Three Algorithms for Interface Synthesis: A Comparative Study

Dirk Beyer Thomas A. Henzinger Vasu Singh

Technical Report No. MTC-REPORT-2006-001
May 16, 2006

Ecole Polytechnique Fédérale de Lausanne
Faculté Informatique & Communications
CH-1015 Lausanne, Switzerland

Three Algorithms for Interface Synthesis: A Comparative Study *

Dirk Beyer   Thomas A. Henzinger   Vasu Singh
EPFL, Switzerland

Abstract

A temporal interface for a system component is a finite automaton that specifies the legal sequences of input events. We evaluate and compare three different algorithms for automatically extracting the temporal interface from the transition graph of a component: (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 component to construct the minimal interface automaton; and (3) a* CEGAR *algorithm that iteratively refines an abstract interface hypothesis by adding relevant state information from the component. Since algorithms (2) and (3) have been published in different software contexts, for comparison purposes, we present the three algorithms in a uniform finite-state setting. We furthermore extend the three algorithms to construct maximally permissive interface automata, which accept all legal input sequences. While the three algorithms have similar worst-case complexities, their actual running times differ greatly depending on the component whose interface is computed. On the theoretical side, we provide families of components that exhibit exponential differences in the performance of the three algorithms. On the practical side, we evaluate the three algorithms experimentally on a variety of real world examples. Not surprisingly, the experimental evaluation confirms the theoretical expectation: learning performs best if the minimal interface automaton is small; CEGAR performs best if only few component variables are needed to prove an interface hypothesis safe and permissive; and the direct (game) algorithm outperforms both approaches if neither is the case.*

* This research was supported in part by the SNSF grants 5005-67322 (MICS-NCCR) and 200021-107600/1.

1. Introduction

Complex systems are built using components and libraries, which are often developed by different teams, or even different companies. Quality component interfaces greatly facilitate the integration and validation process for such systems. This explains the recent interest in rich interfaces for hardware and software components [6, 8]. We consider *temporal interfaces*, which specify the legal sequences of input events for a component, i.e., those sequences of input events that do not cause the component to enter an error state [4]. Consider, for example, the component shown in Fig. 1, which controls read and write accesses to a device. The component requires that the device be opened for read or for read-write access before being read, and be opened for read-write access before being written. Once the device is released, it needs to be reopened again according to the same rule. Thus, the temporal interface of the device controller can be represented by the regular expression $((\mathtt{acq_r} \cdot \mathtt{read}^* \cdot \mathtt{rel}) \cup (\mathtt{acq_rw} \cdot (\mathtt{read} \cup \mathtt{write})^* \cdot \mathtt{rel}))^*$. The interface is both *safe*, in that it accepts no sequence of input events that leads to an error, and *permissive*, in that it accepts all other sequences.

Several algorithms have been proposed for automatically extracting safe and permissive temporal interfaces (in the form of finite automata) from component descriptions. Like

```
void acq_r(Device d) {
  if (!d.rdflag)
    d.rdflag = true;
  else
    d.error = true; }

void read(Device d) {
 if (!d.rdflag)
  d.error = true; }

void acq_rw(Device d) {
 if (!d.rdflag){
  d.rdflag = true;
  d.wrflag = true;
 } else
  d.error = true; }

void rel(Device d) {
 if (d.rdflag){
  d.rdflag = false;
  d.wrflag = false;
 } else
  d.error = true; }

void write(Device d) {
 if (!d.wrflag)
  d.error = true; }
```

Figure 1. Example of an open system that supports read-write access to devices

many questions of sequential synthesis, interface extraction is a *game* problem, namely, to compute the most general environment strategy for providing input events without causing a safety violation in the component. We call the algorithm that solves the safety game on the transition graph of the component, the "direct" algorithm. As the complexity of this algorithm grows with the number of states of the component, two different improvements have been suggested, both in the context of software component libraries. The first improvement is based on techniques for *learning* a finite automaton by repeatedly querying a teacher [2, 7]. The learning algorithm has been applied to interface synthesis by Alur et al. [1]. It learns the interface by querying the component repeatedly. The learning algorithm guarantees the construction of an interface with a minimal number of states, and thus performs well if the number of states required in the interface is much smaller than the number of component states. The second improvement is based on *counterexample-guided abstraction refinement* [3]. The CEGAR algorithm computes an abstraction of the component, then extracts an interface for the abstraction, then checks if the extracted interface is both safe and permissive for the concrete component (using two reachability tests), and if not, iteratively refines the component abstraction [5]. This algorithm performs well if there exists a small abstraction of the component from which a safe and permissive interface can be constructed.

Our aim is to compare and evaluate the three approaches (direct; learning; and CEGAR) both theoretically and experimentally in a neutral (hardware and software independent) setting. 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 [5]. Thus, for a fair comparison, we had to 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 component description language by choosing, as input to the three algorithms, the transition graph of the component. We furthermore choose the transition graphs to be finite-state, so that all three algorithms are guaranteed to terminate (on infinite-state systems, none of the algorithms may terminate, although different algorithms may terminate on different inputs). Finally, for efficient implementations, we choose the finite transition graphs to be given symbolically, by BDDs representing the transition relation, the initial states, and the error states of a component. In order to further level the playing field, we had to make some additions to the published algorithms. For instance, in order to ensure that the learning algorithm extracts a permissive interface (not only a safe interface), we add a permissiveness check to the algorithm of [1]. In the process, we also found some improvements to the published algorithms. For example, in the CEGAR algorithm, we are able to combine the safety and permissive checks by performing a single reachability test (rather than two separate tests on different automata, as suggested in [5]).

On the theoretical side, we construct parametric families of components that cause exponential differences in the performance of the three algorithms. In experiments, we find that these exponential differences do not represent uninteresting corner cases, but commonly occur in applications. In particular, the direct, learning, and CEGAR algorithms exhibit large differences in running time and output size when applied to components that encapsulate various kinds of data structures such as bit arrays and counters. While all three algorithms guarantee that they output an interface automaton accepting the same language —namely, the safe and permissive interface of the component— the number of states of the output automaton can differ greatly: only learning guarantees to output a minimal deterministic automaton, but at extra cost in time (the other outputs may be non-deterministic). As expected, learning performs best if the minimal interface automaton is small; abstraction refinement performs best if only few component variables are needed to prove an interface both safe and permissive; the direct (game) algorithm outperforms both approaches if neither is the case, because it does not involve any of the overhead necessary for either learning or automatic abstraction refinement.

2. Open Systems and Interfaces

Components are *open systems*, which react to inputs provided by an environment (other components, or primary inputs). In order to remove language effects, we describe open systems as labeled transition graphs over a finite set of boolean variables. The labels are input events; one of the variables marks the error states. Certain sequences of input events may lead to an error state. At the concrete level, a component is visibly deterministic: each input sequence either causes or does not cause an error (this will not be true for abstractions of the component). The set of all input sequences that do not cause an error is called the *safe and permissive interface* of the open system. We strive to construct a finite-state representation of that interface.

Open systems. An *open system* $S = (X, \Sigma, s_0, \varphi, x_e)$ consists of a finite set X of boolean variables, whose truth-value assignments $[\![X]\!]$ represent the states of the system; a finite alphabet Σ of input events[§] ; an initial state $s_0 \in [\![X]\!]$;

§ An input event may include a value, either input or result or both.

a set φ containing a transition predicate φ_f over $X \cup X'$ for every event $f \in \Sigma$; and an error variable $x_e \in X$.

Automaton representation of open systems. The semantics of the open system S is given by the finite automaton $A_S = (\llbracket X \rrbracket, \Sigma, s_0, \delta_S)$ and the set E_S of error states. The finite automaton A_S consists of the set $\llbracket X \rrbracket$ of states, and the input alphabet Σ, and the initial state s_0, and the transition relation $\delta_S \subseteq \llbracket X \rrbracket \times \Sigma \times \llbracket X \rrbracket$, where $(s, f, t) \in \delta_S$ if $s \cup t'$[¶] satisfies φ_f. The set E_S of error states is the set of states s with $s(x_e) = \mathrm{T}$. W.l.o.g. we assume that for all states $s \in E_S$, if $(s, f, s') \in \delta_S$ then $s' \in E_S$. We inductively define the *transitive closure* $\xrightarrow{w}_{\delta_S}$ of the transition relation δ_S as: let $s \xrightarrow{\epsilon}_{\delta_S} s'$ if $s = s'$, and let $s \xrightarrow{f \cdot w}_{\delta_S} s'$ if there exists a state s'' s.t. $(s, f, s'') \in \delta_S$ and $s'' \xrightarrow{w}_{\delta_S} s'$. We also define the *reachable region* as $Reach(A_S) = \{s \in \llbracket X \rrbracket \mid s_0 \xrightarrow{w}_{\delta_S} s$ for some word $w \in \Sigma^*\}$. The automaton A_S is *input-enabled* if for all states $s \in \llbracket X \rrbracket$ and for all events $f \in \Sigma$ there exists a state $s' \in \llbracket X \rrbracket$ such that $s \xrightarrow{f}_{\delta_S} s'$. Two states $s, t \in \llbracket X \rrbracket$ are *trace-equivalent* if there is no word $w \in \Sigma^*$ such that $s \xrightarrow{w}_{\delta_S} s'$ and $t \xrightarrow{w}_{\delta_S} t'$ and $s'(x_e) \neq t'(x_e)$. The open system S is *visibly deterministic* if for every word $w \in \Sigma^*$, all states s and t with $s_0 \xrightarrow{w}_{\delta_S} s$ and $s_0 \xrightarrow{w}_{\delta_S} t$ are trace-equivalent.

We require open systems to yield input-enabled automata and to be visibly deterministic. This assumption, which is justified for concrete hardware and software systems (but not for abstractions), is necessary for the three algorithms we compare to produce permissive interfaces. (It would not be required for safety, but safe interfaces are not unique).

Interfaces. An *interface* for the open system S is a closed (in the Cantor topology) set of infinite words over the alphabet Σ. A finite or infinite word $w \in \Sigma^* \cup \Sigma^\omega$ is *safe* for S if for all finite prefixes w' of w, if $s_0 \xrightarrow{w'}_{\delta_S} s$ then $s \notin E_S$. An interface $I \subseteq \Sigma^\omega$ is *safe* for S if all words $w \in I$ are safe for S. The interface I is *permissive* for S if it contains every infinite word that is safe for S.

Automaton representation of interfaces. An interface can be specified by a serial finite automaton $A = (Q, \Sigma, q_0, \lambda)$ with the set Q of states, the input alphabet Σ, the initial state $q_0 \in Q$, and the transition relation $\lambda \subseteq Q \times \Sigma \times Q$. The automaton A is *serial* if for all states $q \in Q$, there exists an event $f \in \Sigma$ and a state $q' \in Q$ such that $q \xrightarrow{f}_\lambda q'$. A *trace* α of A is a finite or infinite sequence $\langle p_0, f_0, p_1, f_1, \ldots \rangle$ s.t. $p_0 = q_0$ and $p_j \xrightarrow{f_j}_\lambda p_{j+1}$ for all $j \geq 0$. The *word* induced by the trace α is the sequence $\langle f_0, f_1, \ldots \rangle$ of events. The *language* $L(A)$ is the set of all finite and infinite words $w \in \Sigma \cup \Sigma^\omega$ s.t. there exists a trace α of A that induces w. The ω-*language* $L^\omega(A)$ is the set of all infinite words $w \in L(A) \cap \Sigma^\omega$. The ω-language $L^\omega(A)$ of a serial automaton is closed, and therefore an interface. *Given an open system S, we wish to find a serial automaton B such that the ω-language $L^\omega(B)$ is the safe and permissive interface for S.*

Checking interfaces for safety. The *product* of an open system $S = (X, \Sigma, s_0, \varphi, x_e)$ and an automaton $A = (Q, \Sigma, q_0, \lambda)$ is the automaton $A_S \times A = (Q^\times, \Sigma, q_0^\times, \lambda^\times)$ with $Q^\times = \llbracket X \rrbracket \times Q$, $q_0^\times = (s_0, q_0)$, and $\lambda^\times = \{((s, q), f, (s', q')) \mid (s, f, s') \in \delta_S$ and $(q, f, q') \in \lambda\}$. The language $L(A)$ of an automaton A is *safe* for S if $s \notin E_S$ for all states $(s, q) \in Reach(A_S \times A)$. We use a procedure $checkSafety(S, A)$ to check if $L(A)$ is safe for S. If $L(A)$ is safe for S, then $checkSafety(S, A)$ returns YES, else 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 $A_S \times A$ s.t. $s_n \in E_S$.

Checking interfaces for permissiveness. The language $L(A)$ of an automaton is *permissive* for S if $Reach(A_S^- \times A^+)$ contains no state of the form (s, q_{sink}), where A_S^- and A^+ are defined as follows. The *errorless automaton* $A_S^- = (\llbracket X \rrbracket, \Sigma, s_0, \delta_S^-)$ has the transition relation $\delta_S^- = \{(s, f, s') \in \delta_S \mid s' \notin E_S\}$. The *input-enabled automaton* $A^+ = (Q \cup \{q_{sink}\}, \Sigma, q_0, \lambda^+)$ 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 use a procedure $checkPermissive(S, A)$ to check whether $L(A)$ is permissive for S. If $L(A)$ is permissive for S, then $checkPermissive(S, A)$ 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 automaton $A_S^- \times A^+$ such that $q_n = q_{sink}$. Like $checkSafety(S, A)$, $checkPermissive(S, A)$ is implemented as a reachability analysis.

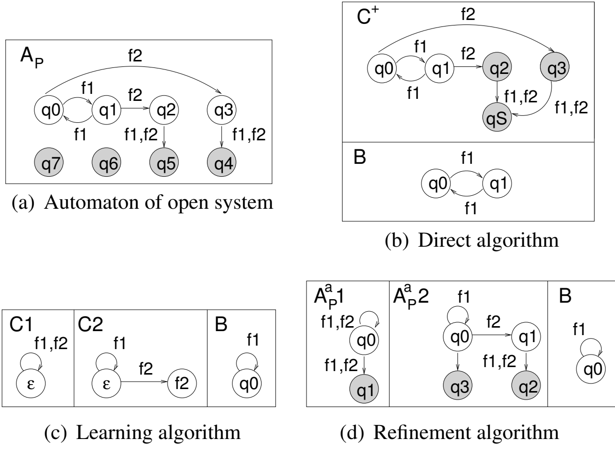
3. Three Algorithms for Interface Synthesis

We discuss three algorithms for synthesizing interfaces. Figure 2(a) shows an example automaton of an open system. The grey circles denote error states.

3.1. Direct Algorithm

Given an open system S, the direct algorithm $Direct(S)$ calls the procedure $Serialize(C)$ (shown in Algorithm 2), which adds state q_{sink} to the errorless automaton A_S^- and prunes backwards, starting from q_{sink}, to eliminate all states whose successors all lead to q_{sink}. We obtain a serial automaton B such that $L^\omega(B)$ is the safe and permissive interface for S. Fig. 2(b) shows an example of how the direct algorithm works. The grey circles represent the set Err. The error states are unreachable and not shown. The state qS is the sink q_{sink}.

¶ $t' \in \llbracket X' \rrbracket$ is obtained from $t \in \llbracket X \rrbracket$ by replacing all variables from X with their primed versions from X'.



(a) Automaton of open system

(b) Direct algorithm

(c) Learning algorithm

(d) Refinement algorithm

Figure 2. Example open system and the output of the three algorithms

Lemma 1. For an open system S, let C be the errorless automaton. Then, the ω-language $L^\omega(C)$ is safe and permissive for S.

Proof 1. We prove the lemma in two parts, both by contradiction.

- $L^\omega(C)$ **is safe:** Suppose an infinite word $w \in L^\omega(C)$ is not safe for S. Then there exists a shortest prefix w^j (w^j is the prefix of length j of word w) s.t. $s_0 \xrightarrow{w^j}_{\delta_S} s$ with $s(x_e) = \texttt{T}$. Let t be a state s.t. $s_0 \xrightarrow{w^{j-1}}_{\delta_S^-} t$. As w^j is the shortest prefix, we have $t(x_e) = \texttt{F}$. By construction of C, we know that $(t, f, s) \notin \delta_S^-$. As S is visibly deterministic, we know that there is no trace with word w on C. This contradicts our assumption that $w \in L^\omega(C)$. Hence, $L^\omega(C)$ is safe for S.
- $L^\omega(C)$ **is permissive:** Suppose an infinite word $w \notin L^\omega(C)$ is safe for S. Then there does not exist a prefix w^j of w s.t, $s_0 \xrightarrow{w^j}_{\delta_S} s$ with $s(x_e) = \texttt{T}$. By construction of C, we know that $s_0 \xrightarrow{w^j}_{\delta_S^-} s$ for all j. This contradicts our assumption that $w \notin L^\omega(C)$. Hence $L^\omega(C)$ is permissive for S.

Theorem 1. Given an open system S with variables X and input events Σ, the direct algorithm produces a serial automaton B such that $L^\omega(B)$ is the safe and permissive interface for S, in time linear in $|\Sigma|$ and exp. in $|X|$.

Note that if S is not visibly deterministic, then the pruning for serialization in the direct algorithm does not guarantee to result in a safe interface.

Algorithm 1 $Direct(S)$

Input: an open system $S = (X, \Sigma, s_0, \varphi, x_e)$
Output: a serial automaton B s.t. $L^\omega(B)$ is the safe and permissive interface for S
return $Serialize(A_S^-)$

Algorithm 2 $Serialize(C)$

Input: an automaton $C = (Q, \Sigma, q_0, \lambda)$
Output: a serial automaton B s.t. $L^\omega(B)$ is the largest closed subset of $L^\omega(C)$
Variables: an automaton C^+, a state $q_{sink} \notin Q$,
three state sets Err, $Wait$, $Pre \subseteq (Q \cup \{q_{sink}\})$
$C^+ :=$ input-enabled automaton $(Q \cup \{q_{sink}\}, \Sigma, q_0, \lambda^+)$ for C
$Err := \{q_{sink}\}$; $\quad Wait := Err$
while $Wait \neq \emptyset$ **do**
choose $s \in Wait$; $\quad Wait := Wait \setminus \{s\}$
$Pre := \{r \in Q \mid (r, f, s) \in \lambda^+$ for some $f \in \Sigma\}$
for each state $r \in Pre$ **do**
if $r \notin Err$ **and for all** $f \in \Sigma : r \xrightarrow{f}_{\lambda^+} s'$ and $s' \in Err)$ **then**
$Err := Err \cup \{r\}$; $\quad Wait := Wait \cup \{r\}$
return $(Q \setminus Err, \Sigma, q_0, \{(q, f, q') \in \lambda \mid q' \notin Err\})$

3.2. Learning Algorithm

Our second algorithm learns the interface language by asking membership and equivalence questions to the teacher (the open system). In a membership question, the algorithm asks whether a particular word is safe for S or not. In an equivalence question, the algorithm asks whether the language of the conjectured automaton $C = (Q, \Sigma, q_0, \lambda)$ is safe and permissive for the open system. To construct the conjecture 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 $(R \cup (R \cdot \Sigma)) \cdot E$ to $\mathbb{B}$. The set R is a set of representative words that lead from the initial state q_ϵ to all other states of the automaton C. For each word $r \in R$ that is safe for S, there exists a state q_r in the automaton C s.t. $q_\epsilon \xrightarrow{r}_\lambda q_r$. The set E is a set of experiment suffix words that distinguish the states. For all representative words $r_1, r_2 \in R$, there exists a word $e \in E$ s.t. only one of $r_1 \cdot e$ and $r_2 \cdot e$ is safe for S. The function G stores the results of the membership questions, i.e., maps a word $w \in (R \cup (R \cdot \Sigma)) \cdot E$ to $\texttt{T}$ if w is safe for S, and to $\texttt{F}$ otherwise. For a detailed description of the learning algorithm we refer to the paper by Alur et al. [1] (cf. also [2] and [7]). As we assume visibly deterministic open systems, our algorithm can and does incorporate a permissiveness check of the interface language.

Algorithm. The learning algorithm starts with R and E set to $\{\epsilon\}$, and G is initialized for every word in $(R \cup (R \cdot \Sigma)) \cdot E$ using membership queries (by procedure $memb(S, w)$).

Then, it checks whether the table (R, E, G) is closed (described in procedure $checkClosure(R, E, G)$). If not, the algorithm adds new representative words and rechecks for closure. Once (R, E, G) is closed, an automaton C is conjectured (done by $makeConjecture(R, E, G)$). Then, $L(C)$ is checked for safety and permissiveness of S (this check represents the equivalence question). If not, a counterexample is returned. The longest suffix of the counterexample (found by procedure $findSuffix(S, R, w)$) is added to E, and the algorithm rechecks for closure. The learning algorithm constructs an automaton C whose states correspond to the trace-equivalence classes of $Reach(A_S)$. Then, the algorithm calls the procedure $Serialize(C)$ to produce the minimal serial automaton B s.t. $L^\omega(B)$ is the safe and permissive interface for S. Figure 2(c) shows an example of how the learning algorithm works. The first two boxes show the two conjectured automata. $C2$ is the final conjecture, which is used to produce the serial automaton B.

Procedures used in the learning algorithm.

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

Time complexity. For the generation of a conjecture automaton with m states of an open system with k variables, the overall time complexity is in $O(m^4 \cdot |\Sigma| \cdot 2^{2k})$ in case the algorithm encounters worst (longest) counterexamples and in $O(m^2 \cdot |\Sigma| \cdot 2^k)$ when best counterexamples are seen. At the end, a call to $Serialize(C)$ takes $O(m \cdot |\Sigma|)$ time. Thus the learning algorithm has the worst-case time complexity $O(|\Sigma| \cdot 2^{6k})$ when the number of trace-equivalence classes is $O(2^k)$.

Theorem 2. Given an open system S with variables X and input events Σ and m trace-equivalence classes in $Reach(A_S)$, the learning algorithm produces the minimal serial automaton B, such that $L^\omega(B)$ is the safe and permissive interface for S, in time polynomial in $|\Sigma|$ and m, and exp. in $|X|$.

Algorithm 3 *Learning*(S)

Input: an open system $S = (X, \Sigma, s_0, \varphi, x_e)$
Output: a serial automaton B s.t. $L^\omega(B)$ is the safe and permissive interface for S
Variables: sets of words R and E over Σ, an array G that maps $R \cup R \cdot \Sigma \times E$ to $\mathbb{B}$, an automaton $C = (Q, \Sigma, q_0, \lambda)$, a trace $\alpha^\times$ of a product automaton

```
R := {ϵ};   E := {ϵ}
for each f ∈ Σ do
  G[ϵ, ϵ] := memb(S, ϵ·ϵ);   G[ϵ·f, ϵ] := memb(S, ϵ·f·ϵ)
while true do
  while (r_new := checkClosure(R, E, G)) ≠ YES do
    R := R ∪ {r_new}
    for each f ∈ Σ, e ∈ E do
      G[r_new · f, e] := memb(S, r_new · f · e)
  C := makeConjecture(R, E, G)
  if (α^× := checkSafety(S, C)) = YES then
    if (α^× := checkPermissive(S, C)) = YES then
      return Serialize(C)
  w := the word induced by trace α^×
  e_new := findSuffix(S, R, w);   E := E ∪ {e_new}
  for each r ∈ R, f ∈ Σ do
    G[r, e_new] := memb(S, r · e_new)
    G[r · f, e_new] := memb(S, r · f · e_new)
```

Note that if S is not visibly deterministic, the learning algorithm cannot check for permissiveness, although it can guarantee to provide the minimal safe interface.

3.3. Refinement Algorithm

The third algorithm that we discuss is based on the CEGAR approach.

Abstraction. An *abstraction* for an open system $S = (X, \Sigma, s_0, \varphi, x_e)$ is a set $X^a \subseteq X$ of variables, where $x_e \in X^a$. The abstraction hides the variables in $X \setminus X^a$. Given a state s over X^a and a state t over X, we define $t \preceq s$ if $t(x) = s(x)$ for all $x \in X^a$. An open system S and an abstraction X^a for S yield the *abstract open system* S^a. The semantics of S^a is given by the abstract automaton A^a_S and the set E^a_S of error states. The *abstract automaton* $A^a_S = (\llbracket X^a \rrbracket, \Sigma, s^a_0, \delta^a_S)$ has the set $\llbracket X^a \rrbracket$ of states, the initial state $s^a_0 \in \llbracket X^a \rrbracket$ such that $s_0 \preceq s^a_0$, and the transition relation $\delta^a_S = \{(s, f, s') \in \llbracket X^a \rrbracket \times \Sigma \times \llbracket X^a \rrbracket \mid$ there is a $(t, f, t') \in \delta_S$ with $t \preceq s$ and $t' \preceq s'\}$. The set E^a_S of error states is the set of states $s \in \llbracket X^a \rrbracket$ with $s(x_e) = \mathrm{T}$.

In this paper, we also propose an improvement over the original algorithm [5] that requires just one abstraction, instead of two as required by the original algorithm. Thus, our improvement greatly simplifies the original algorithm. Our algorithm is based on a new insight, which we formulate in the following two lemmas.

Lemma 2. If the language $L(A)$ of an automaton A is safe and permissive for S^a, then $L(A)$ is safe and permissive for S.

Proof 2. We prove the lemma in two parts, both by contradiction.

- **If $L(A)$ is safe for S^a, then $L(A)$ is safe for S:** Suppose $L(A)$ is not safe for S. Then, we know that there exists a state $(s, q) \in Reach_{A_S \times A}$ s.t. $s(x_e) = \text{T}$. Let w be the corresponding word. By construction of A_S^a, we know that there exists a state t s.t. $s_0^a \xrightarrow{w}_{\delta_S^a} t$ and $s \preceq t$. Thus, $t(x_e) = \text{T}$. This is a contradiction to our assumption that $L(A)$ is safe for S^a.
- **If $L(A)$ is permissive for S^a, then $L(A)$ is permissive for S:** Suppose $L(A)$ is not permissive for S. Then, we know that there exists a state $(s, q_{sink}) \in Reach_{A_S^- \times A^+}$. Let w be the corresponding word. By construction of A_S^{a-}, we know that there exists a state t such that $s_0^a \xrightarrow{w}_{\delta_S^{a-}} t$ and $s \preceq t$. Thus, $(t, q_{sink}) \in Reach_{A_S^{a-} \times A^+}$. This poses a contradiction to our assumption that $L(A)$ is permissive for S^a.

Lemma 3. Given the errorless automaton $C = (Q, \Sigma, q_0, \lambda)$ of an abstract open system S^a, if the language $L(C)$ is safe for S^a, then $L(C)$ is permissive for S^a. Moreover, S^a is visibly deterministic.

Proof 3. We know that the safety and permissiveness conditions are reachability questions on $A_S^a \times C$ and $A_S^{a-} \times C^+$, respectively. We note that automata C and A_S^{a-} are identical. Moreover, from construction of input-enabled automaton we know that if $(q, f, q_{sink}) \in \lambda^+$, then $(q, f, q') \in \delta_S^a$ with $q' \in E_S^a$. Thus, if there exists no state $(s, q) \in Reach_{A_S^a \times C}$ s.t. $s \in E_S^a$, then there exists no state $(s, q_{sink}) \in Reach_{A_S^{a-} \times C^+}$. Hence, $L(C)$ is permissive for S^a. Safety of $L(C)$ for S^a guarantees that there exists no word w that is not safe for S^a and $q_0 \xrightarrow{w}_{\lambda} q$ for some $q \in Q$. As C is the errorless automaton for S^a, this means that there exists no word w such that there exist two states s and t with $s_0 \xrightarrow{w}_{\delta_S^a} s$ and $s_0 \xrightarrow{w}_{\delta_S^a} t$ and $s(x_e) \neq t(x_e)$. Hence, S^a is visibly deterministic.

Algorithm. We start with an abstraction that has the error variable only, that is, $X^a = \{x_e\}$. We construct the abstract open system S^a and also its errorless automaton C. Then, we check whether $L(C)$ is safe for S^a. If yes, we know that S^a is visibly deterministic. Otherwise, we obtain a counterexample trace $\alpha^\times = \langle (s_0^a, q_0), f_0, \dots, (s_n^a, q_n) \rangle$ of the product automaton $A_S^a \times C$. Then, we use procedure $\mathit{findSpuriousTrace}(S, \alpha^\times)$ that first checks whether the word w induced by trace $\alpha_\times$ is safe for the open system S. If safe (unsafe), it declares the trace followed by the automaton A_S^a (resp. C) as spurious. A trace $\langle t_0, f_0, \dots, t_n \rangle$ of an abstract automaton A_S^a or A_S^{a-} is *spurious* if there exists no trace $\langle s_0, f_0, \dots, s_n \rangle$ of the automaton A_S such that $s_i \preceq t_i$ for $0 \leq i \leq n$. We add more variables to the abstraction X^a such that the spurious abstract trace $\langle t_0, f_0, \dots, t_n \rangle$ is eliminated, using Algorithm 5. The algorithm constructs a genuine trace $\langle s_0, f_0, \dots, s_n \rangle$ on A_S and its corresponding abstract trace, which induces the same word as the spurious trace. It locates the position i where the spurious abstract trace differs from the genuine abstract trace. It finds the set Y of states in A_S which cause the spurious abstract trace, and finds new variables in X such that if $s_{i-1} \preceq t$ then there does not exist a state $y \in Y$ with $y \preceq t$. Then, we reconstruct the abstract open system S^a and check whether it is visibly deterministic.

So, the refinement algorithm finds a visibly deterministic abstract open system S^a. Then, we run the direct algorithm on S^a to obtain a serial automaton B such that $L^\omega(B)$ is the safe and permissive interface for S. Figure 2(d) shows an example how the refinement algorithm works. The first box shows $A_S^a 1$ with the abstraction $X_a = \{x_e\}$. Adding one more variable gives $A_S^a 2$, whose abstract open system

---

Algorithm 4 $\mathit{AbstRefine}(S)$

Input: an open system $S = (X, \Sigma, s_0, \varphi, x_e)$
Output: a serial automaton B s.t. $L^\omega(B)$ is the safe and permissive interface for S
Variables: an abstraction X^a, an abstract open system S^a, an automaton C, a finite trace $\alpha^\times$ of a product automaton, and a finite trace α of an automaton

```
X^a := {x_e}
while X^a ≠ X do
   S^a := refineAbstraction(S, X^a);   C := A_S^{a-}
   if (α^× := checkSafety(S^a, C)) = YES then
      return Direct(S^a)
   else
      α := findSpuriousTrace(S, α^×)
      X^a := getNewVars(S, α, X^a)
return Direct(S^a)
```

Algorithm 5 $\mathit{getNewVars}(S, \alpha, X^a)$

Input: an open system $S = (X, \Sigma, s_0, \varphi, x_e)$, a spurious trace $\langle t_0, f_0, \dots, t_n \rangle$ on the abstract automaton A_S^a, and an abstraction X^a
Output: a new abstraction $X_{new} \subseteq X$ that eliminates the given spurious trace
Variables: states $s, s' \in [\![X]\!]$ and $t, t_s, t' \in [\![X^a]\!]$, a set $Y \subset [\![X]\!]$ of states, and an event $f \in \Sigma$

```
s := s_0;   t := s_0^a
for i := 1 to n do
   t_s := t_i;   f := f_{i-1}
   let s' ∈ [[X]] such that s -f->_{δ_S} s'
   let t' ∈ [[X^a]] such that u ⪯ s'
   if t' ≠ t_s then
      Y := {u ∈ Reach(A_S) | u ⪯ t and y ⪯ t_s with
      u -f->_{δ_S} y}
      return splitState(s, Y, X^a)
   s := s';   t := t'
```

is found to be visibly deterministic. Thus, B is computed, using the direct algorithm.

Procedures used in the refinement algorithm.

- $\mathit{splitState}(s, Y, X^a)$ for $s \in [\![X]\!]$, $Y \subseteq [\![X]\!]$, and X^a being the current abstraction, finds a set $X_r \subseteq X$ of variables s.t. there is no state $y \in Y$ s.t. $y \preceq t$ where $t \in [\![X_a \cup X_r]\!]$ with $s \preceq t$. It returns $X_a \cup X_r$.
- $\mathit{findSpuriousTrace}(S, \alpha^\times)$ first checks whether the word w induced by the trace $\alpha^\times$ of the product automaton is safe for S. If yes, it returns the trace $\langle s_0^a, f_0, \dots, s_n^a \rangle$ of A_S^a in $\alpha^\times$. Otherwise, it returns the trace $\langle q_0, f_0, \dots, q_n \rangle$ of C in $\alpha^\times$.
- $\mathit{refineAbstraction}(S, X^a)$ returns the abstract open system S^a.

Time complexity. Let X and abstraction X^a be sets of k and c variables, respectively. An iteration of the algorithm requires $O(|\Sigma| \cdot 2^{k+c})$ time. At the end of the refinement procedure, the call to procedure $\mathit{Direct}(S^a)$ requires time $O(|\Sigma| \cdot 2^l)$, where l is the number of variables in the abstraction that was sufficient to prove safety. Thus, the worst-case time complexity is $O(|\Sigma| \cdot 2^{2k})$, which is encountered at the finest abstraction (with $k-1$ variables). The output automaton produced by the refinement algorithm depends on the order of refinement of the variables. Finding the coarsest abstraction that gives a visibly deterministic open system is an NP-hard problem [3].

Theorem 3. Given an open system S with variables X and input events Σ, $\mathit{AbstRefine}(S)$ produces a serial automaton B with $O(2^l)$ states, where l is the size of an abstraction that suffices to prove safety, such that $L^\omega(B)$ is the safe and permissive interface for S, in time polynomial in $|\Sigma|$ and exp. in $k + l$.

However, note that the abstraction found by the refinement algorithm that suffices to prove safety may not be of minimal size.

4. Theoretical Separation and Experimental Evaluation

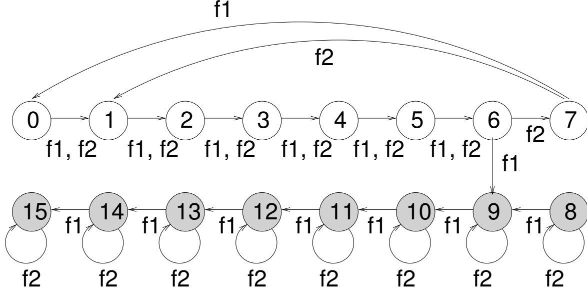
We describe various theoretical classes of examples that manifest the difference in the working of the three algorithms presented in the previous section. Also, we experiment with some practical examples based on the theoretical separation. The experiments suggest that the three algorithms are important in their own right and it is worthwhile to understand their working properly for efficient usage. The objective of the resemblance of the real world examples to the theoretical ones is to suggest the practical importance of the theoretically distinguishing examples.

Distinguishing examples. We consider systems with k variables. We denote the set of states as $\{s_0, s_1, \dots s_{2^k-1}\}$. 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 latter half are error states. We consider all pairs of the direct (**D**), the learning (**L**), and the refinement (**R**) algorithm. We evaluate on both metrics: time complexity and size of output automaton. We provide one example family for all cases where one algorithm performs better than another one. We show graphical examples with $k = 4$ in Fig. 3, which can be scaled to arbitrary k. For sake of clarity, we assume that the refinement algorithm finds the minimal abstraction in each case.

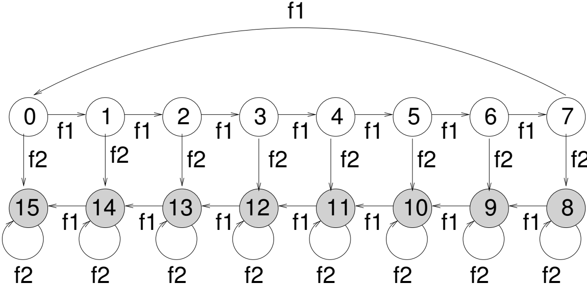
- **D** *beats* **L** *and* **R** *in time.* For the open system in Fig. 3(a), the number of trace-equivalence classes is exponential in k. The direct algorithm requires $O(2^k)$ time. The learning algorithm encounters worst counterexamples of size $O(2^{2k})$, and thus requires time $O(2^{6k})$. The refinement algorithm requires $O(2^{2k})$ time. The size of the automaton produced is $O(2^k)$ for all algorithms.
- **L** *and* **R** *beat* **D** *in output size.* For the open system in Fig. 3(b), the direct algorithm produces an output of size $O(2^k)$ whereas the learning algorithm produces an output of only one state as there are only two trace-equivalence classes. The refinement algorithm also produces an automaton with one state as all $k-1$ variables except the first one can be abstracted away. The time complexity for all algorithms is $O(2^k)$.
- **L** *beats* **R** *in time and output size.* The open system in Fig. 3(c) has three trace-equivalence classes. Thus the learning algorithm requires $O(2^k)$ time and produces an automaton with 2 states. On the other hand, the refinement algorithm requires $O(2^{2k})$ time and produces an automaton with $O(2^k)$ states, as it needs to refine the first $k-1$ variables.
- **R** *beats* **L** *in time.* For the open system in Fig. 3(d), the number of trace-equivalence classes is exponential in the number of variables that need refinement. The learning algorithm requires $O(2^{2k})$ time whereas the refinement algorithm requires only $O(2^k)$ time. The size of output automaton is $O(2^c)$ in both cases, where c is the number of variables that have to be refined.

Implementation. We implemented explicit and symbolic versions of the three algorithms in C++. The symbolic algorithms were implemented using a BDD package.[1] We experimented with some common real world systems that

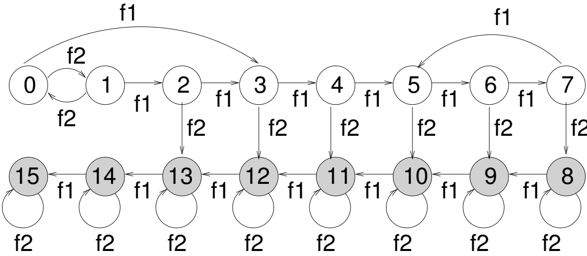
1 Available at http://mtc.epfl.ch/~beyer/CrocoPat



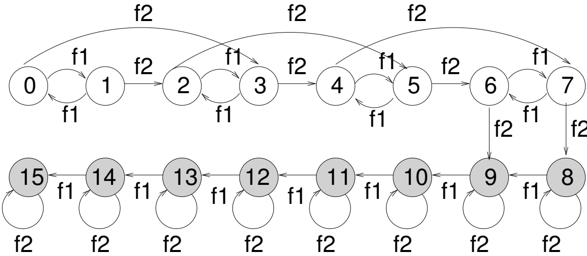
(a) Alg. **D** performs faster than **L** and **R**



(b) Alg. **L** and **R** produce smaller automaton than **D**



(c) Alg. **L** performs faster and produces a smaller automaton than **R**



(d) Alg. **R** performs faster than **L**

Figure 3. Examples of open systems where one algorithm performs better than the others. The grey circles denote the error states

bear a similarity with the theoretically separating examples discussed above. Table 1 reports the results of our experiments. The explicit versions of the learning and refinement algorithms are always more expensive than the explicit direct algorithm (cf. the theorems). The time of the reachability computation is high for the symbolic case because the open system is specified explicitly for all algorithms. We claim that when the input system is provided symbolically, the time required by the symbolic algorithms would be much lower than that required by the explicit algorithms. We use different examples of open systems to assess the performance of all six algorithms.

```
void prev(BitArray b){
 if (!b.valid)
  b.valid = true;
 if (b.ptr > 0)
  b.ptr--;
 else b.ptr = MAX; }
```

```
void next(BitArray b){
 if (!b.valid)
  b.valid = true;
 if (b.ptr < MAX)
  b.ptr++;
 else b.ptr = 0; }
```

```
void access(BitArray b){
 b.valid = false; }
```

```
void modify(BitArray b){
 if (!b.valid)
  b.error = true;
 else {
  b.valid = false;
}}
```

Figure 4. An example of a bit array manipulator

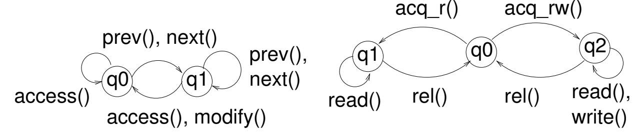


Figure 5. Serial automata produced by learning and refinement algorithms for bit array manipulator (left) and device manager (right)

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

Counter. Consider a system that implements a counter with a maximum value. The counter is encoded with k boolean variables, one for error, and remaining variables to encode the value of the counter. An invocation of $next$ increments the counter and an invocation of $prev$ decrements the counter. An error occurs in the following two cases: $prev$ is invoked when the counter is set to 0, or $next$ is invoked when the counter is at $2^{k-1} - 1$. The counter is initially set to 0. All the algorithms produce an automaton of size 2^{k-1}. Due to low time complexity, the direct algorithm is considered best. In general, the direct algorithm performs best when the number of trace-equivalence classes of automaton A_S is of order of the number of states in A_S.

Refinement performs best. The following systems are similar to Fig. 3(d).

- *Bit array manipulator.* Consider the open system shown in Fig. 4. The bit array manipulator is encoded by k boolean variables, one for error, one for validity of update operation and remaining bits for current location.
- *Device manager.* Consider the open system shown in Fig. 1, which implements device access for read/write operations. The device manager is encoded by k boolean variables, one for error, one for the read flag, one for the write flag and remaining variables for the location currently being accessed by the device.

Table 1. Run-time (in seconds) and output automaton (FA) size of different algorithms. The run-time is measured on a 3.0 GHz Pentium IV machine with 1 GB memory. The dash indicates that the process needed more than 30 minutes. In such cases, the automaton size is determined theoretically. For the data stream experiment, the variables are given as $k(h, d)$.

Num. of	Reach. comp.		Direct			Learning			Refinement		
vars k	Expl. time	Symb. time	Expl. time	Symb. time	FA size	Expl. time	Symb. time	FA size	Expl. time	Symb. time	FA size
Counter											
9	0.001	0.01	0.005	0.09	256	–	–	256	1.01	0.25	256
10	0.002	0.02	0.01	0.22	512	–	–	512	3.92	0.69	512
11	0.004	0.06	0.02	0.48	1024	–	–	1024	16.60	2.16	1024
12	0.009	0.12	0.04	1.11	2048	–	–	2048	69.1	8.29	2048
13	0.02	0.24	0.09	2.54	4096	–	–	4096	310.1	30.64	4096
Bit array manipulator											
12	0.005	0.08	0.04	0.96	1026	6.72	83.13	2	0.09	0.01	2
14	0.02	0.39	0.14	4.91	4098	108.84	1589.25	2	0.39	0.04	2
15	0.03	0.89	0.30	11.02	8194	455.96	–	2	0.80	0.08	2
16	0.07	2.08	0.63	25.04	16386	–	–	2	1.61	0.17	2
17	0.14	5.50	1.33	55.50	32770	–	–	2	3.28	0.29	2
Device manager											
14	0.03	0.28	0.21	5.57	4097	5.05	1.48	3	0.91	0.09	3
15	0.05	0.65	0.45	12.55	8193	10.60	3.32	3	1.83	0.19	3
16	0.11	1.58	0.90	28.33	16385	22.32	7.70	3	3.76	0.45	3
17	0.22	4.37	1.91	62.90	32769	47.82	18.76	3	7.68	1.25	3
18	0.46	9.13	3.95	134.96	65537	105.23	44.38	3	15.66	1.86	3
Data stream											
14(2,12)	0.003	0.08	0.04	0.88	1028	0.48	0.64	2	9.75	1.09	257
14(4,12)	0.01	0.37	0.14	4.52	4112	2.08	3.07	2	10.24	1.18	257
15(8,13)	0.04	0.85	0.31	10.42	8448	4.34	7.03	2	1.25	0.14	33
15(13,13)	0.06	1.35	0.55	20.54	16384	4.38	9.28	2	1.52	0.11	2
17(13,15)	0.18	5.97	1.58	64.35	40960	19.30	41.98	2	4.59	0.31	5

The automata generated by the refinement and learning algorithms for these two examples are shown in Fig. 5. The first few (two for bit array manipulator and three for device manager) variables provide a sufficient abstraction to produce a visibly deterministic open system. Thus, the refinement algorithm performs much faster than the learning algorithm and also outputs the minimal automata. The direct algorithm, though fast, produces automata of size exponential in k. Thus, the refinement algorithm performs the best in these practical examples.

Learning performs best. The following system is similar to Fig. 3(c).

A data stream. Consider a data stream with a header of length 2^h and data of length 2^d, where $h \leq d$. The data stream uses k boolean variables, one for error, and the remaining $k-1$ variables for the pointer location ($k = d+2$). A call to $FirstHeader$ takes the pointer to the first header bit, and a call to $FirstData$ takes the pointer to the first data bit. An invocation of $Next$ moves the pointer within the header or data in a cyclic way (if the pointer is currently at last header (data) bit, it is taken to first header (data) bit). An invocation of $Write$ results in an error, if the pointer points into the header region. The direct algorithm performs the fastest, but the size of the output automaton is exponential in k. If $h \ll d$, then the learning algorithm performs faster and produces much smaller automata than the refinement algorithm. The refinement algorithm produces the minimal automaton (and thus is a good choice) if $h = d$. In general, the learning algorithm is the first choice for such a case.

5. Conclusions

We formalized and implemented three algorithms for interface synthesis in a uniform framework. For each algorithm, we identified classes of open systems for which the algorithm is best suited for interface synthesis. The direct algorithm has the advantage of low time complexity in scenarios when the size of the minimal automaton is large, but tractable. The learning algorithm always produces the minimal deterministic automaton whose language is the safe and permissive interface. Thus, it performs best when the number of trace-equivalence classes is much smaller than the state space. The refinement algorithm is particularly efficient when many variables can be hidden in the interface automaton. Also, the refinement algorithm provides the flexibility to stop when the size of the abstraction becomes so big that further refinement is more expensive than the direct algorithm. We may also combine the algorithms in other ways, e.g., by first constructing a visibly deterministic abstract open system, and then use the learning algorithm on the abstract system rather than on the concrete system.

References

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

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

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

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

[5] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proc. FSE*, pages 31–40. ACM, 2005.

[6] R. Passerone, J.A. Rowson, and A.L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *DAC*, pages 8–13, 1998.

[7] R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[8] J.A. Rowson and A.L. Sangiovanni-Vincentelli. Interface-based design. In *DAC*, pages 178–183, 1997.