Unblockable Compositions of Software Components

Ruzhen Dong
UNU-IIST, Macau
ruzhen@iist.unu.edu

Johannes Faber
UNU-IIST, Macau
jfaber@iist.unu.edu

Zhiming Liu
UNU-IIST, Macau
lzm@iist.unu.edu

Jiří Srba
Dept. of Computer Science
Aalborg University, Denmark
srba@cs.aau.dk

Naijun Zhan
State Key Lab. of Comp. Sci.
Institute of Software, CAS
znj@ios.ac.cn

Jiaqi Zhu
State Key Lab. of Comp. Sci.
Institute of Software, CAS
zhujq@ios.ac.cn

ABSTRACT

We present a new automata-based interface model describing the interaction behavior of software components. Contrary to earlier component- or interface-based approaches, the interface model we propose specifies all the non-blockable interaction behaviors of a component with any environment. To this end, we develop an algorithm to compute the unblockable interaction behavior, called the *interface model of a component*, from its execution model. Based on this model, we introduce composition operators for the components and prove important compositionality results, showing the conditions under which composition of interface models preserves unblockable sequences of provided services.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*languages,patterns*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces*

Keywords

Component-based design, interface theory, composition

1. INTRODUCTION

In component-based software engineering, large software systems are decomposed into components with clearly articulated interfaces in order to facilitate a sound development process across different teams of developers. 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. This means that components can be treated as black-boxes, and the theory allows for independent implementation and deployment of components.

Since the 90s, component-based approaches have drawn a lot of attention in software engineering [19, 1, 2, 16] and there has been a considerable research activity studying how interfaces of components can be formally described in order to automatically decide whether two components can be composed together in a well-formed way [18, 8, 7, 13, 6].

We examine to what extent an interface model can be realized in a way that ensures nonblocking executions for all possible compositions of components respecting the given interfaces, and present an automata-based interface model that possesses this desirable property. In our model, a component comprises of a *provided interface* and a *required interface*. The former describes which executions of services the component offers to its environment, while the latter specifies what services the component needs to call in order to provide the services on its provided interface. The *execution model* of a component is in general non-deterministic and thus contains interactions with other components that may potentially lead to deadlocks.

The *interface model* we propose in this paper supports a black-box composition in the sense that 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 sequences of required services are respected by the implementation. This in particular means that any sequence of services specified in the provided interface cannot be blocked when being composed with any other component. We prove that this is the case if and only if the interface model is *input-deterministic*, meaning that after calling any fixed sequence of provided services, the set of currently available provided services is deterministic, and the selection is decided by the environment.

In order to generate an interface model for any given execution model, we give a new algorithm that computes all unblockable sequences of services provided by a component from its execution model. For the application of our model in the context of a component-based design approach, we also introduce composition operators for manipulating the components. The *composition* of components is used to synchronize services provided by one and required by another component. A specific form of composition is the *plugging* of components, where one component only provides services to the other component without requiring any services from it. We prove that unblockable sequences of services provided by the plugging of execution models are the same as those of the plugging of their interface models.

Related work.

There are two broadly-known approaches to interface theories, the I/O Automata [18, 17] and the Interface Au-

CBSE'12, June 26–28, 2012, Bertinoro, Italy.

tomata [8, 7, 9]. Both are centered around an interface-based design and have some similarities to our approach, however, they have been developed with different aims and are based on different assumptions. In fact, we shall argue that our approach is positioned in between these existing approaches, trying to overcome their limitations.

Input/Output (I/O) Automata [18, 17] were defined by Lynch and Tuttle to model concurrent and distributed discrete event systems. The actions are separated into input, output, and internal actions. In this model, input-enabledness is required in the sense that all the input actions must be enabled at any state. Synchronization is realized by broadcasting on the outputs, i.e., an output synchronizes with all corresponding input actions of other automata. The compatibility of two components is checked in a *pessimistic* way, which requires that there is no deadlock for *all* possible environments. On the contrary, our interface model does not enforce that all inputs are always enabled, because this can be sometimes unrealistic to achieve, but we assume input-determinism in order to guarantee that all sequences of provided services accepted by the interface model can be executed without ever being blocked.

To overcome the restrictions of I/O Automata, which are forced to react in a pessimistic way to every possible input, de Alfaro and Henzinger introduced their Interface Automata [8, 7, 9] that are based on the *optimistic* assumption that a component is normally used in a specific environment and thus does not need to react to every input. In this model, methods of a component that can be called are modeled as input actions; external methods that the component calls are output actions; internal actions are used to model internal method invocations. Interface automata guarantee the specified output executions on the assumption that the environment only invokes methods provided as the inputs. The optimistic compatibility of two interface automata implies that two interface automata are compatible if there is at least one environment avoiding deadlock states in the composed product of the two components. Such a condition for compositionality may be too relaxed for certain applications, since the usability of services specified in the interface automata depends on unspecified environments. To this end, Giannalopoulou et al [10, 11] proposed assume-guarantee rules to reason about compatibility, safety and refinement properties, and developed algorithms to generate weakest assumptions for any given interface automaton. Larson et al [12] defined interfaces for input/output automata by splitting assumptions from guarantees. But these assumptions are generated for closing the interface automata, which implies that all events in the composition are internal.

In contrast to the existing approaches, the model we present in this paper combines both the optimistic and pessimistic views for component-based systems: the interface model of a component directly specifies which sequences of provided services guarantee deadlock-free executions. Thus, it is on the one hand not as pessimistic as I/O automata, because all provided services do not have to be available at every state, and on the other hand, it is not as optimistic as interface automata, because we restrict all the possible environments that may make the service invocations blocked.

In the rest of the paper, we shall analyze the properties of our new interface model and demonstrate in which situations it is superior to the previous models and in which cases using the existing techniques is beneficial.

Summary of contributions.

The contributions of this paper are (1) a new interface model ensuring unblockable compositions of software components; (2) an algorithm to generate the interface model of a component based on its execution model; (3) definitions of basic operations used in component-based design; (4) a formal analysis showing that unblockable behavior is preserved for the complete plugging of components.

Outline of the paper.

The rest of the paper is organized as follows. In Sect. 2, we introduce component automata and an algorithmic way to generate their interface models. In Sect. 3, we present the composition operators to compose components and prove important properties for these operators. In Sect. 4 we conclude the paper and discuss the future work.

2. COMPONENT AUTOMATA

In this section, we will first introduce some notions that will be used throughout the paper and then motivate component automata and component interface automata. At last, we will give an algorithm transforming component automata to component interface automata and prove its correctness.

2.1 Preliminary Definitions

Let $A, B \subseteq \mathcal{L}^*$ be two languages. The set concatenation $A \circ B$ is defined as $\{w_A w_B \mid w_A \in A,\ w_B \in B\}$. The first and the second projection on a pair of elements $\ell = (x, y)$ are denoted by π_1 and π_2, and defined by $\pi_1(\ell) = x$ and $\pi_2(\ell) = y$, and it is naturally extended to sequences of pairs of elements. For a sequence $\rho = \langle x_0, x_1, \ldots, x_k \rangle$ over an alphabet $\mathcal{L}$ and a set $X \subseteq \mathcal{L}$, a projection on the elements from X is denoted by $\rho \lfloor X$ and defined as $\langle x_{i_1}, x_{i_2}, \ldots, x_{i_n} \rangle$ where $0 \leq i_1 < i_2 < \ldots < i_n \leq k$ are all the indices of elements such that $x_{i_j} \in X$ for all j, $1 \leq j \leq n$. The empty sequence is denoted as ϵ. The concatenation of sequences tr_1 and tr_2 is denoted by $tr_1 \circ tr_2$, and also extended to sets of sequences such that $T_1 \circ T_2$ contains all concatenations $tr_1 \circ tr_2$ for $tr_1 \in T_1$ and $tr_2 \in T_2$.

2.2 Execution Model of a Component

An execution of a component can be modeled as an alternating sequence of provided service invocations (initiated by the environment) and a set of sequences of required service invocations (initiated by the component). In our automata-based model, the invocation of a provided service is modeled as an atomic *provided event* and the invocation of the required services as a sequence of atomic *required events*. As there may be many choices of required services in order to provide a service, we allow sets of sequences of required events with the intuition that an implementation of the component may choose which of the sequences it actually implements. The formal definition is as follows.

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

- $\diamond$ *S is a finite set of states,*
- $\diamond$ *$s_0 \in S$ is the initial state,*
- $\diamond$ *P and R are disjoint and finite sets of provided and required events, respectively,*

$\diamond$ $\delta \subseteq S \times \Sigma(P,R) \times S$ *is the transition relation, where the set of labels is defined as* $\Sigma(P,R) = P \times (2^{R^*} \setminus \emptyset)$.

Whenever there is $(s, \ell, s') \in \delta$ with $\ell = (a, T)$, we simply write $s \xrightarrow{a/T} s'$ and call it a transition step.

A component automaton is called *closed* if $T = \{\epsilon\}$ for all transitions $s \xrightarrow{a/T} s'$. Otherwise, it is called *open*.

An alternating sequence of states and labels of the form

$$e = \langle s_0, \ell_0, s_1, \ell_1, \ldots, s_k, \ell_k, s_{k+1} \rangle$$

is called an *execution* of the component automaton C if $s_i \xrightarrow{\ell_i} s_{i+1}$ for all i, $0 \leq i \leq k$, where s_0 is the initial state.

A sequence of labels $tr = \langle \ell_0, \ell_1, \ldots, \ell_k \rangle$ is called a *trace of* C if there is an execution e of C such that $tr = e \lfloor \Sigma(P,R)$. In other words, a trace contains only the labels of an execution and abstracts away from the states. The set of all traces of C is denoted as $\mathcal{T}(C)$, and we write $s_0 \stackrel{tr}{\Longrightarrow} s$ if there is an execution from s_0 to s under the trace tr.

A sequence of provided events $pt \in P^*$ is called a *provided trace* if there is a trace tr such that $\pi_1(tr) = pt$. The set of all provided traces of C is denoted as $\mathcal{T}_p(C)$. For a given state s of a component $C = (S, s_0, P, R, \delta)$, the set of provided traces starting from this state s is defined by $\mathcal{T}_p\big((S, s, P, R, \delta)\big)$ and denoted as $\mathcal{T}_p(s)$.

Given a provided trace $pt \in \mathcal{T}_p(C)$, the set *caused*(pt) of caused traces for pt, which are possibly triggered by the component when pt is invoked by the environment, is formally defined as

$$\begin{aligned} \mathit{caused}(pt) = \{ & T_0 \circ \cdots \circ T_k \mid tr \in \mathcal{T}(C), \\ & \pi_1(tr) = pt, \pi_2(tr) = \langle T_0, \ldots, T_k \rangle \} \ . \end{aligned}$$

Example 1. *As a demonstrating example, we consider a simple internet-connection component presented in Fig. 1. It provides the services login, wifi, print, read, and disc to the environment. The services model the logging into the system, request for wifi connection, invocation of printing a document, an email service, and disconnecting from the internet, respectively. The component calls the services unu1, unu2, cserv, getm. These required services model the process of searching for a wifi router nearby, that is, connecting to the unu1 or unu2 wireless network, connecting to an application server, and fetching an email, respectively. The print service is only available for the wifi network unu1 and the read service can be accessed at both networks.*

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

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

Now $pt = \langle \mathit{login}, \mathit{wifi}, \mathit{print} \rangle$ *is a provided trace of the execution* e *and the set of caused traces of* pt *is* $\mathit{caused}(pt) = \{\mathit{unu1}\}$.

2.3 Unblockable Equivalence

The environment interacts with the component in a way that it chooses a sequence of provided events and the component (non-deterministically) determines the sequence of required events that it will request from other components. In the example, we may observe that including the provided event *print* to the interface of the internet connection component from Fig. 1 is not safe, as the sequence of requested services $\langle \mathit{login}, \mathit{wifi}, \mathit{print} \rangle$ can be executed if the connection

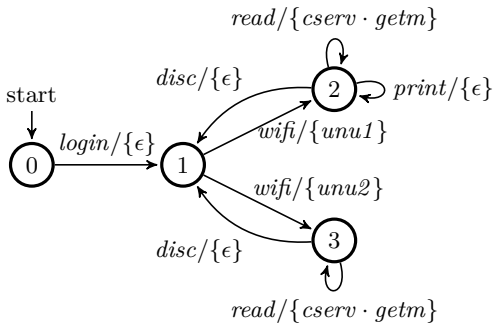

Figure 1: Execution model of internet connection component C_{ic}

is made to the network *unu1*, but it will be impossible if network *unu2* was chosen. This will cause a blocking in the component composition and as the environment is not in control of the selection of the network, exposing *print* to the component interface model is not safe; the service should not be advertised. For this reason, we require that our interface models of components have the property of *input-determinism*, meaning that after any sequence of provided service invocations, the component should be at states with identical provided services. This will guarantee that the provided traces are not blocked during any run-time execution.

Given a component automaton $C = \langle S, s_0, P, R, \delta \rangle$, let us define for any $s \in S$ the set of enabled provided events as $\mathit{enabled}(s) = \{a \in P \mid s \xrightarrow{a/T} s'\}$. Then, we define the concept of input-determinism.

Definition 2 (Input-determinism). *A component automaton is* input-deterministic *iff* $s_0 \stackrel{tr_1}{\Longrightarrow} s_1$ *and* $s_0 \stackrel{tr_2}{\Longrightarrow} s_2$ *with* $\pi_1(tr_1) = \pi_1(tr_2)$ *imply* $\mathit{enabled}(s_1) = \mathit{enabled}(s_2)$.

Now we can also define the notion of an unblockable trace. An unblockable provided trace is never blocked by the component in the sense that the component can always provide the trace independent of how the internal choices of the causality traces are resolved in an implementation of the component.

Definition 3 (Unblockable Trace). *We call a provided trace* $pt = \langle a_0, \ldots, a_k \rangle$ *of a component automaton* $C = (S, s_0, P, R, \delta)$ unblockable *iff whenever* $s_0 \stackrel{tr}{\Longrightarrow} s$ *such that* $\pi_1(tr) = \langle a_0, \ldots, a_i \rangle$ *for some* i, $0 \leq i \leq k-1$, *then* $a_{i+1} \in \mathit{enabled}(s)$. *A trace in* C *is* unblockable *iff its provided trace is unblockable. The set of all unblockable traces of* C *is denoted by* $\mathcal{T}_u(C)$.

The next theorem shows that the property of input-termnism guarantees that all traces are unblockable and vice versa. Due to space limitations, we cannot present the proofs here.

Theorem 1. *A component automaton* $C = (S, s_0, P, R, \delta)$ *is input-deterministic iff all of its traces are unblockable.*

For the purpose of our interface theory, we will consider two component automata equivalent whenever their sets of unblockable traces are the same.

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

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

1: **Initialization:** $S_I := \{(\{s_0\}, s_0)\}$; $\delta_I := \emptyset$; $todo := \{(\{s_0\}, s_0)\}$; $done := \emptyset$
2: **while** $todo \neq \emptyset$ **do**
3: **choose** $(Q, r) \in todo$; $todo := todo \setminus \{(Q, r)\}$; $done := done \cup \{(Q, r)\}$
4: **for each** $a \in \bigcap_{s \in Q} enabled(s)$ **do**
5: $Q' := \bigcup_{s \in Q} \{s' \mid (s \xrightarrow{a/T} s') \in \delta\}$
6: **for each** $(r \xrightarrow{a/T} r') \in \delta$ **do**
7: **if** $(Q', r') \notin (todo \cup done)$ **then**
8: $todo := todo \cup \{(Q', r')\}$
9: $S_I := S_I \cup \{(Q', r')\}$
10: **end if**
11: $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{a/T} (Q', r')\}$
12: **end for**
13: **end for**
14: **end while**

Definition 4 (Unblockable Equivalence). *Let C_1 and C_2 be two component automata. They are equivalent with respect to unblockable traces, denoted by $C_1 \equiv C_2$, if $\mathcal{T}_u(C_1) = \mathcal{T}_u(C_2)$. Similarly, we write $C_1 \equiv_{pt} C_2$ if the sets of unblockable provided traces of C_1 and C_2 are equal.*

2.4 Interface Model of a Component

In the example shown in Fig. 1, the provided trace $\langle$*login*, *wifi*, *print*$\rangle$ may be blocked, when the component resolves the non-determinism by choosing to call *unu*2. This reveals that the component contains a potential deadlock when composed with another component using the *print* service. Thus, we propose an input-deterministic model to support deadlock-free black-box composition as the interface model.

Definition 5. *An* interface automaton *is an input deterministic component automaton.*

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 to its unblockable traces with the component automaton. 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 of the form (Q, r) consisting of a subset of states Q 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 a provided trace in C, which is used to identify all non-blocking provided traces. 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 the C are simulated for the non-blockable traces.

Now we can present an important theorem stating that the constructed interface automaton preserves the set of unblockable traces of the input component automaton.

Theorem 2. *Algorithm 1 terminates for every given component automaton C and produces a unique interface automaton $\mathcal{I}(C)$ such that $C \equiv \mathcal{I}(C)$.*

Corollary 1. *Whenever $C_1 \equiv C_2$ for two component automata C_1 and C_2 then $\mathcal{I}(C_1) \equiv \mathcal{I}(C_2)$ also holds.*

3. COMPOSITION OPERATORS

"Components are for composition." [19] A component interacts with other components by providing services or requiring services. A closed component provides services without the need to require services from other components. We consider closed components as stable service providers, while open components will provide services under assumptions that the required services are guaranteed. Composition allows for components interacting with each other to build up new components. It also enables reusability and decomposition of components. In this section, we will introduce some basic composition operators including the product and the plugging of components.

3.1 Product of Component Automata

The product of two components assembles the components together and synchronizes them when a sequence of required service calls can be provided by the other component. We call two components composable if their provided services are disjoint, because invocations to joint services will trigger both of the components, which will cause nondeterministic executions.

Let us now motivate the product composition of component automata. The definition is similar to the standard product for finite automata, except that we synchronize transitions if a caused service of one component is provided by a corresponding transition of the other component, e.g., for transitions

$$r_1 \xrightarrow{a/\{b_1 \cdot b_2\}} r_1' \text{ and } r_2 \xrightarrow{b_1/\{c_1 \cdot c_2\}} r_2'$$

in C_1 and C_2, the synchronized transition will be

$$(r_1, r_2) \xrightarrow{a/\{c_1 \cdot c_2 \cdot b_2\}} (r_1', r_2').$$

In the definition of product, there is an additional condition for synchronization. Assume that there also exists

$$r_1 \xrightarrow{a/\{d_1\}} r_1''$$

where d_1 is a provided event of C_2 but $d_1 \notin enabled(r_2)$. With this, the component C_1 can trigger the service d_1 that is declared as provided in C_2, but component C_2 cannot provide it in r_2. This causes a potential deadlock in the component C_1, because whether $b_1 \cdot b_2$ or d_1 is called is determined by the component internally, in other words, it is invisible to the outside. The solution to avoid the potential internal deadlock is to make a unavailable at state (r_1, r_2). Informally, should the product construction guarantee a at state (r_1, r_2), it must require that all caused traces of a at state r_1 restricted to the set of provided services of C_2 are also provided by C_2 at state r_2.

We use the notation $caused(s,a) = \{\alpha \mid s \xrightarrow{a/T} s', \alpha \in T\}$ to denote the set of all caused traces for the provided event a at state s; by $events(T)$ we denote the set of required events that occur in the set T of required event sequences.

DEFINITION 6 (PRODUCT). *Two component automata $C_1 = (S_1, s_0^1, P_1, R_1, \delta_1)$ and $C_2 = (S_2, s_0^2, P_2, R_2, \delta_2)$ are composable if $P_1 \cap P_2 = \emptyset$. The product $C_1 \otimes C_2$ is then defined as a component automaton (S, s_0, P, R, δ) where*

- $S = S_1 \times S_2$,
- $s_0 = (s_0^1, s_0^2)$,
- $P = (P_1 \setminus R_2) \cup (P_2 \setminus R_1)$,
- $R = (R_1 \setminus P_2) \cup (R_2 \setminus P_1)$, *and*
- δ *is the smallest relation constructed as follows.*

  Let $r_1 \xrightarrow{a/T} r_1' \in \delta_1$ with $a \in P$ and $r_2 \in S_2$. If there is a sequence $\alpha \in caused(r_1, a)$ such that $\alpha\lfloor P_2 \notin \mathcal{T}_p(r_2)$—that is, a service from the caused traces in r_1 which is provided by the second component is blocked by the possible executions starting in r_2—then there is no corresponding a transition in δ. Otherwise, we distinguish between a non-synchronization step and a synchronized step.

  1. *Non-synchronized transition step: if $events(T) \cap P_2 = \emptyset$, then*

     $$(r_1, r_2) \xrightarrow{a/T} (r_1', r_2) \in \delta.$$

  2. *Synchronized transition step: if $events(T) \cap P_2 \neq \emptyset$, then*

     $$(r_1, r_2) \xrightarrow{a/T'} (r_1', r_2') \in \delta$$

     whenever $r_2 \xRightarrow{\langle (a_0,T_0),\ldots,(a_k,T_k)\rangle} r_2'$ *with*

     $$\langle a_0, \ldots, a_k \rangle = \beta\lfloor P_2, \beta \in T$$

     and

     $$T' = \{\beta[\beta_0/a_0, \beta_1/a_1, \ldots, \beta_k/a_k] \mid \beta_i \in T_i, 0 \leq i \leq k\}.$$

  These presented rules are based on transitions from δ_1; the rules for transitions from δ_2 are symmetric.

The product of two components can generally resolve potentially blocking executions, or in other words, the product of two components can have more unblockable executions than the unblockable traces produced by its constituents. Since the interface only contains the unblockable executions of the model, it is in general not possible to derive the interface of a product from the interfaces of the components.

For this reason, we introduce a special case of product, the plugging of components, where the second component does not require any events provided by the first one. This plugging product has the desired congruence property that the interface model of the composed components can be derived from the interfaces of its constituents.

DEFINITION 7 (PLUGGING). *A component $C_1 = (S_1, s_0^1, P_1, R_1, \delta_1)$ is* pluggable *(can be plugged) by a component automaton $C_2 = (S_2, s_0^2, P_2, R_2, \delta_2)$ if C_1 and C_2 are composable and $P_1 \cap R_2 = \emptyset$. The plugging of C_1 by C_2, denoted by $C_1 \ll C_2$, is then given as the product $C_1 \otimes C_2$.*

We shall now discuss the properties of the plugging operator. The lemma below shows that a provided trace of a component automaton C_1 is preserved in the product with a component automaton C_2 if its causality traces projected to the services provided by C_2 are unblockable in C_2.

In the rest of this section we fix two components $C_1 = (S_1, s_0^1, P_1, R_1, \delta_1)$ and $C_2 = (S_2, s_0^2, P_2, R_2, \delta_2)$ such that C_1 is pluggable by C_2. Let $C_1 \ll C_2 = (S, s_0, P, R, \delta)$.

LEMMA 1. *Let C_1 be pluggable by C_2. If $s_0^1 \xRightarrow{tr_1} r_1$ in C_1 and $s_0^2 \xRightarrow{tr_2} r_2$ in C_2 with $\pi_1(tr_2) \in Q$ where $Q = \{\alpha\lfloor P_2 \mid \alpha \in \pi_2(tr_1)\}$ and all traces in Q are unblockable in C_2, then $(s_0^1, s_0^2) \xRightarrow{tr} (r_1, r_2)$ for some tr with $\pi_1(tr) = \pi_1(tr_1)$.*

The general plugging does not preserve unblockable traces for the following reason: If there are some services provided by C_2 available also to the environment, the behavior of C_2 is uncertain and the unblockable traces cannot be preserved from C_1 to $C_1 \ll C_2$. Therefore, we consider a special case of plugging, *complete plugging*, requiring that all services of C_2 are provided only to C_1 and are not available to the environment, formally $P_2 \subseteq R_1$.

THEOREM 3. *Let C_1 be pluggable by C_2 such that $P_2 \subseteq R_1$ (complete plugging). Then a provided trace pt is unblockable in $C_1 \ll C_2$ if and only if pt is unblockable in C_1 and all traces in $\{\alpha\lfloor P_2 \mid \alpha \in caused(pt)\}$ are unblockable in C_2.*

3.2 Composition of Interface Models

Every component interface automaton is also a component automaton, so the definitions of product and plugging can also be used to define the composition of interface models. But the product construction on interface models may drop some unblockable traces that could be present in the product of their execution models. So for the interface models, we can compose them only when the plugging condition is satisfied. In that case, the composition of interface models is significant in the sense that unblockable traces are preserved.

Hence we define the composition of two pluggable component interface automata as the product construction introduced in the previous subsection, to which the algorithm $\mathcal{I}$ is applied for removing the blockable executions.

DEFINITION 8. *Given two component interface automata I_1 and I_2, if I_1 can be plugged by I_2, then their composition (plugging), denoted by $I_1 \ll_{\mathcal{I}} I_2$, is defined as $\mathcal{I}(I_1 \ll I_2)$.*

We can now show that complete plugging of two components is equivalent (w.r.t. to unblockable equivalence) with the composition of the corresponding interface models. This property shows that unblockable provided traces allowed in the composition of interface models are also available in the plugging of their execution models.

THEOREM 4. *Let C_1 be pluggable by C_2 such that $P_2 \subseteq R_1$ (complete plugging). Then*

$$\mathcal{I}(C_1) \ll_{\mathcal{I}} \mathcal{I}(C_2) \equiv_{pt} C_1 \ll C_2 \ .$$

4. CONCLUSION AND FUTURE WORK

We have presented a formal model for component interfaces that guarantees unblockable composition of software components. To this end, we have provided an algorithm to compute the unblockable interface from a component's execution model. This interface serves as a behavioral contract for the component: as long as the environment calls only the sequences of provided services that are specified by the interface and all required services are provided, then non-blocking behavior is guaranteed.

During the construction of a software system from components, one is in the end interested in closed components where all needed required services are resolved, i.e., there are no dangling references to required services. Therefore, we introduced the operators of composition and plugging. We showed that plugging preserves unblockable behavior such that the unblockable sequences of a plugging can be computed from the interfaces of the plugged components. On the other hand, such a result does not hold for the more general composition. The reason is that a cyclic dependence between components may in principle resolve a blockable behavior such that the composition has a larger interface than the one computed from the interfaces of the components.

Due to this restriction, the presented interface model is not suited to reflect low-level interaction protocols with cyclic behaviour, but instead describes, on a higher level, the interfaces of components in service-oriented computing, where the non-cyclic service call does usually not impose any practical restriction as mentioned in e.g. [3, 5, 4].

There are several open problems left for the future work. We have shown that the plugging operator preserves the unblocking behavior of its constituents, but it is so far not clear for which other composition operators this property holds too. Similarly, it has to be investigated whether the composition operators preserve a refinement relation on components. Intuitively, a refined component provides more services while relying on fewer invocations to required services. It seems that the notion of modal refinement [15, 14] is applicable in this situation. Another research direction are interface models with timing characteristics to support timing analysis of components and scheduling analysis of application processes as well as unblockable behavior in the presence of timed synchronization.

5. ACKNOWLEDGMENTS

This work has been supported by the projects ARV and GAVES funded by Macao Science and Technology Development Fund and the National Natural Science Foundation of China (Grant No. 60970031, 91118007). Partial support was provided by MT-LAB, VKR Centre of Excellence and the Sino-Danish Center IDEA4CPS.

6. REFERENCES

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[2] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, June 2004.

[3] M. Bartoletti, P. Degano, and G. L. Ferrari. Enforcing secure service composition. In *CSFW*, pages 211–223. IEEE Computer Society, 2005.

[4] M. Boreale, R. Bruni, R. D. Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *FMOODS*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.

[5] Z. Chen, Z. Liu, A. P. Ravn, V. Stolz, and N. Zhan. Refinement and verification in component-based model-driven design. *Sci. Comput. Program.*, 74(4):168–196, 2009.

[6] A. David, K. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC*, pages 91–100. ACM, 2010.

[7] L. De Alfaro and T. Henzinger. Interface automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[8] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In T. Henzinger and C. Kirsch, editors, *Embedded Software*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.

[9] L. De Alfaro and T. Henzinger. Interface-based design. *Engineering Theories of Software-intensive Systems*, 195:83–104, 2005.

[10] M. Emmi, D. Giannakopoulou, and C. S. Pasareanu. Assume-guarantee verification for interface automata. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM*, volume 5014 of *LNCS*, pages 116–131. Springer, 2008.

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

[12] K. G. Larsen, U. Nyman, and A. Wasowski. Interface input/output automata. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 82–97. Springer, 2006.

[13] K. G. Larsen, U. Nyman, and A. Wasowski. Modal I/O automata for interface and product line theories. In R. D. Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[14] K. G. Larsen, U. Nyman, and A. Wasowski. On modal refinement and consistency. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 105–119. Springer, 2007.

[15] K. G. Larsen and B. Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society, 1988.

[16] Z. Liu, C. Morisset, and V. Stolz. rCOS: Theory and tool for component-based model driven development. In F. Arbab and M. Sirjani, editors, *FSEN*, volume 5961 of *LNCS*, pages 62–80. Springer, 2009.

[17] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.

[18] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[19] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.