Refinement of interface automata strengthened by action semantics

Sebti Mouelhi, Samir Chouali, Hassan Mountassir [1]

Laboratoire d'Informatique de l'Université de Franche-Comté - LIFC
16, route de Gray - 25030 Besançon cedex, France

Abstract

Interface automata are light-weight models that capture the temporal interface behavior of software components. They have the ability to model both the input requirements and the output behavior of a component. They support the compatibility check between interface models to ensure a correct interaction between components and they adopt an alternating simulation approach to design refinement. In this paper, we extend our previous works on checking interface automata interoperability by adapting their alternating refinement relation to the action semantics. We show the relation between pre and post-conditions of transitions in the abstract version of an interface and their corresponding ones in its concrete version. We illustrate our extensions by a case study of the CyCab car component-based system.

Keywords: Component-based systems, interface automata, alternating refinement.

1 Introduction

Interface formalisms play a central role in the conception of component-based systems. They are increasingly used thanks to their ability to describe, in terms of communicating interfaces, how the components of a system can be connected to each others. Two principles have to be satisfied to design properly component interfaces. First, an interface should describe enough information about the way to make two or more components "working together correctly" by looking only at their interfaces. Second, more information than is required by the first principle should not be exposed. Essentially, component interoperability have to satisfy the type compatibility of operations (the number, order, and types of the parameters). In addition of type check, component composition requires protocol information about how a component must be used in a system design and the order in which their interacting events are enabled. Interfaces that expose protocol information of components can be specified naturally in an automaton-based language like interface automata [1,2].

Interface automata have been introduced as a formalism that captures the temporal Input/Output behavior of a software component. Similarly to Input/Output automata [12],

[1] Email: {sebti.mouelhi, samir.chouali, hassan.mountassir}@lifc.univ-fcomte.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

they are specified by automata labeled by input, output, and internal actions. The approach of interface automata adopts an *optimistic* or *environment-constraining* view where the composition of two compatible interfaces can be used together in at least one design thanks to the non-input-enabled property, which means that at every state, some input actions may not be enabled contrarily to I/O automata and CSP [12]. There interaction is made by synchronizing shared input an output actions, while the internal actions of concurrent automata are interleaved asynchronously. Automatic compatibility verification and refinement checking can be made. The formalism of interface automata has been applied in several cases as a formal model to specify the interface behaviors of software components [4,9]. A path often taken in the literature is to check the interface compatibility of components at the semantic level of operations. In our previous work [6], we improve the model of interface automata to ensure a more reliable verification of components interoperability by taking into account the semantics of actions. The proposed method enrich transitions of interface automata by pre and post-conditions of actions which are atomic prepositions over a set of variables. The scope of our previous work do not cover the totality of interface automata by treating refinement. This paper is essentially written to expose the refinement our extended interface automata.

The role of refinement relation is to formalize the relationship between the abstract and the concrete versions of the same component. For I/O automata, refinement is usually defined as trace containment or simulation [10]; this ensures that the output behaviors of the refined automaton are behaviors that are allowed by the abstract one. Such definitions of refinement do not hold for non-input-enabled setting, such as interface automata: if the set of legal inputs of the refined interface is a subset of the inputs allowed by the abstract one, then the refined interface could be used in fewer environments than the interface abstraction. While a new approach is adapted to compose interface automata, an alternating approach is used to refine them.

Alternating refinement simulation is defined to study refinement between alternating transition systems [13]. They are introduced as a general model for component-based systems which allow the study of adversarial relationships between individual system components. Unlike in labeled transition systems where each transition represents a possible step of the system, each transition of an alternating transition system corresponds to a possible move in a game between different components. The proposed refinement of interface automata is based on this approach by viewing them as alternating transition systems. Explicitly, a refinement of an interface automata expresses that the refined component can offer more services (input actions) and fewer service demands (output actions). In this article, we adapt the alternating refinement simulation of interface automata by taking into account the relation between pre and post-conditions among the input and output transitions of an automaton and their correspondent refinements. In other word, a refined version uses more variables to formulate pre and post-conditions of the refined and added input actions. We suppose that the pre and post-conditions of the remaining output actions do not change. Intuitively, while the offered services are local in the component, we have to strengthen their semantics constraints if which is not the case for demanded services because the component ignore if the demanded service was refined or not in the environment.

Concretely, we strengthen the alternating simulation between states of a refined interface and its correspondent abstract one by establishing equivalence and implications between their similar input and output actions. After this introduction, in section 2 and 3,

we will give an overview of interface automata and we will present our contribution of considering action semantics to verify their interoperability. In section 4, we will detail our adaptation of the alternating refinement simulation of interface automata to the semantics of actions and we will illustrate our works by a case study of the CyCab car component-based system.

2 Preliminary

I/O automata have been introduced by Nancy A.Lynch and Mark.Tutle [12] as labeled transition systems. Commonly, they are used to model distributed and concurrent systems. Labels of I/O automata fall into three categories of actions: input, output, and hidden actions where input actions are enabled at every state of an automaton.

Defintion 2.1 *An I/O automaton* $A = \langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ *consists of*

- *a finite set* S_A *of states;*
- *a subset of initial states* $I_A \subseteq S_A$*;*
- *three disjoint sets* $\Sigma_A^I, \Sigma_A^O and \Sigma_A^H$ *of inputs, output, and hidden actions. All actions, as a whole, are denoted by* $\Sigma_A = \Sigma_A^I \cup \Sigma_A^O \cup \Sigma_A^H$*;*
- *a set* $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *of transitions. It gives a transition relation with the property that for every state s and an input action a there is a transition (s, a, s) in* δ_A*.*

Interface automata have been defined by L.Alfaro and T.Henzinger [1], to model the temporal behavior of software component interfaces. These models are non-input-enabled I/O automata, as previously said, where it is not necessary to enable input actions at every state of one automaton. Every component interface is described by one interface automaton where input actions are used to model methods that can be called, and the end of receiving messages from communication channels, as well as the return values from such calls. Output actions are used to model method calls, message transmissions via communication channels, and exceptions that occur during the method execution. Output actions describe the required actions of a component (represented by the symbol "!"), input actions describe the provided actions of a component (represented by the symbol "?"), and internal (or hidden) actions inside the component itself describe its local operations (represented by the symbol ";"). Both for I/O automata (IOAs) and interface automata (IAs), the input and output actions of an automaton *A* are called external actions uniformly ($\Sigma_A^{ext} = \Sigma_A^I \cup \Sigma_A^O$) while output actions and internal actions are called locally-controlled actions ($\Sigma_A^{loc} = \Sigma_A^O \cup \Sigma_A^H$). We define by $\Sigma_A^I(s)$, $\Sigma_A^O(s)$, $\Sigma_A^H(s)$ the input, output, and internal actions enabled at the state *s*.

Definition 2.2 *An interface automaton* $A = \langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ *consists of*

- *a finite set* S_A *of states;*
- *an subset of initial states* $I_A \subseteq S_A$*. It contains at most one state. If* $I_A = \varnothing$*, then A is called empty;*
- *three disjoint sets* Σ_A^I, Σ_A^O *and* Σ_A^H *of inputs, output, and hidden actions;*
- *a set* $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *of transitions between states. Contrarily to I/O automata, the*

input actions are not necessarily enabled at every state.

For an IA *A*, we define two type of actions $a \in \Sigma_A$ and $a_x \in \Sigma_A^{ext}$, and two type of action sequences $\alpha = a_1 a_2 ... a_n \in (\Sigma_A)^n$ and $\beta = b_1 b_2 ... b_n \in (\Sigma_A^{ext})^n$. Given two states s_1 and s_2, we define the following relations.

- $s_1 \xrightarrow{a}_A s_2$ iff $(s_1, a, s_2) \in \delta_A$;
- $s_1 \xrightarrow{\tau}_A s_2$ iff $s_1 \xrightarrow{b}_A s_2$ for some $b \in \Sigma_A^{int}$;
- $s_1 \xrightarrow{\alpha}_A s_2$ iff $s_1 \xrightarrow{a_1}_A \xrightarrow{a_2}_A ... \xrightarrow{a_n}_A s_2$;
- $s_1 \overset{\varepsilon}{\Longrightarrow}_A s_2$ iff $s_1 (\xrightarrow{\tau}_A)^* s_2$ (* is reflexive and transitive closure and juxtaposition of transitions);
- $s_1 \overset{a_x}{\Longrightarrow}_A s_2$ iff $s_1 \overset{\varepsilon}{\Longrightarrow}_A \xrightarrow{a_x}_A s_2$ (this relation is called input or output sequence of steps according to the type of the action a_x and states between the two extremities s_1 and s_2 are called internal states);

The optimistic view of interface automata incorporates a notion of interface composition that leads to smaller compound automata than the input-enabled view. When we compose two interface automata, the resulting composite automaton may contain *illegal states*, where one automaton issues an output that is not acceptable as input in the other one. The proposed approach to compute compatibility between interface automata based on the fact that each interface expects the environment to provide only legal inputs. The compound interface expects the environment to pass over transitions leading only to legal states. The existence of a such legal environment for the composition of two interfaces indicates that there is a way to use their corresponding components together by ensuring the encounter of their environment assumptions. The composite interface automaton combines the behaviors of the two component interfaces and the environment assumptions under which the components can work together properly.

3 Interface automata strengthened by action semantics

Our approach presented in [6] extends interface automata by considering the action semantics to ensure a more reliable verification of component interoperability. In [1], the checking of the component compatibility uses only action signatures, which are not sufficient to decide if two interfaces are compatible or not. Our contribution uses pre and post-conditions over a set of variables to annotate the actions of interface automata. These constraints on actions show their semantic effects which can be useful to strengthen the compatibility checking. The proposed algorithm to verify the composition and the compatibility between interface automata takes into account of pre and post-conditions of actions.

We introduce a finite set of variables $x \in V$ with their respective domain D_x. These variables are used to represent the effect of actions by updating there values. The variable updates are modeled by pre and post atomic formulas over *V*.

Definition 3.1 *Let A =* $\langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, Pre_A, Post_A, \delta_A \rangle$ *be an IA strengthened by action semantics where*

- *a finite set* S_A *of states;*
- *an initial state* $I_A \subseteq S_A$*;*

- *three disjoint sets* $\Sigma_A^I, \Sigma_A^O and\ \Sigma_A^H$ *of inputs, output, and hidden actions;*
- *Pre and Post are the set of pre and post-conditions of actions, they are atomic formulae over the set of variables V;*
- *a set* $\delta_A \subseteq S_A \times\ Pre_A\ \times \Sigma_A \times\ Post_A\ \times S_A$ *of transitions.*

For $a \in \Sigma_A$*, we denote by* Pre_{A_a} *and* $Post_{Q_a}$ *respectively the precondition and post-condition of the action a in the automaton A.*

The composition condition is the same as the preexisting approach. The composition of two automata may take effect only if their actions are disjoint, except shared input and output actions between them. When we compose them, shared actions are synchronized and all the others are interleaved asynchronously.

Definition 3.2 *Two interface automata* A_1 *and* A_2 *are composable if*

$$\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \cap \Sigma_{A_2} = \Sigma_{A_2}^H \cap \Sigma_{A_1} = \emptyset$$

Shared(A_1,A_2) = $(\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_{A_2}^I \cap \Sigma_{A_1}^O)$ is the set of shared input and output actions between A_1 and A_2. We can now define the product automaton $A_1 \otimes A_2$ properly. We mention that some transitions in A_1 and A_2 may not occur in the product.

Definition 3.3 *Let* A_1 *and* A_2 *be two composable interface automata. The product* $A_1 \otimes A_2$ *is defined by*

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ and $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma_{A_1 \otimes A_2}^I = (\Sigma_{A_1}^I \cup \Sigma_{A_2}^I)\ \setminus\ Shared(A_1, A_2)$;
- $\Sigma_{A_1 \otimes A_2}^O = (\Sigma_{A_1}^O \cup \Sigma_{A_2}^O)\ \setminus\ Shared(A_1, A_2)$;
- $\Sigma_{A_1 \otimes A_2}^H = \Sigma_{A_1}^H \cup \Sigma_{A_2}^H\ \cup\ Shared(A_1, A_2)$;
- $((q_1, q_2), Pre, a, Post, (q_1', q_2')) \in \delta_{A_1 \otimes A_2}$ if
  - $a \notin Shared(A_1, A_2)\ \wedge\ (q_1, Pre_1, a, Post_1, q_1') \in \delta_{A_1}\ \wedge\ q_2 = q_2'\ \wedge\ Pre \equiv Pre_1\ \wedge\ Post \equiv Post_1$
  - $a \notin Shared(A_1, A_2)\ \wedge\ (q_2, Pre_2, a, Post_2, q_2') \in \delta_{A_2}\ \wedge\ q_1 = q_1'\ \wedge\ Pre \equiv Pre_2\ \wedge\ Post \equiv Post_2$
  - $a \in Shared(A_1, A_2)\ \wedge\ ((q_1, Pre_1, a, Post_1, q_1') \in \delta_{A_1}\ \wedge\ a \in \Sigma_{A_1}^I)\ \wedge\ ((q_2, Pre_2, a, Post_2, q_2') \in \delta_{A_2}\ \wedge\ a \in \Sigma_{A_2}^O)\ \wedge\ Pre \equiv Pre_2\ \wedge\ Post \equiv Post_1$ *such that* $Pre_2 \Rightarrow Pre_1\ \wedge\ Post_1 \Rightarrow Post_2$
  - $a \in Shared(A_1, A_2)\ \wedge\ ((q_1, Pre_1, a, Post_1, q_1') \in \delta_{A_1}\ \wedge\ a \in \Sigma_{A_1}^O)\ \wedge\ ((q_2, Pre_2, a, Post_2, q_2') \in \delta_{A_2}\ \wedge\ a \in \Sigma_{A_2}^I)\ \wedge\ Pre \equiv Pre_1\ \wedge\ Post \equiv Post_2$ *such that* $Pre_1 \Rightarrow Pre_2\ \wedge\ Post_2 \Rightarrow Post_1$

Illegal states are the states at which the shared actions do not synchronize. We distinguish two different cases: (i) a component requires a shared action which is not provided by the environment, or (ii) they synchronize on a shared action between them but the required action and the provided one are not compatible at the semantic level.

Definition 3.4 *Given two composable interface automata* A_1 *and* A_2*, the set of illegal states Illegal*(A_1,A_2) $\subseteq S_1 \times S_2$ *of* $A_1 \otimes A_2$ *is defined by* $\{(q_1, q_2) \in S_{A_1} \times S_2 \mid \exists\ a \in$

$Shared(A_1, A_2)$. *such that the following conditions hold* } .

$$\left(\begin{array}{c} a \in \Sigma_1^O(q_1) \wedge a \notin \Sigma_2^I(q_2) \\ \vee \\ (a \in \Sigma_1^O(q_1) \wedge a \in \Sigma_2^I(q_2) \\ \wedge \\ (Pre_1 \not\Rightarrow Pre_2) \vee (Post_2 \not\Rightarrow Post_1)) \end{array}\right) or \left(\begin{array}{c} a \in \Sigma_2^O(q_2) \wedge a \notin \Sigma_1^I(q_1) \\ \vee \\ (a \in \Sigma_2^O(q_2) \wedge a \in \Sigma_1^I(q_1) \\ \wedge \\ (Pre_2 \not\Rightarrow Pre_1) \vee (Post_1 \not\Rightarrow Post_2)) \end{array}\right)$$

The set of illegal states in the product $A_1 \otimes A_2$ describes the possibility that one of the two automata may produce an output action that is an input action of the other, but it is not accepted. In our contribution, we extend the previous definition by the possibility that, for some states (q_1, q_2) in the set of illegal states, an output action issued from q_1 in A_1 can be synchronized with the same action enabled as input at q_2 in A_2 but the precondition of the output action does not imply the the precondition of the input action or its post-condition is not implied by the post-condition of the input one.

Compatible states, denoted by *Comp*(A_1,A_2), are states from which the environment can prevent entering illegal states. The compatibility can be defined differently, A_1 and A_2 are compatible iff their initial state is compatible.

Definition 3.5 *Given two composable interface automata A_1 and A_2. The compostion $A_1 \parallel A_2$ is an interface automaton defined by: (i) $S_{A_1\|A_2}$ = Comp(A_1,A_2), (ii) the initial state is $I_{A_1\|A_2}$ = $I_{A_1\otimes A_2} \cap$ Comp(A_1,A_2), (iii) $\Sigma_{A_1\|A_2} = \Sigma_{A_1\otimes A_2}$, and (iv) the set of transitions is $\delta_{A_1\|A_2}$ = $\delta_{A_1\otimes A_2} \cap$ (Comp(A_1,A_2) $\times$ $Pre_{A_1\otimes A_2}$ $\times$ $\Sigma_{A_1\|A_2}$ $\times$ $Post_{A_1\otimes A_2}$ $\times$ Comp(A_1,A_2)).*

The verification steps in this approach are the same as [1] except that we consider the action semantics. The proposed algorithm [6] verify the compatibility of two interface automata by checking if their composition is nonempty. We mention that our approach does not increase the linear complexity of the previous proposed one. Finally, we add that the associative criterion of the composition operator $\|$ between three automata is undefined when some of them are not composable.

4 Adapting the alternating refinement relation

The aim of the refinement relation is to concretize an abstract version of a component interface. It permits to move a component or an interface from a high-level understanding to a more concrete specification. Contrarily to traditional types of I/O automata, refinement is defined as trace containment, the refinement of interface automata is based on an alternating refinement relation in the spirit of simulation. A more concrete version of an interface have to be used in stronger environments than its abstraction. In other words, the refinement of an interface must allow more legal inputs, and fewer outputs than the abstract version.

By taking the fact that the internal actions are independent, an interface automaton Q

refines another *P* if all input transitions of the second one can be simulated by the first one, and contrarily for output transitions.

4.1 Preliminary

We recall the preliminary notions used to define the alternating simulation relation between interface automata. The ε-closure of a state *s* is the set of all reachable states from *s* by transiting only internal steps. The environment cannot distinguish between *s* and all states of ε-closure(*s*).

Definition 4.1 *Given an interface automaton P and a state $s \in S_P$, ε-closure$_P$(s) is the smallest set $R \subseteq S_P$ such that (1) $s \in R$ and (2) for $s' \in R$, if there exists s" such that $s' \stackrel{\varepsilon}{\Longrightarrow}_P s''$ is a sequence of internal steps, then $s'' \in R$.*

An interface automaton *P* must be able to accept an output action *a* issued from the environment if *a* is accepted at all states in ε-closure(*s*). Contrarily, *P* can issue an output action *b* at least from one state in ε-closure(*s*) to the environment.

Definition 4.2 *The sets of externally enabled output and input actions at a state $s \in S_P$ are defined as follow*

- $ExtEn^O_P(s) = \{a \mid \exists\, r \in \varepsilon\text{-}closure(s).\ a \in \Sigma^O_P(r)\}$
- $ExtEn^I_P(s) = \{a \mid \forall\, r \in \varepsilon\text{-}closure(s).\ a \in \Sigma^I_P(r)\}$.

We redefine also the set of all reachable states from a state *s* by transiting steps labeled by externally enabled actions.

Definition 4.3 *The set $ExtDest_P(s, a)$ of externally reachable states from a state s in an interface automaton P for an externally enabled action $a \in ExtEn^O_P(s) \cup ExtEn^I_P(s)$ is defined by the set $\{r' \mid \exists\, r \stackrel{a}{\longrightarrow}_P r'.\ r \in \varepsilon\text{-}closure(s)\}$*

4.2 Alternating simulation

Let us consider now pre and post-conditions to establish properly the new definition of the alternating simulation between the states of an interface automaton *P* and its refined version *Q*. We extend the set of variables *V* by adding some others, so we define *V'* as a set that includes the set *V*. We assume that pre and post-conditions of the refined interface automaton *Q* are defined over the set *V'*.

Defining pre and post-conditions of actions of the refined interface must obey to some variant constraints. On the one hand, when we refine a component, we add to their provided services (input actions) some other new services by defining new signatures of actions and on the other hand, we strengthen their former operations by adding some other constraints on their pre and post-conditions. We take into account the principle that, in the refined interface, an old input action must have a fewer precondition than the precondition of the same corresponding action in the abstract one and its post-condition must be stronger than the corresponding post-condition in the abstraction.

Things change for required services, constraints on the required services (output actions) in the abstract interface still unchanged in the refinement. In more details, it is

assumed that there is less output actions in the refinement and the required computing results of extinct actions become internal. So, refining remaining ones has no sense. More concretely, the pre and post-conditions of a remaining output action in the abstract interface are equivalent to their correspondents in the refined one. For the requirements of internal actions, we apply the same rules as input actions. We can now define our manner to adapt alternating simulation to the action semantics formally.

Definition 4.4 *A binary relation* $\preceq \subseteq S_P \times S_Q$ *from Q to P is an* alternating simulation *if for all* $s \in S_P$, $r \in S_Q$ *such that* $r \preceq s$ *the following conditions holds*

(i) $ExtEn_P^I(s) \subseteq ExtEn_Q^I(r)$;

(ii) $ExtEn_Q^O(r) \subseteq ExtEn_P^O(s)$;

(iii) $\forall\ a \in ExtEn_P^I(s) \cup ExtEn_Q^O(r)$ *and* $\forall\ r' \in ExtDest_Q(r,a)$: $\exists\ s' \in ExtDest_P(s,a)$ *such that* $r' \preceq s'$ *and*
- *if* $a \in ExtEn_P^I(s)$ *then* $Pre_{P,a} \Rightarrow Pre_{Q,a}$ *and* $Post_{Q,a} \Rightarrow Post_{P,a}$.
- *else if* $a \in ExtEn_Q^O(r)$ *then* $Pre_{P,a} \Leftrightarrow Pre_{Q,a}$ *and* $Post_{P,a} \Leftrightarrow Post_{Q,a}$.

over the set of variables V'.

The first condition of the second part of the definition ensures that all externally enabled inputs of *s* are also externally enabled in *r*, and conversely, all externally enabled outputs of *r* are also externally enabled in *s*.

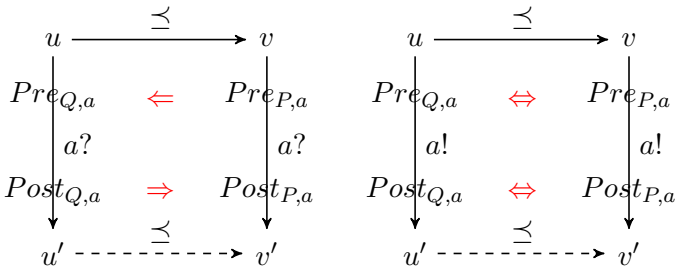

Fig. 1. The extended condition 2 of the alternating simulation definition.

The second one ensures that each input (resp. output) sequence of steps labeled by an externally enabled action *a* from *r* in the refinement must be matched by an input (resp. output) sequence of step labeled by the same action *a* from *s* in the abstraction except that the equivalences and the implications between pre and post-conditions must be checked as it is shown in Figure 1. We can now define the refinement between two interface automata *P* and *Q* as follow

Definition 4.5 *The interface automaton Q* refines *the interface automaton P, written* $Q \preceq P$ *according to the set of variables V' if*

- $\Sigma_P^I \subseteq \Sigma_Q^I$ *and* $\Sigma_P^O \supseteq \Sigma_Q^O$;
- *there is an alternating simulation* $\preceq$ *from Q to P such that* $I_Q \preceq I_P$.

As in [1], we can easily verify that refinement between interface automata in our contribution is reflexive an transitive. But, when we want to establish the relation between the

refinement and the compatibility between automata things have to be more detailed: a more refined version Q of P can replace P in a system design such that $Q \preceq P$ if the environment does not provide some input actions (calls of some offered services) for Q that are not in P. Alternatively, new incompatibilities may be arise when we compose Q with the environment, namely all new input actions in the refined version Q must not be required as output actions by the environment.

Also, another conditions must be verified when we consider pre and post-conditions. A refined version of an interface P remains consistent with the environment if the abstraction is compatible with it under the hypothesis seated previously. As shown in the Figure 2, the step $x \xrightarrow{a!}_{Env} y$ of *Env* is compatible with final step of the input sequences of steps $1 \overset{a?}{\Longrightarrow}_P 2$ of P , then it is compatible also with the final step of $1' \overset{a?}{\Longrightarrow}_Q 2'$ of the refinement Q of P such that $1' \preceq 1$ and $2' \preceq 2$ because $Pre_{Env,a} \Rightarrow Pre_{Q,a}$ and $Post_{Q,a} \Rightarrow Post_{Env,a}$. Things do not change when we consider output steps in the refinement Q. Based on the fact that pre and post-conditions of the remaining actions in the refinement are equivalent to their corresponding ones in the abstraction, the implications $Pre_{Env,a} \Rightarrow Pre_{Q,a}$ and $Post_{Q,a} \Rightarrow Post_{Env,a}$ are also satisfied. The dashed edges in the previous and the following figure represent input or output sequences of steps between states 1 and 2 in P and between 1' and 2' in Q. We can easily deduce that the substitution of an automaton by a more refined one preserve the compatibility between them.

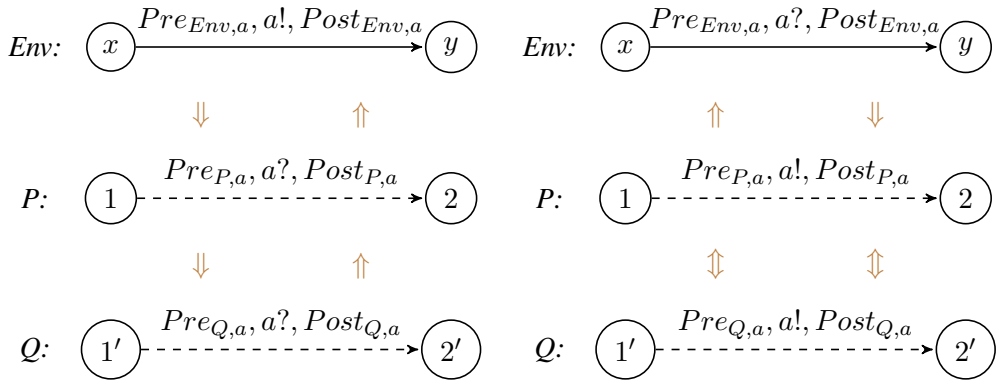

Fig. 2. Relation between the environment and the refined version Q of P

We can now rise the following theorem properly as in [1] without having hindsight.

Theorem 4.1 *Consider three interface automata P, Q, and R such that Q and R are composable and* $\Sigma_I^Q \cap \Sigma_O^R \subseteq \Sigma_I^P \cap \Sigma_O^R$. *If P and R are compatible and* $Q \preceq P$, *then Q and R are compatible and* $Q \| R \preceq P \| R$.

From the hypothesis of this theorem, we can extract the following important corollary establishing that if two interface automata are compatible then their refinements are compatible and the composition of their refinement refines them.

Corollary 4.1 *Consider four automata P, Q, R, and S such that*

- *Q and R are composable*;
- $\Sigma_I^Q \cap \Sigma_O^R \subseteq \Sigma_I^P \cap \Sigma_O^R$;

- *S and Q are composable*;
- $\Sigma_I^S \cap \Sigma_O^Q \subseteq \Sigma_I^R \cap \Sigma_O^Q$;

If P and R are compatible, $Q \preceq P$*, and* $S \preceq R$ *then Q is compatible with R and S is compatible with Q and* $Q \| S \preceq P \| R$.

To check that $Q \preceq P$, we should compute the maximal alternating simulation between the two initial states of P and Q thanks to the algorithm proposed in [1]. Our approach does not increase the complexity of the algorithm.

4.3 Case study of the CyCab car

We illustrate our works by applying refinement to the component vehicle of the CyCab car component-based system studied in [6]. The CyCab [7] car is a new electrical means of transportation conceived essentially for free-standing transport services. It is totally manipulated by a computer system and it can be driven automatically according to many modes.

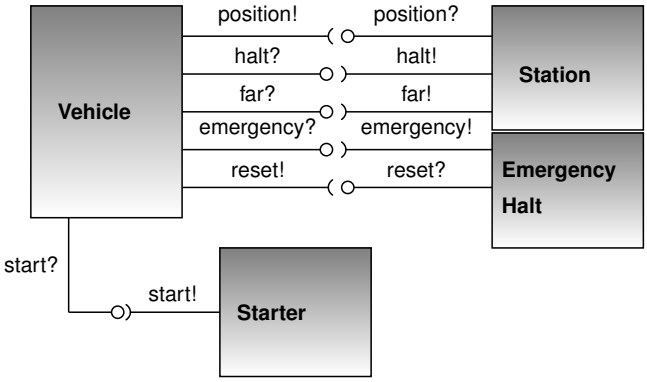

Fig. 3. A UML-like model of the CyCap components.

The goal of the CyCab car system design is to allow for users the displacement of the vehicle from one station to another. As an illustration of its concept, we consider the following requirements and functionalities of the CyCab car and its environment: (i) a CyCab has and appropriate road where stations are equipped by sensors, (ii) the driving of the CyCab is guided by information received from the station allowing to position of the CyCab from the stations, (iii) there is no obstacle in the roads, (vi) the vehicle has a starter and also an emergency halt button.

The CyCab car and its environment can be seen as an abstract system composed of four components: the vehicle, the emergency halt button, the starter, and the station. The Figure 3 represents the UML [2] component model of our system. The emergency halt button can be activated at every moment during the running of the vehicle. It is specified by sending a signal *emergency!*. The starter allows the starting of the vehicle. The station is materialized by a sensor that receives signals *position?* from the vehicle to know its position. The station sends as consequence a signal *far!* or *halt!* to the vehicle to indicate if it is far from the station or not.

[2] The component diagram showed in Figure 3 do not respect exactly the UML 2 notation. It is simply used to clarify the CyCab system

In this section, as shown in Figure 3, we apply our proposed contribution of refinement of the interface automaton of the component vehicle. Assume that A_v is the interface automaton associated to the component vehicle and $V = \{\ car_{strd}, iskn_{pos}, isac_{str}, isrc_{stn}, isnul_{dist}\ \}$ be the set of five boolean variables used to define pre and post-conditions of actions.

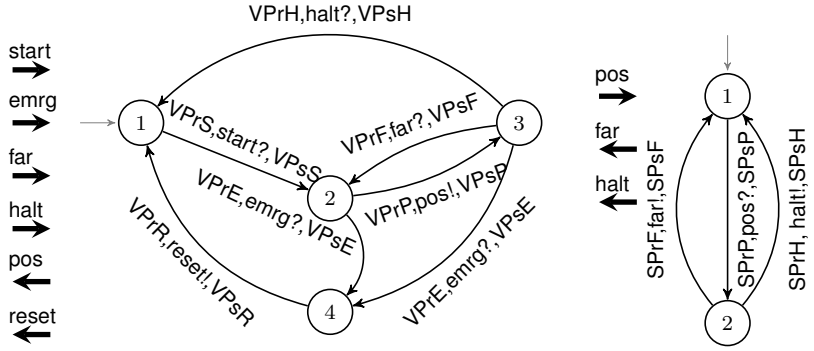

Fig. 4. The IAs A_v and A_s of the Vehicle and the Station

The variable car_{strd} indicates if the vehicle is started or not, the variable $iskn_{pos}$ indicates if the vehicle knows its position from the station, $isac_{str}$ equals to true when the starter is active, $isrc_{stn}$ equals to true when the station is reached, and finally the variable $isnul_{dist}$ indicates if the distance between the vehicle and the station is null or not. The automaton A_v is given by the tuple $\langle S_v, I_v, \Sigma_v^I, \Sigma_v^O, \Sigma_v^H, Pre_v, Post_v, \delta_v \rangle$ where

- Pre_v = $\{VPrH, VPrS, VPrE, VPrF, VPrP, VPrR\}$ where
  - $VPrH \equiv car_{strd} = true \wedge isrc_{stn} = false \wedge iskn_{pos} = true \wedge isnul_{dist} = true$;
  - $VPrS \equiv iskn_{pos} = false \wedge car_{strd} = false \wedge isac_{str} = true$;
  - $VPrE \equiv car_{strd} = true$;
  - $VPrF \equiv car_{strd} = true \wedge isrc_{stn} = false \wedge iskn_{pos} = true \wedge isnul_{dist} = false$;
  - $VPrP \equiv car_{strd} = true \wedge iskn_{pos} = false$;
  - $VPrR \equiv car_{strd} = false \wedge isac_{str} = false$;
- $Post_v$ = $\{VPsH, VPsS, VPsE, VPsF, VPsP, VPsR\}$ where
  - $VPsH \equiv car_{strd} = false \wedge isrc_{stn} = true$;
  - $VPsS \equiv car_{strd} = true$;
  - $VPsE \equiv car_{strd} = false \wedge isac_{str} = false$;
  - $VPsF \equiv car_{strd} = true \wedge isrc_{stn} = false$;
  - $VPsP \equiv car_{strd} = true \wedge iskn_{pos} = true$;
  - $VPsR \equiv isac_{str} = true$.

A possible refinement of this abstract interface of the vehicle component is an automaton that guards all output actions of the abstraction and allows more services (input actions). We can add the requirement that the vehicle functions according to two modes: the *station mode* where the vehicle runs while communicating with stations and the other is the *free-running mode* where the vehicle displace freely without interaction with stations.

A new input action *fstart?* allowing the free-starting of the vehicle. The separation between the starting of the vehicle and its moving off can be interesting as a new refinement requirement. We add the two internal actions *move;* and *stop;* permitting respectively to move off the vehicle and to stop it. The internal action *move;* can be enabled by taking as

precondition the proposition that a vehicle can be started without moving from its place.

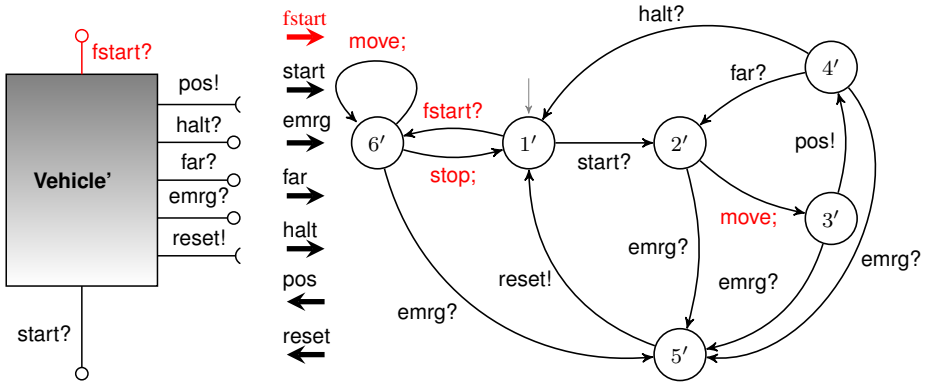

Fig. 5. The refinement automaton A'_v of the Vehicle

The set of variables V is extended by adding two new variables car_{movg} and $frun_{mode}$. The new set of variables $V' = V \cup \{ car_{movg}, frun_{mode} \}$ is used to define pre and post-conditions of the refined automaton A'_v. The first variable car_{movg} is used to make the difference between the act to start the vehicle and the act to move it. The second variable is used to position the two different modes of the vehicle running. The refined automaton *Vehicle'*[3] is showed in the right part of Figure 5. Assuming that atomic formulas are defined now over V', we define the pre and post-conditions of the automaton A'_v as follow

- ${Pre'}_v = \{VPrH', VPrS', VPrE', VPrF', VPrP', VPrR', PrM, PrStp, VPrFS'\}$ where
  - $VPrH' \equiv car_{strd} = true \wedge isnul_{dist} = true$;
  - $VPrS' \equiv iskn_{pos} = false \wedge car_{strd} = false \wedge isac_{str} = true$;
  - $VPrE' \equiv car_{strd} = true$;
  - $VPrF' \equiv car_{strd} = true \wedge isrc_{stn} = false \wedge isnul_{dist} = false$;
  - $VPrP' \equiv car_{strd} = true \wedge iskn_{pos} = false$;
  - $VPrR' \equiv car_{strd} = false \wedge isac_{str} = false$;
  - $PrM \equiv car_{strd} = true \wedge car_{movg} = false$;
  - $PrStp \equiv car_{strd} = true \wedge car_{movg} = true$;
  - $VPrFS' \equiv frun_{mode} = true \wedge car_{strd} = false \wedge isac_{str} = true$;
- ${Post'}_v = \{VPsH', VPsS', VPsE', VPsF', VPsP', VPsR', PsM, PsStp, VPsFS'\}$ where
  - $VPsH' \equiv car_{strd} = false \wedge car_{moving} = false \wedge isrc_{stn} = true$;
  - $VPsS' \equiv car_{strd} = true \wedge car_{movg} = false$;
  - $VPsE' \equiv car_{strd} = false \wedge car_{movg} = false \wedge isac_{str} = false$;
  - $VPsF' \equiv car_{strd} = true \wedge isrc_{stn} = false$;
  - $VPsP' \equiv car_{strd} = true \wedge iskn_{pos} = true$;
  - $VPsR' \equiv isac_{str} = true$;
  - $PsM \equiv car_{strd} = true \wedge car_{movg} = true$;
  - $PsStp \equiv car_{strd} = false \wedge car_{movg} = false$;
  - $VPsFS' \equiv car_{strd} = true \wedge car_{movg} = false$.

[3] The actions of the automaton A'_v are not annotated by pre and post-conditions in order to alleviate the automaton.

According to our new definition of the alternating simulation between states, the reader can remark that, for example, the precondition of the input action *halt?* in the abstraction implies the precondition its correspondent one in the refinement and vice versa for post-conditions ($VPrH \Rightarrow VPrH'$ and $VPsH' \Rightarrow VPsH$). We suppose that the two automata A_v and A_s are compatible. The automaton A_v can be substituted in the product $A_v \otimes A_s$ by the automaton A'_v by applying Theorem 4.1.

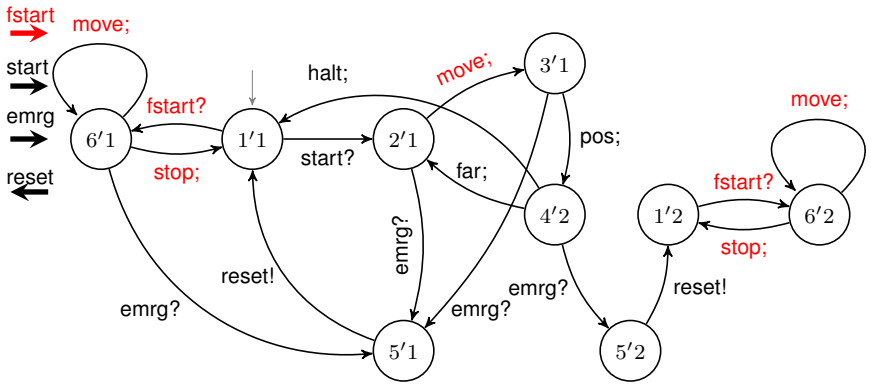

Fig. 6. The composite automaton $A'_v \parallel A_s$

The verification that $A'_v \preceq A_v$ can be easily made using the proposed algorithm in [1] by taking into account the extended third condition of Definition 4.4. The algorithm permits to compute the unique maximal alternating simulation from A'_v to A_v.

Algorithm
Input: Interface automata P and Q extended by pre and post-conditions
Output: The maximal alternating simulation from Q to P.
Let $\succeq_0 = S_P \times S_Q$
repeat
For i $\geq$ 0, define $\succeq_{i+1} \subseteq \succeq_i$ by v $\succeq_{i+1} u$ if $v \succeq_{i+1} u$ and the conditions 1, 2, and 3 of Definition 4.4 hold for v and u where $\succeq = \succeq_i$
until $\succeq_{i+1} = \succeq_i$
return $\succeq_i$

The complexity of the alternating simulation check between our extended IAs is linear like the original model. The unique difference is the consideration of the semantics of actions modeled by relations between pre and post-conditions which can be computed in linear time. Consequently, our approach does not increase the *PTIME-complete* complexity of the classical refinement checking between IAs.

5 Related works

Luca de Alfaro and al. [11] propose ”sociable” interfaces as a different formalism to specify component interfaces from that of interface automata. Their formalism communicates via both actions and shared variables and the synchronization between actions is based on on

two main principles: (i) the first principle is that that same action can label both input and output transitions, and (ii) the second is that global variables can be updated by multiple interfaces. The authors show that the compatibility and the refinement check of sociable interfaces can be made thanks to efficient symbolic algorithms. Their tool called TICC [5] (Tool for Interface Compatibility) implements these algorithms.

Ivana Černá and al. [8] had been founded an automata-based formalism to capture component interactions in hierarchical component-based systems. Their "Component-interaction automata" language represents a flexible model to compose components by respecting the architectural assembly of a system design. Other works are published that extend component-interaction automata by checking LTL temporal properties.

In [3], the authors propose concurrent automata to model component behaviors. The described model consists of a pair: a component signature which captures the static view of a component as depicted in UML 2.0, and a language of component vectors over this signature which describe the behaviour of the component. There is similarity between the notion of a component signature and the static structure of interface automata. The major difference with our model is that whereas in concurrent automata ports are associated to a set of operation calls/signals, thereby ports in our model correspond to individual operations/signals which are furthermore assumed to be sent or received sequentially. So, in our model concurrency is no allowed between ports of the same component.

6 Conclusion and perspectives

In this paper we adapt the alternating refinement relation between interface automata to the semantics of actions. We have improved these automata by pre and post-conditions of component required or provided actions in order to integrate the action semantics in the verification of interface compatibility and interoperability. We strengthen the alternating simulation between states of an abstract version of an interface automaton and its refined version. Equivalence and implications between input and output actions of the refinement and their correspondent ones in the abstraction are established. The compatibility between the refined version of an abstract interface automaton and the environment is preserved in the case when the abstract one is compatible with same environment. These results are applied to the case study of the component-based system of the CyCab car.

As future works, we are interested in implementing a verification tool which takes into account pre and post conditions of actions to check compatibility and refinement between interface automata.

References

[1] L. Alfaro and T. A. Henzinger. Interface automata. *ACM Press, 9th Annual Aymposium of FSE (Foundations of Software Engineering)*, pages 109–120, 2001.

[2] L. Alfaro and T. A. Henzinger. Interface-based design. *NATO Science Series : Mathematics, Physics, and Chemistry, Engineering Theories of Softwareintensive Systems*, 195:83104, 2005.

[3] J. K. F. Bowles and S. Moschoyiannis. Concurrent logic and automata combined: A semantics for components. *Electron. Notes Theor. Comput. Sci.*, 175(2):135–151, 2007.

[4] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequetial processes. *Journal of the ACM (JACM)*, (31):560–599, 1984.

[5] L. D. d. S. M. F. A. L. V. R. B.T. Adler, L. de Alfaro and P. Roy. Ticc: A tool for interface compatibility and composition. *In Computer Aided Verification (CAV06) in Seattle, WA*, pages 59–62.

[6] S. Chouali, H. Mountassir, and S. Mouelhi. An i/o automata based approach to verify component compatibility: application to the cycab car. *LNCS, Springer-Verlag - FESCA of the European joint conference on Theory and Practice of Software (ETAPS'08)*, March 2008.

[7] B. Grard, G. Philippe, M. Herv, and P.-G. Roger. The inria rhône-alpes cycab. *INRIA technical report*, 1466, Avril 1999.

[8] P. V. Ivana Černá and B. Zimmerová. Component-interaction automata modelling language. *Brno, Czech Republic : Faculty of Informatics, Masaryk University, 2006. Technical report FIMU-RS-2006-08*.

[9] Y. Jin, R. Esser, C. Lakos, and J. Janneck. Modular analysis of dataflow process networks. *International conference on Fundamental Approaches to Software Engineering (FASE'03 - ETAPS 2003) - Springer-Verlag*, 2621(31):184–199, 2003.

[10] T. H. L. Alfaro. Interface theories of component-based design. *In Proceedings of the First International Workshop of Embedded Software (EM-SOFT) - LNCS., Springer-Verlag*, 2211:148–165, 2001.

[11] M. F. A. L. P. R. L. de Alfaro, L. Dias Da Silva and M. Sorea. Sociable interfaces. *In FROCOS 2005: 5th International Workshop on Frontiers of Combining Systems, Springer-Verlag*, LNAI 3717, 2005.

[12] N. Lynch and M. Tuttle. Hierarcical correctness proofs for distributed algorithms. *In Proc. 6th ACM Symp. Principles of Distributed Computing*, pages 137–151, 1987.

[13] R. Milner. An algebraic definition of simulation between programs. *In Proc. 2nd International Joint Conference on Artificial Intelligence, The British Computer Society*, pages 481–489, 1971.