Towards a Failure Model of Software Components

Ruzhen Dong[1,2], Naijun Zhan[3]

[1] Dipartmento di Informatica, Università di Pisa, Italy
[2] UNU-IIST, Macau
ruzhen@iist.unu.edu
[3] State Key Lab. of Computer Science, Institute of Software, CAS
znj@ios.ac.cn

Abstract. We present a failure model for describing sequences of services that are provided and required, and services that may be blocked by software components. For any automata-based model introduced in our previous work, there is a corresponding failure model. We show that the failure model is expressive enough to describe non-blockable properties defined in the automata-based models. Plugging operation of failure models is given and proved to be consistent with plugging operation of automata-based models. A kind of components, called coordinators, are then introduced to coordinate behaviors of components, and coordination operation is presented too. An algorithm is developed to generate a coordinator which can filter out sequences of provided service invocations that may be blocked.

Keywords Component-based design, Interface theory, Denotational semantics, Coordination, Composition

1 Introduction

Component-based software development was set to build large software systems by using existing software components [30,24,14]. In order to facilitate a sound development process across different teams of developers exploiting existing software components, interface theories [1,7,16,6,29,20] should then define the basic principles for composing several software components based on their interfaces, as the concrete implementation of the components is invisible to its environment, which are the components that it interacts with.

In our previous work [10,11,9], we developed automata-based models describing how a component interacts with the environment via providing and requiring services. We assume *run-to-completeness* of provided service invocations, which means that an invocation of a provided service either is not executed at all, or has to be completed, cannot be interrupted by its environment during the execution. The interface model is developed to guarantee that all sequences of services specified should not be blocked. An algorithm that generates an interface model, whose non-blockable behaviors are same as the considered component is developed.

Two component automata synchronize on the shared events that are provided by one and required by the other. In this paper, we present a plugging operation that plugs a service provider(a component which doesn't require any services) into the other component. Plugging operation reflects the development process of software systems in practice that primitive components that only provide services are implemented first and then plugged into the components that require these services. A refinement relation based on state simulation [25,7] is given in [11] and it is suitable for substitution of interface models.

A failure model is presented in this paper, inspired by the failure-divergence semantics [17,28]. The model explicitly illustrates the sequences of services that are provided and required, and the services that are refused/blocked by the components. The non-blockableness of sequences of service invocations, input-determinism, and plugging operation are all defined in the domain of failure models. Some primary results are presented in this paper, and our roadmap is to prove that the failure model can serve as a complete and sound denotational semantic model for component automata.

All the provided services specified in the interface can always be invoked by any environment. In some context, certain sequences of services need to be filtered out [4]. A coordinator is simple a deterministic labeled transition system, which serve as a wrapper/adapter controlling the sequences of services that are allowed to be called by the environment. The functionality of coordinators is demonstrated clearly in the coordination operation. Coordinators can also be used to filter out the possible blockable service invocations and an algorithm is developed to produce an interface coordinator that filters out all the possible blockable services and keeps all the non-blockable provided traces of the original component.

Related work. The work is based on the rCOS unified model of components [16,5,31,20,6] that define components in terms of their provided and required interface interaction behaviors, data model and local data functionality (including those models in OO programming [15]). This paper focuses on sets of services that are refused and proposes to develop a denotational semantic model of components.

There are two main well known approaches to interface theories, the Input/Output(I/O) Automata [23,22] and the Interface Automata [7,1,8]. There are also some works extending these with modalities [19,26,27,21] and compatibility checking [12,13,18]. The compatibility defined Interface Automata [7,1,8] is in the optimistic way that two components are compatible if there exists an environment that can make the composition avoid the error state, which limits the use of components as black-box units for building software components, while in Input/Output(I/O) Automata [23,22], property input-enabled is required that all input should be enabled at any state and compatibility is defined in the pessimistic way that the composition should work for any environment, which are not practical in reactive components where there are guards for service invocations. The interface model we proposed is input-deterministic that all the services provided by the interface can be called without being blocked if the

required services are satisfied. However, all of these are based on operational semantics or game semantics of software components. In this paper, we try to give a denotational description of software components, called failure model, and get some primary results. The denotational model provides a new perspective of software components, an easier way for composition, and more intuitive understanding of input-determinism.

Reo [2,3] is a well known channel-based coordination model and focus mainly on how connectors are composed without considering specific components. In this paper, we use labeled transition system, which is not new, as coordinator to coordinate component models introduced in this paper. Components are constrained by coordinators for different uses which shows the flexibility of software components.

Summary of contributions. The contributions of this paper are (1) a failure model of components that exhibits sequences of non-blockable provided services in a more intuitive way (2) a coordinator model that is used to coordinate service invocations between components (3) coordination operation (4) an algorithm producing a coordinator that filters out all the possible blockable service invocations.

Outline of the paper. The rest of the paper is organized as follows. In Sect. 2, we give a brief view of our previous work [10,11], that is, component automata, component interface automata, and refinement relation. In Sect. 3, failure model of components is presented, related definitions are given, and plugging operation is given in the domain of failure models which are proved to be consistent with the plugging operation between component automata. In Sect. 4, coordinators are motivated and formally defined, coordination operation is given and the algorithm producing a coordinator for the interface model is developed. In Sect. 5, we conclude the paper and discuss future work.

2 Component Automata

In this part, we will give a brief description of the automata-based models of software components introduced in [10,11]. A primitive component consists a provided and required interface which describes the services provided and required by the component, respectively.

The automata-based models are operational structure of components, and in this paper, we present a trace-based model, in which components are modeled as sets of sequences of pairs of provided and required events(called *traces*), and sets of provided events that are refused/blocked after given traces.

2.1 Preliminary Definitions

This part illustrates the notions that will be used through this paper. 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 $\{w_1 \circ w_2 \mid w_1 \in A,\ w_2 \in B\}$ where $A, B \subseteq \mathcal{L}^*$ are two sets of sequences of elements from $\mathcal{L}$. Given $a \in L$, we use $w_1 \circ a$ as $w_1 \circ \langle a \rangle$. 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 $\langle\rangle$, 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 a 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 $T\lfloor_\Sigma = \{tr\lfloor_\Sigma \mid tr \in T\}$. Similarly, $tr \restriction^\Sigma$ is a sequence obtained by removing all elements in Σ.

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 Component Automata

In this part, we present our automata-based model describing interaction behaviors of components [10,11,9]. Invocations to provided and required services are modeled as provided and required events, respectively. Internal actions are modeled as internal events. The invocation of a provided service or an internal action will trigger invoking services provided by other components, so the label on a transition step in the formal model consists a provided or internal event and a set of sequences of required events.

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

- *S is a finite set of states, and $s_0 \in S$ is the initial state, $f \in S$ is the error state;*
- *P, R, and A are disjoint and finite sets of provided, required, and internal events, respectively;*
- *$\delta \subseteq (S \setminus \{f\}) \times \Sigma(P, R, A) \times (S \cup \{f\})$ 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 it as $s \xrightarrow{w/T} s'$ and call it a provided transition step if $w \in P$, otherwise internal transition step. We call $s \xrightarrow{a/T} f$ a failure transition. We write $s \xrightarrow{w/} s'$ for $s \xrightarrow{w/\{\epsilon\}} s'$. Component automaton C is called *closed*, if all the transitions are of form $s \xrightarrow{w/\{\epsilon\}} s'$, otherwise, *open*. The internal events are prefixed with ; to differentiate them from the provided events. τ is used to represent any internal event when it causes no confusion. For a state s we use $out(s)$ denote $\{w \in P \cup A \mid \exists s', w, T. s \xrightarrow{w/T} s'\}$

and $out^{\bullet}(s) = out(s) \cap P$ and $out^{\circ}(s) = out(s) \cap A$. We write $s \xrightarrow{w/\bullet} s'$ for $s \xrightarrow{w/T} s'$, when T is not essential. A state s is called *stable*, if $out(s) = out^{\bullet}(s)$.

Formally, we have the following definitions and notations,

- a sequence of transitions $s \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_k} s'$ is called an execution sequence, written as $s \xrightarrow{\ell_1,\ldots,\ell_k} s'$ (possible empty transition $k = 0$, in that case $s = s'$ and $s \stackrel{\epsilon}{\Longrightarrow} s$), and $\langle \ell_1, \ldots, \ell_k \rangle$ is called a *trace* from s to s' and called a *trace* of the component if s is the initial state,
- for a sequence sq over $P \cup A$, we write $s \stackrel{sq}{\Longrightarrow} s'$ if there is a trace tr such that $s \stackrel{tr}{\Longrightarrow} s'$ and $\pi_1(tr) = sq$,
- a state s' is *internally reachable* from state s, denoted by $intR(s, s')$, if there exists $s \stackrel{tr}{\Longrightarrow} s'$ such that $\pi_1(tr) \in A^*$; a set of internally reachable states from state s is denoted as $intR^{\circ}(s) = \{s' \mid intR(s, s')\}$, and $intR^{\bullet}(s) = \{s' \text{ stable} \mid s' \in intR^{\circ}(s)\}$
- for a trace tr and a state s, $target(tr, s) = \{s' \mid s \stackrel{tr}{\Longrightarrow} s'\}$, and $target(tr) = target(tr, s_0)$,
- $\mathcal{T}(s) = \{\langle \ell_1, \ldots, \ell_k \rangle \mid \exists s' \bullet s \xrightarrow{\ell_1,\ldots,\ell_k} s'\}$, and it is called the traces of s,
- $\mathcal{T}(s_0)$ is the set of traces of the component automaton C, it is also denoted by $\mathcal{T}(C)$,
- for a state s, the *provided traces* for s are given by

$$\mathcal{T}_p(s) = \{\pi_1(tr)\lfloor_P \mid tr \in \mathcal{T}(s)\},$$

- the set $\mathcal{T}_p(s_0)$ is called the set of *provided traces* of C, and it is also written as $\mathcal{T}_p(C)$.

2.3 Component Interface Automata

The model of component automata describes how a component interacts with the environment by providing and requiring services. However, some transitions or executions may be blocked due to non-determinism caused by required traces or internal events. The transitions that may lead to the error state or live lock states should also be forbidden. The non-blockable properties of provided events and provided traces will be studied in this part.

The related definitions will be given formally in the following and more details can be found in [10,11,9]. Without explicit stating, the definitions are given for component $C = (S, s_0, f, P, R, A, \delta)$.

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 via a sequence of internal transitions.

Definition 2 (divergent state).
A state s is divergent*, if there exists sq with $sq \in A^+$ such that $s \stackrel{sq}{\Longrightarrow} s$ or there exists s',$sq_1 \in A^+$ and $sq_2 \in A^+$ such that $s \stackrel{sq_1}{\Longrightarrow} s'$ and $s' \stackrel{sq_2}{\Longrightarrow} s'$.*

Definition 3 (nonrefusal provided event).
For any $s \in S$, the set of nonrefusal provided events of s is

$$\mathcal{N}(s) = \bigcap_{r \in intR^{\bullet}(s)} out^{\bullet}(r) \setminus \{a \mid s \stackrel{sq}{\Longrightarrow} f\}, sq\downarrow_P = a.$$

Definition 4 (non-blockable traces).
A sequence of provided events $\langle a_1, \cdots, a_k \rangle$ with $k \geq 0$ is non-blockable *at state s, if $a_i \in \mathcal{N}(s')$ for any $1 \leq i \leq k$ and s' such that $s \stackrel{tr}{\Longrightarrow} s'$ with $\pi_1(tr)\downarrow_P = \langle a_1, \cdots, a_{i-1} \rangle$. A sequence of pairs tr is* non-blockable *at s, if $\pi_1(tr)\downarrow_P$ is non-blockable at s.*

$\mathcal{T}_{up}(s)$ and $\mathcal{T}_u(s)$ are used to denote the set of all non-blockable provided traces and non-blockable 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.

Definition 5 (input-determinism).
A component automaton $C = (S, s_0, f, P, R, A, \delta)$ is input-deterministic *if f is not reachable from s_0 and for any $s_0 \stackrel{tr_1}{\Longrightarrow} s_1$ and $s_0 \stackrel{tr_2}{\Longrightarrow} s_2$ with $\pi_1(tr_1)\downarrow_P = \pi_1(tr_2)\downarrow_P$, implies $\mathcal{N}(s_1) = \mathcal{N}(s_2)$.*

The following theorem states that all the traces of an input-deterministic component automaton are non-blockable.

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

Hereafter, we simply call C *component interface automaton(or interface automaton)* if it is input-deterministic. The following algorithm(see in Algorithm 1), given in [11], can construct an interface automaton $\mathcal{I}(C)$ for any given component automaton C.

Theorem 2 (correctness of Algorithm 1). *The following properties hold for Algorithm 1, for any component automaton C:*

1. *The algorithm always terminates and the error state f is not reachable from the initial state,*
2. *$\mathcal{I}(C)$ is an input deterministic automaton,*
3. *$\mathcal{T}_u(C) = \mathcal{T}_u(\mathcal{I}(C))$.*

2.4 Plugging Operation

Components interact with each other by service invocation that component automata synchronize on the events that are provided by one and required by the other. The general composition operation is given in [11]. In this part, we focus on the composition between open component automata and closed component automata, called *plugging*.

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

Input: $C = (S, s_0, f, P, R, A, \delta)$
Output: $\mathcal{I}(C) = (S_I, (Q_0, s_0), f, P, R, A, \delta_I)$, where $S_I \subseteq 2^S \times S$

1: **if** $f \in intR^\circ(s_0)$ **then**
2: *exit* with $\delta_I = \emptyset$
3: **end if**
4: **Initialization:** $S_I := \{(Q_0, s_0)\}$ **with** $Q_0 = \{s' \mid s' \in intR^\circ(s_0)\}$**;** $\delta_I := \emptyset$**;** $todo := \{(Q_0, s_0)\}$**;** $done := \emptyset$
5: **while** $todo \neq \emptyset$ **do**
6: **choose** $(Q, r) \in todo$**;** $todo := todo \setminus \{(Q, r)\}$; $done := done \cup \{(Q, r)\}$
7: **for each** $a \in \bigcap_{s \in Q} \mathcal{N}(s)$ **do**
8: $Q' := \bigcup_{s \in Q} \{s' \mid s \stackrel{tr}{\Longrightarrow} s', \pi_1(tr)\downarrow_P = \langle a \rangle\}$
9: **for each** $(r \xrightarrow{a/T} r') \in \delta$ **do**
10: **if** $(Q', r') \notin (todo \cup done)$ **then**
11: $todo := todo \cup \{(Q', r')\}$
12: $S_I := S_I \cup \{(Q', r')\}$
13: **end if**
14: $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{a/T} (Q', r')\}$
15: **end for**
16: **end for**
17: **for each** $r \xrightarrow{w/T} r'$ **with** $r' \in Q$ **and** $w \in A$ **do**
18: $\delta_I := \delta_I \cup \{(Q, r) \xrightarrow{w/T} (Q, r')\}$
19: **end for**
20: **end while**

Definition 6 (plugging).
Given a component automaton $C_1 = (S_1, s_0^1, f_1, P_1, R_1, A_1, \delta_1)$ *and a closed component automaton* $C_2 = (S_2, s_0^2, f, P_2, R_2, A_2, \delta_2)$, C_2 *is pluggable to* C_1 *if* $A_1 \cap (P_2 \cup A_2) = \emptyset$, $A_2 \cap (P_1 \cup R_1) = \emptyset$, $P_2 \subseteq R_1$, *and* $R_2 = \emptyset$. *The plugging is* $C_1 \ll C_2 = (S, s_0, f, P, R, A, \delta)$ *where*

- $S = (S_1 \setminus f_1) \times (S_2 \setminus f_2) \cup \{f\}$ *where* f *is the error state of* C,
- $s_0 = (s_0^1, s_0^2)$,
- $P = P_1$,
- $R = R_1 \setminus P_2$,
- $A = A_1 \cup A_2$,
- δ *is given by the following rule: for any reachable state* (s_1, s_2), $s_1 \xrightarrow{w/T} s_1'$
  - $(s_1, s_2) \xrightarrow{w/} f$ *if* $T\downarrow_{P_2} \not\subseteq \mathcal{T}_{up}(s_2)$, *otherwise,*
  - $(s_1, s_2) \xrightarrow{w/T'} (s_1', s_2')$ *where*

$$T' = \{sq\downarrow_R \mid sq \in T, s_2 \stackrel{tr}{\Longrightarrow} s_2' \text{ and } \pi_1(tr)\downarrow_{P_2} = sq\downarrow_{P_2}\}.$$

Closed components can provide services without requiring services from the other components. These are primitive components to support the incremental development.

2.5 Refinement

Refinement is one of the key issues in component based development. It is mainly for substitution at interface level. A refinement relation by state simulation technique [25] is given. The intuitive idea is that a state s' simulates s, if at state s' more provided events are nonrefusal, less required traces are required and the next states following the transitions keep the simulation relation, which is similar to alternating simulation in [7]. We give a brief introduction of *simulation* and *refinement* and more details and proofs of the theorems can be found in [11]

Definition 7 (simulation). *A binary relation R over the set of states of a component automaton is a simulation iff whenever $s_1 R s_2$:*

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

We say that s_2 simulates s_1, written $s_1 \lesssim s_2$, if $(s_1, s_2) \in R$. C_2 refines *C_1, written $C_1 \sqsubseteq_{alt} C_2$, if there exists a simulation relation R such that $s_1^0 R s_2^0$ and $P_1 \subseteq P_2$ and $R_2 \subseteq R_1$.*

The following theorem shows that the trace inclusion properties.

Theorem 3. *Given two component interface automata C_1 and C_2, if $C_1 \sqsubseteq_{alt} C_2$, then $\mathcal{T}_{up}(C_1) \subseteq \mathcal{T}_{up}(C_2)$, and for any non-blockable provided trace $pt \in \mathcal{T}_p(C_1)$, $\mathcal{T}_r^2(pt) \subseteq \mathcal{T}_r^1(pt)$.*

The following corollaries can be obtained from Theorem 5 in [11]

Corollary 1. *Consider four component interface automata C_1, C_1', C_2, and C_2' that C_2 and C_2' are pluggable to C_1 and C_1', respectively, if $C_1 \sqsubseteq_{alt} C_1'$ and $C_2 \sqsubseteq_{alt} C_2'$, then $(C_1 \ll C_2) \sqsubseteq_{alt} (C_1' \ll C_2')$.*

3 Failure Model of Components

The automata-based component model gives the operational descriptions of components. In this section, we propose to develop a denotational description of components. The advantages of denotational models for components are easier for compatibility checking , plugging, and refinement.

In this part, we will give a semantic model of component automata based on traces and provided events that may be refused. The component automata aim at showing interaction behaviors by providing and requiring services with the environment in operational steps, while the failure model of components focus on traces of the component and the set of provided events that may be refused. The basic idea is inspired by the failure-divergence semantics in CSP [17].

Definition 8 (failures sets of component automata). *Consider component automaton $C = (S, s_0, f, P, R, A, \delta)$, a failure of C is a pair (tr, X) of a trace and a set of the events of such that there exists $s_0 \stackrel{tr}{\Longrightarrow} s$ and $X = P \setminus \mathcal{N}(s)$. We use $\mathcal{F}(C)$ to denote the set of failures of C.*

We see that provided events which may lead to the error state or **div** states are refused by the components. This is because the components we consider here aim at providing non-blockable provided services.

3.1 Failure Model of Components

Definition 9 (failure model of components). *A failure model of component is $M = (P, R, A, \mathcal{F})$*

- *P, R, and A are sets of provided, required, and internal events, respectively,*
- *$\mathcal{F} \subseteq \Sigma(P, R, A)^* \times P$ is the failure set where $\Sigma(P, R, A) = (P \cup A) \times (2^{R^*} \setminus \emptyset)$, every element $(tr, X) \in \mathcal{F}$ is called a failure. The following conditions must hold*
  - *If $(tr, X) \in \mathcal{F}$ with $X \neq P$, there exists T, $a \in P \setminus X$, and X' that $(tr \cdot tr' \cdot a/T, X') \in \mathcal{F}$, where $\pi_1(tr') \in A^*$*
  - *If $(tr \cdot a/T, X) \in \mathcal{F}$, then there exists X' that $(tr, X') \in \mathcal{F}$.*

For component automaton C, the failure model is written $[\![C]\!]_F = (P, R, A, \mathcal{F}(C))$.

Similarly, we give the definitions of traces, provided traces, and required traces of failure model M. It can be shown that the set of traces $\mathcal{T}(M)$ and the set of provided traces $\mathcal{T}_p(M)$ are given by

$$\mathcal{T}(M) = \{tr \mid \exists X \bullet (tr, X) \in \mathcal{F}\}$$
$$\mathcal{T}_p(M) = \{\pi_1(tr)\downarrow_P \mid tr \in \mathcal{T}(M)\}$$

Analogy to the traces of open components defined in Sect. 2, for each provided trace pt, there is an associated set of sequences of required events,

$$\mathcal{T}_r(pt) = \bigcup\{E_1 \cdots E_k \mid \exists tr \in \mathcal{T}(M) \bullet \pi_1(tr) = pt \wedge \pi_2(tr) = E_1 \cdots E_k\}.$$

We also give the definition of *input-determinism* and *non-blockableness* in the failure model.

Definition 10. *A failure model M is* input-deterministic *if, for any $(tr_1, X_1) \in \mathcal{F}$ and $(tr_2, X_2) \in \mathcal{F}$, $\pi_1(tr_1)\downarrow_P = \pi_1(tr_2)\downarrow_P$ implies $X_1 = X_2$.*

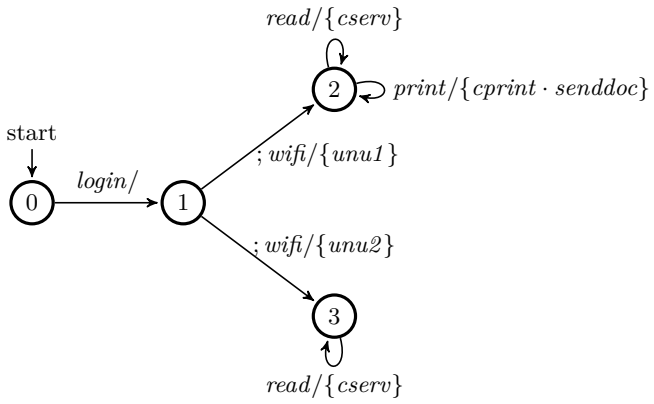


Fig. 1. Execution model of internet connection component C_{ic}

Definition 11 (non-blockable traces). *Let M be a failure model of component, provided trace $pt = \langle a_0, \cdots, a_k \rangle \in \mathcal{T}_p(M)$ is non-blockable, if there does not exist $0 \leq i < k$ and failure (tr, X) that $\pi_1(tr)\downarrow_P = \langle a_0, \cdots, a_i \rangle$ and $a_{i+1} \in X$. The set of non-blockable provided traces of M is written $\mathcal{T}_{up}(M)$*

The following theorem states that the failure model of a component automaton $[\![C]\!]_F$ is consistent with component automaton C in the above definition.

Theorem 4. *Let C be a component automaton and $[\![C]\!]_F$ is the failure model of C. So,*

- $\mathcal{T}(C) = \mathcal{T}([\![C]\!]_F)$, $\mathcal{T}_p(C) = \mathcal{T}_p([\![C]\!]_F)$, $\mathcal{T}_{up}(C) = \mathcal{T}_{up}([\![C]\!]_F)$,
- *C is input-deterministic iff $[\![C]\!]_F$ is input-deterministic.*

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 the document to print and start the printing job. The *print* service is only available for the wifi network *unu1* and *read* can be accessed at both networks.

In the component model of Fig. 1, the failures are $(\epsilon, \{read, print\})$, $(login/, \{login, print\})$, $(\langle login/, ; wifi/\{unu1\}\rangle, \{login\})$,$(\langle login/, ; wifi/\{unu2\}\rangle, \{login, print\})$

3.2 Plugging Operation

In this part, we show how two failure models of software components are composed by plugging.

Definition 12 (plugging).
Given failure models $M_1 = (P_1, R_1, A_1, \mathcal{F}_1)$ *and* $M_2 = (P_2, R_2, A_2, \mathcal{F}_2)$, M_2 *is pluggable to* M_1 *if* $A_1 \cap (P_2 \cup A_2) = \emptyset$, $A_2 \cap (P_1 \cup R_1) = \emptyset$, $P_2 \subseteq R_1$, *and* $R_2 = \emptyset$. *The plugging is* $M_1 \tilde{\ll} M_2 = (P, R, A, \mathcal{F})$ *where*

- $P = P_1$,
- $R = R_1 \setminus P_2$,
- $A = A_1 \cup A_2$,
- $(tr, X_1 \cup X_2) \in \mathcal{F}$, *if*
  - $(tr_1, X_1) \in \mathcal{F}_1$, $conc(\pi_2(tr_1))\downarrow_{P_2} \subseteq \mathcal{T}_{up}(M_2)$, *and* $tr = tr_1 \upharpoonright^{P^2}$;
  - $X_2 = \{a \mid \exists X' \bullet (tr_1 \cdot a/T, X') \in \mathcal{F}_1, conc(\pi_2(tr_1 \cdot a/T) \notin \mathcal{T}_{up}(M_2)\}$

The following theorem shows the compositional properties of the failure models.

Theorem 5. *Given component* C_1 *and* C_2 *that* C_2 *is pluggable to* C_1, *then* $[\![C_2]\!]_F$ *is pluggable to* $[\![C_1]\!]_F$ *and* $[\![C_1]\!]_F \tilde{\ll} [\![C_2]\!]_F = [\![C_1 \ll C_2]\!]_F$.

3.3 Refinement

We propose a refinement in the failure models. The general idea is that a refined model provide more non-blockable traces while require less required services, and refuse more provided services that are blockable in the provided part.

Definition 13 (failure refinement).
Given two failure models $M_1 = (P_1, R_1, A_1, \mathcal{F}_1)$ *and* $M_2 = (P_2, R_2, A_2, \mathcal{F}_2)$, M_2 *is a refinement of* M_1, *if*

- $P_1 \subseteq P_2$, $R_2 \subseteq R_1$,
- $\mathcal{T}_{up}(M_1) \subseteq \mathcal{T}_{up}(M_2)$,
- *Given* $pt \in \mathcal{T}_{up}(M_1)$, *for any* $(tr_2, X_2) \in \mathcal{F}_2$, *there exists* $(tr_1, X_1) \in \mathcal{F}_1$ *such that* $\pi_1(tr_1)\downarrow_{P_1} = \pi_1(tr_2)\downarrow_{P_2} = pt$, $conc(\pi_2(tr_2)) \subseteq conc(\pi_2(tr_1))$, *and* $X_1 \subseteq X_2$.

4 Coordination

The components we consider so far are the basic units for building software systems. In some situations, however, certain services provided by components need to be restricted due to security polices. In this section, we will introduce a kind of components, called *coordinator*, to coordinate services of components.

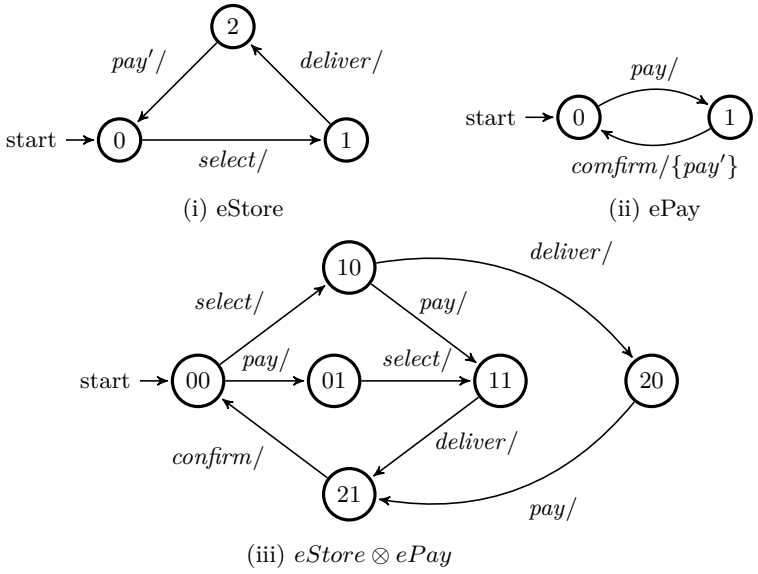


Fig. 2. online shopping system

4.1 Coordinator

We use an online-shopping system shown in Fig. 2 to motivate the need of coordinator.

Example 2. Consider an online marketplace system which provides a consumer-to-consumer platform for retail stores. It consists stores and a payment component trusted by both stores and clients. The store component, called *eStore*, presented in Fig. 2(i). It provides services *select*, *pay′*, and *deliver*, which model select items, pay the money to payment component, and deliver the paid items to the clients, respectively. The payment component, called *ePay* shown in Fig. 2(ii) provides services *pay* and *confirm* which model, receive money from the clients and being confirmed by the client that the items are received. It requires service *pay′* that the component will transfer the money to the store. The composition of *eStore* and *ePay* is in Fig. 2(iii).

In the above example, provided trace $\langle select \cdot deliver \rangle$ is allowed, which means that the store may not get paid even if it delivers the items bought by the clients. So such online marketplace system is not fair to the store retailers. We introduce a kind of components, called *coordinator*, to filter out services provided by components that should not be allowed. A coordinator is modeled as a labeled transitions system, the formal definition is given next.

Definition 14 (Coordinator). *A coordinator F is modeled as a deterministic labeled transition system (Q, q_0, E, σ) where*

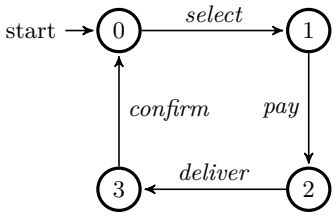


Fig. 3. Coordinator F

- Q *is the set of states with* $q_0 \in Q$ *as the initial state,*
- E *is the set of active events,*
- σ *is a set of transition.*

Similarly, the set of traces of coordinator F, written as $\mathcal{T}(F)$, is $\{\langle a_0, a_1, \cdots, a_k\rangle \mid q_0 \xrightarrow{a_0} \cdots \xrightarrow{a_k} q_{k+1}\}$.

Example 3. In order to filter out the unexpected provided traces in Fig. 2(iii), coordinator F shown in Fig. 3 is built.

4.2 Parallel Composition of Coordinators

Now we define the parallel composition of coordinators. Since coordinators only show the sequences of services that are allowed, two coordinators do not communicate directly. Two coordinators are composable, if they do not have active events in common. Thus, the parallel composition of two coordinators is simply the interleaving execution of the actions of the individual coordinators.

Definition 15 (Parallel composition of coordinators). *Given two coordinators* $F_1 = (Q_1, q_0^1, E_1, \delta_1,)$ *and* $F_1 = (Q_2, q_0^2, E_2, \delta_2,)$*, if* $E_1 \cap E_2 = \emptyset$*, the parallel composition* $F_1 \parallel F_2$ *is* (Q, q_0, E, δ) *where*

- $Q = Q_1 \times Q_2$ *and* $q_0 = (q_0^1, q_0^2)$*,*
- $E = E_1 \cup E_2$*,*
- δ *is given by the rule:* $(q_1, q_2) \xrightarrow{a} (q_1', q_2') \in \delta$ *if either*
  - $q_1 = q_1'$ *and* $q_2 \xrightarrow{a} q_2'$ *is a transition of* F_2*,*
  - $q_2 = q_2'$ *and* $q_1 \xrightarrow{a} q_1'$ *is a transition of* F_1

The following theorem shows that the traces of $F_1 \parallel F_2$ can be obtained from the traces of F_1 and F_2.

Theorem 6. *Given two composable coordinators* F_1 *and* F_2*,* $\mathcal{T}(F_1 \parallel F_2) = \{sq \in E^* \mid sq\downarrow_{E_1} \in \mathcal{T}(F_1), sq\downarrow_{E_2} \in \mathcal{T}(F_2)\}$.

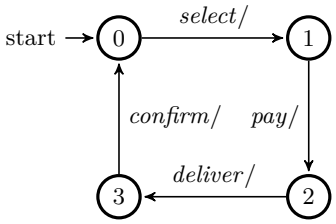


Fig. 4. Coordination of $(eStore \otimes ePay) \ltimes F$

4.3 Coordination Operation

Components are coordinated in the way that all the sequences of services provided should also obey the constraint of the coordinators. The formal definition is given below.

Definition 16 (Coordination).
Given a component automaton $C = (S, s_0, P, R, A, \delta)$ *and a coordinator* $F = (Q, q_0, E, \sigma)$ *such that* $E \subseteq P$*, the coordination of* C *by* F *is:*
$C \ltimes F = (S', s'_0, P', R', \delta')$ *where*

- $S' = S \times Q$,
- $s'_0 = (s_0, q_0)$,
- $P' = P$
- $R' = R$,
- δ' *is a set of transitions by the rule that, the reachable state is* (s, q)
  - $s \xrightarrow{w/T} s'$ *and* $t \xrightarrow{w} t'$*, then* $(s,t) \xrightarrow{w/T} (s', t')$.
  - $s \xrightarrow{w/T} s'$ *with* $w \notin E$*, then* $(s,t) \xrightarrow{w/T} (s', t)$.

Theorem 7. *The failures of* $C \ltimes F$ *is* $\mathcal{F}(C \ltimes F) = \{(tr, D \cup D') \mid (tr, D) \in \mathcal{F}(C), \pi_1(tr)\downarrow_E \in \mathcal{T}(F), D' = \{d \mid \pi_1(tr)\downarrow_E \cdot d \notin \mathcal{T}(F)\}\}$

Example 4. Now, we can see how the component $eStore \otimes ePay$ shown in Fig. 2(iii) is coordinated by coordinator in Fig. 3. The result is presented in Fig. 4.

4.4 Synthesis of A Interface Coordinator for Component Automata

In this part, we will show that given any component automaton C, there exits a coordinator F that coordination of C by F is equivalent with the interface model of C.

We now present a procedure in Algorithm 2 that, given a component automaton, constructs a coordinator which only records non-blockable provided traces. The basic idea is similar to the construction of a deterministic automaton from a non-deterministic one, and the only difference is that in the algorithm only the deterministic traces are kept.

Three key correctness properties of the algorithm are stated in the following theorem.

Algorithm 2: Construction of Interface coordinator

Input: $C = (S, s_0, P, R, A, \delta)$
Output: $\mathcal{G}(C) = (Q, q_0, E, \sigma)$
1: **Initialization:** $q_0 = \{s' \mid s' \in intR^{\circ}(s_0)\}, Q := \{q_0\}, E := P, \sigma := \emptyset$, $todo := \{q_0\}$
2: **while** $todo \neq \emptyset$ **do**
3: **choose one** $q \in todo$ **and** $todo := todo \setminus \{q\}$
4: **for each** $a \in \bigcap_{s \in q} \mathcal{N}(s)$ **do**
5: **let** q' **be** $\{s' \mid s \stackrel{tr}{\Longrightarrow} s',$ **with** $s \in q, \pi_1(tr)\downarrow_P = a\}$
6: **if** $q' \notin Q$ **then**
7: **add** q' **to** Q **and** $todo$
8: **end if**
9: $\sigma := \sigma \cup \{q \stackrel{a}{\rightarrow} q'\}$.
10: **end for**
11: **end while**

Theorem 8 (Correctness of Algorithm 2). *Given any component automaton C, the following properties hold for Algorithm 2.*

- *The algorithm always terminates.*
- *$\mathcal{G}(C)$ is deterministic.*
- *$\mathcal{T}(\mathcal{G}(C)) = \mathcal{T}_{up}(C)$.*

Proof. The termination of the algorithm is obtained, because *todo* will eventually be empty: the size of power set of state S is bounded, only fresh state is added to *todo*, and for each iteration of the loop a state from *todo* is removed.

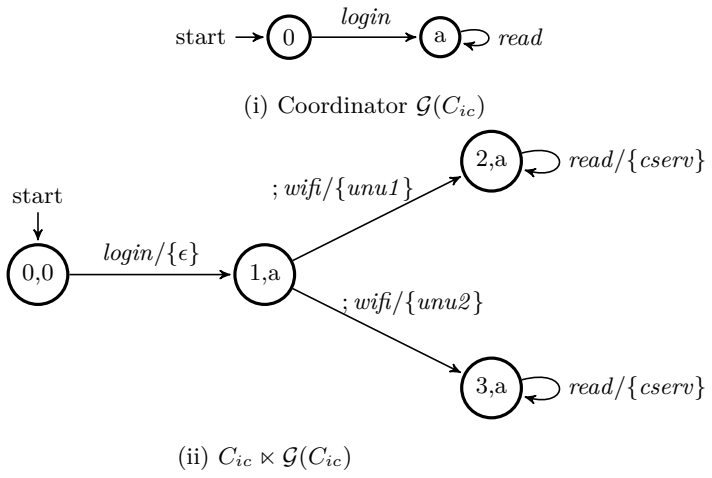
Assume that there exists $q \stackrel{a}{\rightarrow} q_1$ and $q \stackrel{a}{\rightarrow} q_1$, then from Algorithm 2, we $q_1 = q_2 = \{s' \mid s \stackrel{tr}{\Longrightarrow} s',$ with $s \in q, \pi_1(tr)\downarrow_P = a\}$. So $\mathcal{G}(C)$ is deterministic.

We first prove that, for any non-blockable provided trace pt of C, there exists $q_0 \stackrel{pt}{\Longrightarrow} q$ in $\mathcal{G}(C)$ with $q = \{s' \mid s_0 \stackrel{tr}{\Longrightarrow} s', \pi_1(tr)\downarrow_P = pt\}$ by induction on the length of pt. The base case is obvious that $q_0 = \{s' \mid s' \in intR^{\circ}(s_0)\}$. Consider non-blockable trace $sq \cdot a$, so there exists q_1 that $q_1 = \{s' \mid s_0 \stackrel{tr}{\Longrightarrow} s', \pi_1(tr)\downarrow_P = sq\}$ and $q_0 \stackrel{sq}{\Longrightarrow} q_1$. Since $sq \cdot a$ is non-blockable, $a \in \mathcal{N}(s')$ for $s' \in q_1$. From Loop(Line 4-10) in Algorithm 2, we see $q_1' = \{s' \mid s \stackrel{tr}{\Longrightarrow} s',$ with $s \in q_1, \pi_1(tr)\downarrow_P = a\}$,so $q_0 \stackrel{sq \cdot a}{\Longrightarrow} q_1'$ and $q_1' = \{s' \mid s_0 \stackrel{tr}{\Longrightarrow} s', \pi_1(tr)\downarrow_P = sq \cdot a\}$ by hypothesis induction. From above, we see $\mathcal{T}_{up}(C) \subseteq \mathcal{T}(\mathcal{G}(C))$.

Next we prove that sq is non-blockable in C, for any $q_0 \stackrel{sq}{\Longrightarrow} q$, and $q = \{s' \mid s_0 \stackrel{tr}{\Longrightarrow} s', \pi_1(tr)\downarrow_P = sq\}$. The base case follows that ϵ is non-blockable in C. Consider $q_0 \stackrel{sq' \cdot a}{\Longrightarrow} q_2$, then, there exists $q_0 \stackrel{sq'}{\Longrightarrow} q_1$ and $q_1 \stackrel{a}{\rightarrow} q_2$. By hypothesis induction, $q_1 = \{s' \mid s_0 \stackrel{tr}{\Longrightarrow} s', \pi_1(tr)\downarrow_P = sq'\}$ and sq' is non-blockable. $sq' \cdot a$ is non-blockable, since $a \in \mathcal{N}(r)$ for any $r \in q_1$. And $q_2 = \{s' \mid s_0 \stackrel{tr}{\Longrightarrow} s', \pi_1(tr)\downarrow_P = sq' \cdot a\}$. From above, we see $\mathcal{T}(\mathcal{G}(C)) \subseteq \mathcal{T}_{up}(C)$.

So, $\mathcal{T}(\mathcal{G}(C)) = \mathcal{T}_{up}(C)$. □

We can obtain the following corollary from Theorem 7 and Theorem 8.



(i) Coordinator $\mathcal{G}(C_{ic})$

(ii) $C_{ic} \ltimes \mathcal{G}(C_{ic})$

Fig. 5. Coordination of Component C_{ic} by a synthesized coordinator

Corollary 2. *Given a component automaton C, $\mathcal{T}_u(C) = \mathcal{T}(C \ltimes \mathcal{G}(C))$*

Example 5. The component automaton in Fig. 1 is not input-deterministic. A coordinator shown in Fig. 5(i) is obtained by Algorithm 2. We use state a as shorthand for $\{1, 2, 3\}$. The coordination of $C_{ic} \ltimes \mathcal{G}(C)$ is given in Fig. 5(ii)

5 Conclusion and Future Work

We gave a brief view of our previous work on automata-based interface models of software components. We proposed a denotational semantic model, called failure model and showed that traces, non-blockablness, input-determinism, plugging operations can also be defined on the failure models and proved to be consistent with those in component automata.

Future work. There are several open problems left for future work. Firstly, refinement relation based on failure models and the relation with the refinement defined on component automata need further studied. Secondly, more general composition operation instead of plugging needs to be given for failure models. Thirdly, algebraic properties of composition such as associative, commutative, distributive of coordination over composition are also important. The fourth 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.

Acknowledgments. This work has been supported by the Projects GAVES and PEARL funded by Macau Science and Technology Development Fund and

grants from the Natural Science Foundation of China NSFC-61103013 and NSFC-91118007. We thank Zhiming liu for his inspiring comments and discussions. We also thank the anonymous reviewers for the feedback.

References

1. de Alfaro, L., Henzinger, T.: Interface theories for component-based design. In: Henzinger, T., Kirsch, C. (eds.) Embedded Software, LNCS, vol. 2211, pp. 148–165. Springer (2001)
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 329–366 (June 2004)
3. Arbab, F., Baier, C., Rutten, J., Sirjani, M.: Modeling component connectors in reo by constraint automata: (extended abstract). Electronic Notes in Theoretical Computer Science 97(0), 25 – 46 (2004)
4. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. 31(5), 19:1–19:61 (Jul 2009)
5. Chen, X., Liu, Z., Mencl, V.: Separation of concerns and consistent integration in requirements modelling. In: SOFSEM (1). pp. 819–831 (2007)
6. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model-driven design. Science of Computer Programming 74(4), 168 – 196 (2009), special Issue on the Grand Challenge
7. De Alfaro, L., Henzinger, T.: Interface automata. ACM SIGSOFT Software Engineering Notes 26(5), 109–120 (2001)
8. De Alfaro, L., Henzinger, T.: Interface-based design. Engineering Theories of Software-intensive Systems 195, 83–104 (2005)
9. Dong, R., Faber, J., Ke, W., Liu, Z.: rcos: Defining meanings of component-based software architectures. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Unifying Theories of Programming and Formal Engineering Methods, Lecture Notes in Computer Science, vol. 8050, pp. 1–66. Springer Berlin Heidelberg (2013)
10. Dong, R., Faber, J., Liu, Z., Srba, J., Zhan, N., Zhu, J.: Unblockable compositions of software components. In: Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering. pp. 103–108. CBSE '12, ACM, New York, NY, USA (2012)
11. Dong, R., Zhan, N., Zhao, L.: An interface model of software components. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theoretical Aspects of Computing Ð ICTAC 2013. Lecture Notes in Computer Science, vol. 8049, pp. 159–176. Springer Berlin Heidelberg (2013)
12. Emmi, M., Giannakopoulou, D., Pasareanu, C.S.: Assume-guarantee verification for interface automata. In: Cuéllar, J., Maibaum, T.S.E., Sere, K. (eds.) FM. LNCS, vol. 5014, pp. 116–131. Springer (2008)
13. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: ASE. pp. 3–12. IEEE Computer Society (2002)
14. He, J., Li, X., Liu, Z.: Component-based software engineering. In: ICTAC. pp. 70–95 (2005)
15. He, J., Li, X., Liu, Z.: rcos: A refinement calculus of object systems. Theor. Comput. Sci. 365(1-2), 109–142 (2006)
16. He, J., Li, X., Liu, Z.: A theory of reactive components. Electr. Notes Theor. Comput. Sci. 160, 173–195 (2006)

17. Hoare, C.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
18. Larsen, K.G., Nyman, U., Wasowski, A.: Interface input/output automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM. LNCS, vol. 4085, pp. 82–97. Springer (2006)
19. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: Nicola, R.D. (ed.) ESOP. LNCS, vol. 4421, pp. 64–79. Springer (2007)
20. Liu, Z., Morisset, C., Stolz, V.: rCOS: Theory and tool for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) FSEN. LNCS, vol. 5961, pp. 62–80. Springer (2009)
21. Lüttgen, G., Vogler, W.: Modal interface automata. In: Baeten, J., Ball, T., Boer, F. (eds.) Theoretical Computer Science, LNCS, vol. 7604, pp. 265–279. Springer Berlin Heidelberg (2012)
22. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC. pp. 137–151 (1987)
23. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
24. Mcilroy, D.: Mass-produced Software Components. In: Buxton, J.M., Naur, P., Randell, B. (eds.) Proceedings of Software Engineering Concepts and Techniques. pp. 138–155. NATO Science Committee (Jan 1969)
25. Milner, R.: Communication and concurrency. Prentice Hall International (UK) Ltd., Hertfordshire, UK (1995)
26. Raclet, J., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: Modal interfaces: unifying interface automata and modal specifications. In: Proceedings of the seventh ACM international conference on Embedded software. pp. 87–96. ACM (2009)
27. Raclet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: A modal interface theory for component-based design. Fundam. Inf. 108(1-2), 119–149 (Jan 2011)
28. Roscoe, A.: The Theory and Practice of Concurrency. Prentice Hall (1998)
29. Sifakis, J.: A framework for component-based construction. In: Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on. pp. 293–299. IEEE (2005)
30. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1997)
31. Zhan, N., Kang, E.Y., Liu, Z.: Component publications and compositions. In: UTP. pp. 238–257 (2008)