Niveau d'Abstraction Pour l'Analyse et la Génération de Code

SOMMAIRE

5.1 INTRODUCTION 81
5.2 COMPOSANTS REQUIS POUR L'ANALYSE 82
5.2.1 Contraintes spécifiques aux systèmes critiques 83
5.2.2 Vérification et validation de modèles AADL 83
5.2.3 Granularité des composants et niveau d'abstraction 85
5.3 COMPOSANTS REQUIS POUR LA GÉNÉRATION DE CODE 86
5.3.1 Composants concrets pour la génération 87
5.3.2 Granularité des composants architecturaux pour la génération . . 89
5.4 NIVEAU DE RAFFINEMENT ET SPÉCIFICATIONS COMPORTEMENTALES . . 90
5.4.1 Description comportementale des composants threads 90
5.4.2 Description comportementale des composants sous-programmes 95
5.5 SYNTHÈSE : LE PROFIL AADL-HI RAVENSCAR 98

5.1 Introduction

Dans le chapitre 3, nous avons exposé les motivations du raffinement incrémental d'un modèle architectural AADL initial (spécifié par l'utilisateur) pour l'obtention d'un modèle de code complet et analysable. Ce modèle de code est une représentation des composants applicatifs et intergiciels de l'application avec une sémantique proche de son implantation. Le processus de raffinement incrémental constitue le cœur de notre approche et est contruit sous la forme d'une chaîne de transformations endogènes verticales assurant la réduction du niveau d'abstraction de la description initiale (présenté chapitre 7).

Le modèle architectural et comportemental obtenu doit être utilisé comme point d'entrée par nos différents outils et, dans le plus grand nombre de cas possibles, par les outils d'analyse existants de la communauté AADL. Il doit notamment permettre d'affiner les analyses de faisabilité (ordonnancement, comportement, etc.) par la prise en compte des ressources intergicielles. Enfin, ce modèle doit autoriser l'expression d'un *mapping* simple vers les contructions classiques des langages de programmation impératifs (par exemple Ada ou C).

Notre étude préliminaire (chapitre 2) a révélé l'utilisation de représentation du système critique à des niveaux d'abstraction différents par les outils d'analyse et de génération. Se pose

alors la question du niveau d'abstraction requis par notre modèle de code analysable. Selon la définition 1.5 (chapitre 2), celui-ci dépend du niveau de détail exprimé pour la description des composants. Un niveau d'abstraction trop bas facilite la génération de code à partir de la description du composant, mais peut parasiter le modèle et complexifier les transformations vers les formalismes utilisés par les outils d'analyse. Un niveau d'abstraction trop élevé ne permettrait pas de décrire les composants intergiciels dépendants de la plate-forme d'exécution et rendrait difficile une démarche automatique de production. Par conséquent, nous devons déterminer un niveau d'abstraction intermédiaire qui limiterait les tranformations pour l'élaboration des modèles d'analyse et autoriserait l'expression des informations architecturales et comportementales nécessaires pour décrire les composants applicatifs et intergiciels de l'application.

Le langage AADL et son annexe comportementale (présenté chapitre 4) dispose de caractéristiques intéressantes pour la réalisation de cet objectif. L'étude des éléments architecturaux du langage AADL (composant, port, propriétés...) impliqués dans les modèles d'analyse et utilisés par les outils de génération de code détermine un sous-ensemble du langage AADL pertinent. L'intégration des descriptions comportementales à ce sous-ensemble précise le comportement des composants et contribue à réduire le niveau d'abstraction pour permettre la modélisation des composants intergiciels. Ces éléments combinés aux restrictions que nous avons définies à partir des recommandations du profil Ravenscar et pour la réalisation des systèmes critiques (voir chapitre 4) constituent un profil de modélisation (AADL-HI Ravenscar) définissant un niveau d'abstraction intermédiaire pour l'analyse et la génération. Nous élaborons nos patrons de génération (achitecturaux et comportementaux) préservant l'analyse à partir de ce profil.

Ce chapitre est organisé de la manière suivante. La section 5.2 (respectivement 5.3) synthétise les différents composants AADL impliqués par les modèles d'analyse (resp. la génération). La section 5.4 présente les patrons de modélisation comportementale que nous avons définis pour les composants AADL retenus pour notre approche. La section 5.5 synthétise nos résultats pour la définition du profil AADL-HI Ravenscar.

5.2 Composants requis pour l'analyse

Rappel. La vérification d'un système vise à s'assurer que celui-ci est correctement construit et sa validation vise à garantir le respect de ses spécifications. Validation et vérification s'effectuent au travers de différentes analyses reposant sur une abstraction structurée et complète modélisant un aspect particulier du système. Ces aspects sont liés à la satisfaction de contraintes relatives à la spécificité du système.

Le langage AADL permet d'exprimer des contraintes fonctionnelles et non fonctionnelles sous une forme condensée et structurée au sein d'une représentation unique. Il sert de point d'entrée pour procéder à l'analyse du système ou de langage pivot pour la production d'un modèle d'analyse dans un formalisme tiers. Dans cette section, nous rappelons les différentes contraintes inhérentes aux systèmes TR^{2}E critiques. Puis, nous nous intéressons aux principaux outils d'analyse basés sur AADL et plus particulièrement aux composants AADL et à leurs attributs (propriétés...) qu'ils utilisent. Ces éléments nous permettent de déterminer la granularité du composant AADL et le niveau d'abstraction requis pour différentes analyses.

5.2.1 Contraintes spécifiques aux systèmes critiques

Afin de déterminer les composants AADL les plus pertinents requis par les outils d'analyse, nous proposons une classification simple des contraintes fonctionnelles et non fonctionnelles à satisfaire dans le contexte des systèmes TR^{2}E critiques. Cette classification nous permet de distinguer les caractéristiques des différentes suites d'outils supportant une modélisation AADL. Ainsi, nous distinguons six familles de contraintes :

1. les contraintes sur la cohérence de la modélisation : conformité de la description architecturale, hiérarchie entre composants, sémantique des composants ;
2. les contraintes structurelles : informations de déploiement, préparation de l'application en vue de la distribuer (critère distribué de l'application) ;
3. les contraintes sur la consommation des ressources matérielles et logicielles : utilisation processeur, bande passante, empreinte mémoire (critère embarqué de l'application) ;
4. les contraintes temporelles : respect d'échéances temporelles, ordonnançabilité (critère temps-réel de l'application) ;
5. les contraintes de criticité : intégrité des données, sûreté de fonctionnement, sécurité, tolérances aux pannes, déterminisme ;
6. les contraintes comportementales : validation du comportement vis-à-vis des spécifications de l'utilisateur, équité, interblocage, vivacité, etc.

Dans la sous-section suivante, nous présentons les principaux outils d'analyse visant la vérification d'une ou plusieurs de ces contraintes.

5.2.2 Vérification et validation de modèles AADL

Le tableau 5.1 synthétise les principaux outils et les familles de contraintes qu'ils vérifient. Pour certaines plates-formes d'intégration (par exemple TASTE ou AADL INSPECTOR intégrant l'outil CHEDDAR), nous ne considérons que les fonctionnalités supplémentaires offertes ou intégrées.

	OSATE2	OCARINA	CHEDDAR	TASTE	TINA	AADL INSPECTOR
Cohérence	x	x		x		x
Structurelles		x				
Ressources	x	x	x			x
Temporelles			x	x		x
Criticité	x	x				
Comportement					x	x

TABLE 5.1 – Contraintes et outils d'analyses AADL

OSATE2

OSATE2 [SEI/CMU, 2011] est une plate-forme dédiée à l'analyse et à l'intégration d'outils basés sur les modèles AADL. La partie frontale permet de vérifier la cohérence de la description (syntaxe et sémantique des composants). Un certain nombre de plugins d'analyse assurent aussi la vérification de propriétés de sécurité et de sûreté, du flot d'exécution des données, de l'allocation et de la consommation mémoire, de l'intégrité et de la cohérence des données, de la consommation d'énergie et de la propagation d'erreurs.

Tous les composants AADL sont supportés. Les analyses sont effectuées à partir d'un modèle déclaratif AADL ou d'un modèle d'*instance* généré par OSATE2. Celui-ci décrit une instance du système modélisé où les liens d'extension, de raffinement, de protypage et certains calculs sur les valeurs de propriétés sont résolus.

OCARINA

OCARINA [TELECOM ParisTech, 2012] propose une partie frontale assurant l'analyse de la cohérence d'une description architecturale et des informations structurelles (déploiement, répartition) pour la génération du système. Différentes analyses à travers l'implantation du langage de contraintes REAL et l'exportation de modèles vers le formalisme des réseaux de Petri ou vers l'outil CHEDDAR sont possibles :

- l'annexe AADL-REAL [Gilles, 2008] permet la vérification de contraintes et le calcul d'expressions complexes sur les modèles architecturaux AADL. Elle supporte un sous-ensemble de composants (matériels et logiciels) et leurs propriétés associées. Il permet la vérifiation de la cohérence du dimensionnement d'une application (projet MOSIC), des spécifications (respect des patrons de modélisation, respect des restrictions du profil AADL/Ravenscar) et des exigences de sécurité et sûreté au sein de la plate-forme POK.
- OCARINA/RDP est une partie dorsale permettant la transformation des composants architecturaux AADL vers les réseaux de Petri. Ainsi, nous pouvons vérifier le comportement d'un système par l'analyse de contraintes telles l'absence d'interblocage, la famine, la vivacité ou l'équité dans une application répartie (multi-tâches) contenant des ressources partagées. Le *mapping* des composants AADL vers les constructions des réseaux de Petri est décrit dans [Renault, 2009] (chapitre 4, section 4.1.1) et nécessite les composants suivants : sous-programmes, threads, processeurs, variable partagée, port. Les propriétés AADL standards pour ces composants sont aussi requises. Enfin, les composants systèmes et processus sont utilisés pour structurer la spécification et apportent des informations hiérarchiques sous la forme d'espace de nommage permettant la traçabilité des résultats de la vérification et la détection de composants fautifs.

CHEDDAR

CHEDDAR permet la vérification de performances et de l'ordonnançabilité d'un système modélisé en AADL. Dans [Singhoff *et al.*, 2005], les auteurs décrivent quels sont les éléments du langage AADL pris en compte pour l'analyse d'ordonnancement, à savoir : les composants de données (variable partagée), les accesseurs aux composants de données, les threads, les processus, les processeurs et les composants systèmes. CHEDDAR utilise certaines propriétés AADL standards mais aussi un ensemble de propriétés AADL (pour les composants de données, threads, processeurs) spécifiques à l'analyse d'ordonnancement. Enfin, CHEDDAR permet à l'aide de la spécification des ports de threads d'effectuer une analyse mémoire sur la taille et la capacité des tampons pour les communications entre threads.

TASTE (MAST)

TASTE [Perrotin *et al.*, 2011] est une plate-forme d'intégration assurant l'interopérabilité entre différents outils pour le développement et l'analyse de systèmes TR^{2}E. Ainsi, TASTE intégre de nombreux outils tels OCARINA (pour la génération), CHEDDAR, MAST, MARZHIN (pour l'analyse), etc. Nous nous intéressons, ici, uniquement à l'intégration de l'outil MAST

pour l'analyse de l'ordonnancement du système. MAST nécessite une transformation des éléments architecturaux du langage AADL vers les éléments du méta-modèle MAST. Le *mapping* de AADL vers MAST est exprimé dans [Perrotin *et al.*, 2011] (chapitre 16, section 5). Ainsi, les composants processeur, thread, sous-programme, bus, périphérique et les composants de données sont utilisés. Un sous-ensemble des propriétés standards AADL associées à ces composants est requis.

AADL-FIACRE-TINA

L'outil TINA permet l'analyse comportementale de réseaux de Petri et de systèmes temporisés. Dans [Berthomieu *et al.*, 2010], les auteurs proposent une transformation de modèles AADL vers FIACRE pour exprimer le comportement, les aspects temporels et la sémantique d'exécution du système puis une transformation de modèles FIACRE vers le formalisme pris en compte par TINA. Les auteurs décrivent les composants architecturaux et les éléments de l'annexe comportementale AADL transformés. L'annexe comportementale utilisée ici est une ancienne version de celle présentée chapitre 4, basée sur le langage AADL 1.0 et n'intégrant pas de langage d'expression aussi détaillé que la nouvelle. Ainsi, les composants inclus sont les composants threads, les ports de threads, les composants de données (variable partagée), les accesseurs aux composants de données, les modes et les constructions de l'annexe comportementale relatives aux threads. Un sous-ensemble de propriétés AADL standards dédiées à ces composants est aussi employé.

AADL INSPECTOR (MARZHIN)

AADL INSPECTOR [Ellidiss Technologies, 2012] est un outil d'analyse extensible basé sur l'expression et la vérification de règles sur les modèles AADL. Nous nous intéressons ici à la simulation du système avec MARZHIN. Pour ce faire, l'outil fournit une transformation de modèle AADL vers le formalisme d'entrée de MARZHIN. Le sous-ensemble AADL supporté concerne les composants processeur, thread, les ports d'événements, les données (variable partagée), les accesseurs de données, les appels de sous-programmes et les constructions de l'annexe comportementale pour les threads et les sous-programmes. Un sous-ensemble des propriétés AADL standards spécifiques à ces composants est aussi utilisé pour la transformation.

5.2.3 Granularité des composants et niveau d'abstraction

Le tableau 5.2 synthétise l'ensemble des composants AADL et leur rôle au sein des outils d'analyse que nous venons de présenter. Nous constatons que l'ensemble des composants concrets (logiciels et matériels) sont utilisés. Un sous-ensemble d'attributs spécifiques est retenu en fonction du type d'analyse effectué. Ainsi, ce sous-ensemble du langage AADL observé définit le niveau de détails requis pour la description des composants AADL pour l'analyse des systèmes du domaine TR^2E. Ceci nous montre le pouvoir d'expression du langage. Un niveau de détail similaire doit pouvoir être exprimé par le niveau d'abstraction intermédiaire de notre modèle de code complet.

Rappel. Le périmètre d'étude (présenté en introduction de ce mémoire) que nous avons défini pour notre approche ne concerne pas les systèmes adaptatifs et/ou tolérants aux pannes. Ainsi, pour la suite, nous ne tiendrons pas compte des constructions pour les modes et celles de l'annexe de modélisation des erreurs.

Composants AADL	Rôle	Utilité pour l'analyse
Système	Hiérarchie	Nommage - traçabilité
Processus	Hiérarchie - stockage ressources	Nommage - traçabilité
Thread	Envoi/réception/traitement messages	Ordonnancement - comportement
Sous-programme	Unité de calcul	WCET - ordonnancement - comportement
Donnée	Ressource	Typages - mémoires - flots
Processeur	Ressource d'exécution	Ordonnancement - énergie
Mémoire	Ressource	Ordonnancement - Mémoire
Bus	Canal de communication	Bande passante
Ports	Interconnexion de composants	Mémoire - flots
Flots	Flots d'exécution	Flots
Modes	Reconfiguration	Tolérance aux pannes
Annexes	Intégration langages externes	Comportement - erreurs
Propriétés	Attributs non fonctionnels	*tous types*

TABLE 5.2 – Synthèse des composants AADL requis pour l'analyse

Dans la section suivante, nous nous intéressons à présent aux éléments du langage AADL utilisés pour la production du code source des composants applicatifs du système.

5.3 Composants requis pour la génération de code

Les travaux de [Zalila, 2008] et l'annexe de génération de code AADL [SAE Aerospace, 2009c] ont montré les avantages (simplicité vis-à-vis des autres formalismes de description plus généralistes) de la production automatique de code source d'un système critique à partir de sa description architecturale AADL. Pour permettre la transformation des composants logiciels vers les artefacts d'un langage de programmation, la description du système doit exprimer des informations détaillées sur la structure externe et interne (interface, interaction inter-composants, séquence d'appels de sous-programmes, variables, initialisation des données, etc.), sur la configuration et le déploiement du composant.

L'annexe de génération de code AADL propose des patrons de génération pour les composants architecturaux vers les langages de programmation Ada et C. Aucune directive n'est cependant exprimée concernant les constructions de l'annexe comportementale dont nous présentons les bénéfices dans la section 5.4 de ce chapitre. Dans le contexte de nos travaux, nous nous intéressons, dans un premier temps, à la génération de code source vers le langage Ada respectant le profil Ravenscar de façon à pouvoir garantir la production d'un code analysable et correct-par-construction. La généralisation de notre approche pour les langages de programmation impératifs (par exemple, le langage C/POSIX) est discutée comme perspective future dans la conclusion de ce manuscrit (chapitre 11).

Pour réduire la distance sémantique entre modèles d'analyse et l'implantation produite, il est nécessaire de pouvoir exprimer explicitement les liens entre les composants de modélisation et les contructions relatives à leur implantation. Dans notre approche, nous visons la définition d'un modèle intermédiaire, appelé modèle de code avec un niveau d'abstraction proche de celui d'un langage de programmation - *i.e* avec des composants dont la sémantique et le comportement sont équivalents voir très proches des constructions du langage de programmation. Pour ce faire, nous devons déterminer le sous-ensemble des composants logiciels présentant cette dernière caractéristique, puis nous intéresser aux composants essentiels pour

la génération, ayant une forte sémantique et nécessitant une transformation. Nous nous appuierons aussi sur les constructions offertes par l'annexe comportementale pour préciser le comportement des composants de ce sous-ensemble. Ces éléments permettront d'exprimer la granularité de la description des composants de notre modèle de code qui permettent une démarche de production automatisée.

5.3.1 Composants concrets pour la génération

Historiquement, le langage AADL est basé sur le langage de description d'architecture METAH et a été enrichi afin de faciliter la modélisation et l'analyse des systèmes TR^{2}E. Différentes contributions sont notamment issues des retours d'expérience et d'utilisation des acteurs de la communauté AADL sur les langages de description d'architectures (FRACTAL ou ACME), la modélisation UML et les langages de programmation pour système embarqué tel Ada, particulièrement connu pour son emploi dans l'implantation de systèmes critiques. Ainsi, de nombreux éléments du langage AADL ont pour origine certains mécanismes ou artefacts du langage Ada (par exemple, paquetage et visibilité de composants, tâches, génériques, etc). Le tableau 5.3 présente les composants AADL ayant une équivalence sémantique (voir aussi structurelle) directe avec ceux du langage Ada.

Composants AADL	**Eléments du langage Ada**	**Fonction**
Paquetage	Paquetage	Nommage - encapsulation - visibilité
Thread	Tâche	Fil d'exécution
Sous-programme	Sous-programme	Unité de calcul - comportement
Données	Variables	Typages - stockage de données
Données partagées - Accès donnée et sous-programme	Objets protégés	Typages - stockage de données - concurrences
Prototypes	Génériques	APIs génériques & configurables
Propriétés Annexes	Pragma, commentaires, annotations (ex : SPARK)	Configuration - traçabilité - vérification formelle

TABLE 5.3 – Equivalences entre composants AADL et éléments du langage Ada

Des travaux menés par notre équipe ont montré que ce sous-ensemble était sastisfaisant pour la génération de systèmes critiques dans les langages Ada/Ravenscar et C/POSIX [Zalila *et al.*, 2008; Lasnier *et al.*, 2009]. Notamment, nous appliquons les restrictions architecturales et comportementales (définies chapitre 3, section 4.4.2) sur nos patrons de génération pour garantir à la génération la conformité du code Ada produit vers le profil Ravenscar.

Rappel. Le profil Ravenscar limite l'usage du langage Ada aux seules constructions qui garantissent l'analyse statique d'ordonnancement, l'absence d'interblocage et une borne temporelle d'inversion de priorités entre les tâches.

Cas particulier des ports AADL pour la communication inter-thread

Les ports et les connexions du langage AADL expriment les liens de communication et les données échangées entre les composants. Ils facilitent la compréhension, l'analyse et l'implantation d'outils d'analyse (par exemple, analyse de flots d'exécution, des canaux de

communication inter-processus, etc). L'inconvénient est qu'un tel composant n'a pas d'équivalent sémantique explicite et/ou unique dans un langage de programmation. En fonction de la sémantique, des propriétés et des politiques de gestion des files d'attente des ports spécifiés, ce dernier peut se traduire par une interruption, un signal, un message, un tampon de communication ou une structure de données plus complexe nécessitant une ou plusieurs transformations par le compilateur pour son implantation.

Dans notre contexte, l'emploi du profil Ravenscar limite fortement les communications entre les tâches Ada. En effet, seuls les objets protégés à une seule entrée sont autorisés et l'attente de plusieurs sources d'événements simultanément pour une tâche est interdite. Ainsi, transformer les ports et les connexions AADL pour implanter la communication des tâches en Ada/Ravenscar consiste à élaborer un mécanisme de communication qui permet à la fois d'effectuer une seule attente mais aussi de recevoir plusieurs événements à partir d'une spécification autorisant plusieurs événements sources (les ports de l'interface du thread) et plusieurs files d'attente (chaque port AADL dispose de sa propre file d'attente).

Pour ce faire, [Zalila *et al.*, 2008][chapitre 6, section 6.4.2] propose une implantation des ports en Ada (intégrée à l'intergiciel PolyORB-HI), appelée *Global Queue* (file d'attente globale de messages) dont l'architecture est basée sur une table constituée d'une concaténation de tableaux circulaires. Chacun de ces tableaux circulaires représente une file d'attente d'un port pour une tâche donnée. Cette architecture a été implantée sous la forme d'un objet protégé manipulé à l'aide de procédures et de fonctions dont l'exécution est déterministe (constant ou en *O(N)* où *N* est la taille de la donnée spécifiée par le port). Des structures de données supplémentaires complètent l'architecture de la file globale et implante les différents comportements des ports d'un thread (politique de gestion des files, priorité de livraison...) spécifiés à l'aide des propriétés du standard AADL.

Composants pour la modélisation de la *Global Queue*

Pour renforcer la cohérence entre modèle d'analyse et code généré, nous avons choisi de modéliser l'implantation de la *Global Queue* décrite ci-dessus. Notre processus de raffinement sera chargé de transformer les ports et les connexions vers un assemblage de composants architecturaux et comportementaux modélisant la structure et le comportement de la file globale. Les éléments produits seront intégrés de manière automatique aux interfaces des threads de la description initiale.

Les équivalences entre les composants de données partagées AADL et les objets protégés Ada (voir tableau 5.3 précédent) nous permettent de modéliser l'architecture de la file globale. Nous utilisons les constructions des langages d'action et d'expression de l'annexe comportementale pour modéliser le comportement des procédures et des fonctions pour la manipulation de la file. Les composants architecturaux et comportementaux utilisés sont détaillés dans le tableau 5.4. Nous précisons aussi le rôle de chaque élément. Enfin, les règles de transformation pour la production d'une file d'attente globale de messages sont présentées dans le chapitre 7.

Ainsi, la transformation des ports d'un composant vers une *Global Queue* préserve le comportement spécifié par l'utilisateur, autorise l'analyse (par construction respectant le profil Ravenscar) et réduit la sémantique entre le modèle du système et son implantation.

Composants AADL	Rôle	Constructions Ada
Donnée partagée	Structure globale	Objet protégé
Accès aux composants de données	Exclusion mutuelle sur l'accès à l'objet	Sémantique de l'objet protégé
Propriétés AADL : `Concurrency_Protocol => Priority_Ceiling` `Priority => <integer>`	Protocole seuil plafond et priorité (garantie l'inversion de priorité)	Pragma `Ravenscar` et pragma `Priority` de l'objet protégé
Sous-programmes	Comportement et manipulation des données	Procédures/Fonctions
Accès aux sous-programmes et propriété AADL `Access_Right`	Exclusion mutuelle sur l'objet	Sémantique de l'objet protégé
Annexe de modélisation des données	Structures de données	Typages et initialisation des données
Annexe comportementale (sous-programme)	Comportement des routines	Implantation des procédures

TABLE 5.4 – Composants AADL pour la modélisation de la *Global Queue*

5.3.2 Granularité des composants architecturaux pour la génération

Le tableau 5.5 regroupe les observations effectuées dans les sous-sections précédentes et présentées dans les tableaux 5.4 et 5.3. Il présente l'ensemble des composants architecturaux AADL logiciels impliqués pour la génération de code vers les contructions du langage Ada/Ravenscar.

Composants AADL	Eléments du langage Ada	Fonction
Paquetage	Paquetage	Nommage - encapsulation - visibilité
Système	-	Nommage - hiérarchie
Processus	-	Nommage
Thread	Tâche	Fil d'exécution
Sous-programme	Sous-programme	Unité de calcul - comportement
Données	Variables	Typages - Stockage de données
Données partagées - Accès donnée et sous-programme	Objets protégés	Typages - stockage de données - concurrences
Prototypes	Génériques	APIs génériques & configurables
Propriétés Annexes	Pragma, commentaires, annotations (ex : SPARK)	Configuration - traçabilité - vérification formelle
Annexe de modélisation des données	Types de données et valeurs initiales	Typages et initialisation de données

TABLE 5.5 – Composants AADL architecturaux pour la génération de code vers le langage Ada

Nous y ajoutons arbitrairement les composants `systèmes` et `processus`. Ces éléments n'ont pas d'équivalent concret vers une construction du langage Ada. Cependant, le composant processus ajoute des informations de nommage qui nous permettent d'identifier les composants applicatifs et intergiciels produits. Le composant système permet de hiérarchiser la description architecturale et de déterminer uniquement les instances de composants qui constituent l'application finale. Il sert par conséquent à piloter le processus de génération de

code et nous assure de ne pas générer des composants applicatifs et intergiciels ne faisant pas partie de l'application finale et pouvant être source d'erreurs lors de son déploiement.

Ce sous-ensemble du langage AADL définit le niveau de détail que doit exprimer la description architecturale du système pour permettre la production automatique de l'application. Les constructions de l'annexe comportementale, que nous présentons dans la section suivante, permettent de compléter ce sous-ensemble pour préciser le comportement des composants à travers la spécification d'action et d'expression proche des constructions des langages de programmation impératifs.

5.4 Niveau de raffinement et spécifications comportementales

Rappel. L'annexe comportementale (présentée chapitre 4) permet, à travers la définition d'automates à états/transitions et l'utilisation de langages d'interaction, d'action et d'expression, de spécifier le comportement de tout type de composant AADL. Ce formalisme est ainsi utilisé comme langage pivot pour traduire le comportement des composants vers des formalismes tiers usités par les outils de *model checking* ou de simulation. Les langages d'action et d'expression proposent des éléments de langage (boucles, affectations de valeur, appels de sous-programmes, etc.) ayant la même sémantique que ceux issus d'un langage de programmation impératif classique.

Dans cette section, nous présentons nos patrons de modélisation comportementale définis pour l'analyse et la génération de code. Nous nous intéressons uniquement aux composants logiciels qui traduisent le comportement de l'application modélisée à savoir les sous-programmes et les threads. L'utilisation de spécifications comportementales au sein de notre approche permet de raffiner certains composants de la description architecturale initiale, de modéliser et d'intégrer les ressources intergicielles - *i.e.* l'exécutif AADL. Dans les sous-sections suivantes, nous présentons la structure et les éléments de langage de l'annexe sélectionnés. Puis, nous discutons des restrictions définies sur ces automates pour garantir l'analyse statique du système modélisé et des bénéfices des langages de l'annexe pour la génération de code.

5.4.1 Description comportementale des composants threads

Analyse du cycle de vie du thread

Le thread est un composant majeur d'une description architecturale car il est le seul à traduire implicitement les fontionnalités de l'application (implicite au sens où il spécifie une séquence d'appels de composants sous-programmes). Une section entière du standard AADL [SAE Aerospace, 2009b][section 5.4] est consacrée à la description du cycle de vie du composant thread et précise sa sémantique. La figure 5.1 présente l'automate du cycle de vie spécifié par le standard [SAE Aerospace, 2009b][section 5.4.1] :

1. L'état de départ d'un *thread* est "arrêté" (*halted*).
2. Lorsque celui-ci est chargé, il effectue une unique fois une phase d'initialisation.
3. Le *thread* est associé à un *mode* du système. S'il n'appartient pas à la configuration initiale du système (*initial mode*), il attend l'activation de son mode.

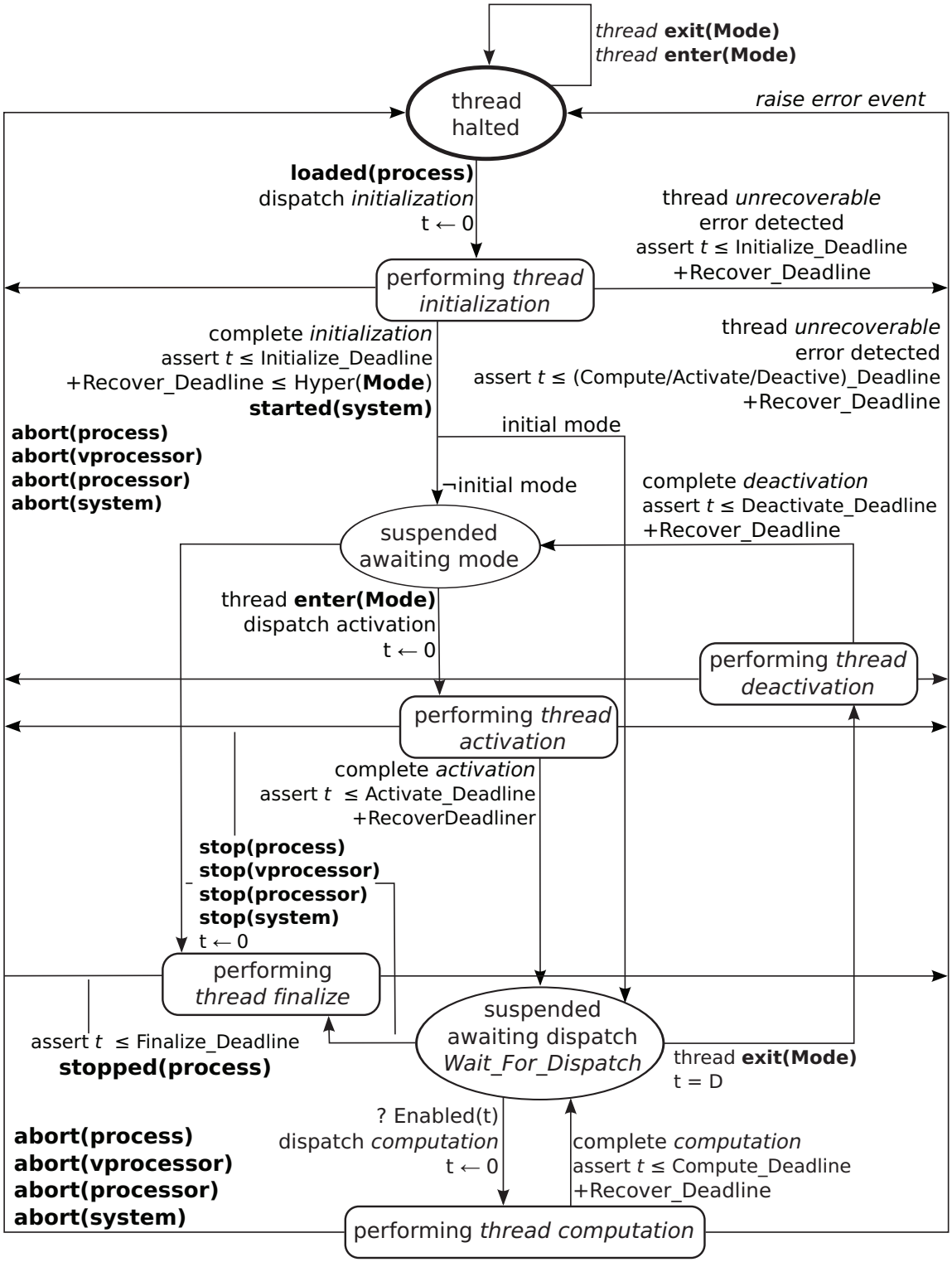

FIGURE 5.1 – Automate du cycle de vie du composant thread (fil d'exécution)

4. Il passe dans un état suspension (*awaiting dispatch*) en attente du déclenchement de son exécution.
5. Si les conditions spécifiées par la fonction *Enabled* sont validées, le *thread* est déclenché (*dispatch*), passe dans un état où il effectue son exécution et réalise toutes les actions qui y sont associées.
6. Une fois son exécution terminée, il retourne dans l'état de suspension en attente du prochain déclenchement de son exécution.

Dans le cas d'une erreur (faute du composant, échéance manquée) ou sur la réception d'une commande d'arrêt (*stop*, *abort*, *exit*), le thread retourne dans l'état "arrêté".

Patron de modélisation comportementale du thread

L'exemple de code AADL 5.1 présente la description architecturale du thread **External_Event_Source** de l'utilisateur et l'automate comportemental généré conformément à notre patron. Sur l'exemple (partie haute), nous pouvons observer la spécification des propriétés temporelles (non fonctionnelles) du thread (période de 5000 millisecondes, échéance de 100 millisecondes...).

Exemple 5.1 – Description architecturale et comportementale d'un thread périodique

```
1   -- user declarative part

3   thread External_Event_Source
      properties
        Activate_Entrypoint_Source_Text    => "Event_Source.Init";
        Compute_Entrypoint_Source_Text     => "Event_Source.New_External_Event";
        Recover_Entrypoint_Source_Text     => "Event_Source.Recover";
8       Dispatch_Protocol                  => Periodic;
        Period                             => 5000 ms;
        Deadline                           => 100 ms;
        Compute_Execution_Time             => 0 ms .. 10 ms;
    end External_Event_Source;
13
    -- generated part

    thread implementation External_Event_Source.impl
      subcomponents
18      Error : data PolyORB_HI_Errors::Error_Kind;

      annex behavior_specification {**
        states
             stInit : initial state;
23       stDispatch : initial complete state;

        transitions
          tInit : stInit -[]-> stDispatch { External_Event_Source_Entrypoint! };
                                          -- call activate entrypoint
28
          tDispatch : stDispatch -[on dispatch sysDispatch]-> stDispatch {
            Event_Source::New_External_Event!;
            -- call compute entrypoint

33          PolyORB_HI_Generated_InS_AADL_Runtime::Send_Output!(Error);

            if (Error /= PolyORB_HI_Errors::Error_Kind.Error_None)
                  Event_Source::Recover_Entrypoint!
                -- call recover entrypoint
38          end if
          };
      **};
    end External_Event_Source.impl;
```

L'automate généré (partie basse de l'exemple) contient deux états : `stInit` définit l'état `initial` du processus avant la phase d'initialisation et `stDispatch` représente son état de suspension (`complete`). La transition de l'état initial à l'état de suspension correspond à la phase d'initialisation du composant. Lors de celle-ci, le sous-programme `External_Event_Source_Activate_Entrypoint` spécifié par l'utilisateur est exécuté une seule fois. Ensuite,

l'automate passe dans une boucle infinie (rebouclement sur l'état `stDispatch`) et effectue l'exécution du sous-programme `External_Event_Source_Compute_Entrypoint` à chaque déclenchement périodique du composant. Ce sous-programme, spécifié par l'utilisateur, encapsule l'implantation du comportement applicatif du thread - *i.e* sa fonction de travail.

Discussion à propos de l'automate réduit

L'automate que nous proposons est volontairement réduit pour respecter les restrictions que nous avons définies (chapitre 3, section 4.4.2). Analysons sa structure :

- *Etats :* l'automate n'est composé que de deux états : l'état initial `stInit` et l'état de suspension `stDispatch`.
- *Transitions* : les transitions `tInit` et `tDispatch` expriment les changements d'état du thread. La première correspond à la phase d'initialisation du thread et déclenche l'exécution du sous-programme spécifié par la propriété `Activate_Entrypoint`. A l'aide de la spécification comportementale, nous rendons l'appel de ce sous-programme explicite.
- *Déclenchement :* Le déclenchement du thread est modélisé par la clause `on dispatch sysDispatch` conforme au standard. Le port spécifique `sysDipatch`[3] est ajouté à l'interface du composant et modélise l'événement déclencheur de l'exécution du composant.
- *Fonction :* L'exécution de la fonction de travail du thread, partie applicative spécifiée par l'utilisateur à l'aide de la propriété `Compute_Entrypoint`, est modélisée explicitement par un appel de sous-programme à chaque exécution du thread.

La même structure pour l'automate du thread sporadique a été définie. Seule change la sémantique du déclenchement de son exécution. Les automates réduits que nous proposons pour modéliser le comportement des composants threads restent conformes au cycle de vie décrit par le standard AADL (voir figure 5.4.1).

Remarques. Les états d'exécution définis par l'annexe comportementale (chapitre 4) n'interviennent pas dans l'automate du thread (l.21-23, exemple 5.1). L'expression des actions attachées à la transition de l'état `stDispatch` (l.30-38) suffit à décrire l'exécution du thread lors de son déclenchement.

Eléments d'analyse pour la vérification

La description architecturale initiale du composant thread présentée dans l'exemple 5.1 (partie haute) spécifie les propriétés non fonctionnelles du thread (par exemple, les propriétés temporelles priorité, échéance, WCET, etc.) utilisées par les outils d'analyses (analyse d'ordonnancement...). Nous avons contraint la structure de notre automate pour qu'ils respectent les restrictions définies pour la réalisation des systèmes critiques et les recommandations du profil Ravenscar (section 4.4.2). Ainsi, par construction, notre automate garantit l'analysabilité statique de l'application. Pour nous assurer de cette propriété, nous pouvons à présent vérifier les restrictions comportementales portant sur l'unique point de suspension d'une tâche (restriction RC1, issue du profil Ravenscar) modélisée par l'absence d'état `final` dans l'automate, et l'exécution infinie d'une tâche (restriction RC2, issue du profil Ravenscar) modélisée par la transition `tDispatch` ayant pour source et destination l'état de suspension `stDispatch` (boucle).

3. Le port `sysDispatch` est l'unique port intervenant dans notre modélisation, il modélise l'événement du système pour le réveil d'une tâche.

La restriction RC2 pouvait auparavant être vérifiée sur la description architecturale (restriction RA2 que nous avions définie) par l'interprétation de la propriété `Dispatch_Protocol` renseignant si le thread était de type sporadique ou périodique. Cependant, la spécification du comportement du thread à l'aide d'une simple propriété (`Compute_Entrypoint`) ne permettait pas de vérifier l'adéquation des éléments spécifiés par ces propriétés. Par conséquent, l'automate que nous proposons permet d'affiner l'analyse.

Le chapitre 9 présente comment nous exprimons ces restrictions architecturales et comportementales à l'aide d'un langage d'expression de contraintes pour la vérification de notre modèle de code complet.

Eléments pour la génération de code

Exemple 5.2 – Thread périodique généré dans le langage Ada

```
1  -- (specification)
   package External_Event_Source_Task is
     Dispatch_Offset : constant Ada.Real_Time.Time_Span := Ada.Real_Time.To_Time_Span(0.0);
4    Task_Period     : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(700.0);
     Task_Deadline   : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(300.0);
     Task_Priority   : constant System.Any_Priority := 2;
     Task_Stack_Size : constant Natural := 100000;

9    task type External_Event_Source_Impl is
       pragma Priority      (Task_Priority);
       pragma Storage_Size (Task_Stack_Size);
     end External_Event_Source_Impl;
   end External_Event_Source_Task;
14
   -- (body)
   package body External_Event_Source_Task is

     Next_Deadline_Val : Time;
19
   task body External_Event_Source_Impl is
      Next_Start : Ada.Real_Time.Time;
      Error : PolyORB_HI_Errors;
   begin
24
      External_Event_Source_Entrypoint;                                    -- call the initialize entrypoint

      Suspend_Until_True (Task_Suspension_Objects (Entity));               -- wait for the system initialization

29    delay until (System_Startup_Time + Dispatch_Offset);                 -- wait startup time
      Next_Start        := System_Startup_Time + Dispatch_Offset + Task_Period;   -- compute next period
      Next_Deadline_Val := System_Startup_Time + Dispatch_Offset + Task_Deadline; -- compute next deadline

      loop    -- main loop
34       Event_Source.New_External_Event;                                  -- call the compute entrypoint

         Error := PolyORB_HI_Generated_InS_AADL_Runtime.Send_Output;       -- call the AADL runtime service

         if Error /= PolyORB_HI_Errors.Error_None then
39          Event_Source.Recover;
         end if;

         delay until Next_Start;                                           -- wait next start
         Next_Start        := Next_Start + Task_Period;                    -- compute next start time
44       Next_Deadline_Val := Ada.Real_Time.Clock + Task_Deadline;         -- compute next deadline
      end loop;
   end External_Event_Source_Impl;

   end External_Event_Source_Task;
```

L'exemple de code 5.2 montre le code Ada généré à partir de la spécification architecturale et comportementale du thread périodique de l'exemple AADL 5.1. La spécification architecturale du composant permet de générer la structure complète d'une tâche Ada périodique. Les propriétés `Period`, `Deadline` et `Priority` du modèle AADL (l.9-11, exemple 5.1) permettent la génération des constantes pour la configuration de la tâche (l.4-7, exemple 5.2). La structure de l'automate réduit (états, transitions, conditions de déclenchement) nous permet de générer le comportement de la tâche (initialisation, calcul de la prochaine période, etc.). Enfin, la

partie fonctionnelle (applicative) de la tâche (l.34-40, exemple 5.2) est assurée par l'appel aux sous-programmes spécifiés par l'utilisateur à l'aide de propriétés AADL (l.5-7, exemple 5.1) et explicités dans l'automate généré (l.25 et 26-38, exemple 5.1). Ces observations montrent la mise en relation simple des éléments de notre automate avec les constructions du langage Ada.

Les règles de transformation d'un composant thread vers une tâche Ada sont présentées dans le chapitre 8. La sous-section présente la description comportementale des composants sous-programmes AADL et leur apport pour l'analyse et la génération de code.

5.4.2 Description comportementale des composants sous-programmes

Modélisation de l'automate d'un sous-programme

L'exemple de code AADL 5.3 montre la description architecturale et comportementale du sous-programme `New_External_Event`. Sa fonction consiste à déterminer et à envoyer une valeur d'interruption (un entier) selon une condition arbitraire.

La description architecturale du composant spécifie les paramètres d'entrée et de sortie, leur type et leur direction (l.6-9). Aucun sous-composant de donnée (modélisant par exemple une variable locale) n'est déclaré dans l'implantation du sous programme. La fonction de calcul modélisant le comportement du sous-programme est quant à elle décrite uniquement à l'aide de l'automate comportemental (l.22-28).

La structure d'automate que nous proposons est réduite et ne contient qu'un état (`stExec` l.17) et qu'une transition (`tExec` l.20) modélisant l'exécution du sous-programme. Ainsi, à chaque exécution du sous-programme, les actions (l.22-28) attachées à la transition sont réalisées.

Exemple 5.3 – Description architecturale et comportementale d'un sous-programme

```
1  -- Calculate an abitrary criterion for the sending
2  -- condition of external interrupts.

   subprogram New_External_Event
     features
       Interrupt_Count : in out parameter Interrupt_Counter;
7      Criterion       : in out parameter Base_Types::Natural;
       Divisors        : in     parameter Divisors_Array;
       Divisor         : in out parameter Divisor_Type;
     properties
       Compute_Execution_Time => 0 ms .. 5 ms;
12 end New_External_Event;

   subprogram implementation New_External_Event.impl
   annex behavior_specification {**
    states
17    stExec : initial final state;

    transitions
     tExec : stExec -[ ]-> stExec
      {
22      if (Criterion mod Divisors[Divisor] = 0)
          Divisor := Divisor + 1;
          Interrupt_Count := Interrupt_Count + 1;
        end if;

27      -- Evaluate the sending condition
        Criterion := Criterion + 1;
      }
   **};
   end New_External_Event.impl;
```

Discussion autour de l'automate

L'automate comportemental décrit dans l'exemple 5.3 est conforme à la syntaxe et à la sémantique définies par l'annexe AADL. La fonction calculatoire pour déterminer si la valeur d'interruption doit être incrémentée est spécifiée à l'aide des langages d'**interaction** (accès aux paramètres d'entrée/sortie), d'**action** (structure conditionnelle) et d'expression (expressions arithmétiques et logiques) définis par l'annexe comportementale.

Eléments d'analyse pour la vérification

L'objectif initial de l'annexe comportementale est de fournir un prototype réaliste du comportement du sous-programme pour les outils d'analyse. La description architecturale permet l'étude de la structure interne du sous-programme - *i.e.* des sous-composants modélisant les variables d'instances de données utilisées par le sous-programme. Cette étude permet notamment d'analyser l'espace mémoire requis par celui-ci. La propriété `Compute_Execution_Time` spécifie le WCET du sous-programme, information utilisée pour les analyses temporelles. Ainsi, il est possible d'estimer (de manière pessimiste) le pire temps d'exécution d'une tâche en sommant les WCET des sous-programmes spécifiés dans sa séquence d'appels.

La description comportementale permet d'affiner ces analyses en explicitant les interactions, les opérations d'affectation, les tests conditionnels et les instructions de boucle lors de l'exécution du programme. Ainsi, il est possible d'exprimer de manière moins pessimiste le calcul du pire temps d'exécution.

De plus, nous présentons dans [Lasnier *et al.*, 2010] le raffinement du niveau d'abstraction d'une description architecturale à l'aide de l'annexe comportementale. Notamment, nous détaillons la modélisation d'une section critique par l'utilisation des primitives de prise/relâchement de verrous au sein d'un sous-programme. Ainsi, le niveau d'abstraction initial induit de la description architecturale (appels de sous-programme au sein d'un thread) se retrouve réduit par l'utilisation de primitives spécifiques au sein de l'automate comportemental du sous-programme, affinant la spécification d'une section critique et l'estimation plus précise du temps passé dans celle-ci. Ce raffinement rend moins pessimiste l'estimation du pire temps d'exécution du sous-programme et, par conséquent, celui de la tâche qui l'exécute.

Eléments pour la génération de code

L'annexe comportementale AADL n'est pas vouée à se substituer à un langage de programmation, mais l'étude des éléments de langage qu'elle propose (voir chapitre 4) révèle que certains éléments (constructions) ont une sémantique, voire une syntaxe, équivalente à certaines constructions d'un langage de programmation impératif comme C ou Ada. Dans le tableau 5.6, nous présentons les équivalences entre les éléments de langage de l'annexe comportementale et ceux du langage Ada.

Le langage d'*interaction* permet précisément de modéliser les appels de sous-programmes avec leurs paramètres respectifs. Le langage d'*action* définit quant à lui les structures conditionnelles (`if` ... `elsif` ... `else`) et les boucles de contrôle (`for`, `forall`, `while`, `do` ... `until`) classiques. Enfin, le langage d'*expression* est directement basé sur celui du langage Ada et permet de spécifier des expressions logiques, relationnelles et arithmétiques.

Enfin, l'intégration de l'annexe au sein de la description architecturale du composant permet la manipulation explicite des composants sous-programmes, des paramètres d'interface

Eléments de l'annexe	Eléments du langage Ada	Fonction
Action de communication `spg!(p1,p2)`	Appels de fonctions/procédures	Exécution de routines
Action d'assignement `:=`	Affectation de valeurs	Affectation d'une valeur à une variable
`if ( ... ) ... elsif ( ... ) ... else ... end if`	`if ... then ... elsif ... then ... else ... end if`	Structures conditionnnelles
`for ( ... in ... ) { ... }`	`for ... in ... range ... loop ... end loop`	Compteurs de boucles
`forall ( ... in ... ) { ... }`	`for ... in ...'Range loop ... end loop`	Compteurs de boucles
`while ( ... ) { ... }`	`while ... loop ... end loop`	Boucle avec condition d'arrêt "au début"
`do ... until ( ... )`	`loop ... exit ... when ... end loop`	Boucle avec condition d'arrêt "à la fin"
Paramètres	Paramètres	Paramètres de routines
Composants et sous-composants de données (donnée partagée)	Variables, objet protégé	Manipulation de variables ou d'objet protégé
Langage d'expression	Langage d'expression	Expression calculatoire

TABLE 5.6 – Equivalences entre annexe comportementale et langage Ada

et des composants de données au sein même de l'automate comportemental. Cela renforce la cohérence entre spécifications architecturale et comportementale.

Contributions. Lors de l'étude des premières versions de l'annexe et de son implantation, nous avons relevé des manques et des incohérences concernant le support de certains éléments architecturaux (tableaux, types de données, sous-programmes pour la manipulation des objets protégés...) et de certains mécanismes pour l'utilisation de composants génériques paramétrables (présentés chapitre 4). Nous avons proposé des corrections et des modifications pour améliorer le support de ces éléments, leur sémantique et certaines règles de l'annexe (légale, consistance, nommage, etc.) lors de nos interventions dans le processus de rédaction et de validation de l'annexe ou lors des comités AADL. Certaines de ces contributions ont aussi été présentées à la communauté scientifique lors de conférences internationales [Lasnier *et al.*, 2010; Lasnier *et al.*, 2011a].

La structure des automates que nous avons définie pour les composants thread et sous-programme et les éléments de langage présentés dans le tableau 5.6 constituent le sous-ensemble des éléments de l'annexe comportementale que nous utilisons pour la définition de nos patrons de modélisation pour l'analyse et la génération de code. Ces éléments ont été sélectionnés pour leurs sémantiques équivalentes à des constructions du langage Ada ou parce qu'ils autorisent des transformations simples vers une ou un ensemble de contructions du langage Ada. Ce sous-ensemble d'éléments de l'annexe comportementale nous permet de compléter les descriptions architecturales des composants thread et sous-programme traduisant le comportement de l'application TR^{2}E.

5.5 Synthèse : le profil AADL-HI Ravenscar

Dans ce chapitre, nous avons présenté les éléments architecturaux du langage AADL (composants, propriétés...) exploités par les outils d'analyse (voir tableau 5.2) et ceux impliqués pour la production automatisée du code source de l'application (voir tableau 5.5). Nous avons notamment expliqué comment remplacer les ports AADL et les connexions modélisant la communication entre les composants threads par un assemblage de composants architecturaux et comportementaux exprimant les mêmes propriétés, le même comportement et respectant les différentes restrictions que nous avons définies pour la réalisation des systèmes critiques et le respect du profil Ravenscar (garantissant l'analysabilité statique de l'application).

Ainsi, l'abstraction particulière des ports AADL est raffinée (par une transformation verticale avec préservation des propriétés) en un composant de donnée partagée et des composants sous-programmes modélisant une file d'attente globale de message et les routines pour sa manipulation. La modélisation du comportement de cette file et de ses routines est basée sur une implantation dans le langage Ada (respectant le profil Ravenscar) intégrée à l'intergiciel PolyORB-HI-Ada (un exécutif AADL) supportant les différentes configurations et propriétés des ports AADL. Dans le chapitre 7, nous détaillons comment est réalisée cette transformation par notre processus de raffinement.

Dans la dernière partie de ce chapitre (section 5.4), nous avons détaillé les éléments d'annexe comportementale qui nous ont permis de définir des patrons de modélisation comportementale (des automates) pour les composants thread et sous-programmes. Nous avons montré comment ces patrons préservent ou affinent l'analyse du système et autorisent une démarche de génération de code simple.

La combinaison des éléments architecturaux AADL, des éléments de langage de l'annexe comportementale et des restrictions que nous avons définies pour garantir l'analysabilité et une implantation du système correcte-par-construction (restrictions Ravenscar, restrictions pour la réalisation des systèmes...) définissent un profil de modélisation qui fixe le niveau d'abstraction de notre modèle de code complet analysable, que nous appelons dans la suite de ce manuscrit : le profil AADL-HI Ravenscar. Dans le chapitre 7, nous détaillons comment raffiner les composants architecturaux de la description initiale vers les composants supportés par notre profil.

Dans le chapitre suivant, nous présentons l'utilisation de ce profil pour la définition d'une bibliothèque de composants (architecturaux et comportementaux) modélisant les ressources intergicielles requises par un système TR^{2}E critique.