reasoning as in `enterReader`, it is easy to establish that we do not need to signal reader threads because `readers--` does not affect the truth value of the predicate `writerIn`. Now, to determine the necessity of signaling writer threads, Expresso generates the following Hoare triple:

$$\{\mathit{readers} \geq 0 \land \neg P_w\}\ \texttt{if(readers > 0) readers--}\ \{\neg P_w\}$$

Since this Hoare triple is not valid, signalling is necessary.

Next, EXPRESSO tries to determine whether it suffices to notify a *single* writer thread or we need to notify all writers blocked at line 13. To answer this question, we ask "Is it possible that P_w stays true after some writer thread t_w executes `enterWriter`?". If not, we have proven that it is unnecessary (and wasteful) to wake up multiple threads, since P_w becomes false after the first writer thread executes. Thus, EXPRESSO generates and checks the following Hoare triple:

$$\{\mathit{readers} \geq 0 \land P_w\}\ \texttt{writerIn = true}\ \{\neg P_w\}$$

Since this triple is valid, EXPRESSO has determined that broadcasting is *not* necessary.

Finally, EXPRESSO checks whether it can signal unconditionally, meaning that P_w is guaranteed to hold after the reader thread t_r exits the monitor. Towards this goal, we perform the following check:

$$\{\mathit{readers} \geq 0 \land \neg P_w\}\ \texttt{if(readers > 0) readers--}\ \{P_w\}$$

This Hoare triple is not valid, so EXPRESSO signals conditionally in order to avoid a spurious wake-up.

```
class RWLock {
  unsigned int readers = 0;
  boolean writerIn = false;
  Lock l = new ReentrantLock();
  Condition readers = l.newCondition(),
            writers = l.newCondition();
  void enterReader() {
    l.lock();
    while(writerIn) readers.await();
    readers++;
    l.unlock();
  }
  void exitReader() {
    l.lock();
    if (readers > 0) readers--;
    if (readers == 0) writers.signal();
    l.unlock();
  }
  void enterWriter() {
    l.lock();
    while(readers != 0 || writerIn) writers.await();
    writerIn = true;
    l.unlock();
  }
  void exitWriter() {
    l.lock();
    writerIn = false;
    if (readers == 0) writers.signal();
    readers.signalAll();
    l.unlock();
  } }
```

Figure 3.2: Explicit-signal monitor for readers-writers lock.

EnterWriter. Using similar reasoning as in `enterReader`, EXPRESSO can establish that `enterWriter` does not need to signal any readers because the following Hoare triple is valid:

$$\{readers \geq 0 \land P_w \land writerIn\} \ \texttt{writerIn = true} \ \{writerIn\}$$

ExitWriter. For `exitWriter`, Expresso establishes that it is necessary to notify both reader and writer threads. Using similar reasoning as in `exitReader`, we can prove that broadcasting for all *writer* threads is not necessary, however, we need to perform conditional signaling to avoid spurious wake-ups. For *reader* threads, Expresso determines that broadcasting is necessary since `!writerIn` continues to hold after executing the statement `readers++`. Furthermore, since the Hoare triple

$$\{readers \geq 0 \land writerIn\} \ \texttt{writerIn = false} \ \{\neg writerIn\}$$

is valid, Expresso can establish that `!writerIn` must be true after the writer thread exits. Thus, Expresso instruments the code to signal reader threads unconditionally.

Summary. For this example, the code generated by Expresso is precisely the same one as the human-written explicit-signal implementation shown in Figure 3.2. Observe that Expresso can prove the gratuitousness of broadcasts, and it can also establish that `enterReader` and `enterWriter` do not need to signal. Finally, note that some of the Hoare triples generated by Expresso could not be established without the useful monitor invariant $readers \geq 0$.

3.2 Source and Target Languages

In this section, we present some preliminary concepts related to concurrent programming and describe the source and target languages that can be used to implement implicit- and explicit-signal monitors respectively. The goal of the two languages presented here is to provide a unified theoretical framework suitable for automatic reasoning. In Section 4.5, we discuss how the target language can be instantiated in a concrete monitor implementation.

3.2.1 Preliminaries

In this thesis, we consider a shared-memory concurrency model in which all accesses to shared resources occur inside a *monitor*. In other words, all variables accessed outside the monitor are assumed to be thread-local. We represent threads using integer identifiers drawn from the set $T \subseteq \mathbb{N}$. Because we do not impose any restrictions on the number of threads that can execute monitor code, our approach is applicable to parametrized concurrent programs.

We partition program variables used in the monitor into two disjoint sets, namely L and G, representing thread-local and shared (global) variables respectively. As stated by the definition below, the state σ of a monitor identifies the values of program variables for each thread.

Definition 8. ***(Monitor state)*** *A monitor state, , is a mapping from (thread identifier, monitor variable) pairs to a value. We require monitor states to agree on the values of shared variables for all threads; i.e.,*

$$\forall t_1, t_2 \in T, v \in G.\ \sigma(t_1, v) = \sigma(t_2, v)$$

3.2.2 Source Language

Because our approach transforms an automatic-signal monitor to an explicit-signal one, we first present the source language in which automatic-signal monitors are implemented.

The syntax of our source language is presented in Figure 4.9. Since our implementation targets Java programs, we consider implicit-signal monitors written in a simple object-oriented language with Java-like syntax. In particular, an automatic-signal monitor consists of a set of field declarations and a set of *atomic* methods – i.e., the body of a method m executes without interruption unless the thread blocks on some `waituntil` statement whose corresponding predicate evaluates to false. To simplify presentation, we will assume that *local variables of different methods have unique names.*

The body of each monitor method is a sequence of statements of the form `waituntil`$(p)\{s\}$, where p is a predicate and s is a statement (assignment, store, sequence, loop etc.). Observe that a statement s is a special case of a `waituntil` statement whose corresponding predicate is *true*. We refer to predicate p as the *guard* of the `waituntil` construct and to statement s as its *body* and sometimes write $w = (p, s)$ to denote a `waituntil` statement with guard p and body s. Given a monitor M, we use the notation $CCRs(M)$ to represent the set of all `waituntil` statements used in any method in M.

While `waituntil` statements can only appear as top-level statements in our source language, we note that this design decision does not sacrifice

$$
\begin{array}{lcl}
\text{Monitor } M & ::= & \texttt{monitor } M \ \{ \ (\mathit{fld} \mid m)^* \ \} \\[1ex]
\text{Field } \mathit{fld} & ::= & f : \tau \\[1ex]
\text{Method } m & ::= & \texttt{atomic } m(\vec{v}) \ \{w^*; \texttt{return } v;\} \\[1ex]
\text{WUntil } w & ::= & \texttt{waituntil}(p)\{ \ s \ \} \\[1ex]
\text{Statement } s & ::= & \mathit{skip} \mid s_1; s_2 \mid v := e \mid v.f := e \\
 & \mid & \texttt{if } (p) \ \{s_1\} \texttt{ else } \{s_2\} \\
 & \mid & \texttt{while } (p) \ \{s\}
\end{array}
$$

Figure 3.3: Implicit-signal monitor language. Here, e and p denote expressions and predicates respectively.

expressiveness. For example, consider the code snippet `if (c) waituntil(p)`, which is not supported by our source language. Observe that the check `if(c)` can be moved outside of the monitor if `c` is not on shared data. On the other hand, if `c` does involve shared data, the condition is either checked in a logically racy way [2] or the programmer knows that `c` cannot change while the thread is blocked on `p`. In either case, the program's logic is preserved or enhanced if condition `c` is moved inside the `waituntil` statement.

In the rest of this thesis, we assume the standard semantics of $\texttt{waituntil}(p)\{s\}$ statements where a thread t atomically performs the following actions: It first evaluates the boolean predicate p. If p evaluates to *true*, t also executes s immediately after the evaluation of p. Otherwise, t is blocked until p becomes true.

[2] i.e., condition c could have changed while the thread is waiting on p

$$(1a) \quad \frac{\begin{array}{c} e = (t, w, \mathit{false}) \\ \overline{e} \notin \mathcal{B} \quad (\sigma, t) \not\models \mathit{Guard}(w) \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma, \epsilon, \mathcal{B} \cup \{\overline{e}\}, \mathcal{N})}$$

$$(1b) \quad \frac{\begin{array}{c} e = (t, w, \mathit{false}) \\ \overline{e} \in \mathcal{N} \quad (\sigma, t) \not\models \mathit{Guard}(w) \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma, \epsilon, \mathcal{B}, \mathcal{N} \backslash \{\overline{e}\})}$$

$$(2a) \quad \frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} \notin \mathcal{B} \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}' = \{(t, w) \mid (t, w) \in \mathcal{B}, (\sigma', t) \models \mathit{Guard}(w)\} \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \epsilon, \mathcal{B}, \mathcal{N} \cup \mathcal{N}')}$$

$$(2b) \quad \frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} = \min(\mathcal{N}) \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}' = \{(t, w) \mid (t, w) \in \mathcal{B}, (\sigma', t) \models \mathit{Guard}(w)\} \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \epsilon, \mathcal{B} \backslash \{\overline{e}\}, (\mathcal{N} \cup \mathcal{N}') \backslash \{\overline{e}\})}$$

$$(3) \quad \frac{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \epsilon, \mathcal{B}', \mathcal{N}')}{(\sigma, e :: \tau, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \tau, \mathcal{B}', \mathcal{N}')}$$

Figure 3.4: Transition relation for implicit-signal monitor traces. Given an event $e = (t, w, b)$, $\overline{e}$ denotes (t, w). We assume there is a total order relation $\prec$ between events and min picks the minimum one with respect to $\prec$.

Since the semantics of statements s are standard, we do not present them in detail and use the notation $\langle s, t, \sigma \rangle \Downarrow \sigma'$ to indicate the resulting monitor state σ' when thread t executes statement s under initial state σ. Given a monitor state σ, thread t, and predicate p, we write $(\sigma, t) \models p$ if p evaluates to *true* and $(\sigma, t) \not\models p$ if p evaluates to false.

Monitor traces. To define the semantics of monitors, we first introduce the notion of a *monitor trace*. A monitor trace τ is a sequence of *monitor events* where each event e is a triple (t, w, b) where t is a thread identifier, w is a `waituntil` statement, and b is a boolean indicating whether the guard of w evaluates to *true* or *false*. In particular, the event (t, w, false) indicates that thread t was blocked on the guard of w, whereas the event (t, w, true) indicates that t was able to execute w in its entirety. Given an event $e = (t, w, b)$, we write $\overline{e}$ to denote the pair (t, w).

We say that a monitor trace is *syntactically well-formed* if it (a) respects the relative ordering of statements within a method, (b) obeys the requirement that a thread cannot execute method m' before finishing the execution of method m, and (c) satisfies the invariant that a thread exits the monitor either by blocking on a predicate or by finishing the execution of a method. A more formal definition of syntactic well-formedness is presented in the Appendix A.

Example 5. *Consider the following monitor M, where we elide the "`atomic`" keywords for brevity:*

```
monitor M {
  ...
  m1() {waituntil(x>0) {...}; waituntil{y>0}{...}}
  m2() {waituntil(z>0) {...}; waituntil{w>0}{...}}
}
```

Let us refer to the j'th `waituntil` statement in method i as w_{ij}. The trace $[(1, w_{12}, \text{true}), (1, w_{11}, \text{true})]$ *is not syntactically well-formed since the same thread cannot execute w_{12} before w_{11} (i.e., it violates requirement (a)). Similarly, the trace* $[(1, w_{11}, \text{false}), (1, w_{21}, \text{true})]$ *is also not syntactially well-formed* *since the same thread cannot execute method* `m2` *before finishing the execution of* `m1` *(violates (b)). Finally, the following trace is also not syntactically well-formed:*

$$[(1, w_{11}, \text{false}), (2, w_{21}, \text{true}), (1, w_{11}, \text{true}), (1, w_{12}, \text{true})]$$

In particular, it violates requirement (c) since thread 2 exists the monitor without getting blocked or finishing the execution of `m2`*. On the other hand, the following trace is syntactically well-formed:*

$$\begin{array}{lll} [& (1, w_{11}, \text{false}), (2, w_{21}, \text{true}), (2, w_{22}, \text{false}), & \\ & (1, w_{11}, \text{true}), (1, w_{12}, \text{true}), (2, w_{22}, \text{true}) &] \end{array}$$

In this trace, thread 1 attempts to execute the body of w_{11} but is blocked (i.e., $x > 0$ evaluates to false). Then, thread 2 executes the first ***waituntil*** *statement in method* `m2`*, but gets blocked on the second one. After thread 2 executes w_{21}, $x > 0$ becomes true, and thread 1 is able to finish executing method* `m1`*. Finally thread 2 finishes executing method* `m2`*.*

Semantics. We now define the semantics of implicit-signal monitors in terms of the feasibility of well-formed monitor traces. Given a monitor M and a monitor state σ, we say that a trace τ is *feasible* under σ iff (a) it is syntactically well-formed and (b) $(\sigma, \tau, \emptyset, \emptyset) \longrightarrow^* (\sigma', \epsilon, \_, \_)$ where $\longrightarrow^*$ denotes the reflexive transitive closure of the transition relation $\longrightarrow$ defined in Figure 3.4.

Transition relations for implicit-signal monitors are described in Figure 3.4 using judgments of the form

$$(\sigma, \tau, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \tau', \mathcal{B}', \mathcal{N}')$$

$$(1a) \qquad \frac{\begin{array}{c} e = (t, w, \mathit{false}) \\ \overline{e} \notin \mathcal{B} \quad (\sigma, t) \not\models \mathit{Guard}(w) \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \Longrightarrow (\sigma, \epsilon, \mathcal{B} \cup \{\overline{e}\}, \mathcal{N})}$$

$$(1b) \qquad \frac{\begin{array}{c} e = (t, w, \mathit{false}) \\ \overline{e} \in \mathcal{N} \quad (\sigma, t) \not\models \mathit{Guard}(w) \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \Longrightarrow (\sigma, \epsilon, \mathcal{B}, \mathcal{N} \backslash \{\overline{e}\})}$$

$$(2a) \qquad \frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} \notin \mathcal{B} \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}_1 = \mathit{GetSignals}(w, \sigma', \mathcal{B}) \\ \mathcal{N}_2 = \mathit{GetBroadcasts}(w, \sigma', \mathcal{B}) \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \Longrightarrow (\sigma', \epsilon, \mathcal{B}, \mathcal{N} \cup \mathcal{N}_1 \cup \mathcal{N}_2)}$$

$$(2b) \qquad \frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} = \min(\mathcal{N}) \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}_1 = \mathit{GetSignals}(w, \sigma', \mathcal{B}) \\ \mathcal{N}_2 = \mathit{GetBroadcasts}(w, \sigma', \mathcal{B}) \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \Longrightarrow (\sigma', \epsilon, \mathcal{B} \backslash \{\overline{e}\}, (\mathcal{N} \cup \mathcal{N}_1 \cup \mathcal{N}_2) \backslash \{\overline{e}\})}$$

$$(3) \qquad \frac{(\sigma, e, \mathcal{B}, \mathcal{N}) \Longrightarrow (\sigma', \epsilon, \mathcal{B}', \mathcal{N}')}{(\sigma, e :: \tau, \mathcal{B}, \mathcal{N}) \Longrightarrow (\sigma', \tau, \mathcal{B}', \mathcal{N}')}$$

Figure 3.5: Transition relation for explicit-signal monitor traces. Given an event $e = (t, w, b)$, $\overline{e}$ denotes (t, w).

$$\begin{array}{lcl}
\mathit{Events}(\mathcal{B}, p) & = & \{(t, w) \mid (t, w) \in \mathcal{B} \wedge \mathit{Guard}(w) = p\} \\[2ex]
\mathit{GetSignals}(w, \sigma, \mathcal{B}) & = & \left\{ (t', w') \;\middle|\; \begin{array}{c} (p, c) \in \mathit{Signals}(w) \wedge \\ (t', w') = \min(\mathit{Events}(\mathcal{B}, p)) \wedge \\ (c = \checkmark \vee (\sigma, t') \models p) \end{array} \right\} \\[4ex]
\mathit{GetBroadcasts}(w, \sigma, \mathcal{B}) & = & \left\{ (t', w') \;\middle|\; \begin{array}{c} (p, c) \in \mathit{Broadcasts}(w) \wedge \\ (t', w') \in \mathit{Events}(\mathcal{B}, p) \wedge \\ (c = \checkmark \vee (\sigma, t') \models p) \end{array} \right\}
\end{array}$$

Figure 3.6: Auxiliary functions used in Figure 3.5

where $\mathcal{B}$ and $\mathcal{N}$ describe blocked and notified threads respectively. In particular, $(t, w) \in \mathcal{B}$ indicates that thread t is currently blocked on the predicate of w. In contrast, $(t, w) \in \mathcal{N}$ indicates that thread t should be woken up to recheck the predicate of w. The meaning of the judgment $(\sigma, \tau, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \tau', \mathcal{B}', \mathcal{N}')$ is that executing the first event e in τ under $\sigma, \mathcal{B}$, and $\mathcal{N}$ yields a new state σ' as well as a new set of blocked and notified threads $\mathcal{B}'$ and $\mathcal{N}'$ respectively. We now explain the transition relations from Figure 3.4 in more detail.

According to rules (1a) and (1b), an event e of the form (t, w, false) is only feasible when $(\sigma, t) \not\models \mathit{Guard}(w)$ (i.e., the predicate of w evaluates to false). If $\overline{e} = (t, w)$ was not previously in the blocked thread set $\mathcal{B}$, rule (1a) adds $\overline{e}$ to $\mathcal{B}$. If $\overline{e}$ was already in $\mathcal{B}$, then e is only feasible if $\overline{e}$ was "notified" by the system (i.e., $\overline{e} \in \mathcal{N}$ in rule (1b)).

The next two rules (2a) and (2b) state that an event $e = (t, w, \mathit{true})$

is only feasible when $(\sigma, t) \models \mathit{Guard}(w)$ (i.e., the predicate of w evaluates to true). Both rules execute the body of w to obtain a new monitor state σ'. Now, since the execution of w may cause the predicates of blocked threads to become true, $\mathcal{N}'$ contains all (t, w) pairs that were previously in $\mathcal{B}$ and whose predicates evaluate to true under σ'.

3.2.3 Target Language

Our target language is very similar to the source language from Figure 4.9, except that the body of `waituntil` statements contain explicit signals. In particular, a `waituntil` construct in the target language looks as follows:

$$\texttt{waituntil}(p)\{\ s;\ \texttt{signal}(S_1);\ \texttt{broadcast}(S_2)\ \}$$

Here, S_1 and S_2 are sets of pairs (p, c) where p is a predicate and $c \in \{\textbf{?}, \checkmark\}$. The informal semantics of `signal` and `broadcast` are as follows: If $(p, \checkmark)$ is in S_1, then the system will notify (i.e., wake up) a *single* thread blocked on predicate p. In contrast, if $(p, \checkmark) \in S_2$, then the system notifies *all* threads blocked on p. On the other hand, if $(p, \textbf{?})$ is in S_1 (resp. S_2), then p will be evaluated at run-time, and, if p evaluates to true, then one thread (resp. all threads) blocked on p will be notified. Given a `waituntil` statement in the target language, we write $\mathit{Signals}(w)$ to indicate S_1 and $\mathit{Broadcasts}(w)$ to represent S_2.

We also describe the formal semantics of explicit-signal monitors in terms of monitor traces, where the definitions of *trace*, *event*, and *well-formedness* remain the same as in Section 3.2.2. However, the concept of *feasibility* is defined with respect to a different transition relation $\Longrightarrow$, shown in Figure 3.5. In particular, we say that an explicit-signal monitor trace τ is *feasible* if (a) it is syntactically well-formed, and (b) $(\sigma, \tau, \emptyset, \emptyset) \Longrightarrow^* (\sigma', \epsilon, \_, \_)$ where $\Longrightarrow^*$ denotes the reflexive transitive closure of the transition relation $\Longrightarrow$ from Figure 3.5.

The transition relation $\Longrightarrow$ is defined similarly as $\longrightarrow$ except for events of the form (t, w, true). In contrast to implicit signal monitors which wake up all threads whose predicates have become true, explicit-signal monitors decide which threads to notify based on $\mathit{Signals}(w)$ and $\mathit{Broadcasts}(w)$. In particular, rules (2a) and (2b) use auxiliary functions *GetSignals* and *GetBroadcasts* (defined in Figure 3.6) to decide which threads to add to the notification set $\mathcal{N}$. If $(p, c) \in \mathit{Signals}(w)$, then we notify a *single* event $(t', w') \in \mathcal{B}$ such that the predicate of w' is p. If $c = \mathbf{?}$, we additionally check that p evaluates to true under σ' before adding (t', w') to the notification set. The function *GetBroadcasts* is defined similarly except that it notifies all threads blocked on the specified predicate rather than a single one.

3.2.4 Equivalence

We are now ready to define what it means for an implicit-signal monitor M from the source language and an explicit-signal monitor M' from the target language to be *equivalent*. Towards this goal, we first define a normal form for traces:

Definition 9. (Normalization) *Let τ be an implicit-signal monitor trace. We say that τ is* normalized *with respect to monitor state σ if we can derive $(\sigma, \tau, \emptyset, \emptyset) \longrightarrow^* (\sigma, \epsilon, \_, \_)$ without using rule (1b) from Figure 3.4 in the derivation.*

Since rule (1b) corresponds to a spurious notification, a trace is normalized if threads are woken up only when their predicates evaluate to true. Observe that we can always find a normalized feasible trace for any feasible trace by changing the order in which threads are woken up.[3]

Definition 10. (Equivalence) *Let M, M' be implicit- and explicit-signal monitors respectively. We say that M and M' are semantically equivalent, written $M \sim M'$, iff for all monitor states σ and all well-formed traces τ, the following two conditions are satisfied:*

1. *If $(\sigma, \tau, \emptyset, \emptyset) \Longrightarrow^* (\sigma', \epsilon, \_, \_)$, then it is also the case that $(\sigma, \tau, \emptyset, \emptyset) \longrightarrow^* (\sigma', \epsilon, \_, \_)$.*

2. *If $(\sigma, \tau, \emptyset, \emptyset) \longrightarrow^* (\sigma', \epsilon, \_, \_)$ and τ is normalized with respect to σ, then $(\sigma, \tau, \emptyset, \emptyset) \Longrightarrow^* (\sigma', \epsilon, \_, \_)$.*

Here, the first condition states that any feasible trace of the explicit-signal monitor M' must also be a feasible trace of its implicit version M.

[3]Recall that our notion of feasibility does not require the sets $\mathcal{B}, \mathcal{N}$ to be empty. In particular, a trace τ is feasible under σ if $(\sigma, \tau, \emptyset, \emptyset) \longrightarrow^* (\sigma', \epsilon, \mathcal{B}, \mathcal{N})$ for any $\mathcal{B}$ and $\mathcal{N}$. Therefore, a notification that would have been eliminated by rule (1b) can just be ignored indefinitely, i.e., remain in the $\mathcal{N}$ set without affecting other transitions.

However, in general, we cannot expect the converse of this statement to hold: Since the explicit-signal monitor may be more efficient than its implicit-signal counterpart, we cannot require that all feasible traces of M to be also feasible in the explicit-monitor case. Thus, the second condition states that any normalized *feasible trace of M should also be feasible in M'.*

3.3 Signal Placement Algorithm

In this section, we describe our algorithm for automatically transforming an implicit-signal monitor M in the source language to an explicit-signal monitor M' in the target language. Our algorithm ensures that M and M' are equivalent in the sense of Definition 10 and also tries to minimize the number of spurious wake-ups and conditional signals in M'. We start with a basic version of the algorithm and then describe extensions and improvements later in this section.

3.3.1 Basic Algorithm

Our basic signal placement algorithm is shown in Algorithm 1. The PlaceSignals algorithm takes as input an implicit-signal monitor M as well as a *monitor invariant* I, which is an assertion that holds every time a thread enters and exits the monitor. Since automated inference of monitor invariants is described in the next section, we will assume that an oracle provides them for the time being. In this section, we further assume that guards used in `waituntil` statements do not contain thread-local variables. Given such

Algorithm 1 Signal Placement Algorithm

```
function PlaceSignals($M$, $I$)
    input: $M$, an implicit signal monitor
    input: $I$, a monitor invariant
    output: $M'$, an explicit signal monitor
    $\Sigma \leftarrow [w \mapsto \emptyset \mid w \in CCRs(M)]$
    for $(w, p) \in CCRs(M) \times Guards(M)$ do
        if $\vdash \{I \wedge Guard(w) \wedge \neg p\}\ Body(w)\ \{\neg p\}$ :
            continue;
        if $\vdash \{I \wedge Guard(w) \wedge \neg p\}\ Body(w)\{p\}$ :
            $cond \leftarrow \checkmark$
        else
            $cond \leftarrow \mathbf{?}$
        if $\forall (p, s') \in CCRs(M).\ \vdash \{I \wedge p\}\ s'\{\neg p\}$ :
            $bcast \leftarrow false$
        else
            $bcast \leftarrow true$
        $\Sigma(w) \leftarrow \Sigma(w) \cup \{(p, cond, bcast)\}$
    return Instrument$(M, \Sigma)$
```

an implicit-signal monitor M and its invariant I, PlaceSignals returns an explicit-signal monitor M' such that $M \sim M'$.

The algorithm maintains a mapping from each *conditional critical region (CCR)* (i.e., `waituntil` statement) w in M to a set of notifications that should be performed after executing w and before exiting the monitor. The algorithm represents these notification as triples of the form $(p, cond, bcast)$, where p is a predicate, $cond \in \{\mathbf{?}, \checkmark\}$ indicates whether the notification is conditional or unconditional, and *bcast* is a boolean indicating whether it is necessary to notify all threads blocked on p as opposed to a single one. Once the algorithm computes this mapping Σ, it instruments the original implicit-signal monitor M as shown in Figure 3.7 to obtain an explicit-signal monitor M'.

The key part of the PLACESIGNALS algorithm is the loop in lines 6–17. For each conditional critical region w and predicate p used in the monitor, the algorithm first decides whether w may need to notify threads blocked on predicate p. This decision is made based on the provability of the following Hoare triple:

$$\{I \land \mathit{Guard}(w) \land \neg p\}\ \mathit{Body}(w)\ \{\neg p\}$$

Essentially, this triple says that executing the body of w in a state in which $\neg p$ holds ensures that predicate p continues to remain false. Hence, any thread t blocked on p will remain blocked after executing w, so there is no need to notify t. Observe that the precondition of the Hoare triples also assumes $I \land \mathit{Guard}(w)$ because (a) $\mathit{Guard}(w)$ is a prerequisite for executing the body of w and, (b) by definition of monitor invariant, I must hold before executing the body of any CCR.

Next, lines 9–12 determine whether the notification should be conditional or not. Recall that a conditional notification for predicate p checks whether p evaluates to true before waking up threads blocked on p. While conditional notifications prevent spurious wake-ups, it is desirable to avoid evaluating p at run-time if p is guaranteed to hold after executing w. Thus, line 9 checks the validity of the following Hoare triple:

$$\{I \land \mathit{Guard}(w) \land \neg p\}\ \mathit{Body}(w)\ \{p\}$$

In other words, assuming we execute w in a state where a thread is blocked on p, the execution of $\mathit{Body}(w)$ results in a state where p is true. Thus, there is no need to evaluate p at run time before signaling threads blocked on p.

The last part of Algorithm 1 (lines 13–16) determines whether we should notify *all* threads blocked on predicate p. Suppose there are n threads $T = \{t_1, \ldots, t_n\}$ blocked on p, and suppose that an arbitrary thread t_i gets unblocked. If executing t_i is guaranteed to result in a state where predicate p is false, then it is not necessary to notify any of the remaining threads $T \backslash \{t_i\}$. Thus, the algorithm checks the following Hoare triple for *all* CCRs w' with guard p:

$$\{I \land p\}\ \mathit{Body}(w')\ \{\neg p\}$$

If this Hoare triple holds for all CCRs with guard p, then it is safe to signal rather than broadcast.

Theorem 3. [4] *Let* $\mathit{PlaceSignals}(M, I) = M'$. *If* I *is a correct monitor invariant and guards of CCRs in* M *do not contain thread-local variables, then* $M \sim M'$.

3.3.2 Handling Thread-Local Variables

To simplify presentation, our algorithm from Section 3.3.1 assumes that guards of CCRs in the input monitor do not contain thread-local variables.

[4]The proofs of all theorems are in the appendix.

$$\frac{\begin{array}{c} w = \texttt{waituntil}(p')\{s\} \\ S_1 = \{(p,c) \mid (p,c,\mathit{false}) \in \Sigma(w)\} \\ S_2 = \{(p,c) \mid (p,c,\mathit{true}) \in \Sigma(w)\} \\ s' = \texttt{signal}(S_1); \texttt{broadcast}(S_2) \end{array}}{\Sigma \vdash w \leadsto \texttt{waituntil}(p')\{s; s'\}}$$

$$\frac{\Sigma \vdash w_1 \leadsto w_1' \quad \ldots \quad \Sigma \vdash w_n \leadsto w_n'}{\Sigma \vdash w_1; \ldots; w_n \leadsto w_1'; \ldots; w_n'}$$

Figure 3.7: Performing instrumentation

However, if the input monitor M does not satisfy this assumption, the explicit-signal monitor M' generated by Algorithm 1 may not be equivalent to M. We illustrate the problem using the following example:

Example 6. *Consider the following monitor:*

```
monitor M {
  int y=0;
  m1(int x) {
    waituntil(x < y) {x = y+1;}
  }
  m2() {
    y = y+2;
  }
}
```

Suppose we have threads t_1, t_2, t_3, where t_1, t_2 are blocked in `m1`*, and t_3 is executing* `m2`*, after which the value of* `y` *becomes 2. Further, suppose that the value of the thread-local variable* `x` *is 0 for t_1 and 1 for t_2. Since the predicate $x < y$ has become true for both t_1, t_2 and executing t_1 does not change the value of the predicate in t_2 (and vice versa), t_3 should notify both threads. Thus, the explicit-signal monitor should use* `broadcast` *instead of* `signal`*.*

However, recall that Algorithm 1 determines whether `m2` *should broadcast or signal by checking the validity of* $\{x < y\}\ x = y + 1\ \{x \geq y\}$*. Since this Hoare triple is valid, we would erroneously conclude that it is safe for* `m2` *to notify a single thread instead of all threads.*

As illustrated by this example, Algorithm 1 is unsound when guards contain thread-local variables. To remedy this situation, we need to rename thread-local variables when checking validity. In particular, recall that PLACESIGNALS checks the validity of Hoare triples of the form $\{P_1 \wedge P_2\}\ S\ \{Q\}$ where P_1 is an assumption about the currently running thread, whereas P_2 and Q are assumptions/assertions about some other thread. Since S and P_1 may refer to thread-local variables that are also used in P_2 and Q, we need to rename thread-local variables and check the validity of the following modified Hoare triple:

$$\{P_1 \wedge P_2[V'/V]\}\ S\ \{Q[V'/V]\}$$

where $V = \mathit{Locals}(P_2) \cup \mathit{Locals}(Q)$ and V' denotes a fresh set of variables not used elsewhere in P_1, P_2, S, and Q.[5]

3.3.3 Improvement over the Basic Algorithm

In this section, we consider an improvement over Algorithm 1 that aims to further reduce the number of broadcasts in the synthesized explicit-signal monitor. Recall that Algorithm 1 determines whether a CCR should notify

[5]Another subtlety with local variables is how to signal conditionally. We discuss evaluation of predicates with local variables in Section 4.5.

one vs. all threads blocked on predicate p by checking the validity of the following Hoare triple for all CCRs w with guard p:

$$\{I \land p\} \; \mathit{Body}(w) \; \{\neg p\} \tag{3.1}$$

In some cases, it is possible to further strengthen the pre-condition of this Hoare triple. In particular, suppose that the signaling CCR is w' with body s' and guard p' and suppose that ϕ is guaranteed to hold after executing s'. In general, we cannot assume ϕ in the pre-condition of Equation 3.1 because other threads may have invalidated ϕ before the notified thread has a chance to execute. However, if s commutes with the body of every other CCR in the monitor, then the monitor state after executing s for *any* interleaving is equivalent to one in which we execute s immediately after s'. In this case, we can safely assume that s executes immediately after s' since the resulting states are equivalent. This insight allows us to strengthen the precondition of Equation 3.1 by using the post-condition ϕ of the signaling thread.

To make this discussion more precise, let us define a predicate $\mathit{Comm}(w, M)$ as follows:

$$\begin{array}{l} \mathit{Comm}(w, M) \;\Leftrightarrow \big(\forall w' \in CCRs(M) \backslash \{w\}. \\ \mathit{Body}(w'); \mathit{Body}(w) \equiv \mathit{Body}(w); \mathit{Body}(w')\big) \end{array}$$

Essentially, this predicate is true if the body of w commutes with every other CCR in the monitor. Now, using this definition, we can state a weaker sufficient condition for CCR w to notify one –rather than all– threads blocked on predicate p. In particular, we can change the condition at line 13 of Algorithm 1 to the following weaker one:

$$\begin{array}{c}\forall w' = (p, s') \in CCRs(M).\ \big(\vdash \{I \wedge p\}s'\{\neg p\} \vee \\ (\mathit{Comm}(w', M) \wedge \vdash \{I \wedge \mathit{Guard}(w) \wedge \neg p\}\mathit{Body}(w); s'\{\neg p\})\big)\end{array} \tag{3.2}$$

The first line of Equation 3.2 corresponds to the same check we perform at line 13 in Algorithm 1 to determine whether it is safe to signal rather than broadcast. However, if this condition does not hold, we may still be able to prove that broadcasting is unnecessary as long as s' commutes with every other CCR in the monitor and we can prove that p is falsified after executing $\mathit{Body}(w); s'$.

The correctness of Equation 3.2 follows from the following theorem (and the proof of Theorem 6):

Theorem 4. *Let* $\tau = \tau_0 e$ *be a monitor trace and let* $\tau' = e\tau_0$ *where* $e = (t, w, b)$*. If* $(\sigma, \tau, \mathcal{B}, \mathcal{N}) \longrightarrow^* (\sigma', \epsilon, \mathcal{B}', \mathcal{N}')$ *and* $\mathrm{Comm}(w, M)$*, then we have* $(\sigma, \tau', \mathcal{B}, \mathcal{N}) \longrightarrow^* (\sigma', \epsilon, \mathcal{B}', \mathcal{N}')$*.*

Remark. Our discussion in this section assumes non-preemptive signal semantics [10] where a signaled thread is not guaranteed to consume the signal immediately. However, if we assume *preemptive* signal semantics, we can perform this optimization more liberally by only checking whether predicate p is invalidated by the sequential composition of the segment that produces the signal and $\mathit{Body}(w)$.

Algorithm 2 Monitor Invariant Inference

```
1: function InferMonitorInv($M$, $\Theta$)
2:     input: $M$, an implicit signal monitor
3:     input: $\Theta$, set of Hoare triples of the form $\{P\}\ s\ \{Q\}$
4:     output: $I$, a monitor invariant
5:     $\Phi \leftarrow \emptyset$
6:     for $\{P\}\ s\ \{Q\} \in \Theta$ do
7:         $\Phi \leftarrow \Phi \cup abduce(P, \texttt{wp}(s, Q))$
8:     do
9:         $numPreds \leftarrow |\Phi|$
10:        for $\psi \in \Phi$ do
11:            if $\nvdash \{true\}\ Ctr(M)\ \{\psi\}$ :
12:                $\Phi \leftarrow \Phi \setminus \{\psi\}$
13:                continue;
14:            $I \leftarrow \bigwedge_{\psi_i \in \Phi} \psi_i$
15:            if $\exists w \in CCRs(M). \nvdash \{I \wedge Guard(w)\}\ Body(w)\ \{\psi\}$ :
16:                $\Phi \leftarrow \Phi \setminus \{\psi\}$
17:    while $numPreds \neq |\Phi|$
18:    return $I$
```

3.4 Inference of Monitor Invariants

Our signal placement algorithm from Section 3.3 relies on a monitor invariant I that holds at the entry and exit of every CCR. In this section, we describe our method for automatically inferring useful monitor invariants.

Our inference algorithm is property-directed in that it only infers invariants that are useful for proving the Hoare triples generated by the signal placement algorithm. Specifically, our inference engine uses *abductive reasoning* [81] to automatically infer predicates that are useful for proving a given set of Hoare triples. Given a universe of predicates Φ generated using abduction, it then infers the *strongest* conjunctive monitor invariant over predicates in Φ.

Therefore, our invariant inference engine can be viewed as marrying the power of abductive reasoning with predicate abstraction [98, 156]. The advantage of this approach is two-fold: First, rather than relying on a hard-coded universe of predicate templates, our algorithm infers useful predicates automatically using abduction. Second, because the predicates inferred using abduction can involve disjunctions, the monitor invariants synthesized by our algorithm are not restricted to pure conjunctions.

With this intution in mind, we now explain our InferMonitorInv procedure from Algorithm 2 in more detail. This procedure takes two inputs, namely, an implicit-signal monitor M and a set Θ of Hoare triples of the form $\{P\}\ s\ \{Q\}$. Note that Θ simply corresponds to the set of Hoare triples generated by Algorithm 1, but with I set to *true*. The return value of InferMonitorInv is a formula I representing a valid monitor invariant of M.

Conceptually, the InferMonitorInv procedure operates in two phases. The first phase (lines 5–7) generates a universe Φ of candidate predicates, and the second phase (lines 8–17) performs fixed-point computation to infer the strongest conjunctive monitor invariant I over predicates in Φ.

In the first phase of the algorithm, we iterate over all Hoare triples $\{P\}\ s\ \{Q\}$ in Θ and look for a strengthening ψ of the precondition such that the Hoare triple $\{P \wedge \psi\}\ s\ \{Q\}$ becomes valid. Because the correctness of the Hoare triple $\{P\}\ s\ \{Q\}$ boils down to checking the validity of the formula $P \Rightarrow \texttt{wp}(s, Q)$, we can find a suitable strengthening of P by solving the following abductive reasoning problem:

$$\begin{array}{l}\text{Find } \psi \text{ such that :}\\ (1)\ P \land \psi \models \mathtt{wp}(s, Q) \qquad (2)\ P \land \psi \not\models \mathit{false}\end{array} \tag{3.3}$$

Here condition (1) states that $\{P \land \psi\}\ s\ \{Q\}$ is a valid Hoare triple, and (2) states that the speculated invariant ψ is consistent with precondition P. Since abductive reasoning is a well-studied problem, we use the *abduce* procedure described in prior work [81] to automatically infer candidate strengthenings ψ. Also, note that a call to *abduce* at line 7 may yield multiple predicates $\psi_1, \ldots, \psi_n$, all of which constitute valid solutions for Equation 3.3.

Since the predicates Φ generated using abduction in lines 5–7 are merely *candidate* invariants, the next phase of the algorithm performs a fixed-point computation in which we drop every $\psi \in \Phi$ that is not a monitor invariant. Specifically, for each predicate ψ in Φ, we check whether (a) it holds initially (lines 11–13) and (b) whether it is preserved by each CCR in the monitor (lines 15–16). To determine whether ψ holds initially, we check the validity of the Hoare triple $\{\mathit{true}\}\ \mathit{Ctr}(M)\ \{\psi\}$, where $\mathit{Ctr}(M)$ denotes the constructor of M.[6] If this Hoare triple is not valid, we simply drop ψ from set Φ. Next, to determine whether ψ is preserved by CCR w, we check the validity of the Hoare triple $\{I \land \mathit{Guard}(w)\}\ \mathit{Body}(w)\ \{\psi\}$, where I denotes the conjunction of all predicates in Φ. If this triple is invalid for any CCR in M, we again drop ψ from the set Φ. We then repeate this process until I satisfies both

[6]For simplicity, we assume a single constructor; if there are multiple ones, this triple needs to be checked for all constructors.

the initiation and consecution requirements. It is easy to see that formula I returned by InferMonitorInv constitutes a valid monitor invariant.

3.5 Implementation

We have implemented our proposed method in a tool called Expresso. Our implementation leverages the Soot program analysis infrastructure [207] and invokes the Z3 SMT solver [75] for checking logical validity. In what follows, we discuss some important design choices that are not addressed in previous sections.

Generating Java code. While the target language (IR) presented in Section 3.2.3 is convenient for describing our transformation, it does not yield valid Java code. Our implementation converts programs in this IR to valid Java code in the following manner. First, we associate a condition variable with the guard of every `waituntil` statement. Now, given a `waituntil` statement w with associated guard p, body s, and condition variable c, we then generate the following code[7]:

```
while(!p) {c.await();};  s
```

Furthermore, for each $(p_i, \textbf{?}) \in \mathit{Signals}(w)$, we generate the code `if`(p_i) c_i`.signal()`, and for $(p_i, \checkmark) \in \mathit{Signals}(w)$, we emit c_i`.signal()`. For each $(p_i, \_) \in \mathit{Broadcasts}(w)$, we generate the same code where `signal` is replaced with `signalAll`.

[7]Note that our implementation uses the ReentrantLock class.

Instrumentation for predicates with local variables. To support conditional signaling for predicates with local variables, Expresso augments the monitor code with a data structure that tracks the values of local variables for any thread that is blocked on a predicate p. The code generated by Expresso then uses this data structure to check whether p actually evaluates to true at program points that require conditional signaling for p.

Lazy broadcasts. Expresso provides an option for performing broadcasts lazily. Consider a `waituntil` statement w such that $(p, \_) \in \mathit{Broadcasts}(w)$. Rather than emitting the code `c.signalAll()` after the body of w, "lazy broadcast" notifies a single thread t blocked on p and ensures that t notifies all other threads by adding the instrumentation `if(p) c.signal()` after every waituntil statement with guard p. In our implementation, we enable this option by default to minimize context switches.

Discharging Hoare triples. Expresso discharges any Hoare triple $\{P\}\ s\ \{Q\}$ by computing the weakest precondition of Q with respect to s and perfoming a validity check. Since s can contain pointers, Expresso uses the points-to information provided by Doop [39] to produce a whole-program model of the heap. In particular, given a store statement $v.f = e$, Expresso generates additional statements of the form `if`$(v = x_i)\ x_i.f = e$ where x_i is a potential alias of v.

3.6 Evaluation

We evaluate Expresso by performing experiments that are designed to answer the following research questions:

- How does the code generated by Expresso compare against hand-written explicit-signal code?
- How does our solution compare against run-time systems that provide support for implicit signals?
- How long does Expresso take to generate code?

Benchmarks. The benchmarks used in our evaluation come from two different sources, namely all AutoSynch benchmarks from [141] and monitors collected from popular open-source projects from Github. We collected the Github benchmarks by writing a crawler that identifies potential monitors in Java programs using keywords such as `wait`, `signal`, `notify` etc. We then manually inspected these results in decreasing order of Github ranking (a mix of stars and forks) and identified self-contained modules (i.e., monitors) that encapsulate shared state. This process requires manual effort because we need to isolate the monitor code and insert it in a stress-testing harness.

Performance evaluation methodology. We evaluate performance using the same methodology used for evaluating AutoSynch in [141]. Specifically, we use *saturation tests* [41] in which threads only access the monitor and perform

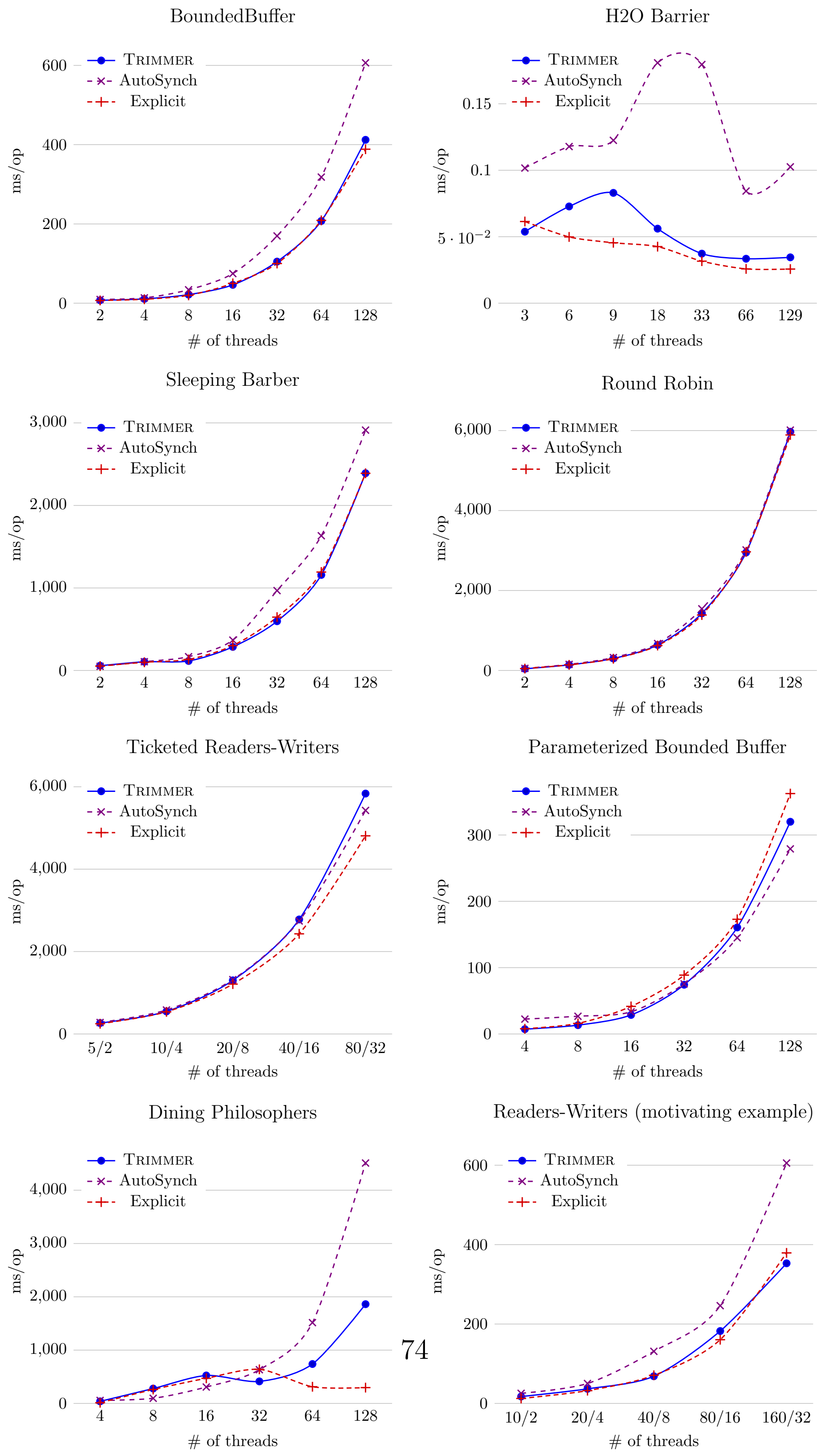


Figure 3.8: Performance over AutoSynch benchmarks and readers-writers example.

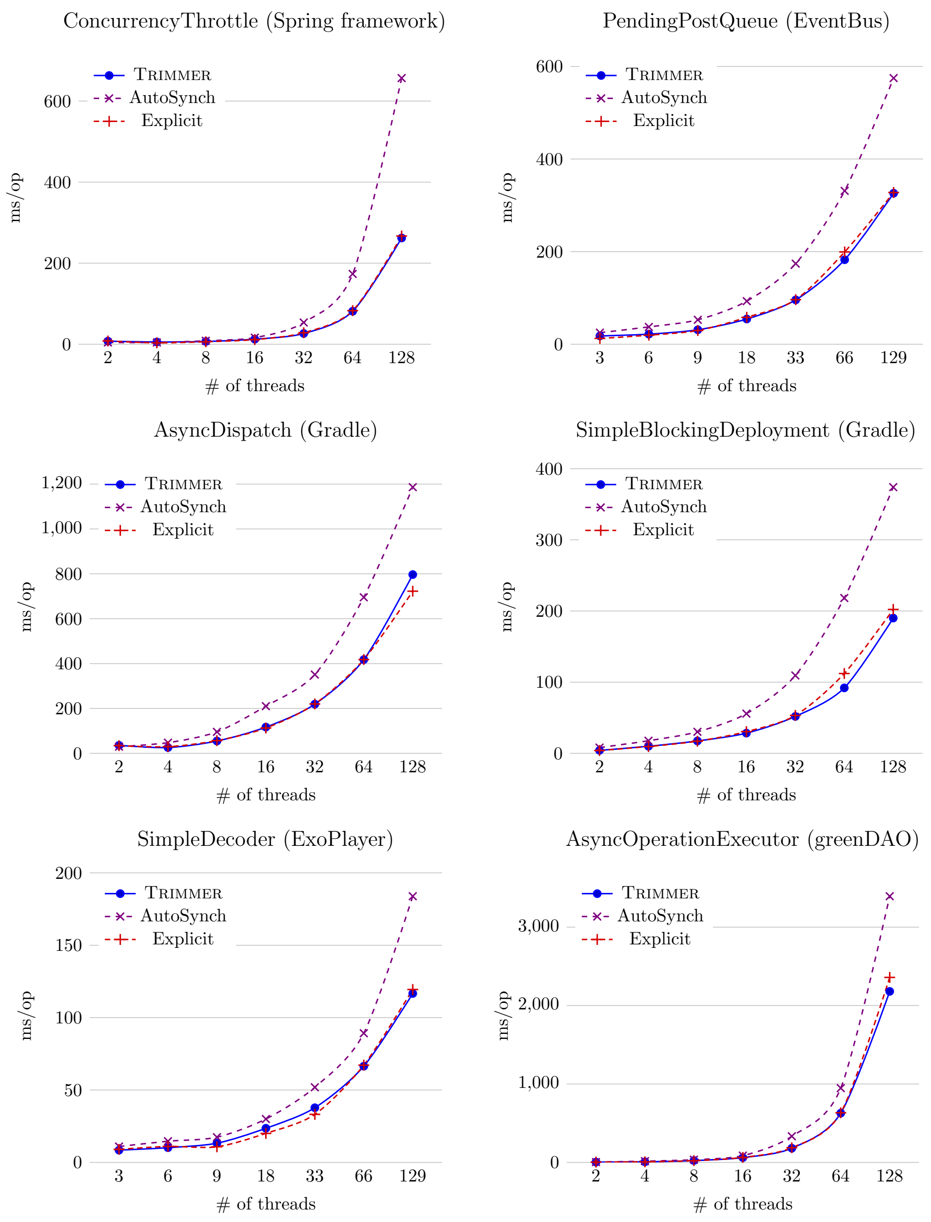


Figure 3.9: Performance over monitor code in popular GitHub projects.

no extra work outside of the monitor. This set-up allows us to stress-test the monitor code and meaningfully compare our solution with run-time solutions and near-optimal hand-written code (from the original AutoSynch benchmarks or from the GitHub project).

We perform measurements using the JMH framework [195], which is a benchmarking tool for rigorous measurement in JVM-based languages. All measurements are conducted on a 16-way (8 core x 2 SMT) Intel Xeon CPU E5-2640 v3 2.60GHz with 132 GB of memory using JDK 1.8.0_101-b13.

Performance results. The results of our performance evaluation are presented in Figures 3.8 and 3.9. Specifically, Figure 3.8 shows the results for the AutoSynch benchmarks, augmented with the readers-writers example of Section 3.1. Figure 3.9 presents results for monitors found in popular GitHub projects. Each graph plots the average time (in milliseconds) per monitor operation (e.g., `enterReader`) against the number of threads.

In virtually all cases, the performance of Expresso-generated code is very close to hand-written explicit-signal code. The only significant differences are in the "H2O Barrier" benchmark under low concurrency and "Dining Philosophers" under high concurrency. In the latter, the explicit signalling code has knowledge of the problem structure itself, so it avoids all wakeups that do not lead to progress.

Comparing to AutoSynch, Expresso outperforms it by 1.56x on average over all benchmarks. Expresso significantly outperforms AutoSynch for about half the benchmarks of Figure 3.8, which are chosen or written by the AutoSynch authors themselves. On a few occasions, AutoSynch slightly outperforms Expresso-generated code. As we discuss in Section 5.2, AutoSynch offers dynamic structures for quick inequality comparisons between shared variables and local values (which are captured as constants while the thread is waiting). This custom optimization can also be added to Expresso but the emphasis of our work has been on statically eliminating unnecessary signalling, rather than minimizing the overhead of dynamic checks.

The monitors found in GitHub projects (Figure 3.9) are more representative of synchronization patterns in-the-wild. Expresso performs very well on these benchmarks, matching hand-optimized code and significantly outperforming AutoSynch: by 1.62x on average, and up to 2.5x on a high-concurrency setting with 128 threads.

Upon closer inspection of these benchmarks, we observe that the symbolic reasoning needed to achieve the results from Figure 3.9 is far from trivial. As a simple example, "ConcurrencyThrottle" from the Spring framework has a waiting condition `threadCount < threadLimit` triggered by the statement `threadCount--` in the monitor exit operation. In order to avoid broadcasts, Expresso needs to infer a monitor invariant that allows it to establish that whenever a thread enters the monitor, the waiting condition has to become true again due to a `threadCount++` operation. Because these increment and decrement operations are distant, symbolic reasoning has to model the semantics of all intervening program statements and establish that the operations commute. This kind of reasoning is necessary for Expresso to achieve the performance results from Figure 3.9 in all benchmarks. Furthermore, the inferred monitor invariants are often intricate—for instance, the “AsyncDispatch” invariant (shown in Appendix) from the Gradle codebase has 22 sub-terms (12 equality/inequality comparisons and arithmetic operations and 10 logical connectives).

To summarize, these results demonstrate the plausibility of a practical and efficient implementation for implicit-signal monitors. In particular, the code generated by Expresso is comparable to hand-written code even for saturation tests that stress-test the monitor. Furthermore, Expresso’s implicit-signal monitor implementation consistently outperforms AutoSynch on monitors extacted from real-world codebases such as Spring framework, Gradle, ExoPlayer, greenDAO, etc.

Analysis time. Table 3.1 shows the time that Expresso takes to synthesize the explicit-signal code from its corresponding implicit-signal version for each benchmark. In most cases, the symbolic reasoning time is in the order of a few seconds. The only exception is the largest benchmark, AsyncDispatch, whose compilation takes 28.3 seconds. This example takes longer to analyze because some of the predicates depend on Java library operations that Expresso also needs to analyze. Overall, these results demonstrate that Expresso is practical and that it generates code whose performance is comparable to hand-written code in virtually all cases.

Benchmark	Time (sec.)
BoundedBuffer	2.5
H2OBarrier	2.3
Sleeping Barber	1.6
Round Robin	1.2
Ticketed Readers-Writers	3.8
Param. Bounded Buffer	2.5
Dining Philosophers	5.4
Readers-Writers	1.5
ConcurrencyThrottle	1.0
PendingPostQueue	0.5
AsyncDispatch	28.3
SimpleBlockingDeployment	0.4
SimpleDecoder	10.7
AsyncOperationExecutor	2.1

Table 3.1: Compilation time for benchmarks.

Generated code. We also assessed the quality of the code generated by Expresso by manually inspecting the synthesized explicit-signal monitors. For most benchmarks obtained from Github projects, we found that the code generated by Expresso is very similar to hand-written code. In the case of the AutoSynch benchmarks, however, we found some examples (e.g., Dining Philosophers) where the Expresso-generated code differs significantly from hand-written code. For these benchmarks, manually-written code leverages dynamic data structures to achieve optimal signaling, whereas Expresso uses the fixed strategy described in Section 4.5 for handling predicates with local variables.

Chapter 4

Verifying Correct Usage of Context-Free API Protocols

Over the last decade, there has been a flurry of research activity on checking the correct usage of APIs [7, 12, 31, 32, 97, 149, 158, 188]. Despite significant advances in this area, almost all existing verification techniques focus on *typestate analysis* [201], 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 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 [73, 145, 174, 178]. However, there has been very little (if any) work on *statically* verifying conformance between a program and a context-free API protocol. In this thesis, 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 [191] or visibly pushdown automata [9], 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 check-

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

ing whether *every* feasible execution of the program belongs to the grammar defined by the protocol (see Figure 4.1).

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 $L_1 \rightarrow f L_2$ 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 corre-

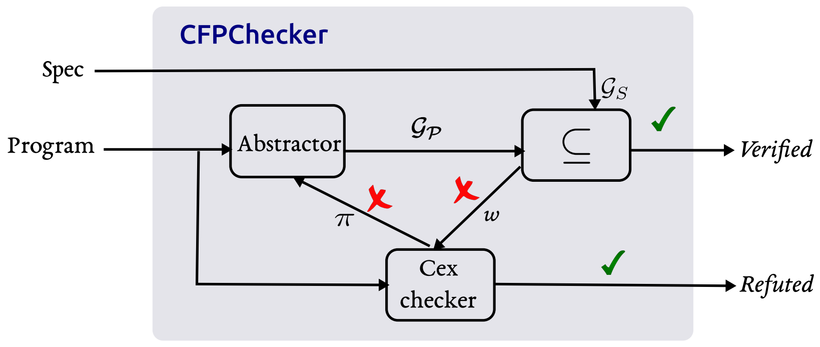


Figure 4.1: Overview of verification approach

sponding 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* [122]. Similar to many other software 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 chapter 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.

4.1 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 $\mathcal{G}_S$:

$$S \;\rightarrow\; \epsilon \;\mid\; \$1.\mathit{lock}()\ S\ \$1.\mathit{unlock}()\ S \tag{4.1}$$

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 4.2(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 4.1 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 4.2(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 4.2(b), which uses a new global variable called `$1` (i.e.,

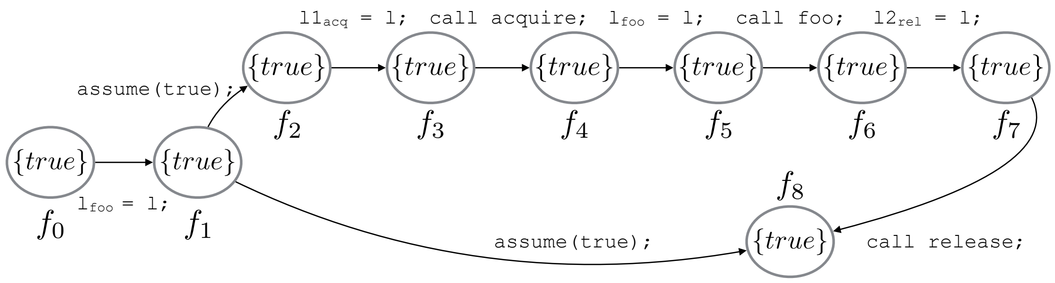
```
void foo(Lock l){
  if (*) {
    acquire(l);
    foo(l);
    release(l);
  }
}

void acquire(Lock l1){
  l1.lock();
}

void release(Lock l2){
  l2.unlock();
}
```

(a) Original Program

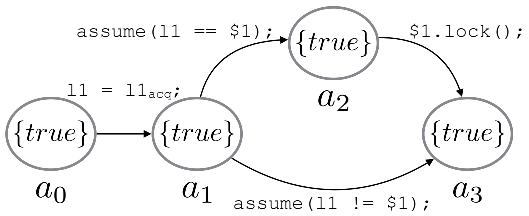
```
static Lock $1 = *;

void foo(Lock l){
  if (*) {
    acquire(l);
    foo(l);
    release(l);
  }
}

void acquire(Lock l1){
  if (l1 == $1)
    $1.lock();
}

void release(Lock l2){
  if (l2 == $1)
    $1.unlock();
}
```

(b) Transformed Program

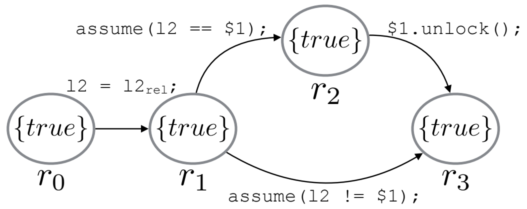
Figure 4.2: Motivating Example



(a) Initial *PCFA* for foo.



(b) Initial *PCFA* for acquire.



(c) Initial *PCFA* for release.

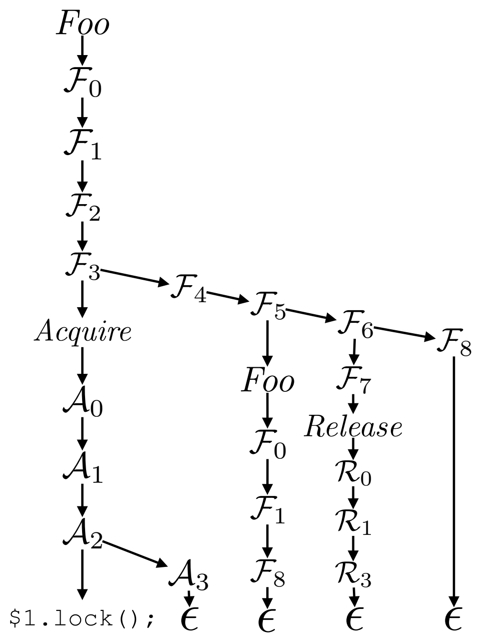
Figure 4.3: Initial *PCFAs* for input program. The PCFAs contain additional formal-to-actual assignments.

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

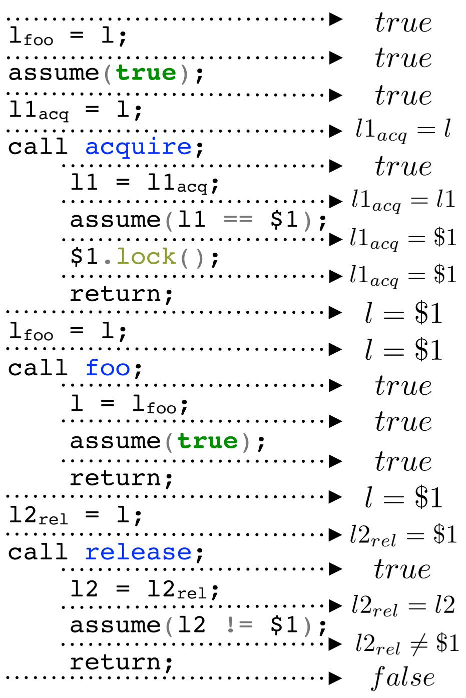
Figure 4.4: Initial context-free grammar.

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



(a) Parse Tree.



```
.......................................► true
lfoo = l;
.......................................► true
assume(true);
.......................................► true
l1acq = l;
.......................................► l1_acq = l
call acquire;
.......................................► true
    l1 = l1acq;
    ...................................► l1_acq = l1
    assume(l1 == $1);
    ...................................► l1_acq = $1
    $1.lock();
    ...................................► l1_acq = $1
    return;
.......................................► l = $1
lfoo = l;
.......................................► l = $1
call foo;
    ...................................► true
    l = lfoo;
    ...................................► true
    assume(true);
    ...................................► true
    return;
.......................................► l = $1
l2rel = l;
.......................................► l2_rel = $1
call release;
    ...................................► true
    l2 = l2rel;
    ...................................► l2_rel = l2
    assume(l2 != $1);
    ...................................► l2_rel ≠ $1
    return;
.......................................► false
```

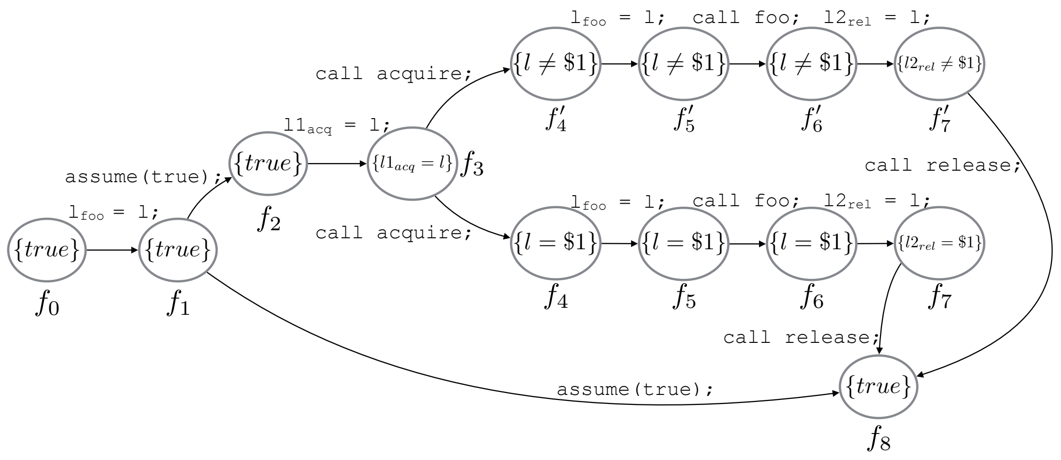
(b) Trace & Interpolants.

Figure 4.5: Tree and Trace for Counterexample `$1.lock()`.

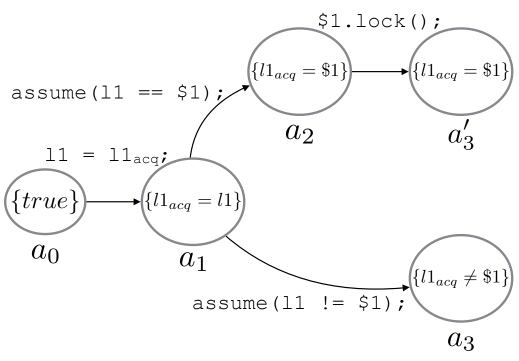
of logical predicates. For example, Figure 4.3 shows the initial PCFAs for Figure 4.2(b): here, nodes correspond to program locations, and edges correspond to transitions. Observe that the PCFAs from Figure 4.3 contain a *single* node for each program location; hence, these PCFAs look like standard *control flow automata (CFA)* used in software model checking [122, 191]. 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 4.4 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: 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 $\mathcal{F}_3 \rightarrow \textit{Acquire}\ \mathcal{F}_4$ models the call from Foo to *Acquire* and $\mathcal{A}_3 \rightarrow \epsilon$ models its corresponding return.

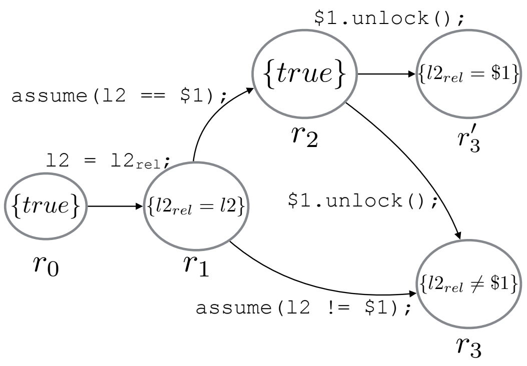
Next, our method checks inclusion between the grammar $\mathcal{G}_\mathcal{P}$ extracted from the program and API protocol $\mathcal{G}_S$. While this problem is, in general, undecidable, we have found the resulting CFG inclusion checking problems to be amenable to automation by modern tools. Going back to our running example, the language of $\mathcal{G}_\mathcal{P}$ from Figure 4.4 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 4.5a that shows how `$1.lock()` can be derived from $\mathcal{G}_\mathcal{P}$. This derivation corresponds precisely to the program path,



(a) Refined *PCFA* for foo.



(b) Refined *PCFA* for acquire.



(c) Refined *PCFA* for release.

Figure 4.6: Refined *PCFAs* for input program.

shown in Figure 4.5b. 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* [122]. In particular, the right-hand side of Figure 4.5b shows the interpolants computed for 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 4.6 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

$$
\begin{aligned}
\mathit{Foo} &\to \mathcal{F}_0 \\
\mathcal{F}_0 &\to \mathcal{F}_1 \\
\mathcal{F}_1 &\to \mathcal{F}_2 \mid \mathcal{F}_8 \\
\mathcal{F}_2 &\to \mathcal{F}_3 \\
\mathcal{F}_3 &\to \mathit{Acquire}_{\phi_1}\ \mathcal{F}_4 \\
&\mid\ \ \mathit{Acquire}_{\phi_2}\ \mathcal{F}'_4 \\
\mathcal{F}_4 &\to \mathcal{F}_5 \\
\mathcal{F}'_4 &\to \mathcal{F}'_5 \\
\mathcal{F}_5 &\to \mathit{Foo}\ \mathcal{F}_6 \\
\mathcal{F}'_5 &\to \mathit{Foo}\ \mathcal{F}'_6 \\
\mathcal{F}_6 &\to \mathcal{F}_7 \\
\mathcal{F}'_6 &\to \mathcal{F}'_7 \\
\mathcal{F}_7 &\to \mathit{Release}_{\phi_3}\ \mathcal{F}_8 \\
\mathcal{F}'_7 &\to \mathit{Rlease}_{\phi_4}\ \mathcal{F}_8 \\
\mathcal{F}_8 &\to \epsilon
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Acquire}_{\phi_1} &\to \mathcal{A}_{0,\phi_1} \\
\mathcal{A}_{0,\phi_1} &\to \mathcal{A}_{1,\phi_1} \\
\mathcal{A}_{1,\phi_1} &\to \mathcal{A}_{2,\phi_1} \\
\mathcal{A}_{2,\phi_1} &\to \texttt{\$1.lock()}\ \mathcal{A}'_{3,\phi_1} \\
\mathcal{A}'_{3,\phi_1} &\to \epsilon
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Acquire}_{\phi_2} &\to \mathcal{A}_{0,\phi_2} \\
\mathcal{A}_{0,\phi_2} &\to \mathcal{A}_{1,\phi_2} \\
\mathcal{A}_{1,\phi_2} &\to \mathcal{A}_{3,\phi_2} \\
\mathcal{A}_{3,\phi_2} &\to \epsilon
\end{aligned}
$$

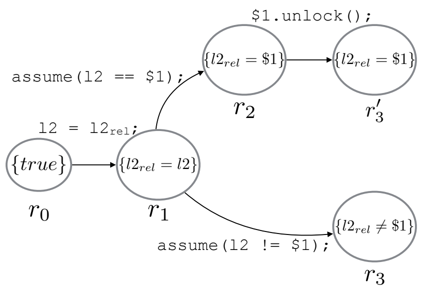
$$
\begin{aligned}
\mathit{Release}_{\phi_3} &\to \mathcal{R}_{0,\phi_3} \\
\mathcal{R}_{0,\phi_3} &\to \mathcal{R}_{1,\phi_3} \\
\mathcal{R}_{1,\phi_3} &\to \mathcal{R}_{2,\phi_3} \\
\mathcal{R}_{2,\phi_3} &\to \texttt{\$1.unlock()} \\
&\qquad \mathcal{R}'_{3,\phi_3} \\
\mathcal{R}'_{3,\phi_3} &\to \epsilon
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Release}_{\phi_4} &\to \mathcal{R}_{0,\phi_4} \\
\mathcal{R}_{0,\phi_4} &\to \mathcal{R}_{1,\phi_4} \\
\mathcal{R}_{1,\phi_4} &\to \mathcal{R}_{2,\phi_4} \\
\mathcal{R}_{1,\phi_4} &\to \mathcal{R}_{3,\phi_4} \\
\mathcal{R}_{2,\phi_4} &\to \texttt{\$1.unlock()} \\
&\qquad \mathcal{R}_{3,\phi_4} \\
\mathcal{R}_{3,\phi_4} &\to \epsilon
\end{aligned}
$$

Figure 4.7: Refined CFG, where $\phi_1 = \{l1_{acq} = \$1\}$, $\phi_2 = \{l1_{acq} \neq \$1\}$, $\phi_3 = \{l2_{rel} = \$1\}$, and $\phi_4 = \{l2_{rel} \neq \$1\}$.

$l1_{acq} = l1$, $l1_{acq} = \$1$ 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 4.7. 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



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

$$Release_{\phi_3} \to \mathcal{R}_{0,\phi_3}$$
$$\mathcal{R}_{0,\phi_3} \to \mathcal{R}_{1,\phi_3}$$
$$\mathcal{R}_{1,\phi_3} \to \mathcal{R}_{2,\phi_3}$$
$$\mathcal{R}_{2,\phi_3} \to \texttt{\$1.unlock()}\ \mathcal{R}'_{3,\phi_3}$$
$$\mathcal{R}'_{3,\phi_3} \to \epsilon$$

$$Release_{\phi_4} \to \mathcal{R}_{0,\phi_4}$$
$$\mathcal{R}_{0,\phi_4} \to \mathcal{R}_{1,\phi_4}$$
$$\mathcal{R}_{1,\phi_4} \to \mathcal{R}_{3,\phi_4}$$
$$\mathcal{R}_{3,\phi_4} \to \epsilon$$

(b) Refined grammar for *Release*.

Figure 4.8: Refined PCFA and grammar for method `release` (second iteration).

for f as there are 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 ($Acquire_{\phi_1}$, $Acquire_{\phi_2}$) representing `acquire` in Figure 4.7, 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 4.6(b). Furthermore, observe that there are two different sets of grammars for $Acquire_{\phi_1}$ and $Acquire_{\phi_2}$, 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 4.7 because node a_2 is not backwards reachable from the exit node labeled with ϕ_2 in Figure 4.6(b).

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 4.6c), 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 4.8.

Class C ::= `class` C { $\mathit{fld}*$ $m*$ }

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 | $\neg p$ | $p_1 \wedge p_2$ | $p_1 \vee p_2$ j $e_1 \oplus e_2$, $\oplus \in \{<, >, =\}$

Figure 4.9: Input Language.

4.2 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.

4.2.1 Input Language

Figure 4.9 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, we assume, without loss of generality, that API calls have no side effects on program state.

For the purposes of this chapter, 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 $\mathit{Traces}(\mathcal{P})$ to denote the (infinite) set of traces that can arise during executions of $\mathcal{P}$.

4.2.2 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 4.1, each

[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.

$\$i_j$ 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 $\Gamma \vdash w : \boldsymbol{\tau}$ to indicate that w is of type $\boldsymbol{\tau}$. We also define a function to extract all wildcard symbols that appear in the grammar:

Definition 11. (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 $\mathcal{G}_S$.*

4.2.3 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 12. (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 13. (Trace-to-Word) *Let τ be a trace and let $\hat{\mathcal{G}} = (T, N, R, S)$ be an (instantiated) API protocol. We define* $\text{TraceToWord}(\tau, \hat{\mathcal{G}})$ *as follows*[3]*:*

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

Example 7. *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 $\hat{\mathcal{G}}$:*

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

Then, we have:

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

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.*

Definition 14. (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:*

$$\forall \tau \in \text{Traces}(\mathcal{P}). \forall \hat{\mathcal{G}} \in \textit{Inst}(\mathcal{G}_S).\ \text{TraceToWord}(\tau, \hat{\mathcal{G}}) \in \mathcal{L}(\hat{\mathcal{G}}) \qquad (4.2)$$

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

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

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

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

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

$$(Class) \qquad \frac{\begin{array}{c} w_i \in \mathcal{W}(\mathcal{G}_S) \qquad \Gamma \vdash w_i : \tau_i \\ f_i' = \texttt{static}\ w_i : \tau_i = \texttt{*} \qquad \Gamma, \mathcal{G}_S \vdash m_i \hookrightarrow m_i' \end{array}}{\Gamma, \mathcal{G}_S \vdash \texttt{cl C \{}\ f_1\ ...\ f_n\ \ m_1\ ...\ m_k\ \texttt{\}} \hookrightarrow \texttt{cl C \{}\ f_1\ ...\ f_n\ f_1'\ ...\ f_j'\ \ m_1'\ ...\ m_k'\ \texttt{\}}}$$

Figure 4.10: 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.

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.

4.3 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 4.10 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 4.10, 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:[4]

Theorem 5. *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. Provided in appendix under supplementary materials. □

[4]The proofs of all theorems are in the appendix.

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.

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.

4.4 Verification Algorithm

Our verification algorithm takes as input a program that has been instrumented as described in Section 4.3. 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.

4.4.1 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 [123, 124, 127]. 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 15. (PCFA) *A predicated control-flow automaton $\mathcal{A}$ is a tuple $\mathcal{A} = (\varSigma, S, \delta)$ where:*

- *$\varSigma$ 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 $\delta \subseteq S \times \varSigma \times S$.*

Notation. Given a state $s = (l, \varphi)$, we use $\mathit{Loc}(s)$ and $\mathit{Pred}(s)$ to denote l and φ respectively. $\mathsf{Trans}(\mathcal{A})$ denotes the transition relation of $\mathcal{A}$. We use the notation $S{\downarrow}l = \{s \in S \mid \mathit{Loc}(s) = l\}$ 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 $\rightsquigarrow$.

4.4.2 Main Algorithm

Figure 4.11 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 chapter, 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

```
procedure $\textsc{Verify}(\mathcal{P}, \mathcal{G}_S)$
    input: $\mathcal{P} : M \to PCFA$, program.
    input: $\mathcal{G}_S$, API-Protocol's context-free grammar.
    output: *Verified* or Counterexample.
    while $true$ do
        $\mathcal{G}_{\mathcal{P}} \leftarrow \textsc{ConstructCFG}(\mathcal{P})$
        if $\exists d.\ d \in InclusionCheck(\mathcal{G}_{\mathcal{P}}, \mathcal{G}_S)$ :
            $(\pi, \rightsquigarrow) \leftarrow derivation2path(d)$
            if $feasible((\pi, \rightsquigarrow))$ : return $\pi$
            else
                $\mathcal{I} \leftarrow Interpolant((\pi, \rightsquigarrow))$
                $\Psi \leftarrow \left\{ l_m \mapsto \left\{ I_j \;\middle|\; \begin{array}{c} I_j \in \mathcal{I}, \sigma_j \in \pi \\ Loc(\sigma_j) = l_m \end{array} \right\} \right\}$
                $\mathcal{P} \leftarrow \textsc{Refine}(\mathcal{P}, \Psi)$
        else return *Verified*
```

Figure 4.11: Verification Algorithm

oracle called *derivation2path* (line 8). Here, we represent an execution path as a *nested trace* [122], which is a tuple ($\pi = \sigma_0 ... \sigma_n, \rightsquigarrow$) 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 ($\pi = \sigma_0 ... \sigma_n, \rightsquigarrow$) and returns an *inductive sequence of nested interpolants* $\mathcal{I} = [I_0, ..., I_{n+1}]$. Following Heizmann et al. [122], we define nested interpolants as a sequence of predicates with the following properties: (1) $I_0 = \mathit{true}$, $I_{n+1} = \mathit{false}$. (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 6. (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. Provided under supplementary materials. □

Theorem 7. (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. Provided under supplementary materials. □

In the following subsections, we describe the Refine (Section 4.4.3) and ConstructCFG (Section 4.4.4) procedures in more detail.

4.4.3 PCFA Refinement

Our PCFA refinement algorithm is summarized in Figure 4.12. Given program $\mathcal{P}$ and mapping Ψ from locations to predicates, the idea is to "clone" any program location $l \in \mathsf{dom}(\Psi)$ based on the predicates $\Psi(l)$. Intuitively,

```
1: procedure REFINE($\mathcal{P}$, $\Psi$)
2:     input: $\mathcal{P} : M \to PCFA$, program.
3:     input: $\Psi : Loc \to \{Pred\}$, new predicates to track.
4:     output: Refined program with respect to $\Psi$
5:     for $(l_m, Preds) \in \Psi$ do
6:         $(\Sigma, S, \delta) \leftarrow \mathcal{P}[m]$
7:         $\Phi \leftarrow \mathsf{CompleteCubes}(Preds)$
8:         $S' \leftarrow \mathsf{CloneStates}(S, l_m, \Phi)$
9:         $\delta' \leftarrow \mathsf{UpdateTransitions}(\delta, l_m, S'{\downarrow}l_m)$
10:        $\mathcal{P}[m] \leftarrow (\Sigma, S', \delta')$
11:    return $\mathcal{P}$
```

Figure 4.12: Program Refinement Algorithm.

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 $l \in \mathsf{dom}(\Psi)$ 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:

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

In other words, $\mathsf{CompleteCubes}(P)$ 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:

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

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 8. *Consider the initial PCFA for method `acquire` from Fig. 4.3b and suppose* $\Psi(a_3) = P = \{l1_{acq} = \$1\}$*. In this case, we have* $\Phi = \mathsf{CompleteCubes}(P) = \{l1_{acq} = \$1, l1_{acq} \neq \$1\}$*. 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 4.6.*

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:

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

where $\mathit{feasible}((s_1, \sigma, s_2))$ is defined as $SAT(sp(\sigma, \mathit{Pred}(s_1)) \wedge \mathit{Pred}(s_2))$. 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, $\mathit{Pred}(s)$, and the semantics of statement σ. Outgoing edges from location l are also updated analogously.

Example 9. *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. 4.6b) because feasible returns false for these edges.*

4.4.4 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 re-

```
procedure ConstructCFG($\mathcal{P}$)
    input: $\mathcal{P} : M \to PCFA$, program.
    output: $\mathcal{G}_{\mathcal{P}}$, context-free grammar that abstracts $\mathcal{P}$.
    $(T, N, R) \leftarrow (\emptyset, \emptyset, \emptyset)$
    $\Theta \leftarrow \{(s, m) \mid s \in \mathit{Exit}(\mathcal{P}[m])\}$
    for $(s_i, m) \in \Theta$ do
        $(T_i, N_i, R_i, S_i) \leftarrow$ GenGrammar($\mathcal{P}[m], s_i, \Theta$)
        $T \leftarrow T \cup T_i,\ N \leftarrow N \cup N_i,\ R \leftarrow R \cup R_i$
        if IsMain($m$) : $S \leftarrow S_i$
    return $(T, N, R, S)$
```

Figure 4.13: Context-Free Grammar Construction

sulting 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 4.13. 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 4.14 summarizes the GenGrammar procedure using inference

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

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

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

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

$$\frac{\varphi = \mathit{Pred}(c)}{\mathcal{A}, c, \Theta \vdash \mathcal{C}_\varphi \to \epsilon \in R_c} \tag{5}$$

Figure 4.14: 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.

rules of the following shape:

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

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 4.14 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 $\mathcal{S}_\varphi \rightarrow \mathcal{S}'_\varphi$ 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 10. *The production $\mathcal{A}_{1,\phi_1} \rightarrow \mathcal{A}_{2,\phi_1}$ in Fig. 4.7 is generated using the* Stmt *rule based on the PCFA from Fig 4.6.*

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 11. *Consider the production $\mathcal{A}_{2,\phi_1} \to$ `$1.lock()` $\mathcal{A}'_{3,\phi_1}$ from Figure 4.7. This production is generated due to the PCFA transition* $(a_2,$ `$1.lock()`$, a'_3)$ *from Figure 4.6.*

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:

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

Example 12. *Consider the PCFAs from Figure 4.6. Here, the production $\mathcal{F}_3 \to \mathit{Acquire}_{\{l1_{acq}=\$1\}}\mathcal{F}_4$ 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 $\mathcal{F}_3 \to \mathit{Acquire}_{\{l1_{acq}=\$1\}}\mathcal{F}'_4$ because $l1_{acq} = l \land l \neq \$1 \land l1_{acq} = \$1$ is unsatisfiable.*

Entry and exit. The last two rules in Figure 4.14 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 13. *For the PCFA from Figure 4.6b, 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*.*

4.5 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 [207] and uses the technique of Madhavan et al. [173] to perform grammar inclusion checks. Our implementation also makes use of SMTInterpol [57] to obtain nested interpolants and leverages Z3 [75] 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 4.4, 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 [199, 211].

From words to execution paths. As mentioned in Section 4.4, we assume that the *InclusionCheck* method returns a derivation $d \in \mathcal{G}_{\mathcal{P}}$ for a word $w \in \mathcal{L}(\mathcal{G}_{\mathcal{P}}) \backslash \mathcal{L}(\mathcal{G}_S)$. 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) [56] for which there is a polynomial algorithm for obtaining a derivation [138], 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 [99]. 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. [17] 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 [167]. 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.).

4.6 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.

4.6.1 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

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()` $Vals \rightarrow \epsilon \mid S\ Vals$

Table 4.1: Java API Protocol Specifications

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 4.1 presents the context-free protocols that clients of these APIs must adhere to. As used as a running example throughout the chapter, `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 $n \geq 1 \rightarrow m \geq 1$. 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

Benchmark Info				CFPChecker Statistics				
	Benchmark	#Classes	LOC	Out.	Total Time	Incl. Check	Preds / BB (Avg/Max)	#Preds
Wifi	ExoPl (↯)	898	89,835	Cex	63.8	0.3	2/6	40
	ExoPl	898	89,839	Safe	35.2	0.4	1/4	44
WakeLock	ExoPl (↯)	898	89,835	Cex	66.6	0.3	2/6	40
	ExoPl	898	89,839	Safe	47.0	0.5	2/6	39
	ConBot (↯)	324	32,506	Cex	392.0	9.3	3/9	107
	ConBot	324	32,504	Safe	2336.5	18.3	3/12	133
ReLock	Hystrix	411	11,457	Safe	20.7	0.3	1/3	21
	Guice	599	16,799	Safe	221.5	2.4	3/9	92
	Bitcoinj	581	53,648	Safe	3175.3	28.0	1/5	115
Canvas	Glide	558	24,959	Safe	562.6	544.7	1/3	19
	RxTool	591	12,666	Safe	56.4	43.8	1/1	3
	Litho	460	33,461	Safe	14.4	0.5	1/3	11
Json	Hadoop	908	51,238	Safe	140.2	64.5	1/4	65
	Hystrix-1	437	12,079	Safe	64.4	2.7	2/4	62
	Hystrix-2	437	12,082	Safe	24.2	0.6	1/4	55

Table 4.2: 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. "Preds / BB": Average and max predicates tracked per basic block, "# Preds": total number of predicates tracked. The number of classes and LOC are prior to slicing. The effects of slicing widely varies across benchmarks, in some cases it eliminates a significant portion of the application leaving no more than 10 classes, whereas in some cases the resulting slice contains more than 100 classes.

		JayHorn			JPF-BugFinder			JPF-Verifier	
	Bench.	Out.	Correct Output?	Time	Out.	Correct Output?	Time	Out.	Correct Output?
Wifi	ExoPl (↯)	UN	✗	1212.4	Safe	✗	0.6	TO	✗
	ExoPl	UN	✗	1119.0	-	-	-	TO	✗
WakeLock	ExoPl (↯)	UN	✗	1105.3	Safe	✗	0.5	TO	✗
	ExoPl	UN	✗	1162.8	-	-	-	TO	✗
	ConBot (↯)	UN	✗	333.0	Safe	✗	0.7	TO	✗
	ConBot	UN	✗	164.0	-	-	-	TO	✗
ReLock	Hystrix	UN	✗	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	✗

Table 4.3: 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, UN: 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.

is an error to call `restore()` more times than `save()` was called."; thus, its specification is of the form $\texttt{save}^n\ \texttt{restore}^m$ 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).

4.6.2 Results for CFPChecker

Table 4.2 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 ↯ symbol) by manually repairing the original bug. We now summarize the key take-away lessons from this evaluation.

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 4.2, 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 [90]; 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 4.15), 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.

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.

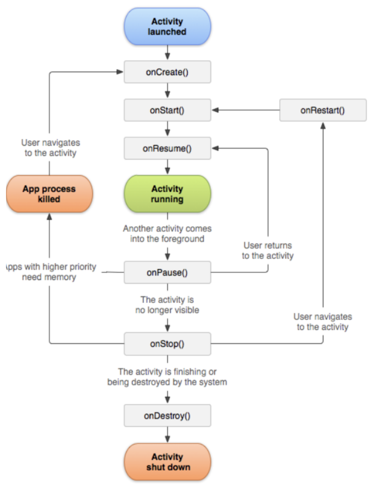

Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
pps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 4.15: Android lifecycle callbacks

4.6.3 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 [151] and JavaPathFinder [11] 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 [52, 145, 178]. 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 4.5 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 backends, we further manually simplified our benchmarks from Section 4.6.1 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 [11]. 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 4.3. 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.

Chapter 5

Related Work

We now discuss prior work that is related to all the techniques presented in this thesis. Section 5.1 discusses work related to Programming Trimming, Section 5.2 discusses related work for our Automatic Signal-Placement technique, and Section 5.3 presents prior work related to our technique for verifying correct usage of Context-Free API protocols.

5.1 Related Work for Program Trimming

Program slicing. One of the most well-known program simplification techniques is *program slicing*, which removes program statements that are not relevant to some criterion of interest (e.g., value of a variable at some program point) [2, 33, 204, 210]. A program slice can be computed either statically or dynamically and includes both forward and backward variants. Program trimming differs from traditional program slicing in two ways: first, trimming focuses on removing execution paths as opposed to statements; second, it is meant as a pre-processing technique for safety checkers rather than a transformation to aid program understanding. In particular, a typical slicing tool may not produce compilable and runnable code that could be consumed by subsequent safety checkers.

More semantic variants of program slicing have also been considered in later work [18, 46, 66, 96, 117, 144]. For instance, Jhala and Majumdar propose *path slicing* to improve the scalability of software model checkers [144]. In particular, path slicing eliminates all operations that are irrelevant toward the reachability of the target location in a given program path. Unlike program trimming, path slicing is not used as a pre-processing step and works on a single program path that corresponds to a counterexample trace.

Prior work has also considered how to slice the program with respect to a predicate [46, 66, 96, 117]. Such techniques can be useful for program understanding, for example, when the user only wants to see statements that affect a given condition (e.g., the predicate of a conditional). In contrast, program trimming is not meant as a program understanding technique and removes program paths that are irrelevant for a given safety property. Furthermore, the trimmed program is not meant for human consumption, as it semantically prunes program paths through the insertion of `assume` statements.

Slicing has been used before invoking a program analyzer [51, 55, 91, 120, 142, 143, 179]. A key difference with these approaches is that the result of trimming is valid code, which compiles and runs, instead of an abstract representation, such as a control flow graph or model.

Pre-processing for program analyzers. In the same spirit as this thesis, prior work has also used program transformations to improve the precision or scalability of program analyzers [58, 60, 61, 115, 157, 194, 213]. For instance, a transformation for faster goal-directed search [157] moves all assertions to a single main procedure with the goal of speeding up analysis. Another program transformation called *loop splitting* aims to improve the precision of program analyzers by turning *multi-phase* loops into a sequence of *single-phase* loops [194]. However, neither of these techniques instrument the program with assumptions to guide safety checking tools.

Recent techniques rely on the verification results of a full-fledged analyzer, such as an abstract interpreter or a model checker, to guide automatic test case generation tools [58, 60, 71, 72] or other static analyzers [29, 59, 61, 213], some even using slicing as an intermediate step [71]. In contrast, program trimming is more lightweight by not relying on previous analyzers and, thus, can be used as a pre-processing step for any safety checker.

Precondition inference. The use of precondition inference dates back to the dawn of program verification [79]. Most verification techniques infer a sufficient condition for program safety and prove the correctness of the program by showing the validity of this condition [22, 24, 49, 79, 100, 132, 133, 135, 181]. In this work, we do not aim to infer the weakest possible safety precondition; instead, we use lightweight, modular static analysis to infer *a* sufficient condition for safety. Furthermore, we use safety conditions to prune program paths rather than to verify the program.

Program trimming hinges on the observation that the negation of a sufficient condition for property P yields a necessary condition for the negation of P. Prior program analysis techniques also exploit the same observation [83–85, 220]. For instance, this duality has been used to perform modular path-sensitive analysis [83] and strong updates on elements of unbounded data structures [84, 85].

While most program analysis techniques focus on the inference of sufficient preconditions to guarantee safety, some techniques also infer *necessary preconditions* [68, 69, 168, 169, 184]. For example, Verification Modulo Versions (VMV) infers both necessary and sufficient conditions and utilizes previous versions of the program to reduce the number of warnings reported by verifiers [169]. Similarly, necessary conditions are inferred to repair the program in such a way that the repair does not remove any "good" traces [168]. Finally, the techniques described by Cousot et al. infer necessary preconditions, which are used to improve the effectiveness of the Code Contracts abstract interpreter [68, 69, 95].

Abductive reasoning. There has been significant work on program analysis using abductive reasoning, which looks for a *sufficient* condition that implies a desired goal [4, 45, 82, 88, 89, 164, 221]. Our analysis for computing safety conditions can be viewed as a form of abductive reasoning in that we generate sufficient conditions that are stronger than necessary for ensuring safety. However, we perform this kind of reasoning in a very lightweight way without calling an SMT solver or invoking a logical decision procedure.

Modular interprocedural analysis. The safety condition inference we have proposed in this thesis is modular in the sense that it analyzes each procedure independently of its callers. There are many previous techniques for performing modular (summary-based) analysis [3, 45, 83, 187, 216]. Our technique differs from these approaches in several ways: First, our procedure summaries only contain safety preconditions, but not post-conditions, as we handle procedure side effects in a very conservative way. Second, we do not perform fixed-point computations and achieve soundness by initializing summaries to *false*. Finally, we use summary-based analysis for program transformation rather than verification.

Property-directed program analysis. There is a significant body of work that aims to make program analyzers property directed. Many of these techniques, such as BLAST [28, 125, 128], SLAM [16, 20, 21], and YOGI [108, 185] rely on counterexample-guided abstraction refinement (CEGAR) [65] to iteratively refine an analysis based on counterexample traces. Another example of a property-directed analysis is the IC3/PDR algorithm [38, 137], which iteratively performs forward and backward analysis for bounded program executions to decide reachability queries. Although abstract interpretation is traditionally not property directed, there is recent work [192] on adapting and rephrasing IC3/PDR in the framework of abstract interpretation. In contrast, we propose a general pre-processing technique to make any eager program analysis property directed.

Path-exploration strategies. Most symbolic execution and testing techniques utilize different strategies to explore the possible execution paths of a program. For example, there are strategies that prioritize "deeper paths" (in depth-first search), “less-traveled paths” [165], “number of new instructions covered” (in breadth-first search), “distance from a target line” [172], or “paths specified by the programmer” [193]. In the context of symbolic execution, program trimming can be viewed as a search strategy that prunes safe paths and steers exploration toward paths that are more likely to contain bugs. However, as shown in our experiments, our technique is beneficial independently of a particular search strategy.

5.2 Related Work for Automatic Signaling

Our implicit-signal monitors have several relatives in the literature. We discuss representative past work, in loose thematic groupings of decreasing affinity with our work.

Language designs and run-time support for concurrency. There is a very rich literature on language support for concurrency, dating back to the early 1960s [10]. Dijkstra originally proposed the concept of *semaphores* to provide a friendlier and more efficient programming abstraction than the busy-wait design [80]. In later work, Hoare proposed the concept of *conditional critical regions (CCR)* [134], which overcome some of the difficulties associated with semaphores by providing a more structured notation for specifying synchronization. In particular, every shared variable in a CCR must belong to a resource, and variables in a resource can only be accessed within so-called *region* statements of the form `region r when B` $\rightarrow$ `S`, where B is a guard and S

is a statement. Concurrently to the introduction of CCRs, Dijkstra proposed the notion of *monitors*, which provide more structure than conditional critical regions and can be implemented as efficiently as semaphores [78]. To this day, monitors remain a popular concurrent programming paradigm, and the "monitor pattern" is widely used in many programming languages, including Java and C++.

Even though early proposals for monitors advocate an implicit signaling mechanism [10, 13], most modern monitor implementations use explicit signaling due to performance considerations. More recent work in this area aim to popularize implicit signal monitors by providing a more efficient implementation [41, 141]. For example, the recent AutoSynch work [141] attempts to improve the efficiency of automatic signaling through a combination of efficient dynamic indexing and simple static analysis. AutoSynch offers sophisticated handling of local state in a thread. If threads wait on conditions based on standard equality/inequality patterns over local variables, the system dynamically snapshots the values of local variables and treats them as run-time constants—the variable values cannot have changed while the thread is waiting. As a result, the predicates have a standard structure of comparisons with constants. The efficient notification algorithm then works much like database indexing: it computes, given which shared values changed, what waiting predicates could possibly have been affected. Our approach is distinguished by its use of reasoning techniques for statically inferring when a condition must,may-not, or may have become true. In order to do so, our technique needs to take into consideration several issues that are not addressed by previous work, e.g., handling of memory aliases, inter-procedural reasoning, etc. As shown in the evaluation, our method significantly outperforms the AutoSynch solution on many benchmarks and is comparable to hand-written code for most examples.

Transactional memoory. CCRs have been recently reified in several language designs, such as that of Harris and Frasier [118]. Such designs can be viewed as special cases of *transactional memory (TM)* [130], which has attracted enormous attention in the literature, as an alternative of lock-based synchronization [159].

Our implicit-signal monitors are both *less* and *more* ambitious than transactional memory techniques, in different respects. Monitors do not attempt to automate-away lock-based synchronization: we explicitly require the programmer to declare different resources together with their synchronization policy. Concretely, each monitor can be thought of as a single lock, whereas transactional memory techniques do away with distinctions between locks in favor of a single `atomic` construct. The same essential feature is kept in Harris and Frasier's conditional critical regions [118]: although `atomic` statements can have a condition associated with them, their code blocks are guaranteed to execute with atomic semantics, relative to any other `atomic` block, under whichever condition. Therefore, implicit-signal monitors are inherently lower-level than TM approaches, requiring more care on behalf of the programmer, but also imposing no overhead for maintaining atomic semantics. At the same time, our implicit-signal monitors are more ambitious on the implementation side. Static reasoning over abstract conditions results in the removal of extraneous signaling overheads, enabling high-performance execution.

Automation of synchronization. Static analysis has been applied to the optimization of synchronization primitives in the past, mostly in the context of auto-locking or lock-inference techniques [54, 93, 131, 150, 175]. The language model these techniques implement is much closer to transactional memory than implicit-signal monitors: the analysis attempts to infer which locks protect which shared data, as well as which data are not thread-shared. Furthermore, these static analyses typically produce a whole-program model of shared data and do not reason over symbolic conditions, as in our approach. Techniques that infer synchronization given specifications and abstractions [48, 92, 208] often use similar reasoning techniques as our approach (e.g., SMT solvers), but start from much more abstract input and place an emphasis on correctness, not on approaching hand-written code performance. Similar comments apply to approaches that synthesize synchronization actions, given abstract models of program behavior, using control theory techniques [152, 209].

Program analysis for concurrency. A major application of static analysis in the domain of concurrency has been in guaranteeing safety or finding bugs. There are techniques for ensuring safe programming using advanced typing [37], analyses for static race detection [183], approaches to concurrency bug fixing [146], tools that flag suspicious concurrency patterns [139, 214], and much more. Additionally, dynamic techniques for concurrency bug detection [1, 77, 148, 171, 196] often benefit from symbolic reasoning, especially in approaches inspired by model checking or symbolic execution techniques [147, 153, 182]. Such past work is only superficially related to ours, since the aims, programming abstractions, and analysis techniques used are quite different.

Abductive reasoning in program analysis. Our proposed approach uses abductive reasoning for automatically inferring monitor invariants. The use of abduction in program analysis is quite common, especially in the context of modular analysis [5, 44, 87], loop invariant generation [86], and specification inference [5, 222]. However, our use of abductive reasoning differs from prior work in that we use abduction to generate candidate predicates over which we synthesize monitor invariants using predicate abstraction.

5.3 Related Work for CFPChecker

We now survey prior work related to our technique for verifying correct usage of context-free API protocols 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 [15, 97, 201]. This problem is commonly known as *typestate analysis* [201], and researchers have proposed many different approaches to solve this problem ranging from language-based solutions [7, 31, 76, 105] to program analysis [35, 36, 97] and model checking [15, 19] to bug finding [149, 217] and run-time verification [8, 52]. Some prior works have also proposed various generalizations of typestate properties, such as multi-object protocols [25, 189].

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 [73, 145, 174, 178] 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 [140]. 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 [62, 63, 110, 114, 121, 126, 127, 129], 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 [170] 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. [104] 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 [114, 126, 176, 177], our method also uses *Craig interpolation* to learn new predicates when a spurious counterexample is discovered. Given an unsatisfiable formula $\phi \land \psi$, a Craig interpolant is another formula χ such that $\phi \Rightarrow \chi$ is valid and $\psi \land \chi$

is unsatisfiable. Prior work has proposed many variants of Craig interpolation, including sequence interpolants [126], tree interpolants [34], nested interpolants [122], and DAG interpolants [6]. In this thesis, we leverage the notion of nested interpolants introduced by Heizmann et al. [122] 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 [14, 70, 101, 113]. 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 et al. [101], they apply control-flow refinement within a single procedure and only inside loops. On the other hand, Flores-Montoya et al. [101] refine cost equations rather than the program abstraction. In contrast to all of these techniques, our technique refines the CFG abstraction, performs cloning interprocedurally, 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 [203]. 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 [26, 107, 112, 203] 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 [119, 155, 173, 186]. While checking inclusion between arbitrary context-free languages is known to be undecidable, prior work has studied decidable fragments, such as LL(k) grammars [186]. Our implementation makes use of the algorithm by Madhavan et al. [173], 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 [191]. This formulation has been used to express several fundamental program analyses, such as context-sensitive pointer analysis [200, 215]. 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 [190]; hence many techniques over-approximate one of the two CFL reachability problems [50, 166, 197, 198, 200, 215, 218] or propose a more precise generalization of CFL reachability [202, 219]. 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 [9, 53, 94, 127]. 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 [47, 205, 206], and such multi-stack VPAs are significantly more expressive compared to standard VPAs. For example, 2-VPAs [47] (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 thesis using 2-VPAs, however, the emptiness problem for 2-VPAs is also undecidable.

Chapter 6

Future Work

The work presented in this dissertation is a just a first step towards automatically aiding developers to develop and analyze their software. The problems addressed in this thesis open the way for several future research directions. Next, we briefly discuss some of those here.

Inferring API Protocols Currently CFPChecker requires the user to provide the API protocol as input. However, as we have seen in Chapter 4 some of these protocols can be quite complicated. Therefore, a promising future direction would be to automatically infer an API's protocol automatically. Besides providing an easier interface to CFPChecker users, such a technique could be used by other automated tools, like Expresso, which can increase their precision by leveraging CFPChecker's technique.

Improve Performance of Concurrent Code As shown in Chapter 3, Expresso ensures that different monitor operations run atomically by acquiring and releasing a single *global* lock upon entering and exiting each of these operations. However, this strategy prevents two independent operations to run in parallel since both of them have to acquire the same lock. As a future direction, we plan to extend EXPRESSO so it produces an explicit-signal monitor that implements a fine-grained locking policy that enables such operations to run in parallel. Our goal is to synthesize a monitor that is optimal with respect to both thread signalling and locking.

Appendices

Appendix A

Proofs for Failure-Directed Program Trimming

A.1 Proof of Theorem 1

Proof sketch. For most statements (e.g., assignment, assumption, assertion), Φ' is just the standard weakest precondition of s with respect to Φ.

For heap reads and writes, we already argued why $\Phi' \Rightarrow \mathit{wp}(s, \Phi)$. The heap allocation rule is also correct since it "havocs" the allocated pointer.

The correctness of the procedure call rule follows from the following two facts: First, $\mathit{summary}(\mathit{prc}, \Upsilon, \bar{v})$ is a conservative safety condition for the call to f. In particular, if $f \in \mathit{dom}(\Upsilon)$, this follows from the soundness of Υ. If $f \notin \mathit{dom}(\Upsilon)$, *false* (resp. *true*) is a sufficient condition for the safety of any procedure that does (resp. does not) contain an assertion. Second, we "havoc" the value of any memory location modified in f. The correctness of our *havoc* operation follows from (a) the correctness of the *store* function, and (b) $\forall v.\phi \Rightarrow \mathit{wp}(v := e, \phi)$ for any expression e. □

A.2 Proof of Theorem 2

Proof. The proof is by induction on the number of statements (i.e., $n - i$).

Suppose $i = n$. If s_n is not an assertion, then the safety condition is *true*, so we add `assume` *false*. Since s_n can never fail, *false* is indeed necessary for failure. If s_n is `assert` ϕ, then the necessary condition for failure is $\neg\phi$. Since the safety condition for s_n is ϕ, our technique instruments the code with `assume` $\neg\phi$.

For the inductive step, suppose $i < n$ and let:

$$\Lambda, \Upsilon, \mathit{true} \vdash s_{i+1}; \ldots; s_n : \Phi$$

By the inductive hypothesis, $\neg\Phi$ is a necessary condition for the failure of $s_{i+1}, \ldots, s_n$. We consider three cases: (1) s_i is an assertion `assert` ϕ. Then, the necessary condition for the failure of $s_i; \ldots; s_n$ is $\neg\phi \vee \neg\Phi$. Since the safety condition for $s_i; \ldots; s_n$ is $\phi \wedge \Phi$, our technique instruments the code with `assume` $\neg\phi \vee \neg\Phi$. (2) If s_i is an assumption `assume` ϕ, the necessary condition for failure is $\phi \wedge \neg\Phi$, which is exactly the trimming condition computed by our technique. (3) Otherwise, the necessary condition for failure is $wp(s_i, \neg\Phi)$. Suppose $\Lambda, \Upsilon, \Phi \vdash s_i : \Phi'$. By soundness of the safety condition inference, we have $\Phi' \Rightarrow wp(s_i, \Phi)$, and we instrument the code with `assume` $\neg\Phi'$. Since s_i is neither an assertion nor an assumption, we have $wp(s_i, \neg\Phi) \equiv \neg wp(s_i, \Phi)$; thus, $wp(s_i, \neg\Phi) \Rightarrow \neg\Phi'$. $\square$

Appendix B

Proofs and Auxiliary Defintions for Automatic Signaling

B.1 Well-formedness of Monitor Trace

In this section, we formalize what it means for a monitor trace τ to be syntactically well-formed. Towards this goal, we first define the projection of a trace τ onto a thread t, denoted as $\tau \downarrow t$.

Definition 16. *The projection of trace τ onto thread t, denoted $\tau \downarrow t$, is defined as follows:*

$$\begin{array}{rcll} (t', w, \text{true}) \downarrow t & = & [w] & \text{if } t' = t \\ (t', w, b) \downarrow t & = & \textit{nil} & \text{if } t' \neq t \text{ or } b = \text{false} \\ e :: \tau \downarrow t & = & (e \downarrow t) :: (\tau \downarrow t) & \end{array}$$

Definition 17. *We say that $\tau \downarrow t$ is well-formed if $\tau \downarrow t = [S_1, \ldots, S_n, S_{n+1}]$, each S_i where $i \in [1, n]$ corresponds to the body of some method, and S_{n+1} is a prefix of the body of some method.*

Using these definitions, we can now define syntactic well-formedness of monitor traces as follows:

Definition 18. *We say that a monitor trace τ is syntactically well-formed if the following conditions are satisfied:*

1. *For every thread t, $\tau \downarrow t = S_1; \ldots; S_n; S_{n+1}$ is well-formed*

2. *If $\tau[i] = (t, w, \text{true})$ and $\tau[i+1] = (t', w', b)$, then either w is the last statement in its method or $t' = t$ and w' is the successor of w.*

B.2 Soundness Proof

In this section, we prove the soundness of the PLACESIGNAL algorithm. The proof of Theorem 6 follows immediately from Lemmas 1 and 2.

In the remainder of this section, we use the term *event* to also refer to pairs (t, w) without the corresponding boolean. We refer to t as $\mathit{thread}(e)$ and, if $w = (p, s)$, we refer to p as $\mathit{pred}(e)$. Given two sets $\mathcal{N}, \mathcal{N}'$, we will assume that the total order relation $\prec$ on events is defined in a way that respects the property $\mathcal{N}' \subseteq \mathcal{N} \Rightarrow \min(\mathcal{N}) = \min(\mathcal{N}')$.

Furthermore, for the rest of the section it is useful to keep in mind that the source language maintains the following invariant:

Invariant If $(\sigma_0, \tau, \emptyset, \emptyset) \longrightarrow^* (\sigma, \_, \mathcal{B}, \mathcal{N})$, then:

- If $(t, w) \in \mathcal{N}$, then we have $(t, w) \in \mathcal{B}$.
- If $(t, w) \in \mathcal{B}$ and $(\sigma, t) \models \mathit{Guard}(w)$, then $(t, w) \in \mathcal{N}$.

Lemma 1. *Let $M' = \mathit{PlaceSignals}(M, I)$. If I is a correct monitor invariant, then for all monitor states σ and all well-formed traces τ, if $(\sigma, \tau, \emptyset, \emptyset) \Longrightarrow^* (\sigma', \epsilon, \_, \_)$ then $(\sigma, \tau, \emptyset, \emptyset) \longrightarrow^* (\sigma', \epsilon, \_, \_)$.*

Proof. Follows from Lemma 3. □

Lemma 2. *Let $M' = PlaceSignals(M, I)$. If I is a correct monitor invariant, then for all monitor states σ and all well-formed traces τ. If $(\sigma, \tau, \emptyset, \emptyset) \longrightarrow^* (\sigma', \epsilon, \_, \_)$ and τ is normalized with respect to σ, then $(\sigma, \tau, \emptyset, \emptyset) \Longrightarrow^* (\sigma', \epsilon, \_, \_)$.*

Proof. Follows from Lemma 4. □

Definition 19. (Invalidation). *Let $e = (w, t)$ be a monitor event where $w = (p, s)$, t be a thread identifier, and p a predicate. We write $\text{Invalidate}(e, t', p')$, if, for any state σ such that $\langle s, t, \sigma \rangle \Downarrow \sigma'$, we have $(\sigma', t') \not\models p'$.*

Definition 20. (Agreement). *We say that $(\sigma, \mathcal{B}, \mathcal{N})$ agrees with $(\sigma, \mathcal{B}', \mathcal{N}')$, written $(\sigma, \mathcal{B}, \mathcal{N}) \sim (\sigma, \mathcal{B}', \mathcal{N})'$ if the following conditions are satisfied:*

1. $\mathcal{B} = \mathcal{B}'$
2. $\mathcal{N} \supseteq \mathcal{N}'$
3. *If $e \in (\mathcal{N} - \mathcal{N}')$, then either*
   (a) $(\sigma, \text{thread}(e)) \not\models \text{pred}(e)$, *or*
   (b) $\exists e' \in \mathcal{N}'.\ e' \prec e \wedge \text{pred}(e) = \text{pred}(e')$
   $\wedge\ \text{Invalidate}(e', \text{thread}(e), \text{pred}(e))$

Lemma 3. *If $(\sigma, \mathcal{B}, \mathcal{N}) \sim (\sigma, \mathcal{B}', \mathcal{N}')$ and*

$$(\sigma, \tau, \mathcal{B}', \mathcal{N}') \Longrightarrow (\sigma', \tau', \mathcal{B}_1, \mathcal{N}_1)$$

then we have

$$(\sigma, \tau, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \tau', \mathcal{B}_2, \mathcal{N}_2)$$

and $(\sigma', \mathcal{B}_1, \mathcal{N}_1) \sim (\sigma', \mathcal{B}_2, \mathcal{N}_2)$.

Proof. The proof is by induction on τ.

- **Base case 1a:** $\tau = e = (t, w, \mathit{false})$ and $(t, w) \notin \mathcal{B}'$. In this case, we have $(\sigma, t) \not\models \mathit{Guard}(w)$ and $(\sigma, e, \mathcal{B}', \mathcal{N}') \Longrightarrow (\sigma, \epsilon, \mathcal{B}' \cup \{\overline{e}\}, \mathcal{N}')$. Since $\mathcal{B} = \mathcal{B}'$, we have $(t, w) \notin \mathcal{B}$. Thus, we also have $(\sigma, e, \mathcal{B}', \mathcal{N}) \longrightarrow (\sigma, \epsilon, \mathcal{B}' \cup \{\overline{e}\}, \mathcal{N})$. Since $\mathcal{N}, \mathcal{N}'$ satisfy conditions (2) and (3) of the agreement definition and the state σ has not changed, $\mathcal{N}, \mathcal{N}'$ continue to satisfy conditions (2) and (3). Thus, we have $(\sigma, \mathcal{B}' \cup \{\overline{e}\}, \mathcal{N}') \sim (\sigma, \mathcal{B} \cup \{\overline{e}\}, \mathcal{N})$.

- **Base case 1b:** $\tau = e = (t, w, \mathit{false})$ and $(t, w) \in \mathcal{N}'$. In this case, we have $(\sigma, e, \mathcal{B}', \mathcal{N}') \Longrightarrow (\sigma, \epsilon, \mathcal{B}', \mathcal{N}' \backslash \{\overline{e}\})$. Since $\mathcal{N} \supseteq \mathcal{N}'$, $(t, w) \in \mathcal{N}$, we also have $(\sigma, e, \mathcal{B}', \mathcal{N}) \longrightarrow (\sigma, \epsilon, \mathcal{B}', \mathcal{N} \backslash \{\overline{e}\})$. Since $\mathcal{N}, \mathcal{N}'$ satisfy conditions (2) and (3) of the agreement definition and the state σ has not changed, $\mathcal{N}, \mathcal{N}'$ continue to satisfy conditions (2) and (3). Thus, $(\sigma, \mathcal{B}', \mathcal{N}' \backslash \{\overline{e}\}) \sim (\sigma, \mathcal{B}', \mathcal{N} \backslash \{\overline{e}\})$.

- **Base case 2a:** $\tau = e = (t, w, \mathit{true})$ and $\overline{e} \notin \mathcal{B}'$. In this case, we have:

$$\frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} \notin \mathcal{B}' \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}_2 = \mathit{GetSignals}(w, \sigma', \mathcal{B}) \\ \mathcal{N}_3 = \mathit{GetBroadcasts}(w, \sigma', \mathcal{B}) \end{array}}{(\sigma, e, \mathcal{B}', \mathcal{N}') \Longrightarrow (\sigma', \epsilon, \mathcal{B}', \mathcal{N}' \cup \mathcal{N}_2 \cup \mathcal{N}_3)}$$

Since $\mathcal{B} = \mathcal{B}'$, this implies $\overline{e} \notin \mathcal{B}$, thus, we have:

$$\frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} \notin \mathcal{B} \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}_1 = \{(t, w) \mid (t, w) \in \mathcal{B}, (\sigma', t) \models \mathit{Guard}(w)\} \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \epsilon, \mathcal{B}, \mathcal{N} \cup \mathcal{N}_1)}$$

Part (a) of the agreement definition trivially holds, since $\mathcal{B} = \mathcal{B}'$. For part (b), we need to show that $\mathcal{N}_1 \cup \mathcal{N} \supseteq \mathcal{N}_2 \cup \mathcal{N}_3 \cup \mathcal{N}'$. That is, if $e^* \in \mathcal{N}_2 \cup \mathcal{N}_3 \cup \mathcal{N}'$, we also have $e^* \in \mathcal{N}_1 \cup \mathcal{N}$. Case 1: Suppose $e^* \in \mathcal{N}'$. By agreement, we have $e^* \in \mathcal{N}$, thus part (b) of agreement holds in this case. Case 2: Suppose $e^* \in \mathcal{N}_2 \cup \mathcal{N}_3$. This means $(\mathit{pred}(e^*), c) \in P_1 \cup P_2$ where $P_1 = \mathit{Signals}(w)$ and $P_2 = \mathit{Broadcasts}(w)$. Case 2a: If $c = \mathbf{?}$, then e^* is only added to $\mathcal{N}_2 \cup \mathcal{N}_3$ if $(\sigma', \mathit{thread}(e^*)) \models \mathit{pred}(e^*)$. But then, we have $e^* \in \mathcal{N}_1$. Case 2b: If $c = \checkmark$, then the PLACESIGNALS algorithm ensures that $\vdash \{\ I \wedge \neg \mathit{pred}(e^*)\}\mathit{Body}(w)\{\mathit{pred}(e^*)\}$. By correctness of this Hoare triple, we have $(\sigma', \mathit{thread}(e^*)) \models \mathit{pred}(e^*)$. Thus, $e \in \mathcal{N}_1$.

For part (c) of the agreement definition, we need to show that if $e^* \in (\mathcal{N} \cup \mathcal{N}_1) \backslash (\mathcal{N}' \cup \mathcal{N}_2 \cup \mathcal{N}_3)$, then either (i) $(\sigma', \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$, or (ii) $\exists e' \in \mathcal{N}_2 \cup \mathcal{N}_3 \cup \mathcal{N}$ such that $e' \prec e^* \wedge \mathit{pred}(e) = \mathit{pred}(e^*) \wedge \mathit{Invalidate}(e', \mathit{thread}(e^*), \mathit{pred}(e^*))$. We consider two cases:

- Case 1: $e^* \in \mathcal{N}$, but $e^* \notin \mathcal{N}' \cup \mathcal{N}_2 \cup \mathcal{N}_3$ (i.e., $e^* \notin \mathcal{N}'$, $e^* \notin \mathcal{N}_2$, $e^* \notin \mathcal{N}_3$). Since $e^* \in \mathcal{N} - \mathcal{N}'$, we have (from agreement) that either (a) $(\sigma, \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$, or
  (b) $\exists e' \in \mathcal{N}'$ such that $e' \prec e^*$, $\mathit{pred}(e') = \mathit{pred}(e^*)$ and $\mathit{Invalidate}(e', \mathit{thread}(e^*), \mathit{pred}(e^*))$.

  * Case 1a: $(\sigma, \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$. If we also have $(\sigma', \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$, then part (c) of agreement definition obviously holds. If $(\sigma', \mathit{thread}(e^*)) \models \mathit{pred}(e^*)$, then $e^* \in \mathcal{N}_1$, and we argue correctness in Case 2.
  * Case 1b: $(\sigma, \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$ and $\exists e' \in \mathcal{N}'$ such that $(e' \prec e^*, \mathit{pred}(e') = \mathit{pred}(e^*)$, and $\mathit{Invalidate}(e', \mathit{thread}(e^*), \mathit{pred}(e^*))$. Since such an e' is also in $\mathcal{N}' \cup \mathcal{N}_2 \cup \mathcal{N}_3$, part (c) of the agreement definition is preserved.
- Case 2: $e^* \in \mathcal{N}_1$, but $e^* \notin \mathcal{N}'$, $e^* \notin \mathcal{N}_2$, and $e^* \notin \mathcal{N}_3$. There are only two ways in which $e^* \in \mathcal{N}_1$ but not in $\mathcal{N}_2$ or $\mathcal{N}_3$:
  * Case 2a: $(\mathit{pred}(e^*), c) \notin P_1 \cup P_2$ where $P_1 = \mathit{Signals}(w)$ and $P_2 \in \mathit{Broadcasts}(w)$. Using the invariant of the PlaceSignals algorithm, this can only happen if $\vdash \{I \wedge \neg \mathit{pred}(e^*)\} \mathit{Body}(w) \{\neg \mathit{pred}(e^*)\}$. If $\mathit{pred}(e^*)$ was true before executing $\mathit{Body}(w)$ (i.e., $(\sigma, \mathit{thread}(e^*)) \models \mathit{pred}(e^*)$), then e^* would also be in $\mathcal{N}$, which implies it is also in $\mathcal{N}'$, so this case is not possible. Otherwise, if $(\sigma, \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$, the correctness of the Hoare triple implies that $(\sigma', \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$, which contradicts the fact that $e^* \in \mathcal{N}_1$.
  * Case 2b: $(\mathit{pred}(e^*), c) \in P_2$, but there exists another e' such that (i) $e' \prec e^*$, and (ii) $\mathit{pred}(e^*) = \mathit{pred}(e)$. The only way in which e^* is added to P_2 but not P_1 by the PlaceSignals algorithm is if the following Hoare triple holds for all waituntil statements w with

guard *pred*(e^*):

$$\{I \wedge \mathit{Guard}(w)\}\ \mathit{Body}(w)\ \{\neg \mathit{Guard}(w)\}\}$$

Since e' has the same guard as *pred*(e^*), the validity of the Hoare triple implies $\mathit{Invalidate}(e', \mathit{thread}(e^*), \mathit{pred}(e^*))$. Thus part (c) of the agreement definition again holds.

- **Base case 2b:** $\tau = e = (t, w, \mathit{true})$ and $\overline{e} = \mathit{min}(\mathcal{N}')$. In this case, we have:

$$\frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} = \min(\mathcal{N}') \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}_2 = \mathit{GetSignals}(w, \sigma', \mathcal{B}) \\ \mathcal{N}_3 = \mathit{GetBroadcasts}(w, \sigma', \mathcal{B}) \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}') \Longrightarrow (\sigma', \epsilon, \mathcal{B}\backslash\{\overline{e}\}, (\mathcal{N}' \cup \mathcal{N}_2 \cup \mathcal{N}_3)\backslash\{\overline{e}\})}$$

Since $\mathcal{N}' \subseteq \mathcal{N}$, we have $\min(\mathcal{N}) = \min(\mathcal{N}')$; hence:

$$\frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} = \min(\mathcal{N}) \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}_1 = \{(t, w) \mid (t, w) \in \mathcal{B}, (\sigma', t) \models \mathit{Guard}(w)\} \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \epsilon, \mathcal{B}\backslash\{\overline{e}\}, (\mathcal{N} \cup \mathcal{N}_1)\backslash\{\overline{e}\})}$$

Because the sets $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$ are constructed in exactly the same way as Base case 2a, the same argument also applies in this case. However, we additionally need to show that if $e^* \in \mathcal{N}\backslash\mathcal{N}'$, and $\mathit{pred}(e) = \mathit{pred}(e^*)$, then

$(\sigma', \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$. Since $e \in \mathcal{N}'$ and $\mathit{pred}(e^*) = \mathit{pred}(e)$ and $e \prec e^*$, we have $\mathit{Invalidate}(e, \mathit{thread}(e^*), \mathit{pred}(e))$. By definition of the *Invalidate* predicate, this means that $(\sigma', \mathit{thread}(e^*)) \not\models \mathit{pred}(e^*)$.

- **Inductive step:** If trace τ contains multiple events, then we have:

$$\frac{(\sigma', e, \mathcal{B}', \mathcal{N}') \Longrightarrow (\sigma_2, \epsilon, \mathcal{B}_2, \mathcal{N}_2)}{(\sigma', e :: \tau, \mathcal{B}', \mathcal{N}') \Longrightarrow (\sigma_2, \tau, \mathcal{B}_2, \mathcal{N}_2)}$$

  and

$$\frac{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma_1, \epsilon, \mathcal{B}_1, \mathcal{N}_1)}{(\sigma, e :: \tau, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma_1, \tau, \mathcal{B}_1, \mathcal{N}_1)}$$

  Using the inductive hypothesis and the assumption that $(\sigma', \mathcal{B}', \mathcal{N}') \sim (\sigma, \mathcal{B}, \mathcal{N})$, we have $(\sigma_2, \mathcal{B}_2, \mathcal{N}_2) \sim (\sigma_1, \mathcal{B}_1, \mathcal{N}_1)$.

□

Lemma 4. *If* $(\sigma, \mathcal{B}, \mathcal{N}) \sim (\sigma, \mathcal{B}', \mathcal{N}')$, τ *is normalized, and*

$$(\sigma, \tau, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \tau', \mathcal{B}_1, \mathcal{N}_1)$$

then we have

$$(\sigma, \tau, \mathcal{B}', \mathcal{N}') \Longrightarrow (\sigma', \tau', \mathcal{B}_2, \mathcal{N}_2)$$

and $(\sigma', \mathcal{B}_1, \mathcal{N}_1) \sim (\sigma', \mathcal{B}_2, \mathcal{N}_2)$.

Proof. The proof is by induction on τ.

- **Base case 1a:** $\tau = e = (t, w, \mathit{false})$ and $(t, w) \notin \mathcal{B}$. In this case, we have $(\sigma, t) \not\models \mathit{Guard}(w)$ and $(\sigma, e, \mathcal{B}', \mathcal{N}') \longrightarrow (\sigma, \epsilon, \mathcal{B}' \cup \{\overline{e}\}, \mathcal{N}')$. Since $\mathcal{B} = \mathcal{B}'$, we have $(t, w) \notin \mathcal{B}'$. Thus, we also have $(\sigma, e, \mathcal{B}', \mathcal{N}) \Longrightarrow (\sigma, \epsilon, \mathcal{B}' \cup \{\overline{e}\}, \mathcal{N})$. Since $\mathcal{N}, \mathcal{N}'$ satisfy conditions (2) and (3) of the agreement definition and the state σ has not changed, $\mathcal{N}, \mathcal{N}'$ continue to satisfy conditions (2) and (3). Thus, we have $(\sigma, \mathcal{B}' \cup \{\overline{e}\}, \mathcal{N}') \sim (\sigma, \mathcal{B} \cup \{\overline{e}\}, \mathcal{N})$.

- **Base case 1b:** $\tau = e = (t, w, \mathit{false})$ and $(t, w) \in \mathcal{N}$. We do not need to consider this case because it contradicts the assumption that the trace is normalized.

- **Base case 2a:** $\tau = e = (t, w, \mathit{true})$ and $\overline{e} \notin \mathcal{B}$. In this case, we have:

$$\frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} \notin \mathcal{B} \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}_1 = \{(t, w) \mid (t, w) \in \mathcal{B}, (\sigma', t) \models \mathit{Guard}(w)\} \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \epsilon, \mathcal{B}, \mathcal{N} \cup \mathcal{N}_1)}$$

Since $\mathcal{B} = \mathcal{B}'$, this implies $\overline{e} \notin \mathcal{B}'$, thus, we also have:

$$\frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} \notin \mathcal{B}' \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma \rangle \Downarrow \sigma' \\ \mathcal{N}_2 = \mathit{GetSignals}(w, \sigma', \mathcal{B}) \\ \mathcal{N}_3 = \mathit{GetBroadcasts}(w, \sigma', \mathcal{B}) \end{array}}{(\sigma, e, \mathcal{B}', \mathcal{N}') \Longrightarrow (\sigma', \epsilon, \mathcal{B}', \mathcal{N}' \cup \mathcal{N}_2 \cup \mathcal{N}_3)}$$

Thus, the proof is exactly the same as Base Case 2a of Lemma 3

- **Base case 2b:** $\tau = e = (t, w, \mathit{true})$ and $\overline{e} = \mathit{min}(\mathcal{N})$. In this case, we have:

$$\frac{\begin{array}{c} e = (t, w, \mathit{true}) \\ \overline{e} = \min(\mathcal{N}) \quad (\sigma, t) \models \mathit{Guard}(w) \\ \langle \mathit{Body}(w), t, \sigma\rangle \Downarrow \sigma' \\ \mathcal{N}_1 = \{(t, w) \mid (t, w) \in \mathcal{B}, (\sigma', t) \models \mathit{Guard}(w)\} \end{array}}{(\sigma, e, \mathcal{B}, \mathcal{N}) \longrightarrow (\sigma', \epsilon, \mathcal{B}\backslash\{\overline{e}\}, (\mathcal{N} \cup \mathcal{N}_1)\backslash\{\overline{e}\})}$$

The existence of e in $\mathcal{N}$ does not guarantee the existence of e in N'. We consider two cases, namely (1) $e \in \mathcal{N}'$ and (2) $e \notin \mathcal{N}'$. For case (1), the explicit-signal transitions also use rule (2b), thus the proof is exactly the same as Case 2b of the proof of Lemma 3.

We will now argue that case 2 (i.e., $e \notin \mathcal{N}'$) is not possible. Suppose $e \in \mathcal{N}$, but not in $\mathcal{N}'$. Then, by part (c) of the agreement definition, we have either (i) $(\sigma, t) \not\models \mathit{Guard}(w)$, or $\exists e' \in \mathcal{N}'.\ e' \prec e \wedge \mathit{pred}(e) = \mathit{pred}(e') \wedge \mathit{Invalidate}(e', t, \mathit{pred}(e))$. Observe that (i) contradicts the assumption $(\sigma, t) \models \mathit{Guard}(w)$. For (ii), we have $\exists e' \in \mathcal{N}'.\ e' \prec e \wedge \mathit{pred}(e) = \mathit{pred}(e') \wedge \mathit{Invalidate}(e', t, \mathit{pred}(e))$. But since $\mathcal{N} \supseteq \mathcal{N}'$, e' is also in $\mathcal{N}$ and furthermore $e' \prec e$. But this contradicts the assumption that e' is the minimum element of $\mathcal{N}$.

- **Inductive step:** Same as the inductive step of the proof of Lemma 3.

$\square$

B.3 Proof of Theorem 4

The proof is by induction on the length of τ_0. For the base case, we have $\tau_0 = \epsilon$, thus $\tau' = \tau$ and the statement trivially holds. For the inductive step, we have $\tau_0 = e'\tau_1$. Suppose $(\sigma, e'\tau_1 e, \mathcal{B}, \mathcal{N}) \longrightarrow^* (\sigma, \epsilon, \mathcal{B}', \mathcal{N}')$ where $(\sigma, e', \mathcal{B}, \mathcal{N}) \longrightarrow^* (\sigma_1, \epsilon, \mathcal{B}_1, \mathcal{N}_1)$ as well as $(\sigma_1, \tau_1 e, \mathcal{B}_1, \mathcal{N}_1) \longrightarrow^* (\sigma', \epsilon, \mathcal{B}', \mathcal{N}')$. By the inductive hypothesis, we have $(\sigma_1, e\tau_1, \mathcal{B}_1, \mathcal{N}_1) \longrightarrow^* (\sigma', \epsilon, \mathcal{B}', \mathcal{N}')$. This implies $(\sigma, e'e\tau_1, \mathcal{B}, \mathcal{N}) \longrightarrow^* (\sigma', \epsilon, \mathcal{B}', \mathcal{N}')$. Finally, from the assumption $\mathit{Comm}(\mathit{Body}(e), M)$, we have $\mathit{Body}(e); \mathit{Body}(e') \equiv \mathit{Body}(e'); \mathit{Body}(e)$. Thus, we also have $(\sigma, ee'\tau_1, \mathcal{B}, \mathcal{N}) \longrightarrow^* (\sigma', \epsilon, \mathcal{B}', \mathcal{N}')$

B.4 Sample Monitor Invariant

To give the reader some idea about the inferred monitor invariants, we show the invariant (in SMTLIB format) for the AsyncDispatch benchmark from Gradle:

```
(let ((a!1 (not (= (queue.size))
                  0)))
      (a!3 (not (>= (queue.size))
                   (+ 1
                      (maxQueueSize)))))
      (a!5 (not (>= (queue.size))
                   (maxQueueSize)))))
(let ((a!2 (not (or (= (state)
                      Stopped)
                   a!1)))
      (a!4 (or (= (queue.size)
                  0)
               (= (queue.first)
                  0)
               (= (state)
                  Stopped)
               a!3)))
(let ((a!6 (and a!4
                (or a!1
                   (= (state)
                   Stopped)
                a!5))))
  (and (or a!2 a!6)
       (not (<= (maxQueueSize)
                0))
       (>= (maxQueueSize)
           0)
       (>= (queue.size)
           0)))))
```

As another example, we show the monitor invariant for the Bounded-Buffer benchmark:

```
(let ((a!1 (not (>= buff.length
                    0)))
      (a!2 (not (>= (count)
                    buff.length)))
      (a!4 (>= (count)
               (+ 1
                  (buff.length))))
(let ((a!3 (or a!1
               (not (or a!1 a!2))
               (not (<= (count)
                        (- 1))))))
  (and a!3
       (>= (count)
           0)
       (or a!1
           (<= (count)
               0)
           (not a!4)))))
```

Appendix C

Proofs and Auxiliary Defintions for CFPChecker

C.1 Correctness of Program Instrumentation

In this section we prove the correctness of the program instrumentation presented in Section 4.3. The proof of Theorem 5 follows from lemma 5.

Lemma 5. *If we have* $\Gamma, \mathcal{G}_S \vdash \mathcal{P} \hookrightarrow \mathcal{P}'$*, then for every trace* $\tau \in \mathcal{P}$ *and* $\hat{\mathcal{G}} \in \mathit{Inst}(\mathcal{G}_S, \tau)$ *there exists a trace* $\tau' \in \mathcal{P}'$ *such that* $\text{TraceToWord}(\tau, \hat{\mathcal{G}}) = \text{TraceToWord}(\tau', \hat{\mathcal{G}})$.

Proof. We assume the existence of a map $\Lambda : \mathit{Loc} \to \mathit{Loc}$, which given a program location in $\mathcal{P}$ it returns the corresponding location in $\mathcal{P}'$. For `api_call` statements, Λ returns the location of the instrumented `if`-then-`else` statement as presented in Figure 4.10.

Now, given a trace in $\mathcal{P}$ of the form $\tau = \langle s_1, \sigma_1 \rangle ... \langle s_n, \sigma_n \rangle$ and a specification instantiation $\hat{\mathcal{G}} \in \mathit{Inst}(\mathcal{G}_S, \tau)$, we show how to construct a trace $\tau' \in \mathcal{P}'$ with the same trace projection as τ. For the rest of the proof, we will only describe the statements we append in τ' without showing the corresponding program states since each program state can be obtained by using the operational semantics (i.e., $\Downarrow$ relation) of the preceding (statement, prog. state) pair in τ'. In what follows, we assume that: 1. $\vec{v}$ are the concrete values that instantiate all the wildcards in the specification, i.e., $\hat{\mathcal{G}} = \mathcal{G}_S[\vec{v}/\vec{w}]$. 2. For any $\langle s_i, \sigma_i \rangle \in \tau$ we assume that σ_{i+1} is the result of executing s_i under σ_i, i.e., $\langle s_i, \sigma_i \rangle \Downarrow \sigma_{i+1}$ and 3. For any statement s_i in τ we assume that l_i represents its location in $\mathcal{P}$.

First, for every wildcard field w_i in $\mathcal{P}'$ we append the assignment w_i `=` v_i to τ', where v_i is the i-th element of vector $\vec{v}$. Next, we iterate over every pair $\langle s_i, \sigma_i \rangle$ in τ and update τ' according to the following rules:

1. If s_i is of the form `api_call m(`$\vec{v1}$`)`, then for every `if (guard(`t_i, s_i`))` t in the statement at $\Lambda[l_i]$ update τ' as follows: If we have that `guard(`t_i, s_i`)` evaluates to true for any pair of s_i, t_i under σ_i[1], then append t_i to τ'. Otherwise, append `skip` to τ'.

2. Otherwise, append s_i to τ'.

Intuitively, the first step, i.e., the field initialization, ensures that the wildcard fields in $\mathcal{P}'$ have the same value as the instantiated specification $\hat{\mathcal{G}}$, whereas, the second step ensures that the branches taken in τ' are semantically consistent with the initialization of the wildcard fields. It is easy to see that τ' can be generated by $\mathcal{P}'$ and that $\mathit{TraceToWord}(\tau, \hat{\mathcal{G}}) = \mathit{TraceToWord}(\tau', \hat{\mathcal{G}})$ since the only guards that will evaluate to true are the ones where *all* the program variables of the API call equal to some value in $\vec{v}$. $\square$

[1]This can be easily determined by checking whether each value in $\vec{v1}$ equals to value that instantiates the corresponding wildcard in $\vec{v}$.

C.2 Soundness and Progress Proofs of Main Algorithm

In this section, we prove Theorems 6 and 7, which state the soundness and progress of our approach respectively. Theorem 6 follows from Lemmas 6 and 8. Theorem 7 follows from Lemmas 6 and 7.

Definition 21. PCFA State Sequence: *A PCFA state sequence is a tuple of the form $(s_0 ... s_n, \rightsquigarrow)$, where each s_i is a PCFA state of some method $m \in \mathcal{P}$ and $\rightsquigarrow$ is a nesting relation over the indices $[0, n]$.*

Definition 22. $\mathcal{P}$-feasible State Sequence: *We say that a state sequence is $\mathcal{P}$-feasible and write $\mathcal{P} \vdash (s_0 ... s_n, \rightsquigarrow)$ if and only if the following hold:*

1. *For all i, j such that $i \rightsquigarrow j$, there exists an edge $(s_i, \texttt{call m}, s_j)$ in $\mathcal{P}$. Furthermore, we have $s_{i+1} \in Entry(\mathcal{P}[m])$, $s_{j-1} \in Exit(\mathcal{P}[m])$, and $SAT(Pred(s_i) \wedge Pred(s_{j-1}) \wedge Pred(s_j))$.*

2. *For all $i \in [0, n-1]$ for which $\rightsquigarrow$ is undefined, there exists an edge (s_i, σ, s_{i+1}) in $\mathcal{P}$.*

Definition 23. $\mathcal{P}$-feasible Execution Path: *An execution path $(\sigma_0 ... \sigma_n, \rightsquigarrow)$ is $\mathcal{P}$-feasible through state sequence $(s_0 ... s_{n+1}, \rightsquigarrow)$, denoted as $\mathcal{P}, (s_0 ... s_{n+1}) \vdash (\sigma_0 ... \sigma_n, \rightsquigarrow)$, if and only if $\mathcal{P} \vdash (s_0 ... s_{n+1}, \rightsquigarrow)$ and for every statement σ_i:*

1. *If $i \rightsquigarrow j$, then σ_i is a call statement, σ_{j-1} is the matching return statement, and (s_i, σ_i, s_j) is an edge in the PCFA.*

2. *If σ_i is not a call or a return statement, then (s_i, σ_i, s_{i+1}) is an edge in the PCFA.*

Oracle *deriv2seq* In a similar manner as *derivation2path*, we assume the existence of a similar oracle named *deriv2seq* which given a derivation $d \in \mathcal{G}_{\mathcal{P}}$ it returns its corresponding PCFA sequence $(s_0 ... s_n, \rightsquigarrow)$. This can be easily derived from a derivation since every non-terminal in $\mathcal{G}_{\mathcal{P}}$ is associated with a singe PCFA state.

Lemma 6. *A derivation d belongs to $\mathcal{G}_{\mathcal{P}}$ if and only if $\mathcal{P} \vdash$ deriv2seq(d).*

Proof. This follows immediately from the construction of $\mathcal{G}_{\mathcal{P}}$. Let's assume that for a derivation $d \in \mathcal{G}_{\mathcal{P}}$ we have that *deriv2seq*$(d) = (s_0 ... s_n, \rightsquigarrow)$.

($\Longrightarrow$) For this direction, we will prove that $\mathcal{P} \vdash (s_0 ... s_n, \rightsquigarrow)$. In order for *deriv2seq* to return this sequence the following must hold:

- For all i, j such that $i \rightsquigarrow j$, there must exist a rule in $\mathcal{G}_{\mathcal{P}}$ of the form $\mathcal{S}_{i\phi} \rightarrow \mathcal{M}_{\phi'} \mathcal{S}_{j\phi}$, where ϕ, ϕ' refer to the method clones of the caller and the callee. Additionally we have that $s_{i+1} \in \mathit{Entry}(\mathcal{P}[m])$, $s_{j-1} \in \mathit{Exit}(\mathcal{P}[m])$, and $\mathit{SAT}(\mathit{Pred}(s_i) \wedge \mathit{Pred}(s_j) \wedge \mathit{Pred}(s_{j-1})))$. By construction of $\mathcal{G}_{\mathcal{P}}$, this implies that there exists an edge of the form $(s_i, \texttt{call m}, s_j)$ in $\mathcal{P}$.

- For all i that $\rightsquigarrow$ is undefined, there must exist a rule in $\mathcal{G}_{\mathcal{P}}$ of the form $\mathcal{S}_{i\phi} \rightarrow \mathcal{S}_{i+1\phi}$ or $\mathcal{S}_{i\phi} \rightarrow t\ \mathcal{S}_{i+1\phi}$ where t is a terminal. This also implies that there exist an edge of the form (s_i, σ, s_{i+1}) in $\mathcal{P}$.

Now, it is easy to see that both these conditions satisfy Definition 22, therefore we have that $\mathcal{P} \vdash (s_0 ... s_n, \rightsquigarrow)$.

($\Longleftarrow$) We now prove that if $\mathcal{P} \vdash (s_0...s_n, \rightsquigarrow)$, then there exists $d \in \mathcal{G}_\mathcal{P}$ such that $\mathit{deriv2seq}(d) = (s_0...s_n, \rightsquigarrow)$.

This follows by the construction of $\mathcal{G}_\mathcal{P}$ and Definition 22. Specifically, the first condition of 22 implies that there exists at least one production in $\mathcal{G}_\mathcal{P}$ of the form $\mathcal{S}_{i\phi} \rightarrow \mathcal{M}_{\phi'}\ \mathcal{S}_{j\phi}$ for some clones ϕ and ϕ'. Whereas, the second condition implies that there exists a rule of the form $\mathcal{S}_{i\phi} \rightarrow \mathcal{S}_{i+1\phi}$ or $\mathcal{S}_{i\phi} \rightarrow t\ \mathcal{S}_{i+1\phi}$ for some method clone ϕ. Therefore, we have that there exists at least one derivation $d \in \mathcal{G}_\mathcal{P}$ for which $\mathit{deriv2seq}(d) = (s_0...s_n, \rightsquigarrow)$. $\square$

Lemma 7. *Let $\mathcal{P}'$ be the program returned by procedure* Refine *for spurious counterexample $(\sigma_0...\sigma_n, \rightsquigarrow)$ and sequence of nested interpolants $\mathcal{I} = [I_0, ..., I_{n+1}]$. Then we have that there does not exist state sequence $(s_0...s_{n+1}, \rightsquigarrow)$ such that $\mathcal{P}', (s_0...s_{n+1}) \nvdash (\sigma_0...\sigma_n, \rightsquigarrow)$.*

Proof. Let's assume that $\mathcal{P}$ is the program before the refinement. To simplify the proof we make the following assumptions: 1. $(\sigma_0...\sigma_n, \rightsquigarrow)$ is the first counterexample, i.e., each PCFA state has true as a predicate 2. a program location appears only once in the counterexample and 3. the last statement of `main` is skip. The third assumption just converts the input program to a normal form. Later we show how to generalize the proof so it does not require the first two assumptions.

By Definition 23 and assumptions 1 and 2, there exists a *singe* state sequence $(s_0...s_{n+1}, \rightsquigarrow)$ such that $\mathcal{P}, (s_0...s_{n+1}) \vdash (\sigma_0...\sigma_n, \rightsquigarrow)$ (i.e., the spurious counterexample is $\mathcal{P}$-feasible). Now, by the way procedure Refine works, we have that states s_0 through s_{n+1} have been replaced by states $s'_0, s''_0, ..., s'_{n+1}, s''_{n+1}$ in $\mathcal{P}'$ such that: $Loc(s'_i) = Loc(s''_i) = Loc(s)$ and $Pred(s'_i) = I_i$, $Pred(s''_i) = \neg I_i$. We now show that for any state sequence of the form $(\bar{s}_0...\bar{s}_{n+1}, \rightsquigarrow)$ where $\bar{s}_i \in \{s', s''\}$ we have that: $\mathcal{P}', (\bar{s}_0...\bar{s}_{n+1}) \nvdash (\sigma_0...\sigma_{n+1}, \rightsquigarrow)$.

Now recall that procedure Refine removes all the edges (s_1, σ, s_2) for which $UNSAT(sp(\sigma, Pred(s_1)) \wedge Pred(s_2))$ holds. Since $I_{n+1} = false$ we have that the counterexample cannot be feasible through any state sequence that ends with s'_{n+1} and from the definition of nested interpolants we also get that $UNSAT(sp(\sigma_n, I_n))$. Hence, the counterexample cannot be feasible through any state sequence of the form $(\bar{s}'_0...s'_n s''_{n+1}, \rightsquigarrow)$. Last, we prove that $\mathcal{P}', (\bar{s}_0...s''_n s''_{n+1}) \nvdash (\sigma_0...\sigma_n, \rightsquigarrow)$ for any combination of $\bar{s}_i$ which concludes the proof.

The proof is by induction on the length of the counterexample:

- **Base case**: Here we need to prove that $\mathcal{P}', (s''_0 s''_1) \nvdash (\sigma_0, \rightsquigarrow)$. Since I_0 is true, we have $UNSAT(sp(\sigma_0, Pred(s''_0)))$ therefore the execution path is not feasible through state sequence $(s''_0 s''_1)$

- **Inductive Step**: By inductive hypothesis we have that $\mathcal{P}', (\bar{s}_0...s''_{n+1}) \nvdash (\sigma_0...\sigma_n, \rightsquigarrow)$ for any $\bar{s}_i$. We will now prove that $\mathcal{P}', (s'_0...s'_{n+1}s''_{n+2}) \nvdash (\sigma_0...\sigma_{n+1})$. Here we have two cases:

  1. σ_{n+1} is not a return statement. By the definition of nested interpolants we have that $sp(\sigma_{n+1}, Pred(s'_{n+1})) \Rightarrow Pred(s'_{n+2})$, this

implies $UNSAT(sp(\sigma_{n+1}, Pred(s'_{n+1})) \wedge Pred(s''_{n+2}))$. Therefore, REFINE removes the edge $(s_{n+1}, \sigma_{n+1}, s''_{n+2})$ which make the execution path infeasible through this predicate sequence.

2. σ_{n+1} is a return statement. By the definition of nested interpolants we have that $sp(\sigma_{n+1}, Pred(s'_{n+1}) \wedge Pred(s'_j)) \Rightarrow Pred(s'_{n+2})$ for $j \rightsquigarrow n+2$. This implies that $UNSAT(sp(\sigma_{n+1}, Pred(s'_{n+1}) \wedge Pred(s'_j)) \wedge Pred(s''_{n+2}))$, hence by Definition 23 we have that the execution path is infeasible through this state sequence.

Now, if we lift assumptions 1 and 2 from earlier this means that each state s_i from the feasible state sequence will be replaced by multiple new states in $\mathcal{P}'$. Recall though that procedure REFINE will clone all the states in $\mathcal{P}$ with the same location as s_i and the resulting clones in $\mathcal{P}'$ will have one of the complete cubes as their predicate. This means that for a given interpolant I_i a clone s'_i in $\mathcal{P}'$ will contain either I_i or $\neg I_i$, which implies that $Pred(s'_i) \Rightarrow I_i$ or $Pred(s'_i) \Rightarrow \neg I_i$. Therefore, the proof in the general case is similar to the one above except that one would have to consider *all* the clones of $\mathcal{P}'$ that contain the predicate $\neg I_i$. $\square$

Lemma 8. *If $\mathcal{P}'$ is the program returned by procedure* REFINE *for sequence of nested interpolants $\mathcal{I} = [I_0, ..., I_{n+1}]$, then there exists state sequence $(s_0...s_m, \rightsquigarrow)$ such that $\mathcal{P}', (s_0...s_m) \vdash (\pi, \rightsquigarrow)$ for every feasible execution path in $\mathcal{P}$.*

Proof. Here we make the same assumptions as in Lemma 7, but as before the proof generalizes for the same reasons.

Let's assume that $(\sigma_0 ... \sigma_n, \rightsquigarrow)$ is the spurious counterexample that triggered the refinement. Again, by Definition 23 and assumptions 1 and 2 from Lemma 7, there exists a *singe* state sequence $(s_0 ... s_{n+1}, \rightsquigarrow)$ such that $\mathcal{P}, (s_0 ... s_{n+1}) \vdash (\sigma_0 ... \sigma_n, \rightsquigarrow)$ (i.e., the spurious counterexample is $\mathcal{P}$-feasible). Now, let $s'_0, s''_0, ... s'_n, s''_n$ be the states that replace each state s_i in $\mathcal{P}$ where $\mathit{Pred}(s'_i) = I_i$ and $\mathit{Pred}(s''_i) = \neg I_i$. Now, in order for an execution path to be eliminated by procedure REFINE it must have a statement that labels an edge whose source or destination state is one of the states s_i (the rest of the graph does not change). We prove this is impossible by contradiction:

Let's assume that there exists an execution path $(\pi = \sigma_0 ... \sigma_m, \rightsquigarrow)$ for which there does not exist state sequence $(s0 ... s_{m+1}, \rightsquigarrow)$ in $\mathcal{P}'$ such that $\mathcal{P}', (s0 ... s_{m+1}, \rightsquigarrow) \nvdash (\pi, \rightsquigarrow)$ and also one of the statements σ_j in π labels an edge whose target state[2] is one of the cloned states s'_{j+1} or s''_{j+1}. Now in order for this execution path to not be feasible we must have one of the following:

- If σ_j is not a return statement, then we must have that $\mathit{sp}(\sigma_j, \mathit{Pred}(s_j)) \wedge \mathit{Pred}(s'_{j+1}))$ and $\mathit{sp}(\sigma_j, \mathit{Pred}(s_j)) \wedge \mathit{Pred}(s''_{j+1}))$ which is impossible since $\mathit{Pred}(s'_{j+1}) = I_{j+1}$ and $\mathit{Pred}(s''_{j+1}) = \neg I_{j+1}$.

- If σ_j is a return statement, then we must have that $\mathit{sp}(\sigma_j, \mathit{Pred}(s_j) \wedge$

[2]The case where it labels an edge whose source state is one from the counterexample is similar.

$Pred(s_i)) \wedge Pred(s'_{j+1}))$ and $sp(\sigma_j, Pred(s_j) \wedge Pred(s_i)) \wedge Pred(s''_{j+1}))$ for $i \rightsquigarrow j+1$. Which again it is impossible for the same reason as above.

$\square$

Bibliography

[1] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pages 51–60, New York, NY, USA, 2006. ACM.

[2] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *PLDI*, pages 246–256. ACM, 1990.

[3] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the Saturn project. In *PASTE*, pages 43–48. ACM, 2007.

[4] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, pages 789–801. ACM, 2016.

[5] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *ACM SIGPLAN Notices*, volume 51, pages 789–801. ACM, 2016.

[6] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. Ufo: verification with interpolants and abstract interpre-

tation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 637–640. Springer, 2013.

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

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

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

[10] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[11] Cyrille Artho and Willem Visser. Java pathfinder at sv-comp 2019 (competition contribution). In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 224–228, Cham, 2019. Springer International Publishing.

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

[13] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. In *POPL '77: Proc. 4th symposium on Principles of Programming Languages*, pages 267–280. ACM Press.

[14] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Refining the control structure of loops using static analysis. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 49–58, New York, NY, USA, 2009. ACM.

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

[16] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213. ACM, 2001.

[17] Thomas Ball, Todd Millstein, and Sriram K. Rajamani. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst.*, 27(2):314–343, March 2005.

[18] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, pages 97–105. ACM, 2003.

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

[20] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.

[21] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.

[22] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[23] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The Spec# experience. *CACM*, 54:81–91, 2011.

[24] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87. ACM, 2005.

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

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

[27] Dirk Beyer. Competition on software verification (SV-COMP), 2017. `https://sv-comp.sosy-lab.org`.

[28] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. *STTT*, 9:505–525, 2007.

[29] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: A technique to pass information between verifiers. In *FSE*, pages 57–67. ACM, 2012.

[30] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

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

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

[33] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

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

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

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

[37] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Sigplan Notices*, volume 37, pages 211–230. ACM, 2002.

[38] Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[39] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 243–262. ACM Press, October 2009.

[40] Peter A Buhr, Michel Fortier, and Michael H Coffin. Monitor classification. *ACM Computing Surveys (CSUR)*, 27(1):63–107, 1995.

[41] Peter A Buhr and Ashif S Harji. Implicit-signal monitors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1270–1343, 2005.

[42] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX, 2008.

[43] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.

[44] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *ACM SIGPLAN Notices*, volume 44, pages 289–300. ACM, 2009.

[45] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300. ACM, 2009.

[46] Gerardo Canfora, Aniello Cimitile, and Andrea de Lucia. Conditioned program slicing. *IST*, 40:595–607, 1998.

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

[48] Pavol Černỳ, Thomas A Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *International Conference on Computer Aided Verification*, pages 951–967. Springer, 2013.

[49] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *PLDI*, pages 363–374. ACM, 2009.

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

[51] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC*, pages 1284–1291. ACM, 2012.

[52] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. In *Proceedings of the 22Nd Annual ACM SIG-*

PLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07, pages 569–588, New York, NY, USA, 2007. ACM.

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

[54] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 304–315, New York, NY, USA, 2008. ACM.

[55] Yunja Choi, Mingyu Park, Taejoon Byun, and Dongwoo Kim. Efficient safety checking for automotive operating systems using property-based slicing and constraint-based environment generation. *Sci. Comput. Program.*, 103:51–70, 2015.

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

[57] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating smt solver. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, pages 248–254, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[58] Maria Christakis. *Narrowing the Gap between Verification and Systematic Testing*. PhD thesis, ETH Zurich, Switzerland, 2015.

[59] Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.

[60] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*, pages 144–155. ACM, 2016.

[61] Maria Christakis and Valentin Wüstholz. Bounded abstract interpretation. In *SAS*, volume 9837 of *LNCS*, pages 105–125. Springer, 2016.

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

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

[64] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *FMSD*, 19:7–34, 2001.

[65] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[66] Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *FME*, volume 1051 of *LNCS*, pages 557–575. Springer, 1996.

[67] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[68] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, volume 7737 of *LNCS*, pages 128–148. Springer, 2013.

[69] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *VMCAI*, volume 6538 of *LNCS*, pages 150–168. Springer, 2011.

[70] John Cyphert, Jason Breck, Zachary Kincaid, and Thomas Reps. Refinement of path expressions for static analysis. *Proc. ACM Program. Lang.*, 3(POPL):45:1–45:29, January 2019.

[71] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. Just test

what you cannot verify! In *FASE*, volume 9033 of *LNCS*, pages 100–114. Springer, 2015.

[72] Przemyslaw Daca, Ashutosh Gupta, Henzinger, and Thomas A. Abstraction-driven concolic testing. In *VMCAI*, volume 9583 of *LNCS*, pages 328–347. Springer, 2016.

[73] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. In *Proceedings of the Third International Workshop on Dynamic Analysis*, WODA '05, pages 1–7, New York, NY, USA, 2005. ACM.

[74] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[75] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

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

[77] Dongdong Deng, Wei Zhang, and Shan Lu. Efficient concurrency-bug detection across inputs. In *Acm Sigplan Notices*, volume 48, pages 785–802. ACM, 2013.

[78] Edsger W Dijkstra. Hierarchical ordering of sequential processes. In *The origin of concurrent programming*, pages 198–227. Springer, 1971.

[79] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18:453–457, 1975.

[80] Edsger W. Dijkstra. *Cooperating Sequential Processes*, pages 65–138. Springer New York, New York, NY, 2002.

[81] Isil Dillig and Thomas Dillig. Explain: a tool for performing abductive inference. In *International Conference on Computer Aided Verification*, pages 684–689. Springer, 2013.

[82] Isil Dillig and Thomas Dillig. Explain: A tool for performing abductive inference. In *CAV*, volume 8044 of *LNCS*, pages 684–689. Springer, 2013.

[83] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, pages 270–280. ACM, 2008.

[84] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, volume 6012 of *LNCS*, pages 246–266. Springer, 2010.

[85] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200. ACM, 2011.

[86] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *Acm Sigplan Notices*, volume 48, pages 443–456. ACM, 2013.

[87] Isil Dillig, Thomas Dillig, Boyang Li, Ken McMillan, and Mooly Sagiv. Synthesis of circular compositional program proofs via abduction. *International Journal on Software Tools for Technology Transfer*, 19(5):535–547, 2017.

[88] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456. ACM, 2013.

[89] Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. Optimal guard synthesis for memory safety. In *CAV*, volume 8559 of *LNCS*, pages 491–507. Springer, 2014.

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

[91] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In *ESEC/FSE*, pages 195–204. ACM, 2007.

[92] E Allen Emerson and Roopsha Samanta. An algorithmic framework for synthesis of concurrent programs. In *ATVA*, pages 522–530. Springer, 2011.

[93] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07: Proc. 34th symposium on Principles of Programming Languages*, pages 291–296. ACM Press, 2007.

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

[95] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010.

[96] John Field, Ganesan Ramalingam, and Frank Tip. Parametric program slicing. In *POPL*, pages 379–392. ACM, 1995.

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

[98] Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for esc/java. In *International Symposium of Formal Methods Europe*, pages 500–517. Springer, 2001.

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

[100] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.

[101] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems*, pages 275–295, Cham, 2014. Springer International Publishing.

[102] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. An abstract domain of uninterpreted functions. In *VMCAI*, volume 9583 of *LNCS*, pages 85–103. Springer, 2016.

[103] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Exploiting sparsity in difference-bound matrices. In *SAS*, volume 9837 of *LNCS*, pages 189–211. Springer, 2016.

[104] Pierre Ganty, Rupak Majumdar, and Benjamin Monmege. Bounded underapproximations. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 600–614, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

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

[106] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[107] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 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, pages 43–56, New York, NY, USA, 2010. ACM.

[108] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56. ACM, 2010.

[109] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

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

[111] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 426–438. ACM, 2015.

[112] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. 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, pages 117–127, New York, NY, USA, 2006. ACM.

[113] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 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, pages 375–385, New York, NY, USA, 2009. ACM.

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

[115] Arie Gurfinkel, Ou Wei, and Marsha Chechik. Model checking recursive programs with exact predicate abstraction. In *ATVA*, volume 5311 of *LNCS*, pages 95–110. Springer, 2008.

[116] Per Brinch Hansen. *Operating system principles.* Prentice-Hall, Inc., 1973.

[117] Mark Harman, Robert M. Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *ICSM*, pages 138–147. IEEE Computer Society, 2001.

[118] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 388–402, Anaheim, CA, 2003. ACM Press.

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

[120] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13:315–353, 2000.

[121] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, et al. Ultimate automizer and the search for perfect interpolants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–451. Springer, 2018.

[122] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 471–482. ACM, 2010.

[123] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2013.

[124] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 232–244, New York, NY, USA, 2004. ACM.

[125] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.

[126] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. Abstractions from proofs. In *ACM SIGPLAN Notices*, volume 39, pages 232–244. ACM, 2004.

[127] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[128] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.

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

[130] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[131] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.

[132] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:576–580, 1969.

[133] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971.

[134] C. A. R. Hoare. Towards a theory of parallel programming. In *International Seminar on Operating System Techniques*, 1971.

[135] C. A. R. Hoare and Jifeng He. The weakest prespecification. *Inf. Process. Lett.*, 24:127–132, 1987.

[136] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. In *The origin of concurrent programming*, pages 272–294. Springer, 1974.

[137] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012.

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

[139] David Hovemeyer and William Pugh. Finding concurrency bugs in Java. In *Proc. of PODC*, volume 4, 2004.

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

[141] Wei-Lun Hung and Vijay K. Garg. Autosynch: An automatic-signal monitor based on predicate tagging. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 253–262, New York, NY, USA, 2013. ACM.

[142] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-Soft: Software verification platform. In *CAV*, volume 3576 of *LNCS*, pages 301–306. Springer, 2005.

[143] Joxan Jaffar and Vijayaraghavan Murali. A path-sensitively sliced control flow graph. In *FSE*, pages 133–143. ACM, 2014.

[144] Ranjit Jhala and Rupak Majumdar. Path slicing. In *PLDI*, pages 38–47. ACM, 2005.

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

[146] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, volume 12, pages 221–236, 2012.

[147] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Computer Aided Verification*, pages 675–681. Springer, 2009.

[148] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM, 2010.

[149] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded java programs. In *Proceedings of the 2008 23rd IEEE/ACM*

international conference on automated software engineering, pages 288–296. IEEE Computer Society, 2008.

[150] Vineet Kahlon. Automatic lock insertion in concurrent programs. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 16–23. IEEE, 2012.

[151] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. Jayhorn: A framework for verifying java programs. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 352–358, Cham, 2016. Springer International Publishing.

[152] Terence Kelly, Yin Wang, Stéphane Lafortune, and Scott Mahlke. Eliminating concurrency bugs with control engineering. *Computer*, 42(12), 2009.

[153] Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 165–176. ACM, 2015.

[154] James C. King. Symbolic execution and program testing. *CACM*, 19:385–394, 1976.

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

[156] Shuvendu K Lahiri and Shaz Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *CADE*, pages 214–229. Springer, 2009.

[157] Akash Lal and Shaz Qadeer. A program transformation for faster goal-directed search. In *FMCAD*, pages 147–154. IEEE Computer Society, 2014.

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

[159] James R. Larus and Ravi Rajwar. *Transactional Memory.* Morgan & Claypool, 2007.

[160] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[161] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289. ACM, 2007.

[162] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley, 2nd edition, 1999.

[163] K. Rustan M. Leino. Efficient weakest preconditions. *IPL*, 93:281–288, 2005.

[164] Boyang Li, Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Mooly Sagiv. Synthesis of circular compositional program proofs via abduction. In *TACAS*, volume 7795 of *LNCS*, pages 370–384. Springer, 2013.

[165] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32. ACM, 2013.

[166] Yuanbo Li, Qirun Zhang, and Thomas Reps. Fast graph simplification for interleaved dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 780–793, New York, NY, USA, 2020. Association for Computing Machinery.

[167] 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. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[168] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *OOPSLA*, pages 133–146. ACM, 2012.

[169] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. In *PLDI*, pages 294–304. ACM, 2014.

[170] Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. Language-theoretic abstraction refinement. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, pages 362–376, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[171] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 553–563. ACM, 2009.

[172] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *SAS*, volume 6887 of *LNCS*, pages 95–111. Springer, 2011.

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

[174] Michael Martin, Benjamin Livshits, and Monica S. Lam. 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, pages 365–383, New York, NY, USA, 2005. ACM.

[175] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL '06: Proc.*

33rd symposium on Principles of Programming Languages, pages 346–358. ACM Press, 2006.

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

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

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

[179] Lynette I. Millett and Tim Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *STTT*, 2:343–349, 2000.

[180] Antoine Miné. *Weakly Relational Numerical Abstract Domains. (Domaines Numériques Abstraits Faiblement Relationnels).* PhD thesis, École Polytechnique, Palaiseau, France, 2004.

[181] Yannick Moy. Sufficient preconditions for modular assertion checking. In *VMCAI*, volume 4905 of *LNCS*, pages 188–202. Springer, 2008.

[182] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and repro-

ducing heisenbugs in concurrent programs. In *OSDI*, volume 8, pages 267–280, 2008.

[183] Mayur Naik, Alex Aiken, and John Whaley. *Effective static race detection for Java*, volume 41. ACM, 2006.

[184] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from tests. In *POPL*, pages 373–386. ACM, 2012.

[185] Aditya V. Nori, Sriram K. Rajamani, Saideep Tetali, and Aditya V. Thakur. The YOGI project: Software property checking via static analysis and testing. In *TACAS*, volume 5505 of *LNCS*, pages 178–181. Springer, 2009.

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

[187] Amir Pnueli and Micha Sharir. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

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

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

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

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

[192] Noam Rinetzky and Sharon Shoham. Property directed abstract interpretation. In *VMCAI*, volume 9583 of *LNCS*, pages 104–123. Springer, 2016.

[193] Koushik Sen, Haruto Tanno, Xiaojing Zhang, and Takashi Hoshino. GuideSE: Annotations for guiding concolic testing. In *AST*, pages 23–27. IEEE Computer Society, 2015.

[194] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, volume 6806 of *LNCS*, pages 703–719. Springer, 2011.

[195] Aleksey Shipilev, Sergey Kuksenko, Anders Astrand, Staffan Freiberg, and Henrik Loef. OpenJDK: JMH.

[196] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 387–400. ACM Press, January 2012.

[197] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL):48:1–48:29, January 2019.

[198] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *ACM SIGPLAN Notices*, volume 41, pages 387–400. ACM, 2006.

[199] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 112–122, New York, NY, USA, 2007. Association for Computing Machinery.

[200] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *ACM SIGPLAN Notices*, volume 40, pages 59–76. ACM, 2005.

[201] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on*

Software Engineering, (1):157–171, 1986.

[202] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 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, page 83–95, New York, NY, USA, 2015. Association for Computing Machinery.

[203] Aditya Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas Reps. Directed proof generation for machine code. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 288–305, Berlin, Heidelberg, 2010. Springer-Verlag.

[204] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3:121–189, 1995.

[205] S. L. Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 161–170, July 2007.

[206] Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. On multi-stack visibly pushdown languages. 2013.

[207] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization frame-

work. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[208] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 327–338, New York, NY, USA, 2010. ACM.

[209] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The theory of deadlock avoidance via discrete control. In *ACM SIGPLAN Notices*, volume 44, pages 252–263. ACM, 2009.

[210] Mark Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE Computer Society, 1981.

[211] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.

[212] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 2012.

[213] Valentin Wüstholz. *Partial Verification Results.* PhD thesis, ETH Zurich, Switzerland, 2015.

[214] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *OSDI*, volume 10, pages 163–176, 2010.

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

[216] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *POPL*, pages 221–234. ACM, 2008.

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

[218] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *ACM SIGPLAN Notices*, volume 48, pages 435–446. ACM, 2013.

[219] Qirun Zhang and Zhendong Su. 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, page 344–358, New York, NY, USA, 2017. Association for Computing Machinery.

[220] Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, pages 365–376. ACM, 2013.

[221] Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *APLAS*, volume 8301 of *LNCS*, pages 290–306. Springer, 2013.

[222] Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *Asian Symposium on Programming Languages and Systems*, pages 290–306. Springer, 2013.