Formalizing and Verifying Compatibility and Consistency of SysML Blocks

Oscar Carrillo Rozo, Samir Chouali, Hassan Mountassir

Formalizing and Verifying Compatibility and Consistency of SysML Blocks

Oscar Carrillo
Femto-ST Institute
University of Franche-Comté
Besançon, France
e-mail: oscar.carrillo@femto-st.fr

Samir Chouali
Femto-ST Institute
University of Franche-Comté
Besançon, France
e-mail: samir.chouali@femto-st.fr

Hassan Mountassir
Femto-ST Institute
University of Franche-Comté
Besançon, France
e-mail:hmountas@femto-st.fr

May 16, 2012

Abstract

The objective of this paper is to define an approach to formalize and verify the SysML blocks in a refinement process. From a Block Definition Diagram which specifies system architecture, it is a matter of decomposing it into several sub-blocks, then of verifying their compatibility. The structural architecture of an abstract block is given by the Internal Block Diagram (IBD) which defines the communication links between sub-blocks. The compatibility verification between sub-blocks is only made on linked sub-blocks. The behaviour of each sub-block is described by an interface automaton which specifies the invocations exchanged with its environment. The verification between blocks is translated into consistency verification between the blocks and compatibility verification between their interface automata. Incompatibilities can be inconsistent at architecture level and at communication level if there are illegal states. Once the verification is established between the sub-blocks, the abstract block can be then substituted by the sub-blocks which compose it.

1 Introduction

The systems become increasingly complex and their implementation asks more rigorous conception approaches. To develop reliable systems, several software engineering approaches have been proposed and particularly top-down approach which allows to build a system, step by step from height abstract specifications. This approach has been used to design component-based systems constituted by communicating entities. This approach allows effectively to enhance development process reliability and reduce costs.

For this article we decide to use SysML [Obj10], which is a graphical modelling language, very widely used in the component-based system development. It offers a standard for modelling, specifying and documenting systems. The improvements, brought by SysML, allowed to increase its popularity for industrial and academic environment. A SysML specification of a system is described by structural diagrams and behaviour diagrams. The architecture refinement of a system is an important concept in SysML and it is based on the development of a process from an abstract level towards more detailed levels which can end in an implementation. In our case it is a question of replacing an abstract block in a specification by a composition of blocks preserving its structural properties and its behavioural properties. Structural diagrams of SysML describe the system in static mode and behavioural diagrams describe the dynamic operation of the system. The blocks are modelled by two diagrams, the block definition diagram, which defines the architecture of the blocks and their performed operations, and the internal block diagram used to define the ports of each block and transactions exchanged between them through their ports.

During the refinement process, these two diagrams can be checked to decide whether the proposed architecture satisfies or it is inappropriate to the requirements diagram.

In this paper, we focus on the decomposition of a SysML block into several blocks whose ports are described by interface automata. These interface automata can be derived from behavioural diagrams as proposed in [CH11]. We also exploit requirement diagrams to specify the properties that must be satisfied during the interaction between components(blocks). The interface automata formalism was introduced by L. Alfaro and T.-A. Henzinger [dAH01] and allows to describe a temporal order of required and offered operations calls. One problem may be the existence of anomalies in the interaction between blocks that can lead to illegal states. These states mean that one of the two blocks is requesting a service from the other and that service is not offered.

This compatibility is verified in two ways: consistency in system architecture level and behaviour compatibility at internal block level of the SysML block. Indeed, two interface automata A_1 and A_2 associated with two internal blocks B_1 and B_2 are compatible if there is an environment that prevent them from reaching illegal states during their interaction.

This paper aims to formalize the decomposition process by focusing on verification of architectural and behavioural aspects of SysML blocks. It is organized as follows. In Section 2 we present the preliminaries about SysML specifications

and the interface automata model. In Section 3, we describe our approach which allows to associate interface automata to SysML blocks and give the conditions to be verified between the sub-blocks of a SysML block. In Section 4, we give an example to illustrate our approach. In Section 5 we present some related works, and finally the conclusion, and perspectives are presented in Section 6.

2 Preliminaries

2.1 The SysML Language

SysML (Systems Modelling Language) [Obj10] is a modelling language dedicated to system engineering applications. It was designed as a response to the Request for Proposals (RFP) made in March 2003 by the Object Management Group (OMG) [omg] for using UML in Systems Engineering [Obj03], it was proposed by the OMG and the International Council on Systems Engineering (INCOSE) and was adopted as standard in May 2006. SysML is a UML 2.0 profile [Obj05] that reuses a subset of its diagrams and adds new features to better fit the needs of systems engineering so that it allows the specification, analysis, design, verification and validation of a wide range of complex systems. These systems may include software, hardware, data, processes, people and facilities.

SysML includes 9 diagrams and according to [sys] they can be defined as follows:
Activity Diagram: describes the system behaviour as control and data flow.
Block definition Diagram: describes the architectural structure of the system as components with their properties, operations and relationships.
Internal block Diagram: describes the internal structures of the components, adding parts and connectors.
Package Diagram: describes how a model is organized into packages, views and perspectives.
Parametric Diagram: describes the parametric constraints between the structural elements.
Requirements Diagram: describes the system requirements and their relationships with other elements.
Sequence Diagram: describes the system behaviour as interactions between system components.
State machine Diagram: describes the system behaviour as states that a component has in response to events.
Use Case Diagram: describes the system functions and actors in the process of using them.

2.2 Interface Automata

Interface automata were introduced by Alfaro and Henzinger [dAH01] to model interfaces in an approach to components. These automata are derived from Input/Output automata [LT87] where it is not necessary to have activable input actions in all states. Each component is described by a single interface automaton. The set of actions is decomposed into three groups: input actions, output actions and internal actions. Input actions allow to model the methods to be called in a component, in which case they are the offered services in a component. They can also model a message reception in a communication channel. These actions are labelled by the character "?". The output actions model the method calls from another component. Therefore, they represent services required by a component. They can also model the transmission of messages in a communication channel. These actions are labelled by the character "!". Internal actions are operations that can be activated locally and are labelled by the character ";".

Definition 1 (Interface Automata). *One* interface automaton A *is represented by the tuple* $\langle\ S_A,\ I_A,\ \Sigma_A^I,\ \Sigma_A^O,\ \Sigma_A^H,\ \delta_A\ \rangle$ *such as:*

- S_A *is a finite set of states*;
- $I_A \subseteq S_A$ *is a subset of initial states*;
- Σ_A^I, Σ_A^O *and* Σ_A^H*, respectively denote the sets of input, output and internal actions. The set of actions of* A *is denoted by* Σ_A;
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *is the set of transitions between states.*

The set of input, output and internal actions of an interface automaton are noted ($\Sigma_A = \Sigma_A^I \cup \Sigma_A^O \cup \Sigma_A^H$). We define by $\Sigma_A^I(s)$, $\Sigma_A^O(s)$, $\Sigma_A^H(s)$, respectively the set of input, output and internal activable actions at the state s. $\Sigma_A(s)$ represents the set of activable actions at the state s.

The verification of the assembly of two components (blocks) is obtained by verifying the compatibility of their interface automata. In order to verify the assembly of two components B_1 and B_2, one verifies if there is an environment for which it is possible to assemble correctly B_1 and B_2. This results in the *composition* of their interface automata and its verification if it is not empty.

Two interface automata A_1 and A_2 are *composables* if $\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \cap \Sigma_{A_2} = \Sigma_{A_2}^H \cap \Sigma_{A_1} = \emptyset$. We define by $Shared(A_1,A_2) = (\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_{A_2}^I \cap \Sigma_{A_1}^O)$ is the set of actions shared between A_1 and A_2.

Definition 2 (Synchronized Product). *Let* A_1 *and* A_2 *two composable interface automata. The* synchronized product $A_1 \otimes A_2$ *of* A_1 *and* A_2 *is defined by:*

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ and $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma_{A_1 \otimes A_2}^I = (\Sigma_{A_1}^I \cup \Sigma_{A_2}^I) \ \setminus\ Shared(A_1, A_2)$;
- $\Sigma_{A_1 \otimes A_2}^O = (\Sigma_{A_1}^O \cup \Sigma_{A_2}^O) \ \setminus\ Shared(A_1, A_2)$;
- $\Sigma_{A_1 \otimes A_2}^H = \Sigma_{A_1}^H \cup \Sigma_{A_2}^H \ \cup\ Shared(A_1, A_2)$;
- $((s_1, s_2), a, (s_1', s_2')) \in \delta_{A_1 \otimes A_2}$ if
  - $a \notin Shared(A_1, A_2) \wedge (s_1, a, s_1') \in \delta_{A_1} \wedge s_2 = s_2'$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, a, s_2') \in \delta_{A_2} \wedge s_1 = s_1'$

– $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.

Two interface automata may be incompatible due to the existence of illegal states in their synchronized product. Illegal states are states from which a shared output action from an automaton can not be synchronized with the same enabled action as input on the other component.

Definition 3 (Illegal States). *Let two composable interface automata A_1 and A_2, the set of* illegal states *$Illegal(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ is defined by $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists a \in Shared(A_1, A_2)$. such that the condition C is satisfied$\}$*

$$C = \begin{pmatrix} a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2) \\ \vee \\ a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1) \end{pmatrix}$$

The interface automata approach is considered an optimistic approach, because the reachability of states in *Illegal* (A_1, A_2) does not guarantee the incompatibility of A_1 and A_2. Indeed, in this approach one verifies the existence of an environment that provides appropriate actions to the product $A_1 \otimes A_2$ to avoid illegal states. The states in which the environment can avoid the reachability of illegal states are called compatible states, and are defined by the set *Comp* (A_1, A_2). This set is calculated in $A_1 \otimes A_2$ by eliminating illegal states, unreachable states and the states that lead to illegal states through internal actions or output actions.

Definition 4 (Composition). *The* composition $A_1 \parallel A_2$ *of two automata A_1 and A_2 is defined by*
(i) $S_{A_1 \| A_2} = Comp(A_1, A_2)$,
(ii) $I_{A_1 \| A_2} = I_{A_1 \otimes A_2} \cap Comp(A_1, A_2)$ *and*
(iii) $\delta_{A_1 \| A_2} = \delta_{A_1 \otimes A_2} \cap (Comp(A_1, A_2) \times \Sigma_{A_1 \| A_2} \times Comp(A_1, A_2))$

3 Proposed Approach

In this section we will introduce the steps of our approach: first we present a general explanation of the verification process, then we describe the phases of the consistency and compatibility verification between blocks and finally we propose an algorithm to follow in order to do this verification.

3.1 Approach Overview

Our approach aims to propose a method to formalize and verify SysML blocks in a process of refinement. We show the general process of our approach in Figure 1. From a Block Definition Diagram (BDD), we decompose it into several sub-blocks and then we perform verifications, under some conditions, to the system architecture and its behaviour. For each block we describe its internal block diagram to show the interactions between sub-blocks that compose it. In order to verify the consistency between a block and its sub-blocks, we verify the conditions of consistency and compatibility during their interactions. To achieve this verification, we associate an interface automaton to each sub-block by using its behaviour specified by sequence diagram and by exploiting the approach proposed in [CH11]. We exploit also the requirements diagram in order to identify requirements that should be satisfied by the compatibility verification between sub-blocks. These requirements identify the properties of interoperability that must hold in the component based system in order to ensure a reliable interaction between components. This allows us to define a relationship (that we call also traceability) between requirements level and formal verification level. We can verify the compatibility and the composability by means of a checking tool like Ticc [ADADS+06].

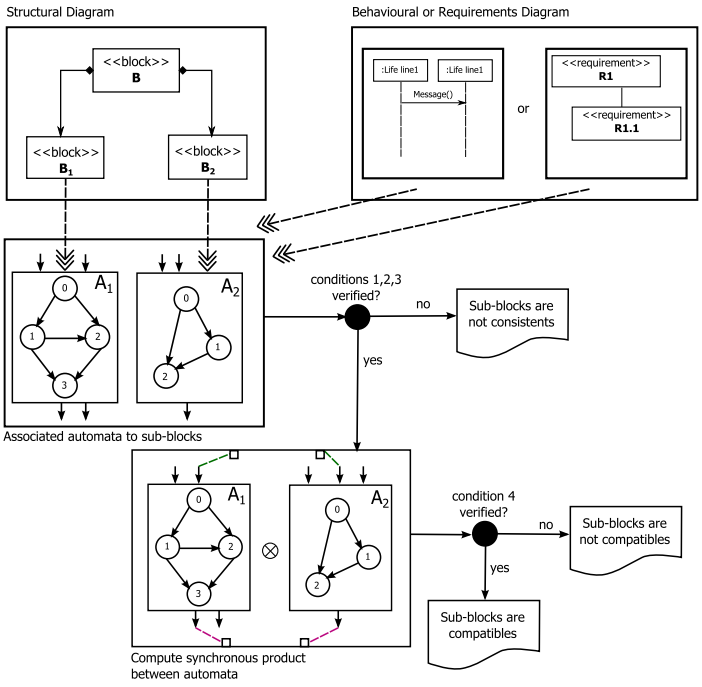

Figure 1: Proposed approach for verifying consistency and compatibility in SysML blocks

3.2 Consistency Verification between Blocks

The consistency conditions between blocks allow us to determine if the inputs of the abstract block are taken by the sub-blocks which compose it. Similarly it is verified whether or not the outputs of the abstract block are provided by the outputs of the sub-blocks. Composability ensures that the blocks in question do not share the same inputs and the same outputs. These conditions result in the following:

We consider two interface automata A_1 and A_2 associated to the sub-blocks B_1 and B_2 (figure 1), Σ^I_B as the input actions and Σ^O_B as the output actions of the abstract block.

- **Condition 1 (Composability)**:

This condition ensures to compose A_1 and A_2 and to apply the interface automata theory to verify their compatibility
$\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \cap \Sigma_{A_2} = \Sigma^H_{A_2} \cap \Sigma_{A_1} = \emptyset$.

- **Condition 2 (At least the same inputs)**:
  This condition ensures that the sub-blocks B_1 and B_2 offer at least the same services (inputs) as the bloc B
  $\Sigma^I_B \subseteq \Sigma^I_{A_1} \cup \Sigma^I_{A_2} \setminus Shared(A_1, A_2)$

- **Condition 3(At most the same outputs)**:
  The sub-blocks B_1 and B_2 require at most the same services (outputs) as the abstract block B
  $\Sigma^O_{A_1} \cup \Sigma^O_{A_2} \setminus Shared(A_1, A_2) \subseteq \Sigma^O_B$

These conditions allow us to apply the approach of interface automata in order to verify then the compatibility of the blocks.

3.3 Compatibility Verification between blocks

The compatibility verification between two blocks B_1 and B_2 is obtained by verifying the compatibility between their interface automata A_1 et A_2. To verify the compatibility between two blocks B_1 and B_2, this approach verifies if there is an environment where it is possible to correctly assemble B_1 and B_2. Thus, we assume the existence of an environment that accepts all output actions of the synchronized product automaton of A_1 and A_2, and does not trigger any input action of $A_1 \otimes A_2$.

Condition 4 (Compatibility): Two interface automata A_1 and A_2 are compatible if and only if their composition $A_1 \parallel A_2$ has at least one reachable state.

The composition $A_1 \parallel A_2$ is calculated according to definition 4 and by applying the algorithm described in [dAH01].

3.4 Verification Algorithm of the Consistency and Compatibility between Blocks

The Algorithm 1 shows the pseudo-code to implement our approach, first we verify the consistency between the blocks and if they are consistent we continue to verify their compatibility.

The complexity of this algorithm is in time linear on the size of the interface automata and is given by $O(|A_1 \times A_2|)$. In fact, the complexity of the compatibility verification is $O(|A_1 \times A_2|)$ [dAH01] and we can easily verify that the complexity of the verification of conditions 1,2 and 3 do not increase this complexity.

Algorithm 1: Verification of the consistency and the compatibility between blocks

Data: two sub-blocks B_1 and B_2 with associated interface automata A_1 and A_2
Result: compatibility between B_1 and B_2?

Consistency verification:

1. Verify the condition 1 (composability)
2. Verify the condition 2 (at least the same inputs)
3. Verify the condition 3(at most the same outputs)

if *one of the conditions is not verified* **then**
  B_1 and B_2 are inconsistents;
else
  Compatibility verification:

  1. Compute the product $A_1 \otimes A_2$
  2. Compute the product $A_1 \parallel A_2 = A_1 \otimes A_2 - Illegal(A_1, A_2)$ and remove the unreachable states
  3. Verify the condition 4

  if *condition 4 is verified* **then**
    B_1 and B_2 are compatible
  else
    B_1 and B_2 are incompatible
  end
end

4 Case Study

To illustrate our approach, we apply the decomposition to a SysML model that specifies the implementation of a security system in a car. Figure 2 shows the requirements diagram associated to the security requirements; in one side the prevention requirements composed by several control elements like the tire pressure checking, the oil level checking of the braking system, the locking state of safety belts by presence of passengers and the prevention of speed excess; and on the other side the reaction requirements to be implemented if the car has an accident such as locking seat belts and airbags inflating. We also have in the reaction side, one requirement to validate the interoperability between an acceleration sensor and a control point that we are going to verify following our interface automata approach.

The block definition diagram associated with this study is shown in Figure 3, it refers to the reaction requirement and is represented by an abstract block named *Control system* that eventually we propose to decompose in four different sub-blocks : *Acceleration sensor*, *Control point*, *Airbag system* and *Seatbelt lock*. As said before, we will verify this proposed decomposition by following our approach through interface automata, at that point we also need to model the internal composition of the abstract block by specifying its internal block definition diagram that we present in Figure 4.

Figure 4 shows the connections between two of the blocks that decompose the abstract block, the input of the abstract block will be taken by the block *Acceleration sensor* and its outputs will be provided by the block *Control point*; the

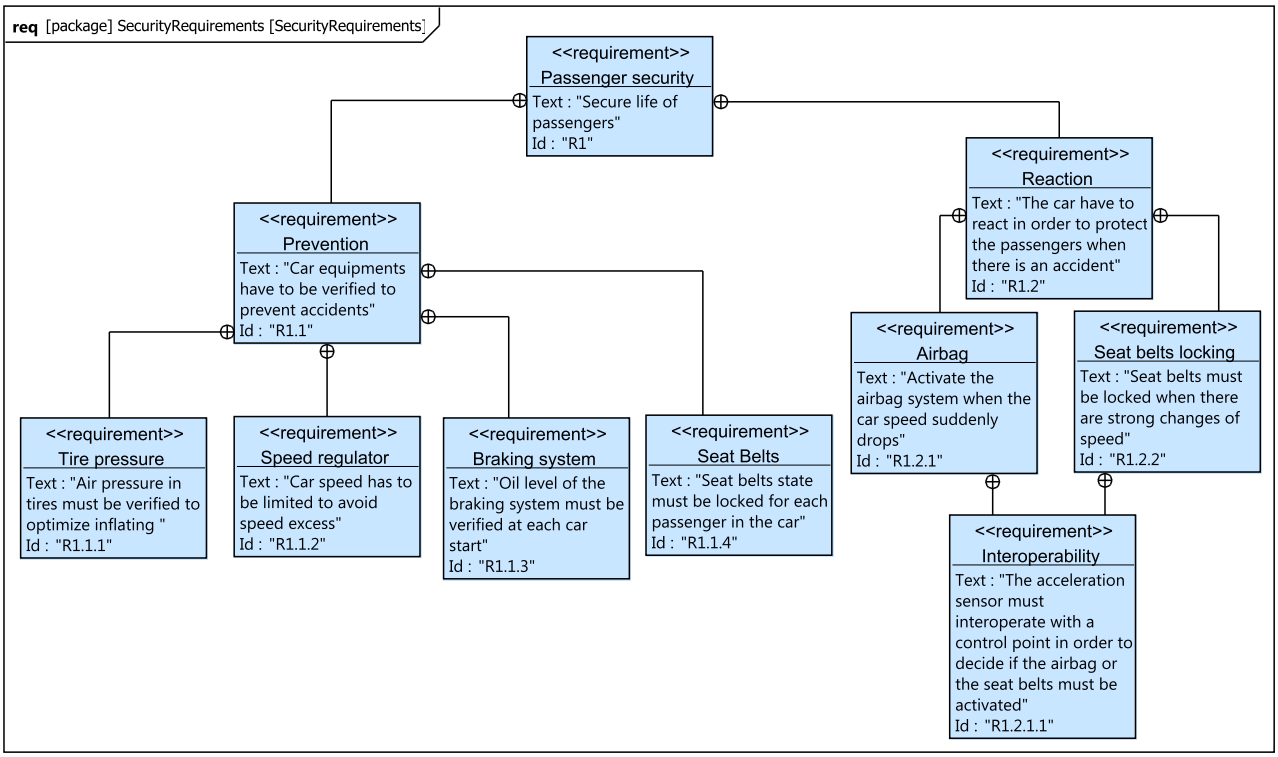

Figure 2: Requirements diagram of the security in a car

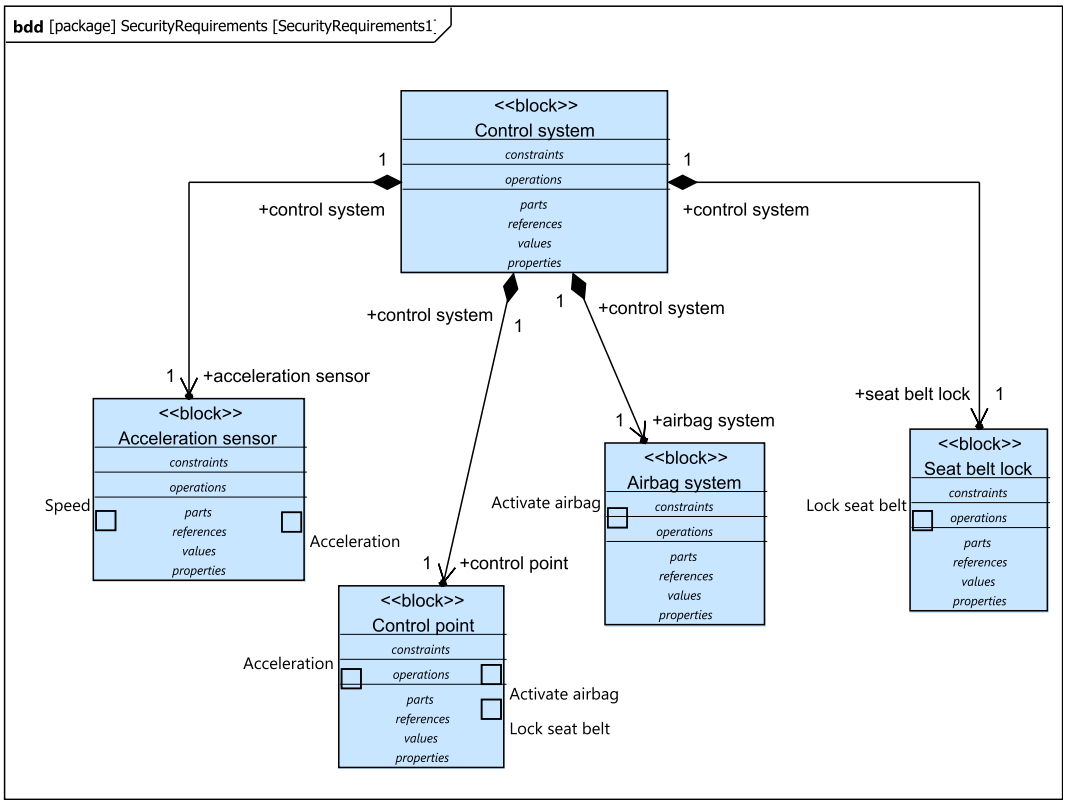

Figure 3: Block definition diagram of the security in a car

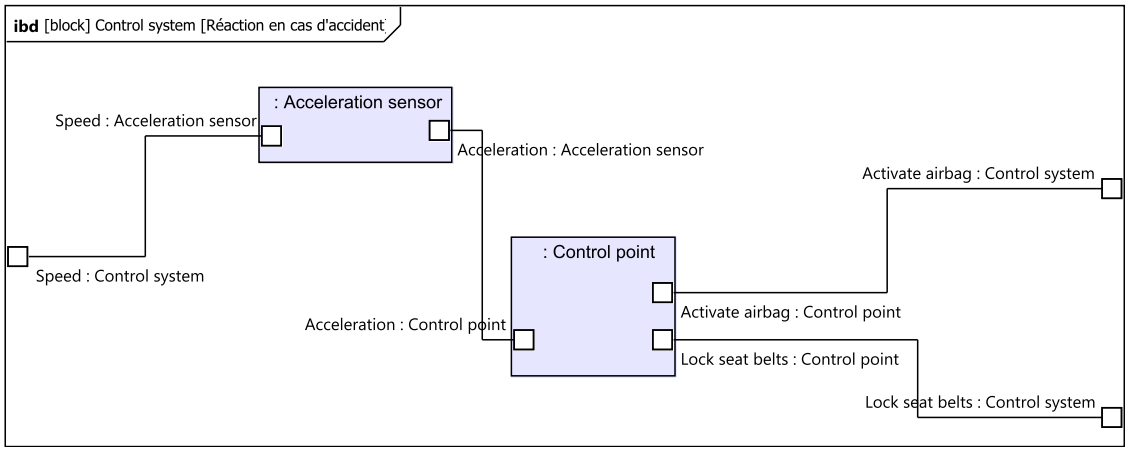

Figure 4: Internal block diagram of the security in a car

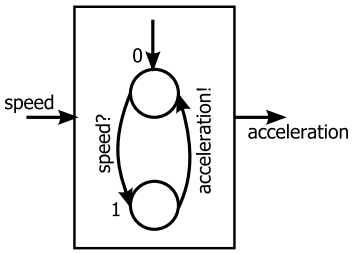

Figure 5: Interface automaton A_1 associated to the *acceleration sensor*

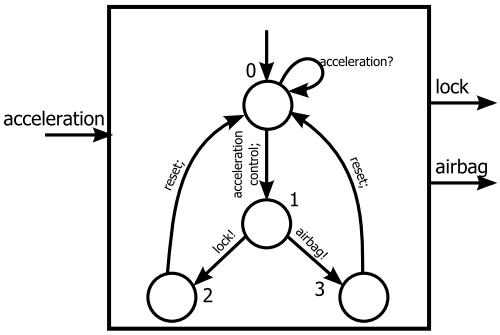

Figure 6: Interface automaton A_2 associated to the *Control point*

interaction between the two sub-blocks will be given through the connector *acceleration*.

From the internal block definition diagram shown in Figure 4 and the definition of the requirements in Figure 2 we associate the interface automata A_1 and A_2 shown in Figures 5 and 6, the actions of these automata are deduced from the requirements diagram and it is not automatic, in this example we show only the automata composition for a controller that will activate the seat belts lock and the airbag.

We verify the conditions of our approach on the block *Control system* and the sub-blocks *Acceleration sensor* and *Control point*.

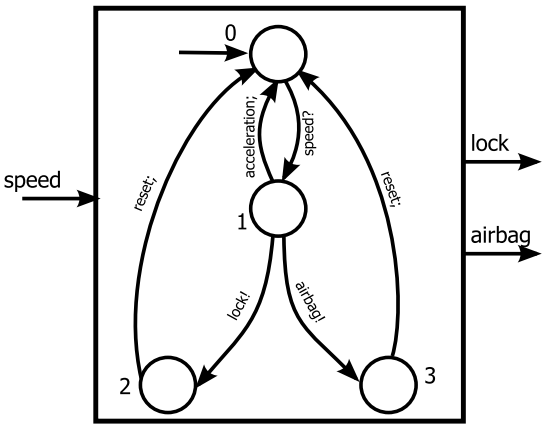

Figure 7: Product $A_1 \parallel A_2$ between the automata *Acceleration sensor* and *Control point*

4.1 Consistency Verification

To verify the consistency of the associated automata, we first check their composability as exposed in condition 1 looking at the inputs, outputs and internal actions of each sub-block as follows:
$\Sigma^I_{A_1} = \{speed\}$, $\Sigma^O_{A_1} = \{acceleration\}$, $\Sigma^H_{A_1} = \emptyset$
$\Sigma^I_{A_2} = \{acceleration\}$, $\Sigma^O_{A_2} = \{lock, airbag\}$, $\Sigma^H_{A_2} = \{accelerationcontrol, ok\}$
$Shared(A_1, A_2) = \{acceleration\}$
and we do not find inputs or outputs that are present simultaneously in both automata, ie. $\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \cap \Sigma^H_{A_2} = \emptyset$; then we check the condition 2 and we find that the input $\Sigma^I_B = \{speed\}$ of the abstract block *Control system* is present in the automaton *Acceleration sensor*, ie. $\Sigma^I_B \subseteq \Sigma^I_{A_1} \cup \Sigma^I_{A_2} \setminus Shared(A_1, A_2)$; and finally for the condition 3 we check that the outputs $\Sigma^O_B = \{lock, airbag\}$ of the abstract block are the same outputs required associated to the sub-blocks present in the interface automaton associated

to the block *Control point*, ie. $\Sigma^O_{A_1} \cup \Sigma^O_{A_2} \setminus Shared(A_1, A_2) \subseteq \Sigma^O_B$. We can therefore conclude that they are consistent.

4.2 Compatibility Verification

To verify the compatibility of the interface automata associated to the two blocks, we compute the synchronous product $A_1 \otimes A_2$ and we eliminate unreachable states to obtain the product $A_1 \parallel A_2$ as indicated by condition 4 of our approach, this synchronous product is given in Figure 7. The calculated automaton contains no illegal states and therefore they are compatible. We deduce that the two blocks *Acceleration sensor* and *Control point* are consistent and compatible and that the block *Control system* can be decomposed into the sub-blocks *Acceleration sensor* and *Control point*. Therefore we conclude that the Interoperability requirement R1.2.1.1 present in the figure 2 is satisfied.

5 Related Works

In the field of components, several approaches have been proposed including those of Szyperski in [Szy99], Medvidovic in 2000 [MT00]. Most models consider the components with their behaviour, connectors and services that are provided or required. The assembly operation of components may occur at different levels of abstraction, from the design of software architectures DSA to the implementation in platforms such as CORBA or .NET. The crucial question that arises to the developer is whether the proposed assembly is valid or not.

In our case, we are interested in SysML blocks specified by their interfaces and their behaviours modelled using automata. We can cite as examples the model of Allen 1997 [AG97] where the protocols are associated with component connectors. Attie in 2006 [ALPC06] combines protocols to interfaces connecting two components. Others, like Becker in 2004 [SSR04] propose a framework for comparing models with three levels of interoperability using the signatures, the protocols associated to the components and quality of service. The protocols of Giannakopoulou et al. 1999 [MKG99] are based on works on automata and competition using the formalism of transition systems, including the analysis of reachability. The composition operation is essential to define the assembly and verify the safety and liveliness properties.

The approach of Moizan et al. 2003 [MRR03] aims to provide UML components with the specification of their protocols. The behaviour description language is based on hierarchical automata inspired by StateCharts. It can support mechanisms for composition and refinement of behaviours. Properties are specified in temporal logic.

In Ardourel in 2005 [AAA05], the authors define a model Kmelia of abstract components with services, who do not take into account the data during the interaction. The behaviours are described by automata associated to services. This environment uses the MEC model checker tool to verify compatibility of components.

Other works deal with the inclusion of real-time constraints as in Etienne et Bouzefrane in 2006 [EB06]. It aims to determine the characteristics of components and to define some criteria to verify compatibility of their specifications during the assembly phase using the tool Kronos.

Our approach allows to analyse the consistency, composability and compatibility between blocks. It combines semi-formal models based on SysML and formal models based on interface automata for correct assembly between blocks.

6 Conclusion and Perspectives

We have shown in this article how to verify the compatibility between SysML blocks. This compatibility can be guaranteed in two ways: behavioural and architectural consistency. Behavioural compatibility uses the model of interface automata proposed by L. de Alfaro and T.-A. Henzinger. It is based on the detection of illegal states during the interaction between blocks. It also allows us to show how to refine an abstract SysML block into several sub-blocks. This approach defines also a traceability between requirement diagram and the compatibility verification step. Indeed, we show that the verification step allows to verify whether the system requirements are satisfied or not.

We plan to continue this work in two directions. The first is to define a new SySML profile in order to associate formal properties to the system requirements, and to consider theme in the interface automata approach. So, we plan to define a formal specification for these properties close to interface automata model, in order to integrate easily these properties in the verification step. The second line of research is to automate this approach using the platform TopCased to make it operational. Our goal is to generate specifications that will be used by the tool TICC [tic, ADADS$^+$06] to verify compatibility between SysML blocks by using an optimistic approach. The diagnosis obtained will be used to detect any anomalies and to assist the designer to correct specifications for compatible blocks. When the blocks are shown incompatible, we plan to used our published works in [CMM12] to generate blocks adapters to make compatible the incompatible blocks.

References

[AAA05] Pascal André, Gilles Ardourel, and Christian Attiogbé. Behavioural Verification of Service Composition. In *ICSOC Workshop on Engineering Service Compositions, WESC'05*, pages 77–84, Amsterdam, The Netherlands, 2005. IBM Research Report RC23821.

[ADADS$^+$06] B. Adler, L. De Alfaro, L. Da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc: A tool for interface compatibility and composition. In *Computer Aided Verification*, pages 59–62. Springer, 2006.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[ALPC06] Paul Attie, David H. Lorenz, Aleksandra Portnova, and Hana Chockler. Behavioral compatibility without state explosion: Design and verification of a component-based elevator control system. In I. Gorton et al., editor, *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, number 4063 in LNCS, pages 33–46. Springer Verlag, 2006.

[CH11] Samir Chouali and Ahmed Hammad. Formal verification of components assembly based on sysml and interface automata. *ISSE*, 7(4):265–274, 2011.

[CMM12] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adaptation sémantique des protocoles des composants par les automates d'interface. *TSI - Technique et Science Informatiques*, 31(*):***–***, 2012. Papier accepté. À paraître.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120. Press, 2001.

[EB06] J.-P. Etienne and S. Bouzefrane. Vers une approche par composants pour la modélisation d'applications temps réel. In *(MOSIM'06) 6ème Conférence Francophone de Modélisation et Simulation*, pages 1–10, Rabat, 2006. Lavoisier.

[LT87] N. Lynch and M. Tuttle. Hierarcical correctness proofs for distributed algorithms. *In the proceeding of the 6th ACM Symp. Principles of Distributed Computing*, pages 137–151, 1987.

[MKG99] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 35–50, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[MRR03] S. Moisan, A. Ressouche, and J. Rigault. Behavioral substitutability in component frameworks: A formal approach, 2003.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[Obj03] The Object Mangagement Group (OMG). *UML for Systems Engineering. Request for Proposal*, 2003.

[Obj05] Object Management Group. *The OMG Unified Modeling Language Specification, UML 2.0*, 2005.

[Obj10] The Object Mangagement Group (OMG). *OMG Systems Modeling Language (OMG SysML) Specification Version 1.2*, 2010. http://www.omg.org/spec/SysML/1.2/.

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

[SSR04] Becker Steffen, Overhage Sven, and Reussner Ralf. Classifying software component interoperability errors to support component adaption. In Crnkovic Ivica, Stafford Judith, Schmidt Heinz, and Wallnau Kurt, editors, *Component Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, pages 68–83. Springer, 2004.

[sys] SysML FAQ - what is the relationship between SysML and UML? http://www.sysmlforum.com/faq/relation-between-SysML-UML.html.

[Szy99] C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.

[tic] http://dvlab.cse.ucsc.edu/Ticc.