Interface of Software Components with Internal Behaviors

Ruzhen Dong[1,2], Naijun Zhan[3], Liang Zhao[4]

[1] UNU-IIST, Macau
[2] Dipartmento di Informatica, Università di Pisa,Italy
ruzhen@iist.unu.edu
[3] Institute of Software, Chinese Academy of Sciences
znj@ios.ac.cn
[4] Institute of Computing Theory and Technology, Xidian University
lzhao@xidian.edu.cn

Abstract. We present an automata-based execution model describing the interaction and internal behaviors of software components extending our previous work with internal behaviors. In order to make the services provided by the component unblockable without knowing the implementation details, we propose an interface model that specifies all the unblockable interaction behaviors with any environment. To this end, we develop an algorithm to generate an interface model from any given execution model. Composition operators and refinement relation are also introduced based on this model, and we justify the refinement relation w.r.t substitution.

1 Introduction

In component-based software engineering, large software systems are decomposed into components with clearly stated interfaces in order to facilitate a sound development process across different teams of developers or existing software components. An interface theory should then define the basic principles for composing several software components based on their interfaces while the concrete implementation of the components is invisible to its environment. On the other hand, an interface theory should also define a refinement relation that if a component C' refines component C then C' can substitute C in any context, that means, those behaviors that are allowed in original software system are not blocked after the substitution. This means that components can be treated as black-boxes, and the theory allows for independent implementation, substitution, and deployment.

In our previous work[4], we studied the interface model of components that comprise *provided* and *required* interface. The provided interface describes which services the component offers to its environment, while the required interface specifies what services the component needs to call in order to provide the services on its provided interface. In this paper, we study the interface model of components that extend the components in [4] with *internal actions* which are

invisible to the environment. Components are adapted with some provided services internalized to restrict these services from being called by the environment for safety or privacy reasons. The interface model should still guarantee to provide services that are unblockable during implementation. That is, the interface model should support a black-box compositions in the sense that the components can be composed without an a priori knowledge of the execution model of their implementation as long as the specified sequences of provided services and the corresponding required services are respected by the implementation. We show that the interface model is *input-deterministic*, meaning that after calling any fixed sequence of provided services with any number of internal behaviors interleaved possibly, the available provided services are deterministic.

In order to generate an interface model for any given execution model with internal behaviors, we first study what kinds of provided services are *enabled* at a given state, that is, the provided services are not blocked when the environment calls at the state. Then we study whether sequence of provided services with possible internal behaviors interleaved is blocked or not. A provided service m is enabled at state s if m is available at all the internally reachable stable states at which there are no internal behaviors available. Based on these, we update the algorithm in the paper[4] for any given execution models with internal behaviors.

Related work. There are two main well known approaches in interface theories, the Input/Output(I/O) Automata [13,12] and the Interface Automata [1,3,2]. And there are also some work extending these with modalities [10,14,15,11].

As argued in the paper [4], our approach is in positioned in between these existing approaches. I/O automata are defined by Lynch and Tuttle to model concurrent and distributed discrete event systems. The main features of the I/O automata are *input-enabledness* and *pessimistic* compatibility. The input-enabledness requires that all the input actions should be available at any state. The pessimism is that two components are compatible if there is no deadlock in the composition for all environment. On the contrary, our interface model does not require that all inputs are always enabled, because there are guards for provided services in software components, while interface model is input-deterministic to guarantee that all sequences of provided services with possible internal behaviors can never be blocked when the environment calls.

Alfaro and Henzinger introduce interface automata to model reactive components that has an assume-guarantee relation with the environment. Two components are compatible in an optimistic way in the sense that two components are compatible if there exists one environment that can make the composition avoid deadlock. This compatibility condition may be too relaxed since the usability of the provided service of a component depends not only the components to be composed with and also the environment for the composition. To this end, Giannalopoulou et al [5,6] develop assume-guarantee rules and tools to reason about compatibility and weakest assumptions for any given interface automaton, while Larson et al [9] present interface models for any given I/O automaton by splitting assumptions from guarantees. In contrary to these approaches, we present an interface model that directly specifies the unblockable sequences of provided

services independent of the environment and develop an algorithm to generate such interface model based on the execution model of any given component.

Summary of contributions. The contributions of this paper are (1) a new interface model ensuring unblockable provided services of components with internal behaviors under any environment. (2) an updated algorithm to generate the interface model of a component based on its execution model with internal behaviors. (3) composition operators and an algorithm to eliminate failure state which is introduced for failed service invocations. (4) refinement relation for substitution of components.

Outline of the paper. The rest of the paper is organized as follows. In Sect. 2, we introduce component automata and related concepts. In Sect. 3, we show unblockable sequences of provided events and the algorithm to generate interface model from execution model. In Sect. 4, we present the composition operators. In Sect. 5, we define a refinement relation based on state simulation and prove compositional results. In Sect. 6, we conclude the paper and discuss the future work.

2 Component Automata

2.1 Preliminary Definitions

For any $w_1, w_2 \in \mathcal{L}^*$, the sequence concatenation of w_1 and w_2 is denoted as $w_1 \circ w_2$ and extended to sets of sequences, that is, $A \circ B$ is $\{w_A \circ w_B \mid w_A \in A,\ w_B \in B\}$ where $A, B \subseteq \mathcal{L}^*$ are two sets of sequences of elements from $\mathcal{L}$. Given a sequence of sets of sequences $\langle A_1, \ldots, A_k \rangle$ with $k \geq 0$, we denote $A_1 \circ \cdots A_k$ as $conc(\langle A_1, \ldots, A_k \rangle)$. We use ϵ as notion of empty sequence,that is, $\epsilon \circ w = w \circ \epsilon = w$. Given a sequence w, we use $last(w)$ to denote the last element of w.

Let ℓ be a pair (x, y), we denote $\pi_1(\ell) = x$ and $\pi_2(\ell) = y$. Given any sequence of pairs $tr = \langle \ell_1, \ldots, \ell_k \rangle$ and set of sequences of pairs T, it is naturally extended that $\pi_i(tr) = \langle \pi_i(\ell_1), \ldots, \pi_i(\ell_k) \rangle$, $\pi_i(T) = \{\pi_i(tr) \mid tr \in T\}$ where $i \in \{1, 2\}$.

Let $tr \in A$ and $\Sigma \subseteq \mathcal{L}$, $tr\lfloor_\Sigma$ is a sequence obtained by removing all the elements that are not in Σ from tr. And we extend this to a set of sequences $T\lfloor_\Sigma = \{tr\lfloor_\Sigma \mid tr \in T\}$.

Given a sequence of pairs tr, $tr\lfloor_P^1$ is a sequence obtained by removing the elements whose first entry is not in P. For a sequence of elements $\alpha = \langle a_1, \cdots, a_k \rangle$, $pair(\alpha) = \langle (a_1, \{a_1\}), \cdots, (a_k, \{a_k\}) \rangle$.

2.2 Execution Model of a Component

Definition 1. *A tuple $C = (S, s_0, P, R, A, \delta)$ is called a* component automaton *where*

- $\diamond$ *S is a finite set of states and $s_0 \in S$ is the initial state;*

- P, R, and A *are disjoint and finite sets of provided, required, and internal events respectively;*
- $\delta \subseteq S \times \Sigma(P,R,A) \times S$ *is the transition relation, where the set of labels is defined as* $\Sigma(P,R,A) = (P \cup A) \times (2^{R^*} \setminus \emptyset)$.

Whenever there is $(s, \ell, s') \in \delta$ with $\ell = (w, T)$, we simply write $s \xrightarrow{\ell} s'$ and call it a provided transition step if $w \in P$, otherwise internal transition step. The internal events are prefixed with ; to differentiate from provided events. We use τ to represent any internal event when it causes no confusion. For a state s we use $out(s)$ denote $\{w \in P \cup A \mid s \xrightarrow{w/T} s'\}$ and $out_P(s) = out(s) \cap P$ and $out_A(s) = out(s) \cap A$. We write $s \xrightarrow{w/\bullet} s'$ for $s \xrightarrow{w/T} s'$, when we don't care what T is.

A component automaton is called *closed* if $T = \{\epsilon\}$ for all transitions $s \xrightarrow{a/T} s'$, that means, the component provide services without requiring services from other components. Otherwise, it is called *open*.

An alternating sequence of states and labels of the form

$$e = \langle s_1, \ell_1, \ldots, s_k, \ell_k, s_{k+1} \rangle$$

is denoted as $s_1 \xRightarrow{\ell_1, \cdots, \ell_k} s_{k+1}$ with $k \geq 0$, if $s_i \xrightarrow{\ell_i} s_{i+1}$ for all i with $0 \leq i \leq k$.

It is called an *execution* of the component automaton C, if s_1 is the initial state s_0. The set $\{\langle \ell_1, \ldots, \ell_k \rangle \mid s \xRightarrow{\ell_1, \ldots, \ell_k} s'\}$ is an execution is denoted as $\mathcal{T}(s)$. An element in $\mathcal{T}(s_0)$ is called a *trace* of component automaton C and $\mathcal{T}(s_0)$ is also written as $\mathcal{T}(C)$.

$\{\pi_1(tr)\lfloor P \mid tr \in \mathcal{T}(s)\}$ is denoted as $\mathcal{T}_p(s)$ and $pt \in \mathcal{T}_p(s_0)$ is called a *provided trace* of component automaton C, so $\mathcal{T}_p(s_0)$ is also written as $\mathcal{T}_p(C)$.

Given a provided trace $pt \in \mathcal{T}_p(C)$,the set $required(pt)$ of required traces for pt is defined as $required(pt) = \{conc(\pi_2(tr)) \mid \pi_1(tr)\lfloor_P = pt,$ and the last element of $\pi_1(tr)$ is a provided event$\}$.

Next, we will use the example shown in [4] with *wifi* hidden as an internal behavior, since after the first connection, the component will automatically connect *wifi* after *login*.

Example 1. As a demonstrating example, we consider a simple internet-connection component presented in Fig. 1. It provides the services *login*, *print*, and*read* available to the environment and there is an internal service ; *wifi*. The services model the logging into the system,invocation of printing a document, an email service, and automatically connecting the wifi, respectively. The component calls the services *unu1*, *unu2*, *cserv*, *cprint*, and *senddoc*. The first three of them model the searching for a wifi router nearby, connecting to the *unu1* or *unu2* wireless network, and connecting to an application server, respectively. The *cprint* and *senddoc* are services that connect to the printer, sends a document for a print and starts the printing job. The *print* service is only available for the wifi network *unu1* and *read* can be accessed at both networks.

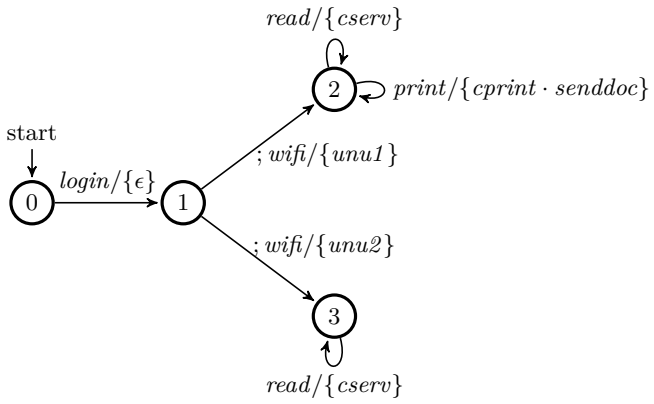

Fig. 1. Execution model of internet connection component C_{ic}

In the component model of Fig. 1 we can perform e.g.

$$e = \langle 0, (login/\{\epsilon\}), 1, (; wifi/\{unu1\}), 2, (print/\{cprint \cdot senddoc\}), 2\rangle \ .$$

Now $pt = \langle login, print\rangle$ is a provided trace of the execution e and the set of required traces of pt is $required(pt) = \{unu1 \cdot cprint \cdot senddoc\}$. This example will be used throughout this paper to show the features of our model.

3 Interface Model

The behavior model of a component describes how the component interacts with the environment by providing and requiring services. However, some provided transition or execution may fail due to non-determinism. In this section, we will discuss about unblockableness of provided events and traces.

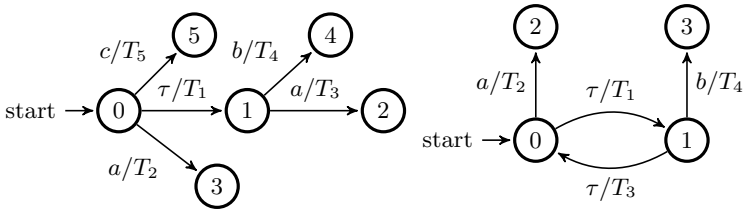

Fig. 2.

We say provided event a is *enabled* at state s, if the environment can require a successfully without being blocked when the component automaton is at state s. Next, we use some examples to show what kinds of provided events are enabled.

Let's consider state 0 of the component automaton shown in the left of Fig. 3. From the view of environment, the component automaton may be at 0 or 1, because there is an internal transition from 0 to 1. We assume that after certain time, the component will move to state 1 eventually, because 0 is not a stable state. So we can see that even $c \in out_p(0)$, c is not *enabled* at state 0, because $c \notin out_p(1)$. However, if the environment requires b, the component can react to this invocation successfully, that is, b is enabled at state 0. The enabled provided events are determined by internally reachable stable states. In the component automaton shown in the right part of Fig. 3, there is no internally reachable states from state 0. The component may behave internally forever without reacting to invocations of the environment, like a and b. Next, we will formally define enabled provided events and related concepts. The intuitive idea of enabledness is borrowed from the failure-divergence model of CSP [7].

We call a state s *divergent* if there exists a sequence of internal transitions to s from s or s can transit to such kinds of states internally.

Definition 2. *A state s is* divergent*, if there exists tr with $\pi_1(tr) \in A^+$ such that $s \stackrel{tr}{\Longrightarrow} s$ or there exists s' such that $s \stackrel{tr}{\Longrightarrow} s'$ and $s' \stackrel{tr'}{\Longrightarrow} s'$ with $\pi_1(tr), \pi_1(tr') \in A^+$.*

The enabled provided events of a divergent state is empty.

A state s is *stable*, if $out(s) \subseteq P$, that is, there is no internal transition from state s. Internally reachable states of state s is $\{s' \mid s \stackrel{tr}{\Longrightarrow} s' \text{ with } \pi_1(tr) \in A^+\}$, written as $\mathcal{R}(s)$. The set of internally reachable stable states, written as $\mathcal{RS}(s)$ is empty if $\mathcal{R}(s)$ contains divergent states, otherwise it is a set of stable states from $\mathcal{R}(s)$, that is, $\{s' \text{ is stable } \mid s' \in \mathcal{R}(s)\}$.

Definition 3 (enabled provided events). *Given a component automaton $C = (S, s_0, P, R, A, \delta)$, for any $s \in S$, the set of enabled provided events of s is enabled$(s) = \bigcap_{r \in \mathcal{RS}(s)} out_P(r)$.*

The enabled provided events assure unblockableness of a single provided event at a given state. Next, we will introduce unblockable provided trace which can never be blocked by the component when the environment requires it.

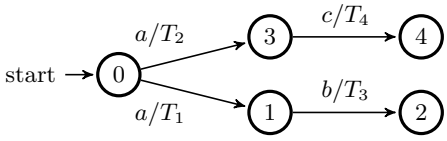

Fig. 3. Example 2

Let's consider the component automaton shown in Fig. 3. We can see that a is enabled at state 0, but, after the invocation of a, the component determines whether move to state 1 or 3. So both of b and c may be blocked after a.

A sequence of provided events $\langle a_1, \cdots, a_k \rangle$ with $k \geq 0$ are *unblockable* at state s, if $a_i \in \mathit{enabled}(s')$ for any $1 \leq i \leq k$ and s' that $s \stackrel{tr}{\Longrightarrow} s'$ with $\pi_1(tr)\lfloor_P = \langle a_1, \cdots, a_{i-1} \rangle$. A sequence of pairs tr is *unblockable* at s, if $\pi_1(tr)\lfloor_P$ is unblockable at s. We use $\mathcal{T}_{up}(s)$ and $\mathcal{T}_u(s)$ to denote the set of all unblockable provided traces and unblockable traces at state s, respectively. $\mathcal{T}_{up}(s)$ and $\mathcal{T}_u(s)$ are also written as $\mathcal{T}_{up}(C)$ and $\mathcal{T}_u(C)$, respectively, when s is the initial state.

3.1 Interface automata

In this part, we will introduce input-deterministic component automaton of which all traces are unblockable and present an algorithm that transforms any given component automaton to an input-deterministic one while preserving all the unblockable traces.

Definition 4 (Input-determinism). *A component automaton* $C = (S, s_0, P, R, A, \delta)$ *is* input-deterministic *if for any*

$$s_0 \stackrel{tr_1}{\Longrightarrow} s_1 \text{ and } s_0 \stackrel{tr_2}{\Longrightarrow} s_2$$

with $\pi_1(tr_1)\lfloor_P = \pi_1(tr_2)\lfloor_P$, *imply*

$$\mathit{enabled}(s_1) = \mathit{enabled}(s_2).$$

Theorem 1. *A component automaton C is input-deterministic iff $\mathcal{T}_p(C) = \mathcal{T}_{up}(C)$.*

Proof. $\mathcal{T}_p(C) = \mathcal{T}_{up}(C)$ means every provided trace of C is unblockable actually.

First, we prove the direction from left to right. From the input-determinism of C follows that for each provided trace $pt = (a_0, \ldots, a_k)$ and for each state s with $s_0 \stackrel{tr}{\Longrightarrow} s$ and $\pi_1(tr) = \langle a_0, \ldots, a_i \rangle$ for $0 \leq i \leq k-1$, the set $\mathit{enabled}(s)$ is the same. Since pt is a provided trace, so $a_{i+1} \in \mathit{enabled}(s)$. This shows that all provided traces are unblockable, and so all traces are unblockable too.

Second, we prove the direction from right to left by contraposition. We assume that C is not input-deterministic, so there exist traces tr_1 and tr_2 with $\pi_1(tr_1)\lfloor_P = \pi_1(tr_2)\lfloor_P$ and $s_0 \stackrel{tr_1}{\Longrightarrow} s_1$, $s_0 \stackrel{tr_2}{\Longrightarrow} s_2$ such that $\mathit{enabled}(s_1) \neq \mathit{enabled}(s_2)$.

Then we can w.l.o.g. assume that there is a provided event a such that $a \in \mathit{enabled}(s_1)$ and $a \notin \mathit{enabled}(s_2)$. Now $\pi_1(tr_1) \circ \langle a \rangle$ is a provided trace of C that is blockable.

□

We say a component automaton is an *interface* if it is input-deterministic, which implies that all provided services provided by the interface are unblockable.

We now present an algorithm that, for a given component automaton C, constructs the largest interface automaton $\mathcal{I}(C)$ that is equivalent with respect

Algorithm 1: Construction of Interface Automaton $\mathcal{I}(C)$

Input: $C = (S, s_0, P, R, A, \delta)$
Output: $\mathcal{I}(C) = (S_I, (\{s_0\}, s_0), P, R, A, \delta_I)$, where $S_I \subseteq 2^S \times S$

Initialization: $S_I := \{(\{s_0\}, s_0)\}$; $\delta_I := \emptyset$; $todo := \{(\{s_0\}, s_0)\}$; $done := \emptyset$
while $todo \neq \emptyset$ **do**
  choose $(Q, r) \in todo$; $todo := todo \setminus \{(Q, r)\}$; $done := done \cup \{(Q, r)\}$
  for each $a \in \bigcap_{s \in Q} enabled(s)$ **do**
    $Q' := \bigcup_{s \in Q} \{s' \mid s \stackrel{tr}{\Longrightarrow} s', \pi_1(tr)\downarrow_P = \langle a \rangle\}$
    for each $(r \xrightarrow{a/T} r') \in \delta$ **do**
      if $(Q', r') \notin (todo \cup done)$ **then**
        $todo := todo \cup \{(Q', r')\}$
        $S_I := S_I \cup \{(Q', r')\}$
      end if
      $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{a/T} (Q', r')\}$
    end for
  end for
  for each $r \xrightarrow{w/T} r'$ **with** $r' \in Q$ **and** $w \in A$ **do**
    $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{w/T} (Q, r')\}$
  end for
end while

to its unblockable traces with the component automaton. The intuitive idea of the algorithm is same as the interface algorithm in the work [4]. The automaton $\mathcal{I}(C)$ can be understood as the interface behavior of the component that can be advertised to other components and guarantees a deadlock-free composition as long as the interaction described by the interface automaton is respected.

A general construction of an interface automaton $\mathcal{I}(C)$ for a given component automaton C is given in Algorithm 1. The states of the interface automaton are pairs (Q, r) consisting of a subset Q of states of the original automaton and a single state r. In the first element of the pair, the power-set construction (similar to the construction of a deterministic automaton from a non-deterministic one) is adopted to identify the set of all states Q that can be reached by all traces of which the projection to the set of provided events are same, which is a bit different from the previous algorithm. In the second element of the pair, we compute the intersection of the languages of provided traces of C and the constructed power-set automaton. By this, the original executions of C are simulated for the non-blockable provided traces.

The termination of the algorithm can be obtained because *todo* will be eventually empty and the set *done* increases iterately and the union of *done* and *todo* is finite. We will show correctness of the algorithm by proving that $\mathcal{I}(C)$ is refined by C in Section. 5 after refinement is introduced.

Next, let's see interface model of C_{ic} in 1 generated by the algorithm.

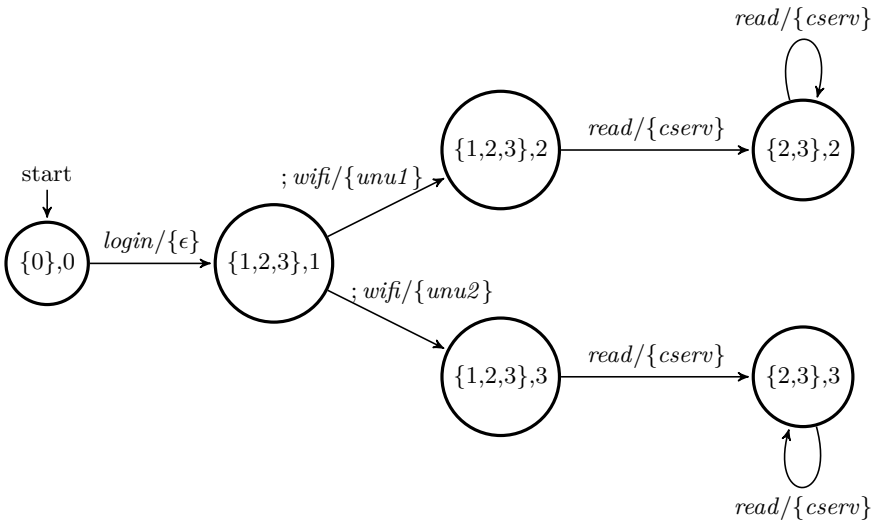

Fig. 4. Interface model of internet connection component C_{ic}

Example 2. In the internet connection component 1, the provided trace $\langle login, read\rangle$ are unblockable but $\langle login, print\rangle$ may be blocked during execution, because after *login* is called, the component may transit to state 3 at which *print* is not available. We use the Algorithm 1 to generate the interface model of C_{ic} shown in Fig. 4

4 Composition operators

In this section, we will present how two components are composed. Components interact with each other by service invocation which is event synchronization in the automata-based execution model. More precisely, component automata synchronize on the events that are provided by one and required by the other. To this end, we show how a set of required traces of one component are provided by the other component by defining the synchronization rules of a finite state machine recognizing the set of required traces and the component to be composed with. We use finite state machine for two main reasons: finite state machine is well-studied model recognizing a set of sequences of elements that describes the step wise transitions of required traces and it is easy to calculate the new required traces after composition since both models are state-based automata models. We also give the definition of composition totally based on the required and provided traces which is proved to be equivalent with the operational way. The operational definition of composition explains well about how two components are composed step by step, while the composition based on traces are useful for calculation.

In a component automaton $C = (S, s_0, P, R, A, \delta)$, for any transition $s_1 \xrightarrow{a/T} s_2$, there is a minimal deterministic finite state machine recognizing T [8]. We

use $\mathcal{M}(T)$ to denote the smallest deterministic finite state machine recognizing T. The finite state machine is of form (Q, R, σ, q_0, F) where Q is the finite set of states, R is the input alphabet, $\sigma : Q \times R \mapsto Q$ is the next-state function, q_0 is the initial state, and F is the set of final states.

In component automaton C_1, for a transition $s_1 \xrightarrow{a/T} s_2$, the component will trigger the calling of required traces of T, when a is invoked by the environment. Before giving a full definition of how two component automata are composed, we show how a finite state machine accepting T is composed with a component automaton, which we call *internal product*.

We give a general definition of a finite state machine M and a component automaton C under a set of shared events E. We use $f(C, M, E)$ with default value as $false$ to denote whether the component C can satisfy the requirement of M under E.

Let's consider the current states of M and C are q and s, respectively. For example, there exists transition $q \xrightarrow{a} q'$ with $a \in E$ of finite state machine M, this means C should provide a, however, if $a \notin enabled(s)$, this cause the request of a fail. If $a \in enabled(s)$ and there exists $s \xrightarrow{a/T} s'$ for some events from E appearing in T, then the invocation of a also fails.

Given a set of sequences of elements T, we use $events(T)$ to denote all the elements appearing in T.

Definition 5 (internal product).
Given a component automaton $C = (S, s_0, P, R, A, \delta)$ and a finite state machine $M = (Q, q_0, \Sigma, \sigma, F)$, the composition of C and M under E is $C \lhd_E M = (Q', q_0', \Sigma', \sigma', F')$ where

- $Q' = S \times Q$, $q_0' = (s_0, q_0)$;
- $\Sigma' = 2^R \cup 2^{\Sigma^*}$;
- σ' *is the smallest set given by the following rules: Suppose* $(r_1, t_1) \in Q'$ *with* $t_1 \notin F$ *and* $r_1 \notin \{r \mid out(r) = \emptyset\}$, *and* $r_1 \xrightarrow{a/T} r_2$ *with* $a \in E \cup A$ *and* $t_1 \xrightarrow{b} t_2$. *Then,*
  - *if* $events(T) \cap E \neq \emptyset$ *or* $b \in E \wedge b \notin enabled(r_1)$, $f(C, M, E) = true$;
  - *otherwise if* $a \in A$, $(r_1, t_1) \xrightarrow{T} (r_2, t_2) \in \sigma'$;
  - *otherwise if* $b \notin E$, $(r_1, t_1) \xrightarrow{\{b\}} (r_1, t_2) \in \sigma'$;
  - *otherwise* $a == b$, $(r_1, t_1) \xrightarrow{T} (r_2, t_2) \in \sigma'$;
- $F' = \{(r, t) \mid r \in S, t \in F\}$

This can be extended to a sequences of transitions. $t \xRightarrow{\alpha} t'$ and $r \xRightarrow{tr} r'$ where $\pi_1(tr)\downarrow_P = \alpha\downarrow_P$ and $events(\pi_2(tr)) \cap E = \emptyset$, then $(r, t) \xRightarrow{\pi_2(tr')} (r', t')$ where $tr'\downarrow^1_{R\setminus E} = pair(\alpha\downarrow_{R\setminus E})$, $tr'\downarrow^1_{P\cup A} = tr\downarrow^1_{P\cup A}$, $last(\pi_1(tr') \in R)$

Next, we present the conditions of a successful composition of a finite state machine and a component automaton, that is, the set of sequences of elements which are obtained by projection T to the set P are unblockable provided traces in the component automaton where T is the language of the finite state machine

and the corresponding required traces of all the unblockable provided traces will not require services in E.

Lemma 1. *$f(C, M, E) = false$ iff $T\lfloor_P \subseteq \mathcal{T}_{up}(C)$ where T is $\mathcal{L}(M)$ and for each $pt \in T\lfloor_P$ and every $\alpha \in required(pt)$ $events(\alpha) \cap E = \emptyset$.*

Proof. First, we prove the direction " $\Longrightarrow$ " by contraposition. There exists β and $\pi_1(tr)\lfloor_P = \beta$ that $s_0 \stackrel{tr}{\Longrightarrow} s$ and $t_0 \stackrel{\beta}{\Longrightarrow} t$. If (s, t) is reachable and $a \notin enabled(s)$ and $t \stackrel{a}{\rightarrow} t'$, so $f(C, M, E) = true$. If $s \stackrel{a/T}{\longrightarrow} s'$ with $events(T) \cap E \neq \emptyset$, so $f(C, M, E) = true$.

Next we show " $\Longleftarrow$ " by contraposition too. If $f(C, M, E) = true$, there exists reachable state (r, t) that $r \stackrel{a/T}{\longrightarrow} r'$ where $events(T)$ $E \neq \emptyset$, then there exists $pt \in T\lfloor_P$ and $\alpha \in required(pt)$ that $events(\alpha) \cap E \neq \emptyset$. Or $t \stackrel{b}{\rightarrow} t'$ with $b \in E$, but $b \notin enabled(r)$, then there exists $\beta \in T\lfloor_P$ but $\beta \notin \mathcal{T}_{up}(C)$ □

Given a component automaton $C = (S, s_0, P, R, A, \delta)$, we use $C(s)$ with $s \in S$ to denote the component automaton (S, s, P, R, A, δ) with s as the initial state.

Two component automata $C_1 = (S_1, s_0^1, P_1, R_1, A_1, \delta_1)$ and $C_2 = (S_2, s_0^2, P_2, R_2, A_2, \delta_2)$ are *composable*, if $(P_1 \cup R_1) \cap A_2 = (P_2 \cup R_2) \cap A_1 = \emptyset$. The shared events $shared(C_1, C_2)$ is $(P_1 \cap R_2) \cup (P_2 \cap R_1)$ and written as *shared* when there is clear about which two components are composed.

Definition 6. *For two composable component automata $C_1 = (S_1, s_0^1, P_1, R_1, A_1, \delta_1)$ and $C_2 = (S_2, s_0^2, P_2, R_2, A_2, \delta_2)$, the product $C_1 \otimes C_2 = (S, s_0, P, R, A, \delta)$.*

- $S \subset S_1 \times S_2$, $s_0 = (s_0^1, s_0^2)$;
- $P = (P_1 \cup P_2)$;
- $R = (R_1 \cup R_2) \setminus (P_1 \cup P_2)$;
- $A = A_1 \cup A_2$;
- δ *is defined as follows:*
  - *for any reachable state $(s_1, s_2) \in S$ and $s_1 \stackrel{w/T}{\longrightarrow} s_1' \in \delta_1$, denoted the internal product $C_2(s_2) \lhd_{shared} \mathcal{M}(T)$ by (Q, R, σ, q_0, F),*
    - † *if $f(C_2(s_2), \mathcal{M}(T), shared) = true$, then $(s_1, s_2) \stackrel{w/}{\longrightarrow} f \in \delta$;*
    - † *otherwise $\{(s_1, s_2) \stackrel{w/T'}{\longrightarrow} (s_1', s_2') \mid s_2' \in S_2, T' = \{conc(\ell) \mid \ell \in \mathcal{L}(Q, R, \sigma, q_0, \{(s_2', t)\}), (s_2', t) \in F\}\} \subseteq \delta$;*
  - *Symmetrically, for any reachable state $(s_1, s_2) \in S$ and $s_2 \stackrel{w/T}{\longrightarrow} s_2' \in \delta_2$, we add some transitions to δ similarly to the above.*

Next, we show how two components are composed by traces, or we can also say how T' is obtained directly by traces without internal product. This is inspired by how traces of interleaving and parallel composition operators are defined in CSP [7].

Lemma 2. *If $f(C_2(s_2, \mathcal{M}(T), shared) = false$ in the product, T' can also be defined as $T' = \{\pi_2(tr') \mid \alpha \in T, s_2 \stackrel{tr}{\Longrightarrow} s_2',$ with $\pi_1(tr)\lfloor_{P_2} == \alpha\lfloor_{P_2}, tr'\lfloor_R^1 = pair(\alpha\lfloor_R), tr'\lfloor_{P_2 \cup A_2}^1 = tr^1\lfloor_{P_2 \cup A_2}, last(\pi_1(tr') \in R_1)\}$*

Proof. To differentiate T' in the product and lemma, we denote the set T' defined in this lemmas as T'', then we prove $T' = T''$ by $T' \subseteq T''$ and $T'' \subseteq T'$.

□

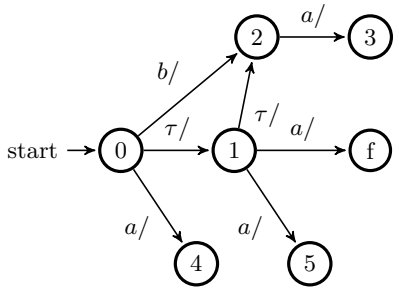

Fig. 5. failure transitions

For states that lead to the failure state internally, we only need to remove these states. However it is more complicated for provided transitions that lead to the failure state.

Let's see the example shown in Fig.5. At state 1, there is a a transition leading to a failure state, that means, it may fail to require a. So, we need to block a at state 1. There is an internal transition from 1 to state 2 where a is enabled. So we also need to remove a transition from 2. But this will also make the trace $\langle b, a\rangle$ blocked, which is actually unblockable.

Now, we present Algorithm 2 to remove failure states. First We remove all the states transiting to the failure state internally. Let's consider a state s that $s \xrightarrow{a} f$, so we need to remove all the $'a'$ transitions from states that are internally reached from s. We assume that $r \in \mathcal{R}(s)$ and $a \in out(r)$. If all states transiting to r are also internally reached from s, we can directly remove $'a'$ transition from r. Otherwise, we add a new state r' that has same outgoing transitions with r except $'a'$ transition and delete the internal transitions from s to r which are added as new internal transitions from s to r' correspondingly.

Hiding In this part, we introduce the hiding operator, which is used to internalize certain provided services.

Definition 7 (hiding). *Given a component automaton $C = (S, s_0, P, R, A, \delta)$ and a set of provided events $E \subseteq P$, hiding E from C, written as $C\backslash E$ is $(S, s_0, P \setminus E, R, A \cup E, \delta)$.*

5 Refinement

In this section, we will study refinement relation between component automata. Refinement is one of the key issues in component based development. It is mainly

Algorithm 2: Remove failure transitions

Input: $C = (S, s_0, P, R, A, \delta)$
Output: $\mathcal{F}(C) = (S_f, s_f^0, P, R, A, \delta_f)$,
1: **Initialization:** $s_f^0 := s_0$; $\delta_f := \delta$;
2: $S_f := S_f \setminus \{s' \mid f \in \mathcal{R}(s')\}$
3: **for each** s,**there exists** a **that** $s \xrightarrow{a/} f$ **do**
4: $E := \{a \mid s \xrightarrow{a/} f\}$
5: $Q' := \{t \in \mathcal{R}(s) \mid \exists b \in E \cap out(t)\}$
6: **for each** $r \in Q'$ **do**
7: $Q_1 := \{t \in \mathcal{R}(s) \mid t \xrightarrow{w/} r\}$.
8: **if** $\{t \mid t \xrightarrow{w/} r\} == Q_1$ **then**
9: $\delta_f := \delta_f \setminus setr \xrightarrow{a/} t \mid a \in E, t \in S_f$
10: **else**
11: **add a fresh state** r' **to** S_f
12: $\delta_f := \delta_f \cup \{t \xrightarrow{w/} r' \mid t \in Q_1, t \xrightarrow{w/} r\} \cup \{r' \xrightarrow{w/} t \mid r \xrightarrow{w/} t, w \notin E\}$
13: $\delta_f := \delta_f \setminus \{t \xrightarrow{w/} r \mid t \in Q_1\}$
14: **end if**
15: **end for**
16: **end for**
17: $S_f := S_f \setminus \{f\}$.

for substitution. We will give a refinement relation by state simulation technique. The intuitive idea is a state s' simulates s, if at state s' more provided events are enabled, less required traces are required and the next states following the transitions keep the simulation relation. For a binary relation $\mathcal{R}$, we write $s_1 \mathcal{R} s_2$ when $(s_1, s_2) \in \mathcal{R}$.

Definition 8 (simulation). *A binary relation $\mathcal{R}$ over the set of states of a component automaton is a simulation iff whenever $s_1 \mathcal{R} s_2$:*

- *if $s_1 \xrightarrow{w/T} s_1'$ with $w \in A \cup enabled(s_1)$, there exists s_2 and T' such that $s_2 \xrightarrow{w/T'} s_2'$ where $T' \subseteq T$ and $s_1' \mathcal{R} s_2'$;*
- *for any transitions $s_2 \xrightarrow{w/T'} s_2'$ with $e \in A \cup enabled(s_1)$, then there exists s_1 and T such that $s_1 \xrightarrow{w/T} s_1'$ where $T' \subseteq T$ and $s_1' \mathcal{R} s_2'$;*
- *if $s_2 \xrightarrow{w/} f$ with $e \in A \in P_1$, then $s_1 \xrightarrow{w/} f$.*

We say that s_2 simulates s_1, written $s_1 \lesssim s_2$, if $(s_1, s_2) \in \mathcal{R}$. C_2 refines *C_1, if there exists a simulation relation $\mathcal{R}$such that $s_1^0 \mathcal{R} s_2^0$.*

Theorem 2. *Given any component automaton C, $\mathcal{I}(C) \sqsubseteq C$.*

Proof. Let $\mathcal{R} = \{((Q, s), s) \mid s \in S, (Q, s) \in S_I\}$. We show $\mathcal{R}$ is a simulation relation.

For any $(Q, s) \mathcal{R} s$,

- $\diamond$ $enabled((Q,s)) \subseteq s$.
- $\diamond$ $(Q,s) \xrightarrow{a/T} (Q',s')$ with $a \in enabled(Q,s)$. From for-loop(line 6-12), then $s \xrightarrow{a/T} s'$. So, $(Q',s')\mathcal{R}s'$; item $(Q,s) \xrightarrow{e/T} (Q',s')$ with $e \in A$, from for-loop(14-16), $s \xrightarrow{;w/T} s'$ and $(Q',s')\mathcal{R}s'$.
- $\diamond$ for any $s \xrightarrow{e/T} s'$, there exists Q that $s, s' \in Q$ from line 5. So $(Q,s) \xrightarrow{e/T} (Q,s')$ such that $(Q,s')\mathcal{R}s'$
- $\diamond$ For any $s \xrightarrow{a/T} s'$ with $a \in enabled(Q,s)$.From Line 11 in for-loop(6-12), then there exists $(Q,s) \xrightarrow{a/T} (Q',s')$ and $(Q',s')\mathcal{R}s'$.

From above, we see $\mathcal{R}$ is a simulation relation. So $\mathcal{I}(C) \sqsubseteq C$, because $(\{s_0\}, s_0)\mathcal{R}s_0$. □

Theorem 3. *Given two component automata C_1 and C_2 such that $C_1 \sqsubseteq C_2$, then $\mathcal{T}_p(C_1) \subseteq \mathcal{T}_p(C_2)$ and for any provided trace $pt \in \mathcal{T}_p(C_1)$, there is $required_2(pt) \subseteq required_1(pt)$.*

Proof. This can be easily proved by induction on the length of pt □

Theorem 4. *Given four component automata $C_1 = (S_1, s_0^1, P_1, R_1, \delta_1)$, $C_1' = (S_1', s_0^{1'}, P_1', R_1', \delta_1')$, $C_2 = (S_2, s_0^2, P_2, R_2, \delta_2)$, and $C_2' = (S_2', s_0^{2'}, P_2', R_2', \delta_2')$, if $C_1 \sqsubseteq C_1'$ and $C_2 \sqsubseteq C_2'$, then $C_1 \otimes C_2 \sqsubseteq C_1' \otimes C_2'$.*

Proof. Let the simulation relation of $C_1 \sqsubseteq C_1'$ and $C_2 \sqsubseteq C_2'$ be $\mathcal{R}_1$ and $\mathcal{R}_2$. Next, we prove the relation $\mathcal{R} = \{((s_1,s_2),(s_1',s_2')) \mid s_1\mathcal{R}_1s_1', s_2\mathcal{R}_2s_2'\}$ be a simulation relation. Let $(s_1,r_1)\mathcal{R}(s_1',r_1')$. It is easy to see that (s_1',r_1') leads to failure states imply that (s_1,r_1) leads to failure states too by Lemma. **??** If $(s_1,r_1) \xrightarrow{w/T_1} (s_2,r_2)$, we can assume that $s_1 \xrightarrow{w/T} s_2$ and $r_1 \xRightarrow{tr} r_2$. By simulation, we get that $s_1' \xrightarrow{w/T'} s_2'$ and $r_1 \xRightarrow{tr'} r_2$, so $(s_1',r_1') \xrightarrow{w/T_2} (s_2',r_2')$.

For any $(s_1',r_1') \xrightarrow{w/T_2} (s_2',r_2')$, we assume that $s_1' \xrightarrow{w/T} s_2'$ and $r_1' \xRightarrow{tr} r_2'$. So there exists $s_1 \xrightarrow{w/T'} s_2$ and $r_1 \xRightarrow{tr'} r_2$ where $T \subseteq T'$, $\pi_1(tr')\downarrow_{shared} \in T'$, and $pi_2(tr) \subseteq \pi_2(tr')$. □

6 Conclusion and Future Work

We have presented an execution model describing the interaction and internal behaviors of software components based on our previous work in which internal behaviors of components are not considered. Then, we studied the unblockable behaviors of the components, that is, provided services that are not blocked at a given state and sequences of provided services that are not blocked. To this end, we have developed an algorithm to generate an interface model in which all provided services are unblockable from a component's execution model.

In order to provide a service in the end, the component needs to call a set of sequences of required services from the environment, which can be recognized

by a finite state machine. To this end, we firstly provide the composition rules of how a finite state machine recognizing the required traces of a given provided service of one component synchronizes with the component automaton that is to be composed with. We also define the composition based on traces, which is proved to be equivalent with syntactic way of composition. Then, we show how two components are composed based on the execution models. The failure state is introduced for the situation when a required service is not provided as claimed or there is cyclic invocation during the service invocation. We have developed an algorithm to remove the failure transitions.

We have developed a refinement relation between components based on state simulation. The intuitive idea is that a refined component can provide more unblockable services while requiring less. We have proved that the refinement relation is kept during composition which implies that the refinement relation is appropriate for component substitution.

There are several open problems left for the future work. Firstly, The components discussed in this paper provide services in a passive way, that is, the components only triggers invocation of services when provided services are called by the environment or when some internal behaviors are available. The kind of components used for actively coordinating the behaviors of several components is needed. Secondly, algebraic properties of composition such as associative, commutative, distributive of coordination over composition are also important. The third research direction is development of execution and interface models for components with timing characteristics, which support timing, deadlock, and scheduling analysis of applications in the presence of timed requirement.

References

1. L. De Alfaro and T.A. Henzinger. Interface automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.
2. L. De Alfaro and T.A. Henzinger. Interface-based design. *Engineering Theories of Software-intensive Systems*, 195:83–104, 2005.
3. Luca de Alfaro and Thomas Henzinger. Interface theories for component-based design. In Thomas Henzinger and Christoph Kirsch, editors, *Embedded Software*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.
4. Ruzhen Dong, Johannes Faber, Zhiming Liu, Jiri Srba, Naijun Zhan, and Jiaqi Zhu. Unblockable compositions of software components. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE '12, pages 103–108, New York, NY, USA, 2012. ACM.
5. Michael Emmi, Dimitra Giannakopoulou, and Corina S. Pasareanu. Assume-guarantee verification for interface automata. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *LNCS*, pages 116–131. Springer, 2008.
6. Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *ASE*, pages 3–12. IEEE Computer Society, 2002.
7. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

8. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, volume 2. Addison-Wesley, 1979.
9. Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface input/output automata. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 82–97. Springer, 2006.
10. Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.
11. Gerald Lüttgen and Walter Vogler. Modal interface automata. In JosC.M. Baeten, Tom Ball, and FrankS. Boer, editors, *Theoretical Computer Science*, volume 7604 of *Lecture Notes in Computer Science*, pages 265–279. Springer Berlin Heidelberg, 2012.
12. Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.
13. Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
14. J.B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. Modal interfaces: unifying interface automata and modal specifications. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 87–96. ACM, 2009.
15. Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. A modal interface theory for component-based design. *Fundam. Inf.*, 108(1-2):119–149, January 2011.