Assembling Components using SysML with Non-Functional Requirements

Samir Chouali, Ahmed Hammad, Hassan Mountassir [1]

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

Abstract

Non-functional requirements of component based systems are important as their functional requirements, therefore they must be considered in components assembly. These properties are beforehand specified with SysML requirement diagram. We specify component based system architecture with SysML block definition diagram, and component behaviors with sequence diagrams. We propose to specify formally component interfaces with interface automata, obtained from requirement and sequence diagrams. In this formalism, transitions are annotated with costs to specify non-functional property. The compatibility between components is performed by synchronizing their interface automata. The approach is explained with the example of the electric car CyCab, where the costs are associated to energy consumption of component actions. Our approach verifies whether, a set of components, when composed according to the system architecture, achieve their tasks by respecting their non-functional requirements.

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

1 Introduction

The idea in component based software engineering (CBSE) is to develop software applications not from scratch but by assembling various library components. A component is a unit of composition with contractually specified interfaces and explicit dependencies, [14]. An interface describes the offered and required services without disclosing the component implementation. It is the only access to the information of a component. Interfaces may describe component information at signature (method names and their types), behaviour or protocol (scheduling of method calls), semantic (pre and post conditions), and quality of services levels. The success of applying the component based approach depends on the interoperability of the connected components. The interoperability can be defined as the ability of two or more entities to communicate and cooperate without anomalies, [16]. The interoperability holds between components when their interfaces are compatible.

The SysML language is a UML profile, that is a language for documenting and graphically specify all aspects of a system consisting of hardware and/or software

[1] Email: {schouali, ahammad, hmountassir}@lifc.univ-fcomte.fr

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

blocks. SysML enjoys unprecedented popularity both in industry and academia. It is in the form of graphical models, used to harmonize the different actors contributing to the achievement of a system, and to ensure consistency and quality of design. It is a well suited language to model embedded systems.

In this paper, we focus on assembling components in SysML specifications and we model components interaction by interface automata. The interface automata based approach was proposed by L.Alfaro and T.Henzinger, [7,8]. They have specified 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 shared actions.

The first essential drawback of the interface automata approach is to consider non-functional properties in component interface specification and in the verification of components interoperability in order to obtain a reliable assembly. In our work we propose a solution by specifying non functional requirements of a system with SysML requirement diagram, and also specifying formally this diagram by associating a cost (an integer positive value) for action to describe some non-functional property as energy consumption or time. Also we propose a verification approach that takes into account this property in components assembly.

The second drawback is that the interface automata approach is unable to accept as an input a set of interface automata, more than two, associated to all components composing a component based system, and also consider system architecture. 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 exploit the SysML block definition diagram which specifies the component based system architecture.

The point we want to address in our paper is to show how to combine SysML and interface automata to verify interoperability in component based systems, by considering non-functional properties. These properties are obtained by the formalization of the requirements specified in the SysML requirement diagram

The paper is organized as follows. In section 2, we give an overview of SysML language. In section 3, we present the example of the CyCab vehicle, and we specify semi-formally system architecture, component behaviors, and non functional requirements, with SysML diagrams. In section 4, we describe a methodology to extract a tree from SysML architecture of a component based system. The obtained tree is used to verify the interoperability between components. Section 4 describes the proposed approach combining SysML and interface automata in order to assemble components and to verify their interoperability by considering system architecture and non-functional requirements. Related works are described in section 6. We conclude our work and trace some perspectives in section 7.

2 SysMl Presentation

SysML is a modeling language for systems engineering. This covers complex systems which include a broad range of heterogeneous domains, in particular hardware and software. *SysML* has been proposed by the Object Management Group (OMG) [1], together with the International Council on Systems Engineering (INCOSE) [2] and the AP233 consortium [3] with the aim to define a general purpose modeling language for systems engineering. It is based on the actual standard for software engineering, the Unified Modeling Language (*UML*) [4] version 2.0, with some extensions and it was developed as a response to the request for proposal (RFP) issued by the OMG in March 2003 [5] and adopted as a standard in May 2006 [6]. *SysML* is a modeling language for representing systems and product architectures, as well as their behavior and structure. It adapts to systems engineering standard modeling techniques from software development, and supports the specification, design, analysis, verification and validation of a broad range of complex systems.

SysML is the first formal UML profile dedicated to the specification of professional engineering systems. It has been developed during many years but has only recently been fully agreed and standardized. *SysML* has meanwhile evolved over several major iterations, including two separated proposals from different teams. As a consequence of this long and often confusing evolution, there are many misconceptions associated with *SysML*, such as its status as a profile, its autonomy as a language and how it can be applied in a better way for systems engineering. *SysML* significantly extends *UML* with system-related formal constructs, such as real-world physical constraints, physical flows and connections between physical components.

3 Modeling a CyCab with SysML

In this section we present an example of component based system, and SysML diagrams to describe the system.

3.1 CyCab Informal Description

As an example, we consider a CyCab car component-based system (in [9]). 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.

3.2 Block Definition Diagram

SysML provides a structural element called a *block*. A block can represent any type of component of the system, physical, logical, functional, or human. Blocks are declared within a *Block Definition Diagram* which describe the structure of the system. SysML Block Definition Diagram (*BDD*) is based on the UML *Composite Structure Diagram*, which extends the UML *Class Diagram*. The role of a *BDD* is to describe the relationships among blocks, which are basic structural elements aiming to specify hierarchies and interconnections of the system to be modeled. Required interfaces (relation uses) and offered (relation implements)of components are also

described. Figure 1 shows an example of a *BDD* with eight blocks. It is the first

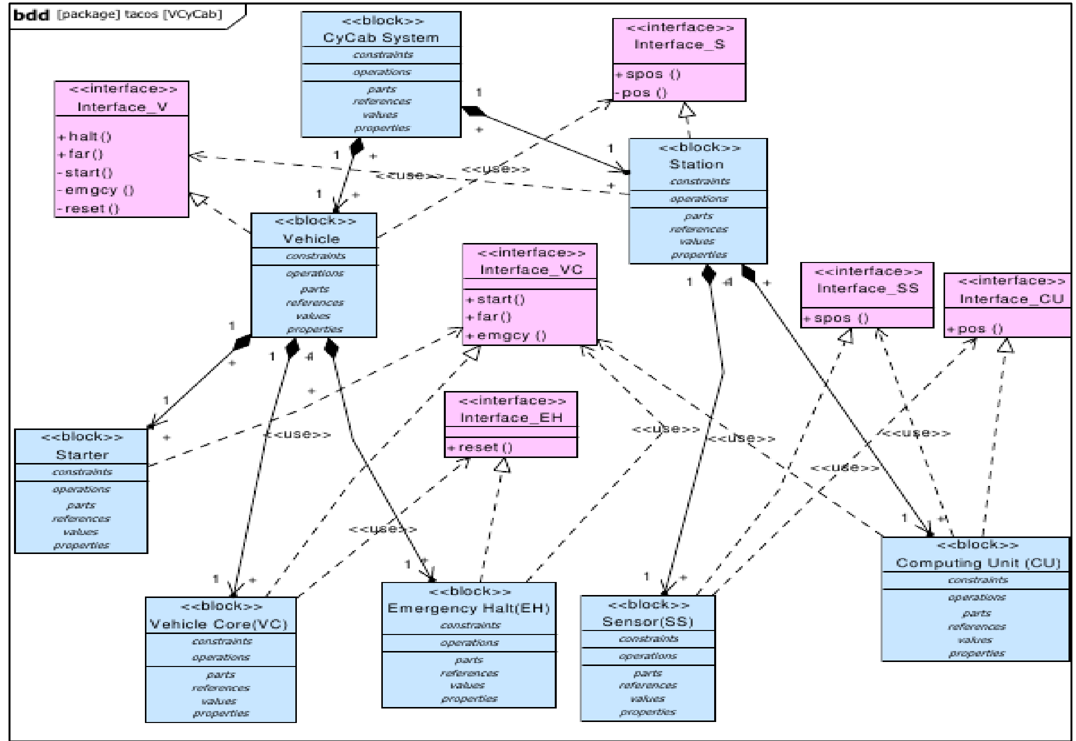

Figure 1. Block Definition Diagram of CyCab

level of modeling of the *CyCab*. The block named ***CyCab* System** represents the system as a whole. It is decomposed into two sub blocks (**Vehicle**, and **Station**) and is linked to them by the composition relationship. The component **Vehicle** is divided into three sub-components which are **Starter**, **Vehicle Core (VC)** and **Emergency Halt (EH)**. **Station** is decomposed into two sub-components that are **Sensor** and **Computer Unit (CU)**. In this paper we exploit a BDD to specify formally the system architecture, and exploit this specification in components assembly.

3.3 Internal Block Diagram

The *Internal Block Diagram* (*IBD*) allows the designer to refine the structural aspect of the model. The *IBD* is the equivalent in SysML of the composite structure diagram in UML . In the *IBD*, parts are basic elements assembled to define how they collaborate to realize the block structure and/or behavior. A *part* in SysML corresponds to an object in UML . Parts represent the physical components of the block while flow ports represent the interfaces of the block, through which its communicates with other blocks. Two types of ports are available in SysML :

- standard ports handling requests and invocations of services with other blocks (basically the same concept as in UML 2.0);
- flow ports which let blocks exchange flows of information.

Flow ports specify the interaction points among blocks and parts supporting the integration of behavior and structure. For standard ports, an interface class is used to list the services offered by the block. For flow ports, a flow specification is created to list the type of data that can flow through the port.

The figure 2 shows the *IBD* of *Vehicle* . For example in Figure 2, the parts **Starter**, **Vehicle Core (VC)** and **Emergency Halt (EH)** cooperate to achieve the functionality of the component **Vehicle**.

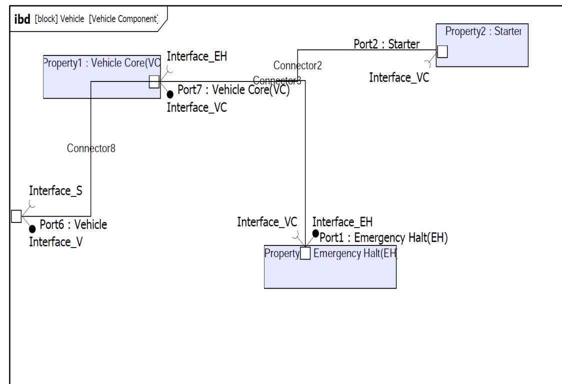

Figure 2. Internal block diagram of vehicle

3.4 Requirement diagram

Requirement specifies capability or condition that must be delivered in the subject (target system). Capability usually refers to the function that the system must support and we call it functional requirement. Condition usually means that the system should be able to run or produce the result in specific constraint, and we call it non-functional requirement The SysML requirement diagram allows several ways to represent requirements relationships. These include relationships for defining requirements hierarchy, deriving requirements, satisfying requirements, verifying requirements and refining requirements. The relationship can improve the specification of systems, as they can be used to model requirements. In Figure 3, the requirement GECC, *Global Maximal Energy Consuming of CyCab*, contains the requirements ECS, *Maximal Energy Consuming of the Station component* and ECV, *Maximal Energy Consuming the Vehicle component*. For example for in the requirement ECV, the identifier *Qhalt* is the number of energy resources necessary to execute the service halt.

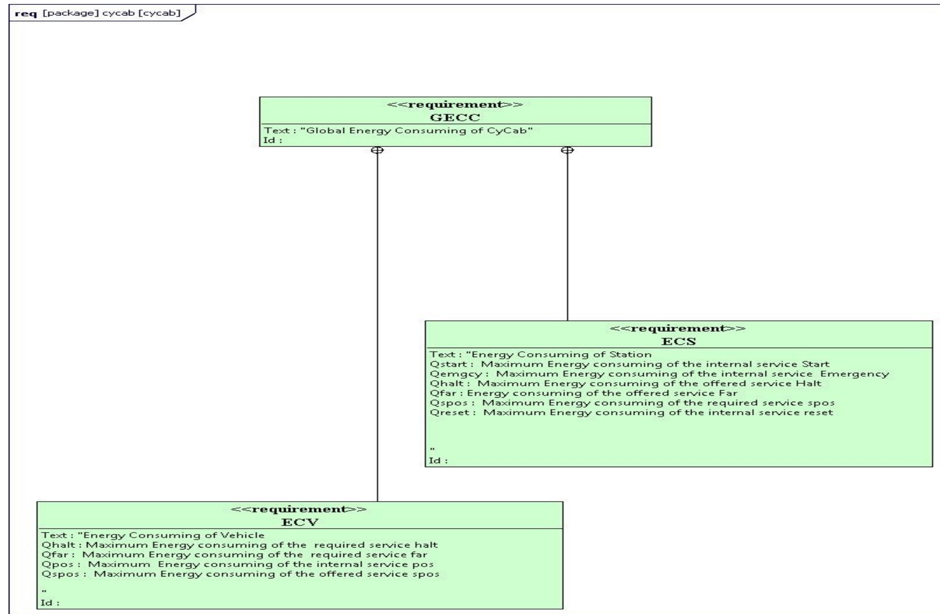

Figure 3. Requirements diagram

3.5 Sequence Diagram

The sequence diagram is used to represent the interaction between structural elements of a block, as a sequence of message exchanges, called also component (or block) protocols. In the CyCab system, 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 components *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 by three primitive components: the *VehicleCore* (*Vc*), the *Starter* (*Sr*), and the embedded *EmergencyHalt* (*Eh*) device. For example, the figure 4 shows the sequence diagram of the part property(component) *ComputingUnit*. These diagrams specifies the component protocols, which exhibit the interaction between the components and their environment. The environment represent the others components in the system.

4 Extracting a Tree from SysML architecture

We specify the SysML architecture as a graph where nodes are the Blocks of the system and edges represent both hierarchical relations between composite Blocks and their subBlocks. The nodes of the graph can be seen as tree if we consider only hierarchical relations. For a SysML architecture M, we denote by $\mathcal{C}_M$ all the

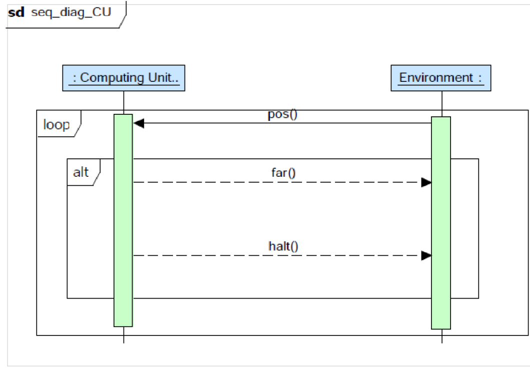

Figure 4. Sequence diagram of *computer unit*

(composite and primitive) components composing M.

Definition 7 (Graph Representation of Architecture). *A* Graph Representation $G_M = \langle\, N_{G_M},\, Cp_{G_M},\, Cn_{G_M} \,\rangle$ *of an* SysML *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 Blocks and their subBlocks*;
- *a finite set* Cn_{G_M} *of edges representing the connections between the nodes representing subcomponents within a same Block.*

In the CyCab example, we associate the graph described in Figure 5, to model formally the system architecture described by BDD and IBD SysML models. The continuous edges represent the hierarchical relations between composite Blocks and their subBlocks. The dashed edges represent the connections between components at the level of composite Blocks. Two Blocks are connected if and only if there is at least one interaction between their interfaces.

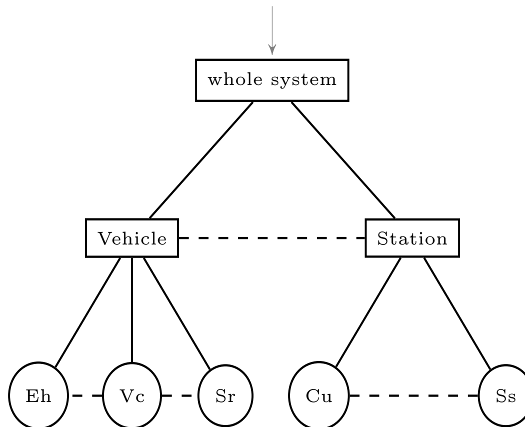

Figure 5. The graph of the CyCab car system

By traversing this graph, we can easily extract the authorized order in which the

components of the whole system (composite system) will be composed, then we exploit this information in the verification of the compatibility between components. For example, the order of the composition associated the CyCab system based the tree described in the figure 5 is:
$(Starter \parallel VehicleCore \parallel EmergencyHalt) \parallel (Sensor \parallel ComputingUnit)$.
We consider that the symbol $\parallel$ is the operator of composition. We note that the operation of composition is associative, so the order of composition has no effect in the the verification of the compatibility between components.

5 Interface automata strengthened by non-functional property and SysML diagrams

In this section we present our formalism based on interface automata to specify formally the component interfaces according to SysML diagrams, in order to verify component interoperability.

5.1 Component interfaces based on SysML sequence diagrams and NF requirements

We propose to specify formally component interfaces by considering component protocols specified by sequence diagrams, and component NF requirement specified by requirements diagrams. So, we propose to exploit the interface automata formalism, which we enrich with NF requirements in order to model formally SysML sequence diagram and requirement diagram.
Interface automata have been defined by L.Alfaro et al. [7], to model the temporal behavior of software component interfaces. They are considered as labeled transition systems, where the transitions are labeled with the names of actions which are divided into three categories: input, output, and hidden actions. 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 ";"). 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.

In this section we show how to exploit the interface automata formalism in order to consider energy consumption (non-functional requirements) of each component action, in the specification and the verification of component assembly. So, we present below a definition of interface automata strengthened by the function that specifies energy consumption.

Definition 2 (Interface Automata). *An* interface automaton $A = \langle\, S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A, \lambda_A \,\rangle$ *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, we denote 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 between states*;
- λ_A : total function that associates to each action the number [2] of energy resource necessary to its execution $\Sigma_A \rightarrow \mathbb{N}$. This function allows to specifies the values which we associate to the maximum of energy consuming of each component action, specified in the requirement diagram.

When we compose two interface automata, the resulting composite automaton, based on the synchronized product of the both automata, may contain *illegal states*, where one automaton issues an output action that is not acceptable as input in the other one. The existence of these illegal states is not sufficient to decide the incompatibility between interfaces . Indeed, the proposed interface automata approach, called also optimistic approach, allows to verify the compatibility between interface automata, based on the fact that there is an environment which provides only legal inputs. The composite interface expects the environment to pass over transitions leading only to legal states. The existence of such legal environment for the composition of two interfaces indicates that there is a way that the components can work together properly. This optimistic approach is different from the classical approaches, where it is sufficient to find one illegal state (called deadlock state) to decide of the incompatibility between interfaces.

The interface automata of the primitive components of the CyCab car system are presented in Figure 6 and Figure 7. The energy consumption information are indicated in the interface automata. These automata specifies formally the sequence diagram and requirement diagram described in the previous section. So the interface automata of the component Computer Unit described in Figure 7, specifies the protocol of this component,therefore it specifies the sequence diagram of Computing Unit described in Figure 4. We also exploit the NF requirements described with requirement diagram in Figure 3 to annotates transition in the interface automata with cots associated to energy consumption by each actions. For example, in Figure 6, we can see in the interface automaton of the component Vehicle core, that the component offers an action $far?$ which necessitates 5 energy units, and the component requires an action *spos*! with at most 6 energy units. These values 5, and 6 correspond respectively to $Qfar$ and $Qspos$ in the requirement ECV in figure 3.

5.2 Blocks Compatibility Verification

The verification of the compatibility between a blocks (component C_1) and a other block (component C_2) is obtained by verifying the compatibility between their interface automata A_1 and A_2. The verification steps of the compatibility are listed below.

Main algorithm

[2] We suppose that this number belongs the set of natural numbers.

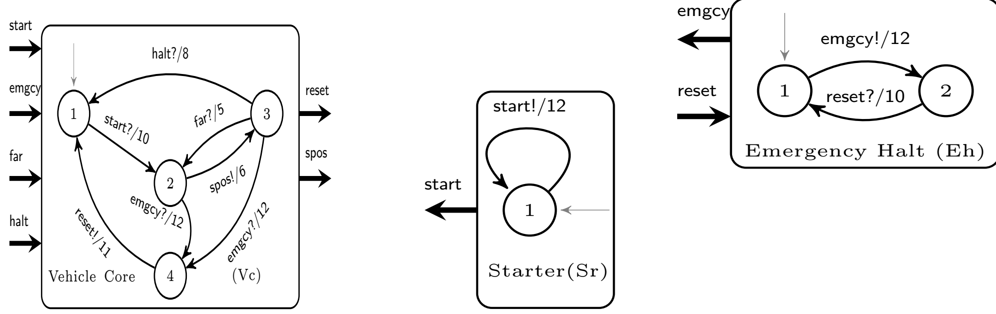

Figure 6. The interface automata of the *Vehicle* subcomponents

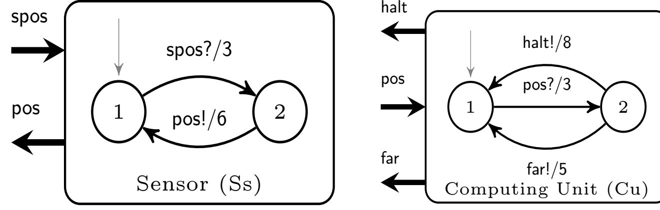

Figure 7. The interface automata of the *Station* subcomponents

Input : SysML modelling
Output : the interface automaton of the composite component if the compatibility is satisfied, or an empty automata in other case.

(i) Generating the corresponding tree to the block diagram and internal block in order to specify formally system architecture,

(ii) Formal specification of sequence diagrams and requirement diagram with interface automata enriched with non-functional property.

(iii) compatibility verification between interface automata by processing the following algorithm (Algorithm 2) and considering the system architecture and the NF property,

Algorithm 2
Input : interface automata A_1, A_2.
Output : $A_1 \parallel A_2$.

(i) verify that A_1 and A_2 are composable,

(ii) compute the product $A_1 \otimes A_2$,

(iii) compute the set of illegal states in $A_1 \otimes A_2$,

(iv) compute the set of incompatible states in $A_1 \otimes A_2$: the states from which the illegal state are reachable by enabling only internal and output actions (one suppose the existence of a helpful environment),

(v) compute the composition $A_1 \parallel A_2$ by eliminating from the automaton $A_1 \otimes A_2$, the illegal state, the incompatible states, and the unreachable states from the initial states,

(vi) if $A_1 \parallel A_2$ is empty then A_1 and 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 and their corresponding component can be assembled properly.

In the following, we present the definitions of formal concepts (composition condition, synchronized product...) exploited in the below algorithm by considering NF property.
The composition operation may take effect only if the actions of the two automata 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 (Composition Condition). *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.

In the following we present the definition of synchronized product between two interface automata taking into account energy consumption constraints. The intuition behind the following definition is, two components can synchronize on shared actions whether one of two interacting components, C_1, requires an action sa (output action) which consumes x energy units, and the other component, C_2, offers the action sa (input action) which consumes y energy units, such that $x \geq y$. This is an obvious condition because :

- first, generally components are reusable, and developed by different teams and companies, so the offered and the required actions of components may not consume the same amount of energy units.
- second, for example: C_1 can not use the offered action, sa, by C_2, if this action necessitates more energy units than those allocated by C_1.

When one synchronizes it is necessary to calculate the real energy consumption of the new internal action (created by the synchronization) by considering the minimum value between the resources of the input and output actions.

Definition 4 (Synchronized product considering Energy Consumption). *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 $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus \mathit{Shared}(A_1, A_2)$;
- $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus \mathit{Shared}(A_1, A_2)$;
- $\Sigma^H_{A_1 \otimes A_2} = \Sigma^H_{A_1} \cup \Sigma^H_{A_2} \cup \mathit{Shared}(A_1, A_2)$;
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ *if*
  · $a \notin \mathit{Shared}(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  · $a \notin \mathit{Shared}(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  · $a \in \mathit{Shared}(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge ((\lambda_{A_1}(a) \leq \lambda_{A_2}(a) \wedge a \in$

$\Sigma^I_{A_1}(s_1) \wedge a \in \Sigma^O_{A_2}(s'_1)) \vee (\lambda_{A_1}(a) \geq \lambda_{A_2}(a) \wedge a \in \Sigma^O_{A_1}(s_1) \wedge a \in \Sigma^I_{A_2}(s'_1)))$

· $\lambda_{A_1 \otimes A_2} : \Sigma_{A_1 \otimes A_2} \rightarrow \mathbb{N}$ such that $\Sigma_{A_1 \otimes A_2} = \Sigma^I_{A_1 \otimes A_2} \cup \Sigma^O_{A_1 \otimes A_2} \cup \Sigma^H_{A_1 \otimes A_2}$, to define $\lambda_{A_1 \otimes A_2}$ we consider the following cases :

$a \in \Sigma_{A_1 \otimes A_2} \wedge \lambda_{A_1 \otimes A_2}(a) = \lambda_{A_1}(a)$ if $a \notin \mathit{Shared}(A_1, A_2) \wedge (a \in \Sigma^I_{A_1} \vee a \in \Sigma^O_{A_1} \vee a \in \Sigma^H_{A_1})$;

$a \in \Sigma_{A_1 \otimes A_2} \wedge \lambda_{A_1 \otimes A_2}(a) = \lambda_{A_2}(a)$ if $a \notin \mathit{Shared}(A_1, A_2) \wedge (a \in \Sigma^I_{A_2} \vee a \in \Sigma^O_{A_2} \vee a \in \Sigma^H_{A_2})$;

$a \in \Sigma_{A_1 \otimes A_2} \wedge \lambda_{A_1 \otimes A_2}(a) = \mathit{min}(\lambda_{A_1}(a), \lambda_{A_2}(a))$ if $a \in \mathit{Shared}(A_1, A_2) \wedge ((a \in \Sigma^I_{A_1} \wedge a \in \Sigma^0_{A_2}) \vee (a \in \Sigma^O_{A_1} \wedge a \in \Sigma^I_{A_2}))$ [3] .

In the following we adapt the definition of illegal states in order to consider energy consumption constraints. So, a state (s_1,s_2) in the product is considered illegal in the following cases:

- one component requires a shared action from the state s_1 which is not provided from the state s_2 in the other component or vice versa.
- one component provides a shared action with a value of energy consumption greater than the value of energy consumption of the required action, by the other component.

Definition 5 (Illegal States considering Energy Consumption). *Given two composable interface automata A_1 and A_2, the set of illegal states $\mathit{Illegal}(A_1, A_2)$* $\subseteq S_{A_1} \times S_{A_2}$ of $A_1 \otimes A_2$ is defined by $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} | \exists a \in \mathit{Shared}(A_1, A_2). ((a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2)) \vee (a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1)) \vee (a \in \Sigma^O_{A_1}(s_1) \wedge a \in \Sigma^I_{A_2}(s_2) \wedge \lambda_{A_2}(a) > \lambda_{A_1}(a)) \vee (a \in \Sigma^I_{A_1}(s_1) \wedge a \in \Sigma^O_{A_2}(s_2) \wedge \lambda_{A_1}(a) > \lambda_{A_2}(a)))\}$.

Definition 6 (Composition). *Given two compatible interface automata A_1 and A_2. The composition $A_1 \parallel A_2$ is an interface automaton defined by: (i) $S_{A_1 \parallel A_2} = \mathit{Comp}(A_1,A_2)$, (ii) the initial state is $I_{A_1 \parallel A_2} = I_{A_1 \otimes A_2} \cap \mathit{Comp}(A_1,A_2)$, (iii) $\Sigma_{A_1 \parallel A_2} = \Sigma_{A_1 \otimes A_2}$, and (iv) the set of transitions is $\delta_{A_1 \parallel A_2} = \delta_{A_1 \otimes A_2} \cap (\mathit{Comp}(A_1,A_2) \times \Sigma_{A_1 \parallel A_2} \times \mathit{Comp}(A_1,A_2))$.*

The complexity for computing the composition $A_1 \parallel A_2$ is in time linear on $|A_1|$ and $|A_2|$ [7]. The verification steps in this approach are the same as the ones presented in [7]. However, in our approach we consider energy consumption in:

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

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

5.3 Illustration on the CyCab

Let us take our previous example, the interface automata of the primitive components of the CyCab car system are presented in Figure 6 and Figure 7. The algorithm

[3] min is a function which returns a minimum value between two positive real numbers

starts first by constructing the *Station* and the *Vehicle* composite components from their subcomponents, according to system architecture (tree) obtained in section 4. The reader can easily verify that the two interface automata *Station* and *Vehicle* are not empty. Then, we construct the whole system composite representing the communication between the *Vehicle* and the *Station*. Figure 8 represents the IAs of the composite components *Vehicle* and *Station*.

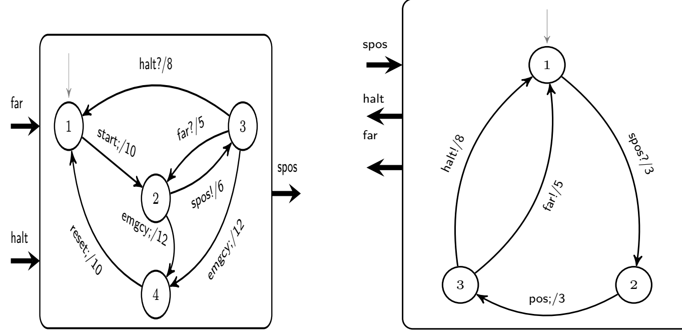

Figure 8. The interface automata of the *Vehicle* composite and the *station* composite.

6 Related works

In this section, we can mention the work proposed in [10], where 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. Non-functional properties are not considered in this approach. The approach proposed in [13] 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 [11], the authors proposed to specify component interface and to verify their compatibility with B method. However component protocols and non-functional properties are not considered in the interfaces. n [15], the authors describes a step of translating *SysML*-based specification to Colored Petri Nets (CPN) which enables rigorous static and dynamic system analysis as well as formal verification of the behavior and functionality of the *SysML*-based design. In [12], the authors propose assume-guarantee interface algebra for real-time components. In the interface specification, they consider the following properties : an arrival rate function and a latency for each task sequence, and a capacity function for the shared resource. The interface specifies that the component guarantees certain task latencies depending on assumptions about task arrival rates and allocated resource capacities. This properties are considered in the verification of interface compatibility. These last approaches treat non-functional properties in component composition but the architecture of the whole system is not considered.

The contribution and the originality of our approach, compared the related works, is the specification of component interfaces with interface automata, which a more general formalism based on rich notations which allow to express more

complex component behaviors. We propose also a connection with *SysML* to verify the component composition by taking into account non-functional properties and system architecture.

7 Conclusion and future work

In this paper, we present a new formal approach to compose components and to verify their interoperability, according to energy consumptions properties specified by SysML requirement diagram, to a system architecture, specified by SysML block definition diagram, and to component protocols specified by sequence diagrams. This approach use interface automata method to specify component interfaces and to verify interface compatibility. We have improved this approach by considering non-functional properties and by exploiting block definition SysML model, in order to specify , connection between components and composites, and hierarchical connection. Block definition diagram corresponding to system architecture is specified formally by a tree, where nodes correspond to blocks and edges specify connections between blocks. From this tree, we deduce information to improve interface automata approach in order to verify interface compatibility. So, we have proposed an algorithm to compose components and composites, based on both, system architecture and component interface automata. We have proposed also a verification approach that takes into account non-functional properties in component assembly. We propose to verify whether the energetic consumption of a component based system is in compliance with the available resources of energy. As future work, We plan to implant the algorithms described in this paper and to evaluate the proposed approach.

References

[1] http://www.omg.org.

[2] http://www.incose.org.

[3] http://ap233.eurostep.com.

[4] http://www.uml.org.

[5] OMG. UML for Systems Engineering. Request For Proposal, 2003.

[6] www.omgsysml.org OMG. OMG Systems Modeling Language (OMG SysML) Specification, OMG Final Adopted Specification, 2006.

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

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

[9] G. Baille, P. Garnier, H. Mathieu, and R. Pissard-Gibollet. *The INRIA Rhône-Alpes CyCab*. Technical report, INRIA, 1999. Describes the package natbib.

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

[11] S. Chouali, M. Heisel, and J. Souquières. Proving component interoperability with b refinement. *Electr. Notes Theor. Comput. Sci.*, 160:157–172, 2006.

[12] T. A. Henzinger. An interface algebra for real-time components. In *In Proc. of IEEE Real-Time Technology and Applications Symposium*, pages 253–263. Society Press, 2006.

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

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

[15] R. Wang and C. H. Dagli. An Executable System Architecture Approach to Discrete Events System Modeling Using SysML in Conjunction with Colored Petri Net. In *Systems Conference, 2008, 2nd Annual IEEE*, 2008.

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