Description Architecturale et Comportementale à l'aide du langage AADLv2

SOMMAIRE

4.1 Introduction

Dans les chapitres précédents, nous avons présenté l'utilisation du langage AADL pour la modélisation, la configuration et le déploiement des composants architecturaux des systèmes TR²E critiques. Dans le chapitre 3, nous avons justifié le choix de l'utilisation du langage

AADLv2 et de ses annexes qui introduisent de nombreux éléments et améliorations permettant une description plus fine des composants architecturaux et de leur comportement.

AADL (*Architecture Analysis and Design Language*) est un langage de modélisation d'architecture qui tire ses origines du langage METAH [Vestal, 2000] développé pour les systèmes avioniques et spatiaux. Destiné lui aussi à l'avionique, la richesse d'expression et les nombreuses extensions du langage AADL ont étendu son activité au domaine des systèmes TR^2E logiciels et matériels. La version 2.0 du standard [SAE Aerospace, 2009b] a été publiée en janvier 2009 et une version 2.1 est en cours de rédaction.

Dans ce chapitre, nous proposons une introduction au langage AADLv2 et à son annexe comportementale (*Behavior Annex* [SAE Aerospace, 2009a]), dont un sous-ensemble d'éléments constitue notre épine dorsale pour la description logicielle architecturale et comportementale d'une application répartie avec une sémantique très proche de son implantation physique.

Ce chapitre s'organise de la manière suivante. La section 4.2 présente les éléments du langage cœur (composants matériels, logiciels, connecteurs, propriétés, etc). La section 4.3 est une introduction aux différents langages définis par l'annexe comportementale et présente nos contributions pour le développement et l'écriture de ce standard. La section 4.4 présente les avantages et les restrictions pour la modélisation des systèmes critiques. Enfin, la section 4.6 conclut ce chapitre.

Remarque. Une partie des illustrations de ce chapitre est issue de la description architecturale de notre cas d'étude détaillé dans le chapitre 10.

4.2 Eléments du langage cœur

AADLv2 permet de décrire les systèmes TR^2E par un assemblage de composants matériels et logiciels connectés. Les propriétés AADL complètent la modélisation du système en autorisant l'attachement d'exigences fonctionnelles (fonction de calcul...) ou non fonctionnelles (contraintes temporelles, informations de déploiement...). Les annexes étendent le pouvoir d'expression du langage en définissant des règles, des recommandations, des patrons de modélisation ou des langages différents (ou externes) pour couvrir différents aspects du système que ce soit à des fins de représentation, d'analyse, d'implantation, etc.

Intéressons-nous à l'élément majeur d'une description architecturale AADL : le composant.

4.2.1 Composants

La première version du langage [SAE Aerospace, 2004] présentait le langage AADL comme un langage classique de description d'architecture (voir la classification [Medvidovic and Taylor, 2000]) construit sur les trois concepts fondamentaux : composants, connecteurs, connexions. Le langage AADLv2 apporte différentes améliorations (un méta-modèle, représentation graphique standardisée, profils, extensions...) caractéristiques des langages de modélisation spécifique à un domaine.

AADLv2 définit des composants matériels et logiciels du système spécifiés par une interface (**component_type**) décrivant les éléments d'interface pour la connexion inter-composants et une implantation (**component_implementation**) spécifiant la structure interne du composant. Un agrégat (ou composition) de composants peut décrire un composant établissant ainsi une hiérarchie composant $\rightarrow$ sous-composants structurant la description architecturale. AADL

propose deux catégories principales de composants : les composants matériels et logiciels et des composants spécifiques pour la composition du système (**system**) ou pour les premières étapes de la modélisation (**abstract**, voir 4.2.7).

Composants matériels, logiciels et système

AADL définit les composants matériels suivants :
- le **device** représente tous les types de périphériques (clavier, souris, capteur, etc.) sous la forme d'une boite noire exprimant ses interfaces mais aucun détail sur son implantation ;
- le **processor** modélise un contrôleur, un processeur ou un système d'exploitation minimal (ordonnanceur, pilotes, etc.) pouvant accéder à des ressources matérielles (périphériques, bus, etc.) ;
- le **virtual processor** spécifie un cœur, un ordonnanceur, une machine virtuelle ou une partition permettant de représenter la structure interne d'un processeur physique ;
- le **bus** modélise les accès entre les processeurs, les périphériques et les mémoires. L'association des connexions ou d'accesseurs permet de décrire le flot de contrôle ou la stratégie de déploiement d'une application distribuée ;
- le **virtual bus** modélise la structure interne d'un bus (qualité de service, protocole, etc.) ;
- le composant **memory** modélise une mémoire physique quelconque (ROM, RAM, disque dur, etc.) pour le stockage de données.

AADL définit les composants logiciels suivants :
- le composant **data** représente des types de données, des instances de données ou de variables partagées ;
- le **process** modélise l'espace d'adressage pour l'exécution des tâches (comme un processus LINUX) ;
- le **thread** modélise un fil d'exécution qui constituent la partie applicative de l'application ;
- le **subprogram** modélise le code (C ou Ada) exécuté par une tâche et dont le comportement est spécifié à l'aide de propriétés, d'annexes ou de séquences d'appels à d'autres sous-programmes.

Enfin, le composant **system** décrit l'assemblage final (ou une configuration) des différents composants logiciels et matériels de l'application distribuée. Il structure et hiérarchise l'assemblage de composants, c'est en quelque sorte la racine du modèle.

Composition

Le standard AADL autorise l'agrégation de composants et la définition de sous-composants selon des règles sémantiques précises. Le tableau 4.1 présente les relations légales entre composants et sous-composants découlant de la logique de composition de l'application. Par exemple, un composant de donnée ne peut pas contenir un processus ou un processeur ; cela ne correspond à aucun assemblage réel.

4.2.2 Eléments d'interfaces et connexions

Les éléments d'interfaces des composants (spécifiés au sein d'un **component_type**) modélisent les flôts de contrôles ou de données. Ils décrivent les mécanismes d'interaction pour l'échange et/ou la synchronisation de données entre les éléments de l'architecture. La communication est alors établie à l'aide des connexions reliant deux éléments d'interfaces. AADL

Composant	Sous-composants
abstract	data, subprogram, subprogram group, thread, thread group, process, processor, virtual processor, memory, bus, virtual bus, device, system, abstract
data	data, subprogram, abstract,
subprogram	data, abstract
subprogram group	subprogram group, subprogram, data, abstract
thread	data, subprogram group, subprogram, abstract
thread group	data, subprogram group, subprogram, thread group, thread, abstract
process	data, subprogram group, subprogram, thread group, thread, abstract
processor	virtual processor, memory, bus, virtual bus, abstract
virtual processor	virtual processor, virtual bus, abstract
memory	memory, bus, abstract
bus	virtual bus, abstract
virtual bus	virtual bus, abstract
device	bus, virtual bus, data, abstract
system	data, subprogram, subprogram group, process, processor, virtual processor memory, bus, virtual bus, device, system, abstract

TABLE 4.1 – Composition légale des composants AADL

définit deux catégories concrètes d'éléments d'interfaces, le **port** et le **parameter**. Une troisième catégorie, les **features**, sert lors la phase préliminaire de la modélisation et est destinée à être raffinée vers l'une des catégories concrètes.

Les ports de type **data** et les paramètres de sous-programmes modélisent l'échange d'une donnée, les ports de type **event** (événements) modélisent l'émission et la réception de signaux. Un sens (**in**, **out**, **in out**) caractérise la nature de l'élément (entrée/sortie).

Le listing 4.1 illustre l'utilisation des éléments d'interfaces et des connexions. Le processus `Transmitter.Impl` contient un thread `Transmitter_Thread.Impl` qui exécute le sous-programme `Transmit`. Le port en entrée du processus est connecté à celui du thread, lui-même connecté au paramètre d'entrée du sous-programme. Le paramètre de sortie du sous-programme est connecté au port en sortie du thread, lui-même connecté au port en sortie du processus.

Les accesseurs de composants (**provides** ou **requires access**) permettent de spécifier les accès aux sous-composants ou les relations entre les composants de données partagées et les sous-programmes pour leur manipulation. Le listing 4.2 décrit le patron de modélisation d'une variable partagée et les sous-programmes pour sa manipulation. Le thread `Producer` qui exécute le sous-programme `Update` déclare requérir l'accès à la donnée partagée `Shared_Data`. Le composant processus spécifiant le thread `Producer` effectue la connexion entre la donnée partagée et le thread.

4.2.3 Propriétés et Annexes

Le langage AADL offre deux mécanismes pour enrichir et étendre les capacités d'expression ou de modélisation du langage cœur : les annexes et les propriétés.

Exemple 4.1 – Connexions entre composants AADL

```
data Message
end Message ;


subprogram Transmit
features
  Input : in parameter Message ;
  Output : out parameter Message ;
end Transmit ;


thread Transmitter_Thread
features
  Input : in event data port Message ;
  Output : out event data port Message ;
end Transmitter Thread ;

thread implementation Transmitter_Thread.Impl
calls
Mycall : { Do_Transmit : subprogram Transmit ; } ;
connections
  p1 : parameter Input -> Do Transmit.Input ;
  p2 : parameter Do Transmit.Output -> Output ;
end Transmitter_Thread.Impl ;


process Transmitter_Process
features
  Input : in event data port Message ;
  Output : out event data port Message ;
end Transmitter_Process ;

process implementation Transmitter_Process.Impl
subcomponents
  Transmitter : thread Transmitter_Thread.Impl ;
connections
  c1 : port Input -> Transmitter.Input ;
  c2 : port Transmitter.Output -> Output ;
end Transmitter_Process.Impl ;
```

Exemple 4.2 – Accesseurs et patron de modélisation des variables partagées en AADL

```
data Shared_Data
features
  Read   : provides subprogram access Read ;
  Update : provides subprogram access Update ;
properties
  Concurrency_Control_Protocol => Priority_Ceiling ;
  Deployment::Priority => 240;
end Shared_Data;


subprogram Read
 features
  Shared_Data : requires data access Shared_Data;
end Read;


subprogram Update
  features
   Shared_Data : requires data access Shared_Data;
end Update;


thread Producer
features
  Shared_Data : requires data access Shared_Data;
properties
  Dispatch_Protocol                => Periodic;
  Period                           => 1000 ms;
  Compute_Execution_time           => 0 ms .. 600 ms;
  Deadline                         => 1000 ms;
  Deployment::Priority => 11;
end Producer;

thread implementation Producer.Impl
calls call_seq : { Do_Update : subprogram Update; };
connections
  Access_Data : data access Shared_Data -> Do_Update.Shared_Data;
end Producer.Impl;


thread Consumer
features
  Shared_Data : requires data access Shared_Data;
  Dispatch_Port : in event port Base_Types::Integer;
properties
  Dispatch_Protocol                => Sporadic;
  Period                           => 600 ms;
  Compute_Execution_time           => 0 ms .. 100 ms;
  Deadline                         => 600 ms;
  Deployment::Priority => 11;
end Consumer;

thread implementation Consumer.Impl
calls call_seq : { Do_Read : subprogram Read; };
connections
  Access_Data : data access Shared_Data -> Do_Read.Shared_Data;
end Consumer.Impl;


process Simple_Producer_Consumer
end Simple_Producer_Consumer;

process implementation Simple_Producer_Consumer.Impl
subcomponents
  Producer    : thread Producer.Impl;
  Consumer    : thread Consumer.Impl;
  The_Shared_Data : data Shared_Data;
connections
  Cnx1 : data access The_Shared_Data -> Producer.Shared_Data;
  Cnx2 : data access The_Shared_Data -> Consumer.Shared_Data;
end Simple_Producer_Consumer.Impl;
```

Annexes

Les annexes permettent d'intégrer à tout composant du modèle d'une application des éléments d'un langage tiers. Ceci permet à la fois d'enrichir la description, de décrire un aspect manquant du système et d'interfacer un modèle AADL avec des outils tiers exploitant la spécification architecturale. Aujourd'hui, plusieurs annexes ont été développées et sont standardisées ou en cours de standardisation. Parmi les plus connues, nous pouvons citer :

- l'annexe de modélisation des données (*Data Model Annex* [SAE Aerospace, 2009d]), qui décrit les patrons de modélisation pour les types de données basiques (nous présentons les différents patrons de modélisation de cette annexe dans le chapitre 8) ;
- l'annexe comportementale (*Behavior Annex* [SAE Aerospace, 2009a]), qui décrit le comportement des composants. Celle-ci est détaillée dans la section 4.3 de ce chapitre ;
- l'annexe de modélisation des erreurs (*Error Model Annex* [SAE Aerospace, 2011]), qui spécifie les erreurs, leur propagation et leur gestion au sein des composants ;
- l'annexe REAL (*Requirements Enforcement Analysis Language* [Gilles, 2008]), qui définit un langage de vérification de contraintes sur les composants AADL.

Le listing 4.4 décrit le comportement d'un composant thread spécifié à l'aide de l'annexe comportementale.

Propriétés

Les propriétés AADL autorisent la spécification d'informations (ou d'annotations) sur un élément AADL (composants, connexions, etc.). Il s'agit d'attributs spécifiant des caractéristiques ou des contraintes s'appliquant aux éléments de l'architecture : contraintes temporelles (pire temps d'exécution, période...), information de déploiement (adresse IP, port de communication), caractéristiques réseaux (bande passante d'un bus...). Le standard AADL définit un ensemble de propriétés standards et offre la possibilité de définir des propriétés spécifiques à l'aide des **property sets**. Le thread `Producer` décrit dans le listing 4.2 (présenté plus haut), spécifie des propriétés temporelles sur son type, sa période, son pire temps d'exécution, etc.

4.2.4 Flots et systèmes adaptatifs avec modes

Le langage AADL introduit aussi des contructions pour la spécification des flux de données et des différentes configurations possibles d'un système adaptatif avec modes.

Les flots AADL permettent de préciser explicitement les flots de données et d'exécution dans un modèle. Cette fonctionnalité vient compléter les analyses de flots de données possibles à partir des éléments d'interfaces des composants, de leur connexion et de leur association au composant bus. Ainsi, les flots AADL permettent de vérifier la cohérence de la topologie de l'application et facilite l'utilisation d'outils d'analyse de flots (les flots sont déjà spécifiés, il n'est pas utile de les déduire de la topologie de l'application).

Les modes AADL permettent l'expression des différentes configurations d'un système. Leur spécification au sein des composants autorise le contrôle des ressources matérielles, logicielles mais aussi des valeurs des propriétés, de l'activation des connexions voir même des sous-composants (par exemple, un système multi-tâche s'exécutant en mode dégradé mono-tâche). La transition d'un mode à un autre s'effectue alors dynamiquement à l'exécution de l'application par la réception d'événements sur les ports de type événement en entrée.

Dans la suite de ce manuscrit, nous ne considérons pas les flots AADL bien que ceux-ci pourraient s'intégrer dans notre processus. De même, nous ne considérons pas la notion de mode. Des travaux au sein de notre équipe ([Borde, 2009]) ont montré que l'analyse et l'implantation des systèmes reconfigurables avec modes nécessitent un processus d'ingénierie spécifique à cette famille de systèmes.

4.2.5 Instanciation d'un modèle AADL

Dans une description architecturale AADL, la plupart des composants peuvent contenir des sous-composants permettant ainsi de définir une description hiérarchique du système. Le composant système est le composant de plus haut niveau contenant toutes les instances des autres composants. Le standard AADL définit la phase d'instanciation d'un modèle AADL, on parle alors de la transformation d'un modèle AADL déclaratif vers un modèle d'instance. Le modèle d'instance donne une vue hiérarchique de la description architecturale AADL à partir d'un nœud racine, le composant système.

L'instanciation d'un système vise à résoudre les références et les valeurs des propriétés des différentes instances de composants comme les calculs et l'héritage des valeurs de propriétés (obtenues par extension ou raffinement de composants), la résolution des prototypes, la résolution bout-en-bout des chemins des connexions des éléments d'interfaces, etc.

4.2.6 Services de l'exécutif AADL

Le standard AADL donne des recommandations sur le comportement et les services que doit fournir un exécutif AADL *i.e.* une runtime supportant les composants logiciels AADL (threads, sous-programmes, etc.). Notamment, le standard spécifie un certain nombre d'interfaces relatives aux services qu'un exécutif AADL doit offrir pour assurer l'émission et la réception des données à travers les composants threads. Nous discuterons plus précisément du rôle de ces services dans le chapitre 6 de ce manuscrit.

4.2.7 Eléments pour la modélisation

AADLv2 introduit différents éléments, présentés dans le listing 4.3, pour structurer les déclarations (paquetages...) et faciliter la description architecturale d'une application répartie (réutilisation, extension, etc.) :

- les **packages** AADL organisent les déclarations des composants permettant entre autres la séparation de la partie matérielle et logicielle et la définition de bibliothèques de composants. Un mécanisme d'importation (**with**) autorise le référencement des composants d'un paquetage ;
- le composant **abstract** permet de modéliser un composant sans connaître son déploiement final. Des mécanismes d'extension et de raffinement de composants permettent plus tard, dans la modélisation, de raffiner un composant abstrait en un composant matériel ou logiciel concret. Ces mêmes mécanismes d'extension et de raffinement permettent la réutilisation des composants ;
- les **prototypes** autorisent la définition de composants génériques qui seront instanciés et configurés plus tard dans la modélisation. L'instanciation et la configuration des prototypes s'effectuent à l'aide du mécanisme d'extension et des **prototype bindings** permettant de typer ou de résoudre (dans le cas des sous-programmes prototypes) le composant prototypé.

Exemple 4.3 – Utilisation des paquetages, des extensions et des prototypes AADL

```
package Compute_Library public

with Base_Types;

subprogram Generic_Spg
  prototypes
    dt      : data;
  features
    input   : in  parameter dt;
    output  : out parameter dt;
end Generic_Spg;

data My_Float extends Base_Types::Float end My_Float;

subprogram Compute_Float extends Generic_Spg (dt => data My_Float)
end Compute_Float;

end Compute_Library;
```

Cette section a présenté les différents composants pour la spécification architecturale d'une application à l'aide du langage AADLv2. Dans la section suivante, nous nous intéressons à la spécification des aspects comportementaux des composants du système.

4.3 Introduction à l'annexe comportementale

L'annexe comportementale AADL [SAE Aerospace, 2009a] a été développée dans l'objectif de préciser et d'affiner les aspects comportementaux implicites issus de la description architecturale AADL du système. Elle propose à travers la spécification d'un automate à états/transitions et de différents langages (actions, expressions...) de décrire le comportement interne des composants à travers leur interface de communication et un ensemble d'actions précisant l'émission, la réception et la manipulation de données. Cette annexe vise l'ensemble des composants architecturaux du langage AADLv2. Cependant, l'accent est logiquement mis sur les principaux composants logiciels traduisant le comportement du système : les composants threads et les sous-programmes.

Cette section présente une introduction à l'annexe comportementale AADL et nous permet de cerner le périmètre d'activité et de raffinement proposé par celle-ci. Notamment, nous pensons que la nouvelle version de cette annexe ([SAE Aerospace, 2009a]) a un pouvoir d'expressivité important autorisant jusqu'à la modélisation complète de l'exécutif AADL sous-jacent partiellement décrit par le standard AADLv2.

4.3.1 Spécification comportementale, structure d'automate et langages

Une description comportementale AADL est spécifiée à l'aide de cinq langages définissant des concepts comportementaux avec une syntaxe, un ensemble de règles de nommage, de règles légales, de règles sémantiques et de règles assurant la cohérence de la spécification comportementale vis-à-vis de la description architecturale.

L'exemple de code AADL 4.4 illustre la combinaison de ces langages pour établir la description comportementale de l'exécution du thread sporadique **Consumer** présenté dans l'exemple 4.2. On y retrouve différents types d'états respectant la sémantique de l'automate d'exécution du thread défini par le standard AADL et détaillé dans le chapitre 5. Notamment, l'état **complete** modélise l'état de suspension et de reprise d'activité du composant.

Une **transition** représente un changement de l'état courant (source) à un état de destination. Le déclenchement d'une transition s'effectue quand une condition spécifique est évaluée

Exemple 4.4 – Description comportementale du thread sporadique `Consumer`

```
thread Consumer.impl
calls call_seq : { Do_Read : subprogram Read; };
connections
 Access_Data : data access Shared_Data -> Do_Read.Shared_Data;

annex behavior_specification {**
 variables
  lastValue : Base_Types::Integer;
 states
  stInit     : initial state;
  stDispatch : complete final state;
 transitions
  On_Init : stInit -[ ]-> stDispatch { lastValue := 0 };

  On_Dispatch : stDispatch -[on dispatch Dispatch_Port]-> stDispatch {
     lastValue := lastValue + Dispatch_Port;

     if (lastValue mod 2 = 0)
       Do_Read!(lastValue);
     end if;
 }
 **};
end Consumer.impl;
```

à vraie. Lorsqu'une transition est déclenchée, les actions attachées à celle-ci sont exécutées (voir le bloc {...} attaché à la transition **On_Dispatch** de l'exemple 4.4).

Les **conditions** (*behavior condition*) sont exprimées à l'aide d'un second langage et affinent l'utilisation des ports et des appels de sous-programmes AADL. Ces deux constructions permettent de spécifier le déclenchement d'activité d'une tâche. Dans notre exemple, le port **Dispatch_Port** défini dans l'interface du composant AADL (voir exemple 4.2, colonne droite) est utilisé comme condition pour le déclenchement de l'activité du composant.

Un troisième langage permet de spécifier les interactions du composant avec ses sous-composants ou avec d'autres composants à travers l'utilisation de ses éléments d'interfaces (données partagées, ports, appels de sous-programmes, etc.).

Un langage d'actions permet quant à lui de décrire les actions exécutées lors du déclenchement d'une transition. Celui-ci utilise le langage d'interaction pour définir les actions entre les composants (dans notre exemple, appel du sous-programme **Do_Read !(...)**).

Enfin, un langage d'expression basé sur le langage d'expression Ada permet la manipulation des données AADL (sous-composants de données, variables d'automates, donnée reçue sur le port, valeur de paramètre...) à travers la spécification d'expression logique, relationelle ou arithmétique.

Nous venons de présenter la structure globale d'un automate comportemental défini par l'annexe. Dans la suite de cette section, nous nous intéressons aux automates particuliers des composants threads et sous-programmes ainsi qu'aux langages d'interactions, d'actions et d'expressions s'apparentant à des constructions classiques d'un langage de programmation impérative.

4.3.2 Spécificités des automates des threads et des sous-programmes

L'annexe comportementale définit des restrictions pour la spécification des automates des threads et des sous-programmes.

Ainsi, l'automate comportemental d'un composant thread doit décrire un unique état initial représentant l'état de la tâche avant son initialisation, un ou plusieurs états de suspension (`complete`), zéro ou plusieurs états d'exécution (modélisant un chemin d'exécution des actions associées aux transitions) et un ou plusieurs états de finalisation. Cette structure d'automate vise à assurer la cohérence et le respect de la sémantique de l'automate d'exécution

du thread défini par le langage cœur et détaillé dans le chapitre 5. Le langage de spécification des conditions pour le déclenchement d'activité d'un composant est restreint au composant thread. Celui-ci spécifie les différentes conditions possibles à savoir la réception d'un événement sur un port, l'appel d'un sous-programme fourni par le composant ou le déclenchement d'un `timeout`.

L'automate d'un composant sous-programme doit, quant à lui, décrire un unique état initial modélisant le point de départ de l'appel du sous-programme, zéro ou plusieurs états intermédiaires d'exécution (chemin d'exécution) et uniquement un état final représentant la terminaison de l'appel (de l'exécution) du sous-programme. Les états de suspension sont strictement interdits par l'annexe afin d'éviter les risques de suspension d'une tâche au milieu de l'exécution d'un sous-programme.

4.3.3 Interactions entre composants

L'annexe comportementale propose un langage d'interaction pour expliciter les interactions et la communication entre les différents composants architecturaux. Ce langage définit notamment :

- le comportement d'un port et de sa file d'attente lors de l'arrivée d'une donnée (gel de la donnée reçue, ordre de délivraison...) ;
- des constructions spécifiques pour la manipulation des données reçues sur un port (lecture d'une donnée, nombre de messages reçus, affectation d'une donnée à un port, envoi d'une donnée à partir d'un port, etc.) ;
- le comportement des données reçues à travers les paramètres de sous-programme ;
- des constructions spécifiques pour la réception, la consultation, l'affectation des données à travers les paramètres de sous-programmes ;
- des constructions spécifiques pour modéliser l'accès aux données partagées (modélisation d'une section critique...).

La précision du comportement des ports est basée sur l'utilisation de constructions de l'annexe et l'interprétation de propriétés AADL. Une syntaxe spécifique est définie pour les constructions autorisant la manipulation des ports, des paramètres de sous-programmes et des sous-programmes modélisant une section critique.

4.3.4 Eléments du langage d'actions

Le langage d'action de l'annexe comportementale définit les différentes actions qui sont exécutées lors du déclenchement d'une transition *i.e.* lorsque la condition ou la garde de la transition est évaluée à vraie.

Ce langage utilise les constructions spécifiées par le langage d'interactions et introduit de nouvelles constructions équivalentes aux constructions classiques d'un langage de programmation impératif comme Ada. On y retrouve :

- la structure conditionnelle `if ... then ... elsif ... else` ;
- les boucles de contrôles `for ... in`, `forall ... in`, `while ...` et `do ... until` ;
- l'opération d'affectation de valeur à une variable, opérateur `:=` ;
- l'opération d'appel de sous-programme `<nom_spg>!(param1,param2)` permettant l'appel d'un sous-programme référencé dans un paquetage AADL ou à l'aide d'un accesseur de sous-programme spécifié dans l'interface du composant ;
- les éléments de syntaxe pour l'accès aux sous-composants de données ou aux composants de données spécifiés à l'aide d'accesseurs ;

– deux opérations pré-définies `computation(min,max)` et `delay(min,max)` exprimant respectivement l'utilisation du processeur et un temps de suspension compris entre durée minimale et maximale.

Des éléments de syntaxe et des règles sémantiques régissent l'utilisation de ces constructions. De plus, l'annexe comportementale introduit la possibilité de spécifier des séquences ordonnées d'actions ou un ensemble d'actions parallélisées. Dans la sous-section suivante, nous donnons un exemple d'utilisation du langage d'actions.

4.3.5 Eléments du langage d'expressions

Le langage d'expression proposé par l'annexe comportementale est basé sur le langage d'expression du langage de programmation Ada. On y retrouve les éléments de syntaxe et les constructions spécifiant des expressions logiques, des expressions relationnelles ou des expressions arithmétiques. Les éléments du langage d'expression s'appliquent uniquement aux composants AADL spécifiant des données, à savoir les ports, les paramètres de sous-programmes, les instances de données (sous-composants de données) et les données partagées (référencées à l'aide des accesseurs de données dans l'interface du composant).

Exemple 4.5 – Description comportementale du sous-programme `Read`

```
1  subprogram Read
2    features
3      Stream : in out parameter Message_Type.Impl;
4      Item : out parameter PolyORB_HI_Streams::Stream_Element_Array;
5      Last : out parameter PolYORB_HI_Streams::Stream_Element_Offset;
6  end Read;
7
8  subprogram implementation Read.Impl
9    subcomponents
10     L1 : data PolyORB_HI_Streams::Stream_Element_Count
11               { Data_Model::Initial_Value => ("Item'Length");
12                 Access_Right => read_only; };
13     L2 : data PolyORB_HI_Streams::Stream_Element_Count
14               { Data_Model::Initial_Value => ("Length (Stream)"); };
15     Tmp_L : data PolyORB_HI_Streams::Stream_Element_Count;
16     Item_First : data PolyORB_HI_Streams::Stream_Element_Offset
17               { Data_Model::Initial_Value => ("Item'First"); };
18   annex behavior_specification {**
19   states
20     s0 : initial final state;
21
22   transitions
23     t0_0 : s0 -[ ]-> s0 {
24       if (L1 < L2)
25         L2 := L1
26       end if;
27
28       Tmp_L := L2 - 1;
29
30       for ( J : PolyORB_HI_Streams::Stream_Element in 0 .. Tmp_L )
31         {
32          Item[Item_First + J] := Stream.Content[Stream.First + J]
33         };
34
35       Last := Item_First + L2 - 1;
36       Stream.First := Stream.First + L2
37     }
38   **};
39 end Read.Impl;
```

L'exemple 4.5 illustre quelques éléments des langages d'interactions, d'actions et d'expressions combinés pour la spécification du comportement d'un sous-programme AADL. Le comportement décrit est l'extraction des données dans un tampon de communication (**Stream** de type **Message_Type.Impl**). On y retrouve l'utilisation des sous-composants de données du sous-programme. L'utilisation des propriétés de l'annexe de modélisation des données nous permet de spécifier les valeurs d'initialisation de ces variables locales (exemple du sous-composant de données **L1**, l.10).

Exemple 4.6 – Types énumérés et structurés en AADL

```
data Enum_Type
properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("InS_K", "WoM_K");
end Enum_Type;

data Struct_Type
properties
  Data_Model::Data_Representation => Struct;
end Struct_Type;

data implementation Struct_Type.impl
subcomponents
  Enum    : data Enum_Type;
  Value   : data Base_Types::Integer;
end Struct_Type.impl;
```

```
subprogram Check_Element
features
  Enum    : in out parameter Enum_Type;
  Output :    out parameter Struct_Type.impl;
end Check_Element;

subprogram implementation Check_Element.Impl
annex behavior_specification {**
  states       stExec : initial final state;
  transitions  tExec : stExec -[]-> stExec {
                 if (Enum = Enum_Type.InS_K)
                       Output.Enum  := Enum_Type.InS_K;
                       Output.Value := Output.Value + 1
                 end if
               }
 **};
end Check_Element.Impl;
```

4.3.6 Contributions à l'annexe comportementale

Les travaux menés durant cette thèse sur la modélisation de composants AADL nous ont amenés à contribuer et à proposer des évolutions à l'annexe comportementale. Nous présentons dans cette section les évolutions majeures que nous avons introduites.

Support pour les types énumérés, structurés et union

Le standard AADL recommande la spécification des types de données à l'aide de l'annexe de modélisation. Celle-ci définit l'ensemble de propriétés *Data_Model* contenant différentes propriétés pour spécifier la nature d'une donnée à l'aide des composants de données. Parmi celles-ci, nous retrouvons le patron de modélisation des types énumérés présentés dans l'exemple de code AADL 4.6.

Le support des types énumérés est manquant dans l'annexe comportementale. En effet, la syntaxe actuelle ne permet pas d'effectuer des comparaisons entre deux instances d'une énumération. Pour palier ce manque, nous avons introduit la syntaxe et les règles légales pour l'accès aux littéraux des énumérations au sein de l'annexe comportementale. La description comportementale du composant **Check_Element.Impl** illustre celle-ci (exemple 4.6).

Comme pour les types énumérés, les types structurés et les types union ne sont pas supportés par l'annexe comportementale. Une syntaxe et des règles légales pour le support de ces types ont été ajoutées sur le standard de l'annexe. La syntaxe pour la manipulation des sous-composants (des champs) d'un type structuré est décrite dans l'exemple 4.6.

Support pour la définition de bibliothèques de sous-programmes génériques

L'annexe comportementale autorise la manipulation des composants prototypes pour la définition de composants génériques à étendre et paramétrer lors de l'élaboration d'un modèle AADL. La spécification actuelle est incomplète car elle ne porte que sur les composants de données. La spécification d'un prototype de type sous-programme ou d'une séquence d'appel de sous-programme est impossible. Nous avons proposé et ajouté ces éléments dans l'annexe.

Cette dernière modification autorise la modélisation de bibliothèques de composants sous-programmes génériques configurables lors des différentes évolutions du modèle AADL. Dans le chapitre 6 et 7 nous illustrons l'utilisation d'une telle bibliothèque.

4.4 Avantages et restrictions pour les systèmes TR^{2}E critiques

4.4.1 Avantages pour l'analyse et l'implantation

Dans notre étude préliminaire 2, nous avons exposé différents travaux proposant des processus d'ingénierie autorisant la modélisation, l'analyse et la production automatisée (génération, déploiement et configuration) des systèmes TR^{2}E à partir d'une description architecturale des composants d'une application dans le langage AADL.

Les composants matériels et logiciels proposés par le langage cœur permettent la spécification des entités concrètes d'une application. Le mécanisme des propriétés s'avère efficace pour l'intégration d'exigences fonctionnelles et non fonctionnelles (contraintes temporelles, informations de déploiement...) permettant d'exploiter le modèle AADL pour de nombreuses analyses. Enfin, l'intégration de spécifications comportementales au sein même de la description architecturale est un avantage conséquent pour expliciter le comportement des composants mais aussi pour assurer la cohérence de ces descriptions comportementales avec la spécification architecturale de l'application.

En outre, [Zalila *et al.*, 2008] expose, dans ses travaux, un processus d'implantation automatisée d'une application TR^{2}E critique piloté uniquement par l'analyse d'une description architecturale AADL. Nous observons tout de même que ces différentes approches n'intègrent aucun élément de son annexe comportementale.

De plus, le langage AADL vise la famille des systèmes TR^{2}E au sens large et autorise des constructions qui ne sont pas conformes aux exigences des systèmes critiques. Ces éléments sont présentés dans la sous-section suivante.

4.4.2 Restrictions architecturales et comportementales

Dans cette section, nous restreignons l'utilisation des éléments du langage AADL aux constructions ne compromettant pas l'analyse statique et l'implantation d'un système critique fiable. Dans le cadre de la production d'un système critique, nous souhaitons respecter les restrictions du profil Ravenscar et celles liées aux caractères des systèmes critiques présentées dans la section 2.3.2 de notre état de l'art (chapitre 2). De plus, afin de faciliter l'automatisation des étapes d'analyse et de génération nous supprimons les éléments de modélisation non concrets pour la réalisation de ces tâches.

Restriction des composants architecturaux

Dans le chapitre précédent, nous avons défini des critères pour restreindre l'utilisation du langage AADL. Le critère C1 porte sur la restriction de l'utilisation des composants abstraits, des prototypes, des ports de communication et sur les patrons de modélisation des types de données.

- *Les composants abstraits et les prototypes* sont utilisés au début de la modélisation avant l'identification précise des composants logiciels et matériels. Ils sont raffinés par héritage, extension ou *bindings* lors de l'évolution de la modélisation. A la fin de cette étape, la réécriture du composant concret final permet d'éliminer l'ensemble de ces constructions non pertinentes pour l'analyse et la génération.
- *Les ports de communication* définissent les connecteurs et les interactions entre les composants et les propriétés AADL qui y sont attachées, et spécifient leur comportement. L'utilisation de patrons comportementaux nous permet d'expliciter ce comporte-

ment ainsi que les éléments du support d'exécution (intergiciel) mis en jeu. Cette réécriture réduit la différence sémantique entre le composant modélisé et son implantation physique.
– La modélisation des données (types, constantes et variables) est très permissive à l'aide du langage AADL. L'utilisation de l'annexe de modélisation des données limite ces patrons de modélisation et renforce la génération de code.

Restriction des constructions comportementales

Pour satisfaire le critère C2, nous avons développé des patrons comportementaux pour les différents types de tâches supportés dans notre approche. Ces patrons ont été élaborés dans le but de préserver l'analysabilité du modèle AADL et de permettre la production du code source du composant à partir de sa spécification comportementale. Ces patrons sont présentés dans le chapitre 5.

Restrictions pour l'analyse statique

Pour satisfaire le critère C3 et ainsi assurer l'analysabilité statique du système modélisé, nous avons contraint l'ensemble de nos patrons de modélisation (architecturaux et comportementaux) selon les recommandations du profil Ravenscar. Pour s'assurer de la conformité avec ce dernier, nous avons défini **trois** restrictions sur la description architecturale (RA1, RA2, RA3) et **deux** (RC1 et RC2) sur la spécification comportementale - *i.e.* les automates :

RA1. Chaque processeur doit être modélisé avec un attribut spécifiant un ordonnancement à priorité fixe (par exemple l'ordonnancement de type *FIFO within priorities*) et un attribut spécifiant que l'ordonnanceur est préemptif.

RA2. Chaque tâche du système doit être cyclique - *i.e. périodique ou sporadique*.

RA3. Chaque objet protégé du système doit être modélisé avec un attribut spécifiant la politique de plafonnement de priorité (PCP).

RC1. Chaque tâche doit posséder au plus un point de suspension.

RC2. Chaque tâche ne doit pas terminer son exécution - *i.e. infinie*.

L'interdiction de la création dynamique de tâches imposée par le profil Ravenscar est assurée par la description architecturale des entités actives de l'application (les composants threads) et par notre processus de production automatique. Ainsi, grâce au modèle AADL, nous pouvons déterminons le nombre exact de tâches à créer avant l'implantation du système.

Nous souhaitons aussi vérifier un certain nombre de restrictions supplémentaires inhérentes aux systèmes critiques. Ainsi, il s'agit de garantir :

1. La présence d'éléments d'interfaces non connectés, compromettant la validation du système et conduisant à la génération de code inutile ;
2. L'utilisation des types de données de taille non bornée. Ceci permet l'évaluation de la taille exacte de toutes les données et l'allocation statique de la mémoire requise.
3. L'utilisation d'une borne temporelle spécifiant le temps minimal entre deux déclenchements d'activités des threads sporadiques.

L'ensemble des restrictions que nous venons de présenter sont vérifiés sur la modélisation AADL à différentes étapes de notre processus de développement. Nous notons que la même approche a été adoptée par OCARINA mais concerne uniquement la partie architecturale de l'application.

Dans les chapitres 7 et 8 nous présentons les différents patrons de modélisation pour l'analyse et la génération de code mettant en œuvre les restrictions que nous avons définies. Nous illustrons ces différents patrons par des exemples concrets de composants AADL.

4.5 Outillages AADL existants

Pour terminer notre introduction sur le langage AADL, nous nous intéressons et présentons dans le chapitre 5 une rapide description des principaux outils de modélisation, d'analyse et de génération basés sur le langage AADL. Nous devons noter que ces outils supportent notamment zéro, une ou plusieurs annexes du langage AADL. Pour compléter le panorama des outils existants, il est possible de consulter le site officiel AADL[2] et l'article suivant [Hugues, 2011].

4.6 Synthèse

Ce chapitre a proposé une introduction aux éléments du langage AADLv2 et aux éléments de son annexe comportementale. Le langage AADL est le formalisme pivot que nous avons retenu pour atteindre les différents objectifs de ces travaux de thèse. Notamment, nous savons que celui-ci dispose des capacités pour la spécification, la configuration et le déploiement des composants d'une application TR^{2}E.

Son annexe comportementale nous permet d'affiner la spécification du comportement des composants architecturaux notamment pour les composants threads et sous-programmes. L'utilisation des constructions s'apparentant à un langage de programmation impératif nous offre des perspectives pertinentes pour la réutilisation de ces descriptions dans un processus de production automatique du code source des composants.

Nous avons décrit l'ensemble des éléments architecturaux et comportementaux spécifiés par le standard du langage cœur dans sa version 2.0 [SAE Aerospace, 2009b] et de la *draft* v2.12 de son annexe comportementale [SAE Aerospace, 2009a]. Durant notre travail de thèse, cette dernière a poursuivi son évolution et nous sommes à l'origine de certaines améliorations proposées par notre équipe de recherche. Ces contributions nous ont positionnés comme contributeur pour l'annexe comportementale et nous nous sommes vus confier la tâche de la rédaction des évolutions et des modifications de l'annexe comportementale [TELECOM ParisTech, 2011].

Après avoir constaté le caractère généraliste du langage cœur et de son annexe, nous avons défini un ensemble de restrictions architecturales, comportementales et non fonctionnelles pour garantir l'analysabilité statique des composants modélisés et faciliter l'implantation des outils d'analyse et de génération.

Dans la suite de ce manuscrit, nous détaillons ce sous-ensemble d'éléments pour la définition d'un profil AADL autorisant l'élaboration de patrons de génération de code préservant l'analyse. Ce profil servira de base pour le processus de raffinement, d'analyse et d'implantation automatisé que nous proposons et fait l'objet du chapitre suivant.

2. http://www.aadl.info

Troisième partie

Détail des contributions