ABSTRACT
A modular program analysis considers components independently and provides a succinct summary for each component, which is used when checking the rest of the system. Consider a system consisting of a library and a client. A temporal summary, or interface, of the library specifies legal sequences of library calls. The interface is safe if no call sequence violates the library’s internal invariants; the interface is permissive if it contains every such sequence. Modular program analysis requires full interfaces, which are both safe and permissive: the client does not cause errors in the library if and only if it makes only sequences of library calls that are allowed by the full interface of the library.

Previous interface-based methods have focused on safe interfaces, which may be too restrictive and thus reject good clients. We present an algorithm for automatically synthesizing software interfaces that are both safe and permissive. The algorithm generates interfaces as graphs whose vertices are labeled with predicates over the library’s internal state, and whose edges are labeled with library calls. The interface state is refined incrementally until the full interface is constructed. In other words, the algorithm automatically synthesizes a typestate system for the library, against which any client can be checked for compatibility. We present an implementation of the algorithm which is based on the Blast model checker, and we evaluate some case studies.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification.
General Terms: Languages, Verification, Reliability.
Keywords: Modular verification, software interfaces.

1. INTRODUCTION
We present an algorithm that automatically summarizes the legal uses of a library of functions. The library has a state, and each function call changes that state. Some sequences of function calls, however, may violate the library’s internal invariants, and thus lead to an error state. In full program analysis, one puts the library together with a client and checks whether the client causes the library to enter an error state. In modular program analysis, the check is decomposed into two independent parts. First, independent of the client, we construct from the library source a summary of the legal uses of the library, namely, an interface. Second, independent of the library source, we check if a given client uses the library in a way allowed by the interface.

Consider the example of a library that implements shared memory by providing acq (acquire lock), read (read state), and rel (release lock) functions. The library enters an error state if read is called before acq, or after a call of rel and before a subsequent call of acq. The library interface is a language L over the alphabet {acq, read, rel}, where each word represents a sequence of library calls. The interface L is safe if no word in L can cause the library to enter an error state. For example, the regular set (acq; read; rel) is a safe interface. However, for modular program analysis, we need interfaces that are not only safe but also maximal (note, for example, that the empty interface L = ∅ is always safe but not useful). An interface is permissive if it contains all call sequences that cannot lead to an error state. The safe and permissive language is called the full interface of the library. In our example, the full interface is (acq; read; rel)∗. Modular program analysis requires full interfaces; otherwise, a perfectly safe client may fail to conform to an interface that is not permissive.

Since interfaces are languages, they are finitely witnessed by state machines (acceptors). For example, the interface (acq; read; rel)∗ can be witnessed by an automaton with two states. An interface witness is a finite automaton whose transitions are labeled with library calls. We require the automaton is complete, i.e., all calls are enabled in each state, but some of the calls may lead to a rejecting sink state. At the heart of our technique lies the observation that each state of an interface witness can be given a typestate interpretation [7]: a witness state q corresponds to a set of internal library states, namely, those states that the library can be in after a client has executed a call sequence leading up to q. In the above example, the two accepting states of the automaton witnessing (acq; read; rel)∗ correspond, respectively, to library states where the lock is held, and to library states where the lock is not held.

We define two abstract typestate interpretations for witnesses, where each witness state corresponds to an abstract state of the library. The existence of a safe interpretation guarantees that the witnessed interface is safe, and the existence of a permissive interpretation guarantees that the witnessed interface is permissive. Hence, the problem of...
finding a safe and permissive interface reduces to the problem of finding a witness with both safe and permissive interpretations. In the first step (S1) of our algorithm, we use a given safety abstraction (initially a trivial seed abstraction) to build an abstract interpretation of the library’s internal states. We obtain a candidate witness by treating the library’s initial abstract state as the initial witness state, and treating the abstract states that contain some error state as rejecting witness states; all other abstract library states correspond to accepting witness states. The soundness of the abstraction guarantees that the witness accepts only call sequences that do not lead to an error state. However, overapproximation may cause the witness to prohibit some sequences that cause no error. In other words, the construction guarantees that the witness is safe, at the possible expense of permissiveness.

The second insight is that given a particular witness, we can verify its permissiveness by checking that, if the witness itself is used as client, then it is not possible that the witness enters a rejecting state without the library entering an error state. In this way, we turn the question of checking permissiveness into a reachability question. This leads to the second step (S2) of our algorithm, where we use a permissiveness abstraction to perform an abstract reachability analysis to verify the permissiveness of the witness created in step S1. If the check succeeds, then the automaton witnesses indeed the full interface of the library, and the algorithm returns with success. As a by-product, we obtain both a safe and a permissive typestate interpretation of the witness which demonstrate the safety and the permissiveness of the synthesized interface.

If the permissiveness check fails, then there is a permissiveness counterexample, i.e., a path of the abstract library which follows a call sequence rejected by the witness but does not lead to an error in the library. There are two cases: either the permissiveness counterexample is spurious, meaning that the given abstract path does not concretely lead to a legal library state, but the analysis is misled into believing otherwise due to the imprecision of the permissiveness abstraction; or the permissiveness counterexample is genuine, meaning that it corresponds to a concrete library path that leads to a legal library state, and thus the corresponding call sequence must be contained in the full interface. The third step (S3) of our algorithm considers both cases. In the first case, we automatically refine the permissiveness abstraction to eliminate the spurious counterexample. In the second case, the legal call sequence was conservatively prohibited owing to the imprecision of the safety abstraction, and we automatically refine the safety abstraction so as to include this call sequence.

Our abstractions are predicate abstractions, and hence in either case, the refinement procedure finds new predicates about the internal library state. In the first case, the new predicates show that certain call sequences lead to error states and hence must be rejected; in the second case, the new predicates show that certain call sequences do not lead to error states and hence must be accepted. We now repeat steps S1 and S2 until we have a safety and a permissiveness abstraction which are precise enough to create a witness that is both safe and permissive. For libraries with finite internal state, the algorithm is guaranteed to terminate with success. We could use a single abstraction of the library for both steps S1 and S2. However, the goals of the two steps are orthogonal, namely proving safety and proving permissiveness, and therefore the use of different predicates often allows more parsimonious abstractions. In other words, two different permissiveness abstractions may be relevant to prove the safety and the permissiveness of an automaton that witnesses the full interface of the library.

We have implemented the synthesis of witnesses for safe and permissive interfaces by extending the Blast model checker, which is based on automatic predicate abstraction refinement [12]. Our tool successfully synthesizes the full interfaces for several classes from JDK1.4, including Socket, Signature, ServerTableEntry, and ListItr. Once the interface witness is constructed, typestate analyses such as [8, 4] can be used to perform the task of checking that a client conforms to the synthesized witness.

Interfaces, in the sense presented here, have been used by many researchers [13, 8, 5, 14, 1, 9, 7]. However, all these approaches either assume that the interface is specified by the programmer [13, 8, 5, 7]; or they use a set of dynamic executions of the library to define its interface [2, 14], with the result that the constructed interface may be unsafe or not permissive; or they perform static analyses that are not precise enough to create permissive interfaces [9, 14, 1]. The closest related work is [1], which uses learning techniques along with predicate abstraction to find interfaces. Unlike our algorithm, they require the user to provide appropriate sets of predicates, and the resulting interfaces are not guaranteed to be permissive. By focusing on the automatic generation of interfaces that are both safe and permissive, we enable modular program analysis.

2. PERMISSIVE INTERFACES

2.1 Open Programs

An open program represents a set of library functions that can be used by clients.

Syntax. For a set $X$ of variables, $\text{Exp}.X$ is the set of arithmetic expressions over the variables $X$, and $\text{Pred}.X$ is the set of boolean expressions (arithmetic comparisons) over $X$, the set $\text{V}.X$ is the set of valuations to $X$, and $\text{Op}.X$ is the set of operations containing: (1) assignments $x := e$, where $x \in X$ and $e \in \text{Exp}.X$, (2) assume predicates $\text{assume}(p)$, where $p \in \text{Pred}.X$, representing a condition that must be true for the operation to be executed, and (3) function calls $f(x)$, where $x \in X$ and $f$ is a function.

A control-flow automaton (CFA) $C = (X_L, X_S, Q, q_0, q_e, \rightarrow)$ comprises (1) two disjoint sets of variables $X_L$ (local variables) and $X_S$ (static variables), (2) a set $Q$ of control locations, with an initial location $q_0 \in Q$ and a final location $q_e \in Q$, (3) a finite set of directed edges labeled with operations $\rightarrow \subseteq Q \times \text{Op}.(X_L \cup X_S) \times Q$. Let $X = X_L \cup X_S$.

An open program $P = (X, F, \text{Outs}, s_0, \Sigma, \Sigma)$ has (1) a set $X$ of typed static variables (integer- or boolean-valued), including a special output variable $\text{out}$, (2) a set $F$ of functions, where each function $f \in F$ is represented as a CFA $f.C$ with static variables $X$, and every function call operation in $f.C$ is a member of $F$, (3) a set $\text{Outs}$ of outputs, (4) an initial state $s_0 \in V.X$, (5) a set $\Sigma \subseteq V.X$ of error states given as a predicate in $\text{Pred}.X$, and, (6) a signature $\Sigma$, which is a subset of $F \times \text{Outs}$. The signature $\Sigma$ represents the externally visible function names and output values.
open program is finite-state if all variables range over the booleans.

**Example 1:** Figure 1(A) shows an open program $P_1$ with the two static variables $P_1.X = \{a, e\}$. The latter is used to define the error states $P_1.\mathcal{E} = \{e \neq 0\}$. The functions are `acq.read`, `rel`, `m.new`, and `m.rd`. There is only one output (unit), which we omit for clarity. The signature functions in $P_1.\Sigma$ are `acq.read`, and `rel`. The functions `m.new` and `m.rd` are internal. Figure 1(A) shows the code for the three signature functions, and CFAs for `read` and `m.new`, which returns a non-zero value. In the CFAs, edges are labeled with function calls, basic blocks of assignments (indicated by boxes), and assume predicates (indicated by brackets). In the initial state, both $a$ and $e$ are 0.

**Semantics.** An $X$-state is an element of $V.X$. For disjoint sets $X$ and $Y$ of variables, if $s \in V.X$ and $t \in V.Y$, we write $s \circ t \in V.(X \cup Y)$ for the $X \cup Y$-state obtained by combining $s$ and $t$. For $X_1.X_2$-states $s_1, s_2$, we say that $s_1 \approx s_2$ if for all $x \in X_1 \cap X_2$, the states agree: $s_1.x = s_2.x$. A set of $X$-states $r$ is called a data region. A predicate over $X$ represents a data region comprising all valuations that satisfy the predicate.

The transition relation for a CFA $C$ written $\mathcal{C} \subseteq (V.X)^2$ is defined as follows: $s \xrightarrow{C} s'$ if there exists a sequence $(q_1, (s_1 \circ t_1)), \ldots, (q_n, (s_n \circ t_n))$ such that $(q_0, s) = (q_1, s_1)$ and $(q_n, s') = (q_0, s_n)$ and for each $1 \leq i \leq n - 1$, we have $q_i \xrightarrow{C} q_{i+1}$ in $C.\mathcal{E}$ and $s_i \circ t_i \xrightarrow{C} s_{i+1} \circ t_{i+1}$, and each $\mathcal{C}_i \subseteq (V.(C.X))^2$ is as defined below. We say that $s \xrightarrow{C} s'$ for $s, s' \in V.X$: if (1) $op$ is `acq`, then $s \models p$ and $s'.y = s.y$ for all $y \in X$; and (2) if $op$ is `read`, then $s'.x = s.x$ and $s'.y = s.y$ for all $y \in X \setminus \{x\}$; and (3) if $op$ is `rel`, then $s.y = s'.y$ for all $y \in X_L$, and there exist $t, t' \in V.X$ such that $t \xrightarrow{C} t'$ and $s \approx t$ and $s' \approx t'$. Parameter passing and return values can be mimicked with static variables. We lift the transition relation to sets of states. For a set $r$ of $X$-states, the strongest postcondition of $r$ w.r.t. $op$, written $sp.r.op$, is the set $\{s' \in V.X \mid \exists r : s \xrightarrow{C} s'\}$. The weakest precondition of $r$ w.r.t. $op$, written $wp.r.op$ is the set $\{s \in V.X \mid \exists s' : r : s \xrightarrow{C} s'\}$. We generalize $sp$ and $wp$ to sequences of operations in the standard way. For assign and assume statements, and $r$ given by a predicate in $Pred.X$, the above can be defined as predicate transformers $[6]$. An open program induces a state space $V.X$. Each $(f, o) \in \Sigma$ has a transition relation $\mathcal{C}(f, o) \subseteq (V.X)^2$ defined as: $s \xrightarrow{C(f, o)} s'$ if $s \xrightarrow{C} s'$ and $s'.out = o$. We extend this operation to data regions by defining $Post.(f, o) = \{s' \mid \exists r : s \xrightarrow{C(f, o)} s'\}$ and its dual, $Pre.(f, o) = \{s \mid \exists s' : r : s \xrightarrow{C(f, o)} s'\}$.

**Example 2:** In the open program $P_1$ of Figure 1(A), the initial states are given by $a = 0 \land e = 0$, and the error states $P_1.\mathcal{E}$ by $e \neq 0$. The strongest postcondition $sp.(a := m.new())$ of the region $r = (a = 0)$ is $a \neq 0$, and $Post.(r.acq())$ is also $a \neq 0$.

**2.2 Interfaces**

A set $r$ of states is safe w.r.t. $E$ if $r \subseteq \mathcal{E}$ for an open program $P$, a sequence $\sigma \in P.\Sigma^*$ is $E$-safe from a set of states if either the sequence $\sigma$ is the empty sequence, or $\sigma \equiv (f, o) \cdot \sigma'$ and $r' = Post.(f, o)$ is such that (1) $r'$ is safe w.r.t. $E$ and (2) $\sigma'$ is $E$-safe from $r'$. An open program $P$ is **visibly deterministic** if for all $\sigma \in P.\Sigma^*$, it is not the case that $\sigma$ is both (1) not $E$-safe from $\{P.s_0\}$, and (2) not $\neg E$-safe from $\{P.s_0\}$. In the sequel, we shall only consider visibly deterministic open programs. A sequence is **legal** if it is $E$-safe from $\{P.s_0\}$, that is, by executing the sequence of calls of the open program does not get into an error state. We write $I.P$ for the set of all legal sequences. A sequence $\sigma \in P.\Sigma^*$ is **realizable** from a set $r$ of states if either the sequence $\sigma$ is the empty sequence, or $\sigma \equiv (f, o) \cdot \sigma'$ and $r' = Post.(f, o)$ is such that (1) $r'$ is not empty and (2) $\sigma'$ is realizable from $r'$. A sequence is **realizable** if it is realizable from $P.s_0$. We write $R.P$ for the set of all realizable sequences.

**Definition 1. [Interfaces]** An interface for the open program $P$ is a prefix-closed language over the signature $P.\Sigma$. An interface $I$ for $P$ is:

1. A safe interface if every sequence in it is legal, i.e., $I \subseteq T.P$.
2. A permissive interface if it contains every legal realizable sequence, i.e., $I.P \cap R.P \subseteq I$.
3. The full interface if it is the set of all legal sequences, i.e., $I = T.P$.

**Example 3:** In the open program $P_1$ from Figure 1(A), when a client calls the function `acq`, the state changes, as `acq` in turn calls `m.new` and sets $a$ to the non-zero value returned by the latter. Since $e$ remains 0, the call is legal. Subsequently, a client may call `read` arbitrarily many times. After each exit from `read`, the variable $e$ remains 0, and hence the sequence `acq.read` of calls is legal. However, if `read` is called from the initial state, then, as $a$ is 0, the function sets $e$ to 1 and thus, $P_1$ reaches an error state. Similarly, `rel` can be called arbitrarily many times. However, after a call to `rel`, `acq` must be called again before calling `read`. Hence, the full interface for $P_1$ is the regular language $L_1 = ((acq \cdot read) \cdot rel)^*$. 

---

**Figure 1:** (A) $P_1$ (B) safe interface witness $W_1$ (C) $P_2$ (D) safe and permissive interface witness $W_2$
Consider now the open program $P_2$ obtained by augmenting $P_1$ with the additional static variable $x$, and the functions $\text{acqx}, \text{write},$ and $\text{relx}$ shown in Figure 1(C). The new functions are added to the functions in $P_1.\Sigma$ to get the signature $P_2.\Sigma$ (again, we omit the single output). Note that though $L$ is still a safe interface for $P_2$, it is too constrained, as it prohibits the legal sequence $\text{acqx} \cdot \text{write} \cdot \text{relx}!$ Indeed, after calling $\text{acqx}$, the client may call $\text{read}$ or $\text{write}$ arbitrarily often and the full interface for $P_2$ is the regular language $(\text{acqx} \cdot \text{read}^* \cdot (\text{rel} + \text{relx})^*) + (\text{acqx} \cdot (\text{read} + \text{write})^* \cdot (\text{rel} + \text{relx})^*) + \text{rel} + \text{relx})^*$.  

### 2.3 Witnesses

**Witness graphs.** We focus on interfaces corresponding to regular languages, which are naturally witnessed by finite state graphs. A witness graph $W = (N, E, n_0)$ comprises (1) a set $N$ of nodes, partitioned into two sets $N^+$, the safe nodes, and $N^-$, the unsafe nodes, (2) a set $E \subseteq N \times P.\Sigma \times N$ of directed edges labeled with elements of $P.\Sigma$, and (3) a root node $n_0 \in N$. We write $n \rightarrow (f, o, n')$ if $(n, (f, o), n') \in E$, and call $n'$ an $(f, o)$-successor of $n$. Additionally, we require that every $n \in N$ has exactly one $(f, o)$-successor for each $(f, o) \in \Sigma$. Intuitively, the interface corresponding to a witness graph is the language accepted by the DFA obtained by considering the safe nodes as accepting and deleting all unsafe nodes. Formally, a sequence $\sigma \in P.\Sigma^*$ is accepted from a node $n \in N^+$ if $\sigma$ is the empty sequence, or $\sigma \equiv (f, o) \cdot \sigma'$ and $n$ has an $(f, o)$-successor $n' \in N^+$ such that $\sigma'$ is accepted from $n'$. No sequence is accepted from any node in $N^-$. The language of a witness graph $L.W$ is the set of all $\sigma$ accepted from $n_0$.

**Example 4:** Figure 1(B) shows a witness graph $W_1$ for $P_1$ (ignoring the "x" and $\text{write}$ labels). The unshaded and shaded nodes are respectively $N^+$ and $N^-$. Note that $L.W_1 = ((\text{acqx} \cdot \text{read}^* \cdot \text{rel}^*)^* \cdot \text{rel}^*)^*$, the full interface of $P_1$.

**Region labels.** To reason about the properties of the languages of the witness graphs, we relate the nodes of the witness graphs with the states of the open program using region labeling functions. A function $\rho : N \rightarrow \text{Pred}(P.X)$ mapping the nodes of a candidate graph $G$ to predicates is a region labeling if (P1): $P.\rho \subseteq P.\rho_0$, i.e., the root node contains the initial state; and (P2) for every $n \rightarrow (f, o) \cdot n'$, we have $\text{Post}(\rho.n).f \cdot o \subseteq \rho.n'$. A region labeling represents an overapproximation of the behavior of an open program: for every node $n \in N^+$, and every sequence of operations $\sigma \in P.\Sigma^*$ marking a path from the root $n_0$ to $n$ in $G$, we have $\text{Post}(\rho.n_0).\sigma \subseteq \rho.n$.

**Proposition 1. [Safe labels]** For an open program $P$ and witness graph $W$, if there is a region labeling $\rho$ such that (P3) for each $n \in N^+$ we have $\rho.n \subseteq \sim P.E$, then $L.W \subseteq I.P$.

The existence of a safe labeling for $W$, namely one that satisfies (P3), guarantees that the language of $W$ is bounded above by $I.P$, i.e., the witness contains only sequences that are permitted by $P$.

**Example 5:** Consider the witness $W_1$ for $P_1$ shown in Figure 1(B), ignoring the $\text{acqx}, \text{relx}, \text{write}$ edge labels. The label for each node is written in the box next to the node; for the safe (unshaded) nodes there is the implicit conjunct $e = 0$, and for the unsafe (shaded) nodes, the label is $e = 1$. We can check that this is a region labeling; e.g., for the edge $n_0 \rightarrow n_1$, we saw that $\text{Post}(a = 0).\text{acqx}$ equals $a \neq 0$, i.e., the successor states from the label of $n_0$ are contained in the label of $n_1$. This labeling is safe, and thus demonstrates that the language of $W_1$ is a safe interface for $P_1$, and indeed this language is the full interface for $P_1$, but this is not always the case. Consider the same witness graph $W_1$ for $P_2$, now with all the edge labels. The labeling described above is a safe labeling for this witness, and hence the witness’ language is a safe interface for $P_3$. However, this interface is too restrictive: it prohibits the client from ever calling $\text{write}$, as the $\text{write}$ edge leads into an unsafe (shaded) state. □

**Proposition 2. [Permissive labels]** For an open program $P$ and witness graph $W$, if there exists a region labeling $\rho$ such that (P4) for each $n \in N^-$ we have $\rho.n \subseteq P.E$, then $I.P \cap \sim P \subseteq L.W$.

The existence of a permissive labeling for $W$, namely one the satisfies (P4), guarantees that the language of $W$ is bounded below by $I.P \cap \sim P$, i.e., the witness contains every realizable sequence that is permitted by $P$.

**Example 6:** The labeling described earlier for the witness $W_1$ shown in Figure 1(B) for $P_1$ is also a permissive labeling, and hence $L.W_1$ is a permissive interface for $P_1$. However, it can be shown that for $P_2$, the augmented version of $W_1$ shown in Figure 1(B) has no permissive labeling. Consider the witness graph $W_2$ in Figure 1(D). It has the safe labeling $[n_0 \rightarrow (a = 0); n_1 \rightarrow a \neq 0; n_2 \rightarrow (a \neq 0 \land x \neq 0)]$, where again $e = 0$ and $e = 1$ are implicit for the unshaded and shaded nodes, respectively. It also has the permissive labeling $[n_0 \rightarrow (a = 0 \land x = 0); n_1 \rightarrow (a \neq 0 \land x = 0); n_2 \rightarrow a \neq 0]$. Thus, the language of $W_2$ is a permissive interface for $P_2$. The labels shown in Figure 1(D) are the conjunction of the labels described above and are both safe and permissive. □

The above examples demonstrated that in order for a witness graph’s language to be a permissive interface, not only must there exist a safe region labeling, which proves that the witness allows only safe legal sequences, but there must also exist a permissive labeling, which guarantees that the witness allows every realizable behavior. Hence, our strategy to construct permissive interfaces is to compute a witness graph that has a safe labeling as well as a permissive labeling. Further, as the example showed, the two region labelings may be different, and in general may be constructed using orthogonal abstractions of the state space.

### 3. CONSTRUCTING INTERFACES

We now describe our technique for constructing witness graphs that have safe and permissive labelings. The method comprises three main ingredients.

1. **Witness Checking.** First, given a witness graph, and two sufficiently precise abstractions, a safety abstraction and a permissiveness abstraction, we show how to check whether there exists safe and permissive labelings for the given graph.

2. **Witness Reconstruction.** Second, given a safety and a permissiveness abstraction, we show how we can construct a witness graph such that if the two abstractions are sufficiently precise, the constructed witness is safe and permissive.

3. **Witness Inference.** Finally, given just two arbitrarily coarse abstractions, we show how we can iteratively
refine them to obtain abstractions precise enough to construct a safe and permissive witness graph.

For witness checking, we convert the given witness into a witness client that exercises the open program in the manner prescribed by the witness. The nodes of the witness graph correspond to program locations of the witness client. The witness client is at a safe (resp. unsafe) location whenever it has executed a sequence of calls allowed (resp. disallowed) by the witness graph. The witness is safe iff whenever the witness client is in a safe location, the sequence of calls made to reach that location is indeed legal, i.e., the open program P is in a legal state (~P\~E). This is a standard safety verification question that can be answered via abstract reachability using the safety abstraction. Dually, the witness is permissive iff whenever the witness client is at an unsafe location, the call sequence made to reach that location is indeed illegal, i.e., the open program is in an unsafe state (P\~E). Hence, we can also convert the permissiveness check into a safety verification problem, which can be solved using the permissiveness abstraction.

For witness reconstruction, we construct a maximal client that generates all possible call sequences. We use the given safety abstraction to compute an overapproximation of the behaviors of P as an abstract reachability graph. From this we obtain a safe candidate witness by treating nodes that interject P\~E as unsafe nodes, and the rest as safe nodes. Next, using the method outlined above, together with the supplied permissiveness abstraction, we verify that this reconstructed candidate is permissive. If so, we are done.

For witness inference, we obtain sufficient abstractions, and through them a safe and permissive witness, via the following loop. First, we construct a candidate witness with the current abstraction using the algorithm for witness reconstruction. Second, we check if this candidate is a safe and permissive witness using the algorithm for witness checking. Third, if witness checking fails, we use the failure to find new predicates that refine the current abstractions, and repeat the loop with the refined abstraction.

3.1 Witness Checking

Given an open program P and a witness graph W for P, the witness checking problem is to find whether or not the witness’ language is a safe and permissive interface for P. For this check we employ a client that exercises the open program in the manner prescribed by W.

Clients. A client for an open program P is a CFA CI = (X, \emptyset, Q, q_0, q_e, \rightarrow) where the operations on the edges are assignments and assumes from before, as well as function calls y := f(), where (f,) \in P\Sigma and y \in X. For convenience, we introduce the operation y := (f, o) as short-hand for the sequence of operations y := f(); assume [y = o], where y is not subsequently read; we further compress this to (f, o).

Closed programs. A closed program (CI, P) consists of a client CI and an open program P. A closed program (CI, P) induces a state space V.(CI.X \uplus P.X). Let s, s' \in V.(CI.X) and t, t' \in V.(P.X). Then (s, t) \rightarrow (s', t') if (1) s \rightarrow s' and t = t' if op is an assignment or assume, and (2) there exists o \in P.O such that t' \rightarrow o and s = s[op] if op is a function call y := f(). For a subset L \subseteq Q of the client locations, we say that (s, t) \in V.(CI.X \uplus P.X) is L-reachable if there exists a sequence (q_0, (s_0 \circ t_0)), \ldots, (q_n, (s_n \circ t_n)) of pairs in CI.Q \times V.(CI.X \uplus P.X) such that (1) t_0 = P.s_0 (the initial state of the open program), (2) for all 0 \leq i \leq n - 1, there is an edge q_i \rightarrow q_{i+1} in CI such that (s_i \circ t_i) \rightarrow (s_{i+1} \circ t_{i+1}), and (3) s_n = s, t_n = t, and q_n \in L.

Witness clients. For every witness graph W = (V, E, v_0) for P, we can construct a witness client CFA Client.W = ((x), \emptyset, V \cup \{pc_e\}, v_0, pc_e, \rightarrow) as follows. The client has a single variable x, which is used to capture the output from function calls. For every edge v_0 \rightarrow s_0 = (f, o) in W, there is a corresponding CFA edge v_0 \rightarrow s_0' in Client.W, where op = x := (f), o. For example, Figure 2(A) shows a witness client corresponding to the witness graph W_1 of Figure 1(B). The witness client can call any sequence of library functions that are allowed by the witness graph.

Interface checking via safety verification. The language of the witness W is a safe interface for P iff the V^+ reachable states of (Client.W, P) are P^\~E-safe, i.e., the client W never reaches a state in P\~E when it is at a safe node. The language of W is a permissive interface for P iff the V^-reachable states of (Client.W, P) are P^-\~E-safe, i.e., the client W never reaches a state in \~P\~E when it is at an unsafe node. As the above reachability checks are undecidable in general, we need abstractions of the open program with which to overapproximate the reachable states. We use predicate abstraction, but any abstract domain for which fixpoints are computable can be used.

Predicate abstraction. For a set \Pi \subseteq Pred.X of predicates and a formula r over X, let Abs.\Pi.r denote the smallest (in the inclusion order) data region containing r expressible as a boolean formula over atomic predicates from \Pi. For example, if \Pi = \{a = 0, b = 0\} and r = (a = 3 \land b = a + 1), then the predicate abstraction of r w.r.t. \Pi is \neg(a = 0) \land \neg(b = 0). The abstract postcondition of r and op w.r.t. \Pi, written SP.\Pi.r, is a boolean combination of predicates from \Pi which overapproximates sp.r.op. The procedure SP computes the abstract postcondition. It takes as input the set \Pi of predicates, an operation op, and the current region r, and returns SP.\Pi.r. For assignments and assumes it directly computes the abstract postcondition [10], but for calls into functions, which may contain loops, the procedure implements a fixpoint computation via a standard abstract reachability algorithm [10, 12].

Abstract reachability graphs. Given an open program P, client CI, and set of predicates \Pi, an abstract reachability graph (ARG) for (CI, P) w.r.t. \Pi is a rooted directed graph where each node is labeled by pairs (pc, r) such that: (1) The root node \mathbf{n}_0 is labeled (pc_0, r_0), where pc_0 is the initial location of CI, and r_0 is the predicate abstraction of the
Algorithm 2 Check

Input: open program $P$, witness $W = (V, E, v_0)$. 
Input: predicates $P$, vertices $V' \subseteq V$, target region $E$. 

Output: TRUE or a counterexample CFD Ctx$\delta$.

1: $A := \text{BuildARG}(P, (\text{Client}, W)).$  
2: $\rho := \lambda r. (V_{\text{a}}(v, r), r)$  
3: if exists $v \in V'$ s.t. $\rho v \not\in E$ then  
4: find $n : (v, r) \in A$ s.t. $r \not\in E$  
5: $\sigma := \text{path from } n_0 \to n$ in $A$  
6: return $\text{dag}_{\text{a}}(\rho, \sigma)$  
7: else  
8: return TRUE

Algorithm 3 ReconstructMax

Input: open program $P$, predicates $P$. 

Output: maximally safe witness graph $W$ for $P$.

1: $A := \text{BuildARG}(mxc(P, P)).$  
2: $V^+ := \{ v \mid (r, r) \in A \land s.t. r \not\subseteq \sim E \}.$  
3: $V^- := A \setminus V^+.$  
4: return witness graph $(V^+ \cup V^-, A, E, A_{n_0})$

Proposition 4. [Witness checking] For an open program $P$, let $W = (V', E, v_0)$ be a witness graph such that for two predicate sets $\Pi_1$ and $\Pi_2$, both Check-$P.W.\Pi_1.V^+.(\sim E)$ and Check-$P.W.\Pi_2.V^+.(\sim E)$ return TRUE. Then $I.P \cap R.P \subseteq \mathcal{E}, W \subseteq I.P$, i.e., $W$ is a safe and permissive witness for $P$.

Example 8 below shows that the abstractions required to demonstrate the two requirements may be quite different. As the running time of BuildARG is exponential in the number of predicates, we keep two separate abstractions.

Example 8: Consider an open program with the variables $x, y$, and $e$, all initially 0, and the two methods $f_1$ and $f_2$:

\begin{align*}
  f_1 : & \text{if } x \neq 0 \text{ then } e := 1; \\
  f_2 : & \text{if } y = 0 \text{ then } e := 1;
\end{align*}

The error states are $e \neq 0$; we omit the output. Consider the witness graph $W = \{ (n_0, n_0), (n_0, f_1, n_0), (n_0, f_2, n_1), n_0 \}$, with $N^+ = \{ n_0 \}$ and $N^- = \{ n_1 \}$. The labeling $\rho_{n_0} = (x = 0 \land e = 0)$ and $\rho_{n_1} = True$ is safe. Since $n_1 \in N^-$ is unsafe, the call $f_2$ is not allowed. The labeling $\rho'_{n_0} = (y = 0 \land e = 0)$ and $\rho'_{n_1} = (e \neq 0)$ is permissive. The safe (resp. permissive) labeling does not track $y = 0$ (resp. $x = 0$).

3.2 Witness Reconstruction

In witness checking, we assumed that in addition to the abstraction predicates $\Pi_\text{p}$ and $\Pi_\text{w}$, we had a given candidate witness $W$. In witness reconstruction, we show how to use $\Pi_\text{p}$ to reconstruct a candidate witness $W$ with a safe region labeling. As before, if we can show (using $\Pi_\text{p}$) that the candidate $W$ has a permissive labeling, then we are done.

Maximal clients. To obtain the candidate witness, we close the open program $P$ using a maximal client that generates all possible sequences of function calls to the library. For an open program $P$, the maximal client $\text{mxc}(P)$ of $P$ is the CFA $(\{ s \}, 0, \{ pc_0 \}, \{ pc_0, pc_0 \})$, where for every $(f, o) \in \Sigma$ we have that $(pc_0, x := (f)(o), pc_0) \notin \sim$. We use the abstraction $\Pi_\text{a}$ to overapproximate the behaviors of $P$ when exercised by this client. Our candidate witness corresponds to the language of the “safe” sequences generated by the maximal client.

Example 9: The maximal client of $P_2$ from Figure 1(C) is $\text{mxc}(P_2)$ shown on the top in Figure 2(A).

The Witness Reconstruction Algorithm. Consider the ARG $A$ for $(\text{mxc}(P, P))$ w.r.t. the predicates $\Pi_\text{p}$. We can convert this ARG into a witness graph $\text{Witness}.A = (V, E, v_0)$ by converting (1) the nodes of the ARG into nodes of $\text{Witness}.G$, (2) the edges of the ARG, which are labeled by operations in $P.\Sigma$ into edges of $\text{Witness}.G$, (3) the root node of the ARG into the root of $\text{Witness}.G$, and (4) letting $V^+$ be the set $\{ n \mid n : (pc, r) \text{ and } r \subseteq \sim \}$, and $V^-$ its complement. As the witness corresponds to an ARG, the ARG node labels form a region labeling. The nodes are partitioned so that the behaviors of the program contain a desirable (permitted) set.
as to make the labeling a safe region labeling. In addition, it can be proved that the witness “reconstructed” in this way is the largest witness that can be shown to be safe using the abstraction $\Pi_S$. The above algorithm is formalized by the procedure $\text{ReconstructMax}$ shown in Algorithm 3.

**Proposition 5. [Maximal witness reconstruction]**

For every open program $P$ and set $\Pi$ of predicates, the procedure $\text{ReconstructMax}(P, \Pi)$ terminates and returns a witness graph $W$ such that (1) $W$ is a safe witness for $P$, and (2) for every witness $W'$, if $\text{Check}(P.W'.\Pi(W.V'))(\neg P.E)$, then $L.W' \subseteq L.W$, i.e., $W$ is the maximal safe witness for $P$ w.r.t. II.

**Example 10:** Upon running the algorithm $\text{BuildARG}$ on $(\text{mxc}.P_2, P_2)$ and the set $\Pi = \{e = 0, a = 0\}$ of predicates, we obtain the ARG $A_1$ shown in Figure 1(B), which translates to the witness $W_1$. The unshaded nodes are those whose regions are contained in $\neg P.E$, i.e., $e \neq 1$; they are safe w.r.t. to $P_2.E$. The label $\text{acq}/x$ indicates two edges, one labeled with $\text{acq}$ and the other with $\text{acq}$. This reconstructed maximal witness $W_1$ prohibits calling $\text{write}$ and hence, while being safe, is not permissive, as $\text{write}$ can be safely called after first calling $\text{acq}$. When computing the $\text{acq}$/successor of $n_1$, the abstract state is the same as that of $n_1$, namely $\neg(e = 0) \wedge (e = 0)$, as there are no predicates on $x$. Hence the algorithm sets $n_1$ to be the $\text{acq}$/successor of $n_0$, thus not permitting any calls to $\text{write}$. While it may seem that this merging of nodes with the same abstract state is premature, this is what guarantees the termination of the abstract reachability loop.

### 3.3 Witness Inference

The above example highlights the importance of finding the right abstractions. In verification, coarse abstractions lead to false positives, for interface synthesis, coarse abstractions lead to massively constrained interfaces. As shown in the prequel, it suffices to find abstractions $\Pi_S$ and $\Pi_P$ such that the maximal (safe) witness for $P$ w.r.t. $\Pi_S$ can be shown to be permissive using $\Pi_P$. We now show how to find such $\Pi_S$ and $\Pi_P$ by automatically refining coarse abstractions, using witness counterexample sequences of $P.E$ which are prohibited by the maximal witness, but which may be permitted by the open program $P$.

**Witness counterexamples.** We use the procedure $\text{BuildARG}$ (via the procedure $\text{Check}$) to see if the abstraction $\Pi_P$ suffices to show that a candidate witness $W$ is a permissive witness for $P$. The permissiveness check fails in line 3 of Check if in the ARG $G$ returned by $\text{BuildARG}$ there exists a node $n : (v, r)$ such that (1) the resulting state of the open program is legal, i.e., $r \subset P.E$; and (2) $v$ is an unsafe witness node, i.e., $v \in V^-$. Consider any path in the ARG $G$ from the root node of $G$ to $n$. The ARG edge labels along this path correspond to a call sequence $\sigma$ that may be legal, as the resulting abstract region is not contained in $P.E$ states, but which is prohibited by $W$, as the sequence ends in an unsafe witness node. Such a call sequence is called a witness counterexample.

**Control-flow dag.** A control-flow dag (CFD) for a function $f$ is a directed acyclic graph that represents a set of paths through the CFA of $f$. The CFD has a single source (single sink) corresponding to the entry (exit) location of $f$. In the procedure $\text{BuildARG}$ (Algorithm 1), we compute $\text{SP}_f.P:op$, where $op = (f, o)$, in the standard way [10, 12] by unrolling the CFA for $f$, computing the abstract state over the predicates $\Pi$ for each unrolled CFA location, and merging nodes with the same abstract state. The returned region $r'$ is the union of the regions of the unrolled exit nodes. On deleting back-edges from the unrolled CFA, and merging all exit nodes, we get a CFD encoding the set of possible paths that a state in $r$ may take through the body of $f$ to reach a state in $r'$. We call this CFD $\text{dag}_{\Pi,op}$. Given a sequence of edges labeled $\sigma$ in the ARG built by $\text{BuildARG}$, we chain together the CFDs for the individual edges to obtain $\text{dag}_{\Pi,\sigma}$, which represents a set of paths that the open program may execute if the sequence of calls $\sigma$ is made. Whenever the check in line 3 of the procedure $\text{Check}$ fails, it returns a CFD $\text{dag}_{\Pi,\sigma}$, corresponding to a witness counterexample $\sigma$.

**Example 11:** Consider the restrictive candidate witness $W_1$ corresponding to the ARG $A_1$ from Example 10. The witness client $\text{Client}.W_1$ is shown in Figure 2(A). An edge with multiple labels stands for several edges, each with one of the labels. We invoke the procedure $\text{Check}$, using the predicates $\Pi_P = \{e = 0, a = 0\}$ and the target states $\neg(e = 1)$, to see if this witness is permissive. To do this, $\text{Check}$ builds an ARG for the closed program, but then finds that the check in line 3 fails, and it finds a node in the ARG with the properties (1) and (2). A path in the resulting ARG leading to this ARG node is shown in Figure 2(B). The node labels are the region labels in the ARG, which are built using the predicates $\Pi_P$. The sequence of calls labeling the edges $\text{acq}$. $\text{write}$ is a witness counterexample, for which Figure 2(C) shows the CFD.

**Predicate refinement.** The visible determinacy of the open program $P$ ensures that the set of feasible paths corresponding to any call sequence either always end in $P.E$, if this call sequence is not permitted, or always end in $\neg P.E$, if

---

**Figure 2:** (A) mxc.$P_2$ (1) Client.$W_1$ (1) (B) witness counterexample (C) counterex. control-flow dag
this call sequence is permitted. In particular, if $\delta = \text{dag}_{\Pi_2} \cdot \sigma$, then the feasible paths corresponding to $\delta$ either always end in $P.E$ or always end in $\neg P.E$. Hence, we have two cases:

(CxPerm) The condition $\neg P.E \land \text{sp}.(P,s_0) \cdot \delta$ is unsatisfiable, implying that $\sigma$ is not a permitted call sequence. In this case, we use a standard predicate discovery algorithm [11] to obtain new predicates $\Pi$ such that $\neg P.E \land \text{sp}.(P,s_0) \cdot \sigma$ becomes unsatisfiable, i.e., the resulting abstraction is precise enough to eliminate the witness counterexample.

(CxSafe) The condition $\neg P.E \land \text{sp}.(P,s_0) \cdot \delta$ is satisfiable, implying that the witness counterexample $\sigma$ is a permitted call sequence that has been prohibited by the maximal safe witness reconstructed from $\Pi_S$. In this case, the above shows that $P.E \land \text{sp}.(P,s_0) \cdot \delta$ must be unsatisfiable, and so the predicate discovery algorithm infers new predicates $\Pi$ such that $P.E \land \text{sp}.(P,s_0) \cdot \sigma$ becomes unsatisfiable, i.e., the resulting abstraction is precise enough that its maximal safe witness contains the permitted call sequence $\sigma$.

The above is made precise by the procedure Refine shown in Algorithm 4. The procedure GetNewPreds refers to the predicate discovery algorithm, which takes as input a CFD $\delta$, and a set $r$ of initial states, and a set $E$ of states such that $\text{sp}.r \cdot \delta \land E$ is unsatisfiable, and returns a set $I$ of predicates such that $\text{sp}.I \cdot \delta \land E$ is unsatisfiable. Refinement in witness inference is different from refinement in safety verification. In safety verification, refinement is done using infeasible paths to error states, and leads to the removal of such paths from the abstraction. In witness inference, witness counterexamples may be feasible or infeasible paths to error states. Even if the counterexample is a feasible path (case CxSafe), the refinement procedure adds new predicates that force the safety abstraction $\Pi_S$ to include this feasible, legal sequence. The role of the refinement in this case is not to remove abstract behaviors, but to introduce additional possible behaviors.

Example 12: Consider the CFD for the witness counterexample of Example 11, shown in Figure 2(C). The feasible paths of this CFD end in safe states, i.e., $e = 0$, though owing to the imprecision of the abstraction, the maximally safe witness built from $\{a = 0, e = 0\}$ prohibits this sequence. Hence, this is the case CxSafe, and the predicate discovery algorithm finds the new predicate $x = 0$. The resulting abstraction is precise enough to permit the sequence $\text{acq} \cdot \text{write}$. Note that our refinement does not add the single witness counterexample sequence; such a process may never terminate. Instead we infer new predicates such that the refined abstraction will contain the earlier prohibited sequence, and as a result we add all other sequences that can be proved to be safe using the refined abstraction.

The predicate refinement method depends on visible determinism. Consider a version $P_2$ of $P$ with the version of $\text{acq}$ shown in Figure 3(A), which can nondeterministically fail to acquire the resource. If $\text{acq}$ fails, it outputs 0; otherwise it outputs 1. Without the output bit $\text{out}$, $P_2$ is not visibly deterministic. In particular, the sequence $\text{acq} \cdot \text{read}$ may or may not lead to error, and neither CxSafe nor CxPerm holds. However, if we model the output $\text{out}$, then $P_2$ is visibly deterministic. Figure 3(B) shows a safe and permissive witness $W_3$.

\begin{algorithm}[h]
\caption{BuildInterface}
\begin{algorithmic}[1]
\Require open program $P$, predicates $\Pi_S$ and $\Pi_P$.
\Ensure safe and permissive witness graph $W$ for $P$.
\State \textbf{Step 1:} $W := \text{ReconstructMax} \cdot \Pi_S$
\State \textbf{Step 2:} $p := \text{Check}.P.W.\Pi_P.(W,V^-).(P.E)$
\If{$p = \text{True}$} \Return{$W$} \EndIf
\State \textbf{Step 3:} $\{W$ is not permissive; $p$ is a witness ctrx.$\}$
\State $\text{CtRx}() := p$
\State \textbf{match Refine.$P.\delta$ with}
\Statex \hspace{1em} $| \text{CxPerm}() \Rightarrow \Pi_P := \Pi_P \cup \Pi; \text{go to Step 2}$
\Statex \hspace{1em} $| \text{CxSafe}() \Rightarrow \Pi_S := \Pi_S \cup \Pi; \text{go to Step 1}$
\end{algorithmic}
\end{algorithm}

The Witness Inference Algorithm. We combine the previous ideas in our witness inference procedure BuildInterface shown in Algorithm 5.

\begin{algorithm}[h]
\caption{BuildInterface}
\begin{algorithmic}[1]
\Require open program $P$, predicates $\Pi_S$ and $\Pi_P$.
\Ensure safe and permissive witness graph $W$ for $P$.
\State \textbf{Step 1:} $W := \text{ReconstructMax} \cdot \Pi_S$
\State \textbf{Step 2:} $p := \text{Check}.P.W.\Pi_P.(W,V^-).(P.E)$
\If{$p = \text{True}$} \Return{$W$} \EndIf
\State \textbf{Step 3:} $\{W$ is not permissive; $p$ is a witness ctrx.$\}$
\State $\text{CtRx}() := p$
\State \textbf{match Refine.$P.\delta$ with}
\Statex \hspace{1em} $| \text{CxPerm}() \Rightarrow \Pi_P := \Pi_P \cup \Pi; \text{go to Step 2}$
\Statex \hspace{1em} $| \text{CxSafe}() \Rightarrow \Pi_S := \Pi_S \cup \Pi; \text{go to Step 1}$
\end{algorithmic}
\end{algorithm}

**Theorem 1.** Let $P$ be an open program; let $\Pi_1$ and $\Pi_2$ be two sets of predicates. (1) If BuildInterface.$P.\Pi_1.\Pi_2$ terminates and returns $W$, then $\Pi_1 \cap \Pi_2 \subseteq \Pi \subseteq \Pi_1 \cup \Pi_2$. (2) The procedure BuildInterface terminates if $P$ is finite-state.

**Example 13:** Let us see how the algorithm BuildInterface computes a safe and permissive interface for $P_2$. We begin with the seed abstraction $\Pi_S = \Pi_P = \{a = 0, a = 0\}$. The second predicate would have been automatically found, but we add it for brevity.

**Iteration 1**

\begin{itemize}
\item **Step 1:** We invoke BuildARG on the maximal client $\text{max}.P_2$ using the predicates $\Pi_S$ to reconstruct the maximal witness $W_1$ described in Example 10.
\item **Step 2:** We now call Check to see if the interface defined by $W_1$ is permissive. The witness client $\text{Client}.W_1$ is shown in Figure 2(A). As noted in Example 11, the permissiveness check returns the witness counterexample $\text{acq} \cdot \text{write}$ (Figure 2(B)), in the form of its CFD (Figure 2(C)).
\item **Step 3:** We infer new predicates using the counterexample witness CFD from the previous step. As discussed in Example 12, this is a CxSafe counterexample and so the new predicate $x = 0$ is added to $\Pi_S$, and we return to Step 1.
\end{itemize}

**Iteration 2**

\begin{itemize}
\item **Step 1:** We now reconstruct the maximal witness w.r.t. the predicates $\{a = 0, x = 0, e = 0\}$. The result $W_2$ is shown in Figure 1(D).
Step 2: Upon calling Check to see if this witness is permissive, we get the witness counterexample \( \text{acq:write} \) in the form of its CFD.

Step 3: This time, we have a \( \text{CxPerm} \) counterexample, as the witness counterexample sequence is not permitted, the new predicate \( x = 0 \) returned by \( \text{Refine} \) is added to \( \Pi_f \), and we return to Step 1. The new predicates \( \Pi_f \) suffice to show that \( W_2 \) is permisive, as the test in line 3 of \( \text{Check} \) succeeds.

The safe and permisive witness \( W_2 \) is returned.

4. TIGHTNESS AND FULL INTERFACES

As we saw in Section 2, a safe and a permissive labeling ensures that the witness language \( \mathcal{L} \mathcal{W} \) is a safe and permissive interface, i.e., \( \mathcal{I} \mathcal{P} \cap \mathcal{R} \mathcal{P} \subseteq \mathcal{L} \mathcal{W} \subseteq \mathcal{I} \mathcal{P} \). In general, as the next example shows, \( \mathcal{L} \mathcal{W} \) is not the full interface.

Example 14: Consider the modified version of the function \( \text{acq} \) shown in the left of Figure 3(C), and the program \( P_2 \) obtained by using the modified function and adding the static variable \( f \). This time, \( \text{acq} \) can nondeterministically fail, but failure can occur at most once. The algorithm \( \text{BuildInterface} \) produces exactly the same witness graph \( W_3 \) (Figure 3(B)) for \( P_2 \). However, there are correct clients, e.g., the client shown on the right in Figure 3(C), which cause failure to occur against this interface. The false alarms arise from the fact that the witness graph is unaware that \( \text{acq} \) can never fail twice. Once \( \text{acq} \) fails (i.e., returns 0), then when called again, it will succeed (return 1), and thus thereafter, \( \text{read} \) is permitted. The problem is not that the client does not check the output of \( \text{acq} \) the second time, but that the client generates the sequence \( (\text{acq}, 0); (\text{acq}, 0); \text{read} \); which, while permitted, is not accepted by \( W_3 \). The sequence \( (\text{acq}, 0); (\text{acq}, 0); \text{read} \) is not in \( \mathcal{I} \mathcal{P} \), and so trivially cannot lead to an error. Hence it is permitted, and this sequence is in \( \mathcal{L} \mathcal{P} \). A permissive labeling on \( W \), however, looks only for feasible sequences precluded by \( W \). As \( W_3 \) permits all feasible sequences, it passes the check.

In order to ensure that the language of a witness graph is the full interface \( \mathcal{I} \mathcal{P} \), we must constrain its region labelings further. A region labeling \( \rho \) is tight if (PS) for every \( n \xrightarrow{\text{acq}, 0} n' \), we have either \( \rho \cdot n' = \emptyset \) or \( \rho \cdot n \subseteq \text{Pre}(f, o) \cdot (\rho \cdot n') \).

Proposition 6: For an open program \( P \) and witness graph \( W \), if there exists a safe labeling \( \rho_1 \), a permissive labeling \( \rho_2 \), and a tight labeling \( \rho_3 \), then \( \mathcal{L} \mathcal{W} = \mathcal{I} \mathcal{P} \).

To generate a witness for the full interface, we add a third phase to our algorithm, which takes a witness graph and a safe labeling, and tries to produce a tight labeling. If it fails, the procedure generates new predicates that refine the current abstractions. This is performed by the procedure \( \text{Tighten} \). The procedure \( \text{Tighten} \) takes a witness graph and a safe region labeling, and tries to ensure that the labeling is tight by iterating over each edge and checking if the labeling is tight for that edge. It returns either that the labeling is tight, or a set of new predicates. For an edge \( \mathcal{L} \mathcal{W} \xleftarrow{\text{acq}, 0} \mathcal{L} \mathcal{W} \), it checks if the labeling \( \rho \) is tight, i.e., if \( \rho \cdot n \subseteq \text{Pre}(f, o) \cdot (\rho \cdot n) \). If not, it finds a set \( \Pi' \) of predicates such that there is a region \( r \) definable with predicates in \( \Pi' \) for which \( \rho \cdot n \land r \neq \emptyset \) but \( \rho \cdot n \land r \land \text{WP}(f, o) \cdot (\rho \cdot n) = \emptyset \). Unfortunately, since \( f \) may have infinitely many paths, we cannot directly compute \( \text{Pre}(f, o) \cdot (\rho \cdot n) \) by iterating the WP operator. Instead, the procedure \( \text{Tighten} \) constructs successive underapproximations of \( \text{Pre} \) in the following way.

The algorithm maintains a finite set of candidate paths in \( f \) as a dag \( \delta \). An underapproximation of \( \text{Pre}(\rho \cdot n) \cdot (f, o) \) is computed by computing WP over this dag \( \delta \). The paths in \( \delta \) certify the tightness condition for all states in \( \rho \cdot n \land \text{WP}(\rho \cdot n) \cdot \delta \) in that for every state \( s \) in this set, there is some path in \( \delta \) that takes \( s \) to a state in \( \rho \cdot n' \). Let \( r' = \rho \cdot n \land \text{WP}(\rho \cdot n) \cdot \delta \) denote the complement of the above set. If \( r' \) is empty, then all states satisfying \( \rho \cdot n \) can reach some state in \( \rho \cdot n' \) when the client calls \( (f, o) \); so the current edge is tight. If \( r' \) is not empty, then either the states in \( r' \) do not reach \( \rho \cdot n' \), or the dag \( \delta \) does not suffice to show that they do. We distinguish these two cases by checking if there is some state in \( r' \) that can reach a state in \( \rho \cdot n' \) on the call \( (f, o) \). This is an abstract reachability question, which either returns an abstract counterexample dag, or that no state in \( \rho \cdot n' \) is reachable. In the first case, we check if the abstract counterexample dag is genuine. Genuine counterexample dags are added to \( \delta \), and we repeat the loop. Spurious counterexamples yield predicates that are added to \( \Pi' \) and the loop is repeated. In the second case, no state in \( r' \) can reach the target \( \rho \cdot n \land \text{WP}(f, o) \cdot (\rho \cdot n) \). When we track the predicates in \( \Pi' \), the predicates \( \Pi' \) allow us to break \( \rho \cdot n \) into states that can reach \( \rho \cdot n' \) via \( (f, o) \) and those that cannot, and these are returned. Note that tightness is not the same as termination, which requires that a function terminates along all paths. Here, we only need to find some path to \( \rho \cdot n' \).

Example 15: For the program \( P_4 \), which uses the modified \( \text{acq} \) of Figure 3(C), the witness \( W_3 \) in Figure 3(B) is not tight. In particular, the edge \( n_0 \xrightarrow{\text{acq}, 0} n_0 \) is not tight. The procedure \( \text{Tighten} \) finds the extra predicate \( f = 0 \); indeed, from every state in \( a = 0 \land f = 0 \), there is a feasible path in \( \text{acq} \) to a state in \( a = 0 \). When we track this additional predicate, the algorithm \( \text{BuildInterface} \) returns the witness graph \( W_4 \) (Figure 3(D)). This witness graph admits a safe, permissive, and tight labeling (shown in Figure 3(D)). Thus it defines the full interface for \( P_4 \).

The procedure \( \text{BuildInterface'} \) results from iterating \( \text{BuildInterface} \) using the additional predicates supplied by the algorithm \( \text{Tighten} \), until \( \text{Tighten} \) confirms that the labeling is tight.

Theorem 2. Let \( P \) be an open program and let \( \Pi_1, \Pi_2 \) be two sets of predicates. (1) If \( \text{BuildInterface'} \cdot P \cdot \Pi_1 \cdot \Pi_2 \) terminates and returns \( W \), then \( \mathcal{L} \mathcal{W} = \mathcal{I} \mathcal{P} \). (2) The procedure \( \text{BuildInterface'} \) terminates if \( P \) is finite-state.

5. EXPERIENCES

We implemented the algorithm to generate permissive interface witnesses using the BLAST software model checker. In order to be practical, we implemented several optimizations of the described method. First, while it is conceptually simpler to explain the counterexample analysis phase separately from the reachability phase, in practice, we integrated counterexample-guided refinement within the reachability procedure (using lazy abstraction) to rule out infeasible abstract counterexamples [12]. Second, we implemented a value-flow analysis, similar to field splitting [14], to detect which global variables are read and written by which library methods, and partition the set of all library methods according to shared internal state.

We ran our implementation on all examples from [1]. For the class \( \text{Signature} \), the exception \( \text{SignatureException} \) was marked as the error condition. For \( \text{ServerTableEntry} \) we
considered the exception \texttt{INTERNAL}, which is raised when the state machine maintained by the class is in a “wrong” state. For \texttt{ListItr}, we considered \texttt{IllegalStateException}.

In addition, we ran the tool on the class \texttt{Socket} of JDK1.4, where we considered the exception \texttt{SocketException}.

In all of these examples, the class maintains the interface internally using a set of private variables. For example, \texttt{Signature} internally maintains a state variable \texttt{state} that is in three states: uninitialized, sign, or verify. Upon initialization using \texttt{initVerify} (resp. \texttt{initSign}), the state becomes verify (resp. sign). Calling the \texttt{sign} (resp. \texttt{verify}) method on an uninitialized object, or one initialized using \texttt{initVerify} (resp. \texttt{initSign}), raises a \texttt{SignatureException}. Our algorithm infers a three-state witness graph that represents this interface, with the expected labeled \texttt{state} = \texttt{initVerify}, \texttt{state} = \texttt{initSign}, and \texttt{state} = \texttt{uninitialized}. This conforms to the documentation in the JDK1.4 API specification, which specifies how this object should be used.

In \texttt{Socket}, we considered the public methods \texttt{connect}, \texttt{close}, \texttt{bind}, \texttt{getInputStream}, \texttt{getOutputStream}, \texttt{shutdownInput}, and \texttt{shutdownOutput}. The \texttt{value} flow finds out that we need to define witnesses for the functions \texttt{connect}, \texttt{close}, \texttt{shutdownInput}, and \texttt{getInputStream} together (and similarly for the output streams). The algorithm finds the state bits maintained by the class, and finds an interface that enforces the requirement that \texttt{getInputStream} can be called only after a call to \texttt{connect}, but when \texttt{close} or \texttt{shutdownInput} has not been called. We require six predicates that keep track of the internal state.

6. MODULAR PROGRAM ANALYSIS

For an open program \( P \), and a client \( Cl \) for \( P \), instead of analyzing \( P \) with the client, we can use a smaller \textit{interface program} constructed from a witness for \( I.P \) to verify the client. A state \( s \) of a closed program \( (Cl, P) \) is safe if \( t \notin P.E \). The program \( (Cl, P) \) is safe if all its reachable states are safe. A client \( Cl \) is safe w.r.t. \( P \) if \( (Cl, P) \) is safe. Given a witness graph \( W \) for \( P \), we construct an open program \( IntfP.W \) as follows. There are three static variables \texttt{state} (whose values range over the set of states \( W.N \) of \( W \)’s \texttt{out} (whose values range over \( P.Outs \)), and a variable \texttt{err}. The signature is \( P.E.W \). For each \((f, o) \in P.E \), there is a function \( (\text{IntfP}.W).f \in (\text{IntfP}.W) \) that encodes the edge relation of \( W \) as follows. There is a branch in \( (\text{IntfP}.W).f \) for each edge \( n \xrightarrow{f_o} n' \) in \( W \). On this branch, the CFA checks that \texttt{state} = \texttt{n} (by \texttt{assume} \( \text{state} = n \)), then sets \texttt{state} to \texttt{n’}, \texttt{out} to \texttt{o}, and \texttt{err} to 1 if \( n' \) is unsafe. In the initial state \((\text{IntfP}.W).s_0 \) the variable \texttt{state} equals the root node \( W.n_0 \), and \texttt{out} and \texttt{err} are 0. The set \((\text{IntfP}.W).Outs \) of outputs is \( P.Outs \). The error region \((\text{IntfP}.W)).E.W = 1 \). It is easy to see that \( I.(\text{IntfP}.W) = L.W \). When verifying that the client uses the library correctly, we can verify the client with this interface program instead of the entire library. The following theorem states the correctness of this substitution.

\begin{theorem}
Let \( P \) be an open program, \( W \) be a witness for \( P \), and \( IntfP.W \) be the interface program of \( W \). For every client \( Cl \) : (1) If \( L.W \) is a safe interface, then \( (Cl, P) \) is safe if \( (Cl, IntfP.W) \) is safe. (2) If \( L.W \) is the full interface for \( P \), then \( (Cl, P) \) is safe iff \((Cl, IntfP.W) \) is safe.
\end{theorem}

7. REFERENCES


