Exploitation de la Hiérarchie pour la Vérification de la Compatibilité des Blocs SysML

Hamida Bouaziz*,** Samir Chouali*
Ahmed Hammad*, Hassan Mountassir*

* Institut Femto-St, Université de Bourgogne Franche-Comté, Besançon, France
** Laboratoire Mécatronique, Université de Jijel, Jijel, Algérie
{hamida.bouaziz, schouali, ahammad, hmountas}@femto-st.fr

Résumé. Le développement de systèmes à base de composants consiste en l'assemblage d'un ensemble d'unités de base qui répondent chacune à une partie des exigences du système. Cette démarche permet la réduction du coût de développement. Cependant, l'opération d'assemblage impose l'adoption d'une approche de vérification qui doit être complète et moins coûteuse. Dans ce papier, nous proposons une approche formelle de vérification de la compatibilité de blocs SysML, en vue d'étudier la possibilité de leur assemblage. Principalement, une spécification SysML d'un système consiste à représenter sa structure sous forme d'un ensemble de blocs en interaction, cette interaction peut être modélisée par des modèles qui exposent un certain niveau de hiérarchie. Notre approche propose de profiter de la hiérarchie présente dans les modèles SysML ainsi que dans les automates afin d'alléger la vérification de la compatibilité des blocs SysML.

1 Introduction

L'ingénierie des systèmes est une approche qui met à la disposition des concepteurs un ensemble d'outils et de processus pour développer des systèmes complexes. Actuellement, dans ce domaine, il y a une grande tendance à utiliser SysML (2012) pour représenter les exigences, la structure et le comportement des systèmes industriels (ex. Pihlanko et al. (2013)Linhares et al. (2006)). L'émergence du SysML dans le domaine des systèmes complexes est due principalement à sa capacité de la hiérarchisation et de l'abstraction. Ces deux critères apparaissent clairement à travers ses diagrammes. L'organisation hiérarchique au niveau de BDD (Block Definition Diagram) de SysML permet de concevoir l'architecture du système de façon incrémentale, ce qui facilite le développement des systèmes. Pour modéliser le comportement, SysML utilise les machines d'état, les diagrammes d'activité et de séquence. Le SD (Sequence Diagram) est utilisé pour représenter les scénarios d'interaction entre blocs, où chaque SD détaille un cas d'utilisation. Cependant, un bloc peut participer à plusieurs cas d'utilisation, ce qui rend son protocole d'interaction partagé entre plusieurs SDs. Donc, la vision globale sur le protocole d'interaction lié à un bloc reste indéfinie.

Dans Bouaziz et al. (2015), nous avons proposé le modèle HPSM (Hierarchical Protocol State machine) pour représenter le protocole d'interaction d'un bloc. Il se base sur la structure des machines à état hiérarchiques, et il exprime la relation entre les services requis et

les services fournis d'un bloc. Le HPSM est un modèle convivial et il permet de donner la vision globale sur les scénarios d'interaction liés à un bloc. Cependant, HPSM n'est pas assez formelle pour permettre la vérification formelle de certaines propriétés. C'est pourquoi, nous avons proposé de transformer HPSM en HIA-ILT (Hierarchical Interface Automata with Inter-Level Transitions), une variante de IA (Interface autoamta) que nous avons proposé dans Bouaziz et al. (2015). Dans ce papier, nous montrons également comment adapter notre approche pour l'appliquer sur des blocs conçus séparément, où il existe des incompatibilités entre leurs interfaces.

Dans Eles et al. (2002); Karlsson et al. (2007), les auteurs proposent une approche de vérification formelle qui se base sur l'utilisation de la logique CTL (Computational Tree Logic) pour représenter les propriétés à vérifier sur le résultat de l'assemblage de deux composants. Dans Inverardi et Tivoli (2001) et Inverardi et Tivoli (2003), les auteurs utilisent la logique LTL (Linear Temporal Logic) Manna et Pnueli (1991) pour représenter les propriétés telles que la sûreté. Dans Carrillo et al. (2012), Carrillo et al. vérifient la consistance et la compatibilité des blocs SysML en utilisant les automates d'interfaces simples. Cependant, dans notre approche, nous avons contribué à alléger la vérification de la compatibilité des blocs SysML en utilisant un modèle hiérarchique HIA-ILT, et en introduisant une étape qui permet de sélectionner les états à aplatir et ceux à considérer comme des états abstraits durant le processus de la vérification. En plus, nous prenons en considération le fait que les blocs peuvent être conçus séparément. Dans ce cas là, nous faisons appel à la notion du contrat d'adaptation.

Le reste de ce papier est organisé comme suit : dans la section 2 et la section 3, nous présentons des préliminaires sur les BDD SysML, IBD SysML et les automates d'interfaces. Dans la section 4 et la section 5, nous présentons nos modèles. Dans la section 6, nous présentons notre approche de vérification de compatibilité. Finalement, dans la section 7, nous concluons et nous présentons les perspectives de notre travail.

2 BDD et IBD de SysML

Dans SysML, le diagramme de définition de bloc (BDD) définit les caractéristiques des blocs du système et les relations entre eux SysML (2012). Chaque bloc peut avoir des propriétés, des opérations, des contraints et des ports. Une relation peut être une association, une navigation, une agrégation, une composition, une dépendance ou une généralisation.

Le diagramme de bloc interne (IBD) de SysML capture la structure interne des blocs composites SysML (2012). Il permet de représenter la relation entre les ports des blocs en utilisant des connecteurs. Ces connecteurs spécifient la correspondances entre les flots et les services des différents blocs.

Exemple 1 :

Nous donnons l'exemple d'un robot de nettoyage *"Roomba"* (figure 1). Dans notre système, nous considérons que Roomba est contrôlé par un opérateur humain. Kinect capture la position de l'opérateur et demande au coordinateur de traiter l'image capturée. Le coordinateur détecte l'action demandée et la transmet vers le Robot. Le robot est composé d'un récepteur et d'un Roomba. Le récepteur se charge de coder l'action demandée sous forme d'une commande *sci*. Roomba peut fonctionner selon deux méthodes : autonome (SAFE) et non-autonome (FULL). Quand l'opérateur choisit la méthode autonome (SAFE), il aura la possibilité de basculer entre ces modes : Le mode Clean : Roomba commence un mouvement en spiral, ensuite il suit le mur

FIG. 1: étude de cas.

jusqu'à ce que toute la pièce soit nettoyée. Le mode Spot : pour nettoyer une petite surface. Et le mode Max : Roomba nettoie jusqu'à ce que la batterie soit vide.

Dans la méthode manuel (FULL), l'opérateur contrôle Roomba en spécifiant la direction. Roomba accepte ces commandes : ADVANCE : pour avancer. RIGHT : rotation à droit. Left : rotation à gauche.

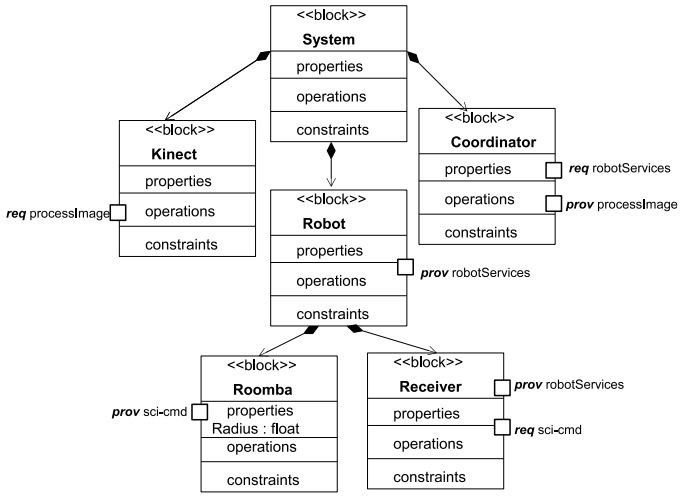


FIG. 2: BDD du Système.

la vérification de la compatibilité des blocs sysml

Dans la figure 2, nous présentons le BDD de notre système. Il y a deux blocs composites *"System"* et *"Robot"*.

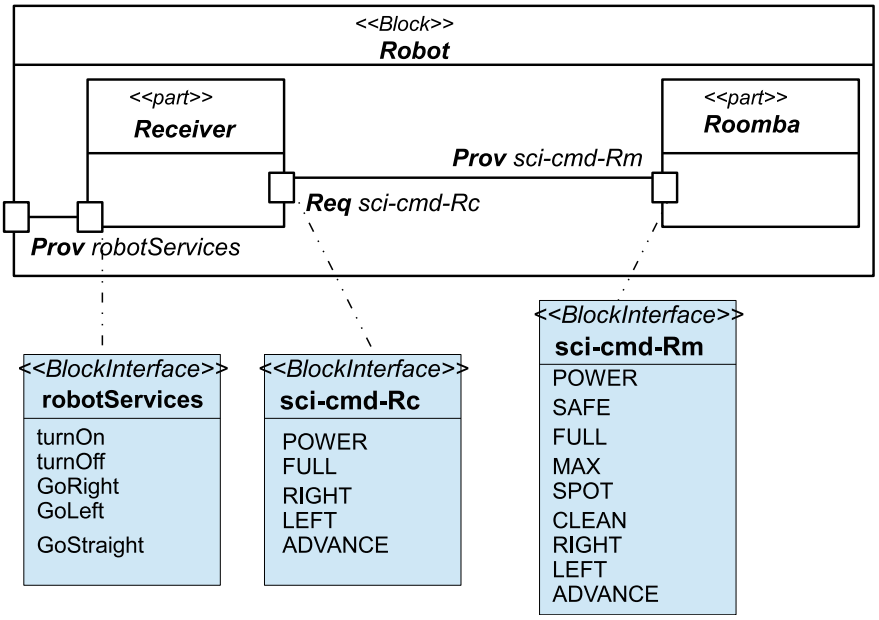


FIG. 3: IBD du Robot.

Dans le reste du papier, nous allons focaliser sur la partie *Robot* du système. Dans la figure 3, nous montrons que le robot est composé de deux parties "Receiver" et "Roomba". Le récepteur fournit ses services à travers le port *"robotServices"* de type *"<<BlockInterface>> robotServices"*, et il contrôle Roomba à travers le port requis *"sci-cmd-Rc"*. Roomba utilise le port *"sci-cmd-Rm"* pour communiquer avec le récepteur.

3 Automate d'interface

de Alfaro et Henzinger (2001) ont introduit les automates d'interface pour spécifier les interfaces de composants et vérifier leur assemblage. Un automate d'interface spécifie les différents scénarios d'interaction du composant. Chaque scénario est représenté par une séquence d'actions que le composant offert ou requiert de son environnement. L'ensemble des actions est décomposé en trois groupes : les actions d'entrée, les actions de sortie et les actions internes. Les actions d'entrée représentent les méthodes implémentées par le composant et qui sont déclenchables par son environnement. L'ensemble des actions de sortie représente les méthodes que le composant demande des autres composants. Alors que les actions internes sont activées localement par le composant.

Définition 1 (Automate d'interface)
Un automate d'interface est représenté par un tuple :

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

Où :
- S_A est un ensemble d'états. $I_A \subseteq S_A$ est un ensemble des états initiaux.
- Σ_A^I est un ensemble des actions d'entrée, elles sont étiquetées par ' ?'. Σ_A^O est un ensemble des actions de sortie, elles sont étiquetées par ' !'. Σ_A^H est un ensemble des actions internes, elles sont étiquetées par ' ;'.
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ est l'ensemble de transitions, où $\Sigma_A = \Sigma_A^I \cup \Sigma_A^O \cup \Sigma_A^H$.

Définition 2 (Automates d'interface composables)
Deux automates d'interface A_1 et A_2 sont composables ***ssi*** $\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma_{A_2}^H = \emptyset$.

Définition 3 (Produit synchrone)
Le produit synchrone capture l'exécution parallèle de deux composants représentés par des automates d'interface. Le produit synchrone entre deux automates d'interface composables A_1 et A_2 est défini par :

$$A_1 \otimes A_2 = \langle \, \mathrm{S}_{A_1 \otimes A_2}, \mathrm{I}_{A_1 \otimes A_2}, \Sigma_{A_1 \otimes A_2}^I, \Sigma_{A_1 \otimes A_2}^O, \Sigma_{A_1 \otimes A_2}^H, \delta_{A_1 \otimes A_2} \, \rangle$$

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

Noter que : ***Shared***$(A_1,A_2) = (\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_{A_1}^O \cap \Sigma_{A_2}^I)$ est l'ensemble des actions partagées entre A_1 et A_2.

Définition 4 (Composition parallèle)
La composition parallèle de A_1 et A_2 ($\mathrm{A}_1 \parallel \mathrm{A}_2$) peut être déduite de produit synchrone $\mathrm{A}_1 \otimes \mathrm{A}_2$, en éliminant les états illégaux et les états atteignables à partir des états illégaux en activant des actions de sortie ou des actions internes. L'ensemble des *états illégaux* de A_1 et A_2 est défini par :

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

$\Sigma_A^I(s_1)$, Σ_A^O (s_1) sont respectivement l'ensemble des actions d'entrée et l'ensemble des actions de sortie possible au niveau de l'état s_1.

Définition 5 (Compatibilité)
A_1 et A_2 sont compatible ***ssi*** l'automate $A_1 \| A_2$ n'est pas vide.

la vérification de la compatibilité des blocs sysml

4 HPSM

SysML utilise le diagramme de séquence (SD) pour représenter les interactions entre les blocs du système. Chaque SD détaille un cas d'utilisation. Quand un bloc participe aux plusieurs cas d'utilisations, son protocole d'interaction sera divisé entre plusieurs SDs, ce qui ne permet pas d'avoir une vision globale sur l'interaction d'un bloc.

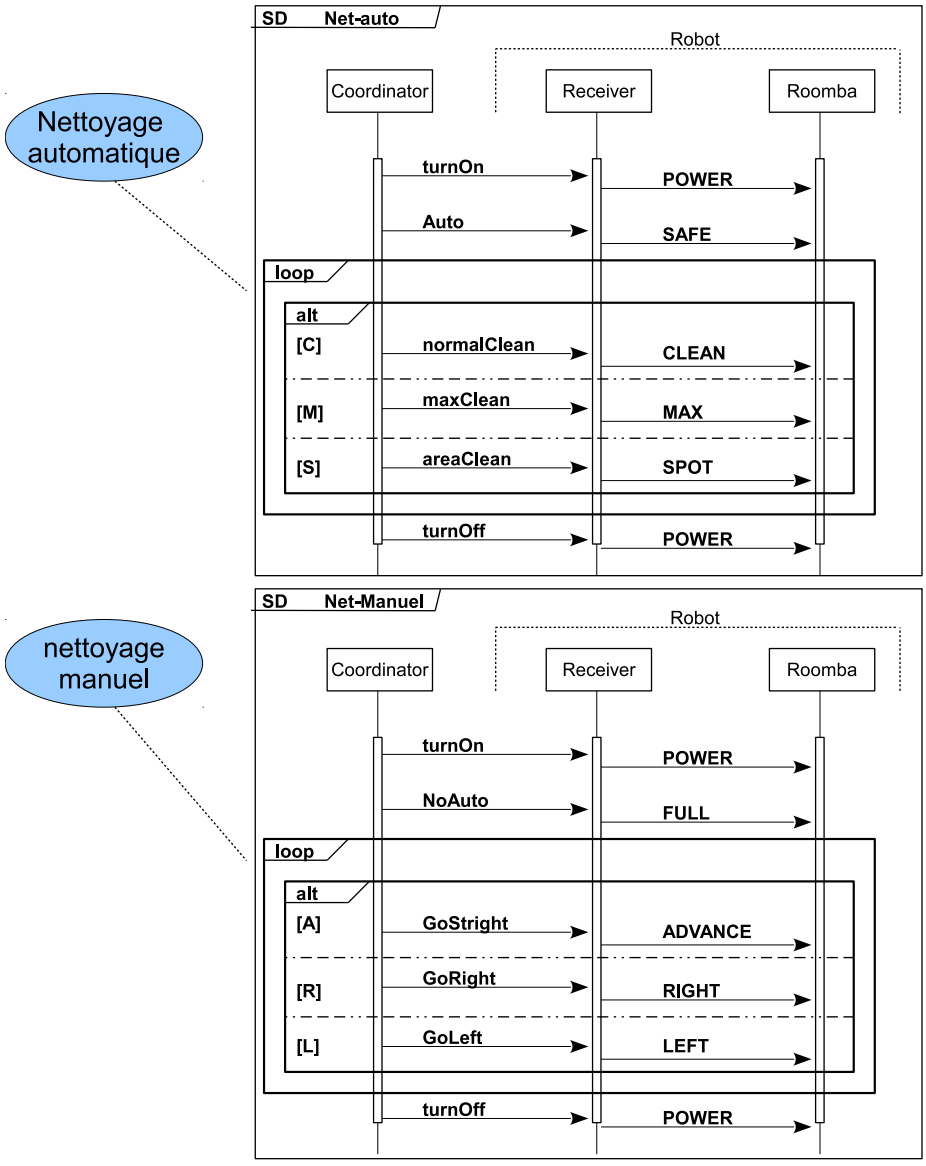


FIG. 4: SDs.

Dans Bouaziz et al. (2015), nous avons proposé le modèle HPSM (Hierarchical protocol state machine) qui permet de représenter le protocole d'interaction d'un bloc. HPSM adopte la même structure des machines d'état hiérarchiques. Cette structure permet de cacher le détail quand il n y a pas un besoin de le visualiser.

Nous définissons le HPSM d'un bloc ***B*** comme suit :

$$HPSM_B = \langle\, SS_B,\, CS_B,\, I_B,\, T_B,\, Prov\_Serv_B,\, Req\_Serv_B \,\rangle$$

Où :

- $\mathbf{SS}_B$ est l'ensemble des états simples.
- $\mathbf{CS}_B$ est l'ensemble des états composés. Nous définissons par $S_B = SS_B \cup CS_B$ l'ensemble de tous les états de *B*.
- $\mathbf{I}_B$ est l'ensemble des états initiaux, $I \subseteq S_B$.
- $\mathbf{T}_B$ est l'ensemble des transitions.
- $\mathbf{Prov\_Serv}_B$ est l'ensemble des services offerts par le bloc *B* à son environnement.
- $\mathbf{Req\_Serv}_B$ est l'ensemble des services requis par le bloc *B*.

L'ensemble de transitions $T \subseteq S_B \times L \times S_B$, où L est l'ensemble des étiquettes des transitions. Une étiquette $l \in L$ prend cette forme :

$$l = \textbf{REC} < ps > / \textbf{SND} < \{rs_i\}_{i=1..n} >$$

Où :

- *REC* représente la réception d'une requête, et *SND* représente l'émission d'une requête à l'environnement.
- $ps \in Prov\_Serv_B$.
- $\{rs_i\}_{i=1..n} = \mathcal{P}(Req\_Serv_B)$ est un sous ensemble de services requis par le bloc *B*.

Quand le bloc *B* reçoit une demande pour fournir l'un de ses services *'ps'*, cette réception peut provoquer une émission d'un ensemble de requêtes aux blocs adjacents.

Exemple 2 : Dans la figure 4, nous voyons clair que le protocole d'interaction du bloc

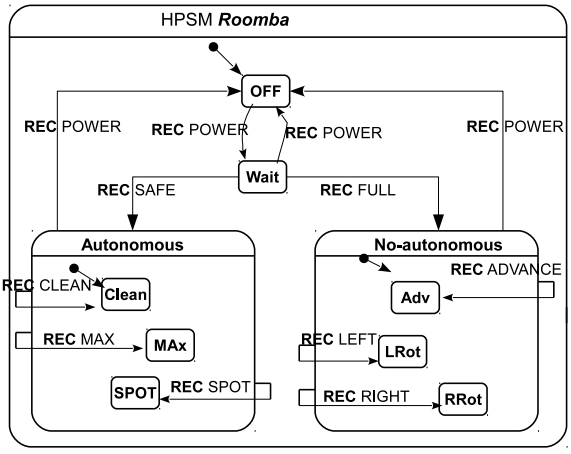


FIG. 5: HPSM du Roomba.

Roomba est partagé sur deux cas d'utilisations *"nettoyage automatique"* et *"nettoyage manuel"*. Dans la figure 5, nous montrons le protocole d'interaction de bloc *Roomba*, il prend la forme d'un HPSM.

Si on considère que le récepteur participe que dans le cas d'utilisation "Nettoyage manuel". Dans ce cas, le protocole d'interaction du récepteur peut être modélisé par le HPSM dans la figure 6.

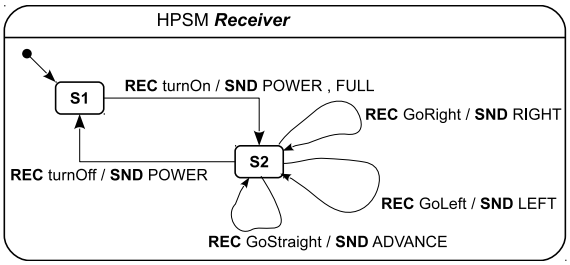


FIG. 6: HPSM du Récepteur.

5 HIA-ILT

Dans cette section, nous introduisons HIA-ILT (Hierarchical Interface Automata with Inter-Level Transitions), le modèle que nous utilisons, dans notre approche, pour formaliser les HPSMs des blocs, dans le but de vérifier formellement leurs interactions. En comparaison avec les IA simples, HIA-ILTs introduisent la notion des états composés et les transitions inter-niveaux. HIA-ILT diffère de HIA proposé dans Völgyesi (2004). En effet, dans HIA-ILT l'état source et cible d'une transition peuvent appartenir à deux différents états composés ou à deux différents niveaux de la hiérarchie, ce qui permet de modéliser des protocoles d'interaction complexes avec des modèles plus petits et plus clairs.

Définition 1 (HIA-ILT)
Nous définissons un HIA-ILT *'HA'* comme suit :

$$\langle SS_{HA}, CS_{HA}, I_{HA}, \sum_{HA}^{I}, \sum_{HA}^{O}, \sum_{HA}^{H}, \delta_{HA} \rangle$$

Où :

- SS_{HA} est l'ensemble des états simples. CS_{HA} est l'ensemble des états composés. I_{HA} est l'ensemble des états initiaux, $I_{HA} \subseteq SS_{HA} \cup CS_{HA}$.
- $\sum_{HA}^{I}$ est l'ensemble des actions d'entré, $\sum_{HA}^{O}$ est l'ensemble des actions de sortie, et $\sum_{HA}^{H}$ est l'ensemble des actions internes.
- δ_{HA} est l'ensemble des transitions,
  - $\delta_{HA} \subseteq (\text{SS}_{HA} \cup \text{CS}_{HA}) \times \sum_{HA} \times (\text{SS}_{HA} \cup \text{CS}_{HA})$, où $\sum_{HA} = \sum_{HA}^{I} \cup \sum_{HA}^{O} \cup \sum_{HA}^{H}$
  - (s_1, a, s_2) est une transition inter-niveaux si s_1 et s_2 n'appartiennent pas au même état composé, ou ils appartiennent à deux différents niveaux de la hiérarchie.

On peut calculer le produit synchrone abstrait entre deux HIA-ILTs, HA_1 et HA_2 si :

- HA_1 et HA_2 sont composables :
  $\Sigma_{HA_1}^{I} \cap \Sigma_{HA_2}^{I} = \Sigma_{HA_1}^{O} \cap \Sigma_{HA_2}^{O} = \Sigma_{HA_1}^{H} \cap \Sigma_{HA_2} = \Sigma_{HA_1} \cap \Sigma_{HA_2}^{H} = \emptyset$.

– aucun état composé de HA_1 et HA_2 possède à l'intérieur une transition étiquetée par une action qui appartient à Shared(HA_1, HA_2)

Définition 2 (Abstraction)
L'abstraction d'un HIA-ILT *"HA"* est un automate d'interface identique à HA, mais en considérant les états composites comme des états simples.

Définition 3 (Produit synchrone abstrait)
Le produit synchrone abstrait de deux HIA-ILTs HA_1 et HA_2 ($HA_1 \otimes_a HA_2$) est obtenu en appliquant le produit synchronede Alfaro and Henzinger (2001) entre l'abstraction des deux automates HA_1 et HA_2.

Définition 4 (Composition parallèle abstraite)
la composition parallèle abstraite de deux HIA-ILTs, HA_1 et HA_2 ($HA_1 \|_a HA_2$) consiste à éliminer de l'automate $HA_1 \otimes_a HA_2$ les états illégaux et les états atteignables à partir des états illégaux en activant des actions de sortie ou des actions internes.

Théorème
L'existence d'un état abstrait $(x, y) \in HA_1 \|_a HA_2$, où (x,y) est un état illégal, implique que tous les états à l'intérieur de l'état (x,y) sont des états illégaux.

Dans Bouaziz et al. (2015), nous avons présenté comment déduire le produit synchrone de deux HIA-ILTs à partir de produit abstrait, nous avons également donné la preuve de la théorème.

6 Notre Approche

Notre approche, de vérification de compatibilité des blocs SysML, est basée sur un ensemble d'étapes. Ces étapes dépendent de deux cas :

1. Les blocs à assembler (sur lesquels on doit vérifier la compatibilité), appartiennent au même système (voir figure 7). Dans ce cas là, il n y a pas de décalage entre les interfaces des blocs.
2. Les blocs à vérifier leur compatibilité, sont conçus séparément : c-à-d. que les blocs sont des blocs réutilisables, ils ont été conçus indépendamment du système auquel ils vont faire partie (voir figure 17). Donc, on peut trouver de décalage entre les interfaces des blocs.

6.1 Les blocs appartiennent au même système

Quand les blocs à assembler, dont on veut vérifier leur compatibilité, appartiennent au même système, on applique le processus sur la figure 7. Ce processus comporte quatre étapes :
étape 1 : Mapping entre HPSM et HIA-ILT

Chaque état simple dans HPSM doit être transformé en un état simple dans le HIA-ILT équivalent, et chaque état composé doit être copié comme un état composé. La différence entre le HPSM et le HIA-ILT réside principalement dans les transitions. Une transition dans HPSM

la vérification de la compatibilité des blocs sysml

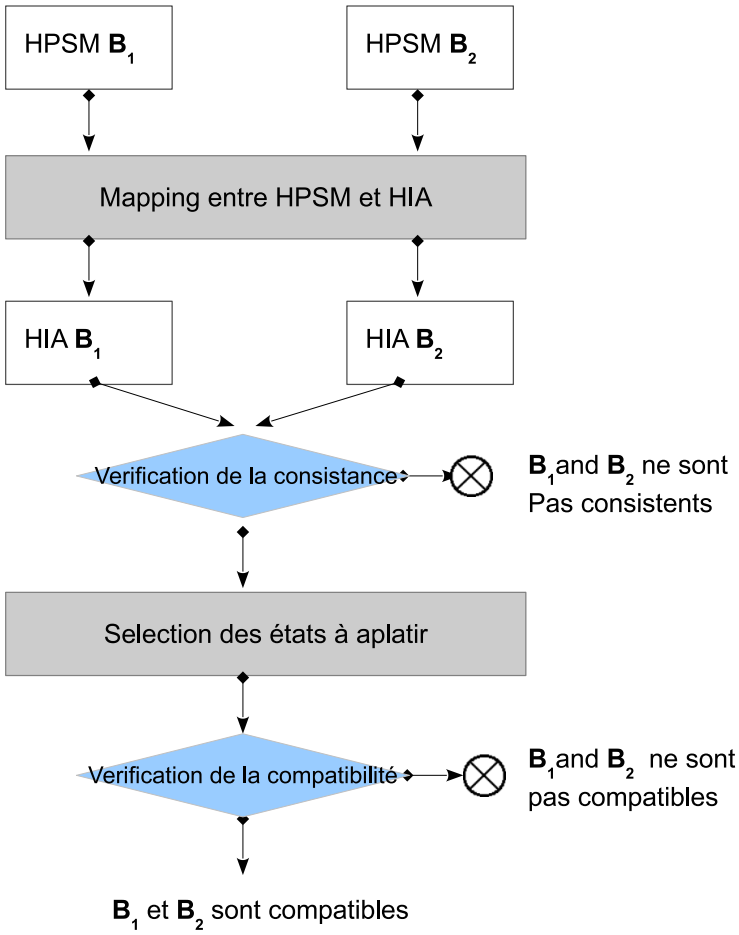


FIG. 7: Processus pour vérifier la compatibilité des blocs appartenant au même système.

peut être décomposée en un ensemble de transitions HIA. Dans la figure 8, nous montrons qu'une transition étiquetée par un service fourni ($\xrightarrow{ps}$) doit être transformée en une transition étiquetée par une action d'entrée ($\xrightarrow{ps?}$). L'ensemble de services requis sur une transition HPSM ($\xrightarrow{/rs_1,..,rs_n}$) doivent être transformés en une séquence de transitions étiquetées par des actions de sortie ($\xrightarrow{rs_1!}...\xrightarrow{rs_n!}$) (voir figure 8).

Pour automatiser cette transformation, nous avons généré des éditeurs pour HPSM et HIA-ILT en utilisant EMF. Nous avons également défini une grammaire ATL 'HPSM2HIA-ILT'.

Pour générer l'éditeur *HPSM*, nous avons défini le méta-modèle HPSM (voir figure 9). La même chose pour l'éditeur de *HIA-ILT* (voir figure 10).

La grammaire ATL 'HPSM2HIA-ILT' contient 8 règles (dont 6 règles sont de type 'rule', une règle est de type 'lazy rule' et la dernière règle est de type 'called rule') et un 'helper'. Dans la figure 11, nous montrons la règle *'CompositeState2CompositeState'*, c'est la règle qui

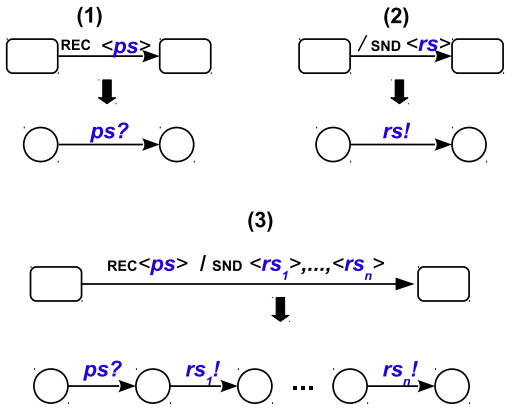


FIG. 8: Correspondances entre HPSM et HIA-ILT.

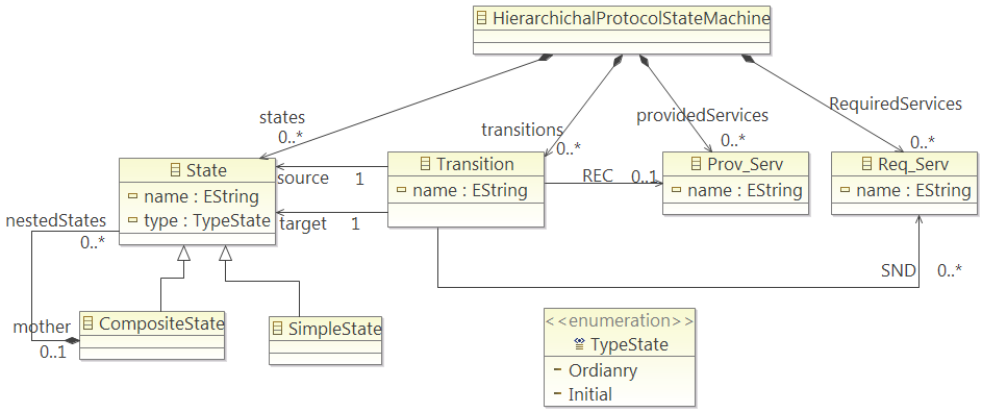


FIG. 9: Méta-modèle HPSM.

permet de générer les états composés de HIA-ILT à partir des états composés de HPSM.

Exemple 3 :

Dans la figure 12, nous montrons le résultat de la transformation des HPSMs (dans la figure 5 et la figure 6) en des HIA-ILTs. La première remarque que nous pouvons voir est que chaque service fourni au niveau du HPSM est transformé en une action d'entrée, et chaque service requis est transformé en une action de sortie, où chaque action correspond à un port.

Dans le HPSM du récepteur, toutes les transitions sont de type *REC ps / SND rs*, donc chaque transition sera transformée en deux transitions au niveau de HIA-ILT. Sauf la transition $(S1,' RECturnOn/SNDPOWER, FULL', S2)$, elle sera découpée en trois transitions $\xrightarrow{turnOn?}$, $\xrightarrow{POWER!}$ et $\xrightarrow{FULL!}$.

Eclipse génère des fichier xmi pour les instances crées à partir d"un meta-modèle EMF. Le

la vérification de la compatibilité des blocs sysml

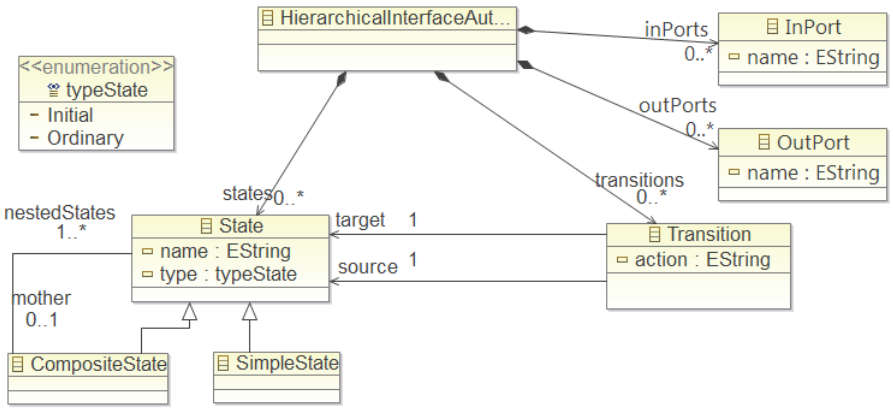


FIG. 10: Méta-modèle HIA-ILT.

```
rule CompositeState2CompositeState {
    from
        cs1:HPSM!CompositeState
        using {
            i: Integer=1;
            transitions:Sequence(MM!Transition) = MM!Transition.allInstances()->
            select(t | cs1.nestedStates.includes(t.source) or cs1.nestedStates.includes(t.target));
            states :Sequence(MM1!State) = transitions->iterate(t; statesList: Sequence(MM1!State) =Sequence{} |
            statesList.union( thisModule.createStatesForTransition(t)));}
    to
        cs2: HIAILT!CompositeState(
            name <- cs1.name,
            mother <- cs1.mother,
            type <- if cs1.type=#Initial then  cs1.type else #Ordinary endif,
            nestedStates <- states)
    do {
        for (t in transitions ){
            if (not t.REC.oclIsUndefined() ){
                thisModule.createTransition(states.at(i), t.REC.name+'?', states.at(i+1));
                i<-i+1;
            }
            if (not t.SND.oclIsUndefined() ){
                for (a in t.SND){
                    thisModule.createTransition(states.at(i), a.name+'!', states.at(i+1));
                    i<-i+1;
                }
            }
            i<-i+1;
        } }}
```

scale=0.65

FIG. 11: exemple d'une règle ATL.

fichier xmi généré par Eclipse pour le HPSM du récepteur est dans la figure 13. En appliquant la grammaire ATL 'HPSM2HIA-ILT sur le HPSM du récepteur, nous avons obtenu le fichier

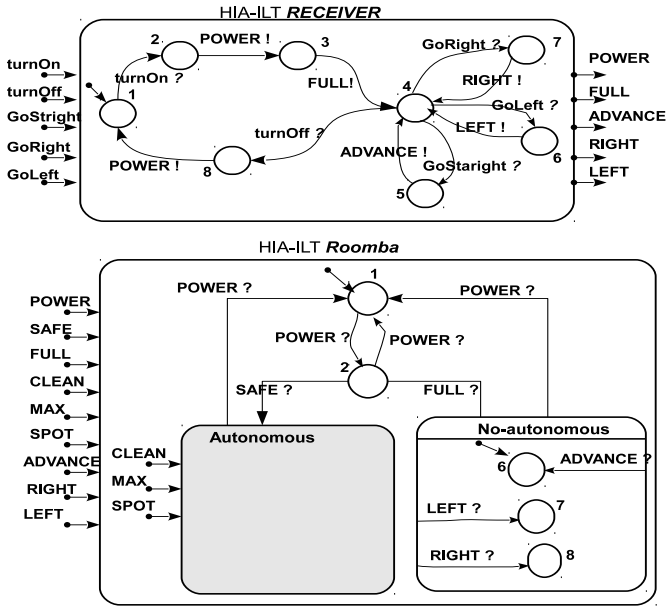


FIG. 12: HIA-ILT de récepteur et de roomba.

```
< ?xml version="1.0" encoding="ASCII" ?>
<HPSM :HierarchichalProtocolStateMachine xmi :version="2.0"
xmlns :xmi="http ://www.omg/XMI" xmlns :xsi="http ://www.w3.org/2001/XMLSchema-instance"
xmlns :HPSM="www.HPSM.com" xsi :schemaLocation="www.HPSM.com HPSM.ecore">
<states xsi :type="HPSM :SimpleState" name="s1" type="Initial"/>
<states xsi :type="HPSM :SimpleState" name="s1"/>
<transitions source="//@states.0" target="//@states.1" REC="//@providedServices.0"
        SND="//@RequiredServices.0 //@RequiredServices.4" name="t1"/>
<transitions source="//@states.1" target="//@states.0" REC="//@providedServices.1"
         SND="//@RequiredServices.0" name="t2"/>
<transitions source="//@states.1" target="//@states.1" REC="//@providedServices.4"
        SND="//@RequiredServices.3" name="t3"/>
<transitions source="//@states.1" target="//@states.1" REC="//@providedServices.3"
        SND="//@RequiredServices.1" name="t4"/>
<transitions source="//@states.1" target="//@states.1" REC="//@providedServices.2"
         SND="//@RequiredServices.2" name="t5"/>
<providedServices name="turnOn"/> <providedServices name="turnOff"/>
<providedServices name="GoRight"/> <providedServices name="GoLeft"/>
<providedServices name="GoStraight"/> <RequiredServices name="POWER"/>
<RequiredServices name="LEFT"/> <RequiredServices name="Right"/>
<RequiredServices name="ADVANCE"/> <RequiredServices name="FULL"/>
</HPSM :HierarchichalProtocolStateMachine>
```

FIG. 13: fichier xmi de HPSM de récepteur.

la vérification de la compatibilité des blocs sysml

```
< ?xml version="1.0" encoding="ASCII" ?>
<HPSM :HierarchichalProtocolStateMachine xmi :version="2.0"
xmlns :xmi="http ://www.omg/XMI" xmlns :xsi="http ://www.w3.org/2001/XMLSchema-
instance"
   xmlns :HIA="www.HIA.com">
<states xsi :type="HIA :SimpleState" name="states.0"/>
<states xsi :type="HIA :SimpleState" name="states.1" type="Ordinary"/>
<states xsi :type="HIA :SimpleState" name="states.2" type="Ordinary"/>
<states xsi :type="HIA :SimpleState" name="states.3" type="Ordinary"/>
<states xsi :type="HIA :SimpleState" name="states.4" type="Ordinary"/>
<states xsi :type="HIA :SimpleState" name="states.5" type="Ordinary"/>
<states xsi :type="HIA :SimpleState" name="states.6" type="Ordinary"/>
<states xsi :type="HIA :SimpleState" name="states.7" type="Ordinary"/>
<transitions source="//@states.0" target="//@states.2" action="turnOn ?"/>
<transitions source="//@states.1" target="//@states.4" action="turnOff ?"/>
<transitions source="//@states.1" target="//@states.5" action="GoStraight ?"/>
<transitions source="//@states.1" target="//@states.6" action="GoLeft ?"/>
<transitions source="//@states.1" target="//@states.7" action="GoRight ?"/>
<transitions source="//@states.2" target="//@states.3" action="POWER !"/>
<transitions source="//@states.3" target="//@states.1" action="FULL !"/>
<transitions source="//@states.4" target="//@states.0" action="POWER !"/>
<transitions source="//@states.5" target="//@states.1" action="ADVANCE !"/>
<transitions source="//@states.6" target="//@states.1" action="LEFT !"/>
<transitions source="//@states.7" target="//@states.1" action="RIGHT !"/>
<inPorts name="turnOn"/> <inPorts name="turnOff"/>
<inPorts name="GoRight"/> <inPorts name="GoLeft"/>
<inPorts name="GoStraight"/> <outPorts name="POWER"/>
<outPorts name="LEFT"/> <outPorts name="RIGHT"/>
<outPorts name="ADVANCE"/> <outPorts name="FULL"/>
</HIA :HierarchichalInterfaceAutomaton>
```

FIG. 14: fichier xmi de HIA-ILT de récepteur.

qui correspond au HIA-ILT du récepteur (voir figure 14).

étape 2 : La vérification de la consistance

Deux blocs B1 et B2 sont consistants si leurs HIA-ILTs, respectivement HA_1 et HA_2 sont composables. La relation de consistance ζ entre B1 et B2 est définie par :

$$B1\ \zeta\ B2 \Leftrightarrow \Sigma^I_{HA_1} \cap \Sigma^I_{HA_2} = \Sigma^O_{HA_1} \cap \Sigma^O_{HA_2} = \Sigma^H_{HA_1} \cap \Sigma_{HA_2} = \Sigma_{HA_1} \cap \Sigma^H_{HA_2} = \emptyset.$$

Exemple 4 :

- $\Sigma^I_{receiver}$= *{turnOn, turnOff, GoRight, GoLeft, GoStraight}*
- $\Sigma^O_{receiver}$ = *{ POWER, FULL ,RIGHT, LEFT, ADVANCE}*
- Σ^I_{roomba} = *{POWER, SAFE, FULL, MAX, SPOT, CLEAN, RIGHT, LEFT, ADVANCE}*

- $\Sigma^{O}_{roomba} = \emptyset$
- $\Sigma^{I}_{receiver} \cap \Sigma^{I}_{roomba}$ = $\Sigma^{O}_{receiver} \cap \Sigma^{O}_{roomba}$ = $\Sigma^{H}_{receiver} \cap \Sigma_{roomba}$ = $\Sigma_{receiver} \cap \Sigma^{H}_{roomba}$ = $\emptyset$. $\Rightarrow receiver\ \zeta\ roomba$, le récepteur et roomba sont composables.

étape 3 : La sélection des états à aplatir
Dans cette étape, il faut construire l'ensemble :

$$\textit{Shared}(HA1,HA2) = (\Sigma^{I}_{HA1} \cap \Sigma^{O}_{HA2}) \cup (\Sigma^{O}_{HA1} \cap \Sigma^{I}_{HA2}).$$

$Shared(HA1, HA2) \neq \emptyset$ signifie que *B1* et *B2* doivent synchroniser sur les actions de cette ensemble. Donc, nous sélectionnons tous les états composés de *B1* et *B2* qui possèdent à l'intérieur une transition étiquetée par une action appartenant à $Shared(HA1, HA2)$. Pour aplatir ces états, nous exploitons les travaux publiés dans David et Moller (2001).

Exemple 5 :

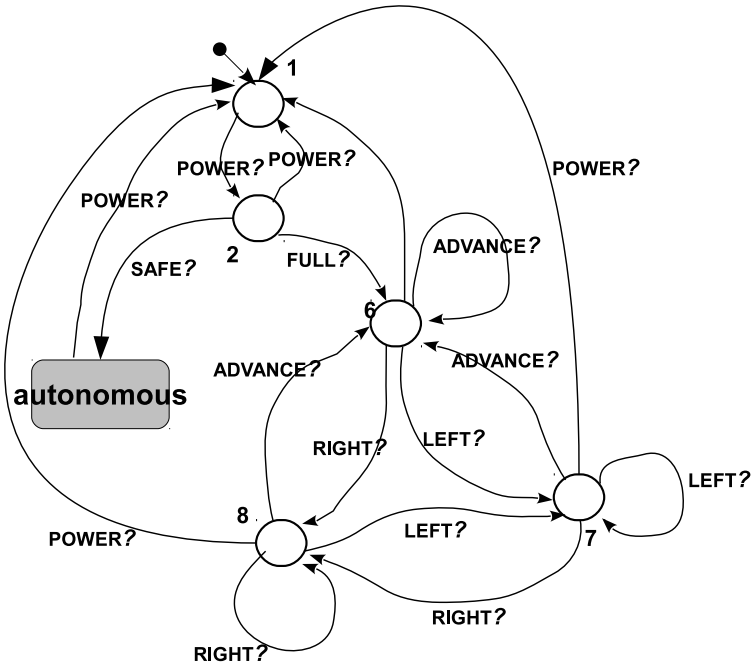


FIG. 15: Aplatissement de l'état 'autonomous' de roomba.

Shared($receiver$,$roomba$) = $(\Sigma^{I}_{receiver} \cap \Sigma^{O}_{roomba}) \cup (\Sigma^{O}_{receiver} \cap \Sigma^{I}_{roomba}) = \{POWER, FULL, RIGHT, LEFT, ADVANCE\}$

Donc l'état *'no-autonomous'* doit être aplati parce qu'il contient des transitions étiquetées par des actions appartenant à l'ensemble *Shared*($receiver$,$roomba$), alors que ce n'est pas le cas pour l'état *'autonomous'* (figure 15).

étape 4 : La vérification de la compatibilité

La vérification de la compatibilité entre deux blocs B1 et B2 revient à vérifier la compatibilité entre leurs HIA-ILTs HA_1 et HA_2. La relation de la compatibilité ζom entre B1 et B2 (leurs protocoles d'interaction sont modélisés par HA_1 et HA_2) est définie par :
B1 ζom B2 $\Leftrightarrow$ $HA_1 \parallel_a HA_2$ contient au moins un état, c.-à-d. que les blocs B1 et B2 peuvent se retrouver au moins dans un état cohérente.

Exemple 6 :

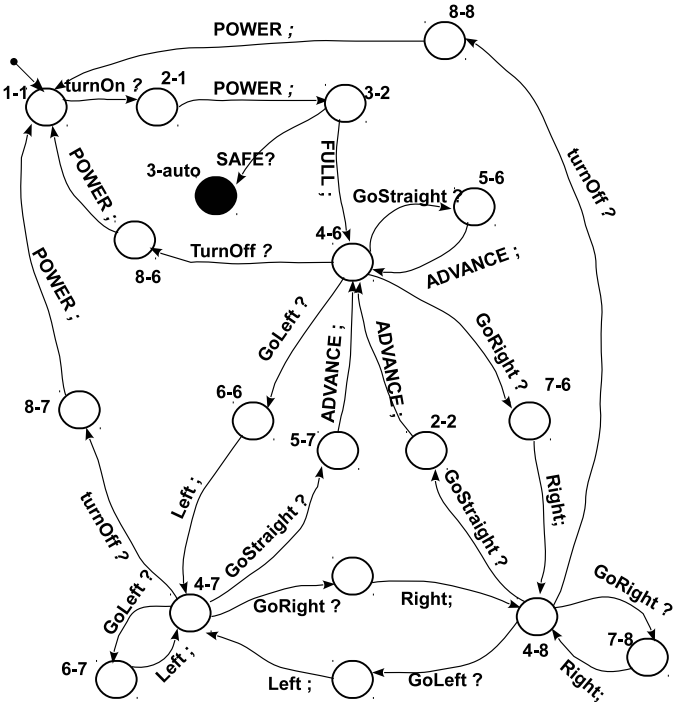


FIG. 16: $HA_{receiver} \otimes_a HA_{roomba}$

Dans la figure 16, nous présentons $HA_{receiver} \otimes_a HA_{roomba}$. Pour calculer la composition parallèle à partir de résultat du produit, on doit éliminer l'état *'3-auto'* parce ce que c'est un état illégal :

full $\in$ *Shared(*$receiver$, $roomba$*)*, *full* $\in \Sigma_3^O$ et *full* $\notin \Sigma_{auto}^I$.

L'automate $HA_1 \parallel_a HA_2$ n'est pas vide, donc il est possible d'assembler ce récepteur et ce roomba dans un même système.

6.2 Les blocs conçus séparément

Dans le cas où les blocs sont conçus séparément, on trouve le problème de décalage entre les libellés des actions, où une action de sortie d'un bloc correspond à une action d'entrée d'un autre bloc, mais elles ne portent pas le même libellé. Dans ce cas là, pour vérifier la compatibilité des blocs SysML, nous appliquons le processus décrit dans la figure 17.

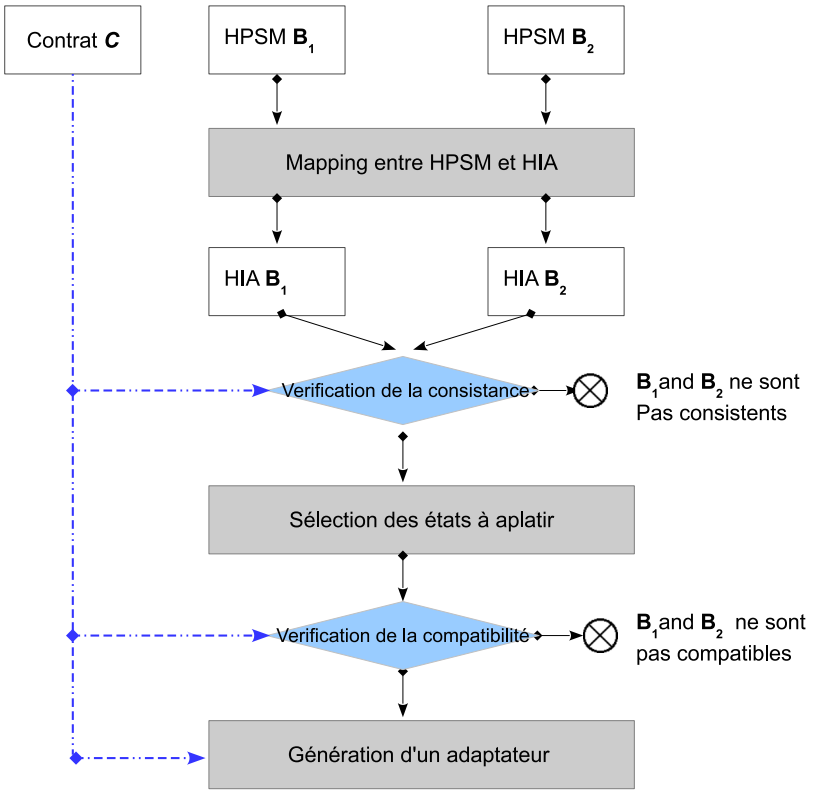


FIG. 17: Processus pour vérifier la compatibilité des blocs conçus séparément.

En comparaison avec le premier cas, dans ce cas, nous prenons en considération un autre élément qui est le contrat ***C***. Ce contrat spécifie la correspondance entre les actions d'entrée et les actions de sortie des blocs. Nous introduisons également une étape pour générer l'architecture d'un adaptateur qui remplace le contrat et joue le rôle d'un médiateur entre les blocs.

Le contrat ***C*** prend la forme d'une matrice à deux colonnes et ***n*** lignes, où ***n*** et le nombre des actions d'entrée et de sortie de bloc *B1* qui correspondent à des actions de bloc *B2*.

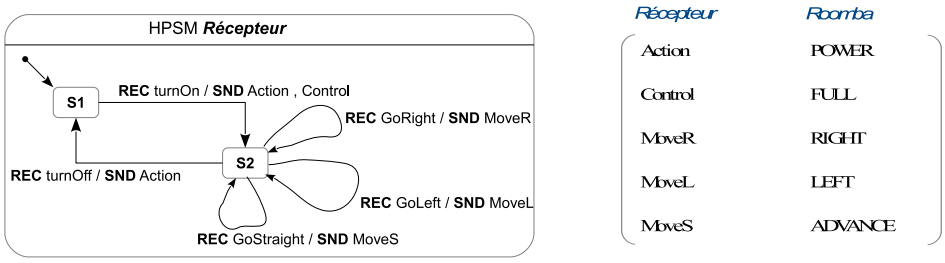
Exemple 7 :

Si nous remplaçons le bloc *Récepteur* dans la figure 6 par le récepteur dans la figure 18a. Dans ce cas là, nous avons besoin de spécifier un contrat (voir figure 18b) pour établir les correspondances entre le *Récepteur* dans la figure 18a et *Roomba* dans la figure 5.

La phase de transformation de HPSMs aux HIA-ILTs n'est pas concernée par aucun changement, il suffit d'appliquer les règles de la grammaire *'HPSM2HIA-ILT'*. Par contre, la phase de la vérification de consistance, elle doit prendre en considération le contrat qui spécifie les correspondances entre ***B1*** et ***B2***.

Pour vérifier la consistance de ***B1*** et de ***B2***, nous avons défini la fonction *'Correspondance'*. Elle prend comme paramètres une action de bloc ***B1*** et le contrat ***C***, et elle retourne l'action de ***B2*** qui correspond à l'action en entrée selon le contrat ***C***.

la vérification de la compatibilité des blocs sysml



(a) HPSM Récepteur.

(b) Contrat Récepteur -Roomba.

FIG. 18: Les correspondances entre le bloc *Roomba* et le bloc *Récepteur*.

$$\text{Correspondance} : \Sigma_{HA1} \times C \rightarrow \Sigma_{HA2} \cup \{\epsilon\}$$

Donc, le bloc B1 et B2 sont consistant si leurs automates d'interface hiérarchiques respectent cette condition :

$$\begin{aligned} B1 \,\zeta\, B2 \Leftrightarrow \{\text{a} \mid \text{a} \in \Sigma^{I}_{HA_1} \wedge \text{Correspondance(a)} \in \Sigma^{I}_{HA_2}\} = \\ \{\text{a} \mid \text{a} \in \Sigma^{O}_{HA_1} \wedge \text{Correspondance(a)} \in \Sigma^{O}_{HA_2}\} = \\ \Sigma^{H}_{HA_1} \cap \Sigma_{HA_2} = \Sigma_{HA_1} \cap \Sigma^{H}_{HA_2} = \emptyset. \end{aligned}$$

La troisième étape de notre processus se résume à une sélection des états à aplatir. Les états sélectionnés sont les états composés de B1 et B2 qui ne contiennent pas une transition étiquetée par une action partagée. Dans ce cas, nous pouvons définir l'ensemble des actions partagées entre l'automate de ***B1*** et l'automate de ***B2*** comme suit :

$$\begin{aligned} \text{Shared(HA1,HA2)} = \{\ \text{a} \in \Sigma_{HA1} \mid \text{Correspondance(a)} \neq \epsilon\ \} \cup \\ \{\ \text{b} \in \Sigma_{HA2} \mid \text{b= Correspondance(a)} \wedge \text{b} \neq \epsilon\ \} \end{aligned}$$

La quatrième étape consiste à vérifier la compatibilité de blocs *B1* et *B2*. Nous vérifions si la composition parallèle de HA1 et HA2 contient au minimum un état. Donc, il faut tout d'abord passer par le produit synchrone de HA1 et HA2 en se basant sur le contrat *C*. Le produit synchronise sur les actions partagées, mais il ne réduit pas les transitions de type $(s_1,s_2) \xrightarrow{a} (s'_1,s_2) \overset{Correspondance(a)}{\longrightarrow} (s'_1,s'_2)$) en une seule transition $(s_1,s_2) \xrightarrow{a;} (s'_1,s'_2)$.

La dernière étape se base sur le résultat de cette quatrième étape, et exactement sur le résultat de la composition parallèle de HA1 et HA2. Si cette composition n'est pas vide (c-à-d. le bloc B1 et le bloc B2 sont compatibles), nous pouvons donc générer un adaptateur pour gérer le décalage des libellés des actions.

Le protocole d'interaction de bloc adaptateur est le résultat de la composition parallèle de HA1 et HA2, mais avec des types des actions inversés. Chaque action d'entrée devient une action de sortie et vice-versa. Cette dernière opération est nécessaire, parce que quand l'un des blocs demande un service, c'est à l'adaptateur de recevoir cette demande, et quand un bloc recoit une demande de l'un de ses services, c'est aussi à l'adaptateur d'envoyer cette demande à ce bloc.

7 Conclusion

Dans ce papier, nous avons présenté une approche de vérification de compatibilité des systèmes à base de composants basée sur HPSM, le diagramme que nous avons proposé pour la modélisation et la représentation des protocoles d'interaction des blocs SysML. Nous avons également présenté le modèle formel HIA-ILT, une variante des automates d'interface, qui permet l'utilisation des états composés et des transitions inter-niveaux. Nous avons fourni les correspondances entre ces deux modèles, et nous avons montré comment exploiter la hiérarchie présente dans HIA-ILT pour vérifier la compatibilité entre les blocs SysML en vue d'alléger la phase de vérification de la compatibilité. Cette vérification est étudiée dans les deux cas : où les blocs sont conçus pour un même système, et le cas où les blocs sont conçus séparément. Dans ce dernier cas, nous avons montré comment gérer le décalage qui se présente entre les libellés des services des blocs.
Comme perspective de ce travail, nous projetons d'appliquer notre approche sur des protocoles d'interaction plus complexes. Nous comptons également étendre notre approche par une phase de vérification des exigences exprimées sur l'assemblage.

Références

ATL. `https://eclipse.org/atl/`.

Bouaziz, H., S. Chouali, A. Hammad, et H. Mountassir (2015). Compatibility Verification of SysML Blocks Using Hierarchical Interface Automata. In *ISPS 2015, 12th Int. Symposium on Programming and Systems*, Algiers, Algeria, pp. 313–322. IEEE.

Carrillo, O., S. Chouali, et H. Mountassir (2012). Formalizing and Verifying Compatibility and Consistency of SysML Blocks. *ACM SIGSOFT Software Engineering Notes 37*(4), 1–8.

David, A. et M. O. Moller (2001). From HUPPaal to Uppaal : A Transition from Hierarchical Timed Automata to Flat Timed Automata. Technical report, BRICS, University of Aarhus, Denmark.

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

Eclipse. `https://eclipse.org/`.

Eles, P., Z. Peng, et D. Karlsson (2002). Formal Verification in a Component-Based Reuse Methodology. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS 2002), October 2-4, 2002, Kyoto, Japan*, pp. 156–161.

EMF. `https://eclipse.org/modeling/emf/`.

Inverardi, P. et M. Tivoli (2001). Automatic Synthesis of Deadlock-free Connectors for COM/DCOM Applications. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pp. 121–131.

Inverardi, P. et M. Tivoli (2003). Deadlock-free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software 65*(3), 173–183.

Karlsson, D., P. Eles, et Z. Peng (2007). Formal Verification of Component-based Designs. *Design Autom. for Emb. Sys. 11*(1), 49–90.

Linhares, M. V., A. J. da Silva, et R. S. de Oliveira (2006). Empirical Evaluation of SysML through the Modeling of an Industrial Automation Unit. In *ETFA*, Prague, Czech Republic, pp. 145–152.

Manna, Z. et A. Pnueli (1991). Completing the temporal picture. *Theor. Comput. Sci. 83*(1), 91–130.

Pihlanko, P., S. Sierla, K. Thramboulidis, et M. Viitasalo (2013). An Industrial Evaluation of SysML : The Case of a Nuclear Automation Modernization Project. In *ETFA*, Cagliari, Italy, pp. 1–8.

SysML (2012). OMG Systems Modeling Language (OMG SysML) Version 1.3.

Völgyesi, P. (2004). Robust Software Composition for Sensor Networks. In *International Carpathian Control Conference, Zakopane, Poland*, pp. 25–28.

Summary

The development of components based systems consists on assembling a set of basic units, where each unit covers a part of system requirements. This approach allows the reduction of development cost. However, the assembling operation requires the adoption of a complete and less costly verification approach. In this paper, we propose a formal approach to verify the compatibility of SysML blocks where the goal is to study the possibility of their composition. Essentially, a SysML specification of a system consists of representing its structure in a form of a blocks set in interaction, this interaction can be modelled with models which exposes a level of hierarchy. Thus, our approach aims to benefit from the hierarchy that we find in SysML models and in automata to alleviate the verification of SyML blocks compatibility.