Permissive Interfaces[*]

Thomas A. Henzinger
EPFL and UC Berkeley
tah@epfl.ch

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Rupak Majumdar
UC Los Angeles
rupak@cs.ucsd.edu

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 $\{\texttt{acq}, \texttt{read}, \texttt{rel}\}$, 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* [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 $(\texttt{acq}; \texttt{read}^*; \texttt{rel})^*$ 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

[*]Supported in part by the ONR grant N00014-02-1-0671 and the NSF grants CCR-0234690, CCR-0225610, and ITR-0326577.

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 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 [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

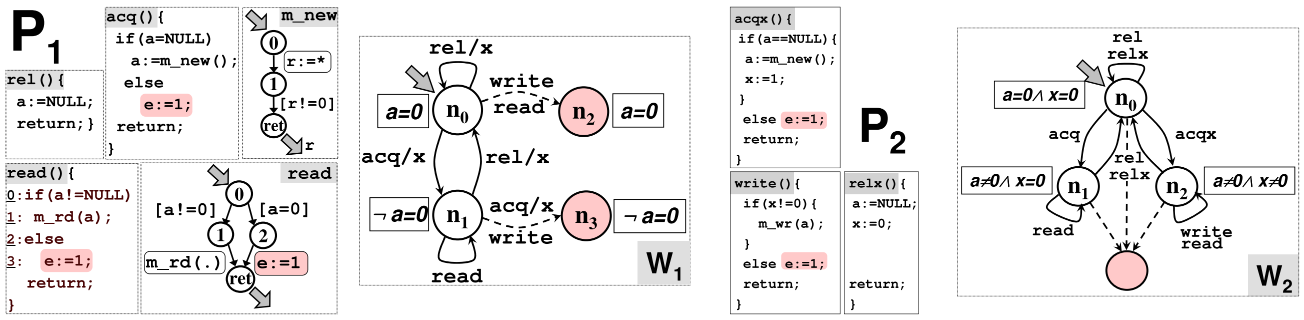
2.1 Open Programs

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

Syntax. For a set X of variables, $\mathsf{Exp}.X$ is the set of arithmetic expressions over the variables X, and $\mathsf{Pred}.X$ is the set of boolean expressions (arithmetic comparisons) over X, the set $\mathsf{V}.X$ is the set of valuations to X, and $\mathsf{Op}.X$ 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 operation to be executed, and (3) function calls $f(\mathtt{x})$, 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 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 $\rightarrow \subseteq Q \times \mathsf{Op}.(X_L \cup X_S) \times Q$. 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 (integer- or boolean-valued), 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 Outs of outputs, (4) an initial state $s_0 \in \mathsf{V}.X$, (5) a set $\mathcal{E} \subseteq \mathsf{V}.X$ of error states given as a predicate in $\mathsf{Pred}.X$, and, (6) a signature Σ, which is a subset of $F \times \mathit{Outs}$. The signature Σ represents the externally visible function names and output values. An



```
rel(){
 a:=NULL;
 return;}
```

```
acq(){
 if(a=NULL)
   a:=m_new();
  else
    e:=1;
 return;
}
```

```
read(){
0:if(a!=NULL)
1: m_rd(a);
2:else
3:  e:=1;
  return;
}
```

```
acqx(){
 if(a==NULL){
  a:=m_new();
  x:=1;
 }
 else e:=1;
 return;
}
```

```
write(){
 if(x!=0){
   m_wr(a);
 }
 else e:=1;
 return;
}
```

```
relx(){
 a:=NULL;
 x:=0;


return;
}
```

Figure 1: (A) P_1 **(B) safe interface witness** W_1 **(C)** P_2 **(D) safe and permissive interface witness** W_2

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 $P_1.X = \{\mathtt{a}, \mathtt{e}\}$. 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 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: $s_1.\mathtt{x} = s_2.\mathtt{x}$. 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 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 $(q_e, s') = (q_n, s_n)$ and for each $1 \leq i \leq n-1$, we have $q_i \xrightarrow{\mathsf{op}_i} q_{i+1}$ 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 `op` is `assume` p, then $s \models p$ and $s'.y = s.y$ for all $y \in X$; and (2) if `op` is $\mathtt{x} := \mathtt{e}$, then $s'.x = s.e$, and $s'.y = s.y$ for all $y \in X \setminus \{x\}$; and (3) if `op` is $\mathtt{f}()$, then $s.\mathtt{y} = s'.\mathtt{y}$ for all $\mathtt{y} \in X_L$, and there exist $t, t' \in \mathsf{V}.X_S$ such that $t \overset{f.C}{\rightsquigarrow} t'$ 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 $\mathsf{sp}.r.\mathsf{op}$, 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. `op`, written $\mathsf{wp}.r.\mathsf{op}$ is the set $\{s \in \mathsf{V}.X \mid \exists s' \in r : s \overset{\mathsf{op}}{\rightsquigarrow} s'\}$. 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 $\mathsf{Pred}.X$, the above can be defined as predicate transformers [6].

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 $s \overset{f.C}{\rightsquigarrow} s'$ 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 of Figure 1(A), the initial states are given by $a = 0 \wedge \mathtt{e} = 0$, and the 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 $\mathsf{Post}.r.\mathtt{acq}()$ is also $a \neq 0$. □

2.2 Interfaces

A set r of states is safe w.r.t. $\mathcal{E}$ if $r \subseteq \neg\mathcal{E}$. For an open program P, a sequence $\sigma \in P.\Sigma^*$ is $\mathcal{E}$-safe from a set r of states 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 $\mathcal{I}.P$ 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 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 not empty and (2) σ' is realizable from r'. A sequence is *realizable* if it is realizeable from $P.s_0$. We write $\mathcal{R}.P$ 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.,* $I \subseteq \mathcal{I}.P$.
(2) A permissive *interface if it contains every legal realizable sequence, i.e.,* $\mathcal{I}.P \cap \mathcal{R}.P \subseteq I$.
(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 `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 $\mathtt{acq} \cdot \mathtt{read}^*$ 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 $L_1 = ((\mathtt{acq} \cdot \mathtt{read}^* \cdot \mathtt{rel})^* \cdot \mathtt{rel}^*)^*$.

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

2.3 Witnesses

Witness graphs. We focus on interfaces corresponding to regular languages, which are naturally witnessed by finite state graphs. A *witness graph* $W = (N, E, \texttt{n}_0)$ 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 $\texttt{n}_0 \in N$. We write $\texttt{n} \xrightarrow{(f,o)} \texttt{n}'$ if $(\texttt{n}, (f, o), \texttt{n}') \in E$, and call $\texttt{n}'$ an (f, o)-successor of $\texttt{n}$. Additionally, we require that every $\texttt{n} \in N$ 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 $\texttt{n} \in N^+$ if σ is the empty sequence, or $\sigma \equiv (f, o) \cdot \sigma'$ and $\texttt{n}$ has an (f, o)-successor $\texttt{n}' \in N^+$ such that σ' is accepted from $\texttt{n}'$. 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 $\texttt{n}_0$.

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 $((\texttt{acq} \cdot \texttt{read}^* \cdot \texttt{rel})^* \cdot \texttt{rel}^*)^*$, 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) $P.s_0 \in \rho.\texttt{n}_0$, *i.e.,* the root $\texttt{n}_0$ label contains the initial state; and (P2) for every $\texttt{n} \xrightarrow{(f,o)} \texttt{n}'$, we have $\mathsf{Post}.(\rho.\texttt{n}).(f, o) \subseteq \rho.\texttt{n}'$. A region labeling represents an overapproximation of the behavior of an open program: for every node $\texttt{n} \in N^+$, and every sequence of operations $\sigma \in P.\Sigma^*$ marking a path from the root $\texttt{n}_0$ to $\texttt{n}$ in G, we have $\mathsf{Post}.(\rho.\texttt{n}_0).\sigma \subseteq \rho.\texttt{n}$.

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

The existence of a *safe labeling* for W, namely one that satisfies (P3), guarantees that the language of W is bounded above by $\mathcal{I}.P$, *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 `acqx`, `relx`, `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 $\texttt{e} = 0$, and for the unsafe (shaded) nodes, the label is $\texttt{e} = 1$. We can check that this is a region labeling; *e.g.,* for the edge $\texttt{n}_0 \xrightarrow{\texttt{acq}} \texttt{n}_1$ we saw that $\mathsf{Post}.(a = 0).\texttt{acq}$ equals $a \neq 0$, *i.e.,* the successor states from the label of $\texttt{n}_0$ are contained in the label of $\texttt{n}_1$. 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 $\texttt{n} \in N^-$ we have $\rho.\texttt{n} \subseteq P.\mathcal{E}$, 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 $\mathcal{L}.W_1$ 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 $[\texttt{n}_0 \mapsto (a = 0); \texttt{n}_1 \mapsto a \neq 0; \texttt{n}_2 \mapsto (a \neq 0 \wedge x \neq 0)]$, where again $\texttt{e} = 0$ and $\texttt{e} = 1$ are implicit for the unshaded and shaded nodes, respectively. It also has the permissive labeling $[\texttt{n}_0 \mapsto (a = 0 \wedge x = 0); \texttt{n}_1 \mapsto (a \neq 0 \wedge x = 0); \texttt{n}_2 \mapsto a \neq 0]$, 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 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 ($\neg P.\mathcal{E}$). 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 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 intersect $P.\mathcal{E}$ 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 $Cl = (X, \emptyset, Q, q_0, q_e, \rightarrow)$ 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 closed 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 $(s \circ t) \overset{\mathsf{op}}{\rightsquigarrow} (s' \circ t')$ if (1) $s \overset{\mathsf{op}}{\rightsquigarrow} s'$ and $t = t'$ if 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 op is the function call $y := f()$. For a subset $L \subseteq Q$ of the client locations, we say that $(s \circ t) \in \mathsf{V}.(Cl.X \cup P.X)$ 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 $Cl.Q \times \mathsf{V}.(Cl.X \cup P.X)$ such that (1) $t_0 = P.s_0$ (the initial state of the open program), (2) for all $0 \le i \le n-1$, there is an edge $q_i \xrightarrow{\mathsf{op}_i} q_{i+1}$ in Cl such that $(s_i \circ t_i) \overset{\mathsf{op}_i}{\rightsquigarrow} (s_{i+1} \circ t_{i+1})$, and (3) $s_n = s$, $t_n = t$, and $q_n \in L$.

Algorithm 1 BuildARG

Input: closed program (Cl, P), predicates Π.
Output: ARG A of (Cl, P) w.r.t. Π.

```
1: $L := \{\mathtt{n} : (Cl.pc_0, \mathsf{Abs}.\Pi.(P.s_0))\}$
2: $seen := \emptyset$; $A := \emptyset$
3: while $L \neq \emptyset$ do
4:   pick and remove state $\mathtt{n} : (pc, r)$ from $L$
5:   if $\mathtt{n} \notin seen$ then
6:     $seen := seen \cup \{\mathtt{n}\}$
7:     for each $(pc, \mathsf{op}, pc')$ of $Cl$ do
8:       $r' := \mathsf{SP}.\Pi.\mathsf{op}.r$
9:       $\mathtt{n}' := \mathsf{Connect}.A.((pc, r), \mathsf{op}, (pc', r'))$
10:      $L := L \cup \{\mathtt{n}'\}$
11: return $A$
```

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 $v \xrightarrow{\mathsf{op}} v'$ in $\mathsf{Client}.W$, where 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 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. The language of W is 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 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 $\Pi \subseteq \mathsf{Pred}.X$ of predicates 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 op w.r.t. Π, written $\mathsf{SP}_\Pi.r.\mathsf{op}$, is a boolean combination of predicates from Π which overapproximates $\mathtt{sp}.r.\mathsf{op}$. The procedure SP computes the abstract postcondition. It takes as input the set Π of predicates, an operation op, and the current region r, and returns $\mathsf{SP}_\Pi.r.\mathsf{op}$. 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, 12].

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

Algorithm 2 Check

```
Input: open program $P$, witness $W = (V, E, v_0)$.
Input: predicates $\Pi$, vertices $V' \subseteq V$, target region $\mathcal{E}$.
Output: TRUE or a counterexample CFD $\textsc{Ctrx}(\delta)$.
1: $A := \mathsf{BuildARG}.P.(\mathsf{Client}.W).\Pi$
2: $\rho := \lambda v.(\bigvee_{\mathtt{n}:(v,r)} r)$
3: if exists $v \in V'$ s.t. $\rho.v \not\subseteq \mathcal{E}$ 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
```

initial state s_0 of the system w.r.t. Π. (2) Each node $\mathtt{n}$ labeled (pc, r) has an op-successor (pc', r') for every edge $pc \xrightarrow{\mathsf{op}} pc'$ in Cl, such that $r' = \mathsf{SP}_{\Pi}.r.\mathsf{op}$. If Π is finite, then the ARG is also finite. The procedure 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. The algorithm BuildARG computes the ARG in Figure 1(B) for $(\mathsf{Client}.W_1, P_1)$ w.r.t the set of predicates $\Pi_1 = \{\mathtt{e} = 0, \mathtt{a} = 0\}$. □

The Witness Checking Algorithm. Notice that from the ARG A constructed by BuildARG, we can construct a region labeling for W by mapping each 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 a witness graph for open program P. Let A be the ARG for $(\mathsf{Client}.W, P)$ w.r.t. a set of predicates Π. Then the map $\rho_G = \lambda v.(\bigvee_{\mathtt{n}:(v,r)} r)$ is a region labeling for W, i.e., ρ_G has the properties* (P1) *and* (P2).

Given a set of predicates Π_S, we invoke the procedure BuildARG to compute an ARG A_S for $(\mathsf{Client}.W, P)$ 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 the algorithm $\mathsf{Check}.P.W.\Pi_S.V^+.(\neg P.\mathcal{E})$ 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 $(\mathsf{Client}.W, P)$, 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 the algorithm $\mathsf{Check}.P.W.\Pi_P.V^-.(P.\mathcal{E})$, shown in Algorithm 2, which returns TRUE if 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: instead of the traditional use of overapproximation —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.

Algorithm 3 ReconstructMax

```
Input: open program $P$, predicates $\Pi$.
Output: maximally safe witness graph $W$ for $P$.
1: $A := \mathsf{BuildARG}.(\mathsf{mxc}.P, P).\Pi$
2: $V^+ := \{v \mid v : (\cdot, r) \in A.N$ s.t. $r \subseteq \neg P.\mathcal{E}\}$
3: $V^- := A.N \setminus V^+$
4: return witness graph $(V^+ \cup V^-, A.E, A.\mathtt{n}_0)$
```

Proposition 4. **[Witness checking]** *For an open program P, let $W = (V^+ \cup V^-, E, v_0)$ be a witness graph such that for two predicate sets Π_S and Π_P, both $\mathsf{Check}.P.W.\Pi_S.V^+.(\neg P.\mathcal{E})$ and $\mathsf{Check}.P.W.\Pi_P.V^-.(P.\mathcal{E})$ return* TRUE. *Then $\mathcal{I}.P \cap \mathcal{R}.P \subseteq \mathcal{L}.W \subseteq \mathcal{I}.P$, 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 $\mathtt{x}$, $\mathtt{y}$, and $\mathtt{e}$, all initially 0, and the two methods f_1 and f_2:

f_1 : $\mathtt{if}\ \mathtt{x} \neq 0\ \mathtt{then}\ \mathtt{e} := 1;$
f_2 : $\mathtt{if}\ \mathtt{y} = 0\ \mathtt{then}\ \mathtt{e} := 1;$

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 $\rho.\mathtt{n}_1 = True$ 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. $\mathtt{x} = 0$). □

3.2 Witness Reconstruction

In witness checking, we assumed that in addition to the abstraction predicates Π_S and 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 show (using Π_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* $\mathsf{mxc}.P$ 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 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 $\mathsf{mxc}.P_2$ shown on the top in Figure 2(A). □

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

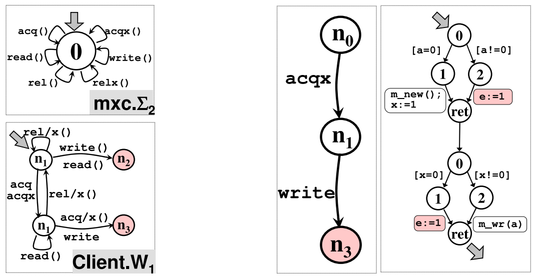


Figure 2: (A) mxc.P_2 (↑) Client.W_1 (↓) **(B) witness counterexample (C) counterex. control-flow dag**

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 Π_S. The above algorithm is formalized by the procedure ReconstructMax shown in Algorithm 3.

PROPOSITION 5. **[Maximal witness reconstruction]** *For every open program P and set Π of predicates, the procedure* 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* 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 the 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 1(B), which translates to the witness W_1. The unshaded nodes are those whose regions are contained in $\neg P.\mathcal{E}$, *i.e.,* $\mathtt{e} \neq 1$; they are 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 `acqx`. When computing the `acqx`-successor of $\mathtt{n}_0$, the abstract state is the same as that of $\mathtt{n}_1$, namely $\neg(\mathtt{a} = 0) \wedge (\mathtt{e} = 0)$, as there are no predicates on `x`. Hence the algorithm sets $\mathtt{n}_1$ to be the `acqx`-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. In verification, coarse abstractions leads to false positives, for interface synthesis, coarse abstractions lead to massively constrained interfaces. As shown in the prequel, 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 and Π_P by automatically refining coarse abstractions, using *witness counterexample* sequences of $P.\Sigma$ which are prohibited by the maximal witness, but which may be permitted by the open program P.

Witness counterexamples. We use the 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. The 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) the resulting state of the open program is legal, *i.e.,* $r \not\subseteq P.\mathcal{E}$; 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 $\mathtt{n}$. The ARG edge labels along this path correspond to a call sequence σ that may be legal, as the resulting abstract region is not contained in $P.\mathcal{E}$ 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 BuildARG (Algorithm 1), we compute $\mathsf{SP}_\Pi.r.\mathsf{op}$, where $\mathsf{op} = (f, o)$, in the standard way [10, 12] by unrolling the CFA for f, computing the abstract state over the predicates Π 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 $\mathsf{dag}_\Pi.\mathsf{op}$. Given a sequence of edges labeled σ in the ARG built by BuildARG, we 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 the 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 Example 10. The witness client 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 Check, using the predicates $\Pi_P = \{\mathtt{e} = 0, a = 0\}$ and the target states $\neg(\mathtt{e} = 1)$, 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 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 Π_P. The sequence of calls labeling the edges `acqx`; `write` is a witness counterexample, for which Figure 2(C) shows the CFD. □

Algorithm 4 Refine

```
Input: open program $P$, CFD $\delta$.
Output: CxPerm($\Pi$) or CxSafe($\Pi$), where $\Pi$ is a set of
    predicates.
1: $\varphi$ := sp.$(P.s_0).\delta$
2: if $\varphi \wedge \neg(P.\mathcal{E})$ is satisfiable then
3:    $\Pi$ := GetNewPreds.$\delta.(P.s_0).(\neg P.\mathcal{E})$
4:    return CxPerm($\Pi$)
5: else
6:    $\Pi$ := GetNewPreds.$\delta.(P.s_0).(P.\mathcal{E})$
      {note: $\varphi \wedge P.\mathcal{E}$ is unsatisfiable}
7:    return CxSafe($\Pi$)
```

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, we have two cases:

(CxPerm) The condition $\neg P.\mathcal{E} \cap \mathsf{sp}.(P.s_0).\delta$ is unsatisfiable, implying that σ is not a permitted call sequence. In this case, we use a standard predicate discovery algorithm [11] 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 the witness counterexample.

(CxSafe) The condition $\neg P.\mathcal{E} \cap \mathsf{sp}.(P.s_0).\delta$ 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 by the procedure Refine shown in Algorithm 4. The procedure GetNewPreds refers to the predicate discovery algorithm, which takes as input a CFD δ, and a set r of intial states, and a set $\mathcal{E}$ of states such that $\mathsf{sp}.r.\delta \cap \mathcal{E}$ is unsatisfiable, and returns a set Π of predicates such that $\mathsf{SP}_\Pi.r.\delta \cap \mathcal{E}$ 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 a 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(C). 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 the case CxSafe, and the predicate discovery algorithm finds the new predicate $x = 0$. The resulting abstraction is precise enough to permit the sequence `acqx`; `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 the version of `acq` shown in Figure 3(A), which 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`; `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. □

Algorithm 5 BuildInterface

```
Input: open program $P$, predicates $\Pi_S$ and $\Pi_P$.
Output: safe and permissive witness graph $W$ for $P$.
1: Step 1: $W :=$ ReconstructMax.$P.\Pi_S$
2: Step 2: $p :=$ Check.$P.W.\Pi_P.(W.V^-).(P.\mathcal{E})$
3: if $p =$ True then
4:   return $W$
5: Step 3: {$W$ is not permissive; $p$ is a witness ctrx.}
6: Ctrx($\delta$) := $p$
7: match Refine.$P.\delta$ with
8:   | CxPerm($\Pi$) $\rightarrow \Pi_P := \Pi_P \cup \Pi$; go to Step 2
9:   | CxSafe($\Pi$) $\rightarrow \Pi_S := \Pi_S \cup \Pi$; go to Step 1
```

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

Step 1. We start with (possibly trivial) abstractions Π_S and Π_P, and use the witness reconstruction algorithm to obtain a candidate witness W, the 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 and 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, then 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 the imprecision of Π_S, *i.e.,* it is a CxSafe witness counterexample, then we infer new predicates Π'_S such that the resulting maximal witness permits the sequence σ, and go to Step 1.

Theorem 1. *Let P be an open program; let Π_1 and Π_2 be two sets of predicates. (1) If* BuildInterface.$P.\Pi_1.\Pi_2$ *terminates and returns W, then $\mathcal{I}.P \cap \mathcal{R}.P \subseteq \mathcal{L}.W \subseteq \mathcal{I}.P$. (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 $\Pi_S = \Pi_P = \{\mathtt{e} = 0, a = 0\}$. 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 mxc.P_2 using the 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 permissiveness check returns the witness counterexample `acqx`; `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 not permitted, the new predicate $x = 0$ returned by Refine is added to Π_P, and we return to Step 1. The new predicates Π_P suffice to show that W_2 is permissive, as the test in line 3 of Check succeeds. The safe and permissive witness W_2 is returned. $\Box$

4. TIGHTNESS AND FULL INTERFACES

As we saw in Section 2, a safe and a permissive labeling ensures that the witness language $\mathcal{L}.W$ is a safe and 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 `f`. This time, `acq` can nondeterministically fail, but failure can occur at most once. The 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 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 $(\mathtt{acq}, 0); (\mathtt{acq}, 0); \mathtt{read}$ which, while permitted, is not accepted by W_3. The sequence $(\mathtt{acq}, 0); (\mathtt{acq}, 0); \mathtt{read}$ 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 realizable sequences precluded by W. As W_3 permits all feasible sequences, it passes the check. $\Box$

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, then $\mathcal{L}.W = \mathcal{I}.P$.*

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 Tighten. The 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, the procedure Tighten constructs successive underapproximations of Pre in the following way.

The algorithm maintains a finite set of candidate paths in f as a dag δ. An underapproximation of $\mathsf{Pre}.(\rho.\mathtt{n}').(f, o)$ is computed by computing WP over this dag δ. The paths in δ certify the tightness condition for all states in $\rho.\mathtt{n} \wedge \mathsf{WP}.(\rho.\mathtt{n}').\delta$ in that for every state s in this set, there is some path in *paths* that takes s to a state in $\rho.\mathtt{n}'$. Let $r' = \rho.\mathtt{n} \wedge \neg\mathsf{WP}.(\rho.\mathtt{n}').\delta$ denote the complement of the above set. If r' is empty, then all states satisfying $\rho.\mathtt{n}$ can reach some state in $\rho.\mathtt{n}'$ 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 $\rho.\mathtt{n}'$, or the dag δ 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 $\rho.\mathtt{n}'$ on the call (f, o). This is an abstract reachability question, which either returns an abstract counterexample dag, or that no state in $\rho.\mathtt{n}'$ is reachable. In the first case, we check if the abstract counterexample dag is genuine. Genuine counterexample dags are added to δ, and we repeat the loop. Spurious counterexamples yield predicates that are added to Π' and the loop is repeated. In the second case, no state in r' can reach the target $\rho.\mathtt{n}' \wedge \mathtt{out} = o$ when we track the predicates in Π'. The predicates Π' allow us to break $\rho.\mathtt{n}$ into states that can reach $\rho.\mathtt{n}'$ 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 $\rho.\mathtt{n}'$.

Example 15: For the program P_4, which uses the modified `acq` of Figure 3(C), the witness W_3 in Figure 3(B) is not tight. In particular, the edge $\mathtt{n}_0 \xrightarrow{\mathtt{acq},0} \mathtt{n}_0$ is not tight. The 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, the algorithm 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. $\Box$

The procedure BuildInterface$'$ results from iterating BuildInterface using the additional predicates supplied by the algorithm Tighten, until Tighten confirms that the labeling is tight.

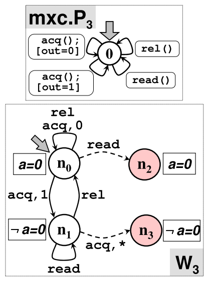
Theorem 2. *Let P be an open program and let Π_1, Π_2 be two sets of predicates. (1) If* BuildInterface$'.P.\Pi_1.\Pi_2$ *terminates and returns W, then $\mathcal{L}.W = \mathcal{I}.P$. (2) The procedure* BuildInterface$'$ *terminates if P is finite-state.*

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 `Signature`, the exception `SignatureException` was marked as the error condition. For `ServerTableEntry` we

```
acq(){
 if (a=NULL){
   if(*){
      m_new();
      out:=1;
   } else{
      out:=0;
   }
 } else e:=1;
 return;
}
```



```
acq(){
 if (a=NULL){
   if (f!=0 || *){
      m_new();
      out:=1;
   } else {
      f:=1;
      out:=0;
   }
 } else e:=1;
 return;
```

```
ok_client(){
 int t;
 t := acq();
 if (t=0) {
   acq();
 }
 read();
 rel();
 return;
}
```

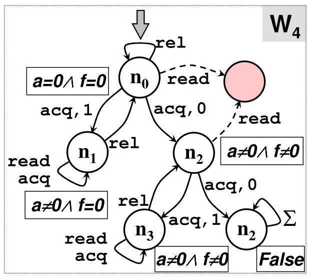


Figure 3: (A) P_3.acq **(B)** mxc.P_3 (↑), **permissive** W_3 (↓) **(C)** P_4.acq (←), **correct client** (→) **(D) permissive** W_4

considered the exception INTERNAL, which is raised when the state machine maintained by the class is in a "wrong" state. For ListItr, we considered IllegalStateException. In addition, we ran the tool on the class Socket of JDK1.4, where we considered the exception SocketException.

In all of these examples, the class maintains 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, which specifies how this object should be used.

In Socket, we considered the public methods connect, close, bind, getInputStream, getOutputStream, shutdownInput, and 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. 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 *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. 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 $= \mathtt{n}$ (by assume [state $= \mathtt{n}$]), 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 and e are 0. The set $(\mathsf{IntfP}.W).Outs$ of outputs 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 every 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.*

7. REFERENCES

[1] R. Alur, P. Cerny, G. Gupta, and P. Madhusudan. Synthesis of interface specifications for Java classes. In *POPL*, pp. 98–109. ACM 2005.

[2] G. Ammons, R. Bodik, and J.R. Larus. Mining specifications. In *POPL*, pp. 4–16. ACM 2002.

[3] J. Cobleigh, D. Giannakopoulou, and C.S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, LNCS 2619, pp. 331–346. Springer 2003.

[4] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pp. 57–68. ACM 2002.

[5] L. de Alfaro and T.A. Henzinger. Interface automata. In *FSE*, pp. 109–120. ACM 2001.

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

[7] M.A. Fahndrich and R. DeLine. Typestates for objects. In *ECOOP*, LNCS 3086, pp. 465–490. Springer 2004.

[8] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pp. 1–12. ACM 2002.

[9] D. Giannakopoulou, C.S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, pp. 3–12. IEEE 2002.

[10] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, LNCS 1254, pp. 72–83. Springer 1997.

[11] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL*, pp. 232–244. ACM 2004.

[12] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pp. 58–70. ACM 2002.

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

[14] J. Whaley, M.C. Martin, and M.S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, pp. 218–228. ACM 2002.