Assembly of components based on interface automata and UML component model

Samir Chouali *, Sebti Mouelhi *, Hassan Mountassir *

*Laboratoire d'Informatique de l'université de Franche-Comté, LIFC
{schouali, smouelhi, hmountassir }@lifc.univ-fcomte.fr

Abstract. We propose an approach which combines component UML model and interface automata in order to assemble components and to verify their interoperability. We specify component based system architecture with component UML model, and component interfaces with interface automata. Interface automata is a common Input Output (I/O) automata-based formalism intended to specify the signature and the protocol level of component interfaces. We improve interface automata approach by component UML model, in order to consider system architecture, in component composition and interoperability verification methods. Therefore, we handle in interface automata, the connection between components, and the hierarchical connections between composite components and their subcomponents.

1 Introduction

Component based systems are made up of collection of interacting entities, called components. The idea in component based software engineering (CBSE) is to develop software applications not from scratch but by assembling various library components, Szyperski (1999); Heineman et Councill (2001). This development approach allows, to extend component based systems via plug and play components, and to reuse components. Therefore one saves on development costs and time.

A component is a unit of composition with contractually specified interfaces and explicit dependencies, Szyperski (1999). An interface describes the services offered and required by a component without disclosing the component implementation. It is the only access to the information of a component. Interfaces may describe component infomation at signature (method names and their types), behaviour or protocol (scheduling of method calls), semantic (method semantics), and quality of services levels. The success of applying the component based approach depends on the interoperability (we say also component compatibility) of the connected components. The interoperability can be defined as the ability of two or more entities to communicate and cooperate despite differences in their implementation language, the execution environment, or the model abstraction, Konstantas (1995); Wegner (1996). The interoperability holds between components when their interfaces are compatible.

In this paper, we focus on assembling components described by interface automata. The interface automata based approach was proposed by L.Alfaro and T.Henzinger, Alfaro et Henzinger (2001, 2005); Alfaro et al. (2002). They have proposed to specify component interfaces with automata, which are labelled by input, output, and internal actions. These automata allow to describe component information at signature and protocol levels. An interesting verification approach was also proposed to detect incompatibilities at signature and protocol levels between two component interfaces. The verification is based on the composition of interfaces, which is achieved by synchronizing component actions.

The essential drawback of the interface automata approach is that, it is unable to accept as an input a set of interface automata, more than two, associated to all components composing a component based sys-

tem, and also consider system architecture. In fact, interface automata are proposed to specify component behaviour only and therefore are unable to describe, the connection between primitives components and composites (composed of others components), and the hierarchical connections between composites and their subcomponents, which also influences component behaviors. Therefore, we propose to enrich this approach by exploiting the component UML model which specifies the component based system architecture. The point we want to address in our paper is to show how to combine UML and interface automata to verify interoperability in component based systems.

The paper is organized as follows : In section 2, we describe the approach based on interface automata to verify component iteroperability. In section 3, we describe our approach which combines UML component model an interface automata in order to assemble components and to verify their interoperability, according to system architecture. The approach was illustrated on the vehicle CyCab use case. Related works are presented in section 4. We conclude our work and present perspectives in section 5.

2 Interface Automata

Interface automata (IAs) have been defined in Alfaro et Henzinger (2001), to model the temporal behavior of software and hardware component interfaces. These models are non-input-enabled[1] like I/O automata in Lynch et Tuttle (1987) . Every component interface is described by one interface automaton where input actions (assigned by ?) 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 (assigned by !) are used to model method calls, message transmissions via communication channels, and exceptions that occur during the method execution. Hidden actions (assigned by ;) describe the local operations of the component.

Definition 1 (Interface Automata). *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* ;
- *a subset of initial states $I_A \subseteq S_A$. Its cardinality card$(I_A) \geq 1$ and if $I_A = \emptyset$ 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.*

The composition operation 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 2 (Composition Condition). *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 actions between A_1 and A_2. We can now define the product automaton $A_1 \otimes A_2$ properly.

Definition 3 (Synchronized product). *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 $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma_{A_1 \otimes A_2}^I = (\Sigma_{A_1}^I \cup \Sigma_{A_2}^I) \setminus Shared(A_1, A_2)$;
- $\Sigma_{A_1 \otimes A_2}^O = (\Sigma_{A_1}^O \cup \Sigma_{A_2}^O) \setminus Shared(A_1, A_2)$;
- $\Sigma_{A_1 \otimes A_2}^H = \Sigma_{A_1}^H \cup \Sigma_{A_2}^H \cup Shared(A_1, A_2)$;

[1] input actions are not enabled at every state of one automaton

- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ *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 4 (Illegal States). *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 5 (Composition). *Given two compatible 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) $\times \Sigma_{A_1 \| A_2} \times$ 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 :**

1. verify that A_1 and A_2 are composable,
2. calculate the product $A_1 \times A_2$,
3. calculate the set of illegal in $A_1 \times A_2$,
4. 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),
5. 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,
6. 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 $|A_2|$ Alfaro et Henzinger (2001).

3 Assembling components according to their IAs specification and an UML-based architecture

UML 2.0 offers concepts to well-design CBS architectures and interactions between their components. These concepts are based on required and provided interfaces to model interactions between components without giving any information about the protocol specification of components. UML 2.0 specifies a component as reusable modular unit, which interacts with its environment. Interfaces describe component operations annotated by their pre and post-conditions and they may also describe component data models.

In this paper, we specify component interfaces by interface automata in order to strengthen UML component models by checking formally the component compatibility at the protocol level. We propose a fully algorithmic approach that exploits a component UML model to improve interface automata approach, in order to verify component composition.

3.1 Improvement of interface automata approach based on component UML model

We specify the UML architecture as a graph where nodes are the components of the system and edges represent both the hierarchical relations between composite components and their subcomponents and the connections between subcomponents within a composite component in the model. The nodes of the graph can be seen as tree if we consider only hierarchical relations. For an UML architecture *M*, we denote by $\mathcal{A}_M$ all the interface automata specifications of all the primitive components constructing the UML architecture *M* and by $\mathcal{C}_M$ all the (composite and primitive) components constructing *M*. A_C is the interface automaton of a component *C*.

Definition 6 (Graph Representation of an UML Architecture). *A* Graph Representation $G_M = \langle N_{G_M}, Cp_{G_M}, Cn_{G_M} \rangle$ *of an UML architecture M, consists of*

- *a finite set* N_{G_M} *of nodes representing* $\mathcal{C}_M$;
- *a finite set* Cp_{G_M} *of edges representing the relations between the nodes representing composite components and their subcomponents* ;
- *a finite set* Cn_{G_M} *of edges representing the connections between the nodes representing subcomponents within a same component*.

By traversing this graph, we propose an algorithm that can at the same time, (i) check the compatibility between connected components at the level of a subcomponent and (ii) construct progressively the IAs specification of the final system. In the case where incompatibilities is detected at most between two interfaces, it returns an empty interface automata. In order to specify this algorithm, We propose to improve interface automata of a each component C, with the set of components connect to C(neighbors of C). This set is deducted from the graph defining component UML model.

The following definitions show the adaptation to introduce in interface automata approach in order to handle the component UML model.

Definition 7 (Improved interface automata). An interface automaton A, which describes a component C_A, is defined by a tuple, $\langle S_A, \mathrm{I}_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A, U_A \rangle$, such that :

- a finite set S_A of states ;
- a subset of initial states $I_A \subseteq S_A$. Its cardinality card$(I_A) \geq 1$ and if $I_A = \emptyset$ 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.
- U_A is the set of the components connected to the component C_A, according to the UML model of the component based system architecture

We adapt also the synchronized product of two interface automata.

Definition 8 (Improved sychronized product). Let A_1 and A_2 be two composable interface automata corresponding respectively to the components C_{A_1} and C_{A_2}. The product $A_1 \otimes A_2$ is defined by $\langle S_{A_1 \otimes A_2}$, $\mathrm{I}_{A_1 \otimes A_2}$, $\Sigma^I_{A_1 \otimes A_2}$, $\Sigma^O_{A_1 \otimes A_2}$, $\Sigma^H_{A_1 \otimes A_2}$, $\delta_{A_1 \otimes A_2}$, $U_{A_1 \otimes A_2}$ $\rangle$, such that :

- $S_{A_1 \otimes A_2}$ = $S_{A_1} \times S_{A_2}$ 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
  - $a \notin Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  - $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.
- $U_{A_1 \otimes A_2}$= $\{U_{A_1} \cup U_{A_2}\} \setminus \{C_{A_1}, C_{A_2}\}$.

When we compose two components C_1, and C_2, by their respective interface automata A_1 and A_2, the set $U_{A_1 \| A_2}$ corresponding the the interface automata, $A_{1\|2}$, which describes the composite $C_1 \parallel C_2$, is defined by : $U_{A_1 \otimes A_2}$. And, for each set U_{A_i}, belonging to an interface automaton A_i, we replace each occurrence of C_1 or C_2 by $C_1 \parallel C_2$.

Remark : The other steps to verify the compatibility between two components, with the interface automata approach, are not affected by the adaptation. They are the same as those explained in the section 3 : after calculating the synchronized product, we calculate the composition by eliminating the illegal states.

3.2 CyCab use case :

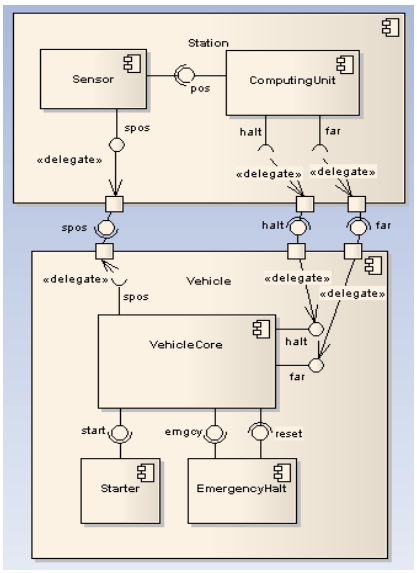

FIG. 1 – *The UML 2.0 architecture of a basic CyCab CBS.*

As an example, we consider a CyCab car component-based system (in Baille Gérard et Pissard-Gibollet (1999)). The CyCab car is a new electrical means of transportation conceived essentially for free-standing transport services allowing users to displace through pre-installed set of stations.

In Figure 1, we describe the CyCab architecture with UML component model. The CyCab system is composed of two main composites : Station and Vehicle. The *Vehicle* sends signals *spos* ! to inform the upcoming station about its positions and it receives as consequence signals (*far* ! or *halt* !) to know if it steels far from the station or not. The two automata *Sensor* (*Ss*) and *ComputingUnit* (*Cu*) are the subcomponents of the station. The *sensor* detects a position signal sent from the vehicle and converts it to geographic coordinates (*pos* !) which will be used by the *ComputingUnit* to compute the distance between the vehicle and the station and decide if they steel far from each other or not. The vehicle is composed from three primitive components : the *VehicleCore* (*Vc*), the *Starter* (*Sr*), and the embedded *EmergencyHalt* (*Eh*) device.

We depict so the CyCab UML architecture as shown in Figure 2. The continuous edges represent the hierarchical relations between composite components and their subcomponents. The dashed edges represent the connections between components at the level of composite components. Two components are connected if and only if there is at least one interaction between their interfaces.

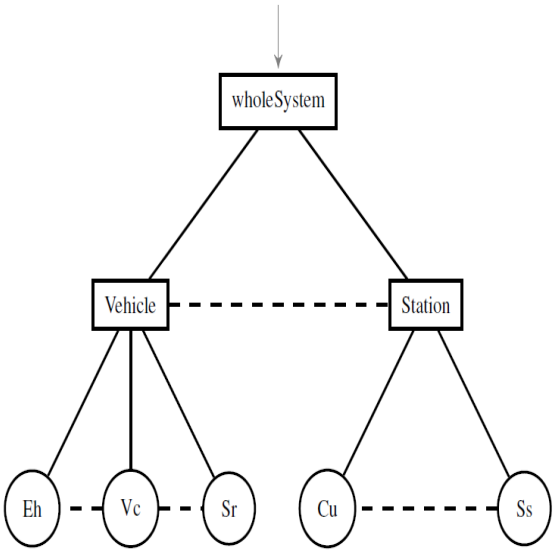

FIG. 2 – *The graph representation of the CyCab car system.*

3.3 The algorithm :

We present the algorithm allowing the component composition and the verification of component interoperability, based on improved interface automata approach and component UML Model.

This algorithm is based on traversing the graph which specifies system architecture, in order to check the compatibility between connected components at the level of a subcomponent, and to construct progressively the IAs specification of the final system. In the case where incompatibility is detected at most between two interfaces, it returns an empty interface automata.

The algorithm, is based on DFS (depth first search) algorithm, and we explain this algorithm by three steps.

We consider a graph G_M which specifies an UML component model M.

3.3.1 First Step

This step consists in traversing the graph in order to :

– find the sets U_{A_i} (neighbors of each components in a system architecture) of each interface automata A_i corresponding to a primitives components in a system. These sets will be used in the second step.

– find the sets of child nodes for each node associated to composite components. These sets will be used in the third step.

3.3.2 Second Step (see algorithm 1)

In this step, we define a function Compos(C_1,...,C_n) which accepts as input a set of components and as output, the interface automata of the resulting composite or the empty set (empty interface automaton) if there is incompatibility between two components. The function is described by **Algorithm 1**.

Algorithm 1: Compos

Input: *A set of components C* $\subseteq \mathcal{C}_M$ *for a given UML architecture M.*

Output: *The IA of the composition of all c* $\in \mathcal{C}$.

begin
  $c_1 \leftarrow \Pi(C)$; /*c_1 is selected form C*/
  repeat
    $c_2 \leftarrow \Pi(U_{c_1})$; /*c_2 is selected form the set of neighbors of c_1, U_{c_1} */
    Verify the compatibility between, A_{c_1} and A_{c_2} , with the interface automata approach (see section 2 and 3.1)
    if A_{c_1} *and* A_{c_2} *are compatible* **then**
      $C \leftarrow \{C \backslash \{c_1, c_2\}\} \cup \{c_1 \parallel c_2\}$;
      $A_{c_1} \leftarrow A_{c_1} \parallel A_{c_2}$;
    else
      return *emptyIA* ;
  until *card*(C)=1 ;
  return A_{c_1} ;
end

3.3.3 Third Step (see algorithm2)

In this step, we define the main function, MainCompos(root), which traverses the graph recursively and calculates the composition of interface automata by calling the function defined in algorithm 1. This function, accepts as input, a node of the graph G_M corresponding to a composite. At the initial state, this function accepts the node corresponding to the whole system (the root of the graph). It returns as output the interface automata of the whole system or the empty set if there is incompatibility between components.

We note by $SetChild$ the set of child nodes corresponding to subcomponents in a composite C_i. We note by $SettoCompose$, the set of components to compose by the function $Compos()$. At the initial state, we associate to each leaf node in G_M, corresponding to a primitive components C, an interface A_C. However, nodes corresponding to composites are not associated to interface automata. The function is described by **Algorithm 2**.

Algorithm 2: MainCompos

Input: the node $root$ corresponding to the whole system in the graph G_M.

Output: *The interface automaton of the whole system.*

begin
  $SetChild$={the set of child nodes of the node $root$} /* this set is calculated in the first step */ **while** *($SetChild$ is not empty)* **do**
    Let N a node in $SetChild$, and corresponding to the component C ;
    if *(the interface automaton associated to N exists, because N is a leaf node or it is a node corresponding to composite which is associated to an interface automaton by this function)* **then**
      $SettoCompose = SettoCompose \cup \{C\}$;
      $SetChild = SetChild \setminus \{N\}$
    else
      $IA_C = MainCompos(N)$ /* calculate the interface automaton IA_C corresponding to N*/
  $IA_C = Compos(C_1, ..C_n)$, where $SettoCompose = \{C_1, ..., C_n\}$; /* call to the function $Compos$ in algorithm 1 in order to compose the components in the set $SettoCompose$*/
  return IA_c ;
end

Remark : Since the proposed algorithms are based on traversing graph G_M, corresponding to system architecture M, with the complexity in time linear on the set of components composing the system, then our method does not augment the complexity of the interface automata approach, which is time linear of the the size of the composed interface automata.

3.3.4 Illustration on the CyCab

Lets take our previous example, the interface automata of the primitive components of the CyCab car system are presented in Figure 3 and Figure 4. The algorithm starts first by constructing the *Station* and the *Vehicle* composite components. from their subcomponents.

The algorithm starts first by constructing the *Station* and the *Vehicle* composite components from their subcomponents. The reader can easily verify that the two interface automata *Station* and *Vehicle* are not empty. Then, We construct the whole composite system representing the communication between the *Vehicle* and the *Station*. Figure 5 represents the IAs specification of the composite component *Vehicle*.

4 Related works

As some related works, we can mention the model in Allen et Garlan (1997) where the protocols are associated to the component connectors. In Brim et al. (2006) the authors proposed a new approach to component interaction specification and verification process which combines the advantages of architecture description languages and formal verification oriented model. So, they proposed component interaction automata to specify component interfaces and verify their compatibility. In Poizat et Royer (2006) the authors proposed an ADL based the Korrigan language which enables to describe the component based systems architectures formally. This ADL supports : integration of fully formal behaviors and data types, expressive component composition mechanisms through the use of modal logic, specification readability through graphical notations, and dedicated architectural analysis techniques.

Others works as the ones in Steffen et al. (2004), 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 Magee et al. (1999) based on transitions systems and concurrency including

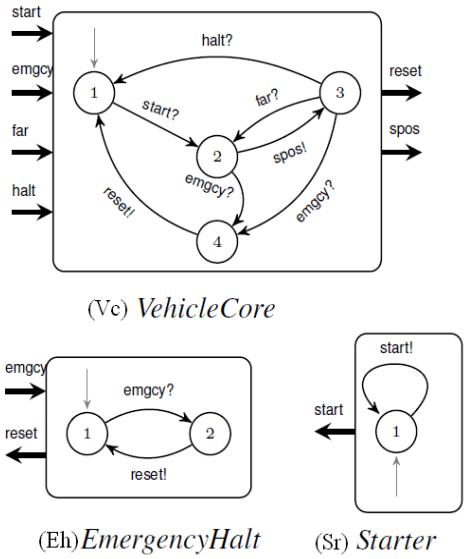

FIG. 3 – *The interface automata of the Vehicule subcomponent*

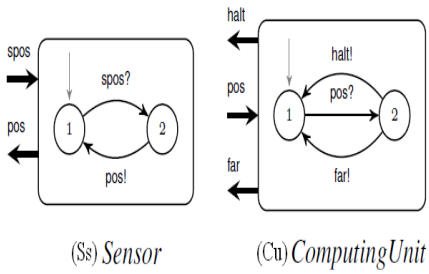

FIG. 4 – *The interface automata of the Station subcomponent*

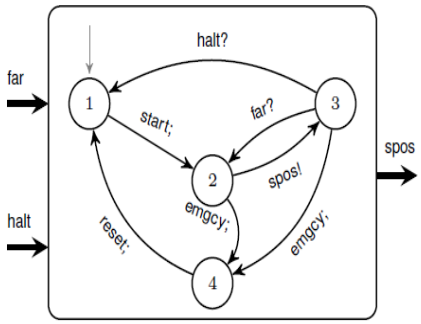

FIG. 5 – *The interface automaton of the Vehicule composite*

the reachability analysis. The composition operation is essential to define assembly and check the surety and vivacity properties. The approach in Moisan et al. (2003) 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 Chouali et al. (2006), the authors proposed to specify component interface and to verify their compatibility with B method. However component protocols are not considered in the interfaces. In André et al. (2005), 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 Etienne et Bouzefrane (2006). The idea is to determine the component characteristics and define certain criteria to verify the compatibility of their specifications using the tool Kronos.

The contribution of our approach, compared the related works, is the specification of component interfaces with interface automata (which is an interesting approach to express component behaviors) and the expression of a component based system architecture in the interface automata method, thanks to the UML component model which specifies the architecture. So, we exploit UML component model and the interface automata method to verify the component composition.

5 Conclusion

In this paper, we present a new formal approach to assemble components and to verify their interoperability, according to a system architecture, specified by component UML model. This approach is based on interface automata method to specify component interfaces and to verify interface compatibility. We have improved this approach by exploiting component UML model, in order to specify , connection between components and composites, and hierarchical connection in composites. Component UML model

corresponding to system architecture is specified formally by a graph, where nodes correspond to components and edges specify connections between components. From this graph, we deduce information to improve interface automata approach in order to verify interface compatibility. So, we have proposed an algorithm to assembles components and composites, based on a both, system architecture and component interface automata.

Actually, we develop a tool in order to implement the proposed approach. This tool will allow to specify component based system architecture with component UML model, and component interfaces with enhanced interface automata. As future work, we plan to specify component interfaces, only with UML models, because UML is more expressive for most of people, and then propose an approach to translate this models automatically to interfaces automata in order to verify component interoperability.

References

Alfaro, L. et T. A. Henzinger (2001). Interface automata. In *9 th Annual Aymposium on Foundations of Software Engineering, FSE*, pp. 109–120. ACM Press.

Alfaro, L. et T. A. Henzinger (2005). Interface-based design. In *Engineering Theories of Software-intensive Systems*, Volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pp. 83–104. Springer: M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare.

Alfaro, L., T. A. Henzinger, et M. Stoelinga (2002). Timed interfaces. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, London, UK, pp. 108–122. Springer-Verlag.

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

André, P., G. Ardourel, et C. Attiogbé (2005). Behavioural Verification of Service Composition. In *ICSOC Workshop on Engineering Service Compositions, WESC'05*, Amsterdam, The Netherlands, pp. 77–84. IBM Research Report RC23821.

Baille Gérard, Garnier Philippe, M. H. et Pissard-Gibollet (1999). *The INRIA Rhône-Alpes CyCab*. Technical report, INRIA. Describes the package natbib.

Brim, L., I. Černá, P. Vařeková, et B. Zimmerova (2006). Component-interaction automata as a verification-oriented component-based system specification. *SIGSOFT Softw. Eng. Notes 31*(2), 4.

Chouali, S., M. Heisel, et J. Souquières (2006). Proving component interoperability with b refinement. *Electr. Notes Theor. Comput. Sci. 160*, 157–172.

Etienne, J.-P. et S. Bouzefrane (2006). Vers une approche par composants pour la modélisation d'applications temps réel. In *(MOSIM'06) 6ème Conférence Francophone de Modélisation et Simulation*, Rabat, pp. 1–10. Lavoisier.

Heineman, G. et W. Councill (2001). *Component Based Software Engineering*. Addison Wesley.

Konstantas, D. (1995). Interoperation of object oriented application. In O. Nierstrasz et D. Tsichritzis (Eds.), *Object-Oriented Software Composition*, pp. 69–95. Prentice Hall.

Lynch, N. et M. Tuttle (1987). Hierarchical correctness proofs for distributed algorithms. In *6th ACM Symp. on Principles of Distributed Computing*, pp. 137–151. ACM Press.

Magee, J., J. Kramer, et D. Giannakopoulou (1999). Behaviour analysis of software architectures. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, Deventer, The Netherlands, The Netherlands, pp. 35–50. Kluwer, B.V.

Moisan, S., A. Ressouche, et J. Rigault (2003). Behavioral substitutability in component frameworks: A formal approach.

Poizat, P. et J.-C. Royer (2006). A formal architectural description language based on symbolic transition systems and temporal logic. *J. UCS 12*(12), 1741–1782.

Steffen, B., O. Sven, et R. Ralf (2004). Classifying software component interoperability errors to support component adaption. In C. Ivica, S. Judith, S. Heinz, et W. Kurt (Eds.), *Component Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, pp. 68–83. Springer.

Szyperski, C. (1999). *Component Software*. ACM Press, Addison-Wesley.

Wegner, P. (1996). Interoperability. *ACM Computing Survey 28*(1), 285–287.

Résumé

Donner la traduction anglaise du résumé dans le préambule avec la commande `\summary{Your abstract ...}`