# SysML Model-Driven Approach to Verify Blocks Compatibility

H. Bouaziz*, S. Chouali, A. Hammad and H. Mountassir

FEMTO-ST Institute,
University of Bourgogne Franche-Comté,
Besançon, France
E-mail: hamida.bouaziz@femto-st.fr
E-mail: schouali@femto-st.fr
E-mail: ahammad@femto-st.fr
E-mail: hmountas@femto-st.fr
*Corresponding author

**Abstract:**
In the component paradigm, the system is seen as an assembly of heterogeneous components, where the system reliability depends on these components compatibility. In our approach, we focus on verifying compatibility of components modelled with SysML diagrams. Thus, we model component interactions with sequence diagrams (SDs) and components with SysML blocks. The SDs constitute a good start point for compatibility verification. However, this verification is still inapplicable directly on SDs, because they are expressed in informal language. Thus, to apply a verification method, it is necessary to translate the SDs into formal models, and then verify the wanted properties. In this paper, we propose a high-level model-driven approach which consists of an ATL grammar that automates the transformation of SDs into interface automata. Also, to allow an easy use of Ptolemy tool to verify properties on automata, we have proposed some Acceleo templates, which generate the Ptolemy entry specification.

**Keywords:** model-driven; SysML; sequence diagram; interface automata; ATL; Acceleo

**Reference** to this paper should be made as follows: Bouaziz, H., Chouali, S., Hammad, A. and Mountassir, H.(xxxx) 'SysML Model-Driven Approach to Verify Blocks Compatibility', *Int. J. Computer Aided Engineering and Technology*, Vol. x, No. x, pp.xxx–xxx.

**Biographical notes:** Hamida Bouaziz is a PhD student at Femto-ST institute, university of Bourgogne Franche-Comté, France. Her area of research includes component-based systems, modelling of complex systems using SysML, adaptation techniques of components and formal verification.

Samir Chouali is an associate professor at the university of Bourgogne Franche-Comté. He is a member of the department of computer science and complex systems (DISC) in FEMTO-ST institute. His research interests include the use of formal methods in the specification and the verification of complex systems, the development and the verification of component-based systems (CBS), and the combination between semi-formal models (UML, SysML) and formal approaches to develop reliable CBS.

Ahmed Hammad received his PhD degree in Computer Science from the university of Toulouse, France. Currently, he is an associate professor at the

university of Bourgogne Franche-comté, and a member of the department of computer science and complex systems (DISC) in FEMTO-ST institute. His area of research includes the use of formal methods in the specification and the verification of complex systems. His research interests concern also Model-Driven Engineering (MDE) and model transformation.

Hassan Mountassir received his HDR (French post-doctoral degree allowing its holder to supervise PhD students) degree in Software Engineering in 2001 from Bourgogne Franche-Comté university. Currently, he is a professor in the department of Computer Sciences at Femto-ST institute. His research interests include component-based systems, formal methods and verification techniques.

## 1 Introduction

The design and the development of large software and systems are addressed by the introduction of new paradigms such as object and component paradigms. The use of components as the development unit allows handling the complexity of these large systems. Basing on the notion of component, OMG and INCOSE have founded the System Modelling Language (SysML) OMG (2012). This language shows the system as a set of blocks OMG (2012). SysML uses the Block Definition Diagram (BDD) and the Internal Block Diagram (IBD) to structure the blocks and to establish links between them. To model the behaviour, SysML uses the State Machine (SM), the Sequence Diagram (SD) and the Activity Diagram (AD).

In SysML, interactions between blocks are modelled using IBDs and SDs. These interactions take the form of architectural links in the IBDs. However, SDs, which interest us in this paper, allow us to model the scheduling of these interactions using life lines of blocks. Thus, the SDs constitute a good start point to verify the interactions inside the system. Since formal verification is still inapplicable directly on SysML models (Bouaziz et al. (2015)), therefore to apply a verification method, it is necessary to translate the SysML models to formal ones, and then verify the wanted properties.

In our work, the interactions are represented using a set of SDs. Each SD is associated with a block, and it describes the interaction scenarios of a block with its environment. To formalize the semantic of SDs, we transform them into interface automata (IAs) (de Alfaro and Henzinger (2001)). IAs constitute a good formal model to represent the scenarios of requesting (output actions) and performing (input actions) services of a block. The composition of interface automata allows us to verify some relations and properties on blocks such as the consistency and the compatibility (i.e. verify the existence of an environment where it is possible to connect these blocks).

Our approach of transforming SDs into IAs is mainly based on meta-modelling (de Lara et al. (2004)) and meta-model transformations (Czarnecki and Helsen (2003)). Such approach consists on defining the meta-models of the source and the target models, and then specifying the correspondences between them in the meta-level. To avoid user errors during the transformation from the SDs to IAs, we have proposed an automated ATL (ATL, n.d.) grammar, which performs this transformation automatically. After, to verify some properties on the resulted interface automata, we have opted for Ptolemy (Ptolemy, n.d.) tool that requires as entry a textual specification. For this purpose, we have used Acceleo (Acceleo, n.d.) to define a set of templates on our meta-model of interface automata, that allows us to

generate automatically the Ptolemy entry specification. Thus, this tool chain, that we have developed, allows to assist the architect during the verification phase by discharging him from doing many tasks.

The remainder of the paper is organized as follows: In section 2, we present the background about SysML sequence diagrams, interface automata, ATL and Acceleo. Next, in section 3, we introduce our proposed approach. After, Section 4, gives details of transforming sequence diagrams into interface automata. Next, Section 5, presents the Acceleo templates that we have proposed to generate Ptolemy entry specification. In Section 6, we present how we verify the compatibility of the blocks. In section 7, we illustrate our approach by a case study. In section 8, we discuss related work. Finally, in Section 9, we conclude and we present perspectives of our work.

## 2 Preliminaries

### *2.1 Sequence Diagram*

Sequence diagram (see figure 3) is a graphical diagram of SysML. It represents the interactions by focusing on the observable exchange of messages between blocks. A sequence diagram has two dimensions, where the vertical dimension represents time and the horizontal one represents the blocks which participate in the interaction (Rumbaugh et al. (2004)). It consists of a set of lifelines which represent the interacting blocks. The temporal execution of interactions is shown as a succession of messages. A message takes the form of an arrow originates at the sender and ends at the receiver. An SD can also contain a set of combined fragments (CFs). CFs are used to express different types of control flows, such as concurrency, choice and loop (Rumbaugh et al. (2004)). They are defined by interaction operators (Alt, Loop, Break, etc) and corresponding interaction operands.

### *2.2 Interface Automata*

Interface automata (de Alfaro and Henzinger (2001)) were introduced by Alfaro and Henzinger to specify components interfaces and also to verify components assembly based on their actions. The set of actions is decomposed into three groups: input actions, output actions and internal actions. Input actions allow us to model the methods that the component exposes to its environment. These actions are labelled by the character '?'. The output actions model the methods that the component needs to invoke from other components. These actions are labelled by the character '!'. Internal actions are methods that can be activated locally and are labelled by the character ';'.

An *interface automaton* $A$ is represented by the tuple:

$$\langle \, \mathrm{S}_A, \mathrm{I}_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \, \rangle$$

Where:

- $S_A$ is a finite set of states. $I_A \subseteq S_A$ is a set of initial states.
- $\Sigma_A^I, \Sigma_A^O$, and $\Sigma_A^H$, 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.

**Definition 1** (Synchronous product):

The *synchronous product* is used to capture the parallel execution of two components represented by their interface automata. Before computing the global behaviour of the two components, it is mandatory to verify if they can be assembled by testing their composability. Two interface automata $A_1$ and $A_2$ are composable if:

$$\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma^H_{A_2} = \emptyset.$$

The synchronous product between *two interface automata* $A_1$ and $A_2$ is defined as:

$$A_1 \otimes A_2 = \langle \, \mathrm{S}_{A_1 \otimes A_2}, \mathrm{I}_{A_1 \otimes A_2}, \Sigma^I_{A_1 \otimes A_2}, \Sigma^O_{A_1 \otimes A_2}, \Sigma^H_{A_1 \otimes A_2}, \delta_{A_1 \otimes A_2} \, \rangle$$

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

We define by *Shared*($A_1$,$A_2$) = $(\Sigma^I_{A_1} \cap \Sigma^O_{A_2}) \cup (\Sigma^O_{A_1} \cap \Sigma^I_{A_2})$ the set of shared actions between $A_1$ and $A_2$.

**Definition 2** (Parallel composition):

The composition of two interface automata $\mathrm{A}_1$ and $\mathrm{A}_2$ is denoted by $\mathrm{A}_1 \parallel \mathrm{A}_2$, it is computed by eliminating from the product $\mathrm{A}_1 \otimes \mathrm{A}_2$ the illegal states and all states reached from these illegal states by enabling output and internal actions. $\mathrm{A}_1$ and $\mathrm{A}_2$ are compatible iff $A_1 \| A_2 \neq \emptyset$

The set of *illegal states* of two interface automata $A_1, A_2$ is defined as:

$$Illegal(A_1, A_2) = \left\{ \begin{array}{c} (s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists a \in Shared(A_1, A_2). \\ \begin{pmatrix} a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2) \\ \vee \\ a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1) \end{pmatrix} \end{array} \right\}$$

We define by $\Sigma^I_A(s_1)$, $\Sigma^O_A(s_1)$, respectively the set of input and output actions enabled at the state $s_1$.

### *2.3 ATL: Atlas Transformation Language*

ATL (ATL, n.d.) is a model transformation language and toolkit. In the field of Model-Driven Engineering (MDE), ATL provides a way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. These rules are based on a mixture of declarative and imperative constructs. The set of the rules constitutes the ATL grammar.

Each ATL rule is characterized by two mandatory elements:

- **`from`** : A pattern on the source model with possible constraints.
- **`to`** : One or more elements of the target model, it indicates how target elements must be initialized from the corresponding source element.

### *2.4 Acceleo*

Acceleo is the result of several man-years of R&D started in the French company Obeo (Obeo, n.d.). Acceleo is a source code generator of the eclipse foundation. It implements the MDA (Model driven architecture) approach to realize application starting from EMF (Eclipse Modelling Framework) models. It is an implementation of the norm of the Object Management Group (OMG) for transforming models to text (M2T), where the transformations take the form of templates.

## 3 Our Methodology

Our approach aims to prepare the SysML blocks for the compatibility verification phase. We

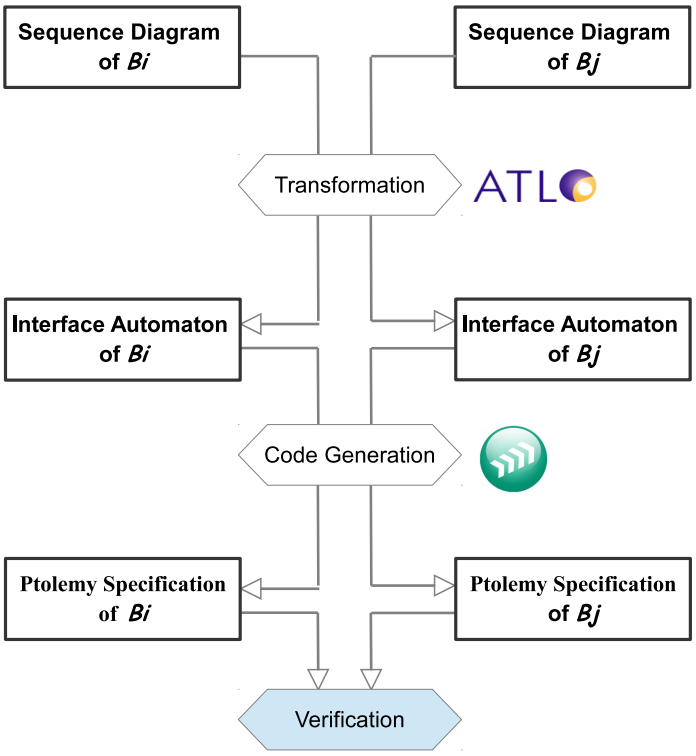


**Figure 1**: Our Methodology

show an overview of our methodology in figure 1. To verify the compatibility of two blocks, modelled using sequence diagrams, we start by applying the ATL grammar, that we will expose in section 4, on their corresponding meta-models , in order to obtain their equivalents

of interface automata. For verifying the compatibility of the blocks, we use the Ptolemy tool. Ptolemy contains a module which allows the verification and the composition of interface automata. To discharge the user from redrawing the interface automata using the Ptolemy user interface, we propose a set of Acceleo templates to generate automatically the Ptolemy entry specifications corresponding to the interface automata specifications obtained in the previous step.

## 4 Transforming SDs of blocks into interface automata

In our work, the sequence diagrams are used to visualize the scheduling of the different interactions of each block with its environment. In the sequence diagram of a block *B*, the environment life line will represent the set of all blocks with which the block *B* can interact. Thus, in our context, we aggregate all the blocks that interact with *B* in one block that we call *ENV*.

To transform these sequence diagrams into interface automata, some correspondences are given in (Chouali and Hammad (2011)). In our paper, we analyse more constructs with more detail. Also, our approach is a meta-model driven approach that define the correspondences in meta-level. To implement these correspondences and to automate the transformation, we propose a set of ATL rules. Our ATL grammar doesn't deal with combined fragments as an isolated units as in the works have already done on Petri nets and the other kinds of automata. It deals with the different cases of nested combined fragments.

This ATL grammar is defined on the meta-model of sequence diagram as source and meta-model of interface automata as target.

### *4.1 Sequence Diagram Meta-Model*

By intention to reuse existed modelling tools, we have used the sub set of Papyrus (Papyrus, n.d.) SysML meta-model and its graphical editor to draw the sequence diagrams. In figure 2, we represent the classes set of Papyrus meta-model that allows us to model sequence diagrams, and in figure 3, we give an example of a sequence diagram which is modelled using Papyrus editor.

In figure 2, the root class is the class *Interaction*. So, sequence diagrams, that we will model, will be the instances of this class. Each interaction can include a set of life lines, a set of messages and finally a set of interaction fragments. The classes:

- **LifeLine:** each instance of this class represents an object which participates in the interaction. It will be the support of sending (resp. receiving) events executed (resp. intercepted) by the object.
- **Message:** defines the messages set interchanged between objects. Each message has two ends; a send end and a receive end.
- **InteractionFragment:** is the super class of the classes: Interaction, CombinedFragment, InteractionOperand and OccurenceSpecification.
- **CombinedFragment:** each combined fragment includes a set of interaction operands, and it has its own interaction operator. The interaction operator takes a value of this list [alt, opt, break, loop, par, ...]

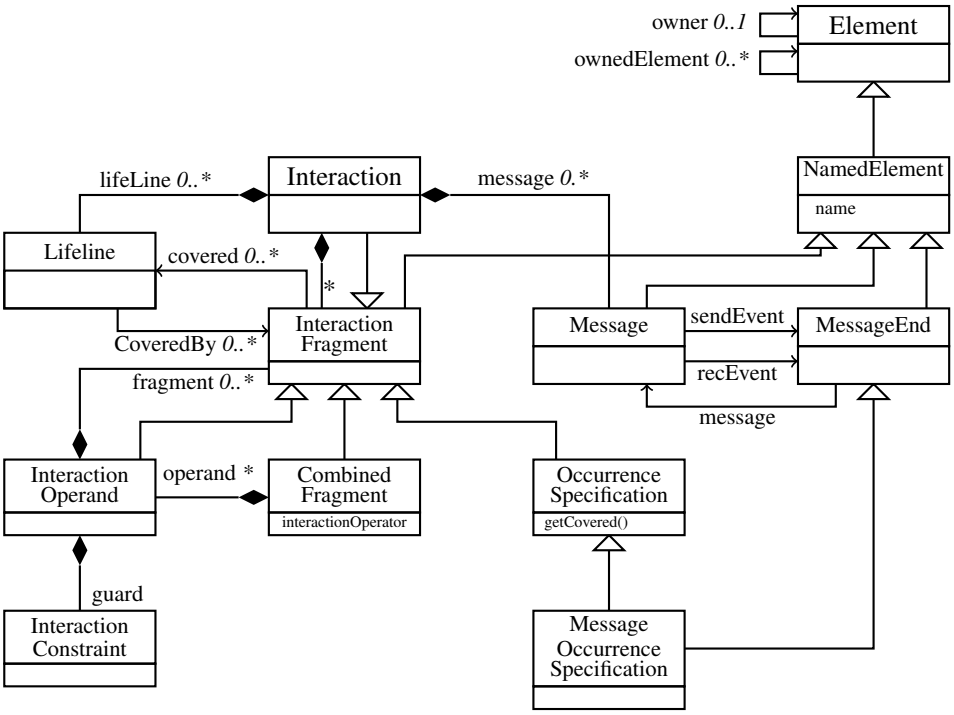


**Figure 2**: Papyrus Meta-Model of SysML Sequence Diagram

- **InteractionOperand:** each operand is associated to a combined fragment, and it can have a guard.

- **MessageOccurenceSpecification:** Each event associated to the life line is represented as a message occurrence specification. It represents an extremity of a message. We can know the life line, to which the specification is associated, by executing the method *getCovered()* of the super class *OccurenceSpecification*. We can also obtain the message started or finished at this specification, by navigating through the association *message* of the super class *MessageEnd*.

  The classes *MessageEnd*, *Message* and *InteractionFragment* inherit the class *NamedElement* which its self inherits the class *Element*. The association '*owner*' allows us to obtain the father element of the current element, however the association *'ownedElement*' allows us to obtain the children elements of the current element.

### *4.2 Interface Automata Meta-Model*

Basing on the formal definition of interface automata formalism given in section 2, we have proposed their meta-model in figure 4. The classes:

- **InterfaceAutomaton:** is the root. Each interface automaton has a name which represents the name of the block to which this automaton is associated. Each instance of this class can include a set of states, a set of transitions and a set of ports(in-ports, out-ports).

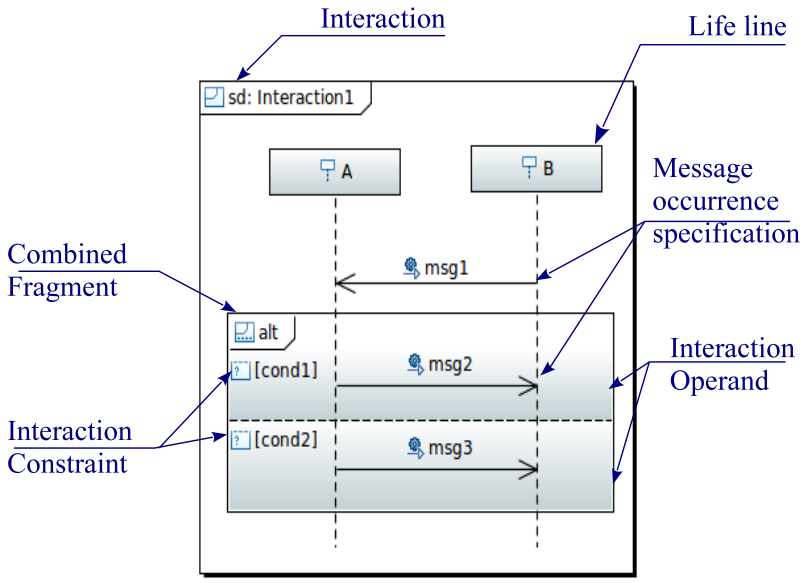


**Figure 3**: Sequence diagram elements

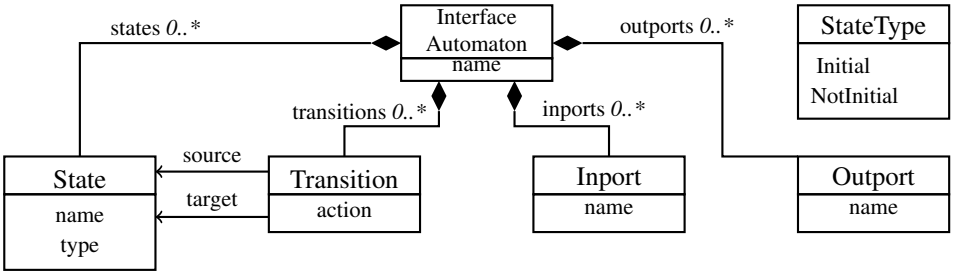


**Figure 4**: Interface Automata Meta-Model

- **State:** Each instance of this class has a name and a type. The type allows specifying if this instance is an initial state or not.
- **Transition:** each instance of this class has three values to specify. The action which is the label of this transition, the source state and the target state.
- **Inport:** represent the ports associated with the input actions.
- **Outport:** represent the ports associated with the output actions.

In Figure 5, we present an interface automaton which is modelled using our editor. We have used the Graphical Modelling Framework (GMF) to specify and generate our graphical editor of interface automata.

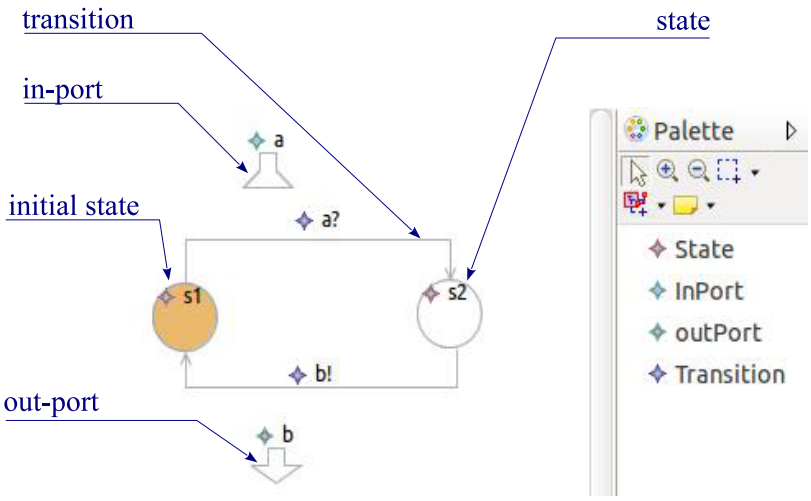


**Figure 5**: Generated Interface Automata Editor

### *4.3 Basic Interaction Transformation Rules*

An interface automaton specifies the interactions of a block *'B'* with its environment, where the environment represents all the blocks with which the block *'B'* can interact. So, the interface automaton will be associated to the life line of the block *'B'*.

To perform the basic transformations (interactions without combined fragments), we have three rules:

- **Rule 1:** *LifeLine2InterfaceAutomaton*

  This rule allows us to initialize the interface automaton which is associated to the block 'B'. The name of the interface automaton will be the name of the block *'B'*. The in-ports are created using the helper ***'createInports'***, because we need to create one in-port for all messages having the same name, which are received by the block *'B'*. To create the out-ports, we have used the helper ***'createOutports'***, which creates one out-port for all messages having the same name, which are emitted by *'B'*.

```
rule LifeLine2InterfaceAutomaton {
from lifeline : SD!Lifeline (lifeline.name<>'ENV')
to   ia : IA!InterfaceAutomaton (
name <-lifeline.name,
states <-IA!State.allInstances(),
transitions <-IA!Transition.allInstances(),
inports <- thisModule.createInports(lifeline),
outports <- thisModule.createOutports(lifeline)
)}
```

- **Rule 2:** *MessageOccurrenceSpecification2State*

  We are only interested with events (sending and receiving of messages) associated to the life line of our block *'B'*. These events (mos) are the instances of the class

*'MessageOccurrenceSpecification'*, where their life line is not the environment but the current block *'B'*( *mos.getCovered* $\neq$ *ENV*). So, we must create a state for each message occurrence specification.

```
rule MessageOccurenceSpecification2State {
from mos : SD!MessageOccurrenceSpecification
(mos.getCovered().name <> 'ENV')
to s : IA!State (name<-mos.message.name.concat('start'))
}
```

- **Rule 3:** *Message2Transition*

A message, which has an extremity that starts or ends at the life line of the current block *'B'*, must be transformed into a transition in the interface automaton of *'B'*. For a message $mos_1 \xrightarrow{mes} mos_2$ (where mos1 and mos2 are message occurrence specifications), we create a new transition. This transition will be labelled with the action 'mes', but to specify the type of this action and to fix the beginning and the end states of this transition, we must analyse three cases:

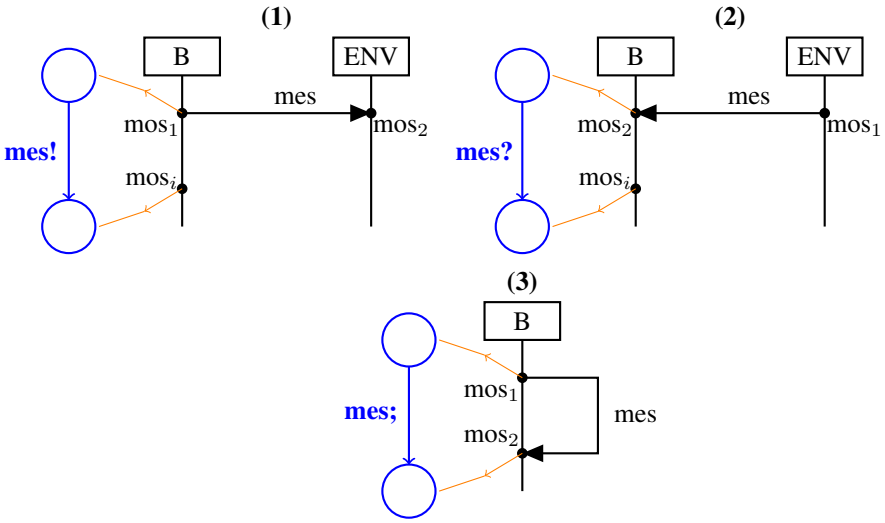


**Figure 6**: Message transformation

– Only $mos_1$ is associated to the life line of *'B'* (see figure 6 (1)): In this case, the label will be an output action *'mes!'*. The transition starts at the state associated to $mos_1$ and ends at the state associated to $mos_i$ (the next message occurrence specification of $mos_1$ on the life line of 'B').

– Only $mos_2$ is associated to the life line of *'B'* (see figure 6 (2)): In this case, the label will be an input action *'mes?'*. The transition starts at the state associated to $mos_2$ and ends at the state associated to $mos_i$ (the next message occurrence specification of $mos_2$ on the life line of 'B').

– $mos_1$ and $mos_2$ are associated to the life line of *'B'* (see figure 6 (3)): In this case, the label will be an internal action *'mes;'*. The transition starts at the state associated to $mos_1$ and ends at the state associated to $mos_2$.

```
rule message2Transition{
from mes : SD!Message, mos : SD!MessageOccurrenceSpecification
.            (mos.getCovered().name <> 'ENV' )
to t : IA!Transition (
action <- mes.name.concat(
      if( mes.sendEvent.getCovered()=mes.receiveEvent.getCovered())
      then ';'
       else    if (mes.sendEvent=mos)then '!' else '?'endif
       endif),
source <- thisModule.resolveTemp(mos, 's'),
target <- thisModule.resolveTemp(
            thisModule.NextMsgOcSpec(mos.getCovered(), mos), 's')
)}
```

– **NextMsgOcSpec (lfn, mos)** is an ATL helper that returns the next occurrence specification of *'mos'* on the life line *'lfn'*.

### *4.4 ALT Combined Fragment Transformation Rules*

The alt fragment allows us to express alternative behaviours according to guards. To transform the alt combined fragment, we have proposed three rules. Two rules for the beginning of alt, and the third one is for processing the end of alt.

For the beginning of alt, we distinguish between two cases:

- **Rule 1:** *TransformAltWichFollowsAMsg*

  In the case when the combined fragment *'alt'* follows a message 'mes', to transform alt, we create a state which represents the beginning of 'alt', and three transitions (see figure 7(1)).

  – ***'t1'*** allows us to connect the beginning of *'alt'* with the previous behaviour using the message just before *'alt'*.
  – ***'t2'*** allows us to connect the beginning of *'alt'* with the behaviour of the first operand using guard as internal action.
  – ***'t3'*** allows us to connect the beginning of *'alt'* with the behaviour of the second operand using guard as internal action.

```
rule TransformAltWichFollowsAMsg {
from alt  : SD!CombinedFragment (thisModule.FollowedAMessage(alt))
to
s : IA!State (name <- 'BeginAlt'),
t1 : IA!Transition (
action <- thisModule.previousMessage(alt).name.concat(...
         - specify the type of action as in rule 2),
source <- thisModule.resolveTemp(
              thisModule.PreviousMessageOccurence(alt),'s'),
target <- s),
```

```
t2 : IA!Transition (
action <- alt.operand->at(1).guard...concat(';'),
source <- s,
target<-thisModule.resolveTemp(thisModule.
            getTheFirstElement(alt.operand->at(1)), 's')),
t3 : IA!Transition (
action <- alt.operand->at(1).guard...concat(';'),
source <- s,
target<-thisModule.resolveTemp(thisModule.
            getTheFirstElement(alt.operand->at(2)), 's')),
}
```

- **`FollowedAMessage (alt)`** is an ATL helper that returns true if *alt* follows a message.
- **`previousMessage(alt)`** is an ATL helper that returns the message which is before *alt*.
- **`PreviousMessageOccurence(alt)`** is an ATL helper that returns the message occurrence specification which is before *alt*.
- **`getTheFirstElement(op)`** is an ATL helper that returns the first element in the operand *op*.

- **Rule 2:** *TransformFirstAltInInteractionOrOperand*

This rule processes 'alt' in case when it is the first element of the global interaction, the first element in an operand, or when it follows a combined fragment. The difference between this rule and the last one, reside in the transition *t1*. In this rule, we don't create the transition *t1* because when *'alt'* is :

- the first element in the interaction (see figure 7(2)): we don't need this transition.
- directly after a combined fragment *'cf'* (see figure 7(4)): this transition will be created by the rule which processes the end of *'cf'*.
- the first element of an operand *'op'*(see figure 7(3)): this transition will be created by the rule which processes the beginning of the combined fragment of *'op'*.

```
rule TransformFirstAltInInteractionOrOperand{
from alt  : SD!CombinedFragment
            (thisModule.FirstElementOrFollowsCF(alt))
to
s : IA!State (- the same as rule 1),
t2 : IA!Transition (- the same as rule 1),
t3 : IA!Transition (- the same as rule 1),
}
```

- **`FirstElementOrFollowsCF (alt)`** is an ATL helper that returns true if *alt* is the first element in the interaction, the first element inside an operand, or it follows directly a combined fragment.

- **Rule 3:** *TransformEndAlt*

  This rule (see figure 7(5)) allows us to process the end of an *'alt'* operand. It takes as parameters this operand and the last message occurrence specification (mos) inside it. It creates a transition between the state associated to mos and the state associated to the next element of the combined fragment to which this operand belongs (*op.owner*). The next element may be a message or a combined fragment. The transition takes as label the name of message whose *'mos'* is one of its ends.

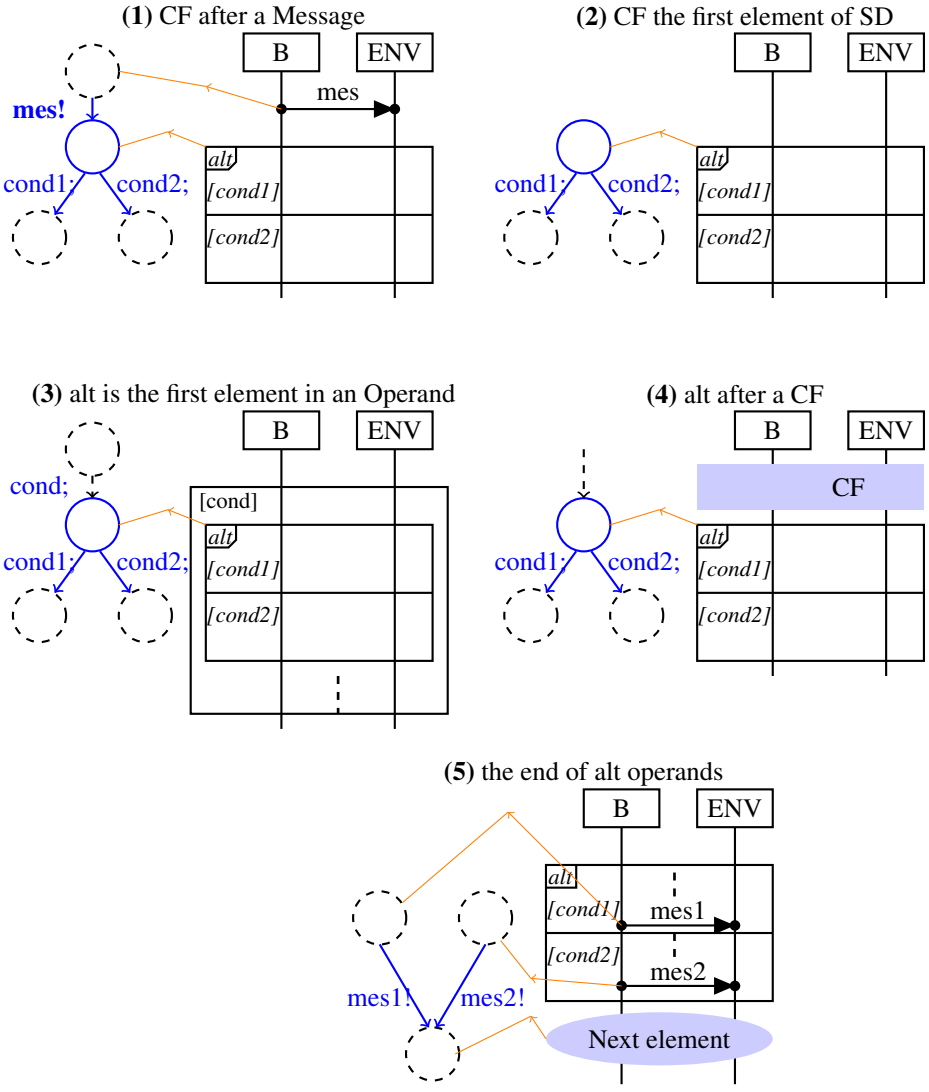


**Figure 7**: Alt transformations

```
rule TransformEndAlt {
from
op  : SD!InteractionOperand,
mos  : SD!MessageOccurrenceSpecification
   (thisModule.isTheLastMessageInOperand(mos.message, op)
                and mos.getCovered().name<>'ENV')
to
t : IA!Transition (
action <- mos.message.name.concat(
          - specify the type of action as in rule 2),
source <- thisModule.resolveTemp(mos,'s'),
target <-thisModule.resolveTemp( thisModule.
                          getNextElement(op.owner),'s'),
}
```

- **isTheLastMessageInOperand(mes, op)** returns true if the message *mes* is the last message in the operand op.
- **getNextElement(cf)** returns the next message or combined fragment of the combined fragment cf.

Using the same manner of thinking, we can define rules for other combined fragments.

## 5 Generation of Ptolemy specification

At this step, and to discharge the user from redrawing the interface automata using the Ptolemy user interface, we propose a set of Acceleo templates to generate automatically the Ptolemy entry specification.

By analysing an entry file of ptolemy interface automaton, and by eliminating information related to the position of nodes on the ptolemy canvas, we have obtained its skeleton and we have defined six Acceleo templates. We have eliminated the information related to the position of nodes on the canvas, because the ptolemy, when it doesn't find information about the position of a node, it uses its default values.

The first Acceleo template '*generateIA*' is the main template, it creates the file of the Ptolemy specification and its header, and calls the other templates. The templates, after the principal one, each one has a name that corresponds to its role.

- `generateInport(inport : Inport)`: it allows us to generate the Ptolemy specification of each in-port of the concerned interface automaton.
- `generateOutport(outport : Outport)`: it will be called iteratively (as the previous template) by the main template to generate the Ptolemy specification for each out-port of the concerned interface automaton.
- `generateState(state : State)`: it allows us to generate the Ptolemy specification for automaton states.
- `generateRelation(transition : Transition, i:Integer)` and `generateLinks ( transition : Transition, i:Integer)`: these

two templates allow us to generate the Ptolemy specification for transitions of the automaton.

```
[comment encoding = UTF-8 /]
[module generate('http://www.interfaceAutomata.ecore')]
[template public generateIA(IA : InterfaceAutomaton)]
[comment @main/]
[file (IA.name.concat('.xml'), false, 'UTF-8')]
<?xml version="1.0" standalone="no"?>
   <!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
         "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
   <entity name="[IA.name/]"
      class="ptolemy.domains.modal.kernel.ia.InterfaceAutomaton">
         [for (outport :Outport| IA.outports )]
         [generateOutport(outport)/]
         [/for]

         [for ( inport :Inport| IA.inports )]
         [generateInport(inport) /]
         [/for]

         [for (state :State| IA.states )]
         [generateState(state) /]
         [/for]

         [for (transition :Transition| IA.transitions)]
         [generateRelation(transition, i)/]
         [/for]

         [for (transition :Transition | IA.transitions)]
         [generateLinks(transition,i)/]
         [/for]
</entity>
[/file]
[/template]
```

```
[template private generateInport(inport : Inport)]
<port name="[inport.name/]" class="ptolemy.actor.TypedIOPort">
         <property name="input"/>
</port>
[/template]
```

```
[template private generateOutport(outport : Outport)]
<port name="[outport.name/]" class="ptolemy.actor.TypedIOPort">
         <property name="output"/>
</port>
[/template]
```

```
[template private generateState(state : State)]
<entity name="[state.name/]" class="ptolemy.domains.modal.kernel.State">
[if (state.type=StateType::Initial)]
      <property name="isInitialState" class="ptolemy.data.expr.Parameter"
      value="true"></property>
[/if]
```

```
</entity>
[/template]
```

```
[template private generateRelation(transition : Transition, i:Integer)]
<relation name="relation[if (i>1)][i/][/if]"
    class="ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
      <property name="label" class="ptolemy.kernel.util.StringAttribute"
         value="[transition.action/]"></property>
</relation>
[/template]
```

```
[template private generateLinks ( transition : Transition, i:Integer)]
<link port="[transition.source.name/].outgoingPort"
      relation="relation[if (i>1)][i/][/if]"/>
<link port="[transition.target.name/].incomingPort"
       relation="relation[if (i>1)][i/][/if]"/>
[/template]
```

## 6 The Blocks Verification

We want to verify the *composability* and the *compatibility* of the blocks. To do that, we base on the interface automata which describe the interaction protocols of these blocks.

- Two blocks $B_1$ and $B_2$ are considered as composable if their interface automata $A_1$ and $A_2$ are composable: $\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma^H_{A_2} = \emptyset$.
- Two blocks are compatible, if they are consistent and their interface automata are compatible. According to the optimistic approach of *Henzinger*, two interface automata are compatible if their composition is not empty: $A_1 \| A_2 \neq \emptyset$

To verify the composability and the compatibility of the blocks, we use the *Ptolemy* tool. we give it, as entry, the generated files (the files that we have generated using our Acceleo templates). Ptolemy computes the composition of interface automata and delivers the result. If the result of composition is not empty, this means that the blocks are compatible.

## 7 Case Study: CyCab

CyCab (Baille et al. (1999)) is a new means of electrical transportation, it is conceived basically for free-standing port services. It is controlled by a computer system. The CyCab system has two major parts: the station and the vehicle. The vehicle is guided by the information received from the station, which allows to situate the vehicle.

In this case study, we are only interested by the 'station' part. The station has a sensor that receives signals from vehicle giving the vehicle position (pos?). The station has also a computing units that sends a signal (far! or halt!) to the vehicle to indicate if it is far from the station or not. In figure 8, we present the architecture of the Cycab System using the SysML BDD.

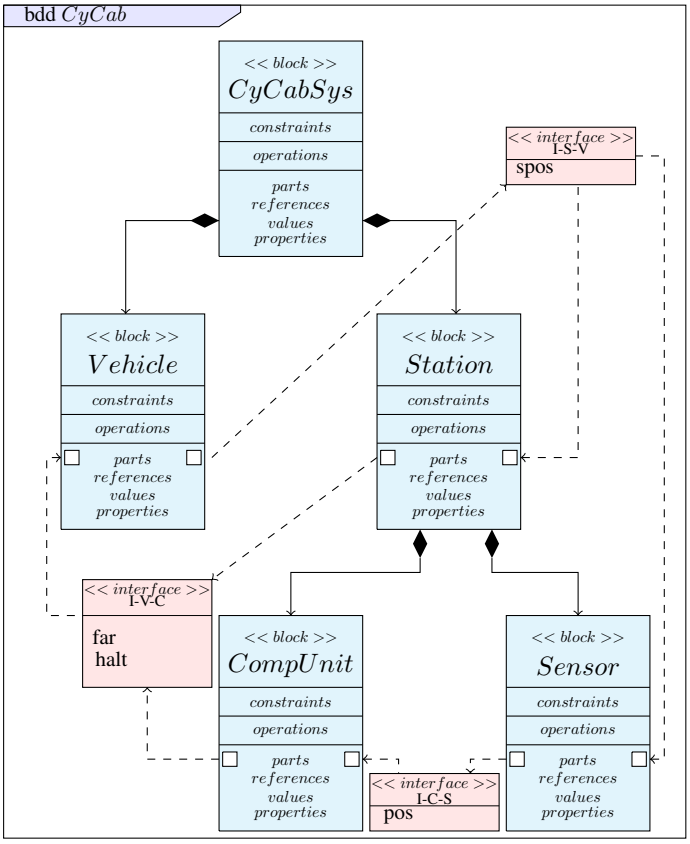


**Figure 8**: Block Definition Diagram of CyCab

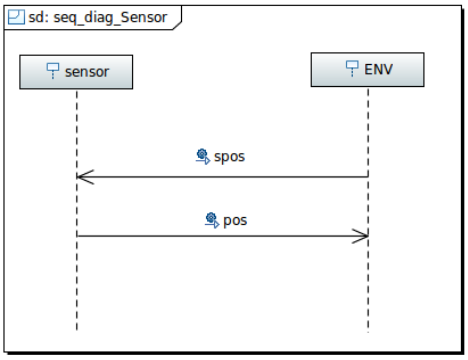


**Figure 9**: SD of Sensor

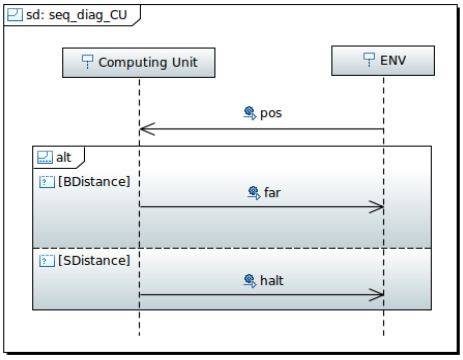


**Figure 10**: SD of Computing-Unit

The interactions of the sensor and the computing-unit blocks are represented as sequence diagrams (see figure 9 and figure 10). To draw sequence diagrams, we have used the papyrus editor.

By applying our ATL rules on the sequences diagrams of the sensor and the computing unit, we have obtained their equivalents of interface automata. In figure 11 and figure 12, we present the resulted interface automata in our graphical editor.

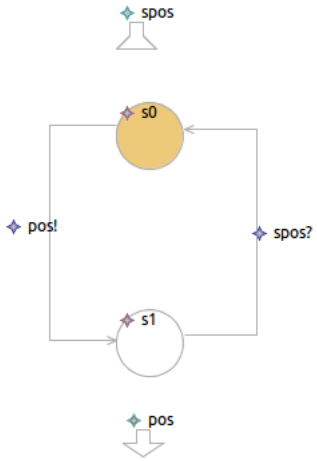


**Figure 11**: SD of Sensor

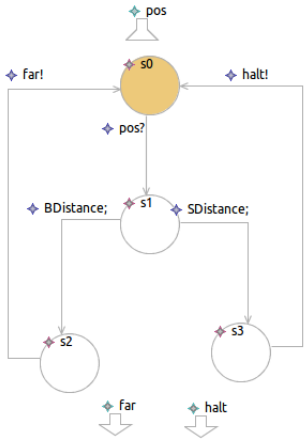


**Figure 12**: SD of Computing-Unit

By applying the acceleo templates, that we have defined to generate Ptolemy specification, we have obtained these files.

- Ptolemy file of Sensor block:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
"http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="sensor" class=
"ptolemy.domains.modal.kernel.ia.InterfaceAutomaton">
  <port name="pos" class="ptolemy.actor.TypedIOPort">
    <property name="output"/>
  </port>
  <port name="spos" class="ptolemy.actor.TypedIOPort">
     <property name="input"/>
  </port>
  <entity name="s1" class="ptolemy.domains.modal.kernel.State">
  </entity>
  <entity name="s0" class="ptolemy.domains.modal.kernel.State">
    <property name="isInitialState" class="ptolemy.data.expr.Parameter"
    value="true"></property>
  </entity>
  <relation name="relation" class=
      "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
    <property name="label" class="ptolemy.kernel.util.StringAttribute"
    value="spos?"></property>
  </relation>
  <relation name="relation2" class=
      "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
  <property name="label" class="ptolemy.kernel.util.StringAttribute"
    value="pos!"></property>
```

```
  </relation>
  <link port="s1.outgoingPort" relation="relation"/>
  <link port="s0.incomingPort" relation="relation"/>
  <link port="s0.outgoingPort" relation="relation2"/>
  <link port="s1.incomingPort" relation="relation2"/>
</entity>
```

- Ptolemy file of Computing-Unit block:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
"http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="Computing Unit" class=
                    "ptolemy.domains.modal.kernel.ia.InterfaceAutomaton">
  <port name="far" class="ptolemy.actor.TypedIOPort">
   <property name="output"/> </port>
  <port name="halt" class="ptolemy.actor.TypedIOPort">
   <property name="output"/> </port>
  <port name="pos" class="ptolemy.actor.TypedIOPort">
   <property name="input"/> </port>
  <entity name="s0" class="ptolemy.domains.modal.kernel.State">
   <property name="isInitialState" class="ptolemy.data.expr.Parameter"
   value="true"></property> </entity>
  <entity name="s2" class="ptolemy.domains.modal.kernel.State">
</entity>
  <entity name="s3" class="ptolemy.domains.modal.kernel.State">
</entity>
  <entity name="s1" class="ptolemy.domains.modal.kernel.State">
</entity>
  <relation name="relation" class=
     "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
    <property name="label" class="ptolemy.kernel.util.StringAttribute"
    value="pos?"> </property>
  </relation>
  <relation name="relation2" class=
     "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
    <property name="label" class="ptolemy.kernel.util.StringAttribute"

    value="BDistance;"> </property>
  </relation>
  <relation name="relation3" class=
     "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
    <property name="label" class="ptolemy.kernel.util.StringAttribute"
     value="SDistance;"> </property>
  </relation>
  <relation name="relation4" class=
     "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
    <property name="label" class="ptolemy.kernel.util.StringAttribute"
    value="far!"> </property>
  </relation>
  <relation name="relation5" class=
     "ptolemy.domains.modal.kernel.ia.InterfaceAutomatonTransition">
    <property name="label" class="ptolemy.kernel.util.StringAttribute"
    value="halt!"> </property>
  </relation>
  <link port="s0.outgoingPort" relation="relation"/>
  <link port="s1.incomingPort" relation="relation"/>
  <link port="s1.outgoingPort" relation="relation2"/>
```

```
    <link port="s2.incomingPort" relation="relation2"/>
    <link port="s1.outgoingPort" relation="relation3"/>
    <link port="s3.incomingPort" relation="relation3"/>
    <link port="s2.outgoingPort" relation="relation4"/>
    <link port="s0.incomingPort" relation="relation4"/>
    <link port="s3.outgoingPort" relation="relation5"/>
    <link port="s0.incomingPort" relation="relation5"/>
  </entity>
```

Using Ptolemy tool, we can use these two files to verify the composability and the compatibility of the sensor and the computing unit blocks.

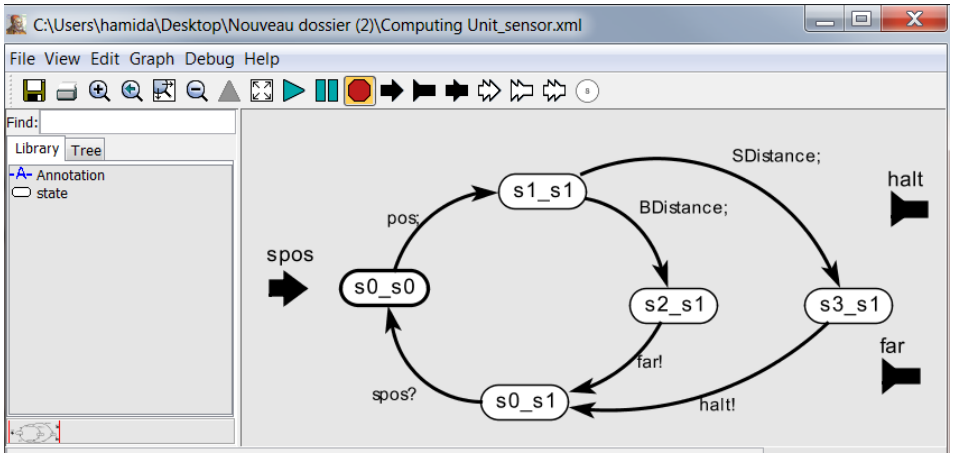


**Figure 13**: Parallel composition of Control Unit and Sensor

In figure 13, we present the result of composing the two interface automata using Ptolemy tool. Because the composition is not empty, we deduce that the control unit and the sensor blocks are composable and compatible. If we assemble them in the same system, we obtain a system part that interacts with the rest of the system according to the scenarios modelled as an interface automaton in figure 13.

## 8 Related work

In literature, various research works have been done to transform informal models to formal ones (Rehab and Chaoui (2015)) (Aouag et al. (2014)) (Guerrouf et al. (2013)) (Debbabi et al. (2010)). These transformations are generally implemented and offered as tools which can assist architects during the verification of their systems. Many of these works are dedicated to generate formal models from UML and SysML diagrams (Rehab and Chaoui (2015)) (Aouag et al. (2014)) (Rahim et al. (2015)), which are considered as informal models. In fact, the sequence diagram, which is a shared model between UML and SysML, was be the matter of many transformation works. The most of the proposed approaches are based on using transformation rules, and they differ essentially in the target model of transformation.

In (Kessentini et al. (2010)), (Ribeiro and Fern (2006)), we find a description of an automated transformation method, which allows transforming the sequences diagrams to

their equivalents of coloured Petri Nets. In (Emadi and Shams (2009)), the authors have proposed some correspondences to transform sequence and use case diagrams to Petri Net. These correspondences formalize interactions composed of messages and combined fragments (alternative, optional and loop). Authors in (Merah et al. (2014)), basing on Meta-modelling and ATL grammars, they have defined a set of ATL rules to transform SDs to Petri Nets. They have proposed rules for the basic constructs of SDs and for a sub set of combined fragments kinds (Alt, Par). In (Chaoui et al. (2009)), a grammar, which based on graph transformation, was proposed to transform the sequence diagram to ECATNets, a variant of Petri Net. The authors have based on the $AToM^3$ tool to implement meta models of SDs and ECATNets, to generate the modelling tools and to implement the graph grammar which performs the transformation. They are also some works which have as target models a textual specification. In (Ait Oubelli et al. (2011)), a graph grammar was used to generate Promela code starting from SDs. The authors used also the tool $AToM^3$ for meta-modelling and for implementing the graph transformation grammar. In (Merah et al. (2013)), The authors, they propose a grammar to transform the communication diagram, which has a near semantic to that of SD, to Buchi automata.

In (Chouali and Hammad (2011)), some correspondences between sequence diagram and interface automata are given. This work was be the reference in (Carrillo et al. (2012)) to prepare the sequence diagram of SysML blocks for the compatibility verification phase. But, in (Carrillo et al. (2012)), this transformation have done manually, which can be considered as a source of user errors. That's why, we propose, in this paper, the correspondences for more constructs, and we propose, also, a set of ATL rules to automate this transformation. Contrary to the works mentioned before, which they don't take into consideration the case of nested combined fragments, in our work, we explain the different cases, and how we deal with them.

The second phase concerns Ptolemy tool, which is used to verify the interface automata compatibility, and to compute their parallel composition. To discharge the user from redrawing the resulted interface automata using the Ptolemy user interface, we have proposed a set of Acceleo templates to generate the entry code of Ptolemy. The other strong point of our work is the use of the meta-model of UML and SysML sequence diagram proposed by Papyrus, and so we base on its graphical editor to draw sequence diagrams.

## 9 Conclusion

The point we want to address in our paper is how can we prepare the SysML blocks interactions for verification. Thus, our proposed approach is based on specifying the correspondences between the blocks sequence diagrams and interface automata. The goal of this paper is to present how it's possible to automate the transformation from sequence diagrams to interface automata using ATL. We have shown the transformation of the basic constructs of sequence diagrams. We have also given the ATL rules to transform the alternative combined fragment. The second objective of our work concerns Ptolemy tool, which is used to verify the interface automata compatibility, and to compute their parallel composition. To discharge the user from redrawing the resulted interface automata using the Ptolemy user interface, which can considered as a source of errors, we have proposed a set of Acceleo templates to generate the entry code of Ptolemy. We have also given an overview of how can we use the generated files to verify the compatibility of blocks. To illustrate our approach, we have apply it through a CyCab case study.

As future works, we plan to utilize the use case diagrams for interlinking the different sequence diagrams of blocks during the transformation into interface automata. We plan also to back-annotate the results of interface automata compatibility verification on the source sequence diagrams.

## References

Acceleo. `http://www.eclipse.org/acceleo/` [Accessed 12/02/2016].

Ait Oubelli, M., Younsi, N., Amirat, A. and Menasria, A. (2011). 'From UML 2.0 Sequence Diagrams to PROMELA code by Graph Transformation using AToM3'. In Abdelmalek Amine, Otmane Ait Mohamed, Boualem Benatallah and Zakaria Elberrichi, ed., 'CIIA', CEUR-WS.org.

Aouag, M., Elmansouri, R. and Allaoua Chaoui (2014). 'From UML 2.0 diagrams to aspect oriented diagrams using graph transformation'. International Journal of Computer Aided Engineering and Technology, vol 6, no 2, pp. 200–233.

ATL: Atlas Transformation Language. `https://eclipse.org/atl/` [Accessed 12/02/2016].

Baille,G., Garnier, P., Mathieu,H. and Pissard-Gibollet, R. (1999). 'The INRIA Rhone-Alpes CyCab' , INRIA, Tech. Rep.

Bouaziz, H., Chouali, S., Hammad, A., and Mountassir, H. (2015). 'SysML Blocks Adaptation', in 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, pp. 417–433.

Carrillo, O. , Chouali, S., and Mountassir, H (2012). 'Formalizing and verifying compatibility and consistency of sysml blocks', ACM SIGSOFT Software Engineering Notes, vol. 37, no. 4, pp. 1–8.

Chaoui, A. , ElMansouri, R. , Saadi, W. , and Kerkouche, E. (2009). 'From uml sequence diagrams to ECATNets: a graph transformation based approach for modelling and analysis', in proceedings of The 4th International Conference on Information Technology ICIT. Orissa, India.

Chouali, S., Hammad, A. (2011) 'Formal verification of components assembly based on sysml and interface automata', ISSE, vol. 7, no. 4, pp. 265–274.

Czarnecki, K. and Helsen, s. (2003). 'Classification of model transformation approaches', OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, Anaheim, USA.

de Alfaro, L. and Henzinger, T. A. (2001). Interface automata. In ESEC/ SIGSOFT FSE, Vienna, Austria, pp. 109–120.

Debbabi, M., Hassa, F., Jarraya, Y., Soeanu, A. and Alawneh,L. (2010). Verification and Validation in Systems Engineering - Assessing UML/ SysML Design Models. Springer.

de Lara, J.,Vangheluwe, H., and Alfonseca, M. (2004). 'Meta-modelling and graph grammars for multi-paradigm modelling in atom3', Software and System Modeling, Vol. 3, No. 3, pp. 194–209.

Emadi,S. and Shams, F. (2009). 'Mapping annotated use case and sequence diagrams to a petri net notation for performance evaluation', in Proceedings of the 2009 Second International Conference on Computer and Electrical Engineering - Volume 02, ser. ICCEE'09. Washington, DC, USA: IEEE Computer Society, pp. 68–71.

Guerrouf,F., Chaoui, A. and Aldahoud A.(2013). 'A graph transformation approach of mobile activity diagram to nested Petri nets'. International Journal of Computer Aided Engineering and Technology, vol 5, no 1, pp. 44–57.

Kessentini, M., Bouchoucha, A., Sahraoui,H. and Boukadoum ,M. (2010). 'Examplebased sequence diagrams to colored petri nets transformation using heuristic search', in Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, pp. 156–172.

Merah, E. , Messaoudi, N. ,Saidi, H. , and Chaoui, A. (2013). 'Design of atl rules for transforming uml 2 communication diagrams into buchi automata', International Journal of Software Engineering and Its Applications, vol. 7, no. 2, pp. 19–34.

Merah, E. , Messaoudi, N. , Bardou, D. , and Chaoui, A. (2014). 'Design of atl rules for transforming uml 2 sequence diagrams into petri nets', International Journal of Computer Science and Business Informatics, vol. 8, no. 1, pp. 1–21.

Obeo, `http://www.obeo.fr` [Accessed 12/02/2016].

OMG, OMG Systems Modeling Language (OMG SysML$^{TM}$) Version 1.3, 2012, `http://www.omg.org` [Accessed 12/02/2016].

Papyrus. `https://eclipse.org/papyrus/` [Accessed 12/02/2016].

Ptolemy Project. `http://ptolemy.eecs.berkeley.edu/` [Accessed 12/02/2016].

Rahim, M., Kheldoun, A., Boukala-Ioualalen, M. and Hammad, A. (2015). 'Recursive ECATNets-based approach for formally verifying System Modelling Language activity diagrams', IET Software, vol 9, no 5, pp. 119–128.

Rehab, R. and Chaoui, A. (2015). 'TGG-based process for automating the transformation of UML models towards B specifications'. International Journal of Computer Aided Engineering and Technology, vol 7, no 3, pp. 378–400.

Ribeiro, O. R. and Fern, J. M. (2006). 'Some rules to transform sequence diagrams into coloured petri nets', in In 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, pp. 237–256.

Rumbaugh, J. , Jacobson, I. , and Booch, G. (2004). 'The Unified Modeling Language reference manual', Second Edition. Addison-Wesley, Essex, UK, UK.