Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2010) 3–13
www.elsevier.com/locate/entcs

An I/O Automata-based Approach to Verify Component Compatibility: Application to the CyCab Car

Samir Chouali, Hassan Mountassir, Sebti Mouelhi [1]

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

Abstract

An interesting formal approach to specify component interfaces is interface automata based approach, which is proposed by L. Alfaro and T. Henzinger. These formalisms have the ability to model both the input and output requirements of components system. In this paper, we propose a method to enrich interface automata by the semantics of actions in order to verify components interoperability at the levels of signatures, semantics, and protocol interactions of actions. These interfaces consist of a set of required and offered actions specified by Pre and Post conditions. The verification of the compatibility between interface automata reuse the L.Alfaro and T.Henzinger proposed algorithm and adapt it by taking into account the action semantics. Our approach is illustrated by a case study of the vehicle CyCab.

Keywords: component based systems, interface compatibility, I/O automata.

1 Introduction

Interface formalisms play a central role in the component-based design of many types of systems. They are increasingly used thanks to their ability to describe, in terms of communicating interfaces, how a component of a system can be composed and connected to the others. An interface should describe enough information about the manner of making two or more components working together properly. Several approaches and models based on components have been proposed notably those of Szyperski [10] and Medvidovic [7]. Most of these models specify the components behaviors, the connectors ensuring their communications and the services provided or requested. Assembling components is performed by passing through different levels of abstraction, from the conception of the software architectures ADL until the

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

doi:10.1016/j.entcs.2010.06.002

implementation using platforms like CORBA, Fractal or .NET. The crucial question that arises to the developers is to know if the proposed assembling is correct or not.

In this paper, our interests concern components which are described by interface automata. These interfaces specify action protocols: scheduling calls of component actions. As some related works, we can mention the model in [3] where the protocols are associated to the component connectors. Others works as the ones in [9], the authors proposed a comparison between models at three grades of interoperability using the operation signatures, the interfaces protocols and the quality of service. The protocols in [6] based on transitions systems and concurrency including the reachability analysis. The composition operation is essential to define assembly and check the surety and vivacity properties. The approach in [8] aims to endow the UML components to specify interaction protocols between components. The behavioral description language is based on hierarchical automata inspired from StateCharts. It supports composition and refinement mechanisms of system behaviors. The system properties are specified in temporal logic. In [4], the authors define a component-based model Kmelia with abstract services, which does not take into account the data during the interaction. The behavior described by automata associated to services. This environment uses the tool MEC model-checker to verify the compatibility of components. Other works consider real-time constraints [5]. The idea is to determine the component characteristics and define certain criteria to verify the compatibility of their specifications using the tool Kronos.

The works of L.Alfaro and T.Henzinger [1,2], allows to specify component interfaces by interface automata. These interfaces are specified by automata which are labelled by input, output, and internal actions. The composition of interfaces is achieved by synchronizing actions. Our approach reuse this model and strengthening it by taking into account the action semantics to ensure a more reliable verification of components interoperability. The paper is organized as follows: In section 2, we describe the interface automata as well as the definitions and the algorithm used to verify the compatibility between component interfaces. In section 3, we present our approach to verify the interface compatibility, and we apply the approach to the case study of the vehicle CyCab in section 4. We conclude our work and present perspectives in section 5.

2 Input/Output Automata

The I/O automata are defined by Nancy A.Lynch and Mark.Tutle [12] as a labelled 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.

Definition 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* Σ_A^I, Σ_A^O*and* Σ_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*.*

2.1 Interface automata

The formalism of interface automata are introduced by L.Alfaro and T.Henzinger [1,2], to model component interfaces. These automata are I/O automata where it is not necessary to enable input actions at every state. Every component is described by one interface automaton. In an interface automaton, output actions define the called actions by a component in his environment. They describe the required actions of a component. They are labelled by the symbole "!". Input actions describe the offered actions of a component. They are labelled by the symbole "?". Internal (or hidden) actions are enabled actions inside a component by the component himself. They are labelled by the symbole ";".

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.*

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.

2.1.1 Composition and Compatibility

In this section we present the approach of L.Alfaro a T.Henzinger [1,2] to verify the compatibility of components which are specified by interface automata. The following definition presents the composition of two interface automata.

Definition 2.3 *Two interface automata A_1 and A_2 are* composable *if*

$$\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \cap \Sigma_{A_2} = \Sigma^H_{A_2} \cap \Sigma_{A_1} = \emptyset$$

$Shared(A_1,A_2) = (\Sigma^I_{A_1} \cap \Sigma^O_{A_2}) \cup (\Sigma^I_{A_2} \cap \Sigma^O_{A_1})$ is the set of shared actions between A_1 and A_2. We can now define the product automaton $A_1 \otimes A_2$ properly.

Definition 2.4 *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 $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^H_{A_1 \otimes A_2} = \Sigma^H_{A_1} \cup \Sigma^H_{A_2} \cup Shared(A_1, A_2)$;
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ *if*
  - $a \notin Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  - $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.

The incompatibility between two composable interface automata is due to the existence of some states (s_1,s_2) in the product where one of the automata outputs a shared action sa from the state s_1 which is not accepted as input from the state s_2 or vice versa. These states are called *illegal states*.

Definition 2.5 *Given two composable interface automata A_1 and A_2, the set of illegal states $Illegal(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ of $A_1 \otimes A_2$* is defined by $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} | \exists a \in Shared(A_1, A_2).\ (a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2)) \vee (a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1))\}$.

The reachability of states in $Illegal(A_1, A_2)$ do not implies that A_1 and A_2 are not compatible. The existence of an environment E that produces appropriate inputs for the product $A_1 \otimes A_2$ ensures that illegal states will not be entered and then A_1 and A_2 can be used together. The 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* if and only if their initial state is compatible.

Definition 2.6 *Given two compatible interface automata A_1 and A_2. The composition $A_1 \parallel A_2$ is an interface automaton defined by: (i) $S_{A_1 \parallel A_2} = Comp(A_1,A_2)$, (ii) the initial state is $I_{A_1 \parallel A_2} = I_{A_1 \otimes A_2} \cap Comp(A_1,A_2)$, (iii) $\Sigma_{A_1 \parallel A_2} = \Sigma_{A_1 \otimes A_2}$, and (iv) the set of transitions is $\delta_{A_1 \parallel A_2} = \delta_{A_1 \otimes A_2} \cap (Comp(A_1,A_2) \times \Sigma_{A_1 \parallel A_2} \times Comp(A_1,A_2))$.*

In this approach, the verification of the compatibility between a component C_1 and a component C_2 is obtained by verifying the compatibility between their interface automata A_1 and A_2. The verification steps of the compatibility between A_1 and A_2 are listed below.

Algorithm
Input : interface automata A_1, A_2
Output : $A_1 \parallel A_2$
Algorithm steps:

(i) verify that A_1 and A_2 are composable,

(ii) calculate the product $A_1 \times A_2$,

(iii) calculate the set of illegal in $A_1 \times A_2$,

(iv) calculate the bad states in $A_1 \times A_2$: the states from which the illegal state are reachable by enabling only the internal action or the output actions (one suppose the existence of a helpful environment),

(v) Calculate $A_1 \parallel A_2$ by eliminating from the automaton $A_1 \times A_2$, the illegal state, the bad state, and the unreachable states from the initial states,

(vi) after performing the above step, if the automaton $A_1 \parallel A_2$ is empty then the interface automata A_1, A_2 are not compatible, therefore C_1 and C_2 can not be assembled correctly in any environment. Otherwise A_1 and A_2 are compatible.

The complexity of this approach is in time linear on $|A_1|$ and $|A_2|$ [1].

3 Considering action semantics in the verification of interface automata compatibility

In this section, we present an approach to verify the compatibility between component interfaces based on the I/O automata and the approach of L.Alfaro and T.Henzinger [1].The contribution of our approach compared to the one presented in [1], is the consideration of the action semantics in the component interfaces and in the verification of the component compatibility. In [1], one verify component compatibility by considering only action signatures. We consider, that action signatures are not sufficient to decide on the component compatibility using an approach based on I/O automata.

We propose to annotate transitions in an interface automaton by pre and post conditions of actions. We adapt the compatibility verification algorithm presented in [1], to take into account pre and post of actions. In the following definitions we formalise the adaptations on the L.Alfaro and T.Henzinger approach in order to introduce action semantics in the interface automata.

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 strength-*

ened 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 Σ_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 *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 $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \ \setminus \ Shared(A_1, A_2)$;
- $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \ \setminus \ Shared(A_1, A_2)$;
- $\Sigma^H_{A_1 \otimes A_2} = \Sigma^H_{A_1} \cup \Sigma^H_{A_2} \ \cup \ Shared(A_1, A_2)$;
- $((q_1, q_2), Pre, a, Post, (q_1', q_2')) \in \delta_{A_1 \otimes A_2}$ if
  - $a \notin Shared(A_1, A_2) \ \wedge \ (q_1, Pre_1, a, Post_1, q_1') \in \delta_{A_1} \ \wedge \ q_2 = q_2' \ \wedge \ Pre \equiv Pre_1 \ \wedge \ Post \equiv Post_1$
  - $a \notin Shared(A_1, A_2) \ \wedge \ (q_2, Pre_2, a, Post_2, q_2') \in \delta_{A_2} \ \wedge \ q_1 = q_1' \ \wedge \ Pre \equiv Pre_2 \ \wedge \ Post \equiv Post_2$
  - $a \in Shared(A_1, A_2) \ \wedge \ ((q_1, Pre_1, a, Post_1, q_1') \in \delta_{A_1} \ \wedge \ a \in \Sigma^I_{A_1}) \ \wedge \ ((q_2, Pre_2, a, Post_2, q_2') \in \delta_{A_2} \ \wedge \ a \in \Sigma^O_{A_2}) \ \wedge \ Pre \equiv Pre_2 \ \wedge \ Post \equiv Post_1$ *such that* $Pre_2 \Rightarrow Pre_1 \ \wedge \ Post_1 \Rightarrow Post_2$
  - $a \in Shared(A_1, A_2) \ \wedge \ ((q_1, Pre_1, a, Post_1, q_1') \in \delta_{A_1} \ \wedge \ a \in \Sigma^O_{A_1}) \ \wedge \ ((q_2, Pre_2, a, Post_2, q_2') \in \delta_{A_2} \ \wedge \ a \in \Sigma^I_{A_2}) \ \wedge \ Pre \equiv Pre_1 \ \wedge \ Post \equiv Post_2$ *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 3.3 *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 3.4 *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 $\delta_{A_1\|A_2} = \delta_{A_1\otimes A_2} \cap (Comp(A_1,A_2) \times Pre_{A_1\otimes A_2} \times \Sigma_{A_1\|A_2} \times Post_{A_1\otimes A_2} \times Comp(A_1,A_2))$.*

In this approach, the verification of the compatibility between a component C_1 and a component C_2 is obtained by verifying the compatibility between their interface automata A_1 and A_2.Therefore, one verify if there is a helpful environment where it is possible to assemble correctly the components C_1 and C_2. So, one suppose the existence of such environment which accepts all the output actions of the automaton of the product $A_1 \times A_2$, and which do no not call any input actions in $A_1 \times A_2$.

In order to verify the compatibility between two components C_1 and C_2, it is necessary to verify of the compatibility between their respective interface automata A_1 and A_2. So, one verify if there is a helpful environment (other components) where it is possible to assemble correctly the components C_1 and C_2. So, one suppose the existence of such environment which accepts all the output actions of the automaton of the product $A_1 \times A_2$, and which do no not call any input actions in $A_1 \times A_2$.

Remark 3.5 The verification steps in this approach are the same as the ones presented in the section 2.1.1(the same steps as in [1]). However, in our approach we consider the action semantics in :

- the interface automata definition,

- the product of two interface automata,
- the definition of the illegal states.

Consequently, our approach does not increase the linear complexity of the verification algorithm.

4 The CyCab case study

Several approaches have been proposed to study the concept of CyCab [11]. The CyCab 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.

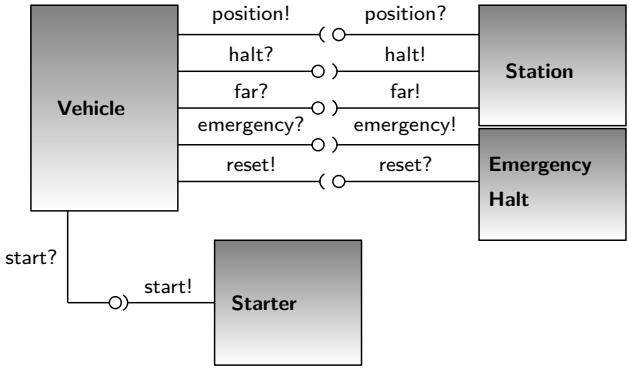

Fig. 1. A UML-like model of the CyCab components.

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 1 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.

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.

[2] The component diagram showed in Figure 1 do not respect exactly the UML 2 notation. It is simply used

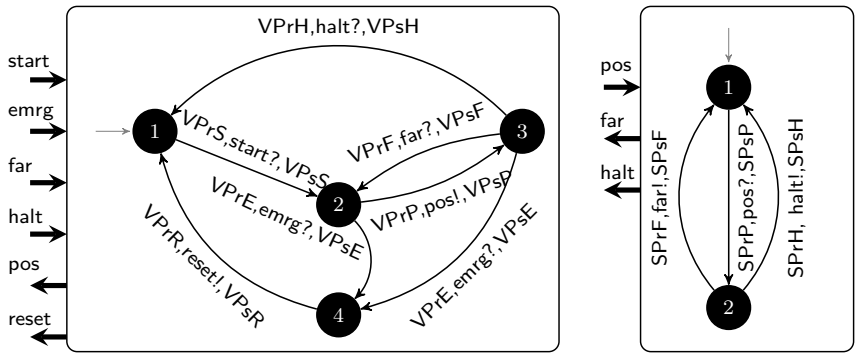

Fig. 2. The IAs A_v and A_s of the Vehicle and the Station

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
  - $VPsH \equiv car_{strd} = false \wedge isrc_{stn} = true$;
  - $VPsS \equiv car_{strd} = true$;
  - $VPsE \equiv car_{strd} = false \wedge isac_{str} = false$;
  - $VPsF \equiv car_{strd} = true \wedge isrc_{stn} = false$;
  - $VPsP \equiv car_{strd} = true \wedge iskn_{pos} = true$;
  - $VPsR \equiv isac_{str} = true$.

The automaton A_s is given by the tuple $\langle Q_s, I_s, \Sigma_s^O, \Sigma_s^H, Pre_s, Post_s, \delta_s \rangle$ where

- $Pre_s = \{SPrP, SPrH, SPrF\}$ where:
  - $SPrP \equiv car_{strd} = true \wedge iskn_{pos} = false$;
  - $SPrH \equiv car_{strd} = true \wedge isrc_{stn} = false \wedge iskn_{pos} = true$;
  - $SPrF \equiv car_{strd} = true \wedge isrc_{stn} = false \wedge iskn_{pos} = true$;
- $Post_s = \{SPsP, SPsH, SPsF\}$ where:
  - $SPsP \equiv car_{strd} = true \wedge iskn_{pos} = true$;
  - $SPsH \equiv car_{strd} = false \wedge isrc_{stn} = true \wedge isac_{str} = true$;
  - $SPsF \equiv car_{strd} = true \wedge isrc_{stn} = false$;

to clarify the CyCab system

The composition of the two interfaces A_v and A_s is possible because the set $Shared(A_v,A_s) = \{position, halt, far\} \neq \varnothing$ and they are composable. The synchronized product between them as shown in the figure 3, have as pre and post conditions of operations $Pre_{v\otimes s} = \{PrS, PrP, PrH, PrF, PrE, PrR\}$ and $Post_{v\otimes s} = \{PsS, PsP, PsH, PsF, PsE, PsR\}$ where

- $PrP \equiv VPrP$ if $VPrP \Rightarrow SPrP$, $PrH \equiv SPrH$ if $SPrH \Rightarrow VPrH$, $PrF \equiv SPrF$ if $SPrF \Rightarrow VPrF$, $PrS \equiv VPrS$, $PrE \equiv VPrE$, and $PrR \equiv VPrR$;
- $PsP \equiv SPsP$ if $SPsP \Rightarrow VPsP$, $PsH \equiv VPsH$ if $VPsH \Rightarrow SPsH$, $PsF \equiv VPsF$ if $VPsF \Rightarrow SPsF$, $PrS \equiv VPsS$, $PrE \equiv VPsE$, and $PrR \equiv VPsR$.

We compute the synchronized product automaton

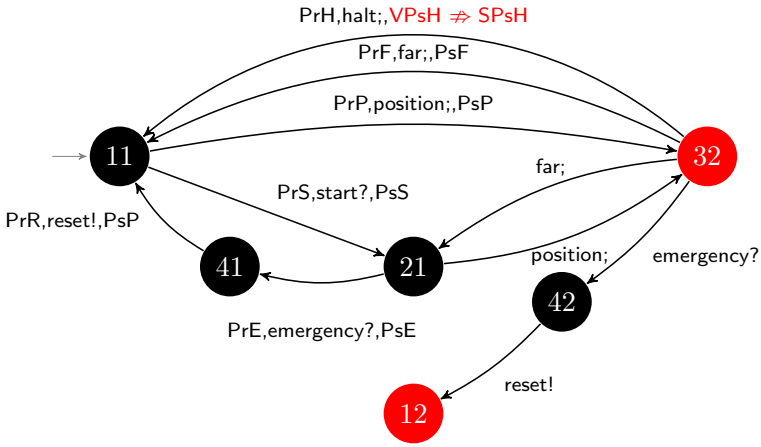

Fig. 3. Illegal states in the product Vehicle⊗Station

After computing the set of illegal states in the product, we obtain the set $Illegal(A_v,A_s) = \{32, 12\}$. The state 32 is an illegal state because from the state 3 in the automaton *Vehicle*, the postcondition of the input shared action, *halt?*, do not imply the postcondition of the corresponding output action, *halt!*, from the state 2 in the component *Station* ($VPsH \not\Rightarrow SPsH$). In fact, the component *Vehicle* inputs the actions *halt* which provokes strictly the vehicle halt, while the component *Station* solicits an action *halt* which provokes the the vehicle halt and the station reach.

Next, we compute by performing a backward reachability analysis from *Illegal* states which traverses only internal and output steps, all states thus reachable. The resulting set is $\{11, 21, 42\}$ and so the set of unreachable states is $\{41\}$. Finally, we remove all incompatible and unreachable states $\{11, 21, 42, 32, 12, 41\}$ and from the product automaton to obtain their composite automaton $A_v \parallel A_s$. The set of remaining states is empty then, the two interfaces *Vehicle* and *Station* are not compatible.

Remark 4.1 If we apply the approach proposed by L.Alfaro and T.Henzinger [1] on the same use case, we can detect a compatibility between the components *Vehicle* and *Station*, which is contrasted by considering the semantics of the action *halt*.

5 Conclusion and perspectives

The proposed work in this paper is a methodology to analyze the compatibility between component interfaces. We are inspired by the method proposed by L. Alfaro and T. Henzinger where interfaces are described by protocols modeled by I/O automata . We improved these automata by pre and post conditions of component actions in order to handle the action semantics in the verification of interface compatibility. This verification is made up of two steps. The first determines if two components are composables or not by checking some conditions on the actions feasibility by considering their semantics. The second aims is to detect inconsistencies between the sequences of action calls given by communicating protocols. This phase is obtained by considering the synchronized product of interface automata. These results are applied on the case study of the autonomous vehicle CyCab.

In this context, we are interesting for two research directions. The first consists in implementing a verification tool which takes into account pre and post conditions of actions to check compatibility between interfaces. The second concerns composite components and their refinement to define under which conditions a set of assembled components satisfies constraints of the composite component.

References

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

[2] L. Alfaro and T. A. Henzinger. *Interface-based design*. Engineering Theories of Softwareintensive Systems (M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare, eds.), NATO Science Series : Mathematics, Physics, and Chemistry, 195 :83-104, 2005.

[3] Robert Allen and David Garlan. *A formal basis for architectural connection*. ACM Transactions on Software Engineering and Methodology, 6(3): 213-249, July 1997.

[4] Pascal Andr, Gilles Ardourel, and Christian Attiogb. *Behavioural Verification of Service Composition*. In ICSOCWorkshop on Engineering Service Compositions,WESC05, pages 77-84, Amsterdam, The Netherlands, 2005. IBM Research Report RC 23821.

[5] J.-P. Etienne and S. Bouzefrane. *Vers une approche par composants pour la modlisation dapplications temps rel*. In (MOSIM06) 6me Confrence Francophone de Modlisation et Simulation, pages 1-10, Rabat, 2006. Lavoisier.

[6] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. *Behaviour analysis of software architectures*. In WICSA1 : Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), pages 35-50, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[7] Nenad Medvidovic and Richard N. Taylor. *A classification and comparison framework for software architecture description languages*. Software Engineering, 26(1): 70-93, 2000.

[8] S. Moisan, A. Ressouche, and J. Rigault. *Behavioral substitutability in component frameworks : A formal approach*, 2003.

[9] Becker Steffen, Overhage Sven, and Reussner Ralf. *Classifying software component interoperability errors to support component adaption*. In Crnkovic Ivica, Stafford Judith, Schmidt Heinz, and Wallnau Kurt, editors, Component Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, Proceedings, pages 68-83. Springer, 2004.

[10] C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.

[11] Baille Grard, Garnier Philippe, Mathieu Herv and Pissard-Gibollet Roger. *The INRIA Rhône-Alpes Cycab*. INRIA technical report, Avril 1999.

[12] N. Lynch and M. Tuttle, Hierarchical Correctness Proofs for Distributed Algorithms, 6th ACM Symp on Principles of Distributed Computing,137-151, ACM Press,1987.