Front. Comput. Sci., 2015, 9(1): 1–14
DOI 10.1007/s11704-014-4016-7

RESEARCH ARTICLE

# Generalized interface automata with multicast synchronization

**Fei HE**[1,2,3], **Xiaoyu SONG**[4], **Ming GU**[1,2,3], **Jiaguang SUN**[1,2,3]

1 Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing 100084, China
2 Key Laboratory for Information System Security, Ministry of Education, Beijing 100084, China
3 School of Software, Tsinghua University, Beijing 100084, China
4 Department of Electrical & Computer Engineering, Portland State University, Oregon 97207, USA



**Abstract** Interface automata are one of the prominent formalisms for specifying interface behaviors of component-based systems. However, only one-to-one communication is allowed in the composition of interface automata. This paper presents multicast interface automata which generalize the classic interface automata and accommodate multicast communication mechanism. The multicast interface automata endorse both bottom-up and top-down design methodologies. Theoretical results on compatibility and refinement are established for incremental design and independent implementability.

**Keywords** interface automata, multicast communication, component interaction, verification

## 1 Introduction

Interface automata [1,2] are a model for specifying component-based systems. They are defined based on labeled transition systems and syntactically similar to input/output automata [3]. The interface automaton model is considered as a light-weight formalism for specifying temporal aspects of software component interfaces.

The essential feature of interface automata against traditional models lies on the fact that they can model both the input requirements and the output behaviors of a system. Two components are compatible if there exists an environment such that their input assumptions can be simultaneously satisfied. This feature distinguishes the interface automata from many traditional models, such as input/output automata [3]. In these formalisms, the models are required to be input-enabled, thus being unable to block inputs from the environment. The compatibility checking requires that the components work correctly in all environments. Since some inputs can be blocked, this formalism usually leads to a simpler and more abstract model.

In the theory of interface automata [1,2], only two interfaces are allowed to be composed at one time. As a result, only one-to-one communication between two components can be directly modeled in interface automata. Such limitation hinders the application of interface automata to complex distributed systems. For instance, consider a communication system shown in Fig. 1(a), the component $P_0$ publishes a message $e$ simultaneously to components $P_1$, $P_2$, and $P_3$. This is a typical multicast communication which occurs in distributed systems. To model such a system with interface automata, we have to use appropriate relabeling shown in Fig. 1(b). After relabeling actions, the message $e$ is relabeled into three messages, $e_1$, $e_2$, and $e_3$. As a result, the publish action is divided into three one-to-one actions. The resulting model in interface automata may cause state space expansion with auxiliary relabeling. A more direct and convenient modeling is desirable.

In this paper, we present a new theory of *multicast interface automata*. It extends the classical interface automata to model multicast synchronism and communication. For

Received January 13, 2014; accepted June 10, 2014

E-mail: hefei@tsinghua.edu.cn

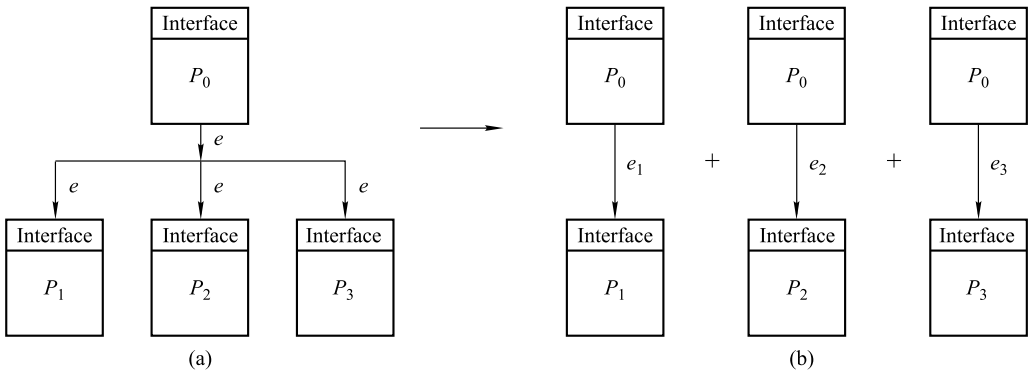


**Fig. 1** An example of using interface automata to model multicast communication

instance, the communication system shown in Fig. 1 can be modeled directly with our automata. Each component (including $P_0$, $P_1$, $P_2$, and $P_3$) is modeled as an automaton, and the message passing from $P_0$ simultaneously to $P_1$, $P_2$, and $P_3$ is modeled as a one-to-many communication directly. The proposed model avoids using relabeling actions to specify multicast communication and thus is more suitable for specifying complex distributed systems.

The proposed automata are syntactically based on the classical interface automata. Each system component is modeled as a multicast interface automaton, which is essentially a labeled transition system. The actions of an automaton are classified as input actions, output actions and internal actions. The interface of a component is composed of input and output actions. A collection of automata can be composed to yield another interface automaton. For composition, the concurrent automata synchronize on input and output actions, while they interleave asynchronously on internal actions. As multicast interface automata are not required to be input enabled, we need to distinguish the concepts of composability and compatibility of a collection of automata. A collection of automata is compatible if there exists an environment such that they can work together properly.

We demonstrate that the proposed automata theory supports both incremental design and independent implementability. These properties guarantee our formalism supports both bottom-up and top-down system design process. In a bottom-up design process, the compatibility test of two or more interfaces can be performed, even before all components are assembled in the final design. In a top-down design process, an individual component can be implemented independently, as long as the implementations conform to the predefined interfaces.

The paper is organized as follows. In Section 2, we address related work. In Section 3, we introduce multicast interface automata. In Section 4, we define the refinement relation between a collection of interface automata. We show several fundamental theorems in multicast interface language theory in Section 5. Finally, Section 6 concludes the paper.

## 2 Related work

I/O automata [3,4] are defined based on labeled transition systems, and suitable for modeling distributed and concurrent systems. I/O automata can be used not only for system modeling, but also for algorithm correctness proving and complexity analysis [5].

From its inception, a multitude of work has been done on extending and improving I/O automata. In [6,7], a timed I/O automaton model is proposed for modeling real-time systems. In [8–10], a probabilistic I/O automaton model is studied. It extends I/O automata with the ability to describing probabilities. Moreover, in [11], Lynch et al. present a hybrid I/O automaton model, which supports modeling of both discrete and continuous behaviors of hybrid systems.

The interface automata were first proposed in [1]. The formalism of interface automata is syntactically similar to I/O automata, and is suitable for capturing the temporal aspects of component interfaces. Theories of interface and component algebra are presented in [12]. The interface algebra addresses compositional refinement, while the component algebra underpins compositional abstraction. In [12], the authors proved that the interface automaton model is an instance of interface algebra and the I/O automaton model is an instance of component algebra. In [2], the interface-based design methodology was suggested for developing component-based systems. To support interface-based design, the interface language needs to meet two requirements, i.e., incremental design and independent implementability. The interface automaton model was proved to be an instance of an interface language. In [13], a method for transforming interface automaton to I/O automaton is presented. In the original the-

ory of interface automata [1,2], each automaton is required to be input-deterministic. This restriction has been relaxed in [14] by slightly revising the definition of parallel composition of interface automata.

Various extensions of interface automata are proposed in [15–18]. In [17], a timed interface automaton model is presented, which is suitable for specifying real time systems. The timed interface supports specifying timing constraints of both input assumptions and their output behaviors. In [18], the resource interface automaton model is investigated. In this model, it is possible to check if a collection of components work well together with limited resources. The sociable interface model is proposed in [19]. It fuels a wide spectrum of communication paradigms.

## 3 Multicast interface automata

**Definition 1** An interface automaton [1] is a tuple

$$P = \langle V_P, v_P^0, A_P^I, A_P^O, A_P^H, \triangle_P \rangle,$$

where

- $V_P$ is a finite set of states;
- $v_P^0 \in V_P$ is an initial state;
- $A_P^I, A_P^O$ and $A_P^H$ are pairwise disjoint sets of input, output and internal actions, respectively. Let the set of actions $A_P = A_P^I \cup A_P^O \cup A_P^H$;
- $\triangle_P \subseteq V_P \times A_P \times V_P$ is a set of transitions.

To distinguish the traditional interface automaton which allows one-to-one communication only, we call the interface automaton which participates in a multicast communication the *multicast interface automaton*.

The automaton $P$ is *closed* if it has only internal actions, i.e., $A_P^I = A_P^O = \emptyset$, otherwise it is *open*. We denote $A_P^X = A_P^I \cup A_P^O$ as the set of *external actions*, and $A_P^L = A_P^O \cup A_P^H$ as the set of *local actions*. We call a transition $(v, a, v')$ the *input* (resp. *output*, *internal*) transition if $a$ is an input (resp. output, internal) action. The set of all input (resp. output, internal) transitions is denoted by $\triangle_P^I$ (resp. $\triangle_P^O$, $\triangle_P^H$).

An action $a$ is *enabled* at a state $v$ if there exists another state $v' \in V_P$ such that $(v, a, v') \in \triangle_P$. We denote $A_P^I(v), A_P^O(v), A_P^H(v)$ the set of input actions, output actions and internal actions enabled at state $v$, respectively. Like interface automata, we do not require multicast automata to be *input enabled*, i.e., we do not require that $A_P^I(v) = A_P^I$ hold at all states.

Given $v_1, v_2 \in V_P$, $a \in A_P$, $a_x \in A_P^X$, and two sequences of actions $\alpha = a_1 a_2 \cdots a_n \in (A_P)^n$, $\beta = b_1 b_2 \cdots b_n \in (A_P^X)^n$ with $n \geqslant 1$, we define operators "$\longrightarrow$" and "$\Longrightarrow$" as follows:

- $v_1 \xrightarrow{a}_P v_2$ iff $(v_1, a, v_2) \in \triangle_P$;
- $v_1 \xrightarrow{\tau}_P v_2$ iff there exists an action $b \in A_P^H$, such that $v_1 \xrightarrow{b}_P v_2$;
- $v_1 \xrightarrow{\alpha}_P v_2$ iff $v_1 \xrightarrow{a_1}_P \xrightarrow{a_2}_P \cdots \xrightarrow{a_n}_P v_2$;
- $v_1 \overset{\epsilon}{\Longrightarrow}_P v_2$ iff $v_1 (\xrightarrow{\tau}_P)^* v_2$;
- $v_1 \overset{a_x}{\Longrightarrow}_P v_2$ iff $v_1 \overset{\epsilon}{\Longrightarrow}_P \xrightarrow{a_x}_P v_2$;
- $v_1 \overset{\beta}{\Longrightarrow}_P v_2$ iff $v_1 \overset{b_1}{\Longrightarrow}_P \overset{b_2}{\Longrightarrow}_P \cdots \overset{b_n}{\Longrightarrow}_P v_2$.

where $(\tau)^*$ returns the reflexive and transitive closure of $\tau$. For simplicity, we usually ignore the subscripts of relation operators.

Given two states $v_1, v_2$ of automaton $P$, we say $v_2$ is *reachable* from $v_1$ if there exists a sequence of action $\alpha \in (A_P)^*$ such that $v_1 \xrightarrow{\alpha} v_2$; especially if the sequence $\alpha$ consists of only internal actions, i.e., $\alpha \in (A_P^H)^*$, we say $v_2$ is *invisibly reachable* from $v_1$; and if $\alpha$ consists of only local actions, i.e., $\alpha \in (A_P^H \cup A_P^O)^*$, we say $v_2$ is *autonomously reachable* from $v_1$. We say a state $v_2$ is (invisibly, autonomously) reachable in automaton $P$ if $v_2$ is (invisibly, autonomously) reachable from the initial state $v_P^0$ of $P$.

We illustrate our definitions by modeling a software component which provides job processing service. Component *Treater* has a method *job* for the user to assign jobs. When this method is called, the component returns either *done* or *fail*. To provide this service, the component needs to consume some necessary resources such as CPU and memory. When a job is given, the component first outputs an action *alloc* for requesting resources, then it expects to receive allocated results from the resource management components. The multicast interface automaton modeling this component is drawn in Fig. 2. The automaton is surrounded by a rounded corner rectangle, whose ports correspond to the interface. With an automaton, we denote the input (resp. output, internal) action by appending a symbol of "?" (resp. "!", ";"),

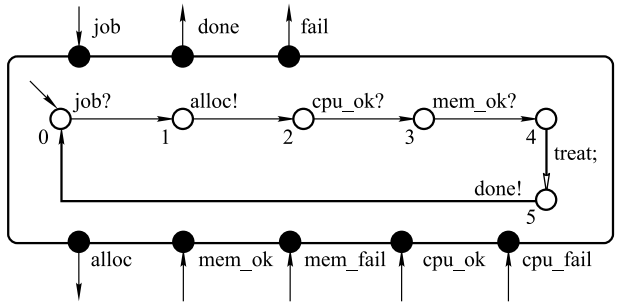


**Fig. 2** Multicast interface automata *Treater*

where circular node represents compatible state, and square node represents error state.

As mentioned before, multicast interface automata need not to be input enabled. There are input actions of $Treater$ which are not accepted at some states. For example, the input action *job* is only accepted at state $s_0$, which reveals an assumption made by $Treater$ about its environment: after the user calls the *job* method, he will wait without calling this method again until he gets the method result.

In the automaton $Treater$, it makes no provisions for handling resource allocation failure. In other words, $Treater$ accepts only success results of resource allocation, and does not accept failure results. This reveals another assumption made by $Treater$ about its environment that all resources are always allocated successfully.

Consider two resource components: $CPU$ and *Memory*. They are in charge of allocation of computation resource and storage resource, respectively. The *alloc* request is simultaneously transmitted to these two components. With the assumption that system has unlimited memory resource and limited computation resource, we draw multicast interface automata shown in Figs. 3 and 4 for these two components, respectively. When component $CPU$ receives the *alloc* request, it calls the method *get_cpu* to get the current usage status of CPU. If CPU is busy, it returns $cpu\_fail$; otherwise $cpu\_ok$.

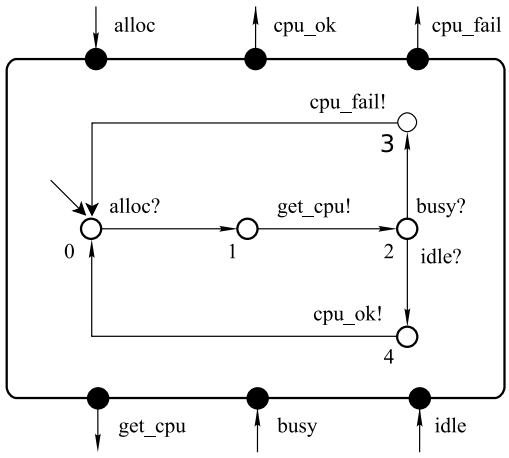


**Fig. 3** Multicast interface automata $CPU$

The automaton of a complex system can be constructed by composing automata modeling each of its components. We now define composition of multicast interface automata. Let $I \subseteq \mathbb{N}$ be a finite and nonempty set of natural numbers.

**Definition 2** A collection $\{P_i\}_{i\in I}$ of multicast interface automata is composable if, for any $i, j \in I, i \neq j$,

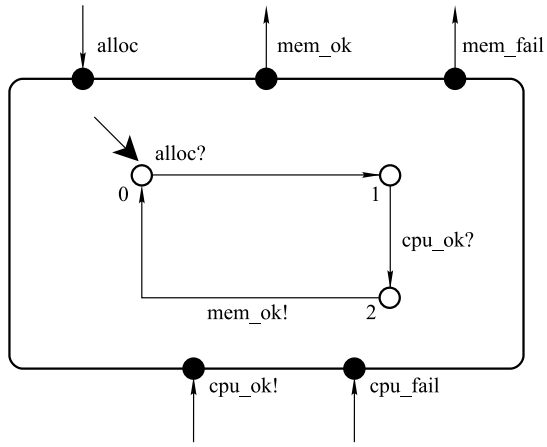


**Fig. 4** Multicast interface automata *Memory*

$$A^O_{P_i} \cap A^O_{P_j} = \emptyset,$$
$$A^H_{P_i} \cap A_{P_j} = \emptyset.$$

Note that the definition of composability is different from that of interface automata. In interface automata, only two automata are allowed to be composed at one time. Here we allow composition of any finite number of automata. As we do not require that the input actions of different automata be disjoint, components with the same input actions are composable. Recalling the job processing example, the collection of automata $Treater$, $CPU$, and *Memory* are composable.

We first define the product of multicast interface automata. As in classical automata theory, the participating automata synchronize on shared actions, and interleave asynchronously on other actions. We then define the composition of automata which is obtained from the product by eliminating incompatible states.

**Definition 3** Given a composable collection $\{P_i\}_{i\in I}$ of multicast interface automata, their product automaton $\mathcal{P} = \prod_{i\in I} P_i$ is defined as follows: [1)]

- $V_{\mathcal{P}} = \prod_{i\in I} V_{P_i}$;
- $v^0_{\mathcal{P}}$ is a vector, where $v^0_{\mathcal{P}}[i] = v^0_{P_i}$;
- $A^I_{\mathcal{P}} = \bigcup_{i\in I} A^I_{P_i} \setminus \bigcup_{i\in I} A^O_{P_i}$;
- $A^O_{\mathcal{P}} = \bigcup_{i\in I} A^O_{P_i}$;
- $A^H_{\mathcal{P}} = \bigcup_{i\in I} A^H_{P_i}$;
- $\triangle_{\mathcal{P}} \subseteq V_{\mathcal{P}} \times A_{\mathcal{P}} \times V_{\mathcal{P}}$ is a set of triples $(v, a, v')$ such that, for all $i \in I$, if $a \in A_{P_i}$ then $(v[i], a, v'[i]) \in \triangle_{P_i}$, and if $a \notin A_{P_i}$, then $v'[i] = v[i]$.

For simplicity, we denote $\prod_{i\in I} P_i$ as $\prod_I P_i$. Given a state $v$

[1)] For simplicity, we use the symbol "×" to denote both product between automata and Cartesian product between sets

of $\prod_I P_i$, it is considered as a vector with the $i$-th element $v[i]$ being the value in $P_i$. Let $v|i$ be the image of $v$ on $P_i$. Moreover, given a subset $J \subseteq I$, let $v|J$ be the image of $v$ on $\prod_J P_j$, which is obtained by deleting $v[k]$ from vector $v$ if $k \notin J$.

The above definition shows the main difference between our automata and interface automata. Here we are allowed to have a product of any finite number of automata and the shared actions are not merged into internal actions immediately. The treatment on actions is similar as that in I/O automata. When computing the product of a collection of automata, the shared input actions are removed, but the shared output actions are kept. The motivation behind this decision lies on the fact that there may be many participants in a multicast communication. It is necessary to keep shared output actions to make the product automaton composable to other components involved in the same communication.

We prove that the product operation of a collection of multicast interface automata is associative.

**Theorem 1** Given a composable collection $\{P_i\}_{i\in I}$ of multicast interface automata. Let $I = I_1 \cup I_2$ where $I_1 \cap I_2 = \emptyset$, then

$$\prod_I P_i = \prod_{I_1} P_i \times \prod_{I_2} P_i.$$

**Proof** According to Definition 1, to prove equivalence of two multicast interface automata, we need to prove that the values of $V$, $v^0$, $A^I$, $A^O$, $A^H$ and $\triangle$ are equal, respectively.

Recalling the job processing example, the product of automata $Treater$ and $CPU$ [2)] is shown in Fig. 5. The shared actions between $Treater$ and $CPU$ are $alloc$, $cpu\_ok$ and $cpu\_fail$. Each state in the product consists of a state of $Treater$ and a state of $CPU$. Each transition is either a synchronization step between $Treater$ and $CPU$, such as the transition from state $(1,0)$ to state $(2,1)$, and the transition from state $(2,4)$ to state $(3,0)$; or an interleaving step between $Treater$ and $CPU$, such as all other transitions. Note that only reachable states are plotted in Fig. 5.

Since multicast interface automata do not need to be input enabled, when computing the product of a collection of automata, one of the automata may output an action that is an input of another automaton, but is not accepted. An *error state* is a state where such a situation happens. In Fig. 5, state $(2,3)$ is an error state, where the component $CPU$ can output an action $cpu\_fail$ while the other component $Treater$ cannot accept it.

**Definition 4** Given a composable collection $\{P_i\}_{i\in I}$ of multicast interface automata, a state $v \in V_{\prod_I P_i}$ is an *error state* if there exist $i, j \in I$ such that

$$(i \neq j) \wedge \left(\exists a \in shared(P_i, P_j).a \in A^O_{P_i}(v[i]) \wedge a \notin A^I_{P_j}(v[j])\right),$$

where $shared(P_i, P_j) = A_{P_i} \cap A_{P_j}$.

For I/O automata, if the product automaton contains any reachable error state, then the participated automata are not compatible. The situation is different in interface automata. In interface automata, we know that the participated automata are not compatible only when there is no environment such that all error states are unreachable in the product automaton. In Fig. 5, state $(2,3)$ is an error state. However, there are some environments where state $(2,3)$ can become unreachable. For example, the component $Scheduler$ shown in Fig. 6 frames one of such environments.

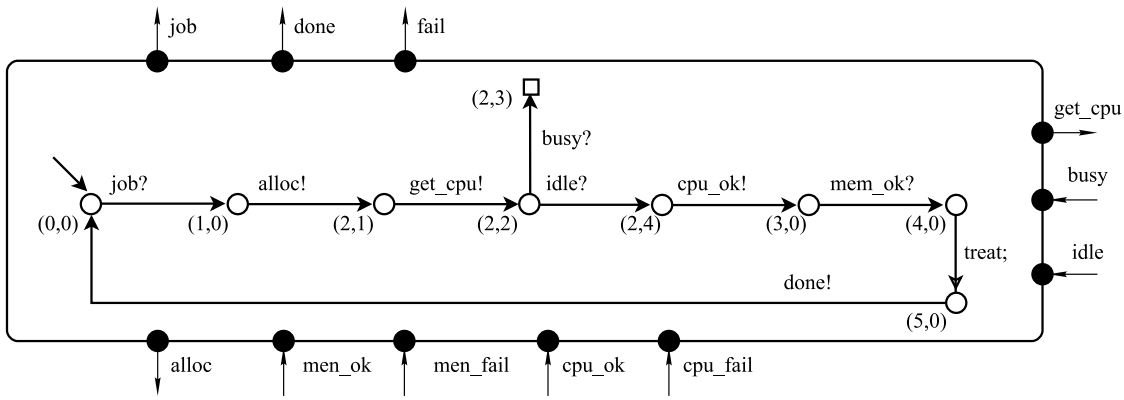


**Fig. 5** Product $Treater \times CPU$

[2)] Just for illustration purpose, we show the product of two automata here. With the definition, we can also compute the product $Treater \times CPU \times Memory$.

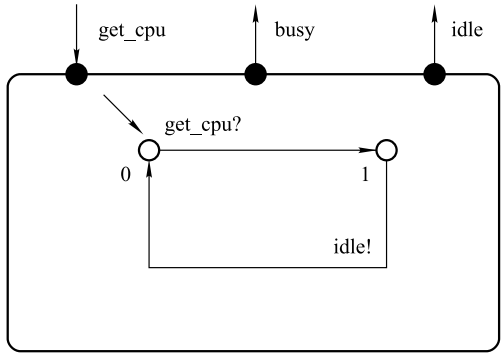


**Fig. 6** Multicast interface automata *Scheduler*

Note that the environment can only affect the component by synchronizing with its input actions. If an error state is autonomously reachable from a state $s$, there is no way for the environment to prevent the automaton from going from $s$ to the error state. Therefore, we have the following definition for compatible states.

**Definition 5** Given a composable collection $\{P_i\}_{i\in I}$ of multicast interface automata, a state $v \in V_{\prod_I P_i}$ is a *compatible state* of $\prod_I P_i$ if there exists no error state $e \in error(\prod_I P_i)$ such that $e$ is autonomously reachable from $v$.

Let $Cmp(\prod_I P_i)$ be the set of all compatible states in $\prod_I P_i$. Given a compatible state $v$, all states that are autonomously reachable from this state must also be compatible states.

**Definition 6** A collection $\{P_i\}_{i\in I}$ of multicast interface automata is *compatible* if they are composable and the initial state of $\prod_I P_i$ is compatible.

To define the composition of a collection of automata, we introduce the concept of dangerous transitions [14]. The composition is obtained by eliminating all dangerous transitions in the product automaton.

**Definition 7** Given a composable collection $\{P_i\}_{i\in I}$ of multicast interface automata, a transition $t = (v, a, v') \in \triangle_{\prod_I P_i}$ is *dangerous* if the following conditions hold:

1) $v$ is a compatible state of $\prod_I P_i$,
2) $a \in A^I_{\prod_I P_i}$ is an input action of $\triangle_{\prod_I P_i}$,
3) $v'$ is not a compatible state of $\prod_I P_i$.

Let $\triangle^D_{\prod_I P_i}$ be the set of all the dangerous transition in $\triangle_{\prod_I P_i}$.

**Definition 8** Given a composable collection $\{P_i\}_{i\in I}$ of multicast interface automata, their composition $\|_{i\in I} P_i$ (for short, $\|_I P_i$) is defined as:

- $V_{\|_I P_i} = V_{\prod_I P_i}, v^0_{\|_I P_i} = v^0_{\prod_I P_i}$;
- $A^I_{\|_I P_i} = A^I_{\prod_I P_i}, A^O_{\|_I P_i} = A^O_{\prod_I P_i}, A^H_{\|_I P_i} = A^H_{\prod_I P_i}$;
- $\triangle_{\|_I P_i} = \triangle_{\prod_I P_i} \backslash \triangle^D_{\prod_I P_i}$.

When all components participated in certain communication have been composed, we need to explicitly "hide" the involved output actions by converting them to internal actions.

**Definition 9** Given a multicast interface automaton $P$ and a set of actions $\Sigma(\subseteq A^O_P)$, the hidden automaton of $P$ with $\Sigma$ is defined as follows:

$$P\backslash\Sigma = \langle V_P, v^0_P, A^I_P, A^O_P \backslash \Sigma, A^H_P \cup \Sigma, \triangle_P \rangle.$$

Note that there is no "hide" operation in classical interface automata where the shared actions of involved automata are immediately merged to an internal action.

Figure shows the composition $Treater\|CPU$ obtained by eliminating all dangerous transitions in product $Treater \times CPU$. Note that the error state $(2, 3)$ becomes unreachable in the composition. Moreover, since the actions $cpu\_ok$ and $cpu\_fail$ are only related to the components $Treater$ and $CPU$, we change them to internal actions using the "hide" operation. Figure 8 shows an implementation of the $User$ component. Figure 9 gives the composition of all the automata in the system which is closed after applying the hide operation.

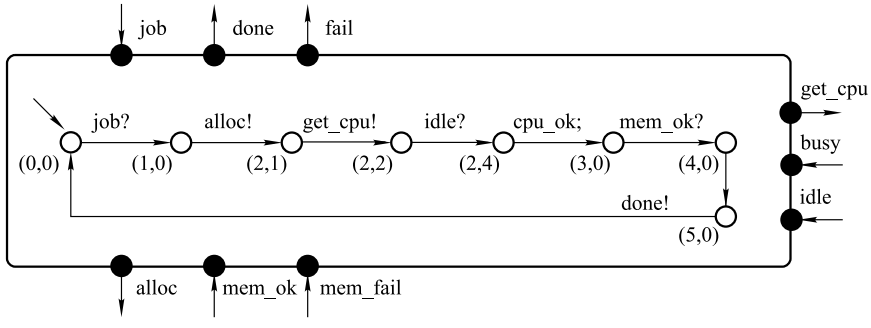


**Fig. 7** Composition $(Treater\|CPU)\backslash\{cpu\_ok, cpu\_fail\}$

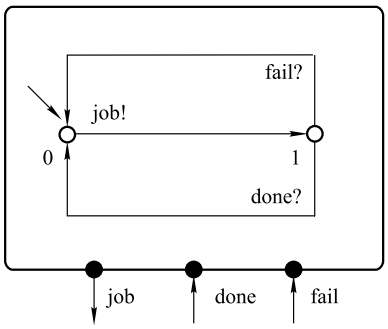


**Fig. 8** Multicast interface automata *User*

## 4 Refinement relation

The refinement relation provides a way to check consistency between different abstraction levels of a component. Similar to the classical interface automata, we define a refinement relation of multicast interface automata based on alternating simulation [20]. Automaton $P$ refines $Q$ if all input steps of $Q$ can be simulated by $P$, and all output steps of $P$ can be simulated by $Q$. To give a formal definition, we need the following concepts.

**Definition 10** Given a state $v$ of multicast interface automaton $P$, the $\epsilon$-*closure* of $v$ (written $\epsilon$-$closure_P(v)$) is the set of states which are invisibly reachable from $v$.

Assume that $v$ is the current state of $P$. Since the internal action of $P$ is invisible, the environment only knows automaton $P$ stays at a state in the set $\epsilon$-$closure(v)$, but does not know exactly at which state automaton $P$ stays. Thus, when the automaton $P$ interacts with its environment, we need to consider situations where $P$ can be at any state in $\epsilon$-$closure(v)$.

**Definition 11** Given a state $v$ of automaton $P$,

$$ExtEn_P^O(v) = \{a|\exists u \in \epsilon\text{-}closure_P(v).a \in A_P^O(u)\},$$
$$ExtEn_P^I(v) = \{a|\forall u \in \epsilon\text{-}closure_P(v).a \in A_P^I(u)\}.$$

Let $ExtEn_P^I(v)$ ($ExtEn_P^O(v)$) be the set of *externally enabled* input (output) actions of $P$ at state $v$.

We use $ExtEn_P^X(v) = ExtEn_P^I(v) \cup ExtEn_P^O(v)$ as the set of externally enabled actions of $P$ at state $v$.

**Definition 12** Given a state $v$ of automaton $P$, for any $a \in ExtEn_P^X(v)$, the set of *externally enabled* destination states of $P$ is defined as

$$ExtDest_P(v, a) = \{u'|\exists(u, a, u') \in \triangle_P.\ u \in \epsilon\text{-}closure_P(v)\}.$$

**Definition 13** Let $P$ and $Q$ be two multicast interface automata. A binary relation $\preceq$ ($\subseteq V_P \times V_Q$) is an alternating simulation from $P$ to $Q$, if for any state pair $(u, v) \in \preceq$, the following conditions hold:

1) $ExtEn_P^I(u) \supseteq ExtEn_Q^I(v)$, $ExtEn_P^O(u) \subseteq ExtEn_Q^O(v)$,
2) For any action $a \in ExtEn_P^O(u) \cup ExtEn_Q^I(v)$, and any state $u' \in ExtDest_P(u, a)$, there is a state $v' \in ExtDest_Q(v, a)$ such that $u' \preceq v'$.

Consider condition 1 in the above definition. An input action of an automaton can be viewed as a service provided by this component. The environment can call this service by synchronizing with the corresponding input action. On the other hand, an output action of an automaton can be seen as a service calling to the environment, and the environment should provide some service to satisfy this call.

**Definition 14** Given two multicast interface automata $P$ and $Q$, $P$ refines $Q$ (written $P \preceq Q$) if

1) $A_P^I \supseteq A_Q^I$, $A_P^O \subseteq A_Q^O$,
2) There is an alternating simulation $\preceq$ from $P$ to $Q$ such that $q_P^0 \preceq q_Q^0$.

It is easy to understand the condition 1 in the above definition. In component-based design, a component $A$ refines

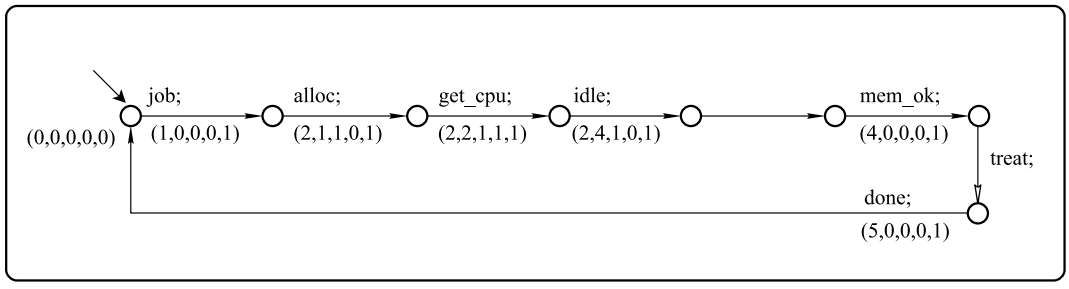


**Fig. 9** Composition $(Treater||CPU||Memory||Scheduler||User)\backslash\Sigma$, where $\Sigma$ is the set of all output actions

a component $B$ only when $A$ provides more services to the environment, and requires less invocation from the environment.

**Theorem 2** The refinement relation between multicast interface automata is a preorder relation.

**Proof** The refinement relation is obviously reflexive. We need only to prove that it is also transitive, i.e., given any automata $P$, $Q$ and $R$, if $P \preceq Q, Q \preceq R$, then $P \preceq R$ holds.

1) According to Definition 14, we have $A_P^I \supseteq A_Q^I \supseteq A_R^I$, $A_P^O \subseteq A_Q^O \subseteq A_R^O$;
2) From the premise, there exists an alternating simulation $\preceq_1$ from $P$ to $Q$ such that $v_P^0 \preceq_1 v_Q^0$, and an alternating simulation $\preceq_2$ from $Q$ to $R$ such that $v_Q^0 \preceq_2 v_R^0$. We construct a binary relation $\preceq_3$ from $P$ to $R$ as follows:

$$\preceq_3 = \{(u, w) \in V_P \times V_R | \exists v \in V_Q, (u, v) \in \preceq_1 \wedge (v, w) \in \preceq_2\}$$

Obviously, $v_P^0 \preceq_3 v_R^0$ holds. Given any $(u, w) \in \preceq_3$, there exists $v \in V_Q$, such that $u \preceq_1 v$ and $v \preceq_2 w$. We then need to prove $\preceq_3$ is an alternating simulation.

- From $u \preceq_1 v$ and $v \preceq_2 w$, we get

$$ExtEn_P^I(u) \supseteq ExtEn_Q^I(v) \supseteq ExtEn_R^I(w),$$
$$ExtEn_P^O(u) \subseteq ExtEn_Q^O(v) \subseteq ExtEn_R^O(w).$$

- Given any action $a \in ExtEn_P^O(u) \cup ExtEn_R^I(w)$, and any state $u' \in ExtDest_P(u, a)$, from $ExtEn_R^I(w) \subseteq ExtEn_Q^I(v)$, we know

$$a \in ExtEn_P^O(u) \cup ExtEn_Q^I(v).$$

Since $u \preceq_1 v$, there is a state $v' \in ExtDest_Q(v, a)$ such that $u' \preceq_1 v'$. Similarly, from $ExtEn_P^O(u) \subseteq ExtEn_Q^O(v)$, we get

$$a \in ExtEn_Q^O(v) \cup ExtEn_R^I(w).$$

Moreover, since $v \preceq_2 w$, and $v' \in ExtDest_Q(v, a)$, there is a state $w' \in ExtDest_R(w, a)$ such that $v' \preceq_2 w'$. Thus $u' \preceq_3 w'$ holds.

## 5 Interface language theory

An interface language should support incremental design and independent implementability [2]. We show that our multicast interface automata meet these two requirements.

The property of incremental design requires that the compatibility of the interfaces be not changed when an interface is added in subsequent design phases. More precisely, if a collection of interfaces are compatible, then any subset of the collection are also compatible. We demonstrate that multicast interface automata adhere to incremental design.

**Theorem 3** Let $\{P_i\}_{i \in I}$ be a composable collection of multicast interface automata. If $J \subseteq I$, then $\{P_i\}_{i \in J}$ are composable.

**Proof** Let $i, j \in J$, and $i \neq j$. Note that $J \subseteq I$, thus $i, j \in I$. Since $\{P_i\}_{i \in I}$ are composable, we have: $A_{P_i}^O \cap A_{P_j}^O = \emptyset$, and $A_{P_i}^H \cap A_{P_j} = \emptyset$. According to the definition of composability, $\{P_i\}_{i \in J}$ are also composable.

**Theorem 4** Let $\{P_i\}_{i \in I}$ be a compatible collection of multicast interface automata. If $J \subseteq I$, then $\left\{P_j\right\}_{j \in J}$ are compatible.

**Proof** Since $\left\{P_j\right\}_{j \in J}$ is composable (according to Theorem 3), we only need to prove the following proposition: if $v_{\prod_I P_i}^0$ is compatible, then $v_{\prod_J P_j}^0$ is also compatible, which can be proved by contradiction.

Assume the above proposition does not hold, then there exists an error state $e$ in $\prod_J P_j$ such that $e$ is autonomously reachable from $v_{\prod_J P_j}^0$. Let

$$v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} v_n$$

be a path from $v_{\prod_J P_j}^0$ to $e$, where $v_0 = v_{\prod_J P_j}^0$, $v_n = e$ and $v_{k-1} \xrightarrow{a_{k-1}} v_k \in \triangle_{\prod_J P_j}$ for $1 \leqslant k \leqslant n$.

Let $v_0' = v_{\prod_I P_i}^0$, then $v^0 = v_0'|J$ holds. Assume for the subpath from $v_0$ to $v_k$ ($k \leqslant n$), there exists a corresponding path $v_0' \xrightarrow{a_0} v_1' \xrightarrow{a_1} \cdots \xrightarrow{a_{k-1}} v_k'$ in $\prod_I P_i$, such that $v_l = v_l'|J$, for $0 \leqslant l \leqslant k$. Now consider the transition $(v_k, a_k, v_{k+1})$ in $\prod_J P_j$, there are two cases of $a_k$:

- $a_k$ is an internal action of $\prod_J P_j$. Let $v_{k+1}'[i] = v_{k+1}[i]$ if $i \in J$, and $v_{k+1}'[i] = v_k'[i]$ if $i \notin J$, then $(v_k', a_k, v_{k+1}')$ must be a transition in $\prod_I P_i$.
- $a_k$ is an output action of $\prod_J P_j$. For any automaton $P_i$ ($i \notin J$) which takes $a_k$ as its input action, $a_k$ must be enabled on its state $v_k'[i]$, otherwise according to Definition 4, $v_k'$ is an error state. Let $v_{k+1}'[i] = v_{k+1}[i]$ if $i \in J$, and equal to one of the successors of $v_k'[i]$ on $a_k$ if $i \notin J$ and $a_j \in A_{P_i}^I$, and equal to $v_k'[i]$, otherwise, then $(v_k', a_k, v_{k+1}')$ must be a transition in $\prod_I P_i$.

In both cases, either we prove $v_k'$ is an error state, or we prove $(v_k', a_k, v_{k+1}')$ is a transition in $\prod_I P_i$. If $v_k'$ is an error state, we stop here and report the path $v_0' \xrightarrow{a_0} v_1' \xrightarrow{a_1} \cdots \xrightarrow{a_{k-1}} v_k'$. Otherwise, we continue to consider the transition

$(v_{k+1}, a_k, v_{k+2})$. In the worst case, until $k = n$ we get

$$v'_0 \xrightarrow{a_0} v'_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} v'_n,$$

which is a path in $\prod_I P_i$ satisfying $v_k = v'_k|J$, for $0 \leqslant k \leqslant n$. By Definition 4, $v'_n$ is also an error state in $\prod_I P_i$. Therefore, no matter what cases, we prove that there is an error state $v'_k$ which is reachable from the initial state in $\prod_I P_i$.

According to the definition of composability, the output or internal actions remain to be output or internal actions after being composited. So the action $a_l$ is still an output or internal action of $v'_l$ for $(0 \leqslant l < k)$. Then the error state $v'_k$ is autonomously reachable from $v^0_{\prod_I P_i}$, and thus $v^0_{\prod_I P_i}$ is not a compatible state. This conflicts with the premise. So the assumption is not true, and the proposition holds.

Independent implementability is supported if the individual components can be implemented independently, as long as the implementations conform to the predefined interfaces. More precisely, given three multicast interface automata $P$, $P'$ and $Q$ with certain interacting relation, if $P'$ and $Q$ are compatible and $P \preceq P'$, then $P$ and $Q$ are also compatible and $P\|Q \preceq P'\|Q$.

To verify our multicast interface automata supports independent implementability, we need to prove several lemmas.

**Lemma 1** Let $P$, $P'$, $Q$ be three multicast interface automata such that $P'$ and $Q$ are composable, $P \preceq P'$. If $shared(P, Q) \subseteq shared(P', Q)$ and $A^H_P \cap A^O_{P'} = \emptyset$ hold, then $P$ and $Q$ are also composable.

**Lemma 2** Let $P$, $P'$, $Q$ be three multicast interface automata such that $P'$ and $Q$ are composable, $P \preceq P'$, $A^H_P \cap A^O_{P'} = \emptyset$, and $shared(P, Q) \subseteq shared(P', Q)$. Let $\preceq$ be an alternating simulation from $P$ to $P'$, and $(s_{P'}, s_Q) \in Cmp(P', Q)$ be a compatible state in $P' \times Q$. Given a state $s_p \in V_P$ suth that $s_P \preceq s_{P'}$, we have:

1) If there is a sequence of actions $\alpha \in \left(A^O_{P\times Q}\right)^*$ and a state $(s'_P, s'_Q) \in V_{P\times Q}$ such that $(s_P, s_Q) \overset{\alpha}{\Longrightarrow}_{P\times Q} \left(s'_P, s'_Q\right)$, then there must be a state $s''_P \in V_P$ and $s'_{P'} \in V_{P'}$ such that $s'_P \in \epsilon\text{-}closure(s''_P)$, $s''_P \preceq s'_{P'}$, $(s_{P'}, s_Q) \overset{\alpha_n}{\Longrightarrow}_{P'\times Q} (s'_{P'}, s'_Q)$ and $(s'_{P'}, s'_Q) \in Cmp(P', Q)$.
2) $(s_P, s_Q) \in Cmp(P, Q)$.

**Lemma 3** Let $P$, $P'$, and $Q$ be three multicast interface automata such that $P$ and $Q$ are compatible, $P'$ and $Q$ are compatible, and $P \preceq P'$. Given states $s_P \in V_P$, $s_{P'} \in V_{P'}$ such that $s_P \preceq s_{P'}$, then, for any $s_Q \in V_Q$, the following relations hold:

$$ExtEn^I_{P\times Q}((s_P, s_Q)) \supseteq ExtEn^I_{P'\times Q}((s_{P'}, s_Q)),$$

$$ExtEn^O_{P\times Q}((s_P, s_Q)) \subseteq ExtEn^O_{P'\times Q}((s_{P'}, s_Q)).$$

The following theorem and corollaries show that our multicast interface automata endorse independent implementability.

**Theorem 5** Let $P$, $P'$, and $Q$ be three multicast interface automata such that $shared(P, Q) \subseteq shared(P', Q)$ and $A^H_P \cap A^O_{P'} = \emptyset$. If $P'$ and $Q$ are compatible and $P \preceq P'$, then $P$ and $Q$ are compatible and $P \parallel Q \preceq P' \parallel Q$.

**Proof** We first prove the compatibility of $P$ and $Q$. Based on Lemma 1, $P$ and $Q$ are composable. From the compatibility of $P'$ and $Q$, we get $(v^0_{P'}, v^0_Q) \in Cmp(P', Q)$. From $P \preceq P'$, we know $v^0_P \preceq v^0_{P'}$. According to Lemma 2, we conclude $(v^0_P, v^0_Q) \in Cmp(P, Q)$. Thus $P$ and $Q$ are compatible.

We then prove $P \parallel Q \preceq P' \parallel Q$ holds.

1) From $P \preceq P'$, there are $A^I_P \supseteq A^I_{P'}, A^O_P \subseteq A^O_{P'}$. Then $A^I_P \cup A^I_Q \supseteq A^I_{P'} \cup A^I_Q$ and $A^O_P \cup A^O_Q \subseteq A^O_{P'} \cup A^O_Q$. Thus
$$A^I_{P\|Q} \supseteq A^I_{P'\|Q}, A^O_{P\|Q} \subseteq A^O_{P'\|Q}.$$
2) Construct a binary relation from $V_{P\times Q}$ to $Cmp(P', Q)$ as
$$\preceq' = \{((s_P, s_Q), (s_{P'}, s_Q)) \in V_{P\times Q} \times Cmp(P', Q) | \exists s^*_P \in V_P, s_P \in \epsilon\text{-}closure(s^*_P) \wedge s^*_P \preceq s_{P'}\}.$$
Then we can prove $\preceq'$ is an alternating simulation.
3) Let $v^0_P$, $v^0_Q$, and $v^0_{P'}$ be initial states of $P$, $Q$, and $P'$, respectively. Since $P$ and $Q$, and $P'$ and $Q$ are compatible, there is
$$\left((v^0_P, v^0_Q), (v^0_{P'}, v^0_Q)\right) \in Cmp(P, Q) \times Cmp(P', Q).$$
Since $P \preceq P'$, then $v^0_P \preceq v^0_{P'}$. Therefore,
$$(v^0_P, v^0_Q) \preceq' (v^0_{P'}, v^0_Q).$$

**Corollary 1** Let $P$, $P'$, $Q$, and $Q'$ be four multicast interface automata such that
$$shared(P, Q) \subseteq shared(P', Q) \subseteq shared(P', Q').$$
If $P'$ and $Q'$ are compatible, $P \preceq P'$, $A^H_P \cap A^O_{P'} = \emptyset$, $Q \preceq Q'$, $A^H_Q \cap A^O_{Q'} = \emptyset$, then $P$ and $Q$ are also compatible, and $P \parallel Q \preceq P' \parallel Q'$.

**Proof** With the premises, we know that $Q \preceq Q'$, $Q'$ and $P'$ are compatible, and $shared(P', Q) \subseteq shared(P', Q')$, $A^H_Q \cap A^O_{Q'} = \emptyset$. Then according to Theorem 5, we conclude that $Q$ and $P'$ are also compatible, and $P'\|Q \preceq P'\|Q'$ holds. Since $P \preceq P'$, $shared(P, Q) \subseteq shared(P', Q)$, $A^H_P \cap A^O_{P'} = \emptyset$,

by applying theorem 5 again, we get that $P$ and $Q$ are compatible too, and $P\|Q \preceq P'\|Q$ holds. According to the transitivity of alternating simulation relation, $P\|Q \preceq P'\|Q'$ holds.

Given a collection $\{P_i\}_{i\in I}$ of multicast interface automata, we denote the set of shared actions as

$$shared(\{P_i\}_{i\in I}) = \bigcup_{i,j\in N, i\neq j} (A_{P_i} \cap A_{P_j}).$$

For instance, in our our job processing example, we have:

$$shared(Treater, CPU, Memory) = \{alloc, cpu\_ok, cpu\_fail, mem\_ok, mem\_fail\}.$$

**Corollary 2** Let $P_1, P_2, \ldots, P_n$ and $P'_1, P'_2, \ldots, P'_n$ be two collections of multicast interface automata such that

$$shared(P_1, P_2, \ldots, P_n) \subseteq shared(P'_1, P_2, \ldots, P_n) \subseteq shared(P'_1, P'_2, P_3, \ldots, P_n) \subseteq \cdots \subseteq shared(P'_1, P'_2, \ldots, P'_n).$$

If $P'_1, P'_2, \ldots, P'_n$ are compatible, $P_i \preceq P'_i$ and $A^H_{P_i} \cap A^O_{P'_i} = \emptyset$ hold for $1 \leqslant i \leqslant n$, then $P_1, P_2, \ldots, P_n$ are also compatible and $\|^n_{i=1} P_i \preceq \|^n_{i=1} P'_i$.

**Proof** This corollary can be proved in a similar way as for corollary 1.

Consider the job processing example in Fig. 2. The design of *Treater* requires action *cpu_ok* being taken before the action *mem_ok*. This constraint would be unreasonable in some circumstances. To solve the problem, we need to refine the original design. For example, the automaton *BetterTreater* in Fig. 10 gives a refinement of the *Treater* component.

Before inserting *BetterTreater* into the system, we need to prove its compatibility with the system. We define a relation:

$$\preceq = \{(0', 0), (1', 1), (2', 2), (3', 3), (4', 4), (5', 5)\}.$$

It is easy to prove that this relation is an alternating simulation from *BetterTreater* to *Treater*. Then by Theorem 5, we can prove the compatibility of *BetterTreater*.

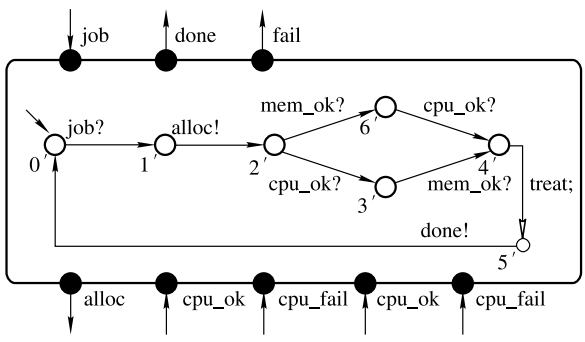


**Fig. 10** Multicast interface automata *BetterTreater*

## 6 Conclusion

We presented multicast interface automata in this paper. The formalism extended the classical interface automata by introducing multicast communication, and therefore is more suitable for specifying complex distributed systems. The proposed model endorses both incremental design and independent implementability methodologies and can be used in both bottom-up and top-down design process.

**Acknowledgements** This work was supported by the Chinese National 973 Plan (2010CB328003), the National Natural Science Foundation of China (Grant Nos. 61272001, 60903030, 91218302), the Chinese National Key Technology R&D Program (SQ2012BAJY4052), the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions (YETP0167), and the Tsinghua University Initiative Scientific Research Program.

## Appendix

### A.1 Proof to Theorem 1

**Proof** According to Definition 1, to prove equivalence of two multicast interface automata, we need to prove that the values of $V$, $v^0$, $A^I$, $A^O$, $A^H$ and $\triangle$ are equal, respectively.

For simplicity, we use *left* and *right* to denote the left and right parts of the above formula respectively.

$$V_{\text{right}} = V_{\prod_{I_1} P_i} \times V_{\prod_{I_2} P_i} = \prod_{I_1} V_{P_i} \times \prod_{I_2} V_{P_i} = \prod_{I} V_{P_i} = V_{\text{left}}.$$

$$v^0_{\text{right}} = v^0_{\prod_{I_1} P_i} \times v^0_{\prod_{I_2} P_i} = \prod_{I_1} v^0_{P_i} \times \prod_{I_2} v^0_{P_i} = \prod_{I} v^0_{P_i} = v^0_{\text{left}}.$$

$$A^O_{\text{right}} = A^O_{\prod_{I_1} P_i} \cup A^O_{\prod_{I_2} P_i} = \bigcup_{I_1} A^O_{P_i} \cup \bigcup_{I_2} A^O_{P_i} = \bigcup_{I} A^O_{P_i} = A^O_{\text{left}}.$$

$$A^H_{\text{right}} = A^H_{\prod_{I_1} P_i} \cup A^H_{\prod_{I_2} P_i} = \bigcup_{I_1} A^H_{P_i} \cup \bigcup_{I_2} A^H_{P_i} = \bigcup_{I} A^H_{P_i} = A^H_{\text{left}}.$$

$$\begin{aligned} A^I_{\text{right}} &= \left(A^I_{\prod_{I_1} P_i} \cup A^I_{\prod_{I_2} P_i}\right) \setminus \left(A^O_{\prod_{I_1} P_i} \cup A^O_{\prod_{I_2} P_i}\right) \\ &= \left(\bigcup_{I_1} A^I_{P_i} \setminus \bigcup_{I_1} A^O_{P_i}\right) \cup \left(\bigcup_{I_2} A^I_{P_i} \setminus \bigcup_{I_2} A^O_{P_i}\right) \\ &\quad \setminus \left(\bigcup_{I_1} A^O_{P_i} \cup \bigcup_{I_2} A^O_{P_i}\right) \\ &= \left(\bigcup_{I} A^I_{P_i} \setminus \bigcup_{I} A^O_{P_i}\right) \\ &= A^I_{\text{left}}. \end{aligned}$$

As to the set of transitions $\triangle$, we need to prove that for any transition $\tau$, if $\tau \in \triangle_{\text{left}}$, then $\tau \in \triangle_{\text{right}}$, and vice versa.

1) Given $\tau = \langle v, a, v' \rangle \in \triangle_{\text{left}}$. According to Definition 3,

for all $i \in I$, if $a \in A_{P_i}$ then $(v[i], a, v'[i]) \in \triangle_{P_i}$; and if $a \notin A_{P_i}$ then $v'[i] = v[i]$. Since $I_1 \subseteq I$ and $I_2 \subseteq I$, we have

$$\langle v|I_1, a, v'|I_1\rangle \in \triangle_{\prod_{I_1} P_i}, \text{ or } v|I_1 = v'|I_1.$$

$$\langle v|I_2, a, v'|I_2\rangle \in \triangle_{\prod_{I_2} P_i}, \text{ or } v|I_2 = v'|I_2.$$

Then we conclude $\langle v, a, v'\rangle \in \triangle_{\text{right}}$.

2) Given $\tau' = \langle u, b, u'\rangle \in \triangle_{\text{right}}$. Since $u$ and $u'$ are states in $\prod_{I_1} P_i \times \prod_{I_2} P_i$, and $I_1 \cup I_2 = I$, $u$ and $u'$ are also states in $\prod_I P_i$. No matter if $b$ is an action in $\prod_{I_1} P_i$ or in $\prod_{I_2} P_i$, since $I_1 \subseteq I$ and $I_2 \subseteq I$, $b$ must be an action in $\prod_I P_i$. Thus, $\langle u, b, u'\rangle \in \triangle_{\text{left}}$.

### A.2 Proof to Lemma 1

**Proof** Since $P'$ and $Q$ are composable, then

$$shared(P', Q) = (A^I_{P'} \cap A^O_Q) \cup (A^O_{P'} \cap A^I_Q) \cup (A^I_{P'} \cap A^I_Q).$$

Since $A_Q = A^H_Q \cup A^X_Q$, we have

$$shared(P, Q) = (A_P \cap A^H_Q) \cup (A_P \cap A^X_Q).$$

Note that there is no element of $A^H_Q$ in $shared(P', Q)$. By applying this fact to $shared(P, Q) \subseteq shared(P', Q)$, we get

$$A_P \cap A^H_Q = \emptyset. \tag{1}$$

Note

$$shared(P, Q) = (A_P \cap A^X_Q) = (A^H_P \cap A^X_Q) \cup (A^X_P \cap A^X_Q).$$

We now prove $A^H_P \cap A^X_Q = \emptyset$ by contradiction. Assume $A^H_P \cap A^X_Q \neq \emptyset$. There exists an action $a \in A^H_P$, and $a \in A^X_Q$. Obviously $a \in shared(P, Q)$ holds. Note $shared(P, Q) \subseteq shared(P', Q)$, then $a \in shared(P', Q)$. Consider following cases of $a$.

1) $a \in A^O_Q$: In a similar way, with $a \in A^O_Q$, we get $a \notin A^I_Q$. Applying this to Eq. (2), we get $a \in (A^I_{P'} \cap A^O_Q)$, then $a \in A^I_{P'}$. Since $a \in A^H_P$, so we have $A^I_{P'} \cup A^H_P \neq \emptyset$. As $P \preceq P'$, $A^I_P \supseteq A^I_{P'}$, we obtain $A^I_P \cup A^H_P \neq \emptyset$. Here comes contradiction.
2) $a \in A^I_Q$: Since $a \in A^I_Q$, then $a \notin A^O_Q$. Applying this to Eq. (2), we have $a \in (A^O_{P'} \cap A^I_Q) \cup (A^I_{P'} \cap A^I_Q)$, then $a \in A^O_{P'} \cup A^I_{P'}$, i.e., $a \in A^X_{P'}$. Note $a \in A^H_P$, so $A^X_{P'} \cap A^H_P \neq \emptyset$. This conflicts with the premise.

In both cases, the assumption conflicts with the premise, so the assumption does not hold, that is

$$A^H_P \cap A^X_Q = \emptyset.$$

Comparing this formula with equation Eq. (1), we get

$$A^H_P \cap A_Q = \emptyset. \tag{2}$$

Note

$$shared(P, Q) = A^X_P \cap A^X_Q = (A^I_P \cap A^I_Q) \cup (A^I_P \cap A^O_Q) \cup (A^O_P \cap A^I_Q) \cup (A^O_P \cap A^O_Q).$$

By considering the elements in $shared(P, Q)$ and $shared(P', Q)$ which belong to $A^O_Q$, we have

$$(A^I_P \cap A^O_Q) \cup (A^O_P \cap A^O_Q) \subseteq A^I_{P'} \cap A^O_Q.$$

Since $A^I_P \supseteq A^I_{P'}$, we have $A^I_P \cap A^O_Q \supseteq A^I_{P'} \cap A^O_Q$, and

$$A^O_P \cap A^O_Q = \emptyset, \quad A^I_P \cap A^O_Q = A^I_{P'} \cap A^O_Q. \tag{3}$$

By considering Eqs. (1), (2), and (3) together, we conclude that $P$ and $Q$ are composable.

### A.3 Proof to Lemma 1

**Proof** Based on Lemma 1, $P$ and $Q$ are composable. We now prove these two propositions as follows.

1) Let $|\alpha| = n$, we prove the proposition 1 by induction on the length of $\alpha$.
   a) The base: the proposition holds when $n = 0$, i.e., if $(s'_P, s'_Q) \in \epsilon\text{-}closure((s_P, s_Q))$, then there must be a state $s''_P \in V_P$ and $s'_{P'} \in V_{P'}$ such that $s'_P \in \epsilon\text{-}closure(s''_P)$, $s''_P \preceq s'_{P'}$, $(s'_{P'}, s'_Q) \in \epsilon\text{-}closure((s_{P'}, s_Q))$ and $(s'_{P'}, s'_Q) \in Cmp(P', Q)$. Since $(s'_P, s'_Q) \in \epsilon\text{-}closure((s_P, s_Q))$, then
   $$s'_P \in \epsilon\text{-}closure(s_P), \text{ and } s'_Q \in \epsilon\text{-}closure(s_Q).$$
   Let $s''_P \in \epsilon\text{-}closure(s'_P)$ and $s'_{P'} \in \epsilon\text{-}closure(s_{P'})$, then obviously the proposition holds.
   b) The inductive step: if the proposition holds for $n$, the proposition also holds for $n + 1$. Let $\alpha = a_1, a_2, \ldots, a_{n+1}$, where $a_i \in A^O_{P\times Q}$ for $1 \leqslant i \leqslant n + 1$, and $\alpha_n = a_1, a_2, \ldots, a_n$ be the $n$th prefix of $\alpha$. Assume $(s_P, s_Q) \overset{\alpha_n}{\Longrightarrow}_{P\times Q} (s^n_P, s^n_Q)$, there must exist a sequence of states $s^0_P, s^1_P, \ldots, s^n_P \in V_P$, such that
   $$s^0_P = s_P,$$
   $$s^i_P \in ExtDest_P(s^{i-1}_P, a_i), \text{if } a_i \in ExtEn^X_P(s^{i-1}_P);$$
   $$\text{and } s^i_P = s^{i-1}_P, \text{otherwise.}$$

There also exists a sequence of states $s_Q^0, s_Q^1, \ldots, s_Q^n \in V_Q$, such that

$$s_Q^0 = s_Q,$$
$$s_Q^i \in ExtDest_Q(s_Q^{i-1}, a_i), \text{if } a_i \in ExtEn_Q^X(s_Q^{i-1});$$
$$\text{and } s_Q^i = s_Q^{i-1}, \text{otherwise.}$$

According to the premise, we know there exists a state $s_P^* \in V_P$ and $s_{P'}^n \in V_{P'}$ such that $s_P^n \in \epsilon\text{-}closure(s_P^*)$, $s_P^* \preceq s_{P'}^n$, $(s_P, s_Q) \overset{\alpha}{\Longrightarrow}_{P'\times Q} (s_{P'}^n, s_Q^n)$ and $(s_{P'}^n, s_Q^n) \in Cmp(P', Q)$. As shown in Fig. 11, let

$$s'_P \in ExtDest_P(s_P^n, a_{n+1}), \text{if } a_{n+1} \in ExtEn_P^X(s_P^n);$$
$$\text{and } s'_P = s_P^n, \text{otherwise,}$$
$$s'_Q \in ExtDest_Q(s_Q^n, a_{n+1}), \text{if } a_{n+1} \in ExtEn_Q^X(s_Q^n);$$
$$\text{and } s'_Q = s_Q^n, \text{otherwise.}$$

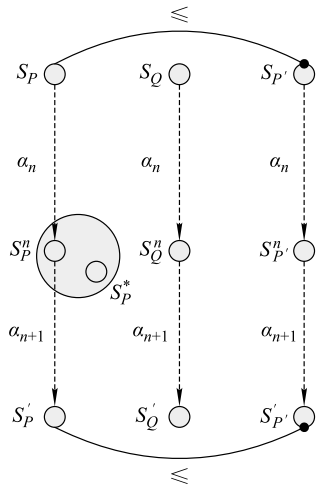


**Fig. 11** Illustration for Lemma 2

Note $a_{n+1} \in ExtEn_{P\times Q}^O(s_P^n, s_Q^n)$, there are two cases:

i) $a_{n+1} \in ExtEn_P^O(s_P^n)$:
Since $s'_P \in ExtDest_P(s_P^n, a_{n+1})$, and $s_P^* \in \epsilon\text{-}closure(s_P^n)$, then $s'_P \in ExtDest_P(s_P^*, a_{n+1})$. According to $s_P^* \preceq s_{P'}^n$, there must exist a state $s'_{P'} \in ExtDest_{P'}(s_{P'}^n, a_{n+1})$, such that $s'_P \preceq s'_{P'}$. Moreover, since $s'_Q \in ExtDest_Q(s_Q^n, a_{n+1})$, we have $(s'_{P'}, s'_Q) \in ExtDest_{P'\times Q}\left((s_{P'}^n, s_Q^n), a_{n+1}\right)$ and $(s'_{P'}, s'_Q) \in Cmp(P', Q)$. Thus proposition 1 holds.

ii) $a_{n+1} \in ExtEn_Q^O(s_Q^n)$:
We discuss two subcases here.

A) $a_{n+1} \in ExtEn_P^I(s_P^n)$:
In this case $a_{n+1} \in ExtEn_P^I(s_P^n) \cap ExtEn_Q^O(s_Q^n)$, then obviously $a_{n+1} \in A_P^I \cap A_Q^O$. Based on Eq. (3), we get $a_{n+1} \in A_{P'}^I \cap A_Q^O$, and thus $a \in A_{P'}^I$. Moreover, since $(s_{P'}^n, s_Q^n)$ is a compatible state, and $a_{n+1} \in ExtEn_Q^O(s_Q^n)$, we have $a_{n+1} \in ExtEn_{P'}^I(s_{P'}^n)$. From $s_P^* \preceq s_{P'}^n$, there must exist a state $s'_{P'} \in ExtDest_{P'}(s_{P'}^n, a_{n+1})$, such that $s'_P \preceq s'_{P'}$. Furthermore, as $s'_Q \in ExtDest_Q(s_Q^n, a_{n+1})$, we get

$$(s'_{P'}, s'_Q) \in ExtDest_{P'\times Q}\left((s_{P'}^n, s_Q^n), a_{n+1}\right),$$
$$(s'_{P'}, s'_Q) \in Cmp(P', Q).$$

Thus proposition 1 holds for this case.

B) $a_{n+1} \notin ExtEn_P^I(s_P^n)$:
Note $P$ and $Q$ are composable, and $a_{n+1} \in ExtEn_Q^O(s_Q^n)$, we get $a_{n+1} \notin ExtEn_P^O(s_P^n)$. Then $a_{n+1} \notin ExtEn_P^X(s_P^n)$, thus $s'_P = s_P^n$. Based on $s_P^* \preceq s_{P'}^n$, and $s_P^* \in \epsilon\text{-}closure(s_P^n)$, there is $ExtEn_P^I(s_P^n) \supseteq ExtEn_{P'}^I(s_{P'}^n)$. Thus $a_{n+1} \notin ExtEn_{P'}^I(s_{P'}^n)$. On the other hand, since $P'$ and $Q$ are composable, we can get $a_{n+1} \notin ExtEn_{P'}^O(s_{P'}^n)$. Thus $a_{n+1} \notin ExtEn_{P'}^X(s_{P'}^n)$. We set $s'_{P'} = s_{P'}^n$, then obviously the proposition 1 holds.

2) We prove the second proposition by contradiction. Assume $(s_P, s_Q) \notin Cmp(P, Q)$, there must be a sequence of actions $\alpha \in (A_{P\times Q}^O)^*$ and an error state $(s'_P, s'_Q) \in Illegal(P, Q)$ such that $(s_P, s_Q) \overset{\alpha}{\Longrightarrow}_{P\times Q} (s'_P, s'_Q)$. According to the first proposition, there exists a state $s''_P \in V_P$ and $s'_{P'} \in V_{P'}$, such that

$$s'_P \in \epsilon\text{-}closure(s''_P),\ s''_P \preceq s'_{P'},\ (s'_{P'}, s'_Q) \in Cmp(P', Q).$$

There are two cases for $(s'_P, s'_Q) \in Illegal(P, Q)$:

a) $\exists a \in shared(P, Q)$, s.t. $a \in A_P^O(s'_P)$, $a \notin A_Q^I(s'_Q)$:
From $s'_P \in \epsilon\text{-}closure(s''_P)$, we get $a \in ExtEn_P^O(s''_P)$. Then based on $s''_P \preceq s'_{P'}$, we get $a \in ExtEn_{P'}^O(s'_{P'})$, i.e., $\exists s''_{P'} \in \epsilon\text{-}closure(s'_{P'})$, such that $a \in A_{P'}^O(s''_{P'})$. As $a \notin A_Q^I(s'_Q)$, therefore $(s''_{P'}, s'_Q) \in Illegal(P', Q)$. Additionally, since $(s'_{P'}, s'_Q) \overset{\epsilon}{\Longrightarrow}_{P'\times Q} (s''_{P'}, s'_Q)$, then $(s'_{P'}, s'_Q) \in Illegal(P', Q)$. This conflicts with the premise.

b) $\exists a \in shared(P, Q)$, s.t. $a \notin A_P^I(s'_P)$, $a \in A_Q^O(s'_Q)$:
Since $a \notin A_P^I(s'_P)$, it must be $a \notin ExtEn_P^I(s''_P)$.

From $s''_P \preceq s'_{P'}$, we know $a \notin ExtEn^I_{P'}(s'_{P'})$ i.e., $\exists s''_{P'} \in \epsilon\text{-}closure(s'_{P'})$, such that $a \notin A^I_{P'}(s''_{P'})$. As $a \in A^O_Q(s'_Q)$, therefore $(s''_{P'}, s'_Q) \in Illegal(P', Q)$. Additionally, since $(s'_{P'}, s'_Q) \overset{\epsilon}{\Longrightarrow}_{P'\times Q} (s''_{P'}, s'_Q)$, then $(s'_{P'}, s'_Q) \in Illegal(P', Q)$. This conflicts with the premise too.

In summary, there is a conflict between the conclusion and premise, therefore the assumption is not true, and the proposition 2 holds.

### A.4 Proof to Lemma 3

**Proof** We prove the first formula by contradiction. Assume the first formula does not hold, then

$$\exists a \in ExtEn^I_{P'\times Q}\left((s_{P'}, s_Q)\right), a \notin ExtEn^I_{P\times Q}\left((s_P, s_Q)\right).$$

Then $\exists s'_P \in \epsilon\text{-}closure(s_P), \exists s'_Q \in \epsilon\text{-}closure(s_Q)$, such that $a \notin A^I_{P\times Q}\left((s'_P, s'_Q)\right)$.

We discuss in the following three cases for $a \notin A^I_{P\times Q}\left((s'_P, s'_Q)\right)$:

- $a \notin A^I_P(s'_P)$ and $a \notin A^I_Q(s'_Q)$:
  Since $a \notin A^I_P(s'_P)$, then
  $$a \notin ExtEn^I_P(s_P). \tag{4}$$
  For any $s'_{P'} \in \epsilon\text{-}closure(s_{P'})$, since $a \in ExtEn^I_{P'\times Q}\left((s_{P'}, s_Q)\right)$, there is $a \in A^I_{P'\times Q}(s'_{P'}, s'_Q)$. Moreover, since $a \notin A^I_Q(s'_Q)$, there must be $a \in A^I_{P'}(s'_{P'})$. Then
  $$a \in ExtEn^I_{P'}(s_{P'}). \tag{5}$$
  From Eqs. (4) and (5), we conclude that $ExtEn^I_P(s_P) \supseteq ExtEn^I_{P'}(s_{P'})$ does not hold. This conflicts with the premise that $s_P \preceq s_{P'}$.
- $a \in A^O_P(s'_P)$:
  Since $a \in A^O_P(s'_P)$, then
  $$a \in ExtEn^O_P(s_P). \tag{6}$$
  Since $a \in ExtEn^I_{P'\times Q}\left((s_{P'}, s_Q)\right)$, then for any $s'_{P'} \in \epsilon\text{-}closure(s_{P'})$, there is $a \notin A^O_{P'}(s'_{P'})$. Thus
  $$a \notin ExtEn^O_{P'}(s_{P'}). \tag{7}$$
  From Eqs. (6) and (7), we conclude that $ExtEn^O_P(s_P) \subseteq ExtEn^O_{P'}(s_{P'})$ does not hold. This conflicts with the premise of $s_P \preceq s_{P'}$.
- $a \in A^O_Q(s'_Q)$:
  For any $s'_{P'} \in \epsilon\text{-}closure(s_{P'})$, since $a \in A^O_Q(s'_Q)$, then $a \in A^O_{P'\times Q}\left((s'_{P'}, s'_Q)\right)$. This conflicts with the premise of $a \notin A^I_{P'\times Q}\left((s'_{P'}, s'_Q)\right)$.

Since all the cases lead to contradiction, we then conclude the first formula holds. The second formula can be proved in a similar way.

## References


1. Alfaro L D, Henzinger T A. Interface automata. In: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2001, 109–120
2. Alfaro L D, Henzinger T A. Interface-based design. Engineering Theories of Software-intensive Systems, 2005, 195: 83–104
3. Lynch N A, Tuttle M R. Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing. 1987, 137–151
4. Lynch N A, Tuttle M R. An introduction to input/output automata. CWI-Quarterly, 1989, 2(3): 219–246
5. Lynch N A. Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo, CA, 1996.
6. Kaynar D K, Lynch N A, Segala R, Vaandrager F. Timed I/O automata: a mathematical framework for modeling and analyzing realtime systems. In: Proceedings of the 24th IEEE Real-Time Systems Symposium. 2003, 166–177
7. Kaynar D K, Lynch N A, Segala R, Vaandrager F. The theory of timed I/O automata. Synthesis Lectures on Computer Science, 2006, 1(1): 1–114
8. Wu S H, Smolka S A, Stark E W. Composition and behaviors of probabilistic I/O automata. Theoretical Computer Science, 1997, 176(1–2): 1–38
9. Stark E W, Smolka S A. Compositional analysis of expected delays in networks of probabilistic I/O automata. In: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science. 1998, 466–477
10. Stark E W, Cleaveland R, Smolka S A. A process-algebraic language for probabilistic I/O automata. Lecture Notes in Computer Science, 2003, 2761: 193–207
11. Lynch N A, Segala R, Vaandrager F W. Hybrid I/O automata. Information and Computation, 2003, 185(1): 105–157
12. Alfaro L D, Henzinger T A. Interface theories for component based design. Lecture Notes in Computer Science, 2001, 2211: 148–165
13. Wen Y J, Wang J, Qi Z C. Bridging refinement of interface automata to forward simulation of I/O automata. In: Proceedings of the 6th International Conference on Formal Engineering Methods. 2004, 259–273
14. Wen Y J, Wang J, Qi Z C. 2/3 alternating simulation between interface automata. In: Proceedings of 7th International Conference on Formal Engineering Methods. 2005, 173–187.
15. Chakrabarti A, Alfaro L D, Henzinger T A, Jurdzinski M, Mang F Y C. Interface compatibility checking for software modules. Lecture Notes in Computer Science, 2002, 2404: 428–441
16. Chakrabarti A, Alfaro L D, Henzinger T A, Mang F Y C. Synchronous and bidirectional component interfaces. Lecture Notes in Computer

Science, 2002, 2404: 414–427
17. Alfaro L D, Henzinger T A, Stoelinga M. Timed interfaces. Lecture Notes in Computer Science, 2002, 2491: 108–122
18. Chakrabarti A, Alfaro L D, Henzinger T A, Stoelinga M. Resource interfaces. Lecture Notes in Computer Science, 2003, 2855: 117–133
19. Alfaro L D, Silva L D D, Faella M, Legay A, Roy P, Sorea M. Sociable interfaces. Lecture Notes in Computer Science, 2005, 3717: 81–105
20. Alur R, Henzinger T A, Kupferman O, Vardi M. Alternating refinement relations. Lecture Notes in Computer Science, 1998, 1466: 163–178

Fei He received the BS degree from National University of Defense Technology in 2002, and the PhD degree from Tsinghua University in 2008. He is currently an associate professor in the School of Software at Tsinghua University, Beijing, China. His research interests include satisfiability, model checking, compositional reasoning, and their applications to embedded systems.

Xiaoyu Song received the PhD degree from the University of Pisa, Italy in 1991. From 1992 to 1999, he was on the faculty at the University of Montreal, Canada. In 1998, he worked as a senior technical staff in Cadence, San Jose. In 1999, he joined the faculty at Portland State University. He is currently a professor in the Department of Electrical & Computer Engineering at Portland State University, Portland. His current research interests include formal methods, design automation, embedded system design, and emerging technologies. He has been awarded as the Intel Faculty Fellow during 2000–2005. He served as an associate editor of IEEE Transactions on Circuits and Systems and IEEE Transactions on VLSI Systems.

Ming Gu received the BS degree in computer science from National University of Defence Technology, Changsha, China in 1984, and the MS degree in computer science from the Chinese Academy of Science at Shengyang in 1986. Since 1993, she has been working as a lectuer/associate professor/researcher in Tsinghua University. Her research interests include formal methods, middleware technology, and distributed applications.

Jiaguang Sun received the BS degree in automation science from Tsinghua University in 1970. He is currently a professor in Tsinghua University. He is an expert in computer graphics, computer-aided design, formal verification of software, software engineering, and system architecture. He was elected to the Chinese Academy of Engineering in 1999. He is currently serving as the dean of the School of Information Science & Technology in Tsinghua University, and the director of the National Laboratory for Information Science & Technology.