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 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.
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 \((S_A, I_A, \Sigma^I_A, \Sigma^O_A, \Sigma^H_A, \delta_A)\) such as:

- \( S_A \) is a finite set of states;
- \( I_A \subseteq S_A \) is a subset of initial states;
- \( \Sigma^I_A, \Sigma^O_A \) and \( \Sigma^H_A \) respectively denote the sets of input, output and internal actions.
- The set of actions of \( A \) is denoted by \( \Sigma_A \);
- \( \delta_A \subseteq S_A \times \Sigma_A \times S_A \) is the set of transitions between states.

The set of input, output and internal actions of an interface automaton are noted \((\Sigma_A = \Sigma^I_A \cup \Sigma^O_A \cup \Sigma^H_A)\). We define by \( \Sigma_A(s), \Sigma^I_A(s), \Sigma^O_A(s), \Sigma^H_A(s) \), 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 \( \Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \cap \Sigma^H_{A_2} = \emptyset \). We define by \( \text{Shared}(A_1, A_2) = (\Sigma^I_{A_1} \cap \Sigma^O_{A_2}) \cup (\Sigma^I_{A_2} \cap \Sigma^O_{A_1}) \) is the set of 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 \( A_1 \times A_2 \) is defined by:

- \( S_{A_1 \times A_2} = S_{A_1} \times S_{A_2} \) and \( I_{A_1 \times A_2} = I_{A_1} \times I_{A_2} \);
- \( \Sigma^I_{A_1 \times A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus \text{Shared}(A_1, A_2) \);
- \( \Sigma^O_{A_1 \times A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus \text{Shared}(A_1, A_2) \);
- \( \Sigma^H_{A_1 \times A_2} = \Sigma^H_{A_1} \cup \Sigma^H_{A_2} \cup \text{Shared}(A_1, A_2) \);
- \( ((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \times A_2} \) if:
  - \( a \notin \text{Shared}(A_1, A_2) \land (s_1, a, s'_1) \in \delta_{A_1} \land s_2 = s'_2 \)
  - \( a \notin \text{Shared}(A_1, A_2) \land (s_2, a, s'_2) \in \delta_{A_2} \land s_1 = s'_1 \)
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 $\text{Illegal}(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ is defined by \{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists a \in \text{Shared}(A_1, A_2). such that the condition $C$ is satisfied\}

$$C = \left( a \in \Sigma_{A_1}(s_1) \land a \notin \Sigma_{A_2}(s_2) \lor a \in \Sigma_{A_2}(s_2) \land a \notin \Sigma_{A_1}(s_1) \right)$$

The interface automata approach is considered an optimistic approach, because the reachability of states in $\text{Illegal}(A_1, A_2)$ 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 $\text{Comp}(A_1, A_2)$. 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) $S_{A_1 \parallel A_2} = \text{Comp}(A_1, A_2)$,

(ii) $I_{A_1 \parallel A_2} = I_{A_1} \otimes I_{A_2} \cap \text{Comp}(A_1, A_2)$ and

(iii) $\delta_{A_1 \parallel A_2} = \delta_{A_1} \otimes \delta_{A_2} \cap (\text{Comp}(A_1, A_2) \times \Sigma_{A_1} \parallel \Sigma_{A_2} \times \text{Comp}(A_1, A_2))$

**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].

**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:

- **Condition 1 (Composability):**
This condition ensures to compose $A_1$ and $A_2$ and to apply the interface automata theory to verify their compatibility

$$\Sigma_{A_1} \cap \Sigma_{A_2} = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \cap \Sigma_{A_2}^H = \Sigma_{A_1} \cap \Sigma_{A_2} = \emptyset.$$ 

- **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 $B$
  $$\Sigma_B \subseteq \Sigma_{A_1}^I \cup \Sigma_{A_2}^I \setminus \text{Shared}(A_1, A_2)$$

- **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 $B$
  $$\Sigma_{A_1}^O \cup \Sigma_{A_2}^O \setminus \text{Shared}(A_1, A_2) \subseteq \Sigma_B^O$$

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.

### 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
interaction between the two sub-blocks will be given through the connector \textit{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 \textit{Control system} and the sub-blocks \textit{Acceleration sensor} and \textit{Control point}.

\subsection*{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:
\begin{align*}
\Sigma_{A_1}^I &= \{\text{speed}\}, \quad \Sigma_{A_1}^O = \{\text{acceleration}\}, \quad \Sigma_{A_1}^H = \emptyset \\
\Sigma_{A_2}^I &= \{\text{acceleration}\}, \quad \Sigma_{A_2}^O = \{\text{lock, airbag}\}, \quad \Sigma_{A_2}^H = \{\text{acceleration control, ok}\}
\end{align*}
\begin{align*}
\text{Shared}(A_1, A_2) &= \{\text{acceleration}\}
\end{align*}
and we do not find inputs or outputs that are present simultaneously in both automata, ie. $\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \emptyset$, $\Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \emptyset$, $\Sigma_{A_1}^H \cap \Sigma_{A_2}^H = \emptyset$; then we check the condition 2 and we find that the input $\Sigma_{B}^I = \{\text{speed}\}$ of the abstract block \textit{Control system} is present in the automaton \textit{Acceleration sensor}, ie. $\Sigma_{B}^I \subseteq \Sigma_{A_1}^I \cup \Sigma_{A_2}^I \setminus \text{Shared}(A_1, A_2)$; and finally for the condition 3 we check that the outputs $\Sigma_{B}^O = \{\text{lock, airbag}\}$ 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, i.e., $\Sigma_{A_1} \cup \Sigma_{A_2} \setminus \text{Shared}(A_1, A_2) \subseteq \Sigma_B$. 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 Kmedia 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, compositionality 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 incompatibl.

### References


http://dvlab.cse.ucsc.edu/Ticc.