

# Learning Component Interfaces with May and Must Abstractions

Rishabh Singh<sup>1,\*</sup>, Dimitra Giannakopoulou<sup>2</sup>, and Corina Păsăreanu<sup>2</sup>

<sup>1</sup> MIT CSAIL/ MCT Inc.

<sup>2</sup> CMU/ NASA Ames

**Abstract.** Component interfaces are the essence of modular program analysis. In this work, a component interface documents correct sequences of invocations to the component’s public methods. We present an automated framework that extracts finite *safe*, *permissive*, and *minimal* interfaces, from potentially infinite software components. Our proposed framework uses the L\* automata-learning algorithm to learn finite interfaces for an infinite-state component. It is based on the observation that an interface *permissive* with respect to the component’s must abstraction and *safe* with respect to its may abstraction provides a precise characterization of the legal invocations to the methods of the concrete component. The abstractions are refined automatically from counterexamples obtained during the reachability checks performed by our framework. The use of must abstractions enables us to avoid an exponentially expensive determinization step that is required when working with may abstractions only, and the use of L\* guarantees minimality of the generated interface. We have implemented the algorithm in the ARMC tool and report on its application to a number of case studies including several Java2SDK and J2SEE library classes as well as to NASA flight-software components.

## 1 Introduction

Component interfaces are a central concept in component-based software engineering. In current practice, interfaces typically describe the services that a component *provides* and *requires* at a purely syntactic level. However, the need has been identified for interfaces that document richer aspects of component behavior. For example in this work, as in others [1,5,8,11,12,16], interfaces describe correct sequences of invocations to public methods of a component. Richer interfaces can serve as a documentation aid to application programmers, but can also be used by verification tools in checking that the components are invoked correctly within a system. In fact, interfaces are key for modular program analysis [8,11,12]. They reduce the task of verifying a system consisting of a component and a client, to the more tractable task of verifying that the client satisfies the component’s interface.

---

\* Work done while the author, supported by MCT Inc., was visiting NASA Ames.

Given the source-code of a library component  $C$ , we address the problem of extracting a precise component interface in the form of a deterministic finite-state automaton (DFA)<sup>1</sup>, labeled with the public method names of  $C$ . By precise, we mean *safe* and *permissive*. An interface is safe if it accepts no illegal sequence of calls to  $C$ , and permissive if it includes all the legal sequences of calls to  $C$  [16]. In contrast to our previous work [12], we combine interface generation algorithms with predicate abstraction techniques, that allows us to handle components with very large or infinite state spaces. The novelty of our proposed algorithms lies in the fact that we use a combination of under-, and over-approximations of the component behavior, in the form of must and may abstractions, respectively. Our approach is based on the observation that an interface that is safe with respect to the may abstraction and permissive with respect to the must abstraction is safe and permissive with respect to  $C$  itself. We use the L\* learning algorithm [4] to generate safe and permissive interfaces for  $C$ , by iteratively checking may and must abstractions of  $C$ . These abstractions are gradually refined during the learning process, based on counterexamples. If the algorithms terminate, then the returned interface is the *minimal* DFA capturing the precise interface for  $C$ .

Extended interfaces can be difficult to characterize precisely without the help of automated tools, making interface generation an area of active research [1,5,16]. The approaches closest to ours are those presented in [1,16]. Both approaches construct only over-approximations of the component behavior, which may be non-deterministic. Checking permissiveness when (abstracted) components are non-deterministic requires a potentially expensive determinization step. Alur et al. [1] avoid this step by using heuristics, and therefore cannot guarantee permissiveness of the generated interfaces. On the other hand, Henzinger et al. [16] build “abstract regions”, which is equivalent to performing a determinization step. Their abstractions are subsequently checked for safety and permissiveness. These steps cannot be combined in an on-the-fly algorithm, so the complete abstract reachability graph needs to be constructed, even if a counterexample exists early in the search.

Furthermore, the abstraction mechanisms in [16] cannot guarantee minimal interfaces. Even if these interfaces were to be minimized, this approach would suffer from potentially large intermediate interfaces that subsequently get compacted. This latter problem is more pronounced in the presence of the determinization step, which is exponential, in the worst case. In contrast, L\*-based approaches like ours and [1] directly generate minimal interfaces. Note however that the technique by [1] does not provide criteria to automatically detect the need for abstraction refinement. Their refinements are based on inspection of the generated interfaces, and are performed manually.

**Contributions.** We present a framework for automated generation of *minimal*, *safe* and *permissive* interfaces for large or infinite-state components. The framework uses L\* with automatically generated and refined may and must abstractions of the component behavior. It guarantees permissiveness without

---

<sup>1</sup> In this work, we assume that component interfaces have a regular language. We therefore do not consider components methods with recursive invocations.

requiring determinization, and performs all checks on-the-fly. We present a basic algorithm and also an optimized version that re-uses results across abstraction-refinement iterations. We also describe the implementation of our algorithm in ARMC, and the application to the benchmarks presented in [1,16], as well as new benchmarks including J2SEE classes and NASA software components.

**Other related work.** Work on predicate abstraction for modal transition systems, e.g. [13], similarly distinguishes between may and must transitions. However, to the best of our knowledge, the use of may and must abstractions for interface generation is novel. Other approaches generate interfaces by using static analysis [27], or a combination of static and dynamic analyses [28], or by extracting information from sample execution traces [3]. All these techniques generate approximate interfaces, as opposed to our work that aims at producing precise interfaces that provide correctness guarantees.

Interface generation is related to assume-guarantee reasoning [2,10,18,23], since component interfaces can be used as assumptions in this context. Shoham et al. [26] describe a compositional framework for modal transition systems, based on techniques taken from the 3-valued game-based model checking for abstract models [9,14]. Those approaches do not use explicit interfaces (or assumptions). Finally, recent work [15] uses may and must information in the form of procedure summaries in a compositional framework that performs program analysis.

## 2 Example

Our running example, taken from [16], is illustrated in Figure 1. It consists of a component  $C$  with 3 static variables and 6 public library methods. Variable  $e$  defines the error states in the component ( $e \neq 0$ ), variable  $a$  denotes the possession of lock and variable  $x$  enables method `write`. Methods `acq/relx` and their variations `acqx/relx` are used to acquire/release a lock, respectively. Methods `read` and `write` are used to access and update the shared memory, respectively.

It can be observed that  $C$  enforces several requirements such as `read` can only be called after `acq` or `acqx`. Similarly `write` can be called safely only after calling `acqx`. Once `acq` is called, it can only be called again after calling `rel` or `relx`. The interface  $A$  for  $C$  should capture all such correct sequences of invocation of public methods and reject the incorrect ones.

<pre>void rel(){   a = NULL;   return;}  void relx(){   a = NULL;   x = 0;   return;}</pre>	<pre>void acq(){   if(a==NULL)     a=get_lock();   else     e=1;   return;}</pre>	<pre>void read(){   if(a!=NULL)     m_read(a);   else     e=1;   return;}</pre>	<pre>void acqx(){   if(a==NULL){     a=get_lock();     x=1;}   else     e=1;   return;}</pre>	<pre>void write(){   if(x!=0)     m_write(a);   else     e=1;   return;}</pre>
---	---	---	---	--

Fig. 1. Read-write-acq example

### 3 Preliminaries

**Components and Interfaces.** A component  $C = (X_s, F, s_0, P_{err}, \Sigma)$  consists of: a set  $X_s$  of static global variables shared across the methods ( $[[X_s]]$  denotes the valuations of variables in  $X_s$  and represents the states of the component); a set  $F$  of *library methods*; initial state  $s_0$  of the component,  $s_0 \in [[X]]$ ; a global set  $P_{err}$  of error predicates over variables in  $X$ ; and a finite alphabet  $\Sigma$  of the method names. The error predicates denote the error conditions in the library such as runtime exceptions, assertion violations etc. A component state  $s \in [[X_s]]$  is an *error state* if  $s$  satisfies an error predicate.

*Example 1.* The example component in Figure 1 can be expressed as  $C = (X, F, s_0, P_{err}, \Sigma)$  where  $X = \{a, x, e\}$ ,  $F$  is the set of CFAs for methods (described below), the start state  $s_0 = \{a = NULL, x = 0, e = 0\}$ , the error predicate  $P_{err} = \{e \neq 0\}$  and alphabet set  $\Sigma = \{\text{acq, read, rel, write, relx, acqx}\}$ .

Every library method  $f \in F$  is represented as a *control-flow automaton* (CFA)  $f = (X_s, X_l, Q, q_s, q_r, \mathcal{T})$  consisting of a disjoint set of static variables ( $X_s$ ) and local variables ( $X_l$ ), a set  $Q$  of *control locations*; a start location  $q_s \in Q$ , a return location  $q_r \in Q$ , and a finite set of *method transitions*  $\mathcal{T}$ . Each transition  $\tau \in \mathcal{T}$  is labeled with a *from* location  $q_{from} \in Q$ , a *to* location  $q_{to} \in Q$  and the method statement operation represented as a guarded command,  $g(\mathbf{x}) \mapsto \mathbf{x} = e(\mathbf{x})$  where  $g(\mathbf{x})$  is a guard and  $e(\mathbf{x})$  are updates to variables in  $\mathbf{x} \in (X_s \cup X_l)$ . We use a special no-op skip transition to model multiple return locations with one return location.

**CFA and Component Semantics.** We give the definition of CFA semantics in terms of method transitions and of component semantics in terms of method calls. A state in the CFA is modelled as  $(q, s)$  where  $q \in Q$  is a control location and  $s \in [[(X_s \cup X_l)]]$  represents the valuation of (both global and local) variables in that state, whereas a state in the component is represented by  $s$  where  $s \in [[X_s]]$  denotes the valuation of (only global) variables in that state.

A binary transition relation  $\rho_\tau \subseteq (Q \times [[X_s \cup X_l]])^2$  captures the semantics of the transitions  $\tau \in \mathcal{T}$  in a CFA.  $((q, s), (q', s')) \in \rho_\tau$  if  $q = \tau.q_{from}$ ,  $q' = \tau.q_{to}$ ,  $s \models \tau.g$  and  $s' = \tau.e(s)$ . We write  $s \xrightarrow{\tau} s'$  for  $((\tau.q_{from}, s), (\tau.q_{to}, s')) \in \rho_\tau$ .

Let  $s \circ t$  denote the combination of valuations  $s \in [[X_s]]$  (static variables) and  $t \in [[X_l]]$  (local variables). The transition relation for the component  $\delta_C \subseteq [[X_s]] \times \Sigma \times [[X_s]]$  denotes the successful termination of method  $f$  when applied on some state  $s \in [[X_s]]$  resulting in state  $s' \in [[X_s]]$ . It is defined as follows:  $(s, f, s') \in \delta_C$  if  $\exists$  sequence  $(q_1, (s_1 \circ t_1)), (q_2, (s_2 \circ t_2)), \dots, (q_n, (s_n \circ t_n))$  such that  $(q_s, s) = (q_1, s_1)$  ( $q_1$  is the start location of  $f$ ),  $(q_r, s') = (q_n, s_n)$  ( $q_n$  is the return location of  $f$ ), and  $\bigwedge_{1 \leq i \leq n-1} ((q_i, (s_i \circ t_i)), (q_{i+1}, (s_{i+1} \circ t_{i+1}))) \in f_c.\rho_\tau$ ,  $s_i \in [[X_s]]$ ,  $t_i \in [[X_l]]$  (every transition in the sequence is a valid transition in  $f.\mathcal{T}$ ). For simplicity we assume error states have no outgoing method transitions, except for return. We write  $s \xrightarrow{f} s'$  for  $(s, f, s') \in \delta_C$ .

The semantics of the component  $C$  is captured by a (possibly infinite) deterministic transition system  $S_C = ([[X]], \Sigma, s_0, \delta_C)$ . A *component sequence*

$\text{Seq} = f_1, f_2, \dots, f_n$  is the sequence of method calls corresponding to a computation  $s_0, s_1, \dots, s_n$  of  $S_C$  such that  $\forall i \ s_i \in [[X]], (s_{i-1}, f_i, s_i) \in \delta_C$ . An error sequence is a component sequence that leads the component to an error state. The language  $L(S_C) \subseteq \Sigma^*$  denotes all the component sequences of  $C$ ;  $L^E(S_C) \subseteq L(S_C)$  denotes the language of error sequences, and  $L^{\text{safe}}(S_C)$  denotes the language of *safe method sequences* which is defined to be the complement of  $L^E(S_C)$ , i.e.  $L^{\text{safe}}(S_C) = \overline{L^E(S_C)}$ . Note that while  $L(S_C)$  and  $L^E(S_C)$  contain only *feasible* traces,  $L^{\text{safe}}(S_C)$  may contain both feasible and infeasible component sequences.

**Safe and Permissive Interfaces.** An interface for a library component  $C$  is a prefix-closed regular set over the alphabet  $\Sigma$  of library method names. We represent interfaces as (deterministic) finite state automata  $A = (Q, \Sigma, q_0, \delta)$  where:  $Q$  is a finite non-empty set of accept states;  $\Sigma$  is a finite alphabet of method names;  $q_0 \in Q$  is the initial state; and the transition relation  $\delta \subseteq Q \times \Sigma \times Q$  (the set of accepting states is  $Q$ ).  $L(A)$  is the set of words accepted by  $A$ . We let  $L^E(A) = \overline{L(A)}$  denote the set of error traces of  $A$ .  $L^E(A)$  is the language accepted by automaton  $A_{\text{err}}$ , representing  $A$  completed with an error state which is the only accepting state, i.e.,  $A_{\text{err}} = (Q', \Sigma, q_0, \delta')$ , where  $Q' = Q \cup \{\text{err}\}$  and  $\delta' = \delta \cup (q, a, \text{err}) \ \forall q, q' \in Q, a \in \Sigma : (q, a, q') \notin \delta$ .

Interface  $A$  is **safe** if every word  $w \in L(A)$  is a safe sequence of method calls in  $C$ , i.e.  $L(A) \subseteq L^{\text{safe}}(S_C)$ ; equivalent to  $L^E(S_C) \subseteq L^E(A)$  or  $L(A) \cap L^E(S_C) = \emptyset$ .

Interface  $A$  is **permissive** if it accepts *all* safe sequences of method calls in  $C$ , i.e.  $L^{\text{safe}}(S_C) \subseteq L(A)$ ; equivalent to  $L^E(A) \subseteq L^E(S_C)$  or  $L^{\text{safe}}(S_C) \cap L^E(A) = \emptyset$ .

From the above definitions, since  $L^E(S_C) \subseteq L^E(A)$  and  $L^E(A) \subseteq L^E(S_C)$ , it follows that  $L^E(S_C) = L^E(A)$ .

*Example 2.* For the component in Figure 1, the string  $\sigma_1 = (\text{acq}, \text{read}, \text{rel}) \in L^{\text{safe}}(S_C)$  and  $\sigma_1 \in L(S_C)$  as the corresponding method sequence is **safe** for the component. The string  $\sigma_2 = (\text{read}, \text{acq}, \text{rel}) \in L^E(S_C)$  as the method sequence causes the component to be in an error state.

**Composition.** Let  $S_C = ([[X]], \Sigma, s_0, \delta_C)$  be the transition system capturing the semantics of library component  $C$ , and  $A = (Q, \Sigma, q_0, \delta)$  be an interface automaton. The composite transition system  $G = S_C \parallel A$  obtained by composing  $S_C$  and  $A$  is defined as  $G = (Q^\times, \Sigma, q_0^\times, \delta^\times)$ , where  $Q^\times = Q \times [[X]]$ ,  $q_0^\times = (q_0, s_0)$ , and  $\delta^\times = \{((q, s), f, (q', s')) \mid (q, f, q') \in \delta \text{ and } (s, f, s') \in \delta_C\}$ .

**Abstraction.** We build *may* and *must* abstractions of software components using predicate abstraction – a special instance of abstract interpretation [7] that maps a potentially infinite state transition system into a finite state transition system via a finite set of predicates  $\text{Preds} = \{p_1, \dots, p_n\}$  over the program variables. We require  $P_{\text{err}} \subseteq \text{Preds}$ . An abstract state  $a \subseteq \text{Preds}$  is an *error state* if it satisfies an error predicate.

An abstraction function  $\alpha$  maps a concrete program state  $s$  to a set of predicates that hold in  $s$ :  $\alpha(s) = \{p \in \text{Preds} \mid s \models p\}$ . For transition  $\tau \in \mathcal{T}$  of method  $f$ , we define *may* and *must* transitions;  $a, a'$  denote abstract states,  $s, s'$  denote concrete states:

- $a \xrightarrow{\tau}_{must} a'$  iff  $\forall s$  s.t.  $\alpha(s) = a$ ,  $\exists s'$  s.t.  $\alpha(s') = a'$  and  $s \xrightarrow{\tau} s'$ .
- $a \xrightarrow{f}_{must} a'$  iff  $\forall s$  s.t.  $\alpha(s) = a$ ,  $\exists s'$  s.t.  $\alpha(s') = a'$  and  $s \xrightarrow{f} s'$ .
- $a \xrightarrow{\tau}_{may} a'$  iff  $\exists s$  s.t.  $\alpha(s) = a$  and  $\exists s'$  s.t.  $\alpha(s') = a'$  and  $s \xrightarrow{\tau} s'$ .
- $a \xrightarrow{f}_{may} a'$  iff  $\exists s$  s.t.  $\alpha(s) = a$  and  $\exists s'$  s.t.  $\alpha(s') = a'$  and  $s \xrightarrow{f} s'$ .

Given component  $C$  with transition system  $S_C$ , the must and may abstractions with respect to the set of abstract predicates  $\text{Preds}$  are defined as  $S_{C, \text{Preds}}^{must} = (2^{\text{Preds}}, \Sigma, \alpha(s_0), \longrightarrow_{must})$  and  $S_{C, \text{Preds}}^{may} = (2^{\text{Preds}}, \Sigma, \alpha(s_0), \longrightarrow_{may})$ , respectively. We sometimes write  $S_C^{must}$  or  $S_C^{may}$  when  $\text{Preds}$  is clear from the context.

Algorithms for computing may and must abstractions with the help of a theorem prover are given in e.g. [24]. Note that the set of may transitions is a super-set of the must transitions. We also note from the above definitions it follows that the may and must abstractions define simulations [21] between  $S_C^{must}$  and  $S_C$ , and between  $S_C$  and  $S_C^{may}$ , respectively. Since simulation implies trace inclusion, we have the following characterization of under- and over- approximations (that we will use later):

**Proposition 1.**  $L(S_C^{must}) \subseteq L(S_C) \subseteq L(S_C^{may})$ . Furthermore,  $L^E(S_C^{must}) \subseteq L^E(S_C) \subseteq L^E(S_C^{may})$ .

**Weakest Precondition.** For automated abstraction refinement, we use weakest precondition calculations over counterexample traces [24]. Let  $\phi$  be a predicate characterizing a set of states. The weakest precondition of  $\phi$  with respect to transition  $\tau$  is  $\text{wp}(\phi, \tau) = \forall s'. (s \xrightarrow{\tau} s' \Rightarrow \phi(s'))$ , and it characterizes the largest set of states whose successors by transition  $\tau$  satisfy  $\phi$ .

**The  $L^*$  Algorithm.**  $L^*$  was developed by Angluin [4] and later improved by Rivest and Schapire [25].  $L^*$  learns an unknown regular language  $U$  over alphabet  $\Sigma$  and produces a *minimal deterministic* finite state automaton (DFA) that accepts it.  $L^*$  interacts with a *Minimally Adequate Teacher* that answers two types of questions from  $L^*$ . The first type is a *membership query* asking whether a string  $\sigma \in \Sigma^*$  is in  $U$ . For the second type, the learning algorithm generates a *conjecture*  $A$  and asks whether  $L(A) = U$ . If  $L(A) \neq U$  the Teacher returns a counterexample, which is a string  $\sigma$  in the symmetric difference of  $L(A)$  and  $U$ .  $L^*$  is guaranteed to terminate with a minimal automaton  $A$  for  $U$ . If  $A$  has  $n$  states,  $L^*$  makes at most  $n - 1$  incorrect conjectures. The number of membership queries made by  $L^*$  is  $O(kn^2 + n \log m)$ , where  $k$  is the size of  $\Sigma$ ,  $n$  is the number of states in the minimal DFA for  $U$ , and  $m$  is the length of the longest counterexample returned when a conjecture is made.

## 4 Interface Generation

Let  $C$  be a component corresponding to a potentially infinite-state transition system  $S_C$ . From now on, for simplicity, we will use  $C$  to represent the component and its transition system. Our proposed interface-generation algorithms operate by analyzing *finite-state* abstractions of  $C$ . The essence of our approach lies in the following observation:

**Theorem 1.** *Let us assume a component  $C$ , a may abstraction  $C^{may}$  for  $C$  and a must abstraction  $C^{must}$  for  $C$ . If an interface  $A$  for  $C$  is permissive with respect to  $C^{must}$  and safe with respect to  $C^{may}$ , then  $A$  is safe and permissive with respect to  $C$ .*

Our approach for interface generation is therefore based on constructing may and must abstractions for a concrete component  $C$  ( $C^{may}$  and  $C^{must}$ , respectively). We first briefly describe a basic algorithm, followed by an optimized one; both algorithms use a combination of automated learning and abstraction refinement techniques. These algorithms involve procedures for checking whether an interface is safe and permissive, which we provide first.

**CheckSafe.** Checking that an interface  $A$  is safe for some component abstraction  $C^{Abst}$  (corresponding to  $C^{may}$  or  $C^{must}$ ), reduces to checking reachability of a state  $(s_a, s_c)$  in  $A \parallel C^{Abst}$  such that  $s_c$  is an error state in  $C^{Abst}$ . A counterexample is returned if such a state is found.

**CheckPermissive.** The key to our approach is that our algorithms only check permissiveness for  $C^{must}$ . Must abstractions are always deterministic since we assume that our concrete components are also deterministic. As a result, checking permissiveness reduces to a simple reachability check. The interface  $A$  is first completed with an error state  $err$  to get  $A_{err}$ .  $C_{sink}^{must}$  is then built by similarly completing  $C^{must}$  with a new state  $sink$ , which is an accepting state (see [12] for explanations of the need for such completions). The permissiveness check then reduces to checking, in automaton  $A_{err} \parallel C_{sink}^{must}$ , reachability of some state  $(err, s_c)$ , where  $s_c$  is a non-error state in  $C_{sink}^{must}$ . If such a state is detected,  $A$  is not permissive, and a counterexample is returned. The counterexample illustrates a correct sequence of invocations to the component that is rejected by the interface.

## 4.1 Algorithms

**Algorithm BuildInterface:** The high-level steps of our basic approach to generating interfaces using may and must abstractions is illustrated in Algorithm 1. Given that  $C^{must}$  is finite-state, the  $L^*$  algorithm is used to generate a safe and permissive interface for  $C^{must}$ , expressed as a DFA  $A^{must}$  over the alphabet of the component. The procedure **LearnInterface** used for this purpose is similar to the one presented in [12]. The interface  $A^{must}$  produced by **LearnInterface** is subsequently checked for safety with respect to  $C^{may}$ . If safe, then based on theorem 1,  $A^{must}$  is a safe and permissive interface for  $C$ . Otherwise, the counterexample  $t$  obtained from the safety check is used to guide the automatic refinement of the predicate set used for building the component abstractions, as described later in this section. Another iteration of the algorithm is then performed, with the new set of predicates.

**Algorithm LearnReuse:** Despite its conceptual clarity, the basic algorithm needs to restart the learning process every time an abstraction is refined. We would ideally like to reuse information learned by  $L^*$  across abstraction refinement iterations. In contrast to the basic algorithm that uses learning on  $C^{must}$ , the

---

**Algorithm 1.** BuildInterface( $C$ )

---

```

1:  $A^{must} := \text{LearnInterface}(C^{must})$ 
2:  $t := \text{CheckSafe}(A^{must}, C^{may})$ 
3: if  $t == \text{null}$  then
4:   return  $A^{must}$ 
5: else
6:    $\text{Preds} := \text{Preds} \cup \text{Refine}(t)$ 
7:   Go to step 1.
8: end if

```

---



---

**Algorithm 2.** Query( $\sigma, C$ )

---

```

1: if  $\text{CheckSafe}(ts(\sigma), C^{must})! = \text{null}$ 
   then
2:   return no
3: else
4:    $t := \text{CheckSafe}(ts(\sigma), C^{may})$ 
5:   if  $t == \text{null}$  then
6:     return yes
7:   else
8:      $\text{Preds} := \text{Preds} \cup \text{Refine}(t)$ 
9:     invoke Query( $\sigma, C$ ) (new Preds)
10:  end if
11: end if

```

---



---

**Algorithm 3.** Oracle 1

---

```

1:  $t := \text{CheckSafe}(A, C^{may})$ ;
2: if  $t == \text{null}$  then
3:   invoke Oracle 2
4: else
5:    $\sigma := \text{project}(t)$ 
6:    $\text{result} := \text{Query}(\sigma, C)$ 
7:   if  $\text{result} == \text{no}$  then
8:     return  $\sigma$  to  $L^*$ 
9:   else
10:    invoke Oracle 1 (new Preds)
11:  end if
12: end if

```

---



---

**Algorithm 4.** Oracle 2

---

```

1:  $t := \text{CheckPermissive}(A, C^{must})$ 
2: if  $t == \text{null}$  then
3:   return  $A$  as safe & permissive
4: else
5:    $\sigma := \text{project}(t)$ 
6:    $\text{result} := \text{Query}(\sigma, C)$ 
7:   if  $\text{result} == \text{yes}$  then
8:     return  $\sigma$  to  $L^*$ 
9:   else
10:    invoke Oracle 2 (new Preds)
11:  end if
12: end if

```

---

optimized algorithm directly learns an interface for component  $C$ , meaning that answers to queries and conjectures represent component  $C$  itself. As a result, the learning process evolves in parallel with the abstraction refinements. Note that the algorithm never actually uses  $C$  itself, but rather its finite-state abstractions  $C^{must}$  and  $C^{may}$ . We use  $\text{Preds}$  to denote a global set of abstraction predicates.

**Queries.** The procedure for queries is illustrated by Algorithm 2. At a high level, a query on  $\sigma$  must return no if  $\sigma \in L^E(C)$  and yes otherwise. We briefly explain here how we are able to determine to use  $C^{must}$  and  $C^{may}$  instead of  $C$ . From Proposition 1,  $L^E(C^{must}) \subseteq L^E(C) \subseteq L^E(C^{may})$ . Therefore, if a counterexample is obtained at line 1, it means that  $\sigma \in L^E(C^{must})$ , which implies that  $\sigma \in L^E(C)$ , so the query returns no. If no counterexample is obtained at line 4, then it means that  $\sigma \notin L^E(C^{may})$ , which implies that  $\sigma \notin L^E(C)$ , so the query returns yes. Otherwise, we know that the counterexample  $t$  obtained belongs to  $C^{may}$  but not to  $C^{must}$  (if it did, then the check on line 1 would not have returned null).  $t$  is then used to refine the abstraction.

**Conjectures.** We use Theorem 1 to answer the conjectures using two oracles, as illustrated in Algorithms 3 and 4.



**Oracle 1:** Is the conjectured assumption  $A$  safe with respect to  $C^{may}$ ?

**Oracle 2:** Is  $A$  permissive with respect to  $C^{must}$ ?

Oracle 1 is invoked first. If it finds that  $A$  is safe with respect to  $C^{may}$ , Oracle 2 gets invoked. If Oracle 2 finds that  $A$  is also permissive with respect to  $C^{must}$ , we conclude from Theorem 1 that  $A$  is a safe and permissive interface for  $C$ . All remaining cases require either the refinement of  $A$  by  $L^*$ , or the refinement of the component abstractions. We use queries to help us determine what needs to be refined. Our approach is described in detail below.

Oracle 1: If  $A$  is not safe with respect to  $C^{may}$ , we obtain a counterexample  $t$ , which leads to error in  $C^{may}$ . We subsequently query  $\sigma = project(t)$  on the component (lines 5 and 6, Algorithm 3); here  $project(t)$  denotes the sequence of method calls corresponding to the sequence of transitions in  $t$ , so that  $\sigma$  is over the interface alphabet that  $L^*$  is learning. From line 1, we know that  $\sigma \in L(A)$ . The querying procedure may involve refinement of the abstraction; let  $C^{may'}$  denote the may abstraction used in the last iteration of the query, when it returns. If the query returns no, then it means that  $\sigma$  should not be in the language of  $A$ , so  $\sigma$  is returned to  $L^*$  for  $A$  to be refined. Otherwise, we invoke Oracle 1 again, knowing that **Preds** have been updated. The reason is that, since the result of the query is yes,  $\sigma$  is safe in  $C^{may'}$ , meaning  $\sigma \notin L^E(C^{may'})$  (lines 4 and 5, Algorithm 2), but is unsafe in  $C^{may}$ , meaning  $\sigma \in L^E(C^{may})$  (line 1, Algorithm 3).

Oracle 2: If  $A$  is not permissive with respect to  $C^{must}$ , we obtain a counterexample  $t$ , which leads to some state  $(err, s_c)$  in  $A_{err} \parallel C_{sink}^{must}$ , where  $s_c$  is a non-error state in  $C_{sink}^{must}$ . Therefore  $t$  does not lead to error in  $C_{sink}^{must}$ . Moreover,  $\sigma = project(t)$  is not in  $L(A)$ . We subsequently query  $\sigma$  on the component (line 6, Algorithm 4). The querying procedure may involve refinement of the abstraction; let  $C^{must'}$  denote the must abstraction used in the last iteration of the query, when it returns. If the query returns yes, then it means that  $\sigma$  should be in the language of  $A$ , so it is returned to  $L^*$ . If the query returns no, then we invoke Oracle 2 again, knowing that **Preds** have been updated. The explanation is as follows. When the query returns no, it means that: 1)  $\sigma$  is unsafe in  $C^{must'}$  (line 1, Algorithm 2); and 2)  $\sigma \in L^E(C)$ . On the other hand,  $\sigma$  must be safe in  $C^{must}$ ; if  $\sigma$  were unsafe in  $C^{must}$ , then the permissiveness check of line 1 could not have returned  $t$  as a counterexample, since  $\sigma = project(t)$ . Therefore clearly,  $C^{must'}$  is more refined than  $C^{must}$ . Note that since  $\sigma \in L^E(C)$  but  $\sigma \notin L^E(C^{must})$ ,  $t$  cannot be a trace of  $C^{must}$ , but is rather a sink trace in  $C_{sink}^{must}$ .

## 4.2 Abstraction Refinement

In the algorithms **BuildInterface** and **LearnReuse** presented above, the abstraction refinement procedure is applied whenever a violating trace  $t$  is discovered that belongs to  $C^{may}$  but not to  $C^{must}$ . Consequently,  $t$  must contain a *may* transition  $(a_i \xrightarrow{\tau}_{may} a_{i+1})$  that is not a *must* transition. This means that there exists at least another abstract state  $a'_{i+1}$  that is a successor of  $a_i$  by  $\tau$  via a *may* transition,

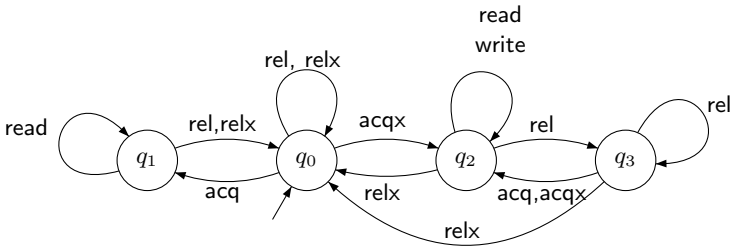
i.e.  $a_i \xrightarrow{\tau}_{may} a'_{i+1}$ . The reason is that  $a_i$  does not distinguish between concrete states of two types: those whose successors are abstracted to  $a_{i+1}$  and those whose successors are abstracted to  $a'_{i+1}$ .

Automated abstraction refinement consists in adding new abstraction predicates (based on weakest pre-conditions). As a result, we split  $a_i$  into two or more new abstract states, corresponding to predicates in  $a_i \wedge \text{wp}(a_{i+1}, \tau)$  and  $a_i \wedge \neg \text{wp}(a_{i+1}, \tau)$  respectively, that separate the concrete states of type (i) and (ii) above. Note that this results in a finer partition of the concrete states. The new abstraction will no longer contain  $\tau$  as a *may* and non-*must* transition and therefore we have the following proposition:

**Proposition 2.** *If trace  $t$  has a transition  $\tau$  that is of type may but not must, the refined abstraction results in a strictly finer partition and does not contain transition  $\tau$ .*

In practice, given a sequence of transitions as a counterexample  $\text{Cex} = \{\tau_1, \tau_2, \dots, \tau_n\}$ , we compute refinement predicates using  $\text{wp}$  computations recursively  $\text{wp}(\text{true}, \text{Cex}) = \text{wp}(\text{wp}(\text{true}, \tau_n), \{\tau_1, \tau_2, \dots, \tau_{n-1}\})$ .

Our refinement algorithm uses weakest precondition calculations to compute new abstraction predicates that are guaranteed to eliminate these may transitions, and returns the newly discovered predicates. We note that unlike standard approaches to counterexample-based abstraction refinement [6], we do not refine solely based on “spurious” counterexamples. The counterexamples obtained from failed safety checks may be feasible, but they may still lead to refinement since they contain non-must transitions.



**Fig. 2.** Read-Write-Acq Example Interface

*Example 3.* For the example of Figure 1, our algorithms generate the *safe* and *permissive* interface  $A$  shown in Figure 2. The interface captures the enforcements imposed by the library. Method *read* can only be called after calling *acq* ( $q_1$ ) or *acqx* ( $q_2$ ). However, method *write* can only be called after calling *acqx* ( $q_2$ ). Consecutive calls of *acq* or *acqx* are inhibited and *acq* once called can be called again only after calling *rel* or *relx*.

The generated interface has one state more than the interface presented in [16] for the same example. On closer inspection, we see that the automaton accepts the string  $\sigma = \text{acqx,write,rel,acq,write}$  which is not accepted by their interface. After calling the method *acqx* from the start state  $s_0 = \{x = 0, a = \text{NULL}, e = 0\}$ ,

the variable  $a$  becomes non-null and  $x$  is set to 1. The method `write` does not modify  $a$  or  $x$ . The next method call `rel` only sets  $a$  to `NULL` but leaves  $x$  unchanged (which remains 1). Now after the `acq` method  $a$  is again set to non-null. Since  $a \neq \text{NULL}$  and  $x = 1$ , the `write` method can now be called safely. When we contacted the authors of [16], they observed a discrepancy between the example as it appeared in their paper and their implemented case study, which explains the difference in our respective results.

### 4.3 Correctness and Termination

In this section we argue the correctness and termination of our algorithms. We will be using  $Alg$  to represent either `BuildInterface` or `LearnReuse`, when our presented arguments hold for both algorithms.

**Theorem 2 (Correctness).** *If algorithm  $Alg$  terminates (with final abstractions  $C^{must}$  and  $C^{may}$ ), then the constructed interface  $A$  is safe and permissive for  $C$ . Furthermore,  $L^E(C^{must}) = L^E(C^{may}) = L^E(C)$ .*

**Termination.** For infinite-state components, the predicate abstraction refinement used in  $Alg$  may not always terminate. However, we can make the following termination argument:

**Theorem 3.** *If  $Alg$  computes an abstraction such that  $L^E(C^{must}) = L^E(C^{may}) = L^E(C)$ , then the  $Alg$  terminates.*

Furthermore, from previous work on automatic abstraction refinement [22,19], we know that if the component  $C$  has a finite bisimulation quotient [20], then the refinement based on weakest precondition calculations is guaranteed to converge to that finite quotient.

**Theorem 4 (Bisimulation Completeness [22,19]).** *If the component  $C$  has a finite bisimulation quotient, then there exists a refinement iteration bound such that the abstraction  $C^{may}$  is bisimilar to  $C$ .*

Since bisimulation implies trace equivalence [21] and from Proposition 1, it follows that if  $C$  has a finite bisimulation quotient, then there exists a refinement iteration bound such that for the obtained set of abstraction predicates,  $L^E(C^{must}) = L^E(C^{may}) = L^E(C)$ . Therefore, together with Theorem 4 we conclude the following:

**Theorem 5 (Termination).** *If the component  $C$  has a finite bisimulation quotient then  $Alg$  terminates.*

We observe that this termination condition is not very tight as our algorithms also terminate for systems for which predicate abstraction with refinement results in an abstraction such as  $L^E(C^{must}) = L^E(C^{may}) = L^E(C)$ , which is a weaker condition than the existence of a finite bisimulation quotient (see Theorem 3).

Let us finally note that although in general our algorithms may not terminate, they can be made to return results “any time”. At any stage, we may use  $L^*$  to compute interfaces for  $A^{may}$  for  $C^{may}$  and  $A^{must}$  for  $C^{must}$ . The language of

the safe and most permissive interface  $A$  for component  $C$  is bounded between the languages of  $A^{may}$  and  $A^{must}$ .

## 5 Implementation and Experiments

**Implementation.** We have implemented the algorithms presented in Section 4 in the ARMC tool [24]. ARMC already had support for may abstractions; we extended it with support for must abstractions. Furthermore, ARMC provides abstraction refinement algorithms based on Craig interpolation [17]. We have integrated these algorithms in our approach, as an alternative to refinement based on weakest preconditions.

We note that the algorithms presented previously use the explicit composition of the abstraction with the interface. Instead of performing this explicit composition, our implementation builds the abstract graph of the composite automaton implicitly, by method inlining. This helps us avoid un-necessary computation and only constructs a part of component abstractions which are sufficient to prove (or disprove) the *safety* and *permissive* checks.

We observe that in the basic algorithm, only *feasible* counterexample traces can add error behaviors to the must abstraction  $C^{must}$ . The spurious counterexamples only remove error behaviors from the may abstraction  $C^{may}$ . Therefore it suffices to restart learning only after refining feasible counterexamples. In the case of spurious counterexamples, the CheckSafe algorithm is restarted after the may abstraction is refined; it terminates when either a feasible counterexample is found or the interface is discovered to be *safe*.

**Experiments.** We evaluate our interface generation algorithm on several sample Java2SDK library classes presented in [1,17] as well as some benchmarks from J2SEE and the NASA CEV 1.5 EOR-LOR mission profile case study [12]. A brief description about the modelling and generated interfaces follows. The experiments were run on a dual core 1.80 GHz Intel Pentium processor with 3 GB of RAM. Table 1 presents the empirical results obtained from following different algorithmic schemes: **wp**: BuildInterface with weakest precondition refinement; **wp+craig**: BuildInterface with craig interpolation refinement for infeasible counterexamples (wp+craig); **refine-may**: BuildInterface with refining only may abstraction for infeasible counterexamples (refine-may + craig); **learn-reuse**: LearnReuse with craig. The table also presents the number of predicates (#Preds) discovered, the number of learning iterations (#Iterations), the number of states in the final interface (#States) and the running times.

The primary purpose of our experiments is to assess the feasibility of our approach. We additionally provide, with a smaller emphasis, a comparison between algorithms BuildInterface and LearnReuse. Our results show that the proposed approach is feasible, and also quantify the expected improvement achieved by LearnReuse. We can additionally use our experimental results as approximate indications of the practical savings achieved by LearnReuse over previous approaches that perform learning and abstraction separately [1]. These approaches

are based on manual refinement, but if automated, their performance would be similar to `BuildInterface` since they do not perform abstraction on demand during the learning process.

For the `Signature` class we selected five methods as the alphabet  $\Sigma = \{\text{initSign}, \text{initVerify}, \text{sign}, \text{update}, \text{verify}\}$ . The exception `SignatureException` was modelled as the error predicate. The states in the generated interface correspond to the labellings of uninitialized, sign and verify respectively. The `ServerTableEntry` class is taken from the package `com.sun.corba.se.internal.Activation`. We selected six methods as the alphabet  $\Sigma = \{\text{activate}, \text{register}, \text{registerPorts}, \text{install}, \text{uninstall}, \text{holdDown}\}$  and modelled the exception `INTERNAL` as the error condition. The generated interface only keeps track of three states: `activated(register)`, `running(install/uninstall)` and other states as one state. The `ListItr` class is an inner class of `AbstractList` from the package `java.util`. We selected five methods as the alphabet  $\Sigma = \{\text{next}, \text{remove}, \text{previous}, \text{set}, \text{add}\}$  and the exception `IllegalStateException` was modelled as the error predicate. The interface captures the inhibition of calls of methods `set` and `remove` after calling methods `remove` or `add`.

The `PipedOutputStream` class is taken from the package `java.io` and is an implementation of an abstract class `OutputStream`. We selected five methods as the alphabet  $\Sigma = \{\text{close}, (\text{connect},0), (\text{connect},1), \text{flush}, \text{write}\}$ , where we model invocations of `connect` method returning different values (0 or 1) as different methods (`(connect,0)` or `(connect,1)`) similar to the approach taken in [5]. The exception `NullPointerException` was modelled as the error predicate. The interface captures precisely two states where `sink = null` and `sink  $\neq$  null`. Only a successful `connect` call can enable `flush` and `write` methods. The `Socket` class is part of `java.net` package which implements client sockets. We considered seven methods as the alphabet  $\Sigma = \{\text{close}, \text{bind}, \text{getInputStream}, \text{getOutputStream}, \text{shutdownInput}, \text{shutdownOutput}\}$  and the exception `SocketException` was modelled as the error predicate. The interface enforces the requirement that `bind` cannot be called after `connect`, `shutdownInput` can only be called after calling `connect`, `getInputStream` can only be called after the `connect` call and not after `close` or `shutdownInput` has been called. After calling `close` no other method calls are allowed. The class `TransactionManager` is taken from the package `javax.transaction`, and we selected six methods as the alphabet  $\Sigma = \{\text{begin}, \text{suspend}, \text{resume}, \text{commit}, \text{rollback}, \text{setrollbackonly}\}$ . The exception `IllegalStateException` was modelled as the error predicate. The generated interface captures the precise sequence of method calls for performing a transaction with appropriate handling of `commit` and `rollback` actions.

We also applied our technique to obtain the interface for the simplified state machine for NASA CEV 1.5 EOR-LOR mission profile case study. It models the `Ascent`, `EarthOrbit`, `TransitEarthMoon` and `Entry` phases of the space-craft, the events (like `srbIgnition`, `stage1separation` etc.), the vehicle configuration and various failure modes. The Java model is available with the JPF distribution under `examples/jpfESAS`. We modelled the 22 events as the alphabet set for the interface and the error predicate was modelled as the failure modes and the event calls from inappropriate states. Events with parameters like `abort(boolean`

**Table 1.** Experiment results on benchmark case studies

Class name	Algorithm	#Preds	#Iterations	#States	Running Time
ListItr	wp	12	5	2	40.6s
	wp+craig	<b>7</b>	6	2	42.2s
	refine-may	8	<b>4</b>	2	39.3s
	learn-reuse	6	1	2	<b>12.7</b>
Signature	wp	8	9	3	72.7s
	wp+craig	<b>5</b>	6	3	42.9s
	refine-may	7	<b>4</b>	3	33.2s
	learn-reuse	5	1	3	<b>16.6s</b>
ServerTableEntry	wp	10	10	3	98.1s
	wp+craig	<b>6</b>	7	3	64.9s
	refine-may	10	<b>5</b>	3	51.3s
	learn-reuse	7	1	3	<b>19.2</b>
PipedOutputStream	wp	4	5	2	16.4s
	wp+craig	2	3	2	11.4s
	refine-may	2	3	2	11.6s
	learn-reuse	2	1	2	<b>7.4s</b>
read-write-acq	wp	6	6	4	122.8s
	wp+craig	<b>4</b>	5	4	75.4s
	refine-may	7	5	4	74.3s
	learn-reuse	6	1	4	<b>31.1</b>
Socket	wp	25	5	6	468.5s
	wp+craig	13	5	6	272.9s
	refine-may	13	5	6	228.0s
	learn-reuse	<b>12</b>	1	6	<b>65.8</b>
TransactionManager	wp	15	8	4	138.6s
	wp+craig	9	7	4	103.5s
	refine-may	9	4	4	76.9s
	learn-reuse	9	1	4	<b>30.4s</b>
NASA-Ascent	wp	34	14	5	1685.6s
	wp+craig	20	14	5	1433.9s
	refine-may	20	6	5	712.4s
	learn-reuse	20	1	5	<b>75.6s</b>
NASA-Complete	learn-reuse	74	1	14	3115.6s

controlMotorField) were modelled as two events one with controlMotorField parameter true (abortctr) and the other with controlMotorField parameter false (abortnctr) which increased the alphabet size to 26. The interface has 14 states and required 74 predicates. For such large interfaces, only the LearnReuse algorithm finished in reasonable time. Table 1 also documents the results for NASA-Ascent interface where only the Ascent phase of the space-craft was modelled. These interfaces in addition to being helpful in verifying the space-craft code are also a useful tool to help debug the system early in the designing process of such critical software.

## References

1. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL, pp. 98–109 (2005)
2. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: Mocha: Modularity in model checking. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
3. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL, pp. 4–16 (2002)
4. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* 75(2), 87–106 (1987)
5. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for interface synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 4–19. Springer, Heidelberg (2007)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM, New York (1977)
8. Das, M., Lerner, S., Seigle, M.: Esp: Path-sensitive program verification in polynomial time. In: PLDI, pp. 57–68 (2002)
9. de Alfaro, L., Godefroid, P., Jagadeesan, R.: Three-valued abstractions of games: Uncertainty, but with precision. In: LICS (2004)
10. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 262–277. Springer, Heidelberg (2002)
11. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: PLDI, pp. 1–12 (2002)
12. Giannakopoulou, D., Pasareanu, C.S.: Interface generation and compositional verification in javapathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
13. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, p. 426. Springer, Heidelberg (2001)
14. Godefroid, P., Huth, M., Jagadeesan, R.: A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 275–287. Springer, Heidelberg (2003)
15. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: Unleashing the power of alternation. In: POPL (2010)
16. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: ESEC/SIGSOFT FSE, pp. 31–40 (2005)
17. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244 (2004)
18. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
19. Lee, D., Yannakakis, M.: Online minimization of transition systems. In: ACM Symposium on Theory of Computing (1992)
20. Lee, D., Yannakakis, M.: Online minimization of transition systems (extended abstract). In: STOC, pp. 264–274. ACM, New York (1992)

21. Milner, R.: *Communication and Concurrency*. Prentice Hall, New York (1989)
22. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: Emerson, E.A., Sistla, A.P. (eds.) *CAV 2000*. LNCS, vol. 1855, pp. 435–449. Springer, Heidelberg (2000)
23. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: *Logics and models of concurrent systems*, pp. 123–144 (1985)
24. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) *PADL 2007*. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2006)
25. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. *Inf. Comput.* 103(2), 299–347 (1993)
26. Shoham, S., Grumberg, O.: Compositional verification and 3-valued abstractions join forces. In: Riis Nielson, H., Filé, G. (eds.) *SAS 2007*. LNCS, vol. 4634, pp. 69–86. Springer, Heidelberg (2007)
27. Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. In: *ESEC / SIGSOFT FSE*, pp. 188–197 (2003)
28. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. In: *ISSTA*, pp. 218–228 (2002)