Permissive Interfaces

Thomas A. Henzinger   Ranjit Jhala   Rupak Majumdar

EPFL, Switzerland   UC Berkeley   UC Los Angeles

Abstract

A modular program analysis considers components independently and provides succinct summaries for each component, which can be used when checking the rest of the system. Consider a system comprising of two components, 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.

1 Introduction

A modular program analysis considers components independently and provides succinct summaries for each component, which can be used when checking the rest of the system. 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 $\{\texttt{acq}, \texttt{read}, \texttt{rel}\}$, 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 $(\texttt{acq};\texttt{read};\texttt{rel})^*$ 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 $L = \emptyset$ 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 $(\texttt{acq};\texttt{read}^*;\texttt{rel})^*$. 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 $(\texttt{acq};\texttt{read}^*;\texttt{rel})^*$ 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* [?]: 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 $(\texttt{acq};\texttt{read}^*;\texttt{rel})^*$ 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 has 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 that it includes 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 typestate interpretations 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 [13]. 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 [9, 5] 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 [14, 9, 6, 15, 1, ?]. However, all these approaches either assume that the interface is specified by the programmer [14, 9, 6, ?]; or use a set of dynamic executions of the library to define its interface [2, 15, 11], with the result that the constructed interface may be unsafe or not permissive; or perform static analyses that are not precise enough to create permissive interfaces [15, 1]. 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, $\mathsf{Exp}.X$ is the set of arithmetic expressions over the variables X, the set $\mathsf{Pred}.X$ is the set of boolean expressions (arithmetic comparisons) over X, $\mathsf{V}.X$ is the set of valuations to X, and the set $\mathsf{Op}.X$ is the set of operations containing: (1) assignments `x := e`, where $\mathtt{x} \in X$ and $\mathtt{e} \in \mathsf{Exp}.X$, (2) *assume predicates* `assume [p]`, where $\mathtt{p} \in \mathsf{Pred}.X$, representing a condition that must be true for the edge to be taken, and (3) function calls $f()$, where $\mathtt{x} \in X$ 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 of control locations Q, with an initial control location $q_0 \in Q$ and final control location $q_e \in Q$, (3) a finite set of directed edges labeled with operations $\rightarrow \subseteq Q \times \mathsf{Op}.(X_L \cup X_S) \times Q$. Let $X = X_L \cup X_S$.

An *open program* $P = (X, F, \mathit{Outs}, s_0, \mathcal{E}, \Sigma)$ has (1) a set X of typed *static variables* including a special output variable `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 of outputs Outs, (4) an initial state $s_0 \in \mathsf{V}.X$, (5) a set of error states $\mathcal{E} \subseteq \mathsf{V}.X$ given as a predicate in $\mathsf{Pred}.X$, and, (6) a *signature* Σ which is a subset of $F \times \mathit{Outs}$. An open program is finite-state if all variables range over booleans. The signature Σ represents the externally visible function names and output values.

EXAMPLE 1: Figure 1(A) shows an open program P_1 with the two static variables $P_1.X = \{\mathtt{a}, \mathtt{e}\}$. The latter is used to define the error states $P_1.\mathcal{E} = (\mathtt{e} \neq 0)$. 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 $\mathsf{V}.X$. For disjoint sets X and Y of variables, if $s \in \mathsf{V}.X$ and $t \in \mathsf{V}.Y$, we write $s \circ t \in \mathsf{V}.(X \uplus Y)$ for the $X \uplus Y$ state obtained by combining s and t. For an X_1, X_2-states s_1, s_2, we say that $s_1 \approx s_2$ if for all $\mathtt{x} \in X_1 \cap X_2$, the states agree: $s_1.\mathtt{x} = s_2.\mathtt{x}$. We assume all variables are integer-valued. 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 $\overset{C}{\rightsquigarrow} \subseteq (\mathsf{V}.X_S)^2$ is defined as follows: $s \overset{C}{\rightsquigarrow} s'$ if there exists $(q_1, (s_1 \circ t_1)) \ldots, (q_n, (s_n \circ t_n))$ such that $(q_0, s) = (q_1, s_1)$ and $(q_e, s') = (q_n, s_n)$ and for each $1 \leq i \leq n-1$ we have

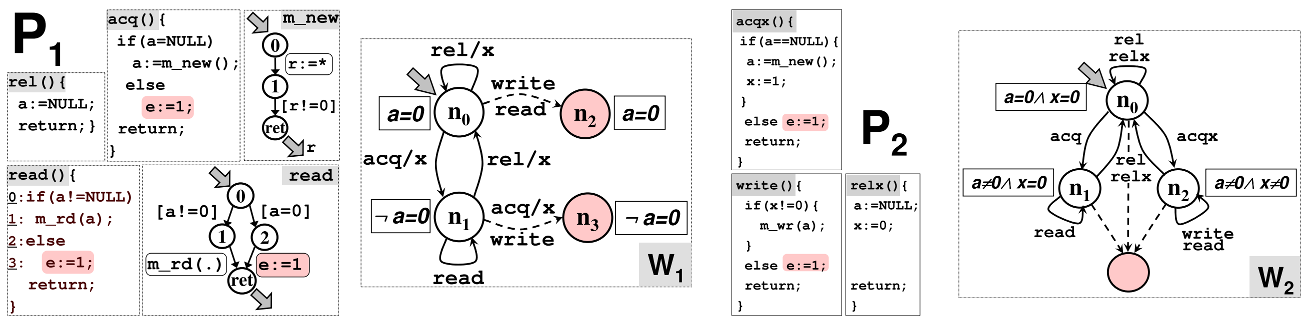

Figure 1: (A) P_1 (B) safe W_1 (C) P_2 (D) safe,permissive W_2

$q_i \xrightarrow{\mathsf{op}_i} q_{i+1}$ in $C.E$ and $s_i \circ t_i \overset{\mathsf{op}_i}{\rightsquigarrow} s_{i+1} \circ t_{i+1}$, and each $\overset{\mathsf{op}_i}{\rightsquigarrow} \subseteq (\mathsf{V}.(C.X))^2$ is as defined below. We say that $s \overset{\mathsf{op}}{\rightsquigarrow} s'$ for $s, s' \in \mathsf{V}.X$ if: (1) if $\mathtt{op}$ is $\mathtt{assume}\ p$ then $s \models p$ and $s'.y = s.y$ for all $y \in X$, and (2) if $\mathtt{op}$ is $\mathtt{x} := \mathtt{e}$ then $s'.x = s.e$ and for all $y \in X \setminus \{x\}$, $s'.y = s.y$, and, (3) if $\mathtt{op}$ is $\mathtt{f}()$ then for all $\mathtt{y} \in X_L$, $s.\mathtt{y} = s'.\mathtt{y}$ and there exists $t, t' \in \mathsf{V}.X_S$ such that (a) $t \overset{f.C}{\rightsquigarrow} t'$, and (b) $s \approx t, s' \approx t'$. Parameter passing and return values can be mimicked with static variables.

We lift the transition relation to sets of states as follows. For a set of X-states r, the *strongest postcondition* of r w.r.t. $\mathtt{op}$, written $\mathsf{SP}.r.\mathtt{op}$, is the set $\{s' \in \mathsf{V}.X \mid \exists s \in r : (s \overset{\mathsf{op}}{\rightsquigarrow} s')\}$. The *weakest precondition* of r w.r.t. $\mathtt{op}$, written $\mathsf{WP}.r.\mathtt{op}$ is the set $\{s \in \mathsf{V}.X \mid \exists s' \in r : (s \overset{\mathsf{op}}{\rightsquigarrow} s')\}$. We can generalize SP, WP to sequences of operations in the standard way. For assign and assume statements, and r represented using predicates in $\mathsf{Pred}.X$, the above can be defined as predicate transformers [7].

An open program induces a state space $\mathsf{V}.X$. Each $(f, o) \in \Sigma$, has a transition relation $\overset{(f,o)}{\rightsquigarrow} \subseteq (\mathsf{V}.X)^2$ defined as: $s \overset{(f,o)}{\rightsquigarrow} s'$ if such that $s \overset{f.C}{\rightsquigarrow} s'$ and $s'.\mathtt{out} = o$. We extend this operation to data regions by defining $\mathsf{Post}.r.(f, o) = \{s' \mid \exists s \in r : s \overset{(f,o)}{\rightsquigarrow} s'\}$, and its dual, $\mathsf{Pre}.r.(f, o) = \{s \mid \exists s' \in r : s \overset{(f,o)}{\rightsquigarrow} s'\}$.

Example 2: In the open program P_1 (Figure 1(A)), the initial states are given by $a = 0 \wedge \mathtt{e} = 0$, and error states $P_1.\mathcal{E}$ by $(\mathtt{e} \neq 0)$. The strongest postcondition $\mathsf{SP}.r.(\mathtt{a} := \mathtt{m\_new}())$ of the region $r = (a = 0)$ is $(a \neq 0)$, and $\mathsf{Post}.r.(\mathtt{acq}(), \mathtt{o})$ is also $(a \neq 0)$ as only the "if" path is feasible. □

2.2 Interfaces

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

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

Consider now the open program P_2 obtained by augmenting P_1 with the additional static variable x, and the functions $\mathtt{acq\_x}$, $\mathtt{write}$, and $\mathtt{rel\_x}$ 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 $\mathtt{acq\_x} \cdot \mathtt{write} \cdot \mathtt{rel\_x}$! Indeed, after calling $\mathtt{acq\_x}$, the client may call $\mathtt{read}$ or $\mathtt{write}$ arbitrarily often and the full interface for P_2 is the regular language $((\mathtt{acq} \cdot \mathtt{read}^* \cdot \mathtt{rel}^*) + (\mathtt{acq\_x} \cdot (\mathtt{read} + \mathtt{write})^* \cdot \mathtt{rel\_x}^*) + \mathtt{rel} + \mathtt{rel\_x})^*$. □

2.3 Witnesses

Witness Graphs. We focus on interfaces corresponding to regular languages, which are naturally witnessed by finite state graphs. A *witness graph* $W = (N, E, \mathtt{n}_0)$ 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 $\mathtt{n}_0 \in N$. We write $\mathtt{n} \xrightarrow{(f,o)} \mathtt{n}'$ if $(\mathtt{n}, (f, o), \mathtt{n}') \in E$, and call $\mathtt{n}'$ a (f, o)-successor of $\mathtt{n}$. Additionally, we require that every $\mathtt{n} \in N$ 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 nodes and deleting all the unsafe nodes. Formally, a sequence $\sigma \in P.\Sigma^*$ is *accepted* from a node $\mathtt{n} \in N^+$ if σ is the empty sequence, or $\sigma \equiv (f,o)\cdot\sigma'$ and $\mathtt{n}$ has a (f,o)-successor $\mathtt{n}' \in N^+$ such that σ' is accepted from $\mathtt{n}'$. No sequence is accepted from any node in N^-. The *language* of a witness graph $\mathcal{L}.W$ is the set of all σ accepted from $\mathtt{n}_0$.

EXAMPLE 4: Figure 1(B) shows a witness graph W_1 for P_1 (ignoring the "x" and write labels). The unshaded and shaded nodes are respectively N^+ and N^-. Note that $\mathcal{L}.W_1$ is $((\mathtt{acq}\cdot\mathtt{read}^*\cdot\mathtt{rel})^*\cdot\mathtt{rel}^*)^*$, 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 $\rho : N \to \mathsf{Pred}.(P.X)$ mapping the nodes of a candidate graph G to predicates is a *region labeling* if:
(P1) $P.s_0 \in \rho.\mathtt{n}_0$, *i.e.,* the root $\mathtt{n}_0$ label contains the initial state, and
(P2) For every $\mathtt{n}\xrightarrow{(f,o)}\mathtt{n}'$ we have $\mathsf{Post}.(\rho.\mathtt{n}).(f,o) \subseteq \rho.\mathtt{n}'$.
A region labeling represents an overapproximation of the behavior of an open program: for any node $\mathtt{n} \in N^+$, and any sequence of operations $\sigma \in P.\Sigma^*$ marking a path from the root $\mathtt{n}_0$ to $\mathtt{n}$ in G, we have $\mathsf{Post}.(\rho.\mathtt{n}_0).\sigma \subseteq \rho.\mathtt{n}$.

Proposition 1 **[Safe Labels]** *For an open program P and witness graph W, if there exists a region labeling ρ such that:* (P3) *for each $\mathtt{n} \in N^+$ the label $\rho.\mathtt{n} \subseteq \neg P.\mathcal{E}$, then $\mathcal{L}.W \subseteq \mathcal{I}.P$.*

The existence of a *safe labeling* for W, namely one the satisfies (P3), guarantees that the language of W is *bounded above* by $\mathcal{I}.P$, *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 acq_x, rel_x, write edge labels. The label for each node is written in the box next to the node; for the safe (unshaded) nodes there is an implicit conjunct ($\mathtt{e} = 0$ holds), and for the unsafe (shaded) nodes, the label is $\mathtt{e} = 1$. We can check that this is a region labeling: *e.g.,* for the edge $\mathtt{n}_0\xrightarrow{\mathtt{acq}}\mathtt{n}_1$ we saw that $\mathsf{Post}.(a = 0).\mathtt{acq}$ equals $a \neq 0$, *i.e.,* the successor states from the label of $\mathtt{n}_0$ are contained in the label of $\mathtt{n}_1$. 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_2. However, this interface is too restrictive: it prohibits the client from ever calling write, as the 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 ρ such that:* (P4) *for each $\mathtt{n} \in N^-$ the label $\rho.\mathtt{n} \subseteq P.\mathcal{E}$, then $\mathcal{I}.P \cap \mathcal{R}.P \subseteq \mathcal{L}.W$.*

The existence of a *permissive labeling* for W, namely one the satisfies (P4), guarantees that the language of W is *bounded below* by $\mathcal{I}.P \cap \mathcal{R}.P$, *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 $\mathcal{L}.W_1$ contains all legal, realizable sequence, and hence, its language 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 a safe labeling:$[\mathtt{n}_0 \mapsto (a = 0); \mathtt{n}_1 \mapsto a \neq 0; \mathtt{n}_2 \mapsto (a \neq 0 \wedge x \neq 0)]$, again, $\mathtt{e} = 0$ and $\mathtt{e} = 1$ are implicit for the unshaded, shaded nodes respectively. It also has a permissive labeling: $[\mathtt{n}_0 \mapsto (a = 0 \wedge x = 0); \mathtt{n}_1 \mapsto (a \neq 0 \wedge x = 0); \mathtt{n}_2 \mapsto a \neq 0]$, 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 they are simultaneously safe and permissive. □

The above examples demonstrate that in order witness graph's language 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 this example shows, the two region labelings may be different, and in general may be shown using orthogonal overapproximations of the state space. Hence, we treat the two labelings separately.

3 Constructing Permissive 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 witness 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 ($\neg P.\mathcal{E}$). 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 ($P.\mathcal{E}$). 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 then use the given safety abstraction, to compute an overapproximation of the behaviors of P, as an abstract reachability graph, from which we obtain a safe candidate witness by treating nodes that intersect $P.\mathcal{E}$ as unsafe nodes, and the rest as safe nodes. Next, we use the method outlined above, together with the supplied permissiveness abstraction, to verify that this reconstructed candidate is permissive, and if so, we are done.

For witness inference, we obtain sufficient abstractions, and through them a safe, 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. To do this check we shall 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 $Cl = (X, \emptyset, Q, q_0, q_e, \rightarrow)$ where the operations on the edges are assignments and assumes from before, as well as *function calls* $y := f()$, where $(f, \cdot) \in P.\Sigma$, and $y \in X$. For convenience, we introduce the operation $y := (f(), o)$ as shorthand for the sequence of operations $y := f(); \texttt{assume}\ [y = o];$, where y is not subsequently read, we further compress this to (f, o).

Closed Programs. A *closed* program (Cl, P) consists of a client Cl and an open program P. A program (Cl, P) induces a state space $\mathsf{V}.(Cl.X \uplus P.X)$. Let $s, s' \in \mathsf{V}.(Cl.X)$ and $t, t' \in \mathsf{V}.(P.X)$. Then $s \circ t \overset{\mathtt{op}}{\rightsquigarrow} s' \circ t'$ if (1) $s \overset{\mathtt{op}}{\rightsquigarrow} s'$ and $t = t'$ if $\mathtt{op}$ is an assignment or assume, and (2) there exists $o \in P.O$ such that $t \overset{(f,o)}{\rightsquigarrow} t'$ and $s' = s[o/y]$ if $\mathtt{op}$ is the function call $y := f()$. For a subset $L \subseteq Q$ of the client locations, we say $s \circ t \in \mathsf{V}.(Cl.X \cup P.X)$ is *L-reachable* if there exist $(q_0, (s_0 \circ t_0) \ldots, (q_n, (s_n \circ t_n)) \in Cl.Q \times \mathsf{V}.(Cl.X \cup P.X)$, 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 $q_i \xrightarrow{\mathtt{op}_i} q_{i+1}$ in Cl such that $s_i \circ t_i \overset{\mathtt{op}_i}{\rightsquigarrow} s_{i+1} \circ t_{i+1}$, 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 $\mathsf{Client}.W = (\{\mathtt{x}\}, \emptyset, V \cup \{pc_e\}, v_0, pc_e, \rightarrow)$ as follows. The client has a single variable $\mathtt{x}$, which is used to capture the output from function calls. For every edge $v \xrightarrow{(f,o)} v'$ in W, there is a corresponding CFA edge $v \xrightarrow{\mathtt{op}} v'$ in $\mathsf{Client}.W$ where $\mathtt{op}$ is $\mathtt{x} := (f(), o)$. 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:

Algorithm 1 BuildARG

Input: A Closed Program (Cl, P), Set of Predicates Π
Output: Abstract Reachability Graph A of (Cl, P) w.r.t Π.

```
L := {n : (Cl.pc_0, Abs.Π.(P.s_0))}
seen := ∅, A := ∅
while L ≠ ∅ do
  pick and remove state n : (pc, r) from L
  if (n ∉ seen) then
    seen := seen ∪ {n}
    for each (pc, op, pc') of Cl do
      r' := SP.Π.op.r; n' := Connect.A.((pc, r), op, (pc', r'))
      L := L ∪ {n'}
return A
```

(1) A safe interface for P iff the V^+-reachable states of $(\mathsf{Client}.W, P)$ are $P.\mathcal{E}$-safe, *i.e.,* the client W never reaches a state in $P.\mathcal{E}$ when it is at a "safe" node,
(2) A permissive interface for P iff the V^--reachable states of $(\mathsf{Client}.W, P)$ are $\neg P.\mathcal{E}$-safe, *i.e.,* the client W never reaches a state in $\neg P.\mathcal{E}$ when it is at an "unsafe" wnode.

As the above reachability checks are undecidable in general, we need abstractions of the open program, with which to overapproximate the reachable states.

Predicate Abstraction. For a set of predicates $\Pi \subseteq \mathsf{Pred}.X$ and a formula r over X, let $\mathsf{Abs}.\Pi.r$ denote the smallest (in the inclusion order) data region containing r expressible as a boolean formula over atomic predicates from Π. For example, if $\Pi = \{a = 0, b = 0\}$ and $r = (a = 3 \ \wedge \ b = a + 1)$, then the predicate abstraction of r w.r.t. Π is $\neg(a = 0) \wedge \neg(b = 0)$. The abstract postcondition of r and $\mathtt{op}$ w.r.t. Π, written $\mathsf{SP}_\Pi.r.\mathtt{op}$, is a boolean combination of predicates from Π which overapproximates $\mathsf{SP}.r.\mathtt{op}$. The procedure SP computes the abstract postcondition. It takes as input the set of predicates Π, an operation $\mathtt{op}$, and the current region r, and returns $\mathsf{SP}_\Pi.r.\mathtt{op}$. 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, 13].

Abstract Reachability Graphs. Given an open program P, client Cl and set of predicates Π an *abstract reachability graph*(ARG) for (Cl, P) w.r.t. Π, is a rooted directed graph where each node is labeled by pairs (pc, r) such that: (1) The root node $\mathtt{n}_0$ is labeled (pc_0, r_0) where pc_0 is the initial location of Cl and r_0 is the predicate abstraction of the initial state s_0 of the system w.r.t. Π. (2) Each node $\mathtt{n}$ labeled (pc, r) has an $\mathtt{op}$-successor (pc', r') for every edge $pc \xrightarrow{\mathtt{op}} pc'$ in Cl, such that $r' = \mathsf{SP}_\Pi.r.\mathtt{op}$. If Π is finite, then the ARG is also finite. Procedure $\mathsf{BuildARG}$ shown in Algorithm 1 constructs ARGs using predicate abstraction. It takes as input an open program P, a client Cl, and a set of predicates Π. The algorithm incrementally builds the ARG, by constructing successors of nodes, and merging nodes that have the same abstract state.

EXAMPLE 7: Consider again P_1 from Figure 1(A), and the witness W_1 shown to its right. Algorithm $\mathsf{BuildARG}$ computes the ARG in Figure 1(B) for $(\mathsf{Client}.W_1, P_1)$ w.r.t the set of predicates $\Pi_1 = \{\mathsf{e} = 0, \mathsf{a} = 0\}$. □

The Witness Checking Algorithm. Notice that from the ARG A constructed by $\mathsf{BuildARG}$, we can construct a region labeling for W by mapping node v of W to $\vee_{\mathtt{n}:(v,r)} r$, the union of all regions r marking the nodes of the ARG A where the client is at location v.

Proposition 3 [Abstract Region Labelings] *Let W be*

Algorithm 2 Check

Input: Open Program P, Witness $W = (V, E, v_0)$
Input: Predicates Π, Vertices $V' \subseteq V$, Target Region $\mathcal{E}$
Output: TRUE or a counterexample CFD CTRX(δ)

1: $A :=$ BuildARG.P.(Client.W).Π
2: $\rho := \lambda v.(\bigvee_{\mathtt{n}:(v,r)} r)$
3: **if exists** $v \in V' : \rho.v \not\subseteq \mathcal{E}$ **then**
4: Find $\mathtt{n} : (v, r)$ s.t. $r \not\subseteq \mathcal{E}$
5: $\sigma :=$ path from $\mathtt{n}_0$ to $\mathtt{n}$ in A
6: **return** $\mathsf{dag}_\Pi.\sigma$
7: **else**
8: **return** TRUE

a witness graph for open program P. Let A be the Abstract Reachability Graph for (Client.W, P)*, w.r.t. any set of predicates Π. The map $\rho_G = \lambda v.(\bigvee_{\mathtt{n}:(v,r)} r)$, is a region labeling for W, i.e., ρ_G has properties* (P1),(P2).

Hence, given a set of predicates Π_S, we invoke procedure BuildARG to compute an ARG A_S for (Client.W, P), w.r.t. Π_S, and hence a region labeling ρ_S. This labeling is *safe*, *i.e.,* satisfies (P3), and so we are guaranteed that W is a safe witness. This check is precisely stated by invoking Algorithm Check.$P.W.\Pi_S.V^+.(\neg P.\mathcal{E})$, shown in Algorithm 2, which returns TRUE if the region labeling constructed in line 2 is a safe labeling for W.

More importantly, we can use the same technique to check if W is permissive: given a set of predicates Π_P, we can use BuildARG to compute an ARG A_P for (Client.W, P), and if the corresponding region labeling ρ_P is a *permissive* region labeling, *i.e.,* satisfies (P4), then we are guaranteed that W is a permissive witness. Once again, this check is carried out by invoking Algorithm Check.$P.W.\Pi_P.V^-.(P.\mathcal{E})$, shown in Algorithm 2, which returns TRUEif the region labeling constructed is a permissive labeling for W. Notice, that in doing so we have used an overapproximate analysis (BuildARG), to obtain a *lower bound* on the language of W, *i.e.,* instead of the traditional use of overapproximation – namely to guarantee that the behaviors of a program *are contained in* some (safe) set, we have used an overapproximation to ensure that the behaviors of the program *contain* a desirable (permitted) set.

Proposition 4 [Witness Checking] *For an open program P, let $W = (V^+ \cup V^-, E, v_0)$ be a witness graph s uch that for predicate sets Π_S, Π_P:*
(1) Check.$P.W.\Pi_S.V^+.(\neg P.\mathcal{E})$ *returns* TRUE *and,*
(2) Check.$P.W.\Pi_P.V^-.(P.\mathcal{E})$ *returns* TRUE.
Then $\mathcal{I}.P \cap \mathcal{R}.P \subseteq \mathcal{L}.W \subseteq \mathcal{I}.P$, i.e., W is a safe, permissive witness for P.

Example 83.1 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. Also, while we use predicate abstraction, checking can be performed with any abstract domain for which fixpoints are computable.

EXAMPLE 8: Consider an open program with variables x, y, and e, all initially 0, and two methods f_1 and f_2 defined as: f_1 : `if(x` $\neq$ `0) then e := 1;` f_2 : `if(y = 0) then e := 1;`. The error states are $\mathtt{e} \neq 0$, we omit the output. Consider the witness graph $W = (\{\mathtt{n}_0, \mathtt{n}_1\}, \{(\mathtt{n}_0, f_1, \mathtt{n}_0), (\mathtt{n}_0, f_2, \mathtt{n}_1\}, \mathtt{n}_0)$, with $N^+ = \{\mathtt{n}_0\}$ and $N^- = \{\mathtt{n}_1\}$. The labeling $\rho.\mathtt{n}_0 = (\mathtt{x} = 0 \wedge \mathtt{e} = 0)$ and $\rho.\mathtt{n}_1 = \mathtt{true}$ is safe. Since $\mathtt{n}_1 \in N^-$ is unsafe, the call f_2 is not allowed. The labeling $\rho'.\mathtt{n}_0 = (\mathtt{y} = 0 \wedge \mathtt{e} = 0)$ and $\rho'.\mathtt{n}_1 = (\mathtt{e} \neq 0)$ is permissive. The safe (resp. permissive) labeling does not track $\mathtt{y} = 0$ (resp. $\mathtt{x} = 0$). □

Algorithm 3 ReconstructMax

Input: Open program P, set of predicates Π.
Output: A maximally safe witness graph W for P.

$A :=$ BuildARG.(mxc.P, P).Π
$V^+ := \{v \mid v : (\cdot, r) \in A.N \text{ s.t. } r \subseteq \neg P.\mathcal{E}\}$
$V^- := A.N \setminus V^+$
return witness graph $(V^+ \cup V^-, A.E, A.\mathtt{n}_0)$

3.2 Witness Reconstruction

In witness checking, we assumed that in addition to the abstraction predicates Π_S, Π_P, we had a given candidate witness W. In witness reconstruction, we show how to use Π_S to *reconstruct* a candidate witness W with a safe region labeling. As before, if we can then show (using Π_P) that the candidate W has a permissive labeling then we are done.

Maximal Clients. To obtain this candidate, we shall "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* mxc.P of P is the CFA $(\{\mathtt{x}\}, \emptyset, \{pc_0\}, pc_0, pc_0, \rightarrow)$, where for every $(f, o) \in \Sigma$ we have that $(pc_0, \mathtt{x} := (f(), o), pc_0) \in \rightarrow$. We then use the abstraction Π_S, 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 mxc.P_2 shown on the top in Figure 2(A). □

The Witness Reconstruction Algorithm. Consider the ARG A for (mxc.P, P) w.r.t. the predicates Π_S. We can convert this ARG into a witness graph Witness.$A = (V, E, v_0)$ by converting: (1) the nodes of the ARG into nodes of Witness.G, (2) the edges of the ARG, which were labeled by operations in $P.\Sigma$ into the edges of Witness.G, (3) the root node of the ARG as the root node of Witness.G, and letting (4) V^+ be the set $\{\mathtt{n} \mid \mathtt{n} : (pc, r) \text{ and } r \subseteq \neg P.\mathcal{E}\}$, and V^- be the complement.

As the witness corresponds to an ARG, the ARG node labels form a region labeling. The nodes were partitioned so as to make the labeling just described a *safe* region labeling. In addition, it can be shown that the witness "reconstructed" above is the biggest witness that can be shown to be safe using the abstraction Π_S. The above algorithm is formalized in Algorithm ReconstructMax in Algorithm 3.

Proposition 5 [Maximal Witness Reconstruction] *For every open program P and set of predicates Π,* ReconstructMax.$P.\Pi$ *terminates and returns a witness graph W such that:*
(1) W is a safe witness for P,
(2) For every witness W', if Check.$P.W'.\Pi.(W.V^+).(\neg P.\mathcal{E})$ *then $\mathcal{L}.W' \subseteq \mathcal{L}.W$, i.e., W is the* maximal safe witness *for P w.r.t.* Π.

EXAMPLE 10: Upon running Algorithm BuildARG on (mxc.P_2, P_2) and the set $\Pi = \{\mathtt{e} = 0, \mathtt{a} = 0\}$ of predicates, we obtain the ARG A_1 shown in Figure 2(B), which translates to the witness W_1. The unshaded nodes are those whose regions are contained in $\neg P.\mathcal{E}$,*i.e.,* $\mathtt{e} \neq 1$, *i.e.,* are

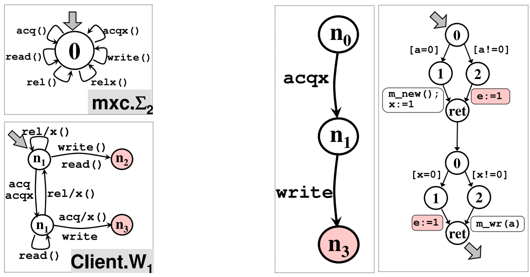

Figure 2: (A) $\mathsf{mxc}.P_2$ (↑) $\mathsf{Client}.W_1$ (↓) (B) Witness Counterexample (C) Ctrx. CF-Dag

safe w.r.t to $P_2.\mathcal{E}$. The label `acq/x` indicates two edges, one labeled with `acq` and the other with `acqx`. This reconstructed maximal witness W_1 prohibits calling `write` and hence, while being safe, is not permissive, as `write` can be safely called after first calling `acq_x`. When computing the `acq_x`-successor of $\mathtt{n}_0$, the abstract state is the same as that of $\mathtt{n}_1$, namely $\neg(\mathsf{a} = 0) \wedge (\mathsf{e} = 0)$, as there are no predicates on `x`. Hence the algorithm sets $\mathtt{n}_1$ to be the `acq_x`-successor of $\mathtt{n}_0$, thus not permitting any calls to `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. As for verification, coarse abstractions leads to false positives, for interface synthesis, coarse abstractions lead to massively constrained interfaces. As shown in the prequel, we it suffices to find abstractions Π_S and Π_P such that the maximal (safe) witness for P w.r.t. Π_S, can be shown to be permissive using Π_P. We now show how to find such Π_S, Π_P by automatically refining coarse abstractions, using *witness counterexamples* sequences of $P.\Sigma$ that are prohibited by the maximal witness but which may be permitted by the open program P.

Witness Counterexamples. We use procedure BuildARG (via the procedure Check) to see if the abstraction Π_P suffices to show that a candidate witness W is a permissive witness for P. This permissiveness check fails in line 3 of Check if in the ARG G returned by BuildARG there exists a node $\mathtt{n} : (v, r)$ such that:
(1) $r \not\subseteq P.\mathcal{E}$, *i.e.,* the resulting state of the open program is legal, and,
(2) $v \in V^-$, *i.e.,* v is an "unsafe" witness node. Consider *any* path in the ARG G from the root node of G to $\mathtt{n}$; the ARG edge labels along this path, correspond to a sequence of calls σ that:
(1') may be legal as the resulting abstract region is not contained in $P.\mathcal{E}$ states, *i.e.,* contains legal states, but which is
(2') prohibited by W, as the sequence ends in an "unsafe" witness node. Such a call sequence is 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 procedure BuildARG (Algorithm 1), we compute $\mathsf{SP}_\Pi.r.\mathtt{op}$

Algorithm 4 Refine

Input: Open program P, Control-flow Dag δ.
Output: CxPerm(Π), or, CxSafeΠ), where Π is a set of predicates.

$\varphi := \mathsf{SP}.(P.s_0).\delta$
if $\varphi \wedge \neg(P.\mathcal{E})$ is satisfiable **then**
  $\Pi :=$ GetNewPreds.$\delta.(P.s_0).(\neg P.\mathcal{E})$
  return CxPerm(Π)
else
  $\Pi :=$ GetNewPreds.$\delta.(P.s_0).(P.\mathcal{E})$ {Note: $\varphi \wedge P.\mathcal{E}$ is unsatisfiable}
  return CxSafe(Π)

where $\mathtt{op} = (f, o)$, in the standard way [10, 13] by "unrolling" the CFA for f, compute the abstract state (over the predicates Π) for each unrolled CFA location, and merge nodes with the same abstract state. The returned region r' is the union of the region unrolled exit nodes. On deleting back-edges from the unrolled CFA, and merging all exit nodes, we get a CFD encoding the possible set of paths that a state in r may take through the body of f to reach a state in r'. We shall call this CFD $\mathsf{dag}_\Pi.\mathtt{op}$. Given a sequence of edges labeled σ in the ARG built by BuildARG, we can chain together the CFDs for the individual edges to obtain $\mathsf{dag}_\Pi.\sigma$, which represents a set of paths that the open program may execute, if the sequence of calls σ is made. Whenever the check in line 3 of procedure Check fails, it returns a CFD $\mathsf{dag}_\Pi.\sigma$, corresponding to a witness counterexample σ.

Example 11: Consider the restrictive candidate witness W_1 corresponding to the ARG A_1, from the previous example 3.2. The witness client $\mathsf{Client}.W_1$ is shown in Figure 2(A). An edge with multiple labels stands for several edges, each with one of the labels. We invoke procedure Check, using the predicates $\Pi_P = \{\mathsf{e} = 0, a = 0\}$, and the target states $\neg(\mathsf{e} = 1)$ to see if this witness is permissive. To do this, 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 properties (1),(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 Π_P. The sequence of calls labeling the edges `acq_x`; `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 $P.\mathcal{E}$, if this call sequence is not permitted or *always* end in $\neg P.\mathcal{E}$, if this call sequence is permitted. In particular, if $\delta = \mathsf{dag}_\Pi.\sigma$, then the feasible paths corresponding to δ either always end in $P.\mathcal{E}$ or always end in $\neg P.\mathcal{E}$. Hence, either:
(CxPerm) $\neg P.\mathcal{E} \cap \mathsf{SP}.(P.s_0).\delta$ is unsatisfiable, meaning that σ is *not* a permitted call sequence. In this case we use a standard predicate discovery algorithm [12, **?**] to obtain new predicates Π such that $\neg P.\mathcal{E} \cap \mathsf{SP}_\Pi.(P.s_0).\sigma$ becomes unsatisfiable, *i.e.,* the resulting abstraction is precise enough to eliminate this witness counterexample, or,
(CxSafe) $\neg P.\mathcal{E} \cap \mathsf{SP}.(P.s_0).\delta$ is satisfiable, implying that the witness counterexample σ is a permitted call sequence that has been prohibited by the maximal safe witness reconstructed from Π_S. In this case, the above shows that $P.\mathcal{E} \cap \mathsf{SP}.(P.s_0).\delta$ must be unsatisfiable and so the predicate discovery algorithm infers new predicates Π such that $P.\mathcal{E} \cap \mathsf{SP}_\Pi.(P.s_0).\sigma$ becomes unsatisfiable, *i.e.,* the resulting abstraction is precise enough that its maximal safe witness contains the permitted call sequence σ.

The above is made precise in algorithm Refine shown in

Algorithm 5 BuildInterface

Input: Open program P, sets of predicates Π_S and Π_P.
Output: A safe, permissive witness graph W for P.

```
Step 1: W := ReconstructMax.P.Π_S
Step 2: p := Check.P.W.Π_P.(W.V⁻).(P.E)
if  p = TRUE then
   return  W
Step 3: {W not permissive, p is a witness ctrx. δ}
CTRX(δ) := p
match Refine.P.δ with
   | CXPERM(Π) → Π_P := Π_P ∪ Π; go to Step 2.
   | CXSAFE(Π) → Π_S := Π_S ∪ Π; go to Step 1.
```

Algorithm 4. Procedure GetNewPreds refers to the predicate discovery algorithm, which takes as input a CFD δ, and a set of intial states r, and set of states $\mathcal{E}$ s.t. $\mathsf{SP}.r.\delta \cap \mathcal{E}$ is unsatisfiable, and returns a set of predicates Π s.t. $\mathsf{SP}_{\Pi}.r.\delta \cap \mathcal{E}$ 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 feasible path (case CxSafe), the refinement procedure adds new predicates, that force the safety abstraction Π_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(D). The feasible paths of this CFD end in safe states, *i.e.,* ($\mathtt{e} = 0$), though owing to the imprecision of the abstraction, the maximally safe witness built from $\{a = 0, \mathtt{e} = 0\}$ prohibits this sequence. Hence, this is case (CxSafe), and the predicate discovery algorithm finds the new predicate $x = 0$; the resulting abstraction is precise enough to permit the sequence `acq_x`; `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_3 of P_1 with a version of `acq` shown in Figure 3(A) that can nondeterministically fail to acquire the resource. If `acq` fails it outputs 0, otherwise it outputs 1. Without the output bit `out`, P_3 is not visibly deterministic. In particular, the sequence `acq` $\cdot$ `read` may or may not lead to error, and neither CxSafe nor CxPerm holds. However, if we model the output `out`, then P_3 is visibly deterministic. Figure 3(B) shows a safe and permissive witness W_3. □

The Witness Inference Algorithm. We combine the previous ideas together in our Witness Inference Algorithm BuildInterface shown in Algorithm 5.
Step 1. We start with a (possibly trivial) abstraction Π_S, Π_P, and use the witness reconstruction algorithm to obtain a candidate witness W, largest interface that we can show is safe using Π_S.
Step 2. Next, we use the witness checking algorithm to see if the candidate W is permissive. If so, we are done, and we **output** the safe, permissive witness W. If not, the witness checking algorithm returns a witness counterexample in the form of a CFD δ.
Step 3. In the procedure Refine we check if the witness counterexample CFD δ corresponds to a permitted sequence prohibited by W. If not, *i.e.,* it is a CxPerm witness counterexample, we infer new predicates Π'_P to refine the permissiveness abstraction, and **go to step 2**. If it is a permitted sequence that is prohibited by the witness W owing to imprecision of Π_S, *i.e.,* it is a CxSafe witness counterexample, we infer new predicates Π'_S such that the resulting maximal witness will permit the sequence σ, and **go to step 1**.

Theorem 1 *Let P be an open program and let Π_1, Π_2 be two sets of predicates.*
(1) If BuildInterface.P.Π_1.Π_2 *terminates and returns W then $\mathcal{I}.P \cap \mathcal{R}.P \subseteq \mathcal{L}.W \subseteq \mathcal{I}.P$.*
(2) Procedure BuildInterface *terminates if P is finite-state.*

Example 13: Let us see show Algorithm BuildInterface computes a safe and permissive interface for P_2. We shall begin with the seed abstraction $\Pi_S = \Pi_P = \{\mathtt{e} = 0, a = 0\}$. The second predicate would have been automatically found, but we add it for brevity.
Iteration 1
Step 1_1 We invoke BuildARG on the maximal client $\mathsf{mxc}.P_2$ using predicates Π_S to reconstruct the maximal witness W_1 described in Example 10.

Step 2_1 We now call Check to see if the interface defined by W_1 is permissive. The witness client Client.W_1 is shown in Figure 2(A). As noted in example 11,the permissive check procedure returns the witness counterexample `acq_x`; `write`, Figure 2(B), in the form of its CFD Figure 2(C).

Step 3_1 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 Π_S, and we return to step 1.
Iteration 2
Step 1_2 We now reconstruct the maximal witness w.r.t. the predicates $\{a = 0, x = 0, \mathtt{e} = 0\}$. The result W_2 is shown in Figure 1(D).

Step 2_2 Upon calling Check to see if this witness is permissive, we get the witness counterexample `acq`; `write`, in the form of its CFD.

Step 3_2 This time, we have a CxPerm counterexample, as the witness counterexample sequence is indeed not permitted, the new predicate ($x = 0$) returned by Refine is added to Π_P, and we return to step 1. The new Π_P suffices to show W_2 is permissive, as the test in line 3 of Check succeeds. The safe, permissive witness W_2 is returned. □

4 Tightness and Full Interfaces

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

Example 14: Consider the modified version of the function `acq` shown in the left of Figure 3(C), and the program P_4 obtained by using the modified function and adding the static variable `flag`. This time, `acq` can nondeterministically fail, but failure can occur at most once. Algorithm BuildInterface produces exactly the same witness graph W_3 (Figure 3(B)) for P_4. However, there are correct clients, *e.g.,* the client shown on the right in Figure 3(C), which cause false alarms

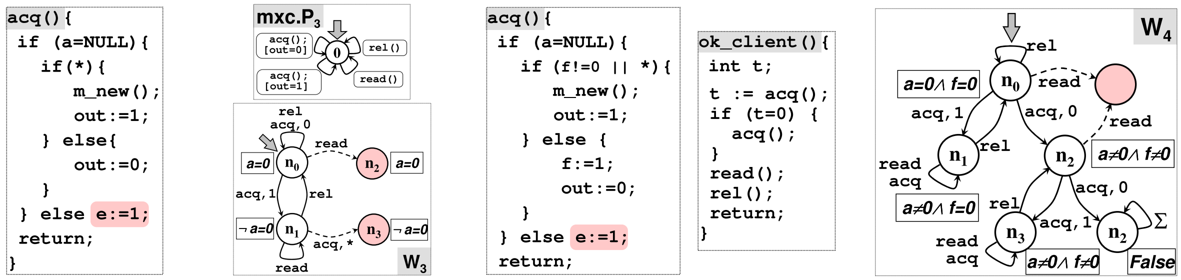

Figure 3: (A) P_3.acq (B) mxc.P_3 ($\uparrow$), Perm. W_3 ($\downarrow$) (C) P_4.acq ($\leftarrow$), Correct Client ($\rightarrow$) (D) Perm. W_4

when analyzed against this interface. The false alarms arise from the fact that the witness graph is unaware that `acq` can never "fail" twice. Once `acq` fails (*i.e.,* returns 0), then when called again, it will "succeed" (return 1), and thus thereafter, `read` is permitted. The problem is not that the client does not check the output of `acq` the second time, but that the client generates the sequence $(\texttt{acq}, 0); (\texttt{acq}, 0); \texttt{read};$ which, while permitted, is not accepted by W_3. The sequence $(\texttt{acq}, 0); (\texttt{acq}, 0); \texttt{read};$ is not in $\mathcal{R}.P$, and so trivially, cannot lead to an error. Hence it is permitted, and this sequence is in $\mathcal{I}.P$. A permissive labeling on W, however, looks only for realzable sequences precluded by W. As W_3 permits all feasible sequences, it passes the check. $\square$

In order to ensure that the language of a witness graph is the full interface $\mathcal{I}.P$, we must constrain its region labelings further. A region labeling ρ is *tight* if: (P5) For every $\mathtt{n} \xrightarrow{(f,o)} \mathtt{n}'$ we have either $\rho.\mathtt{n}' = \emptyset$ or $\rho.\mathtt{n} \subseteq \mathsf{Pre}.(f, o).(\rho.\mathtt{n}')$.

Proposition 6 *For an open program P and witness graph W, if there exists a safe labeling ρ_1, a permissive labeling ρ_2 and a tight labeling ρ_3 of W then $\mathcal{L}.W = \mathcal{I}.P$.*

To generate a witness for the full interface, we add a third phase to our algorithm that 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 Tighten. Procedure 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 $\mathtt{n} \overset{(f,o)}{\to} \mathtt{n}'$, it checks if the labeling ρ is tight, *i.e.,* if $\rho.\mathtt{n} \subseteq \mathsf{Pre}.(f, o).(\rho.\mathtt{n})$. If not, it finds a set Π' of predicates such that there is a region r definable with predicates in Π' for which $\rho.\mathtt{n} \wedge r \neq \emptyset$ but $\rho.\mathtt{n} \wedge r \wedge \mathsf{WP}.(f, o).(\rho.\mathtt{n}') = \emptyset$. Unfortunately, since f may have infinitely many paths, we cannot directly compute $\mathsf{Pre}.(f, o).(\rho.\mathtt{n}')$ by iterating the WP operator. Instead, we modify the Algorithm BuildARG. We omit the details of this algorithm for lack of space. Note tightness is not the same as termination, which requires that a function will terminate along all paths. Here, we only need to find if there is some path to $\rho.\mathtt{n}'$.

Example 15: For the program P_4 (that uses the modified `acq` in Figure 3(C)), the witness W_3 in Figure 3(B) is not tight. In particular, the edge $\mathtt{n}_0 \xrightarrow{\texttt{acq},0} \mathtt{n}_0$ is not tight. Procedure Tighten finds the extra predicate $f = 0$; indeed, from every state in $(a = 0 \wedge f = 0)$, there is a feasible path in `acq` to a state in $a = 0$. When we track this additional predicate, Algorithm BuildInterface returns the witness graph W_4 (Figure 3(D). This witness graph admits a safe, permissive, and tight labeling (as shown in Figure 3(D)). Thus it defines the full interface for P_4. $\square$

Procedure BuildInterface$'$ adds this tighten phase to BuildInterface by checking algorithm Tighten and re-running the BuildInterface loop with the additional predicates should tighten fail.

Theorem 2 *Let P be an open program and let Π_1, Π_2 be two sets of predicates. (1) If* BuildInterface$'.P.\Pi_1.\Pi_2$ *terminates and returns W then $\mathcal{L}.W = \mathcal{I}.P$. (2) Procedure* BuildInterface$'$ *terminates if P is finite-state.*

5 Experiences

Implementation. We have implemented the algorithm to generate permissive interface witnesses in the Blast software model checker. In order to be practical, we implement some optimizations over the described method. First, while it is conceptually simpler to explain the counterexample analysis phase separately from the reachability phase, in practice, we integrate counterexample guided refinement within the reachability procedure (using lazy abstraction) to rule out infeasible abstract counterexamples. Second, we implement a value flow analysis, similar to field splitting [15] to detect which global variables are read and written by which library methods, and partition the set of all library methods to those that are connected by sharing internal state. Third, as suggested by Example 3.1, we keep the upper and lower bound checks separate.

Experiences. We ran our implementation on the examples from [1], on the code available from http://www.cis.upenn.edu/jist/examples.html. We looked at the classes `Signature`, `ServerTableEntry`, and `ListItr` from this web page, and additionally classes `Socket` from JDK1.4. Our algorithm, unlike the algorithm of [1], does not require the user to provide a fixed predicate abstraction for an open program.

For `Signature`, the exception `SignatureException` was marked as the error condition. For `ServerTableEntry` we considered the exception `INTERNAL` that is raised when the state machine maintained by the class is in a "wrong" state. For `ListItr` we consider `IllegalStateException`. In addition, we ran the tool on the class `Socket` of JDK1.4, where we considered the exception `SocketException`. In each case, the tool was able to construct a permissive and tight witness within 30s (on a 3GHz machine with 512M RAM).

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

In `Socket`, we consider public methods `connect`, `close`, `bind`, `getInputStream`, `getOutputStream`, `shutdownInput`, `shutdownOutput`. The value flow finds out that we need to define witnesses for the functions `connect`, `close`, `shutdownInput`, and `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 `getInputStream` can be called only after a call to `connect`, but when `close` or `shutdownInput` has not been called. We require six predicates that keep track of the internal state.

6 Applications: Modular Program Analysis

We now describe some applications of safe, permissive (and full) interfaces in modular program analysis.

Compositional Verification. For an open program P, and any client Cl for P, instead of analyzing the P with the client, we can use the (smaller) *interface program* constructed from a witness for $\mathcal{I}.P$ to verify the client. A state $s \circ t$ of a closed program (Cl, P) is *safe* if $t \notin P.\mathcal{E}$. The program (Cl, P) is *safe* if all its reachable states are safe. A client Cl is safe w.r.t. to P if (Cl, P) is safe.

Given a witness graph W for P we construct an open program $\mathsf{IntfP}.W$ as follows. There are three static variables `state` (whose values range over the set of states $W.N$ of W), `out` (whose values range over $P.Outs$), and a variable `err`. The signature is $P.\Sigma$. For each $(f, o) \in P.\Sigma$, there is a function $(\mathsf{IntfP}.W).f \in (\mathsf{IntfP}.W).F$ that encodes the edge relation of W as follows. There is a branch in $(\mathsf{IntfP}.W).f$ for each edge $\mathtt{n} \xrightarrow{(f,o)} \mathtt{n}'$ in W. On this branch, the CFA checks that `state` = `n` (by `assume` [`state` = `n`]), then sets `state` to $\mathtt{n}'$, `out` to o, and `e` to 1 if $\mathtt{n}'$ is unsafe. In the initial state $(\mathsf{IntfP}.W).s_0$ the variable `state` equals the root node $W.\mathtt{n}_0$, and `out`, `e` are 0. The set of outputs $(\mathsf{IntfP}.W).Outs$ is $P.Outs$. The error region $(\mathsf{IntfP}.W).\mathcal{E}$ is `e` = 1. It is easy to see that $\mathcal{I}.(\mathsf{IntfP}.W) = \mathcal{L}.W$. When verifying that the client uses a 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.

Theorem 3 *Let P be an open program, W be a witness for P and $\mathsf{IntfP}.W$ be the interface program of W. For any client Cl: (1) If $\mathcal{L}.W$ is a safe interface then (Cl, P) is safe if $(Cl, \mathsf{IntfP}.W)$ is safe. (2) If $\mathcal{L}.W$ is the full interface for P, then (Cl, P) is safe iff $(Cl, \mathsf{IntfP}.W)$ is safe.*

Interface Decomposition. The internal invariants of an open program may be violated in several different and independent ways. If we consider all possible error states together, the synthesized interface can be quite large and complex. The following proposition lets us decompose the interface construction by decomposing the sets of error states.

Proposition 7 *Let P be an open program, and let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two subsets of $\mathsf{V}.(P.X)$. Let $\mathcal{I}_i$ be a full (resp. permissive, safe) interface for P when $P.\mathcal{E} = \mathcal{E}_i$ for $i \in \{1, 2\}$. Then $\mathcal{I}_1 \cap \mathcal{I}_2$ is a full (resp. permissive, safe) interface for P when $P.\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$.*

Given an open program P such that $P.\mathcal{E} = \cup_i \mathcal{E}_i$, we can now find separate witnesses for each $\mathcal{E}_i$, and then check the client against each witness independently. The client correctly uses P iff all the checks succeed. Checking against each witness program separately is more efficient that checking against a single (product) witness.

Proposition 8 *Let P be an open program, W_i a witness for $P[\mathcal{E} \mapsto \mathcal{E}_i]$, and $\mathsf{IntfP}.W_i$ the interface program of W_i. For any client Cl: (1) $(Cl, P[\mathcal{E} \mapsto \cup_i \mathcal{E}_i])$ is safe $(Cl, \mathsf{IntfP}.W_i)$ is safe if for all i. (2) If each W_i is permissive and tight for $P[\mathcal{E} \mapsto \mathcal{E}_i]$ then $(Cl, P[\mathcal{E} \mapsto \cup_i \mathcal{E}_i])$ is safe iff $(Cl, \mathsf{IntfP}.W_i)$ is safe for all i.*

References

[1] R. Alur, P. Cerny, G. Gupta, and P. Madhusudan. Synthesis of interface specifications for Java classes. POPL, 2005.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL 02: Principles of Programming Languages*, pages 4–16. ACM, 2002.

[3] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer, 2000.

[5] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.

[6] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[7] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *PLDI 99: Programming Languages Design and Implementation*, pages 192–203. ACM, 1999.

[9] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.

[10] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer, 1997.

[11] M. Harder, J. Mellen, and M.D. Ernst. Improving test suites via operational abstraction. In *ICSE 03: Software Engineering*, pages 60–73. ACM, 2003.

[12] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.

[13] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[14] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[15] J. Whaley, M.C. Martin, and M.S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA 02: Software Testing and Analysis*, pages 218–228. ACM, 2002.