Analyse d'un modèle AADL à l'aide de Pola

Pierre-Emmanuel Hladik[1,2], Florent Peres[3], Xiaomu Shi[4]
[1] CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
[2] Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France
[3] Université de Lille Nord de France - F59000 Lille, INRETS, ESTAS, F59650 Villeneuve d'Ascq
[4] VERIMAG - Centre quation, 2 avenue de Vignate, F-38610 Gières, France
pehladik@laas.fr; florent.peres@inrets.fr; xiaomu.shi@doctorant.univ-grenoble.fr

Résumé

Cet article présente l'intégration d'un langage formel de vérification, Pola, dans le processus de conception des systèmes temps réel critiques en se basant sur le langage AADL (Architecture Analysis and Design Language). Pola (Policies analyzer) est un langage dédié (Domain Specific Language) qui permet de décrire le comportement dynamique et temporel de systèmes temps réel complexes en vue de leur vérification (ordonnançabilité, absence de blocages, etc), laquelle est effectuée de manière automatique par *model checking*. Le travail présenté dans cet article consiste à utiliser les méthodes et outils offerts par l'ingénierie des modèles pour transformer un modèle AADL vers un modèle Pola. Une attention particulière est portée sur les informations liées à l'ordonnancement qui sont contenues dans les modèles AADL et leur traduction dans le contexte Pola. Un premier outil utilisant le langage de transformation ATL et l'outil Acceleo a été développé pour en valider la faisabilité.

Mot-clés : AADL, *model checking*, analyse d'ordonnançabilité, transformation de modèles

1 Introduction

Les exigences et la complexité des systèmes embarqués temps réel devenant croissantes, il est nécessaire de disposer de méthodes et d'outils pour assister les concepteurs lors du processus de développement. Pour cela l'ingénierie des modèles offre des solutions intéressantes et prometteuses.

Actuellement, deux langages sont particulièrement étudiés dans le cadre de la modélisation des systèmes embarqués critiques : UML avec le profil MARTE [10] et AADL (Architecture Analysis and Design Language) [12]. Afin d'effectuer la vérification des modèles issus de ces langages, des chaînes d'outils sont développées qui adoptent des points de vues différents. Dernièrement, le projet européen SPICES [19] recensait pas moins d'une douzaine d'outils et de méthodes autour du langage AADL permettant de vérifier des propriétés sur l'occupation mémoire, l'ordonnancement, la consommation d'énergie, la génération de code, etc..

Au niveau de la vérification du comportement temporel d'un système temps réel critique plusieurs méthodes et approches existent. Toutes se basent sur la transformation d'un modèle haut niveau écrit en AADL ou MARTE vers des modèles spécialisés pour la vérification. Ces derniers, sont soit dédiés à des outils spécifiques — comme Cheddar [18] pour la simulation ou MAST [11] pour l'analyse d'ordonnançabilité — soit se basent sur des sémantiques fortes — comme pour le langage Fiacre [3] ou IF [1] — pour ensuite être utilisés avec des méthodes de vérification formelles, par exemple à base de réseaux de Petri ou d'automates.

Cet article présente une méthode intermédiaire consistant en l'introduction de Pola [17], un langage formel de vérification dédié aux systèmes temps réel, au sein d'une chaîne de transformation et de vérification débutant par un modèle du langage AADL. La partie 2 introduit rapidement les principaux concepts utilisés dans AADL et présente quelques outils existants permettant l'analyse comportementale de modèles AADL. La partie 3 est une introduction au langage Pola et à la chaîne d'outils utilisée pour mener la vérification. Dans la partie 4, la démarche adoptée dans cet article est mise en perspective avec les outils et méthodes existants. La partie 5 fait une synthèse sur la transformation des composants AADL vers Pola. La partie 6 aborde l'implémentation de cette transformation et sa validation expérimentale. La dernière partie conclue le travail et introduit quelques perspectives.

2 Présentation de AADL

AADL (Architecture Analysis and Design Language) est un langage de description d'architecture conçu pour les systèmes temps réel embarqués dans les domaines avionique, spatial, robotique, santé, etc. Il a été développé à partir des retours d'expérience de MethaH et a été porté comme base de standardisation sous l'autorité du SAE (International Society for Automotive Engineers) et de son ASD (Avionics System Division).

En novembre 2004, le SAE a produit la première version du standard AADL (SAE AS5506). Depuis, des compléments ont été apportés et une version 2.0 a été produite en janvier 2009 [12]. Le standard SAE AADL est composé de : une spécification du langage avec une syntaxe textuelle, une sémantique et une représentation graphique ; un profil UML du SAE AADL ; d'une spécification XML/XMI comme format de modèle ; et plusieurs annexes dont l'une formalisant le comportement, l'une sur la conformité des interfaces avec C et Ada, l'une sur une extension du modèle d'erreur, etc.

Le langage AADL étant extrêmement riche, il est impossible d'en donner ici une vue exhaustive. Les lecteurs peuvent se référer aux notes techniques disponibles sur le site de la SAE pour avoir plus de détails ou bien directement se référer au standard [12] pour disposer de la syntaxe et de la sémantique.

Le standard SAE AADL [12] fournit les concepts de modélisation par composant d'une architecture logicielle et matérielle pour les systèmes temps réel critiques. Il permet de décrire le moteur d'exécution d'application en terme de tâches concurrentes ainsi que leurs interactions et leur allocation sur le matériel. Une architecture est décrite par une hiérarchie de composants. Nous ne présentons ici que les composants manipulés dans cet article.

Un composant logiciel est : soit un *thread* pour représenter le flot séquentiel d'exécution d'une fonction et modéliser une unité ordonnançable ; soit un *process* pour représenter un composant structuré pour les groupes logiques de *thread*, *data* et *thread group* ; soit un *data* pour modéliser les données manipulées dans le code source et les protocoles de gestion de partage en cas d'accès concurrent ; soit un *subprogram* (qui peut être appelé à partir de *thread* ou d'autres *subprogram*) pour représenter le point d'entrée d'une exécution dans le code source.

La plate-forme d'exécution est modélisée à l'aide du composant *processor* qui est une abstraction du matériel et du logiciel responsable de l'ordonnancement et de l'exécution des *threads*.

Le composant composite *system* qui fait le lien entre les applications logicielles et la plate-forme d'exécution.

Chaque composant est décrit en AADL par son type et son interface fonctionnelle, visible par les autres composants. L'interface comprend des propriétés (*properties*) avec les valeurs des attributs et des caractéristiques du composant, ainsi que des *feature* qui spécifient comment le composant est interfacé avec les autres. Plusieurs catégories de *feature* sont distinguées : les

port qui sont les interfaces de communication pour les échanges de données ou d'événements ; les *subprogam* qui sont les interfaces d'appel entrant ou sortant des sous-programmes ; et les *subcomponent access* qui représentent une donnée interne accessible par un composant externe.

L'implémentation d'un composant définit sa structure interne en terme de sous-composants et de connections. Elle autorise la spécification des valeurs des propriétés non-fonctionnelles du composant, des états opérationnels ainsi que des instants de l'exécution permettant de passer d'un état à l'autre lors d'un changement de *mode*.

AADL peut être étendu pour introduire des spécificités propre à chaque application en plus des propriétés pré-définies. Ces extensions facilitent les modélisations spécifiques et permettent des analyses. Elles se présentent sous la forme d'annexes dans lesquelles sont déclarées les extensions du langage sous forme conceptuelle et syntaxique. Dans le travail présenté ici, il a été décidé de ne pas introduire de nouvelles propriétés pour procéder à l'analyse du comportement, mais uniquement d'utiliser celle qui sont fournies dans le standard.

2.1 Exemple d'un système en AADL

La figure 1 contient la description en AADL d'une application. Elle se compose d'un *process* (lignes 14–24) contenant trois *threads* (lignes 40–51, 53–62 et 64–74). Un *processor* (ligne 26–32) sert de support d'exécution aux *threads* (lignes 9–11). Une donnée est déclarée, typée et implémentée (ligne 34–38).

Les *thread* `thr1` et `thr2` sont activés périodiquement (lignes 42–43 et 55–56), alors que le *thread* `thr3` est apériodique (ligne 59) et réveillé sur la terminaison du *thread* `thr1` (ligne 23).

Les *thread* `thr1` et `thr2` partagent l'accès à la données `data1` (lignes 47 et 61). Le protocole de partage n'est pas spécifié.

La politique d'ordonnancement est définie comme une propriété du *processor* `proc` (ligne 28) et suit la politique `DMS` (*Deadline Monotonic Scheduling*), c'est-à-dire que la priorité des *threads* est d'autant plus grande que leur échéance est petite. Ici, le *thread* `thr3` est plus prioritaire que `thr1`, qui est lui-même plus prioritaire que `thr2`.

Remarquons que les valeurs que peut prendre la propriété `Scheduling_Protocol` ne sont pas formalisées dans l'annexe A du standard AADL. Cette liste est *project-specified*, c'est-à-dire que les valeurs doivent être renseignées par le concepteur. Cela pose un problème important pour la vérification automatique, car tous les aspects non renseignés ou laissés à l'arbitraire du concepteur sont autant de « trous » sémantiques qu'il convient impérativement de combler avant de pouvoir espérer obtenir une quelconque information sur le système. Or, ici, combler un trou sémantique signifie rajouter de nouvelles règles au moteur de transformation. La définition de la sémantique de la politique `DMS` fait l'objet d'un très large consensus, ainsi cette politique est naturellement bien connue et de ce fait est incluse dans la transformation. Ceci peut bien sûr ne pas être le cas pour une politique plus confidentielle, dont il va falloir préciser la sémantique avant de pouvoir en faire l'analyse, puis intégrer cette nouvelle politique au traducteur sous forme de nouvelles règles de traduction.

2.2 Vérification comportementale d'un modèle AADL

Plusieurs travaux ont été menés autour de la vérification comportementale des systèmes modélisés à l'aide d'AADL. L'un des points sur lequel une attention particulière est portée est celui de l'ordonnançabilité. Pour cela divers outils sont utilisés conjointement avec une transformation d'une application décrite en AADL.

L'une des méthodes proposée utilise Cheddar [18], un outil offrant quelques méthodes d'analyse d'ordonnançabilité en plus d'un simulateur. Pour cela AADL est étendu par le biais d'une

```
system sys
end sys;

system implementation sys.imp
subcomponents
  pro1: process process1.impl1;
  proc: processor proc.impl1;
properties
  Actual_Processor_Binding =>
reference proc applies to pro1.T1;
  Actual_Processor_Binding =>
reference proc applies to pro1.T2;
  Actual_Processor_Binding =>
reference proc applies to pro1.T3;
end sys.imp;

process process1
end process1;

process implementation process1.impl1
subcomponents
  T1: thread thr1.imp;
  T2: thread thr2.imp;
  T3: thread thr3.imp;
connections
  event port T1.Complete -> T3.Dispatch;
end process1.impl1;

processor proc
properties
  Scheduling_Protocol => DMS;
end proc;

processor implementation proc.impl1
end proc.impl1;

data data1
end data1;

data implementation data1.imp
end data1.imp;

thread thr1
properties
  Dispatch_Protocol => Periodic;
  Period => 2 Ms;
  Compute_Deadline => 2 Ms;
  Compute_Execution_Time => 1 Ms .. 1 Ms;
features
  requireData1: requires data access data1.imp;
end thr1;

thread implementation thr1.imp
end thr1.imp;

thread thr2
properties
  Dispatch_Protocol => Periodic;
  Period => 3 Ms;
  Compute_Deadline => 3 Ms;
  Compute_Execution_Time => 2 Ms .. 2 Ms;
  Dispatch_Offset => 1 Ms;
features
  requireData2: requires data access data1.imp;
end thr2;

thread implementation thr2.imp
end thr2.imp;

thread thr3
properties
  Dispatch_Protocol => Aperiodic;
  Compute_Deadline => 1 Ms;
  Compute_Execution_Time => 1 Ms .. 1 Ms;
end thr3;

thread implementation thr3.imp
end thr3.imp;
```

FIG. 1 – Exemple de modèle en AADL

```
system sys is
        behavior is
                    tr t1end -> t3begin
                    lb sys.T1.act1 t1end
                    lb sys.T3.released t3begin
        res proc is preemptable
        res data is not preemptable
        policy DM is min D
        task T1 is
                action act1 in [1,1] with alloc1
                period [2,2]
                deadline 2
                policy DM
        end
        task T2 is
                action act1 in [2,2] with alloc1
                period [3,3]
                deadline 3
                offset 1
                policy DM
        end
        task T3 is
                action act1 in [1,1] with alloc2
                deadline 1
                policy DM
        end
        allocation alloc1 is
                resources proc, data
                tasks T1, T2
        allocation alloc2 is
                resources proc
                tasks T3
end
```

FIG. 2 – Exemple de modèle en Pola

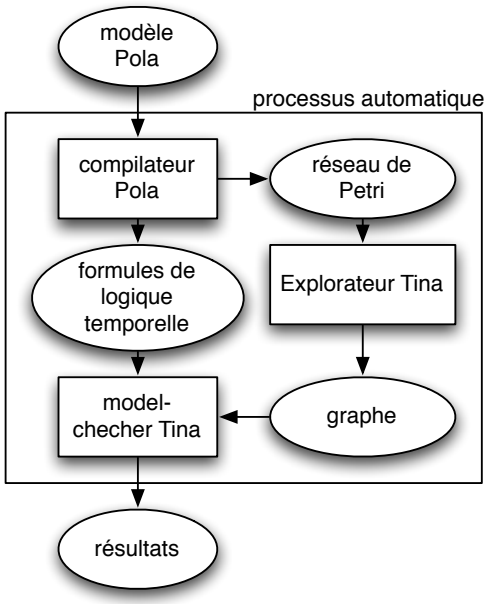

FIG. 3 – Processus de vérification avec Pola

annexe avec des propriétés spécifiques pour procéder à l'analyse d'ordonnançabilité. Certaines de ses propriétés font maintenant partie du standard AADL depuis la version 2.0. Les données du modèle AADL sont extraites pour être restructurées sous la forme d'un modèle Cheddar et ensuite procéder à une analyse temporelle pour le calcul des pires temps de réponse ou l'occupation des buffers sur les *queuing ports*. L'inconvénient principal de cet outil réside dans l'impossibilité de modéliser des comportements complexes pour une architecture donnée et la limite des modèles utilisables pour l'analyse d'ordonnançabilité.

Des travaux similaires ont été menés autour de l'outil MAST [11]. Les modèles analysables par cet outil étant plus expressifs que ceux de Cheddar, il est plus adapté pour les calculs d'ordonnançabilité.

D'autres travaux abordent la vérification comportementale des modèles AADL par des méthodes formelles. La démarche choisie se base sur les travaux sur l'ingénierie des modèles pour transformer un modèle AADL dans un langage pivot plus adapté à la vérification formelle. Par exemple, le langage Fiacre [4] est utilisé comme langage pivot vers des outils dédiés à l'analyse formelle comme Tina [7] ou CADP [9]. Les transformations actuellement définies se focalisent peu sur les aspects temporels et les analyses menées ne sont pas capables de prendre en considéraion le comportement temporel avec un ordonnancement. Une problématique similaire a été traitée avec le langage IF [1] et se heurte aux mêmes limites.

3 Le langage Pola

Pola [17] est un langage spécifique au domaine des systèmes de tâches temps réel ordonnancée. Son expressivité est limitée au domaine des systèmes temps réel, ainsi que par ses possibilités en terme de vérification, c'est-à-dire que toute notion ne s'intégrant pas dans une approche de vérification par *model checking* en est exclue. Cette limitation a cependant un intérêt important : tout modèle du langage est garanti vérifiable, dans les limites des possibilités du *model checking*.

L'aspect spécifique de Pola à un domaine permet la spécification d'un système de manière intelligible pour un utilisateur expert en temps réel, tout en cachant les détails de l'analyse, qui, le plus souvent ne font pas partie de son domaine d'expertise, à l'aide d'une chaîne de traduction/vérification automatique : un modèle Pola est traduit dans un langage adapté au *model checking*, puis la vérification proprement dite est effectuée à partir de cette traduction. La figure 3 présente l'ensemble de la chaîne de vérification qui peut être mise en oeuvre à partir d'une description d'une architecture temps réel faite en Pola.

Les réseaux de Petri servent de langage en entrée du *model-checking*, étendus par une notion de temps avec les réseaux de Petri temporels [14], une notion de priorité [6], ou plus généralement une notion de contraintes sur les quantités temporelles du réseau avec les relations d'inhibitions et de permissions de [2], ainsi que l'extension des chronomètres [5] pour modéliser la préemption.

Un ensemble de formules de logique temporelle est également fourni (pour l'instant prédéfini) pour spécifier les contraintes à vérifier, telles que « les tâches doivent respecter leurs échéances ».

Le langage Pola est composé de deux parties distinctes : une partie déclarative et une partie comportementale. La partie déclarative du langage permet de modéliser les ressources d'exécution et les tâches d'un système temps réel. Une tâche (**task**) est un flot d'exécution séquentiel qui peut être décomposé en actions (**action**) pour représenter son comportement interne. Les caractéristiques temporelles associée à une tâche sont sa période (**period**), son échéance (**deadline**) et son temps d'exécution (intervalle associé aux actions).

Une tâche est aussi attachée à une politique d'ordonnancement qui contrôle son exécution. Se basant sur les travaux de J. Migge [15], une politique d'ordonnancement est définie par une fonction d'ordre (« min » pour un ordre croissant ou « max » pour un ordre décroissant) entre

les tâches dont le critère est évalué par une combinaison linéaire des caractéristiques des tâches, par exemple « min d » signifie qu'à un instant donné la tâche devant être exécutée est celle ayant la plus petite échéance courante. Les politiques usuelles telles que *Rate Monotonic* ou *Deadline Monotonic* sont prises en compte, ainsi que toute politique *ad-hoc* dépendant d'un niveau de priorité fixé par l'utilisateur – lequel niveau est attribué à chacune des tâches. Les politiques dynamiques telles que *Earliest Deadline First* (EDF) ou *First In First Out* (FIFO) peuvent être prise en considération en utilisant des caractéristiques dynamiques des tâches, ce qui n'est pour le moment pas possible dans l'implémentation courante de l'outil de vérification.

L'ordonnancement sous Pola est défini à l'aide de deux éléments : la politique d'ordonnancement, liée aux tâches comme expliqué ci-dessus, et l'allocation (**allocation**) définissant les ressources dont les tâches ont besoin pour s'exécuter. Une tâche ne peut s'exécuter que si toutes les ressources nécessaires sont libres (ou récupérables en préemptant les tâches qui les possèdent) et si elle est la plus prioritaire parmi les tâches actives pouvant s'exécuter sur cet ensemble de ressources (en admettant que l'ensemble de ressources ne permet d'exécuter qu'une tâche à la fois, sinon c'est le sous-ensemble de tâches le plus prioritaire qui est exécuté, selon les ressources disponibles).

La partie comportementale permet d'ajouter des comportements non génériques au système. Bien que s'éloignant de la démarche des langages spécifiques, l'utilisateur peut décrire les comportements à l'aide d'un réseau de Petri dans une partie dédiée (introduite par **behavior**).

Les parties déclaratives et comportementales de Pola sont liées entre elles à l'aide d'*accesseurs* de deux types : ceux d'*état* et ceux d'*action*. Ils se traduisent, après transformation de Pola, par des étiquettes qui servent à synchroniser les réseaux de Petri issus des parties déclaratives et comportementales. Un accesseur d'état permet d'accéder à l'état d'une variable de Pola. Par exemple, l'état activée d'une tâche est accessible grâce à l'accesseur *<sys>.<task>.released*, avec *<sys>* le nom du système auquel appartient la tâche *<task>*. Un accesseur d'action rend accessible les actions qui se produisent dans un modèle Pola, comme par exemple, l'allocation d'une ressource à une tâche par l'ordonnanceur, ou la fin d'une action.

3.1 Exemple d'un système en Pola

La figure 2 présente un système modélisé en Pola avec deux ressources (lignes 6 et 7) et trois tâches (lignes 9–14, 15–21 et 22–26). Chaque tâche est composée d'une seule action qui porte sa durée d'exécution et la manière dont les ressources lui sont allouées (lignes 10, 16 et 23).

La ressource `data` est uniquement partagée par les tâches `T1` et `T2`, alors que la ressource `proc` l'est par toutes, d'où les allocations `alloc1` et `alloc2`.

Les deux tâches `T1` et `T2` sont périodiques (lignes 11 et 17), tandis que `T3` est apériodique. L'activation de `T3` est spécifiée dans la partie *behavior* (lignes 2–5) et se produit sur la terminaison de `T1`.

La politique d'ordonnancement est spécifiée ligne 8 et donne la priorité à la tâche avec l'échéance la plus petite.

Nous verrons dans la suite que les exemples présentés dans la figure 1 et 2 sont équivalents au niveau comportemental.

4 Motivations du travail

AADL est un langage très riche de part son expressivité. Il permet de modéliser un grand nombre d'applications avec des comportements variés et complexes. En faire une vérification exhaustive au niveau comportementale est un processus difficile.

Dans ce domaine, deux approches existent : la première consiste à n'étudier qu'un point particulier du comportement du système, par exemple l'ordonnancement ou l'occupation des *buffers* de communication. Pour cela, seules les informations pertinentes pour ce problème sont extraites du modèle AADL pour ensuite être exploitées par un outils dédié à l'analyse. C'est par exemple la démarche choisie pour faire de la vérification de modèle AADL avec les outils Cheddar [18] ou MAST [11]. L'inconvénient d'une telle approche est la limite imposée par l'expressivité des outils de vérification, par exemple l'analyse d'ordonnançabilité par des techniques analytiques est restreinte à des modèles simples de tâches (périodiques sans synchronisation complexe), ce qui contraint généralement les modèles AADL à certaines classes d'applications pour pouvoir les vérifier.

La seconde approche, plus générique, consiste à transformer un modèle AADL vers un modèle formel à la sémantique bien définie, comme dans l'approche utilisant Fiacre [3] ou IF [1]. Cette méthode est généralement exhaustive du point de vue comportemental et permet de modéliser un grand nombre d'applications, mais est rapidement confrontée à la complexité des techniques de vérification et leur explosion combinatoire.

Le choix fait dans cet article est de suivre la méthodologie basée sur les modèles formels en exploitant la spécialisation du langage Pola pour l'analyse comportementale. Cela permet de modéliser efficacement un large ensemble de comportements, en particulier au niveau de l'ordonnancement, et d'en faire l'analyse. Par rapport à la démarche basée sur des outils dédiés, cette méthode n'est pas contrainte à des modèles standards pour l'analyse, mais permet intrinsèquement de composer des comportements complexes, comme des ordonnanceurs hiérarchiques.

Pour faciliter le travail, nous nous plaçons dans le cadre des processus issus de l'ingénierie des modèles pour transformer un modèle AADL vers Pola. Cependant, la principale difficulté est de garantir que cette transformation permet d'exprimer le même comportement dans un modèle AADL que dans celui en Pola. Ce point est le sujet de la partie suivante.

5 Transformation AADL vers Pola

Une chaîne de vérification basée sur des transformation de modèles n'est possible et pertinente que si la transformation garantit que le comportement exprimé par le modèle d'origine est identique à celui de la cible. Pour cela, il est nécessaire d'étudier les sémantiques de chaque langage et s'il est possible de traduire chaque élément d'un langage à l'autre. Cette partie met en évidence les similarités entre AADL et Pola au niveau comportemental ainsi que les points bloquants pour définir une transformation entre ces deux modèles.

5.1 Les principaux composants AADL et leur pendant en Pola

Le comportement d'une application est principalement capturée en AADL par les *threads*, qui se traduisent en Pola par la notion de tâche (**task**). La partie 5.2 présente les relations et les similitudes entre ces deux concepts. Le comportement temporel des *threads* est aussi lié à l'ordonnancement spécifié par les propriétés `Scheduling_Protocol` et `Preemptive_Scheduler` des composants *processor*. Au niveau Pola, cela va se traduire par une politique d'ordonnancement (**policy**) et des allocations (**alloc**) (voir partie 5.3).

Un *subprogam* en AADL décrit une partie de code qui peut être appelée par des *threads*. Il a donc une influence sur l'exécution et le comportement de ces derniers. Cette notion se traduit en Pola sous la forme d'actions (**action**). Pour décrire les interactions entre les *subprograms* et les *threads*, il est nécessaire de disposer d'un modèle comportemental des *threads*, ce qui n'existe pas dans le langage standard AADL. L'annexe comportementale [8] comble ce manque, mais,

pour le travail présent, nous limitons l'étude aux comportements spécifiés dans le standard, c'est pourquoi nous n'évoquerons plus les *subprogram* par la suite.

Les données partagées entre les *thread* ont aussi une influence sur le comportement des applications. En AADL, les données sont modélisées à l'aide du composant *data* et la propriété associée `Concurrency_Control_Protocol` qui spécifie leur politique d'accès. En Pola, cela se traduit par une ressource (**res**). Les politiques d'accès nécessite quant à elle l'écriture de patterns dédiés dans la partie comportementale de Pola (**behavior**). De la même manière que pour les *subprogram*, n'ayant pas accès au comportement d'exécution d'un *thread*, nous supposerons que les ressources sont prises par un *thread* pendant toute son exécution. La partie 5.3 aborde ce point.

Le tableau 1 présente une synthèse des différentes transformations de AADL vers Pola.

System	
AADL	Pola
system *name ...* **end** *name ;*	**System** *name* **is**... **end**
actual_ processor_ binding => ... applies to ...	spéciale
Processor	
AADL	Pola
processor implementation *name ...* **end** *name ;*	**res** *name* **is preemptable**
Preemptive_ Scheduler => false	**res** *...* **is not preemptable**
Scheduling_ Protocol => DMS	**policy** *DM* **is** *min D* + **task policy** *DM*
Scheduling_ Protocol => EDF	**policy** *EDF* **is** *min d* + **task policy** *EDF*
Thread	
AADL	Pola
thread implementation `name ... end name ;`	**task** *name* **is**... **end**
`Dispatch_Protocol => periodic + Period => X`	**period** *[X,X]*
Dispatch_ Protocol => sporadic + Period => X	**period** *[X,w]*
Dispatch_ Protocol => aperiodic	**behavior** avec réseau de Petri spécifique
Compute_ Execution_ Time => X ..Y	**action** *...* **in** *[X,Y]*
Compute_ Deadline => X	**deadline** *X*
Priority => X	**level** *X*
Data	
AADL	Pola
data *name ...* **end** *name ;*	**res** *name* **is not preemptable**
`Concurrency_Control_Protocol`	**behavior** avec réseau de Petri spécifique

TAB. 1 – Synthèse des transformations de AADL vers Pola

5.2 Modélisation du contrôle d'exécution

5.2.1 Comparaison entre les états d'un thread et d'une tâche

En AADL, le contrôle d'exécution d'une application est modélisé par les *threads*. Un *thread* exécute une séquence de code quand il est activé et élu par l'ordonnanceur. Les différents états qu'il peut prendre — les états d'intialisation, d'erreur et de changement de mode ne sont pas considérés ici — sont (voir figure 4) :

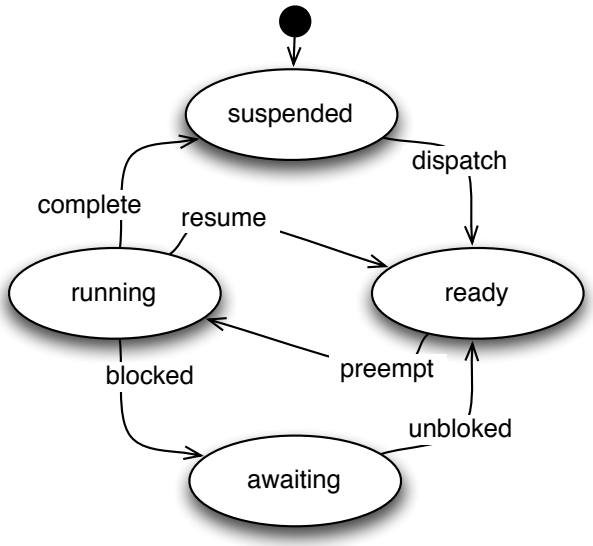

FIG. 4 – Les états d'un *thread* AADl

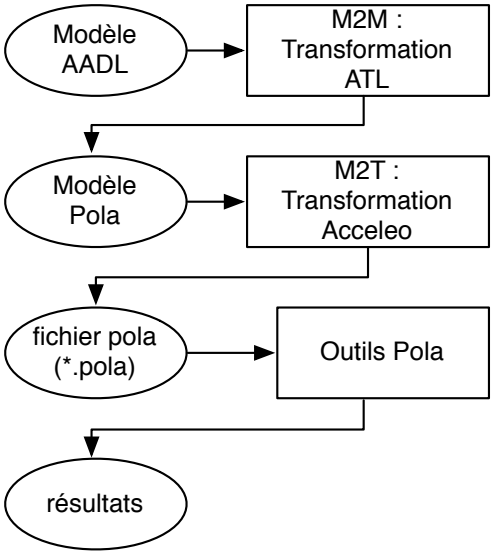

FIG. 5 – Processus de vérification d'un modèle AADL avec Pola

- *suspended* : Une fois initialisé, le *thread* attend d'être activé pour commencer son exécution. Il retourne dans cet état à la fin de chacune de ses exécutions.
- *ready* : Une fois activé, le *thread* attend d'être choisi par l'ordonnanceur pour s'exécuter. Cet état est atteint plusieurs fois pendant une exécution s'il est préempté ou bloqué.
- *running* : Le *thread* exécute son code. Il reste dans cet état jusqu'à avoir terminé son exécution ou suite à une préemption, ou l'attente d'une ressource partagée.
- *awaiting* : Le *thread* attend la libération d'une ressource — ou le résultat d'un appel à un sous-programme, mais nous ne considérons pas ce cas.

Dans Pola, les éléments qui modélisent l'exécution de l'application sont les tâches (**task**) et les actions (**action**), ces dernières offrant une modélisation plus fine du comportement. Les états d'une tâche sont identiques à ceux d'un *thread*, mais ne sont pas explicitement représentés dans la modélisation Pola. Ils sont uniquement accesibles et identifiés à l'aide d'accesseurs (voir partie 3) utilisés comme propositions atomiques :

- *suspended* correspond à $\neg released$, avec *released* l'accesseur d'état dénotant le caractère activée d'une tâche (c'est-à-dire qu'une nouvelle instance de la tâche a été crée qui peut soit être en attente de son exécution, soit être en train de s'exécuter).
- *ready* correspond à $released \wedge (\exists r \in RES(task))(\neg task.r)$, avec $task.r$ est l'accesseur d'état dénotant la possession de le ressource r par $task$, et $RES(task)$ est l'ensemble des ressources nécessaire à l'exécution de *task*.
- *running* correspond à l'état $released \wedge (\forall r \in RES(task))(task.r)$. Une tâche *task* s'exécute quand elle est activée et que toutes les ressources nécessaires à son exécution sont libres.
- *awaiting* correspond à l'état *ready* et $(\forall a \in ALLOC(task))(\neg a.active)$, où $a.task.active$ est l'accesseur d'état notant l'activité de l'allocation a de la tâche *task*, et *ALLOC(task)* est l'ensemble des allocations permettant d'allouer les ressources necessaires à l'exécution de *task*. Dans cet état (comme l'état *ready*), la tâche est donc activée, mais ne possède pas toutes les ressources nécessaire à son exécution et (contrairement à *ready*) aucune allocation ne peut les lui donner car toutes sont désactivées.

5.2.2 Comparaison entre les transitions d'un thread et d'une tâche

Les transitions entre les états d'un *thread* (voir figure 4) sont :

- *dispatch* : le *thread* est activé suite à l'arrivée d'un événement. Celui-ci peut être implicitement défini avec des patrons de type périodique (propriété `Dispatch_Protocol`) ou bien être directement connecté au port *Dispatch*. Il est ainsi possible de spécifier les événements déclencheurs du réveil en les connectant directement à ce port. La ligne 23 de la figure 1 montre une telle utilisation du port *Dispatch*.
- *complete* : le *thread* termine son exécution. Lors de la terminaison d'un *thread*, un événement est produit sur le port `Complete`. Dans l'exemple de la figure 1, la terminaison de `thr2` est utilisée pour provoquer le réveil de `thr3` (ligne 23).
- *preempt* et *resume* : le *thread* est préempté ou reprend son exécution. Ces transitions sont contrôlées par l'ordonnanceur (voir la partie suivante).
- *blocked* et *unblocked* : le *thread* est bloqué suite à la demande d'accès à une ressource indisponible. Il est débloqué lorsque la ressource devient libre. Dans cet article, la prise de ressource partagée est toujours considérée comme survenant au début de l'exécution du *thread* et la libération à sa fin.

Au niveau Pola, les transitions associées aux états d'une tâche sont identiques à celles en AADL et peuvent être contrôlées soit directement à partir de la partie comportementale, à l'aide des accesseurs, soit de manière implicite. Par exemple, l'événement *task.period* provoque une activation périodique d'une tâche sans qu'il y ait besoin de le faire apparaître explicitement, alors que dans l'exemple de la figure 2 les accesseurs *sys.T1.act1* et *sys.T3.release* sont utilisées dans la partie comportementale pour provoquer l'activation de la tâche *T3* à la terminaison de la tâche *T1*.

5.3 Propriétés temporelles

Outre les informations architecturales apportées par les différents composants — *threads*, *process*, *processor*, etc. — les propriétés normatives de l'annexe A du standard apportent aussi des informations au niveau temporel et de l'ordonnancement. Chaque propriété d'un composant permet d'enrichir sa description et doit aussi être traduit en Pola.

Les propriétés du composant *system* La propriété *Actual_ Processor_ Binding* spécifie l'allocation des *process*, des *threads* et des *data* sur les ressources de traitement. Dans le cas des *process*, tous les *thread* qui le composent, sont alors liés au même *processor*.

Si plusieurs *processors* sont disponibles et qu'aucune spécification n'est faite sur l'allocation d'un *thread*, alors celui-ci est considéré comme pouvant s'exécuter sur n'importe lequel suite à une décision d'un ordonnanceur globale, c'est-à-dire que l'ordonnanceur choisit globalement sur quelle ressource il alloue l'exécution d'un *thread*. Le standard AADL n'est pas suffisamment précis concernant la possibilité de migration des *thread* pendant leur exécution, c'est-à-dire la possibilité de changer de processeur suite à un ré-ordonnancement.

En Pola, l'allocation est faite à l'aide du mot **alloc** suivi d'une combinaison de tâches et de ressources.

Les propriétés du composant *processor* La propriété `Scheduling_Protocol` spécifie par une chaîne de caractères le nom de la politique d'ordonnancement qui gère le partage de la ressource *processor* entre les *threads* qui lui sont liés.

Dans l'annexe A du standard, une liste de politiques est fournie en exemple, comme `RMS` pour la politique *Rate Monotonic* ou `EDF` pour la politique *Earliest Deadline First*. Cepen-

dant, les politique d'ordonnancement ne sont pas formalisées ce qui rend impossible une traduction automatique pour la vérification. De plus, les exemples fournis pour illustrer l'énumération `Supported_Scheduling_Protocols` p.276 de [12] sont ambiguës et contradictoires.

Au niveau Pola, cela se traduit directement par l'ajout d'une politique d'ordonnancement avec le mot **policy**, décrite par une combinaison linéaire des caractéristiques des tâches. Par exemple, RMS est traduit $\min P$, qui signifie que la tâche la plus prioritaire est celle dont la période est la plus petite. Au niveau Pola, il est possible d'exprimer la majorité des politiques d'ordonnancement classiques – FIFO, RR, RMS, EDF, FP, etc. –, cependant celles qui sont dynamiques, c'est-à-dire dont les paramètres d'évaluation de la priorité changent au cours du temps, peuvent être décrite, mais ne sont pas gérées dans le cadre de la vérification de modèle.

La propriété `Preemptive_Scheduler` définit si l'ordonnanceur est préemptif ou non, ce qui se traduit en Pola par la spécification **preemptable** ou **not preemptable** sur la ressource qui modélise le *processor*.

Les propriétés `Process_Swap_Execution_Time` et `Thread_Swap_Execution_Time` qui définissent le temps nécessaire à un changement de contexte, soit au niveau *process*, soit au niveau *thread*, ne sont pas pris en considération dans la traduction. En effet, les traduire consisterait à intercaler un délai entre le choix de la prochaine tâche et le démarrage effectif de la tâche. Le manque de précision sémantique sur les possibilités de perturbations au cours de ce délai nous ont conduit à ne pas considérer ce cas dans l'immédiat.

Au niveau du composant *thread* La propriété `Dispatch_Protocol` spécifie le comportement sur le réveil d'un *thread*. Elle peut prendre différentes valeurs :

- `Periodic` : Le *thread* est activé périodiquement. La période est spécifiée par la valeur de la propriété `Period`. Cette propriété est exprimée en Pola par la caractéristique **period** [`Period`,`Period`] associée à une tâche.
- `Sporadic` : Deux activations successives du *thread* sont toujours séparées par une quantité de temps minimale. Cette quantité est donnée par la valeur de la propriété `Period`. Cette propriété est traduite en Pola par **period** [`Period`,w[.
- `Aperiodic` : Le *thread* est activé par un événement externe qui est connecté au port `Dispatch`. En Pola, une tâche est apériodique si **period** [0,w[, mais la plupart du temps il est préférable ne pas spécifier de période et lui associer un comportement spécifique décrit en réseau de Petri dans la partie **behavior**. Ne rien spécifier comme période ou en déclarer une, décrit des comportements différents : dans le premier cas, la tâche ne pourra pas se réveiller spontanément (un évènement extérieur est donc requis) ; tandis que dans le second, la tâche peut être réveillée n'importe quand.

Pour compléter les propriétés temporelles sur l'activation d'un *thread*, `First_Dispatch_Time` donne la date de la première requête d'activation ce qui se traduit directement en Pola par un **offset [X,X]** avec `X` la valeur du décalage.

Les valeurs `Timed`, `Hybrid` et `Background` de la propriété `Dispatch_Protocol` n'ont pas été abordée dans cette étude.

La propriété `Compute_Execution_Time` est l'intervalle de temps pendant lequel le *thread* est dans l'état *compute*. Cela se traduit au niveau des tâches par une unique action associée à un intervalle temporel **action ... in** `[X,Y]`. De même, `Compute_Deadline` est la durée maximale pouvant séparer le *dispatch* d'un *thread* et sa terminaison, ce qui se traduit directement par **deadline** en Pola.

Pour terminer, la propriété `Priority` qui permet de définir des politiques d'ordonnancement à priorité fixes quelconques se traduit en Pola par l'ajout dans une tâche de **level** `X`, où `X` est la valeur entière définissant le niveau de priorité.

Au niveau du composant *data* Un composant *data* peut être accessible par plusieurs *threads* et nécessite donc la spécification d'une politique d'accès. En AADL, ces protocoles sont spécifiés à l'aide de la propriété `Concurrency_Control_Protocols` qui peut prendre diverses valeurs : `Semaphore`, `Interrupt_Masking`, `Priority_Ceiling`...

En Pola, la majorité de ces protocoles doivent être spécifiés de manière ad-hoc à l'aide d'un comportement (**behavior**). Pour l'instant, ce travail n'est pas fait et est conséquent, en particulier pour les protocoles dynamiques comme le protocole à priorité plafond.

Dans un premier temps, nous avons exprimé en Pola les données sous la forme d'une ressource non préemptible qui est associée à toutes les allocations des tâches qui ont accès à cette ressource. L'utilisation d'une ressource par un *thread* est spécifié en AADL à l'aide d'un `features` de type **requires data access** sur une donnée.

6 Implémentation de la transformation

Du point de vue de la mise en œuvre, nous nous appuyons sur les méthodes de l'ingénierie dirigée par les modèles (IDM) pour réaliser les transformation de AADL vers Pola. Pour cela, nous avons utilisé le langage de transformation de modèle ATL (ATLAS Transformation Language) [13] qui permet de spécifier les règles pour produire un modèle cible à partir d'un modèle source. Ce langage est inspiré du standard QVT de l'OMG et est disponible en tant que module dans le projet Eclipse. ATL introduit un ensemble de concepts qui rend possible la description d'une transformation de modèle en s'appuyant sur les technologies liées aux métamétamodèles MOF (Meta Object Facilities) de l'OMG ou Ecore [20] du kit de composants logiciels EMF.

Le métamodèle Ecore de AADL utilisé est celui proposé dans l'atelier TOPCASED [21] en conformité avec les spécifications du SAE. Pour le langage Pola, un métamodèle ad-hoc a été produit en notation Ecore. Ce métamodèle comprend la partie propre au langage Pola ainsi que la partie permettant de décrire le comportement en réseau de Petri .

De plus, les modèles Pola devant être réécrit sous une forme textuelle pour servir d'entrée à la chaîne de traitements Pola, nous avons employé Acceleo [16] de la société Obeo pour faire du *model-to-text* (M2T). Cet outil permet de produire simplement du code à partir de modèles. Plusieurs générateurs sont déjà disponibles pour divers langages ainsi que différents patrons de génération pouvant être édités sous Eclipse. Son intégration directe sous TOPCASED, nous a permis de développer une chaîne complète de transformation dans cet environnement (voir figure 5).

A partir d'une description d'un système au format .pola les outils liés à Pola et ceux basés sur Tina sont vus comme une boîte noire accomplissant automatiquement la vérification. Les propriétés LTL à vérifier sont automatiquement générées à partir de la spécification AADL, par exemple, les expressions liées à la vérification du respect des échéances sont automatiquement produites.

Une première implémentation de la transformation a été réalisée sur un sous-ensemble de AADL. Cette implémentation comprend :

- La réécriture des composants de contrôle présentés dans la partie 5.2.
- La prise en compte des politiques d'ordonnancement *Rate Monotonic* et à priorité fixe.
- L'activation périodique, sporadique et apériodique sur le port *Dispatch*.
- La prise en compte des données partagées avec une politique d'accès de type sémaphore.

7 Conclusion

Cet article propose une méthode de transformation d'un modèle AADL vers Pola, un langage de vérification dédié aux systèmes temps réel critiques et ordonnancés. Après avoir montré les correspondances entre ces deux modèles et les moyens pour passer d'une description à une autre, nous avons exposé la mise en œuvre de cette transformation en suivant les principes de l'ingénierie des modèles. Le prototype actuel, bien que limité à un sous-ensemble de propriétés du standard AADL, démontre la faisabilité d'une telle approche.

Les travaux futurs s'attacheront à l'intégration complète de cet outil dans l'atelier TOPCASED et à une extension des propriétés pouvant être prise en considération par la transformation. En particulier, il est d'intégrer la description internes des *threads* à l'aide de l'annexe comportementale AADL pour pouvoir affiner l'analyse au niveau Pola. De plus, des travaux sont en cours sur la prise en charge de politiques d'ordonnancement dynamiques avec Pola, ce qui permettra, à terme, de vérifier des systèmes avec de tels ordonnancements.

Références

[1] T. Abdoul, J. Champeau, P. Dhaussy, P.-Y. Pillain, and J.-C. Roger. AADL execution semantics transformation for formal verification. In *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*, pages 263–268, 2008.

[2] T. Agerwala and M. Flynn. Comments on capabilities, limitations and "correctness" of Petri nets. In *1st annual Symposium on Computer architecture (SCA '73)*, 1973.

[3] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal-Zilio, M. Filali, and F. Vernadat. Formal verification of AADL specifications in the topcased environment. In *Ada-Europe*, pages 207–221, 2009.

[4] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre : an intermediate language for model verification in the TOPCASED environment. In *4th European Congress Embedded Real-Time Software (ERTS)*, 2008.

[5] B. Berthomieu, D. Lime, O. H. Roux, and F. Vernadat. Reachability. problems and abstract state spaces for time Petri nets with stopwatches. *Journal Discrete Event Dynamic Systems*, 2004.

[6] B. Berthomieu, F. Peres, and F. Vernadat. Bridging the gap between timed automata and bounded time petri nets. *In Lecture Notes in Computer Science*, 2006.

[7] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – construction of abstract state spaces for Petri nets and time petri nets. *International Journal of Production Research*, 42(14), 2004.

[8] R. B. Franca, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas. The AADL behaviour annex – experiments and roadmap. In *12th IEEE International Conference on Engineering Complex Computer Systems (CECCS '07)*, 2007.

[9] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006 : A toolbox for the construction and analysis of distributed processes. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2007.

[10] S. Gérard, J. Medina, and D. Petriu. MARTE : A new standard for modeling and analysis of real-time and embedded systems. In *19th Euromicro Conference On Real-Time Systems (ECRTS 07)*, 2007.

[11] M. González Harbour, J. J. Gutiérrez García, J. C. Palencia Gutiérrez, and J. M. Drake Moyano. MAST : Modeling and analysis suite for real time applications. In *13th Euromicro Conference on Real-Time Systems (ECRTS 01)*, pages 125–134, 2001.

[12] SAE International. *Architecture Aanalysis & Designe Language (AADL) Standard Version 2*, 2009.

[13] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *ACM Symposium on Applied Computing (SAC 06)*, 2006.

[14] P. M. Merlin and D. J. Farber. Recoverability of communication protocols : Implications of a theoretical study. *IEEE Transaction on Communications*, 24(9), 1976.

[15] J. Migge. *L'ordonnancement sous contraintes temps-réel : un modèle à base de trajectoires.* PhD thesis, Université de Nice, 1999.

[16] Obeo. Acceleo : générateur de code, http ://www.acceleo.org/, mars 2010.

[17] F. Peres, P.-E. Hladik, and F. Vernadat. Specification and verification of real-time system using the POLA tool. In *Third International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2009)*, 2009.

[18] F. Singhoff, J. Legrand, L. Nana, and L. Marçé. Cheddar : a flexible real time scheduling framework. In *ACM Special Interest Group on Ada (SIGAda'2004)*, 2004.

[19] SPICES. Support for Predictable Integration of mission Critical Embedded System, http ://www.spices-itea.org, mars 2010.

[20] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF : Eclipse Modeling Framework, Second Edition*, chapter Ecore Modeling Concepts. Addison-Wesley Professiona, 2008.

[21] TOPCASED. The Open-Source Toolkit for Critical Systems, http ://www.topcased.org, mars 2010.