Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 295 (2013) 31–47

www.elsevier.com/locate/entcs

Assembling Components using SysML with Non-Functional Requirements

Samir Chouali, Ahmed Hammad, Hassan Mountassir [1]

FEMTO-ST Institute UMR CNRS 6174
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 diagrams. 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 assembly, interface automata, SysML, non-functional properties, system architecture.

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, [19]. 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), behavior 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 holds between components when their interfaces are compatible.

[1] Email: {schouali, ahammad, hmountas}@femto-st.fr

1571-0661 © 2013 Elsevier B.V. Open access under CC BY-NC-ND license.
http://dx.doi.org/10.1016/j.entcs.2013.04.003

The *SysML* language is an UML profile, that is a language for documenting and graphically specify all aspects of a system consisting of hardware and/or software blocks. *SysML* has been proposed by the Object Management Group (OMG) [1] to define a general purpose modeling language for systems engineering. *SysML* enjoys unprecedented popularity both in industry and academia, It is 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 specified at the first step with *SysML* diagrams. In the second step, we propose a formal specification and verification approach to verify components composition based on the *SysML* models of the first step. Our approach exploit and adapt the interface automata formalism to verify components composition. The interface automata based approach was proposed by L.Alfaro and T.Henzinger, [2,3]. They have specified component interfaces with automata, which are labelled by input, output, and internal actions. These automata 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.

In this context we propose to specify component based system (CBS) architecture, components requirements, and component behavior, with *SysML* diagrams. Our goal is to exploit the graphical formalism of *SysML* to model CBS. We argue that is a suitable formalism to specify components and to communicate between CBS specifiers and developers on CBS projects. Another goal is to treat CBS Non Functional (NF) requirements, which is facilitated by exploiting *SysML* requirement diagrams. In order to verify formally the assembly between components, specified with *SysML* models, we propose to adapt and to exploit the interface automata approach. We propose two adaptations of this approach : (i) The first is to handle the system architecture specified with *SysML* in the formal specification and verification with interface automata. In fact Interface automata are proposed to specify component behaviors 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 sub-components, which also influences component behaviors. (ii) The second is to consider the NF-requirements in the interface automata formalism and in the verification of components composition. These NF-requirements consist in the energy consumption of each services offered or required by components. So, we propose to annotate transitions in interface automata with costs, to obtain a king of weighted automata, and we propose to handle this annotations in the verification of components composition.

The paper is organized as follows. In Section 2, we present the example of the CyCab vehicle, and we specify semi-formally system architecture, component behaviors, and NF-requirements, with *SysML* diagrams. In Section 3, we describe a methodology to specify formally *SysML* architecture of a component based system.

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 NF-requirements. Related works are described in Section 5. We conclude our work and trace some perspectives in Section 6.

2 Modeling a CyCab with SysML

In this section we present an example of component based system, and propose SysML diagrams to describe the system. These diagrams are considered as the first step of component based system modeling, which will be exploited in the formal specification and verification step (second step).

2.1 CyCab Informal Description

As an example, we consider a CyCab car component-based system (in [6]). 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. It is totally controlled by a computer system and it can be driven automatically according to many modes. The goal of the CyCab is to allow to a clients to use the vehicle to move from one station to another. To illustrate this concept, we consider the following system constraints:
(1) A CyCab has an appropriate road where stations are equipped with sensors and computing units. (2)We propose that the driving of the CyCab is guided by the information received from the stations, which situates the CyCab compared to the stations. (3)There is no obstacle in the road. (4)The vehicle is activated by the starter component. The vehicle has also an emergency halt button, associated to the emergency halt component. The emergency halt button can be activated at every moment during the CyCab moves. It is specified by sending the signal *emgcy!*.

The CyCab and its environment can be seen as an abstract system composed of two composite components : the vehicle, and the station. The vehicle is also composed by the primitive components : Vehicle Core, Starter, and Emergency Halt (associated to emergency halt button). The station is composed by the primitive components : Sensor, and Computing unit.

We consider also the following constraints :

- The station is materialized by a sensor that receives signals from vehicle giving the vehicle position (*spos?*). The Sensor converts this position to geographic coordinates and send a message (*pos*!) to the Computing Unit. After treating the vehicle position, the Computing Unit sends, as consequence, a signal (*far!* or *halt!*), to the corresponding vehicle to indicate if it is far from the station or not.
- The vehicle sends also a signal *reset!* to the component emergency halt in order to reset the system after activating the emergency halt button.

2.2 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* (*BDD*) which describes the structure of the system. 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

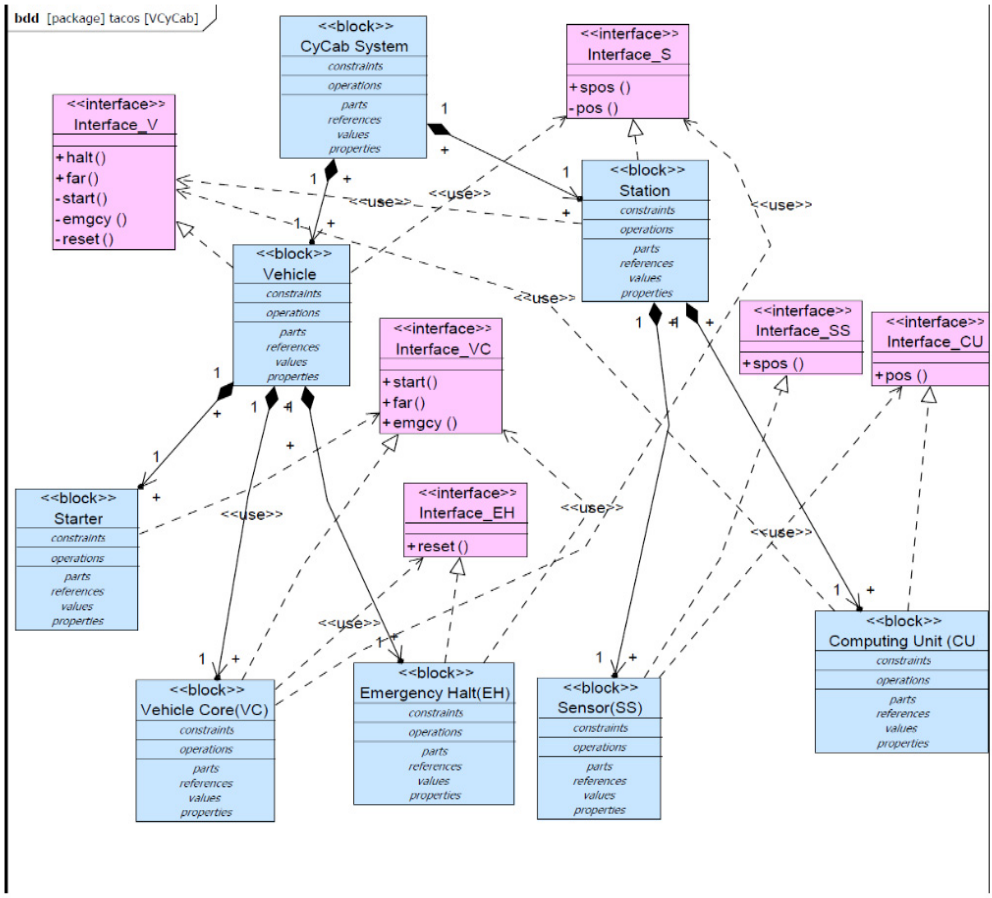

Fig. 1. Block Definition Diagram of CyCab

BDD with eight blocks. It is the first 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.

2.3 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.

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**. This diagram shows the assembly links between components, so we exploit it to deduce the assembly order between components.

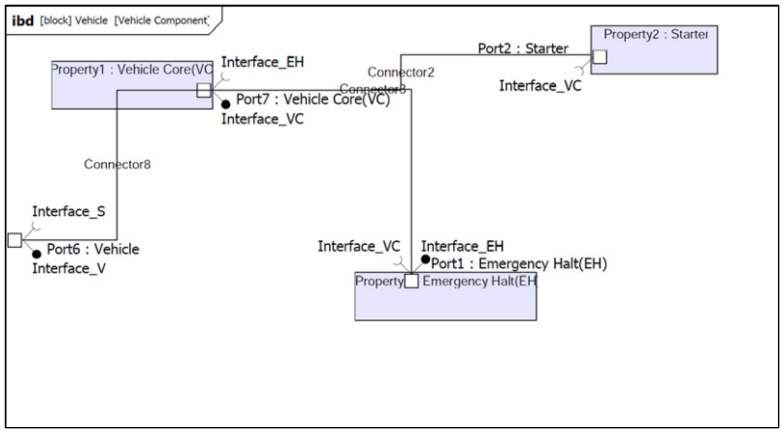

Fig. 2. Internal block diagram of vehicle

2.4 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

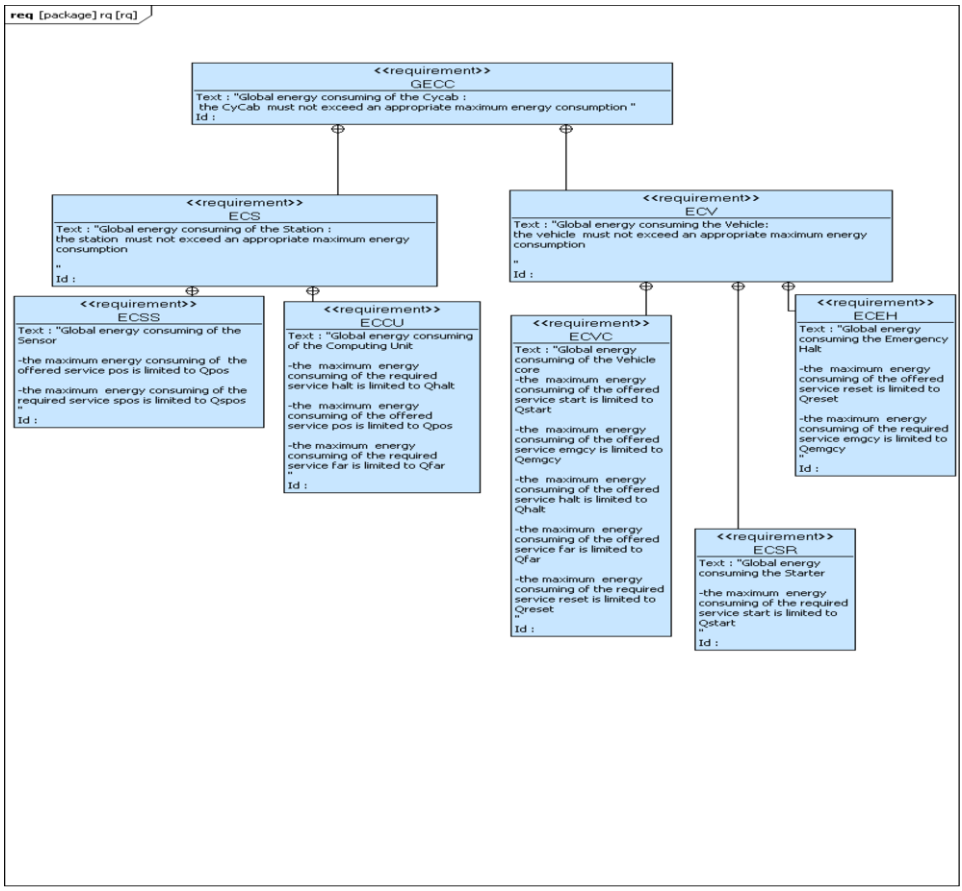

Fig. 3. Requirements diagram

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*, indicates that the CyCab must not exceed the limit of energy consuming. This requirement contains the requirements ECS, *Maximal Energy Consuming of the Station component* and ECV, *Maximal Energy Consuming of the Vehicle component*. The requirement ECS contains the requirements ECSS, *Maximal Energy Consuming of the Sensor component*, and ECCU, *Maximal Energy Consuming of the Computing Unit component*. The requirement ECV contains the requirements ECVC, *Maximal Energy Consuming of the Vehicle Core component*, ECSR, *Maximal Energy Consuming of the Starter component*, and ECEH, *Maximal Energy Consuming of the Emergency Halt component*. For example in the requirement ECSS, the identifier *Qpos* is the number of energy resources necessary to execute the offered service pos by other components, and the identifier *Qspos* is the maximum of energy resources that the component Sensor could use to execute the required service spos.

2.5 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

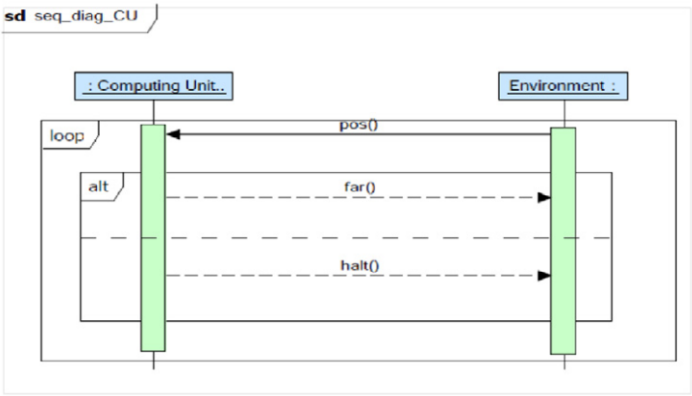

Fig. 4. Sequence diagram of *computer unit*

figure 4 shows the sequence diagram component *ComputingUnit*. This diagram specifies the component protocols, which exhibits the interaction between the component and its environment. The environment represent the others components in the system. So the Computing Unit receives the message, pos, from the environment (which is the Sensor component) and responds by sending the messages far or halt the environment (vehicle component).

3 Formal specification of SysML system architecture

In the previous section, we have proposed to specify component based system architecture with BDD and IBD *SysML* diagrams. BDD diagram show the system global structure, and the relation between Composite blocks (composite component) and their sub-blocks (sub-components). IBD diagram show the composition links between blocks. In this section, we propose to specify formally this architecture as a graph where nodes correspond to the Blocks of the system and edges represent both

hierarchical relations between composite Blocks and their sub-blocks. The nodes of the graph can be seen as tree if we consider only hierarchical relations. 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 sub-blocks, which are specified in the BDD model. The dashed edges represent the connections between components at the level of composite Blocks. This connection links are specified in IBD models. Two Blocks are connected if and only if there is at least one interaction between their interfaces.

The formal definition of this graph is presented in the definition 1. For a *SysML* system architecture M, we denote by $\mathcal{C}_M$ all the (composite and primitive) components composing M.

Definition 1 (Graph Representation of Architecture). *A* Graph Representation $G_M = \langle N_{G_M}, Cp_{G_M}, Cn_{G_M} \rangle$ *of a system* 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 sub-blocks*;
- *a finite set* Cn_{G_M} *of edges representing the connections between the nodes representing sub-components within a same Block.*

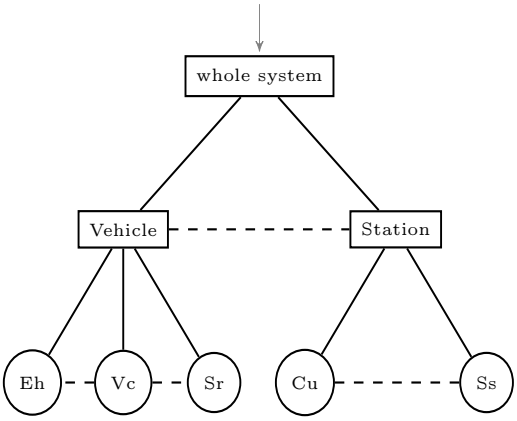

Fig. 5. The graph of the CyCab car system

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: $(Starter \parallel VehicleCore \parallel EmergencyHalt) \parallel (Sensor \parallel ComputingUnit)$. In fact we can see in the figure 5 that the nodes Eh, Vc, Sr, associated respectively to the components EmergencyHalt, Vehi-

cleCore, and Starter, are connected with dashed links, and they are also associated to the node, Vehicle, which corresponds to th composite component $Vehicle$. So they can be assembled together by ($Starter \parallel VehicleCore \parallel EmergencyHalt$), in order to obtain the composite Vehicle. We apply the same process on the components Sensor and ComputingUnit which are associated to the nodes Cu and Ss, and the composite Station, obtained by ($Sensor \parallel ComputingUnit$). We can see also in the same figure that the node Vehicle is related to the node Station by a dashed link, and they are also related to the whole system. So to obtain a system one have to compose : ($Starter \parallel VehicleCore \parallel EmergencyHalt$) $\parallel$ ($Sensor \parallel ComputingUnit$). 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.

4 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.

4.1 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 Non Functional (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. [2], 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. So we define a kind of weighted automata where transitions are annotated with costs which correspond to energy consumption of actions.

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* $I_A \subseteq S_A$;
- *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 of energy resources necessary to its execution $\Sigma_A \rightarrow \mathbb{N}$. This function 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.

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 annotate 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.

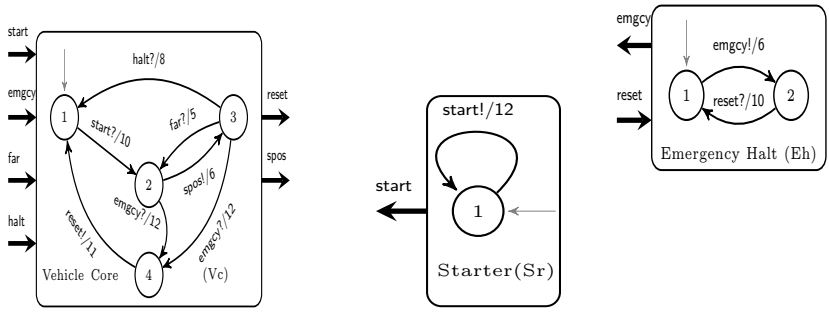

Fig. 6. The interface automata of the *Vehicle* subcomponents

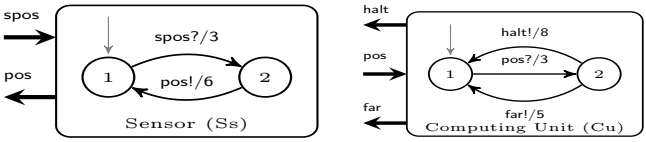

Fig. 7. The interface automata of the *Station* subcomponents

4.2 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
Input : *SysML* modelling
Output : the interface automaton of the composite component if the compatibility is satisfied, or an empty automata in other case.
Algorithm steps :
(*i*) Generating the corresponding tree to the block definition diagram and internal blocks in order to specify formally system architecture : this step is performed by applying transformation rules from BDD and IBD to obtain the tree. These rules are obvious. For example each block corresponds to a tree node, and the links between block are treated in two steps, in order to obtain the links between blocks and their sub-blocks from BDD and the links between the sub-blocks from IBD models. (*ii*) Formal specification of sequence diagrams and requirement diagram with interface automata enriched with non-functional property: this step is performed by applying transformation rules from sequence diagram and requirement diagram to obtain interface automata with energy consumption constraints. Rule transformation from sequence diagrams to interface automata are presented in [10] . The originality in this paper is the consideration of the requirement diagrams and the NF-requirements in the interface automata approach. We exploit these diagrams to annotate transitions with costs in interfaces automata. (*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 : $A_1 \parallel A_2$.
Algorithm steps :
(*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 states, 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.
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 $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$; $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus Shared(A_1, A_2)$; $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus Shared(A_1, A_2)$;
$\Sigma^H_{A_1 \otimes A_2} = \Sigma^H_{A_1} \cup \Sigma^H_{A_2} \cup Shared(A_1, A_2)$;
$((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ *if*

- $a \notin Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
- $a \notin Shared(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
- $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge ((\lambda_{A_1}(a) \leq \lambda_{A_2}(a) \wedge a \in \Sigma^I_{A_1}(s_1) \wedge a \in \Sigma^O_{A_2}(s'_1)) \vee (\lambda_{A_1}(a) \geq \lambda_{A_2}(a) \wedge a \in \Sigma^O_{A_1}(s_1) \wedge a \in \Sigma^I_{A_2}(s'_1)))$
- $\lambda_{A_1 \otimes A_2} : \Sigma_{A_1 \otimes A_2} \to \mathbb{N}$ 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 $a \notin Shared(A_1, A_2) \wedge (a \in \Sigma^I_{A_1} \vee a \in \Sigma^O_{A_1} \vee a \in \Sigma^H_{A_1})$;
  - $a \in \Sigma_{A_1 \otimes A_2} \wedge \lambda_{A_1 \otimes A_2}(a) = \lambda_{A_2}(a)$ if $a \notin Shared(A_1, A_2) \wedge (a \in \Sigma^I_{A_2} \vee a \in \Sigma^O_{A_2} \vee a \in \Sigma^H_{A_2})$;
  - $a \in \Sigma_{A_1 \otimes A_2} \wedge \lambda_{A_1 \otimes A_2}(a) = min(\lambda_{A_1}(a), \lambda_{A_2}(a))$ if $a \in Shared(A_1, A_2) \wedge ((a \in \Sigma^I_{A_1} \wedge a \in \Sigma^0_{A_2}) \vee (a \in \Sigma^O_{A_1} \wedge a \in \Sigma^I_{A_2}))$ [2].

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: (i) 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. (ii) 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 $Illegal(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ of $A_1 \otimes A_2$* is defined by $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} | \exists a \in Shared(A_1, A_2). ((a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2)) \vee (a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1)) \vee (a \in \Sigma^O_{A_1}(s_1) \wedge a \in \Sigma^I_{A_2}(s_2) \wedge \lambda_{A_2}(a) > \lambda_{A_1}(a)) \vee (a \in \Sigma^I_{A_1}(s_1) \wedge a \in \Sigma^O_{A_2}(s_2) \wedge \lambda_{A_1}(a) > \lambda_{A_2}(a)))\}$.

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) $S_{A_1 \| A_2} = Comp(A_1,A_2)$ [3], (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 $\delta_{A_1 \| A_2} = \delta_{A_1 \otimes A_2} \cap (Comp(A_1,A_2) \times \Sigma_{A_1 \| A_2} \times Comp(A_1,A_2))$.*

The complexity for computing the composition $A_1 \parallel A_2$ is in time linear on $|A_1|$ and $|A_2|$ [2]. The verification steps in this approach are the same as the ones

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

[3] The set of compatible states which are not reachable from illegal states.

presented in [2]. However, in our approach we consider energy consumption in the interface automata definition, in the product of two interface automata, and in the definition of the illegal states. Consequently, our approach does not increase the complexity of the verification algorithm.

4.3 Illustration on the CyCab

The reader can easily verify that by applying the algorithms described in the section 4.2 on the interface automata of the *Vehicle* and *Station* composites (Figures 6 and 7), we obtain the following results. In the interface automaton (Figure 8) obtained by calculating the synchronized product ($Vc \otimes Eh$) between the *Vehicle Core* and the *Emergency Halt* components, all the states are illegal, this is due to the action $emgcy!/6$ in Eh automaton which is not compatible with the actions $emgcy?/12$ in Vc automaton, so the components in the *Vehicle* composite are incompatible. However the components in the *Station* composite are compatible because the initial state in the interface automaton $Cu \otimes Ss$, is not reachable from the illegal state 4, so the composition $Cu \parallel Ss$ is not empty after eliminating illegal and incompatible states (Figure 9).

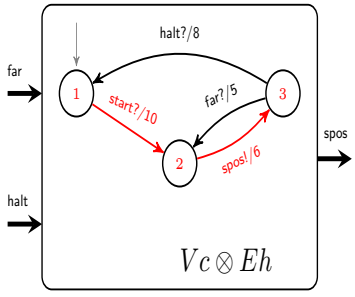

Fig. 8. Compatibility verification in the *Vehicle* composite

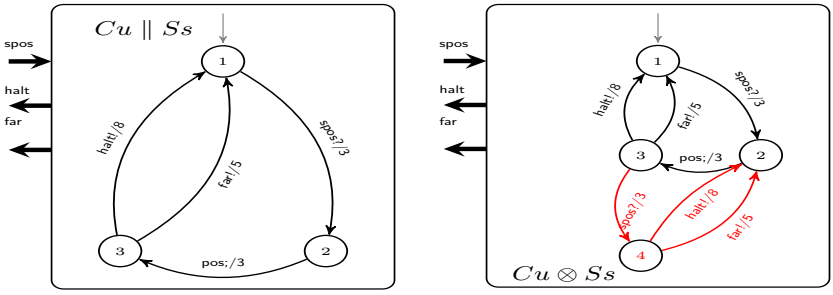

Fig. 9. Compatibility verification in the Station composite

5 Related works

In this section, we present a short survey of existing works about the non-functional properties (NFP) modeling and evaluation on components based system. Specific

to component-based systems, numerous studies have been conducted around the analysis, modeling and management of non-functional properties in components assembly. Analysis models and property theories are integrated to component technology in [17], and they allow one to guarantee, by construction, the predictability of some properties on components assembly. But, they require advanced analysis models and techniques, and are mostly dedicated to specific properties, such as latency [17], reliability [15], [4] or memory usage [13]. Some of these models could also be extended to other properties that are related to the system architecture and can be modeled in generic ways that allows to reason on them. In [16], 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. These 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. In [20], the author presents a way to introduce non-functional properties in a system with components expressed with $TLA+$. The main contribution of this paper is to show how these concepts can be expressed formally. In [9], the authors propose resource interfaces to specify component interfaces with requirements on limited resources. This approach allows verifying if a collection of components when put together exceed the available resources. These interfaces communicate with the environment with input and output variables, which decorate automata states, however in our case the communication is performed with input and output actions. In [12], the authors propose an extension of the interface automata approach to capture in addition to component protocols, the timing dimension of component interfaces. Timed interface is encoded as a timed game between input and output players. A verification algorithm for interfaces compatibility was also proposed. A close formalism to timed interfaces was proposed in [11], were the authors proposed a complete specification framework for real time systems based on timed Input/Output automata formalism. This approach support refinement, consistency checking, and composition. The difference with our approach is, in our case we treat a kind of NF properties which is energy consumptions, this constraint necessitates only to exploit weighted automata (associate a cost to transitions), however in [12], and [11] they exploit weighted timed automata: costs associated to transitions and valued states with clock variables in [11], and valued states with clock variables in [12]. Which increase the complexity of the verification of components composition. In our approach we have also to respect *SysML* diagrams where state variables are not allowed in the description of components, but only the order of actions and their energy consumptions. Consequently, our formalism is different from these formalisms and it is more suitable to model *SysML* diagrams.

In this section, we present works which treat the combination of graphical and formal languages to model NFP and to verify them. For example, the work proposed

in [8], the authors present a new approach to component interaction specification and verification process which combines the advantages of architecture description languages and model based formal verification. The approach proposed in [18] 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 [7], [14], the authors have presented work attempting to merge techniques from software performance engineering with component-based software engineering. They distinguish two model layers: the software model which represents the logical component structure of a system, and the machinery model which models properties relevant for performance analysis. In [5], this article focuses on the verification of a non-functional property which is a kind of *deadline*. the system is modeled in the form of data flow diagrams using a subset of the UML activity diagram. This is mapped to hierarchical and modular time Petri nets . This check is performed with the TINA tool. The contribution and the originality of our approach, compared the these related works is the specification of component interfaces with interface automata, which are more general formalism based on rich notations which allow to express more complex component behaviors. We propose also a connection with *SysML* to verify components composition by taking into account non-functional properties and system architecture.

6 Conclusion and future work

In this paper, we present an approach which combines formal and semi-formal formalisms, to compose components and to verify their interoperability. This verification is performed according to energy consumptions properties specified by *SysML* requirement diagrams, 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 adapted this approach to make it more appropriate to the formalization of *SysML* models. So we have considered non-functional properties and exploited BDD, and IBD *SysML* models, in order to specify system architecture. These models are specified formally by a tree, where nodes correspond to blocks and edges specify connections between blocks. From this tree, we deduce information to adapt interface automata approach in order to verify interface compatibility by considering the architecture of the whole system. Requirement and sequence diagrams are also exploited to annotate transitions in interface automata with costs corresponding to energy consumption of components services. The adaptations introduced in interface automata approach do not increase its complexity. 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 on more realistic case study.

References

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

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

[3] 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.

[4] R. R. an H. Schmidt and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 65(3), 2003.

[5] C. André, F. Mallet, and M.-A. P. Frati. Non-functional property analysis using UML2.0 and model transformations. Technical report, Reserach Report INRIA, RR-5913., 2006.

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

[7] Bertolino and R. A., Mirandola. Towards component based software performance engineering. In *6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction at ICSE 2003, ACM/IEEE (2003) 1?6.*, 2003.

[8] 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.

[9] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *EMSOFT*, pages 117–133, 2003.

[10] S. Chouali and A. Hammad. Formal verification of components assembly based on sysml and interface automata. *ISSE*, 7(4):265–274, 2011.

[11] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In *HSCC'10*, pages 91–100, 2010.

[12] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed interfaces. In *EMSOFT*, pages 108–122, 2002.

[13] E. Eskenazi, A. Fioukov, D. Hammer, and M. Chaudron. Estimation of Static Memory Consumption for Systems Built from Source Code Components. In *9th IEEE Conference and Workshops on Engineering of Computer-Based Systems*, 2002.

[14] V. Grassi and R. Mirandola. Towards automatic compositional performance analysis of component-based systems. In *In Dujmovic, J., Almeida, V., Lea, D., eds.: Proc. 4th Intl Workshop on Software and Performance WOSP 2004*, pages 59–63, California, USA, 2004. ACM Press.

[15] D. Hamlet, D. Mason, and D. Woit. Theory of Software Reliability Based on Components. In *Proceedings of in ICSE 2001. IEEE Computer, 2001.*, 2001.

[16] 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.

[17] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Enabling predictable assembly. *Journal of Systems and Software*, 65(3), 2003.

[18] S. Moisan, A. Ressouche, and J. Rigault. Behavioral substitutability in component frameworks: A formal approach. In *SAVCBS*, pages 22–28, 2003.

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

[20] S. Zschaler. Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. *SoSyM*, 9:161–201, Apr. 2009.