Algorithms for Interface Synthesis[⋆]

(Invited Tutorial)

Dirk Beyer[1], Thomas A. Henzinger[2], and Vasu Singh[2]

[1] Simon Fraser University, B.C., Canada
[2] EPFL, Switzerland

Abstract. A temporal interface for a software component is a finite automaton that specifies the legal sequences of calls to functions that are provided by the component. We compare and evaluate three different algorithms for automatically extracting temporal interfaces from program code: (1) a *game* algorithm that computes the interface as a representation of the most general environment strategy to avoid a safety violation; (2) a *learning* algorithm that repeatedly queries the program to construct the minimal interface automaton; and (3) a *CEGAR* algorithm that iteratively refines an abstract interface hypothesis by adding relevant program variables. For comparison purposes, we present and implement the three algorithms in a unifying formal setting. While the three algorithms compute the same output and have similar worst-case complexities, their actual running times may differ considerably for a given input program. On the theoretical side, we provide for each of the three algorithms a family of input programs on which that algorithm outperforms the two alternatives. On the practical side, we evaluate the three algorithms experimentally on a variety of Java libraries.

1 Introduction

Large software systems are built using components and libraries, which are often developed by different teams, or even different companies. Quality component interfaces facilitate the integration and validation process for such systems. This explains the recent interest in rich interfaces for existing code, such as software libraries. We consider *temporal interfaces* [4], which specify the legal sequences of function calls to a library, i.e., those sequences that do not cause the library to enter an error state. Consider, for example, the library shown in Fig. 1, which supports read and write accesses to files. The safe use of the library requires that a file be opened for read or for read-write access before being read, and be opened for read-write access before being written. The library interface can be represented by the regular expression $((\texttt{ropen} \cdot \texttt{read}^* \cdot \texttt{close}) \cup (\texttt{rwopen} \cdot (\texttt{read} \cup \texttt{write})^* \cdot \texttt{close}))^*$. This interface is both *safe*, in that it accepts no sequence of function calls that leads to an error in the library, and *permissive*, in that it accepts all other sequences.

[⋆] This research was supported in part by the grant SFU/PRG 06-3, and by the Swiss National Science Foundation.

```
void ropen(File f) {
  if (!f.rdflag)
    f.rdflag = true;
  else
    f.error = true; }
```

```
void close(File f) {
 if (f.rdflag) {
  f.rdflag = false;
  f.wrflag = false; }
 else
  f.error = true; }
```

```
void rwopen(File f) {
 if (!f.rdflag) {
  f.rdflag = true;
  f.wrflag = true; }
 else
  f.error = true; }
```

```
void read(File f) {
 if (!f.rdflag)
  f.error = true; }
```

```
void write(File f) {
 if (!f.wrflag)
  f.error = true; }
```

Fig. 1. Example of a library that supports read and write accesses to files

Several algorithms have been proposed for automatically extracting safe and permissive temporal interfaces (in the form of finite automata) from library code. Like many questions of sequential synthesis, interface extraction is fundamentally a *game* problem, namely, the problem to compute the most general environment strategy for calling library functions without causing a safety violation. We call the algorithm that solves the safety game on the library code the 'direct' algorithm. As the complexity of this algorithm grows with the number of library states, two very different improvements have been suggested. The first is based on techniques for *learning* a finite automaton by repeatedly querying a teacher [1]. The learning algorithm guarantees the construction of a deterministic interface automaton with a minimal number of states, and thus performs well if the number of states required in the interface is small. The second improvement is based on *counterexample-guided abstraction refinement* [3]. The CEGAR algorithm computes a library abstraction, then extracts an interface automaton for the abstract library, then checks if the extracted interface is both safe and permissive for the concrete library (using two reachability tests), and if not, iteratively refines the library abstraction [5]. This algorithm performs well if there exists a small abstraction of the library from which a safe and permissive interface can be constructed.

Our aim is to compare and analyze the three approaches (direct; learning; and CEGAR) both theoretically and experimentally. Even though they address the same problem, the three algorithms proceed very differently. Moreover, the learning algorithm was published and previously implemented in the context of Java libraries without guaranteeing interface permissiveness [1], and the CEGAR algorithm was published and previously implemented in the context of C programs without ensuring interface minimality [5]. For a fair comparison, we formalize and reimplement all three algorithms in a uniform setting. In order to disregard orthogonal issues as much as possible, we remove all effects of the programming language by choosing, as input to the three algorithms, the transition graph of a library. We assume the transition graphs to be finite-state, so that all three algorithms are guaranteed to terminate (on infinite-state systems, none of the algorithms is guaranteed to terminate, although different algorithms may terminate on different inputs). In order to further level the playing field, we add a permissiveness check to the learning algorithm of [1], and we add a minimization step to the direct and the CEGAR algorithm. We also make some improvements to the published algorithms. For example, we simplify the CEGAR algorithm by combining the safety and permissiveness checks into a single reachability test (rather than using two separate tests on different automata, as suggested in [5]).

On the theoretical side, we construct parametric families of input programs that amplify the differences in the performance of the three algorithms. In experiments, we find that these input families do not represent uninteresting corner cases, but commonly occur in applications such as Java libraries. As expected, abstraction refinement performs best if only few program variables[1] are needed to prove an interface both safe and permissive. If this is the case, then the resulting interface automaton has few states. Learning also requires the interface automaton to be small, and performs better than CEGAR if the interface states reflect the values of many different program variables. The direct (game) algorithm outperforms both other approaches if the interface is not small, but the size of the state space is not too large to be explored and minimized (this is because the direct algorithm does not involve any of the overhead necessary for either learning or automatic abstraction refinement).

2 Open Programs and Interfaces

We investigate sequences of calls to a software library. We formalize the library code as an open program. In order to remove language effects, we describe an open program as a labeled transition graph over a finite set of boolean variables. The labels are function calls; one of the variables marks the error states. Certain sequences of function calls may lead the open program to an error state. At the concrete level, an open program is deterministic, and thus each sequence of function calls either causes or does not cause an error (this will not be true in general for abstractions of open programs). The set of all sequences of function calls that do not cause an error is called the *safe and permissive interface* of the open program. We strive to construct a minimal deterministic finite-state representation of that interface, called an *interface automaton*.

Finite automata. Consider a finite automaton $A = (Q, \Sigma, q_0, \delta)$ with the set Q of states, the input alphabet Σ, the initial state $q_0 \in Q$, and the transition relation $\delta \subseteq Q \times \Sigma \times Q$ (there are no accepting states). The automaton A is *serial* if for all states $q \in Q$, there exists an input symbol $f \in \Sigma$ and a state $q' \in Q$ such that $(q, f, q') \in \delta$. The automaton A is *input-enabled* if for all states $q \in Q$ and all input symbols $f \in \Sigma$, there exists a state $q' \in Q$ such that $(q, f, q') \in \delta$. The automaton A is *deterministic* if for all states $q, q', q'' \in Q$ and all input symbols $f \in \Sigma$, if $(q, f, q') \in \delta$ and $(q, f, q'') \in \delta$, then $q' = q''$. The transitive closure $\xrightarrow{w}_\delta$ of the transition relation is defined as usual: let $q \xrightarrow{\epsilon}_\delta q'$ if $q = q'$, and let $q \xrightarrow{f \cdot w}_\delta q'$ if there exists a state q'' such that $(q, f, q'') \in \delta$ and $q'' \xrightarrow{w}_\delta q'$. The *reachable region* of the automaton is $Reach(A) = \{q \in Q \mid \exists w : q_0 \xrightarrow{w}_\delta q\}$. A *trace* α of A is a finite or infinite sequence $\langle p_0, f_0, p_1, f_1, \ldots\rangle$ such that $p_0 = q_0$,

[1] We perform abstraction by hiding variables. Similar criteria can be obtained for predicate abstraction.

and $p_j \xrightarrow{f_j}_\delta p_{j+1}$ for all $j \geq 0$. The *word* induced by the trace α is the sequence $f_0 \cdot f_1 \cdot f_2 \cdots$ of input symbols. The *language* $L(A)$ is the set of finite and infinite words $w \in \Sigma^* \cup \Sigma^\omega$ such that there exists a trace of A that induces w. The *ω-language* $L^\omega(A)$ is the set of infinite words in $L(A)$; that is, $L^\omega(A) = L(A) \cap \Sigma^\omega$.

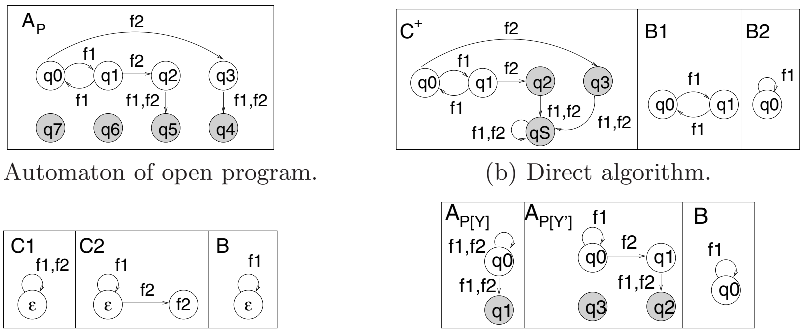
Open programs. An *open program* $P = (X, \Sigma, s_0, \varphi, x_e)$ consists of a finite set X of boolean variables, whose truth-value assignments $[\![X]\!]$ represent the states of the program; a finite alphabet Σ of exported function names; an initial state $s_0 \in [\![X]\!]$; a set φ containing a transition predicate φ_f over $X \cup X'$ for every function $f \in \Sigma$, where the set X' contains a primed variable x' for each variable $x \in X$; and an error variable $x_e \in X$. The semantics of the open program P is given by a finite automaton $A_P = ([\![X]\!], \Sigma, s_0, \delta_P)$ and a set E_P of error states. The transition relation δ_P is defined by $(s, f, t) \in \delta_P$ iff $s \cup t'$ satisfies the transition predicate φ_f, where the state $t' \in [\![X']\!]$ is obtained by giving each primed variable $x' \in X'$ the value $t(x)$. We require of every open program P that the automaton A_P be input-enabled. The open program P is *concrete* if A_P is deterministic. For a concrete open program, in every state, every function call leads to a unique successor state. We will also consider open programs that result from abstraction; in general these do not have deterministic transition relations. The set E_P of *error states* is the set of states s with $s(x_e) = \mathrm{T}$. Without loss of generality we assume that for all states $s \in E_P$, if $(s, f, s') \in \delta_P$, then $s' \in E_P$.

Interfaces. An *interface* for an open program P is a closed[2] (in the Cantor topology) set of infinite words over the alphabet Σ of function names. A finite or infinite word $w \in \Sigma^* \cup \Sigma^\omega$ is *safe* for P if for all finite prefixes w' of w, if $s_0 \xrightarrow{w'}_{\delta_P} s$, then $s \notin E_P$. A language $L \subseteq \Sigma^* \cup \Sigma^\omega$ is *safe* for P if every word in L is safe for P. A language $L \subseteq \Sigma^* \cup \Sigma^\omega$ is *permissive* for P if L contains every word that is safe for P. The *safe and permissive interface* for P is the set $I(P) \subseteq \Sigma^\omega$ of infinite words that are safe for P. Interfaces for P can be specified by serial automata over the input alphabet Σ. We look for deterministic interface specifications, which can be used to monitor the legality of a sequence of function calls. Such serial and deterministic automata can be minimized. Thus, the interface synthesis problem is defined as follows:

> *Given a concrete open program P, we wish to find the (unique) minimal serial and deterministic finite automaton B such that the ω-language $L^\omega(B)$ is the safe and permissive interface for P; that is, $L^\omega(B) = I(P)$.*

Checking interface automata for safety. Let $P = (X, \Sigma, s_0, \varphi, x_e)$ be an open program, and let $B = (Q, \Sigma, q_0, \lambda)$ be a finite automaton. The *product* of P and B is the finite automaton $A_P \times B = (Q^\times, \Sigma, q_0^\times, \lambda^\times)$ with $Q^\times = [\![X]\!] \times Q$, $q_0^\times = (s_0, q_0)$, and $\lambda^\times = \{((s, q), f, (s', q')) \mid (s, f, s') \in \delta_P \text{ and } (q, f, q') \in \lambda\}$. The language $L(B)$ is safe for P iff $s \notin E_P$ for all states $(s, q) \in \mathit{Reach}(A_P \times B)$. Based on this characterization of safety, we use a procedure *checkSafe*(P, B) to check if $L(B)$ is safe for P. If $L(B)$ is safe for P, then *checkSafe*(P, B) returns

[2] A set L of infinite words is *closed* if for every infinite word w, if every finite prefix of w is a prefix of some word in L, then $w \in L$.



(a) Automaton of open program.

(b) Direct algorithm.

(c) Learning algorithm.

(d) CEGAR algorithm.

Fig. 2. Example concrete open program and the output of the three algorithms

YES; otherwise it returns a finite trace $\langle (s_0, q_0), f_0, (s_1, q_1), f_1, \ldots, (s_n, q_n) \rangle$ of the product $A_P \times B$ such that $s_n \in E_P$.

Checking interface automata for permissiveness. Given an open program $P = (X, \Sigma, s_0, \varphi, x_e)$, the *errorless automaton* $A_P^- = (\llbracket X \rrbracket, \Sigma, s_0, \delta_P^-)$ has the transition relation $\delta_P^- = \{(s, f, s') \in \delta_P \mid s' \notin E_P\}$. Given a finite automaton $B = (Q, \Sigma, q_0, \lambda)$, the *serialized automaton* $B^+ = (Q \cup \{q_{sink}\}, \Sigma, q_0, \lambda^+)$ has the sink state q_{sink} and the transition relation $\lambda^+ = \lambda \cup \{(q, f, q_{sink}) \mid q \in Q$ and $f \in \Sigma$, and $(q, f, q') \notin \lambda$ for all $q' \in Q\} \cup \{(q_{sink}, f, q_{sink}) \mid f \in \Sigma\}$. We have the following sufficient condition on permissiveness [5]: the language $L(B)$ is permissive for P if $Reach(A_P^- \times B^+)$ contains no state of the form (s, q_{sink}). For deterministic B, the other direction also holds: if $L(B)$ is permissive for P, then $Reach(A_P^- \times B^+)$ contains no state of the form (s, q_{sink}). Based on this characterization of permissiveness, we use, for deterministic B, a procedure *checkPermissive*(P, B) to check if $L(B)$ is permissive for P. If $L(B)$ is permissive for P, then *checkPermissive*(P, B) returns YES; otherwise it returns a finite trace $\langle (s_0, q_0), f_0, (s_1, q_1), f_1, \ldots, (s_n, q_n) \rangle$ of the product $A_P^- \times B^+$ such that $q_n = q_{sink}$. The procedures *checkSafe*(P, B) and *checkPermissive*(P, B) are implemented as reachability analyses.

3 Three Algorithms for Interface Synthesis

We discuss three different algorithms for synthesizing interface automata. Figure 2(a) shows the automaton of a concrete open program, which we use as an example. The grey circles denote the error states.

3.1 Direct Algorithm

Given a concrete open program P, the algorithm *Direct* first constructs the errorless automaton A_P^-, and then calls the procedure *Prune*, which prunes the serialized automaton $(A_P^-)^+$ backwards, starting from q_{sink}, to eliminate all states

Algorithm 1. *Direct*(P)

```
Input: a concrete open program $P = (X, \Sigma, s_0, \varphi, x_e)$
Output: the minimal serial deterministic automaton $B$ such that $L^\omega(B) = I(P)$
  return $Minimize(Prune(A_P^-))$
```

all of whose successors lead to q_{sink}. The pruning removes all unrecoverable states of P, from which all infinite input sequences cause an error. Formally, a state $q \in Q$ of a deterministic automaton $C = (Q, \Sigma, q_0, \lambda)$ is *recoverable* if there exists an infinite trace $\langle p_0, f_0, p_1, f_1, \ldots \rangle$ of C such that $p_0 = q$, and $p_i \neq q_{sink}$ for all $i \geq 0$. This yields a (still deterministic) automaton D, which we refer as the intermediate automaton obtained in the direct algorithm. Then the procedure *Minimize* produces from D a minimal automaton B, using the DFA minimization algorithm [6]. (More precisely, we serialize D to obtain D^+, and consider the sink state of D^+ as rejecting, and all other states as accepting. We then minimize the automaton and remove the introduced sink state.) The result B is the minimal serial and deterministic automaton such that $L^\omega(B)$ is the safe and permissive interface for P.

Example. Figure 2(b) shows the serialized errorless automaton (C^+), its pruned version ($\mathsf{B1}$), and its minimized version ($\mathsf{B2}$), for the automaton $\mathsf{A_P}$ from Fig. 2(a) The grey circles represent the set *Err* in the procedure *Prune*. The error states from E_P are unreachable and not shown. The state qS is the sink state q_{sink}.

Time complexity. For an open program with k variables, pruning requires worst-case time $O(|\Sigma| \cdot 2^k)$. If the pruned automaton D has n states, then subsequent minimization needs $O(|\Sigma| \cdot n \cdot \log n)$ time. The worst case occurs if $n = O(2^k)$, giving a running time of $O(|\Sigma| \cdot k \cdot 2^k)$ for the direct algorithm.

Theorem 1. *Given a concrete open program P with variables X and exported function names Σ, the direct algorithm (Alg. 1) produces the minimal serial and deterministic finite automaton B such that $L^\omega(B)$ is the safe and permissive interface for P, in time linear in $|\Sigma|$ and exponential in $|X|$.*

Note that if A_P is not deterministic, then the pruning performed by the direct algorithm does not guarantee to result in a safe interface. To work on abstract open programs, the direct algorithm would have to be preceded by an exponential determinization step, i.e., subset construction. However, even for concrete open programs P, where no determinization is necessary, the direct algorithm needs to explore the entire state space of P and minimize an intermediate automaton D of possible size $O(2^k)$, where k is the number of variables of P. In software libraries, we expect many recoverable states —i.e., states from which some sequences of function calls are allowed— and this gives rise to large intermediate automata. Hence the direct algorithm is often too expensive. Therefore the following two alternative algorithms have been proposed. While no better in worst-case complexity, in many cases the two alternatives outperform the direct algorithm. They do so by employing very different strategies: the learning

Algorithm 2. *Prune*(C)

```
Input: a deterministic automaton $C = (Q, \Sigma, q_0, \lambda)$
Output: a serial deterministic automaton $B$ such that $L^\omega(B) = L^\omega(C)$
Variables: a serial automaton $C^+$, a state $q_{sink} \notin Q$, and
           three state sets $Err, Wait, Pre \subseteq (Q \cup \{q_{sink}\})$
  $C^+ :=$ serialized automaton $(Q \cup \{q_{sink}\}, \Sigma, q_0, \lambda^+)$ for $C$
  $Err := \{q_{sink}\}$;   $Wait := Err$
  while $Wait \neq \emptyset$ do
    choose $s \in Wait$;   $Wait := Wait \setminus \{s\}$
    $Pre := \{r \in Q \mid (r, f, s) \in \lambda^+$ for some $f \in \Sigma\}$
    for each state $r \in Pre$ do
      if $r \notin Err$ and $(\forall f \in \Sigma : r \xrightarrow{f}_{\lambda^+} s'$ and $s' \in Err)$ then
        $Err := Err \cup \{r\}$;   $Wait := Wait \cup \{r\}$
  return $(Q \setminus Err, \Sigma, q_0, \{(q, f, q') \in \lambda \mid q' \notin Err\})$
```

algorithm queries the concrete open program; the CEGAR algorithm automatically constructs and refines an abstract open program.

3.2 Learning Algorithm

An approach based on learning the interface was proposed by Alur et al. [1]. The learning algorithm learns the interface language by asking membership and equivalence questions to the teacher, i.e., the given concrete open program P. In a membership question, the algorithm asks whether a particular word is safe for P or not. In an equivalence question, the algorithm asks if the language of the conjectured automaton $C = (Q, \Sigma, q_0, \lambda)$ is safe and permissive for P. To construct the conjectured automaton, the learning algorithm maintains information about a finite collection of words over Σ in an observation table (R, E, G), where R and E are finite sets of words over Σ, and G is a function from $(R \cup (R \cdot \Sigma)) \times E$ to $\mathbb{B}$. The set R is a set of representative words. For each word $r \in R$ that is safe for P, there exists a state q_r in the automaton C such that $q_\epsilon \xrightarrow{r}_\lambda q_r$. The set E is a set of suffix words that distinguish the states. For all representative words $r_1, r_2 \in R$, there exists a word $e \in E$ such that only one of $r_1 \cdot e$ and $r_2 \cdot e$ is safe for P. The function G stores the results of the membership questions, i.e., it maps a pair of two words $r \in R \cup (R \cdot \Sigma)$ and $e \in E$ to T if $r \cdot e$ is safe for P, and to F otherwise. For a detailed description of the learning algorithm we refer to Alur et al. [1] (cf. also [2] and [7]). For a fair comparison between the algorithms, the learning algorithm described here learns the interface from the concrete open program rather than from a manual abstraction of the same, as proposed by Alur et al. [1]. Since a concrete open program is deterministic, the learning algorithm produces an interface that is not only safe, but also permissive.

Algorithm. The learning algorithm starts with R and E set to $\{\epsilon\}$, and G is initialized for every combination of two words from $R \cup (R \cdot \Sigma)$ and E using membership questions (procedure *memb*). Then, the algorithm checks whether the table (R, E, G) is closed (procedure *checkClosure*). If not, the algorithm adds

Algorithm 3. *Learning*(P)

```
Input: a concrete open program $P = (X, \Sigma, s_0, \varphi, x_e)$
Output: the minimal serial deterministic automaton $B$ such that $L^\omega(B) = I(P)$
Variables: two sets of words $R$ and $E$ over $\Sigma$, two words $r_{new}$ and $e_{new}$ over $\Sigma$
           an array $G$ that maps $(R \cup (R \cdot \Sigma)) \times E$ to $\mathbb{B}$,
           an automaton $C = (Q, \Sigma, q_0, \lambda)$, and
           a finite trace $\alpha^\times$ of a product automaton
  $R := \{\epsilon\}$;   $E := \{\epsilon\}$;   $G[\epsilon, \epsilon] := memb(P, \epsilon \cdot \epsilon)$;
  for each $f \in \Sigma$ do
    $G[\epsilon \cdot f, \epsilon] := memb(P, \epsilon \cdot f \cdot \epsilon)$
  while true do
    $r_{new} := checkClosure(R, E, G)$
    while $r_{new} \neq$ YES do
      $R := R \cup \{r_{new}\}$
      for each $f \in \Sigma, e \in E$ do
        $G[r_{new} \cdot f, e] := memb(P, r_{new} \cdot f \cdot e)$
      $r_{new} := checkClosure(R, E, G)$
    $C := makeConjecture(R, E, G)$
    $\alpha^\times := checkSafe(P, C)$
    if $\alpha^\times =$ YES then
      $\alpha^\times := checkPermissive(P, C)$
      if $\alpha^\times =$ YES then
        return $Prune(C)$
    $w :=$ the word induced by the trace $\alpha^\times$
    $e_{new} := findSuffix(P, R, w)$;   $E := E \cup \{e_{new}\}$
    for each $r \in R$ and $f \in \Sigma$ do
      $G[r, e_{new}] := memb(P, r \cdot e_{new})$
      $G[r \cdot f, e_{new}] := memb(P, r \cdot f \cdot e_{new})$
```

new representative words and rechecks for closure. Once (R, E, G) is closed, an automaton C is conjectured (procedure *makeConjecture*). Then, the algorithm checks if $L(C)$ is safe and permissive for P (this check represents an equivalence question). If not, a counterexample trace is returned. The longest suffix of the counterexample (found by the procedure *findSuffix*) is added to E, and the algorithm rechecks for closure. The learning algorithm constructs a deterministic automaton C whose states correspond to the trace-equivalence classes of $Reach(A_P)$. Two states $s, t \in [\![X]\!]$ are *trace-equivalent* if there are no word $w \in \Sigma^*$ and no states $s', t' \in [\![X]\!]$ such that $s \xrightarrow{w}_{\delta_P} s'$ and $t \xrightarrow{w}_{\delta_P} t'$ and $s'(x_e) \neq t'(x_e)$. Then, the algorithm calls the procedure *Prune* to produce the minimal serial and deterministic finite automaton B such that $L^\omega(B)$ is the safe and permissive interface for P.

Example. Figure 2(c) shows in the first two boxes the two conjectured automata. Automaton $\mathsf{C2}$ is the final conjecture, which is used to produce the serial deterministic finite automaton B.

Procedures used in the learning algorithm

- *memb*(P, w) returns T if w is safe for P. Otherwise it returns F.
- *checkClosure*(R, E, G) returns YES if for every $r \in R$ and $f \in \Sigma$, there exists an $r' \in R$ such that $G[r \cdot f, e] = G[r', e]$ for every $e \in E$. Otherwise it returns the word $r \cdot f$ such that there is no r' satisfying the above condition.
- *makeConjecture*(R, E, G) returns a deterministic automaton $C = (Q, \Sigma, q_0, \lambda)$, where $Q = R \setminus \{r \in R \mid G[r, \epsilon] = \text{F}\}$, and $q_0 = \epsilon$, and for every $r \in Q$ and every $f \in \Sigma$, if $G[r \cdot f, \epsilon] = \text{T}$, then $(r, f, r') \in \lambda$, where r' is the word such that $G[r \cdot f, e] = G[r', e]$ for every $e \in E$.
- *findSuffix*(P, R, w) finds the longest suffix w' of w such that for some $r \in R$ and $f \in \Sigma$, *memb*$(P, r \cdot f \cdot w') \neq$ *memb*$(P, r' \cdot w')$, where $r \xrightarrow{f}_{\lambda} r'$.

Time complexity. For an open program with k variables and m trace-equivalence classes, the generation of a conjectured automaton has the time complexity $O(2^k \cdot (m^2 \cdot |\Sigma| + m \cdot \log c))$, where c is the length of the longest counterexample trace $\alpha^{\times}$ seen by the algorithm. At the end, a call to the procedure *Prune* takes $O(m \cdot |\Sigma|)$ time. Thus the learning algorithm has the worst-case time complexity $O(|\Sigma| \cdot 2^{3k})$ when the number of trace-equivalence classes is $O(2^k)$. However, when the number m of trace-equivalence classes (which determines the size of the output automaton) is small compared to the number 2^k of concrete program states, then the learning algorithm may perform better than the direct algorithm. This is because learning produces the minimal interface automaton, whereas the direct algorithm needs to explicitly minimize an intermediate automaton of potential size $O(2^k)$.

Theorem 2. *Given a concrete open program P with variables X and exported function names Σ, and m trace-equivalence classes in Reach(A_P), the learning algorithm (Alg. 3) produces the minimal serial and deterministic finite automaton B (with $O(m)$ states) such that $L^{\omega}(B)$ is the safe and permissive interface for P, in time linear in $|\Sigma|$, quadratic in m, and exponential in $|X|$.*

3.3 CEGAR Algorithm

A different approach based on automatic abstraction refinement was proposed by Henzinger et al. [5].

Abstraction. An *abstraction* for an open program $P = (X, \Sigma, s_0, \varphi, x_e)$ is a set $Y \subseteq X$ of variables, where $x_e \in Y$. The abstraction hides the variables in $X \setminus Y$. Given a state $s \in [\![X]\!]$, the state $s[Y]$ is the valuation in $[\![Y]\!]$ such that $s(x) = s[Y](x)$ for all $x \in Y$. An open program P and an abstraction Y for P yield the (*abstract*) *open program* $P[Y] = (Y, \Sigma, s_0[Y], \varphi[Y], x_e)$, where for each $f \in \Sigma$, the transition predicate $\varphi_f[Y]$ is the projection $\exists (X \cup X') \backslash (Y \cup Y') : \varphi_f$ of φ_f to the variables in $Y \cup Y'$ (existential abstraction). The semantics of $P[Y]$ is given by the abstract automaton $A_{P[Y]}$ and the set $E_{P[Y]}$ of abstract error states. Note that $(s, f, s') \in \delta_{P[Y]}$ iff $(t, f, t') \in \delta_P$ for some concrete states $t, t' \in [\![X]\!]$ with $s = t[Y]$ and $s' = t'[Y]$. The original CEGAR algorithm for interface synthesis [5] uses two abstractions: one for checking safety and a possibly different one for

Algorithm 4. *CEGAR*(P)

```
Input: a concrete open program $P = (X, \Sigma, s_0, \varphi, x_e)$
Output: the minimal serial deterministic automaton $B$ such that $L^\omega(B) = I(P)$
Variables: an abstraction $Y$ for $P$, the open program $P[Y]$, an automaton $C$
  $Y := \{x_e\}$
  while $Y \neq X$ do
    $C := A^-_{P[Y]}$
    $\alpha^\times := \mathit{checkSafe}(P[Y], C)$
    if $\alpha^\times = $ YES then
      return $\mathit{Minimize}(\mathit{Prune}(\mathit{Determinize}(C)))$
    else
      $\alpha := \mathit{findSpuriousTrace}(P, \alpha^\times)$;    $Y := \mathit{getNewVars}(P, \alpha, Y)$
  return $\mathit{Minimize}(\mathit{Prune}(\mathit{Determinize}(A^-_P)))$
```

checking permissiveness. We use a single abstraction, based on the following observations. An open program P with initial state s_0 and error variable x_e is *visibly deterministic* [5] if there is no word $w \in \Sigma^*$ and no states $s, t \in [\![X]\!]$ such that $s_0 \xrightarrow{w}_{\delta_P} s$ and $s_0 \xrightarrow{w}_{\delta_P} t$ and $s(x_e) \neq t(x_e)$.

Lemma 1. *Given the errorless automaton $A^-_P = (Q, \Sigma, q_0, \lambda)$ of an open program P, if the language $L(A^-_P)$ is safe for P, then $L(A^-_P)$ is permissive for P and P is visibly deterministic.*

Proof. (i) We know that the safety and permissiveness conditions are reachability questions on $A_P \times A^-_P$ and $A^-_P \times {A^-_P}^+$, respectively. As A_P is input-enabled, we know that if $(q, f, q_{sink}) \in \lambda^+$, then $(q, f, q') \in \delta_P$ with $q' \in E_P$. Thus, if there exists no state $(t, q) \in \mathit{Reach}_{A_P \times A^-_P}$ such that $t \in E_P$, then there exists no state $(t, q_{sink}) \in \mathit{Reach}_{A^-_P \times {A^-_P}^+}$. Hence, $L(A^-_P)$ is a permissive interface for P.

(ii) The fact that $L(A^-_P)$ is safe for P guarantees that there exists no word w such that w is not safe for P and $q_0 \xrightarrow{w}_\lambda q$ for some state $q \in Q$. Also, we know that the automaton A^-_P is the errorless automaton for P, and $L(A^-_P)$ is permissive for P. Therefore, there exists no word w such that there exist two states u and v with $s_0 \xrightarrow{w}_{\delta_P} u$ and $s_0 \xrightarrow{w}_{\delta_P} v$ and $u(x_e) \neq v(x_e)$. Hence, P is visibly deterministic. □

Lemma 2. *Let Y be an abstraction for a concrete open program P such that $P[Y]$ is visibly deterministic. If a language L is safe and permissive for $P[Y]$, then L is safe and permissive for P.*

Proof. Let a word $w \in L$ be a counterexample for safety of P. By construction of $A_{P[Y]}$, we know that the word w is also unsafe for the abstract open program $P[Y]$, which is a contradiction to our assumption that L is safe for $P[Y]$. Similarly, permissiveness of L for $P[Y]$ guarantees that L is permissive for P. □

Algorithm 5. *getNewVars*(P, α, Y)

```
Input: a concrete open program $P = (X, \Sigma, s_0, \varphi, x_e)$, an abstraction $Y$ for $P$, and
       a finite trace $\alpha = \langle t_0, f_0, \ldots, t_n \rangle$ of the automaton $A_{P[Y]}$
Output: a new abstraction $Y'$ for $P$ such that $Y \subset Y'$ and $\alpha$ is not a trace of $A_{P[Y']}$
Variables: states $s, s' \in [\![X]\!]$ and $t, t_s, t' \in [\![Y]\!]$, a set $R \subseteq [\![X]\!]$ of states, and $f \in \Sigma$
  $s := s_0$;   $t := s_0[Y]$
  for $i := 1$ to $n$ do
    $t_s := t_i$;   $f := f_{i-1}$
    let $s' \in [\![X]\!]$ be such that $s \xrightarrow{f}_{\delta_P} s'$;   let $t' \in [\![Y]\!]$ be such that $t' = s'[Y]$
    if $t' \neq t_s$ then
      $R := \{r \in Reach(A_P) \mid t = r[Y]$ and there exists $u \in [\![X]\!]$ such that $t_s = u[Y]$
                and $r \xrightarrow{f}_{\delta_P} u\}$
      return $splitState(s, R, Y)$
    $s := s'$;   $t := t'$
```

Algorithm. We start with an abstraction that contains only the error variable; that is, $Y = \{x_e\}$. We construct the abstract open program $P[Y]$ and its errorless automaton $C = A^{-}_{P[Y]}$. Then, we check whether $L(C)$ is safe for $P[Y]$. If so, then we know that $L(C)$ is also permissive and that $P[Y]$ is visibly deterministic (by Lemmas 1 and 2). Otherwise, we obtain a counterexample trace $\alpha^{\times} = \langle (s_0[Y], q_0), f_0, \ldots, (s_n[Y], q_n) \rangle$ of the product automaton $A_{P[Y]} \times C$. Now we use the procedure *findSpuriousTrace*$(P, \alpha^{\times})$ to check if the word w induced by the trace $\alpha^{\times}$ is safe for the concrete open program P. If w is safe (resp. unsafe) for P, then the procedure declares the projection α of $\alpha^{\times}$ that is followed by the component automaton $A_{P[Y]}$ (resp. C) as spurious. Formally, the finite trace $\alpha = \langle t_0, f_0, \ldots, t_n \rangle$ of the abstract automaton $A_{P[Y]}$ (resp. $C = A^{-}_{P[Y]}$) is *spurious* if there exists no trace $\langle s_0, f_0, \ldots, s_n \rangle$ of the concrete automaton A_P such that $t_i = s_i[Y]$ for all $0 \leq i \leq n$. Next, we add more variables from X to the abstraction Y such that the spurious trace α is eliminated from $A_{P[Y]}$ (resp. C). This is done by the procedure *getNewVars*, which constructs a trace $\beta = \langle s_0, f_0, \ldots, s_n \rangle$ of A_P, and its corresponding abstract trace $\beta[Y]$, such that β induces the same word as α. The procedure locates the first position i where the spurious abstract trace α differs from the genuine abstract trace β. Then it finds a set R of states in A_P that cause the spurious abstract trace, and a set $Y' \subseteq X$ of variables such that $Y \subset Y'$ and if $t = s_{i-1}[Y']$, then there does not exist a state $r \in R$ with $t = r[Y']$. This concludes one refinement step.

In the next refinement iteration, we construct the refined abstract open program $P[Y']$, and check if it is safe (and therefore visibly deterministic using Lemma 1). We say that an abstraction Y *suffices to prove the safety* of P if $P[Y]$ is visibly deterministic. Once the CEGAR algorithm finds a visibly deterministic abstract open program $P[Y]$, we call the procedure *Determinize* followed by *Prune* and *Minimize*, to obtain the minimal serial and deterministic finite automaton B such that $L^{\omega}(B)$ is the safe and permissive interface for P. This is because before minimization, the abstract automaton $A_{P[Y]}$ found by the CEGAR algorithm may not be minimal. Note that given a concrete open program P, finding an abstraction Y for P with a minimal number of variables such that the abstract open program $P[Y]$ is visibly deterministic, is NP-hard [3].

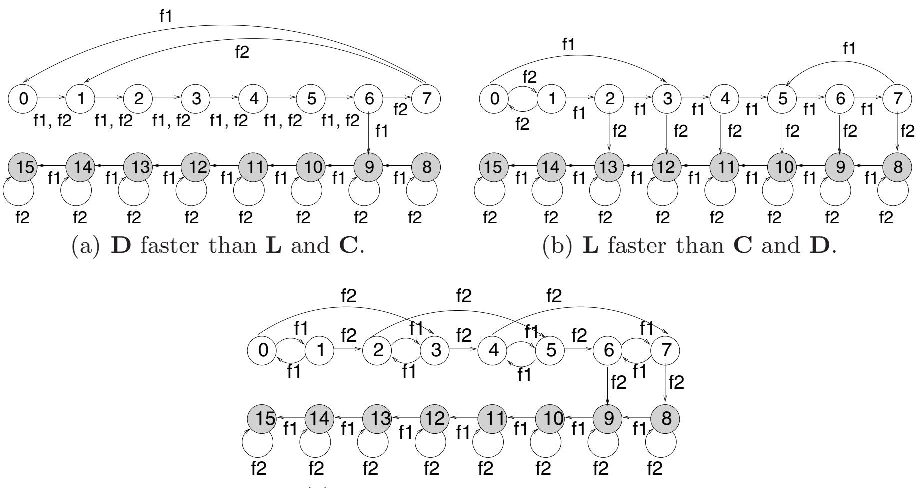
Example. Figure 2(d) shows in the first box automaton $\mathsf{A}_{\mathsf{P[Y]}}$ with the abstraction $Y = \{x_e\}$. Adding one more variable yields the automaton $\mathsf{A}_{\mathsf{P[Y']}}$, whose open program is found to be visibly deterministic. The result B is computed by first determinizing, then pruning and minimizing.

Procedures used in the CEGAR algorithm

- *splitState*(s, R, Y) for $s \in [\![X]\!]$, $R \subseteq [\![X]\!]$, and Y being the current abstraction, finds a set $Y' \subseteq X$ of variables such that $Y \subset Y'$ and there is no state $r \in R$ such that $t = r[Y']$, where $t \in [\![Y']\!]$ with $t = s[Y']$. It returns Y'.
- *findSpuriousTrace*$(P, \alpha^\times)$ first checks whether the word w induced by the finite trace $\alpha^\times$ of the product automaton is safe for P. If so, then returns the trace $\langle s_0[Y], f_0, \dots, s_n[Y]\rangle$ of $A_{P[Y]}$ in $\alpha^\times$. Otherwise, it returns the trace $\langle q_0, f_0, \dots, q_n\rangle$ of $C = A^-_{P[Y]}$ in $\alpha^\times$.
- *Determinize*(A) determinizes the serial automaton $A = (Q, \Sigma, q_0, \delta)$. We note that since A is the automaton for a visibly deterministic open program, the automaton A is 'almost' deterministic, and it is straightforward to determinize A. Determinization does not change the set of states, the alphabet, and the initial state, only the transition relation: for every state $q \in Q$ and every function $f \in F$, if there exist more than one transitions from q on f, then we choose arbitrarily one of the transitions $(q, f, q') \in \delta$ to be in the new transition relation of the resulting deterministic automaton.

Time complexity. Let X and abstraction Y be sets of k and c variables, respectively. One iteration of the algorithm requires $O(|\Sigma| \cdot 2^{\max\{k,2c\}})$ time. At the end of the refinement procedure, the call to procedure *Determinize* runs in time $O(|\Sigma| \cdot 2^l)$, where l is the number of variables in the abstraction that suffices to prove the safety of P. The procedure *Prune* requires time $O(|\Sigma| \cdot 2^l)$ followed by the procedure *Minimize*, which takes time $O(|\Sigma| \cdot l \cdot 2^l)$. Thus, the worst-case time complexity of the CEGAR algorithm is $O(|\Sigma| \cdot 2^{2k})$, which is encountered if the abstraction refinement introduces all k program variables. However, when the number l of variables that suffice to prove the safety of P (which determines the size of the output automaton) is small compared to the number k of all program variables, then the CEGAR algorithm may perform better than the direct algorithm, because the exponential time dependency on k is due only to the cost of constructing the abstract program. To be precise, to check whether there is an abstract transition $(s, f, s') \in \delta_{P[Y]}$ between two given abstract states $s, s' \in [\![Y]\!]$ requires time $O(2^k)$ but only space $O(k)$. Such a check can often benefit from symbolic methods.

Theorem 3. *Given an open program P with variables X and exported function names Σ, the CEGAR algorithm (Alg. 4) produces the minimal serial and deterministic finite automaton B (with $O(2^l)$ states) such that $L^\omega(B)$ is the safe and permissive interface for P, in time linear in $|\Sigma|$ and exponential in $|X| + l$, where l is the size of an abstraction that suffices to prove the safety of P.*



(a) **D** faster than **L** and **C**.

(b) **L** faster than **C** and **D**.

(c) **C** faster than **L** and **D**.

Fig. 3. Examples of concrete open programs where one algorithm performs better than others. The grey circles denote the error states.

4 Theoretical Separation of the Algorithms

We describe three theoretical classes of examples that amplify the differences between the three algorithms presented in the previous section. These examples suggest that the three algorithms are important in their own right, and it is worthwhile to understand them properly for efficient usage.

We consider concrete open programs with k variables and a fixed alphabet $\Sigma = \{f_1, f_2\}$. We denote the set of states by $\{s_0, s_1, ...s_{2^k-1}\}$. The boolean value of the variables is encoded in the index of the state; for example, at s_1, the first $k-1$ variables are 0, and the last variable is 1. Also, the first variable is the error variable. Thus, the first half of the states are non-error states, and the second half are error states. We consider all pairs of the direct (**D**), learning (**L**), and CEGAR (**C**) algorithm. We evaluate the pairs on the metric of time complexity. We show graphical examples with $k = 4$ in Fig. 3, which can be scaled to arbitrary k. We assume that the CEGAR algorithm finds the minimal sufficient abstraction in each case.

- **D** *faster than* **L** *and* **C**. For the open program in Fig. 3(a), the interface automaton has size $O(2^k)$; that is, the interface is no smaller than the library. The direct algorithm requires $O(2^k \cdot k)$ time, whereas the learning algorithm requires $O(2^{3k})$ time and the CEGAR algorithm requires $O(2^{2k})$ time. The direct algorithm is the fastest, because it avoids the overhead of learning and abstraction refinement.
- **L** *faster than* **C** *and* **D**. The interface automaton for the open program in Fig. 3(b) has two states (for all k). Hence, the learning algorithm requires $O(2^k)$ time. On the other hand, the CEGAR algorithm has to continue

Table 1. Run time for different algorithms, measured on a 3.0 GHz Pentium IV machine with 1 GB memory. The parameter k is described for each class in the text.

k	Interface automaton size	Learning time	CEGAR time
List Iterator			
5	2	0.19 s	0.91 s
6	2	0.43 s	1.53 s
7	2	1.10 s	4.31 s
8	2	2.31 s	12.12 s
Piped Output Stream			
12	2	2.12 s	0.89 s
13	2	5.30 s	1.83 s
14	2	12.32 s	3.76 s
15	2	27.82 s	7.68 s

refinement until adding $k-1$ variables to the abstract program, and thus produces an intermediate automaton with $O(2^k)$ states. Hence, including minimization, CEGAR requires $O(2^{2k})$ time. The direct algorithm prunes the concrete program. This yields an intermediate automaton with $O(2^k)$ states, which is then minimized to obtain the interface automaton with two states. Thus, the direct algorithm requires $O(2^k \cdot k)$ time.

– **C** *faster than* **L** *and* **D**. For the open program in Fig. 3(c), the number of trace-equivalence classes is exponential in the number l of variables that suffice to prove the safety of the program, and l is logarithmic in the number of all variables; that is, $l = O(\log k)$. Thus the CEGAR algorithm requires $O(2^k \cdot \log k)$ time. On the other hand, the learning algorithm requires $O(2^k \cdot k^2)$ time, because it depends quadratically on the size of the interface automaton. The direct algorithm again has to minimize $O(2^k)$ recoverable states, to produce the interface automaton with $O(2^l)$ states. Thus, the direct algorithm runs in $O(2^k \cdot k)$ time.

5 Practical Evaluation of the Algorithms

We implemented the three algorithms in C++. We experimented with a variety of Java libraries [1], all of which have a finite number of states. For comparison purposes, we wanted the direct algorithm to succeed on the concrete open programs; thus we first simplified the Java classes. We retained all fields in a class, but reduced their sizes, and hence the state spaces. The Java libraries were manually translated into such simplified, but still concrete, open programs. The input to the tool is the transition relation of a concrete open program. Table 1 reports some results of our experiments.

Direct works fastest. The following example is similar to Fig. 3(a), where the direct algorithm performs better than the other algorithms.

- *java.util.Stack:* We consider the class with $k+1$ boolean variables. The first variable encodes error, and the remaining k variables encode the current size of the stack (thus, the maximal size of the stack is 2^k). We create an interface for the methods *push*(), *pop*(), and *peek*(). The algorithms produce the interface automaton with $O(2^k)$ states. The direct algorithm is fastest.

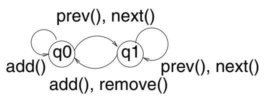
In general, the direct algorithm performs best if either the number of recoverable library states is small, or as in the example above, the number of trace-equivalence classes of the library is of the order of the number of library states. Neither is the case for the following three examples.

Learning works fastest. The following example is similar to Fig. 3(b).

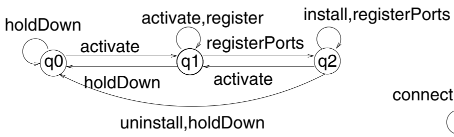
- *java.util.ListIterator:* We compute the interface for four methods: *next*(), *prev*(), *remove*(), *add*(). The list iterator is encoded by $2k+1$ boolean variables, one for error, k to encode the previous returned iterator location l_p, and another k to encode the current iterator location l_c. The methods *next*() and *prev*() store l_c in l_p, and update l_c. The method *remove*() checks if l_p is valid; if so, then *remove*() removes the entry in location l_p. Both *add*() and *remove*() invalidate l_p. The CEGAR algorithm finds that only the last k variables are redundant, and thus produces an automaton of size $O(2^k)$. This automaton is then minimized to obtain the interface automaton with two states, which is shown in Fig. 4(a). On the other hand, the learning algorithm learns that only one value of l_p (reached on calling *add*() or *remove*()) marks the previous iterator location as invalid, and that all other values are equivalent. Hence, the learning algorithm finishes after distinguishing two states of the interface.

CEGAR works fastest. The following programs are similar to Fig. 3(c).

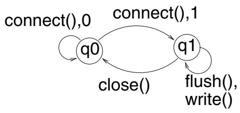
- *com.sun.se.impl.activation.ServerTableEntry:* We encode the class using $k+3$ boolean variables. The first variable encodes error, the next two encode the state of the system, and the remaining k encode the current server ID. The interface is built for six methods, as shown in Fig. 4(b). The CEGAR algorithm finds that the two variables that encode the state of the system suffice to prove the safety of the class; after minimization it produces the interface automaton with three states, which is shown in Fig. 4(b). The learning algorithm learns the three distinguishable states of the interface, but takes a longer time to do so.
- *java.io.PipedOutputStream:* The class is represented by $k+2$ boolean variables. The first variable encodes error, the second encodes the connect flag, and the remaining k variables encode the buffer. We build an interface for the following methods: *connect*(), *write*(), *flush*(), and *close*(). We model invocations of *connect*() returning different values (0 or 1) as different methods. The CEGAR algorithm discovers that the variable that encodes the connect flag suffices to prove the safety of the class. The output is an interface automaton with two states, which is shown in Fig. 4(c). Again, the learning algorithm needs more time to find the two distinguishable states.



(a) List iterator.



(b) Server table entry.



(c) Piped output stream.

Fig. 4. Interface automata for list iterator, server table entry, and piped output stream

6 Conclusion

We formalized and implemented three different algorithms for interface synthesis in a uniform framework. For each of the three algorithms, we identified classes of open programs for which the algorithm is better suited for interface synthesis than the two alternatives. The direct algorithm has the advantage in scenarios where the interface automaton of the library is large, or the program has few recoverable states, i.e., states from which some sequences of function calls are legal. The CEGAR algorithm is the most efficient solution when many variables of the input program can be hidden in the interface automaton. The learning algorithm performs best if the interface automaton is much smaller than the set of recoverable program states, but does not correspond to an abstraction of the input program over a small set of program variables.

References

1. Alur, R., Cerny, P., Gupta, G., Madhusudan, P.: Synthesis of interface specifications for Java classes. In: Proc. POPL, pp. 98–109. ACM Press, New York (2005)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Proc. CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
4. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. FSE, pp. 109–120. ACM Press, New York (2001)
5. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: Proc. FSE, pp. 31–40. ACM Press, New York (2005)
6. Hopcroft, J.E.: An $n \cdot \log n$ algorithm for minimizing states in a finite automaton. In: Proc. Theory of Machines and Computations, pp. 189–196. Acad. Press, San Diego (1971)
7. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Information and Computation 103, 299–347 (1993)