Extension d'ABAReL par les Propriétés d'Exécution

Malika Benammar[*], Faiza Belala[**]
Kamel Barkaoui[***], Nadira Benlahrache[**]

*Département d'Informatique, Université de Batna, Algérie
m_benammar@ekit.com
**Département d'Informatique, Université Mentouri de Constantine, Algérie
belalafaiza@hotmail.com
***CEDRIC-CNAM, 292, Rue Saint-Martin, Paris 75003, France
barkaoui@cnam.fr

Résumé. Dans un travail antérieur, nous avons proposé une annexe comportementale ABAReL (AADL Behavioral Annex based on generalized Rewriting Logic), basée sur la logique de réécriture révisée, décrivant le composant "thread AADL" tout en préservant sa syntaxe initiale. Un modèle mathématique représenté par une théorie de réécriture décrivant son comportement simplifié, a été alors validé et implémenté sous l'environnement Maude. L'objectif de cet article est de raffiner l'annexe ABAReL et l'enrichir par d'autres constructions formelles de la logique de réécriture, à travers une extension de son langage Maude qui est RT-Maude (Real Time Maude). Cet enrichissement permettra de décrire les modes et les propriétés relatives au temps d'exécution théorique (la période, la politique d'expédition, etc.), déclarées dans le composant thread et leur prise en compte lors de son exécution. L'approche de formalisation proposée offre un cadre sémantique général, approprié pour raisonner sur le comportement de ces unités d'exécution concurrentes et pour pouvoir ensuite les analyser en tirant profit des outils existant autour de Maude.

1 Introduction

Face à la complexité des systèmes embarqués, souvent composés de multiples sous-systèmes ou unités fonctionnelles, il est nécessaire d'adopter la description architecturale dans leur processus de développement afin d'exprimer la structure haut niveau de tels systèmes et de raisonner sur leurs propriétés. AADL (Architecture Analysis and Design Language) est un langage de description d'architecture basé sur le langage MetaH qui décrit un système embarqué comme une collection de composants matériels et logiciels en interaction liée à une plateforme d'exécution. Ce langage se focalise sur les aspects architecturaux et permet la description des dimensions des composants et leurs connexions, mais ne traite pas directement de leur implantation comportementale, ni de la sémantique des données manipulées. Ces aspects peuvent être ajoutés au moyen d'annexes, ou en associant des descriptions externes à l'aide des propriétés.

Ces propriétés permettent d'exprimer les différentes caractéristiques des composants AADL telles que: le temps d'exécution théorique, la période d'un thread, le protocole de file d'attente utilisé pour un port d'événement/donnée, etc. Elles font partie intégrante de la syntaxe AADL, et sont plus adaptées pour décrire les caractéristiques des architectures.

Les annexes permettent d'incorporer des éléments rédigés dans une syntaxe différente de celle d'AADL. Elles permettent d'étendre la syntaxe AADL tout en utilisant des outils existants. Plusieurs annexes du noyau standard AADL ont été approuvées et publiées en Mai 2006 (Feiler et al., 2006). Citons notamment l'annexe des notations graphique AADL, l'annexe méta-modèle AADL, l'annexe du modèle d'erreur etc. Dans ce contexte, nous avons proposé, dans un travail antérieur (Benammar et al., 2008), l'annexe ABAReL (**A**ADL **B**ehavioral **A**nnex based on generalized **Re**writing **L**ogic), basée sur la logique de réécriture révisée, associant à chaque composant thread un modèle mathématique représenté par une théorie de réécriture T = (Σ, E, R). La signature (Σ, E) décrit sa structure statique, et les règles de réécriture R décrivent son comportement par un suivi de ses configurations qui englobent dans chaque étape son état local et l'état de chaque port de connexion lié à ce thread.

Nous proposons dans cet article une extension de l'annexe comportementale ABAReL**,** dans laquelle nous enrichissons la spécification des configurations d'un thread par les paramètres capturés à partir de ses propriétés d'exécution. La prise en compte des modes et des transitions de mode qui entraînent l'activation ou la désactivation des threads pour l'exécution est aussi considérée dans cette nouvelle version d'ABAReL. Le modèle mathématique déduit est alors prototypé en utilisant la version étendue RT-Maude (Ölveczky, 2007) du langage Maude à base de la logique de réécriture. L'approche de formalisation proposée offre un cadre sémantique général, approprié pour raisonner sur le comportement de ces unités d'exécution concurrentes *'les threads'* et pour pouvoir ensuite les analyser en tirant profit des outils qui existent autour du système Maude.

La logique de réécriture offre un cadre sémantique formel nécessaire pour la spécification et l'étude du comportement des systèmes concurrents ayant des états et évoluant en termes de transitions (Marti-Oliet et Meseguer, 1996). En plus, les extensions récentes développent de nouvelles bases sémantiques pour une version révisée de cette logique qui supporte plusieurs caractéristiques nouvelles dont l'expressivité a été trouvée très utile dans la pratique. La logique de réécriture bénéficie aussi de la présence de nombreux langages et environnements opérationnels, le plus connu étant Maude.

Dans la suite de l'article, nous décrivons tout d'abord les concepts fondamentaux de la logique de réécriture, en se focalisant sur la puissance expressive de son formalisme dans ces extensions récentes. Nous présentons ensuite le langage de description d'architecture AADL. La troisième section est consacrée à la présentation de l'annexe comportementale ABAReL, basée logique de réécriture révisée, à travers un exemple illustratif. Dans la section 4, nous montrons comment étendre le modèle d'ABAReL par la spécification des propriétés d'exécution d'un thread, ses modes d'exécution et leurs transitions correspondantes. Finalement, nous concluons par une synthèse du travail réalisé et des perspectives liées à sa poursuite.

2 Concepts fondamentaux

2.1 La logique de réécriture

La logique de réécriture est une logique de changement concurrent prenant en compte l'état et le calcul des systèmes concurrents. Elle a été montrée comme cadre sémantique unificateur de plusieurs systèmes et modèles concurrents (Marti-Oliet et Meseguer, 1996). Dans ce contexte, nous pouvons citer sans être exhaustifs, les systèmes de transitions étiquetés, les

réseaux de Petri, CCS, LOTOS (Meseguer, 2002). En plus, cette logique peut constituer une base formelle rigoureuse pour la description des architectures logicielles (Meseguer et Talcott, 1997 ; Jerad et al., 2008 ; Belala et al., 2008 ; Bouanaka et al., 2007).

Dans la logique de réécriture l'aspect dynamique d'un système est représenté par des théories de réécritures décrivant les transitions possibles entre les états du système concurrent. Une théorie de réécriture ***R*** est un quadruplet *(Σ, E, L, R)*, où *(Σ,E)* désigne la signature définissant la structure des états du système, *L* est un ensemble d'étiquettes et *R* est un ensemble de règles de réécritures (notées $[t] \rightarrow [t']$) modélisant les transitions possibles entre les états du système concurrent. Etant donné une théorie de réécriture, nous disons que *R* implique une formule $[t] \rightarrow [t']$ si et seulement si elle est obtenue par une application finie des règles de déduction suivantes:

1. La réflexivité : pour chaque terme $[t] \in T_{\Sigma,E}(X)$, $\overline{[t] \rightarrow [t]}$ où $T_{\Sigma,E}(X)$ est l'ensemble des Σ-termes avec variables construits sur la signature Σ et les équations *E*.

2. La congruence : pour chaque fonction $f \in \sum_n$, $n \in N$, $\dfrac{[t_1] \rightarrow [t_1'] \ldots [t_n] \rightarrow [t_n']}{[f(t_1,\ldots,t_n)] \rightarrow [f(t_1',\ldots,t_n')]}$

3. le remplacement : pour chaque règle $r : [t(x_1,\ldots,x_n)] \rightarrow [t'(x_1,\ldots,x_n)]$ dans *R* :
$$\frac{[w_1] \rightarrow [w_1'] \ldots [w_n] \rightarrow [w_n']}{[t(\overline{w} / \overline{x})] \rightarrow [t'(\overline{w'} / \overline{x})]}$$

4. la transitivité : $\dfrac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$

Les extensions récentes de la logique de réécriture développent de nouvelles bases sémantiques pour une version révisée de cette logique qui supporte plusieurs caractéristiques nouvelles dont l'expressivité a été trouvée très utile dans la pratique. La révision de l'expression de son formalisme a été proposée selon plusieurs dimensions. D'abord en choisissant la logique équationnelle d'adhésion (membership) comme logique équationnelle fondamentale. Ensuite en permettant des conditions très générales dans les règles conditionnelles de réécriture qui peuvent être de la forme :

$$(\forall X)\, r : t \rightarrow t' \; if \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \rightarrow t_l'$$

où ***r*** : est la règle étiquetée, tous les termes $(\boldsymbol{p_i}, \boldsymbol{q_i}, \boldsymbol{w_j}, \boldsymbol{s_j}, \boldsymbol{t_l}, \boldsymbol{t_l'})$ sont des *Σ*-termes, et les conditions peuvent être des réécritures, des équations d'adhésion dans *(Σ, E)*, ou n'importe quelle combinaison des deux. La troisième dimension permet de déclarer certains arguments d'opérateurs comme gelés (frozen), pour les bloquer en réécriture. Ceci mène à définir une logique de réécriture révisée comme un quadruplet ***R*** = *(Σ, E, Φ, R)* où *(Σ, E)* est une théorie équationnelle membership, *R* est un ensemble de règles de réécriture conditionnelles étiquetées, et *Φ* est une fonction assignant à chaque opérateur $f : k_1, \ldots, k_n \rightarrow k$ dans *Σ* le sous ensemble $\Phi(f) \subseteq \{1, \ldots, n\}$ de ses arguments gelés (Bruni et Meseguer, 2006).

2.2 Le langage RT-Maude

Les concepts théoriques de la logique de réécriture sont implémentés à travers le système Maude de haute performance, défini par J. Meseguer (Clavel et al., 2002). Un des aspects importants qui favorise l'utilisation du langage Maude dans la vérification des systèmes est la présence d'un ensemble d'outils tels que son model-checker LTL, son analyseur ITP,

son outil de complétion Knuth-Bendix et son analyseur de terminaison et de cohérence. De plus, Maude supporte non seulement la programmation fonctionnelle mais aussi la programmation orientée objet. Il a été donc utilisé pour la spécification, le prototypage, la vérification d'un large éventail d'applications entre autres celles qui sont à base de temps (RT-Maude) (Ölveczky, 2007).

En général, un programme écrit dans le langage déclaratif Maude représente une théorie de réécriture de la logique de réécriture, c'est-à-dire, une signature et un ensemble de règles de réécriture. Le calcul dans ce langage correspond à la déduction en logique de réécriture. Dans RT-Maude qui est une extension de Maude, une théorie de réécriture temps réel contient alors: *(1)* la spécification d'une sorte de données *Time* pour spécifier le domaine de temps discret ou dense, *(2)* la sorte *GlobalSystem* un constructeur libre {_}, dénotant que {*t*} est le système entier dans l'état *t*, *(3)* les règles de réécriture ordinaires modélisant le changement instantané et *(4)* une règle de réécriture particulière *tick,* ayant la forme *crl* [*l*] : {*t*} => {*t`*} *in time* R *if cond*, modélisant l'écoulement d'un temps R dans un système dont l'état est *t* si une condition est vérifiée.

2.3 Le langage AADL

AADL est un langage destiné à la description des systèmes embarqués temps réel. Il se distingue notamment par sa capacité à rassembler au sein d'une même notation l'ensemble des informations concernant l'organisation de l'application et son déploiement.

Pour mieux présenter les éléments architecturaux de base de ce langage, nous allons reprendre un exemple AADL modélisant un système de contrôle de navigation (figure 1) que nous avons emprunté à (Ifran, 2005). Il s'agit d'un système AADL (*system*) nommé *flight Control System*. Ce système devrait afficher les informations de navigation au pilote et indiquer l'état actuel de l'Autopilote. Il est composé de deux sous-systèmes logiciels : le système de navigation autopiloté *S-NAP* et le système d'interface *S-HCI*. Chaque sous-système est lié à un processeur séparé à savoir *NAP-Processor* et *HCI-Processor*. Les deux sous-systèmes sont connectés ensemble par un bus nommé *LAN-Bus* qui permet la transmission des données et des événements entre les composants des deux sous-systèmes. Ces liens sont de types *data*, *event et event data* et sont portées respectivement à travers les connexions *Pos-Data,*

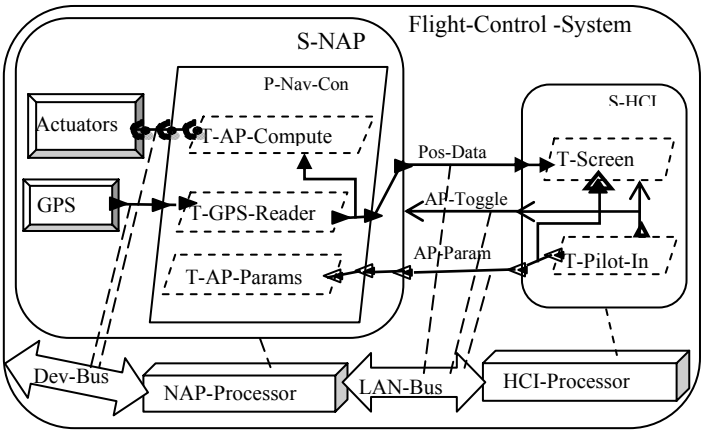

FIG. 1 – *Un système de contrôle de navigation en AADL.*

AP-Toggle et *AP-Param*. Le système *S-NAP* contient un ensemble de déclencheurs (*Aileron*, *Elevator*, *Rudder*, *Engine*), que nous avons regroupé dans un seul composant de type *device* nommé *Actuators*, un *GPS* et un processus (*process*) nommé *P-Nav-Con.*

Les composants et connexions AADL. La déclaration abstraite d'un composant AADL se compose d'un type composant et d'une ou plusieurs implémentations. La déclaration d'un type composant peut contenir des sous clauses qui représentent : des caractéristiques (*features*), des flux (*flows*), et des associations de propriétés (*property associations*). Une implémentation d'un composant spécifie sa structure interne en termes de sous composants (*subcomponents*), connexions (*connections*) entre ces sous composants, les flux (*flows*) traversant les sous composants, les modes (*modes*) pour représenter les états opérationnels, et les propriétés (*properties*). Une connexion spécifie le chemin d'un flux de contrôle et de données entre différents composants au moment de l'exécution.

Dans l'exemple de la figure 1, le processus *P-Nav-Con* contient trois threads. Les threads communiquent par l'intermédiaire des ports de données (*data port*), d'événements (*event port*) ou d'événements/données (*event data port*), et/ou des groupes de ports (*port group*). Les connexions entre les composants *Actuators* et *GPS* et le processus *P-Nav-Con* sont reliées au bus *Dev-Bus*. Une autre connexion qui a comme ultime source le thread *T-Pilot-In* dans le système *S-HCI* et comme ultime destination le thread *T-AP-Param* dans le processus *P-Nav-Con* du système *S-NAP* possède quatre autres connexions élémentaires.

Un exemple de spécification de flux sur cette même figure 1 est décrite par le flux partant du composant GPS (ou *flow source*) traversant la connexion *GPS ->P-Nav-Con* puis la connexion *P-Nav-Con -> T-GPS-Reader*, passant ensuite par le *thread T-GPS-Reader* par un *flow path* et enfin la connexion *T-GPS-Reader -> T-AP-Compute*. Le thread *T-AP-Compute* est le composant destinataire *flow sink.*

Les modes et configurations AADL. Les modes représentent les états opérationnels du logiciel, de la plateforme d'exécution, et des composants compositionnels dans le système physique modélisé. Un changement de mode peut changer l'ensemble des composants actifs et connexions. Sur la figure 2, apparaît la description de trois modes pour le composant *process P-Nav-Con*: *(1)* le mode initial noté *GPSupAPdown* qui correspond à un GPS mis en marche (positionné à *up*) et un arrêt de l'Autopilote (indiqué par *down*), *(2)* le mode

```
process P-Nav-Con
features
   GPSerror : in event port;
   APtoggle : in event port;
end P-Nav-Con;
process implementation P-Nav-Con
subcomponents
   T-GPS-Reader:  thread in modes  (GPSupAPdown, GPSupAPup);
   T-AP-Compute:  thread in mode  (GPSupAPup);
 modes
   GPSupAPdown : initial mode;
   GPSupAPup : mode;
   GPSdown : mode;
   GPSupAPdown  –[ APtoggle ] ->  GPSupAPup;
   GPSupAPdown  –[ GPSerror ] ->  GPSdown;
   GPSupAPup  –[ GPSerror ] -> GPSdown;
end P-Nav-Con;
```

FIG. 2 – *Spécification AADL du processus 'P-Nav-Con'.*

GPSupAPup pour décrire la mise en marche du *GPS* et de l'Autopilote. Ce mode est activé par l'événement *AP-Toggle* provenant du thread *T-Pilot-In* exprimant le choix du pilote, *(3)* le mode *GPSdown* où le *GPS* arrête de fonctionner suite à une erreur. Dans ce cas l'Autopilote s'arrête aussi puisqu'il ne peut pas fonctionner sans le *GPS*.

Une configuration représente un graphe de composants et de connecteurs. Les connecteurs pour AADL sont spécifiés par des flux et des modes.

Les threads. Nous nous intéressons principalement dans ce papier à ce type de composant actif. Considérons dans notre cas d'exemple le thread *T-GPS-Reader*. Sa description AADL (figure 3) montre qu'il s'agit d'un thread périodique qui fonctionne selon deux modes, le mode *GPSupAPup* et le mode *GPSupAPdown*. Ce thread lit la position courante du *GPS* et la convertit en une représentation interne pour l'envoyer au thread *T-screen* dans le système *HCI-system* si le mode courant est *GPSupAPdown*, et au thread *T-AP-Compute* si le mode courant est *GPSupAPup* (voir figure 1). Les valeurs déclarées des propriétés *Compute_Execution_Time* et *Period* changent selon le mode de fonctionnement du thread.

```
thread T-GPS-Reader
 features
     GPSPositionInput : in data port ;
     GPSPositionOutput : out data port;
   end T-GPS-Reader;
thread implementation T-GPS-Reader.PowerPC
   modes
     GPSupAPdown :  initial mode;
     GPSupAPup :  mode;
properties
   Dispatch_Protocol => Periodic;
   Compute_Execution_Time => 10ms in modes (GPSupAPup);
   Period => 20 ms in modes (GPSupAPup);
   Compute_Execution_Time => 15ms in modes (GPSupAPdown);
   Period => 30 ms in modes (GPSupAPdown);
end T-GPS-Reader.PowerPC;
```

FIG. 3 – *Exemple de spécification AADL d'un thread.*

3. Présentation d'ABAReL

ABAReL est une annexe comportementale pour le langage AADL, à base de la logique de réécriture révisée, proposée dans (Benammar et al., 2008). Elle permet d'associer au composant AADL *Thread* un support mathématique formel représenté par une théorie de réécriture $T = (\Sigma, E, R)$. La théorie équationnelle (Σ, E) décrit sa structure statique à travers la sous logique *membership equational logic*, et les règles de réécriture R décrivent son comportement simplifié. Σ constitue la signature de notre modèle, c'est-à-dire la spécification de l'ensemble de sortes, de sous sortes et de l'ensemble des opérateurs utiles pour décrire statiquement un thread générique. E représente l'ensemble des équations de notre modèle ainsi que les attributs équationnels des opérateurs. En effet, une approche générique de formalisation a été adoptée. Elle consiste à spécifier séparément chaque clause de la description globale d'un thread : caractéristiques, propriétés, type et implémentation par des modules fonctionnels Maude: *ThreadFeatures*, *Properties-Impl, ThreadType*, et *ThreadImplementation* (voir figure 4). L'aspect statique du composant *Thread* est donc spécifié formellement par la

```
fmod THREAD is extending ThreadType .
extending ThreadImplementation .
sorts Thread Threaddeclaration .
subsort threadType < Thread .
subsort threadImplementation < Thread .
op none : -> Thread [ctor] .
op __ : threadType threadImplementation -> Thread
[ctor ] .
op _in_ : Thread threadState -> Threaddeclaration
[ctor] .
op threadtype-of_ : Thread -> threadType [ctor] .
op threadImplementation-of_ : Thread -> threadIm-
plementation [ctor] .
var T : threadType . var I : threadImplementation .
eq threadtype-of none = none .
eq threadImplementation-of none = none .
eq threadtype-of T ; I = T .
eq threadImplementation-of T ; I = I .
endfm
```

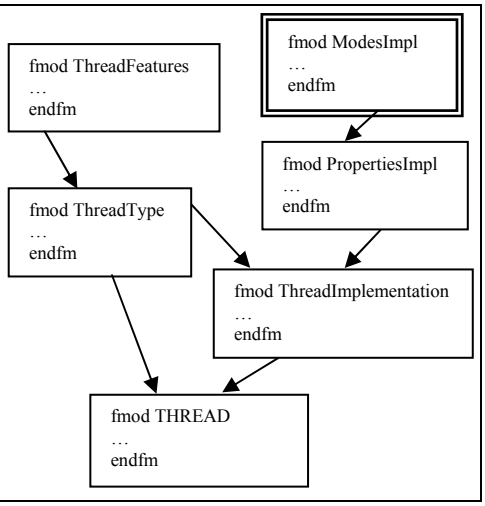

FIG. 4 – *Les modules Maude pour formaliser un thread AADL.*

théorie de réécriture équationnelle représentée par le module Maude *THREAD* de la figure 4. C'est un module générique utilisé quelque soit le thread AADL considéré. De plus, l'expressivité et la flexibilité de la logique de réécriture ont permis la déclaration très naturelle des constructions syntaxiques d'AADL en préservant sa syntaxe.

La transcription du thread *T-GPS-Reader* (figure 3), en logique de réécriture se fait alors par la déclaration d'un nouveau module fonctionnel Maude *T-GPS-Reader* (figure 5) qui

```
fmod T-GPS-reader is extending THREAD .
ops in`data`port in`event`port in`event`data`port out`data`port out`event`port out`event`data`port : -> Data-
Type [ctor] .                                   ops GPSupAPup GPSupAPdown : -> modeName [ctor] .
op GPSPositionInput : -> IPortName [ctor] .     ops GPSPositionOutput : -> OPortName [ctor] .
op T-GPS-Reader : -> threadName [ctor] .        op PowerPC : -> threadImplid [ctor] .
ops wait compute : -> threadState [ctor] .      ops receive wait-in send wait-out : -> PortState [ctor] .
ops initial`mode mode : -> modeid [ctor] .      op Periodic : -> propertyValue [ctor] .
op threadDecl1 : -> Thread [ctor] .             op APToggle : -> eventName [ctor] .
ops Dispatch-Protocol Period Compute-Execution-Time : -> propertyName [ctor] .
ops value1 value2 value3 value4 : -> Time [ctor] .
eq threadDecl1 = thread T-GPS-Reader
        features
          GPSPositionInput : in data port (wait-in) ; GPSPositionOutput : out data port (wait-out) ;
      end T-GPS-Reader (compute) ;
      thread implementation T-GPS-Reader . PowerPC
        modes
          GPSupAPup : initial mode ;      GPSupAPdown : mode ;
        properties
          Dispatch-Protocol => Periodic ;
          Period => value1 in mode (GPSupAPup) ;
          Compute-Execution-Time => value2 in mode (GPSupAPup) ;
          Period => value3 in mode (GPSupAPdown) ;
          Compute-Execution-Time => value4 in mode (GPSupAPdown) ;
      end T-GPS-Reader . PowerPC ; in compute .
endfm
```

FIG. 5 – *Le thread "T-GPS-Reader" en Maude.*

enrichit le module générique *THREAD* avec des opérations spécifiques instanciant, dans la partie équation (eq dans la figure 5) du module, ses ports spécifiques: *GPSPositionInput* et *GPSPositionOutput*, ses modes: *GPSupAPup* et *GPSupAPdown*, ses propriétés: *Dispatch-Protocol*, *Period* et *Periodic*, et enfin une implémentation possible de ce thread: *PowerPC*.

En effet, cette équation spécifie clairement et d'une manière globale la description AADL du thread *T-GPS-Reader*. Le module Maude *T-GPS-Reader* contient toutes les instanciations typiques du modèle générique *THREAD*. L'apport de cette approche de formalisation est qu'elle peut directement servir à l'extension de la description d'un composant thread AADL en ajoutant naturellement d'autres modules fonctionnels pour la spécification de clauses additionnelles tels que les sous composants, les flux, les connexions, etc. Ce mécanisme d'extension nous permet également d'enrichir la description AADL d'un thread par la

```
mod behavior-T-GPS-Reader is extending T-GPS-reader .
sort action .                                   sort ThreadBehavior .
op Execute : action [ctor] .                    op Data-receive`(_,_`) : IPortName PortState -> action .
 op Receive-complete : -> action .              op Compute-complete : -> action .
op  Initialise`(_`) : modeName -> action .      op | _,_| : Threaddeclaration action -> ThreadBehavior .
var I : ThreadImplementation .                  var T : threadType .
op Signal-send`(_,_`) : OPortName PortState -> action  [ctor] .
rl [rule1] : <thread  T-GPS-Reader
          features
            GPSPositionInput : in data port (wait-in) ;    GPSPositionOutput: out data port (wait-out) ;
          end T-GPS-Reader (wait) ; I in wait,  Data-receive (GPSPositionInput, receive )  >
=>    <thread  T-GPS-Reader
          features
            GPSPositionInput : in data port (receive) ;    GPSPositionOutput : out data port (wait-out) ;
         end T-GPS-Reader (wait) ; I in wait , Data-receive (GPSPositionInput, receive ) > .
rl [rule2] : <thread  T-GPS-Reader
          features
            GPSPositionInput : in data port (receive) ;    GPSPositionOutput : out data port (wait-out) ;
         end T-GPS-Reader (wait) ; I in wait, Receive-complete >
  =>       <thread  T-GPS-Reader
          features
            GPSPositionInput : in data port (wait-in) ;    GPSPositionOutput : out data port (wait-out) ;
          end T-GPS-Reader (compute) ; I in compute, Execute>  .
rl [rule3] : < thread  T-GPS-Reader
             features
               GPSPositionInput : in data port (wait-in) ;    GPSPositionOutput : out data port (wait-out) ;
             end T-GPS-Reader (compute) ; I in compute, Compute-complete>
  =>    <thread  T-GPS-Reader
             features
               GPSPositionInput : in data port (wait-in) ;    GPSPositionOutput : out data port (send) ;
             end T-GPS-Reader (wait) ; I in wait, Signal-send (GPSPositionOutput, send ) >
rl [rule4] : <thread  T-GPS-Reader
          features
            GPSPositionInput : in data port (wait-in) ;    GPSPositionOutput : out data port (send) ;
          end T-GPS-Reader (wait) ; I in wait,  Signal-send (GPSPositionOutput, send )>
  =>      < thread  T-GPS-Reader
          features
            GPSPositionInput : in data port (receive) ;    GPSPositionOutput : out data port (wait-out) ;
          end T-GPS-Reader (wait) ; I in wait, Data-receive (GPSPositionInput, receive )> .
endm
```

FIG. 6 – *Spécification du comportement du thread T-GPS-Reader.*

formalisation de son comportement. Cet aspect a été aussi abordé dans l'annexe ABAReL mais de façon assez simplifiée. Il s'agit tout simplement de décrire les changements de configurations visibles dans le thread par l'ensemble des règles de réécriture *R*.

Dans le cas de notre exemple, cet aspect est spécifié dans le module système Maude '*behavior-T-GPS-Reader'* (figure 6) qui enrichit le module fonctionnel '*T-GPS-Reader*' par les règles de réécriture (rule1, rule2, etc.) définissant les changements des états du thread liés à ses ports de connexion. La première règle de réécriture *(rule1)* de ce module par exemple, spécifie le changement de l'état du thread en question (*T-GPS-Reader*) après avoir effectué l'action '*Data-receive*' permettant d'exprimer la réception des données (*receive*) à travers le port d'entrée *(GPSPositionInput).* Dans la règle 2 *(rule2)*, nous exprimons l'effet de l'action '*Receive-complete*' qui termine la réception des données sur le port '*GPSPositionInput*'. A la différence avec la règle précédente, celle-ci identifie en plus des changements visibles sur ses ports et la transition de son état *wait* vers l'état *compute*.

En général dans le langage AADL, une description comportementale implantant l'algorithme régissant le fonctionnement interne d'un composant est fournie par l'utilisateur, dans un langage de programmation ou dans un autre formalisme, selon la façon dont on souhaite l'exploiter au sein de la modélisation AADL. L'apport du modèle ABAReL est l'utilisation d'un seul formalisme (logique de réécriture) pour définir les aspects structurels et dynamiques d'un thread AADL. Nous généralisons ce modèle dans la section suivante pour définir aussi les changements les états hiérarchiques ou composites.

4. Extension d'ABAReL par les propriétés d'exécution

Le standard AADL associe à chaque thread une sémantique dynamique à base d'automate (SAE, 2004) décrivant les états du thread et les conditions de transition de ces états, mais ne révèle rien sur les configurations successives d'un thread en exécution (dans l'état *Compute*). En effet, un thread peut être stoppé, inactif, ou en activité. Un thread actif peut être en attente pour une expédition, ou en exécution. Dans l'état *'Compute'*, qualifié d'hiérarchique, le thread peut avoir d'autres sous états (figure 7) où il peut être prêt pour l'exécution (*Ready*), en exécution (*Running*), ou bloqué sur l'accès d'une ressource (*Awaiting-Resource*).

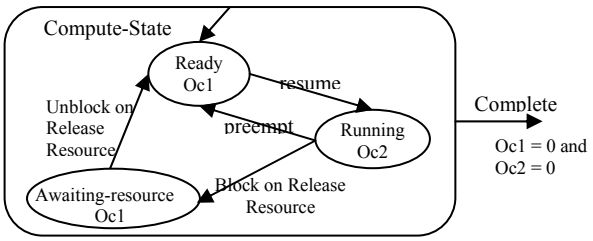

FIG. 7 – *Sous états de l'état 'Compute' du thread T-GPS-Reader.*

De plus, ce standard décrit le comportement dynamique de l'architecture d'exécution en modélisant tout simplement des modes opérationnels et des transitions de mode. Or, les transitions de mode représentent la commutation entre les configurations d'exécution d'un système. Celles-ci peuvent avoir l'effet d'activation ou désactivation des threads pour l'exécution, ou encore un changement du chemin de connexion entre les threads, et enfin des changements des caractéristiques internes des composants. Par conséquent, un thread peut être actif dans un

mode et inactif dans un autre, seul les threads actifs exécutent leurs instructions (Abdoul et Dhaussy (2006)). Cette sémantique d'exécution n'apparaît pas au niveau de la description AADL du thread qui se contente d'une clause de déclaration des différents modes possibles, suivie par une clause pour la déclaration des propriétés d'exécution (*Compute-Execution-Time* et *Period*) spécifiques à chaque mode (figure 3).

L'objectif majeur de cet article est de doter les spécifications ABAReL existantes par des constructions supplémentaires permettant la description formelle et complète des configurations d'exécution d'un thread ainsi que son état composite *Compute*. Nous exploitons la version RT-Maude destinée à la modélisation et l'analyse des applications temps réels pour définir le comportement d'un thread en considérant ses propriétés d'exécution temporelles. Nous choisissons pour notre cas d'exemple, un thread périodique qui fonctionne selon deux modes. Il s'agit du thread *T-GPS-Reader*. Nous reprenons tout d'abord le module fonctionnel *PropertiesImpl* dans ABAReL pour l'enrichir avec un autre module fonctionnel *ModesImpl* (figure 4) contenant d'autres constructions intégrant les déclarations de modes dans la version RT-Maude. L'expression des états du thread et ses transitions (figure 6) dans le module système Maude '*behavior-T-GPS-Reader* ' sont alors plus raffinées. La déduction dans la logique de réécriture contribue à l'exécution concurrente et éventuellement synchronisée de ces transitions d'états du thread AADL modélisé.

Dans une deuxième phase, nous nous sommes intéressés à la considération des états composites ou hiérarchiques d'un thread et en particulier l'état actif *'Compute'*. Une définition

```
tmod ComputeStateof T-GPS-Reader is extending behavior-T-GPS-Reader .
protecting NAT-TIME-DOMAIN-WITH-INF .
sorts Sstate Compute-Substate  clock.                    subsort clock < Time .
subsort Compute-Substate < threadState .                 subsort ThreadBehavior < System .
ops Running Ready Awaiting-resource Complete  : -> Sstate [ctor] .
op <_,_,_> : Sstate Time Time -> Compute-Substate [ctor] .   op mte : System -> Time [frozen (1)] .
op delta : System Time -> System  [frozen (1)] .         op stopped-clock : Time -> clock [ctor] .
var I : ThreadImplementation .                           var T : threadType .
vars Oc1 Oc2 : Time .                                    var M : modeName .
eq delta(|T ; I in <Sstate, Oc1 , Oc2> , Execute|, R) = |T; I in <Sstate, Oc1 - R , Oc2 - R>, Execute| .
eq delta(|T ; I in <Sstate, Oc1 , stopped-clock(Oc2)> , Execute|, R) = |T; I in <Sstate, Oc1 - R , stopped-
clock(Oc2)>, Execute| .
eq mte(|T ; I in <Running, Oc1 , Oc2> , Execute|, R) = Oc2 .
ceq mte(|T ; I in <Sstate, Oc1 , stopped-clock(Oc2)> , Execute|, R) = Oc1  if  Sstate =/= Running .
rl [Initialise] : |T ; I in compute , Initialise(M) |  => |T ; I in <Ready , access-value(Compute-Execution-Time , M ,
properties-of I), access-value( Period, M , properties-of I)> , Execute| .
rl [finish] : |T ; I in <Complete, 0, 0>, Execute| => |T ; I in wait, Signal-send(GPSPositionOutput, send )| .
crl [tick] : {|T;I in <Sstate , Oc1 , Oc2> , Execute|} => {delta (|T ; I in <Sstate, Oc1 , Oc2> , Execute|, R)} in Time R
if R <= mte(|T ; I in <Sstate, Oc1 , Oc2> , Execute|) [nonexec] .
crl [resume] : |T ; I in <Ready, Oc1,Oc2>, Execute|
  =>      |T ; I in <Running, Oc1,Oc2 >, Execute| if  Oc2  >  0 .
crl [preempt] : |T ; I in <Running, Oc1, Oc2>, Execute|   =>      |T ; I in <Ready, Oc1, Oc2>, Execute| if  Oc1 > 0 .
crl [block-on-Release-Resource] : |T ; I in <Running, Oc1, Oc2>, Execute|
  =>      |T ; I in <Awaiting-resource, Oc1, Oc2>, Execute| if  Oc1 > 0 .
crl [Unblock-on-Release-Resource] : |T ; I in <Awaiting-resource, Oc1, Oc2>, Execute|
   =>      |T ; I in <Ready, Oc1, Oc2>, Execute| if  Oc1 > 0 .
 crl [complete] : |T ; I in <Sstate, Oc1, Oc2>, Execute|
    =>      |T ; I in <Complete, Oc1, Oc2>, Compute-complete| if Oc1 == 0  /\ Oc2 == 0 .
endtm
```

FIG. 8– *Formalisation de l'exécution d'un thread.*

précise de sa sémantique est alors donnée via un module RT-Maude '*ComputeStateof T-GPS-Reader*' pour l'exemple illustratif dans lequel nous avons définit les sous états et leurs transitions en tenant compte des propriétés d'exécution (*Compute-Execution-Time* et *Period* pour le thread *T-GPS-Reader*). Dans ce module, la première règle de réécriture fait passer l'implémentation du thread de l'état *'Compute'* au sous état *'Ready*' avec une initialisation des horloges par les valeurs capturées à partir des propriétés d'exécution en tenant compte du mode de fonctionnement actif. La règle de réécriture *'finish'* remet le thread du sous état *'Complete*' à l'état *'wait'* après écoulement de la période et du temps d'exécution. Les règles de réécritures conditionnelles *resume, preempt, block-on-Release-Resource, Unblock-on-Release-Resource* et *complete* représentent les transitions entre les sous états de l'état *'Compute*' (figure 7) avec une mise en garde de la période et du temps d'exécution par les horloges *Oc1* et *Oc2*. Cette mise en garde est modélisée dans RT-Maude par les opérations *delta* et *mte*. L'opération *delta* calcule l'effet du passage d'un temps R sur la configuration du thread. L'opération *mte* évalue le temps maximal pouvant s'écouler avant qu'une action importante s'exécute (ici c'est le minimun des valeurs des deux horloges). La règle *tick* fait usage des deux opérations '*delta*' et '*mte*' pour interpréter l'écoulement du temps.

5. Travaux connexes

Le standard AADL version 1.0 publié en Nov. 2004 définit la syntaxe textuelle et la sémantique du noyau de ce langage de description architecturale extensible. Dés lors, le développement des extensions AADL reste un point ouvert de recherche, permettant d'introduire des annexes et des propriétés additionnelles qui nécessitent de nouvelles approches d'analyse. Plusieurs annexes du noyau AADL, utilisant différents formalismes, ont été approuvées et publiées en Mai 2006 (Feiler et al., 2006). Parmi lesquelles nous pouvons citer : l'annexe des notations graphiques AADL utilisée pour modéliser graphiquement des architectures avec AADL, l'annexe méta-modèle AADL et format d'échange XML/XMI pour la représentation abstraite et le format d'échange des modèles AADL, l'annexe d'interface de programmation d'application définissant le mapping d'AADL aux langages de programmation Ada95 et C/C++, le profil UML pour AADL, qui est devenu ensuite une base pour le projet MARTE (Modeling and Analysis of Real-Time Embedded systems) (OMG, 2008) et aussi l'annexe du modèle d'erreur associé au noyau des composants AADL.

Dans ce contexte, deux études Européennes ASSERT (Automated proof-based System Software Engineering for Real-Time systems) et COTRE (Composants Temps REel) (Berthomieu et al., 2002) ont identifié AADL en tant que technologie prometteuse et ont donc participé à l'extension du langage AADL par le développement d'outils de modélisation textuelle et graphique et de vérification sémantique, fondés sur l'environnement OSATE (Open Source AADL Tool Environment) (Feiler, 2005) utilisant la plateforme Eclipse. Nous pouvons citer comme exemples d'outils SPIN, UPPAA, TINA, CHEDDAR, etc.

L'ensemble des travaux autour de la formalisation d'AADL se base sur le modèle des réseaux de Petri pour profiter de la batterie des outils d'analyse existante. En particulier, l'auteur de (Vernaud, 2006), génère un réseau de Petri coloré à partir de la description AADL pour étudier certaines propriétés structurelles comme le blocage des architectures. Cependant, il est difficile de prévoir le nombre d'états des réseaux produits puisqu'il dépend à la fois du nombre de threads AADL et de leurs interconnexions qui varient au moment de l'exécution.

Notre approche de formalisation à base de la logique de réécriture révisée, offre un seul cadre sémantique approprié pour spécifier et raisonner sur le comportement concurrent des

architectures logicielles AADL. La logique de réécriture constituant un cadre sémantique et logique unificateur où plusieurs modèles sémantiques (Réseaux de Petri, automates, CCS, structures d'événements, etc) ont été intégrés, offre aux concepts formalisés d'AADL un support mathématique rigoureux permettant une analyse formelle en tirant profit de tous les outils qui existent autour du système Maude. Par exemple, nous pouvons utiliser le model checker LTL de Maude, pour valider la structure de l'application vis-à-vis des flux et des propriétés d'exécution déclarées. Dans le cadre de ce travail, nous procédons au raffinement de la description formelle du thread AADL de façon générique et naturelle pour prendre en compte d'une part, la modélisation des états concurrents et hiérarchiques d'un thread et d'autre part, la déclaration de ses propriétés d'exécution ainsi que leur intervention dans les exécutions faites à travers le temps. Dans les travaux antérieurs cités, ses aspects ne peuvent pas être tous modélisés par un seul formalisme.

6. CONCLUSION

Les systèmes embarqués temps réel apparaissent souvent comme des composants de systèmes complexes dont la sûreté de fonctionnement est critique. Plusieurs approches basées composants ont été utilisées pour modéliser et analyser ce type de systèmes, en particulier le langage de description d'architecture AADL. Cependant, ce langage se focalise sur les aspects architecturaux seulement sans se préoccuper des aspects comportementaux des composants. Dans cet article, nous avons proposé une extension de l'annexe comportementale ABAReL par les propriétés d'exécution temporelles capturées à partir de la déclaration AADL d'un thread. Cette extension a permis d'enrichir le modèle mathématique d'ABAReL, basé sur la logique de réécriture révisée, par des constructions supplémentaires permettant de définir d'une part les configurations successives d'un thread périodique en exécution, et d'autre part un de ses états composites, à savoir l'état '*Compute*'. Une définition de ses sous états et leurs transitions ainsi que ses propriétés d'exécution temporelles sont alors données en exploitant la version RT-Maude. Les constructions formelles déduites sont utiles pour raisonner et analyser le comportement de l'application qui sera générée. En plus, la réalisation de cette formalisation en Maude offre une spécification exécutable, lisible et extensible, dû à la flexibilité et la puissance de ce langage.

Le standard AADL définit quatre politiques de déclenchement d'un thread (périodique, apériodique, sporadique ou background) à l'aide de la propriété *Dispach-Protocol*. Il définit aussi un grand nombre de propriétés permettant d'exprimer des contraintes spatiales et temporelles, les descriptions des implantations et d'autres caractéristiques sur les entités AADL. Les propriétés de descriptions des implantations permettent d'associer un code source aux composants, il s'agit des propriétés (*Source-Language, Source-Text, Source-Name*). Notre approche de formalisation est assez générique et facilement extensible pour considérer tous ces concepts dans nos futurs travaux. Nous pensons, dans un avenir proche, à la modélisation du dispatcher ainsi que ses propriétés.

Références

Abdoul, T., et P. Dhaussy (2006). *Transformation de modèles AADL en systèmes à automates communicants IF*. Rapport Master Recherche Informatique, Laboratoire Développement Technologies Nouvelles ENSIETA, Brest, France.

Belala, F., F. Latreche , et M. Benammar (2008). *Vers l'Intégration des Propriétés non Fonctionnelles dans le Langage SADL*. 2ième Conférence Francophone Sur les Architectures Logicielles CFP-CAL'08, RNTI, pp.91-105. Cépaduès-Editions.

Benammar, M., F. Belala, et F. Latreche (2008). *AADL Behavioral Annex Based on Generalized Rewriting Logic.* Proceeding of the IEEE International Conference on Research Challenges in Information Science, RCIS'08, pp.7-13.

Berthomieu, B., P-O. Ribet, F. Vernadat, J. L. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Filali, G. Padiou, P. Michel, P. Farail, P. Gauffilet, P. Dissaux, et J-L. Lambert (2003). *Towards the verification of real-time systems in avionics : the Cotre approach.* Electronic Notes in Theoretical Computer Science 80.

Bouanaka, C., F. Belala, et A. Choutri (2007). *On Generating Tile System for a Software Architecture : Case of a Collaborative Application Session.* In ICSOFT'2007 (the Second Conference on Software and Data Technologies), pp. 123-128.

Bruni, R., et J. Meseguer (2006). *Semantic foundations for generalized rewrite theories*. Theoretical Computer Science 360, pp. 386-414.

Clavel, M., F. Duran, S. Eker, N. Marti-Oliet, P. Lincoln, J. Meseguer, et C. Talcott (2002). *Maude 2*. Available: (http://maude.cs.uiuc.edu)

Feiler, P., B. Lewis, et S. Vestal (2006). *The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems*. Proceedings of the IEEE Conference on Computer Aided Control Systems Design, Munich, Germany.

Feiler, P. (2005). *Open Source AADL Tool Environment (OSATE)*. AADL Workshop, Paris, Oct 2005. Available: http://www.axlog.fr/R_d/aadl/workshop2005_en.html

Ifran, H. (2005). *Flight Control System: Modeling of a Hardware/Software System in AADL*. Computer Science & Networks Department, ASSERT AADL Workshop.

Jerad, C., et K. Barkaoui (2008). *On the use of Real-Time Maude for Architecture Description and Verification: A Case Study*. In BCS International Academic Conference "Visions of Computer Science".

Marti-Oliet, N., et J. Meseguer (1996). *Rewriting logic as a logical and semantic framework*. RWLW96, First International Workshop on Rewriting Logic and its Applications, Vol. 4, no.1, pp. 190-225.

Meseguer, J., et C. Talcott (1997). *Formal Foundations for Compositional Software Architectures*. Position Paper, OMG-DARPA-MCC Workshop on Compositional Software Architectures.

Meseguer, J. (2002). *Rewriting Logic Revisited*. Université Illinois de Urbana-Champaign. USA.

Ölveczky, P. C. (2007). *Real-Time Maude 2.3 Manual.* Department of Informatics, University of Oslo.

OMG, (2008). *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2 (convenience document with change bars).* OMG Document Number: ptc/2008-06-08.

SAE International Avionics Systems Division (ASD) AS-2C Subcommittee (2004). *Avionics Architecture Description Language Standard.* SAE Document AS 5506, Nov. 2004. Available: (http://www.sae.org)

Vergnaud, T. (2006). *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées.* Ph. D. Thesis, Ecole Nationale Supérieure des Télécommunications de Paris.

Summary

In a previous work, we proposed a behavioral annex ABAReL (AADL Behavioral Annex based on revised Rewriting Logic), describing the 'thread' component while preserving its initial syntax. A mathematical model represented by a rewrite theory T= (Σ, E, R), where (Σ, E) described thread static structure, and the rewrite rules R describe its simplified behavior, was then validated and implemented using Maude environment. This paper aims to refine ABARel annex and enrich it by other formal constructions of rewriting logic, through an extension of its Maude language which is RT-Maude (Real Time Maude). This extension will make it possible to describe the modes and the properties related to theoretical execution times (Period, Dispatch protocol, Compute-Execution-Time, etc.), declared in AADL thread component and their taking into account during its execution. The formalization approach proposed offers a general semantic framework, adapted to reason on the behavior of these concurrent execution units and to be able analyze them by taking advantage of tools which exist around the Maude system.