Développement et validation d’architectures dynamiques

Jean-François Rolland

▶ **To cite this version:**

Jean-François Rolland. Développement et validation d’architectures dynamiques. Génie logiciel [cs.SE]. Université Paul Sabatier - Toulouse III, 2008. Français. tel-00367994

HAL Id: tel-00367994

https://tel.archives-ouvertes.fr/tel-00367994

Submitted on 13 Mar 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Toulouse III **EDMITT**

Formation Doctorale Sécurité et développement du logiciel

Développement et validation d'architectures dynamiques

THÈSE

présentée et soutenue publiquement le 12 décembre 2008

pour l'obtention du

Doctorat de l'Université Toulouse III – Paul Sabatier
(spécialité informatique)

par

Jean-François Rolland

Composition du jury

Président :	M. Peter Feiler	Senior Technical Staff (Carnegie Mellon University)
Rapporteurs :	M. Yvon Kermarrec	Professeur des universités (ENST Bretagne)
	M. Stefan Van Baelen	Senior Researcher (K.U.Leuven)
Examinateurs :	M. Peter Feiler	Senior Technical Staff (Carnegie Mellon University)
	M. Mamoun Filali Amine	Chargé de Recherche CNRS
Invités :	M. Alain Rossignol	ASTRIUM
	M. David Chemouil	CNES
	M. Pierre Gaufillet	AIRBUS
Directeur de thèse :	M. Jean-Paul Bodeveix	Professeur des universités (Université Toulouse III)

Institut de Recherche en Informatique de Toulouse — UMR 5505

Développement et validation d'architectures dynamiques

Jean-François Rolland
Directeurs de thèse : Mamoun Filali, Jean-Paul Bodeveix

Résumé

La notion d'architecture est un concept de base pour le développement de systèmes en général. Dans le cadre de cette thèse, on s'intéresse plus particulièrement au développement d'un logiciel de vol satellite.

Jusqu'à présent, dans un tel cadre, la notion d'architecture est abordée selon un aspect statique dans lequel il s'agit principalement de vérifier que les interfaces statiques des composants de l'architecture étaient respectées. Les aspects dynamiques (propriétés temporelles, dimensionnement, performance, fiabilité) n'étaient généralement pas adressés. Les langages d'architectures (ADL) ont notamment pour vocation d'intégrer ces aspects très tôt dans le processus de développement.

Dans le cadre de cette thèse, nous nous proposons d'étudier le développement et la validation de systèmes dans un contexte temps réel asynchrone. On a choisi d'utiliser le langage AADL pour ses spécificités issues de l'avionique, domaine proche du spatial, et pour la précision de la description de son modèle d'exécution. Le travail de cette thèse se divise en deux axes principaux : d'une part, on étudie l'utilisation du langage AADL dans le cadre du développement d'un logiciel de vol ; et d'autre part, on présente une version réduite du langage AADL, et la définition formelle de son modèle d'exécution à l'aide du langage TLA+.

L'objectif de la première partie est d'envisager l'utilisation d'AADL dans le cadre d'un processus de développement existant dans le domaine du spatial. Dans cette partie, on a cherché à identifier des motifs de conceptions récurrents dans les logiciels de vol. Enfin, on étudie l'expression en AADL des différents éléments de ce processus de développement.

La seconde partie comporte la définition d'un mini AADL suffisant pour exprimer la plupart des concepts de parallélisme, de communication et de synchronisation qui caractérisent AADL. La partie formalisation est nécessaire afin de pouvoir vérifier des propriétés dynamiques. En effet, la définition formelle du modèle d'exécution permet de décrire le comportement attendu des modèles AADL. Une fois ce modèle défini, on peut à l'aide d'un vérificateur de modèles (model-checker) animer une modélisation AADL ou aborder la vérification de propriétés dynamiques. Cette étude a par ailleurs été menée dans le cadre de la standardisation du langage AADL.

Mots clés : langage de description d'architecture, model-checking, modèle d'exécution, propriétés dynamiques.

Le consensus est un horizon, il n'est jamais acquis.
Jean-François Lyotard, La condition postmoderne.

Remerciements

Ce document présente le fruit de trois ans de travail, je profite de cet instant pour remercier ceux qui ont contribué, de près ou de loin à cette thèse. En premier lieu, je tiens à remercier mes directeurs de thèse, Jean-Paul Bodeveix et Mamoun Filali qui m'ont fait découvrir le monde de la recherche. De part leurs connaissances, leurs conseils et leur disponibilité, ils ont très certainement contribué plus que tout autre à l'élaboration de ce travail.

Je veux ensuite remercier David Chemouil du CNES ainsi que Dave Thomas, Ana-Elena Rugina, Luc Planche et Alain Rossignol d'ASTRIUM pour m'avoir permis de mener à bien cette thèse. Les discussions que nous avons eut ont toujours été très enrichissantes.

Merci a Yvon Kermarrec et Stephan Van Baelen, pour l'intérêt qu'ils ont porté mon travail en acceptant d'être rapporteurs de cette thèse. Leur corrections et leurs remarques ont grandement permis d'améliorer ce mémoire.

Je tiens aussi à remercier Peter Feiler ainsi que tous les gens du SEI pour leur accueil chaleureux. Je remercie aussi Peter pour nos nombreuses discussions, toujours agréables, intéressantes et fertiles en idées nouvelles. Cette collaboration a été très encourageante pour moi.

Merci à Pierre Gaufillet, Isabelle Perseil, et Agusti Canals, j'ai eut un grand plaisir à vous rencontrer et à échanger des idées avec vous.

Je remercie aussi toutes les personnes de l'IRIT avec qui j'ai partagé mon quotidien pendant ces années. Je pense notamment aux gens avec qui j'ai partagé mon bureau, Julien, Abbassia, Lei, Marjorie et tous les autres. Une mention spéciale pour Jérome qui m'a accueilli régulièrement et m'a offert de nombreux thés.

Je remercie mes parents ainsi que toute ma famille pour m'avoir soutenu pendant toutes ces années et pour leurs encouragements permanents.

Un grand merci à tous mes amis Agnes, Olivier, Rachid, Karin, Christophe, Nash, Laure, Charlotte, Véro, Le Nain, Juju, Aurélie, Fab, Fred, Guillaume, Guilhem et tous les autres. Tous ces repas, week-end, soirées ont contribué à rendre ces années plus agréables.

Enfin un grand merci à Anne, Franck, et Delphine pour leur soutien constant et pour avoir réussi à me supporter pendant ces trois ans, cette thèse n'aurait pas vu le jour sans eux.

Table des matières

1 Introduction **1**

2 État de l'art **7**

2.1 Les systèmes réactifs 7

2.1.1 Les langages de description d'architecture 8

2.1.2 Modèles d'exécutions 12

2.1.3 Noyaux d'exécution temps réel 15

2.2 Methodes formelles 16

2.2.1 Le B-événementiel 16

2.2.2 Le langage TLA+ 19

2.2.3 UPPAAL et les automates temporisés 22

2.3 Synthèse 24

3 Le langage AADL et son modèle d'exécution **25**

3.1 Présentation du langage 25

3.1.1 Structure générale d'un composant 26

3.1.2 Les composants logiciels 27

3.1.3 Les composants de la plateforme d'exécution 30

3.1.4 Les systèmes 31

3.1.5 Les interfaces et les connections 31

3.1.6 Les modes 35

3.1.7 Propriétés et annexes 36

3.1.8 Exemples d'annexes standard 37

3.1.9 Les additions de AADL V2 38

3.2 Le modèle d'exécution 39

3.2.1 Ordonnancement 39

3.2.2 Initialisation et finalisation d'un thread 40

3.2.3 Communications 41

3.2.4 Le protocole de changement de mode 43

3.3 Synthèse 44

4 L'étude de cas ArchiDyn 45
4.1 Le développement de logiciels de vol 45
4.1.1 Fonctions et spécificités des logiciels de vols 45
4.1.2 Processus de développement 47
4.2 L'étude de cas ArchiDyn 49
4.2.1 Pléiades et son logiciel de vol 50
4.2.2 Modélisations AADL du logiciel de vol 52
4.2.3 Méthodologie et procédé 57
4.3 Impact de cette étude sur le langage 60
4.4 Synthèse 62

5 Un mini AADL pour ArchiDyn 63
5.1 Le langage et son modèle d'exécution 63
5.1.1 Le modèle de threads 64
5.1.2 Le modèle de communication 66
5.1.3 Les modes 69
5.2 Formalisation du modèle en TLA+ 71
5.2.1 Architecture générale et choix de conception 71
5.2.2 Définition d'un ordonnanceur simple 72
5.2.3 Spécification des ports 77
5.2.4 Les données partagées 85
5.2.5 Comportement interne des threads 89
5.2.6 Les modes au niveau du thread 91
5.3 Les modes systèmes 93
5.3.1 Une abstraction des modes systèmes 94
5.3.2 Les modes en UPPAAL 100
5.3.3 Integration des modes dans le système global 106
5.4 Vérification et Prototype 109
5.4.1 Vérification 109
5.4.2 Prototype 111
5.5 Synthèse 112

6 Les modèles ArchiDyn en mini-AADL 113
6.1 Modèle d'analyse d'ordonnancement simple 113
6.1.1 Définition du modèle étudié 113
6.1.2 Analyse du modèle 114
6.2 Introduction des synchronisations 118
6.3 Un exemple d'illustration 120
6.4 Synthèse 122

7 Conclusion et Perspectives 123

A Description des constantes du modèle TLA 127

B Code TLA **131**

C Exemple ArchiDyn en AADL **163**

D Modèle de traduction Acceleo **169**

Chapitre 1

Introduction

Dans cette thèse, on étudie le processus de développement utilisé dans l'industrie spatiale. On se propose d'étudier le développement et la validation de logiciel temps réel critique par une approche dirigée par les modèles. On envisage d'utiliser conjointement à un langage de description d'architecture une notation formelle nous permettant de vérifier des propriétés sur les modèles.

Le domaine spatial

Cette thèse porte sur l'étude du développement des logiciels de vols embarqués sur satellites. Ces logiciels sont des systèmes temps réel critiques destinés à faire fonctionner le satellite. Un logiciel de vol prend en charge les différentes grandes fonctions du satellite : la communication avec la terre, l'exécution de la mission, la gestion interne du satellite (température, alimentation, prise en compte des erreurs). Ce type de logiciel est comparable aux logiciels embarqués développés par l'industrie automobile ou aéronautique. On va retrouver les même types de contraintes (ordonnancement, temps de traitement, cohérence des données...) et le même besoin de sûreté de fonctionnement. On notera toutefois que contrairement aux logiciels aéronautiques les logiciels de vol n'ont pas besoin d'être certifiés. Cependant, les particularités d'un satellite imposent de nombreuses contraintes supplémentaires au logiciel et à son développement. Ces particularités ont plusieurs origines, on citera notamment l'environnement spatial, les possibilités de communications avec le satellite, et son cycle de vie particulier.

L'environnement dans lequel évoluent les satellites est très perturbateur, de nombreux phénomènes tels que les orages magnétiques, les vents solaires, ou des différences de températures élevées peuvent endommager le matériel ou provoquer des pertes de données. Ceci impose aux constructeurs d'utiliser des composants durcis spécifiques au domaine du spatial. La principale

conséquence de l'utilisation de tels composants est une performance très réduite par rapport au matériel disponible au sol (rapport 1 : 20 au niveau de la puissance de calcul et 1 : 100 au niveau des capacités mémoires). Ces contraintes de performances du matériel obligent les développeurs à produire du code très efficace tant en termes de consommation processeur qu'en occupation mémoire.

L'éloignement et la visibilité souvent intermittente du satellite amènent les concepteurs à le doter d'une autonomie lui permettant de réagir aux différents incidents. Des stratégies de reconfiguration et de passage dans un état sûr sont intégrées à la partie gestion des erreurs du logiciel de vol. En cas de problème important, par exemple, un satellite va chercher à maintenir son alimentation en dirigeant ses panneaux solaires vers le soleil, va pointer ses moyens de communications vers une station sol , et attendra que les opérateurs reprennent la main. De plus, une fois lancé, il n'y a plus d'accès physique possible au satellite. Or, des modifications de son logiciel peuvent être nécessaires. La mise à jour du logiciel doit avoir été envisagée pour pouvoir être possible à distance. Ce type de procédure est très délicat et peut être dangereux. Ces contraintes imposent au logiciel un niveau de confiance élevée.

Enfin, le développement d'un satellite est une tâche complexe et très particulière sur de nombreux points. Tout d'abord, il s'agit d'un processus très long (5 à 10 ans pour un satellite d'observation) qui implique de nombreux partenaires (souvent plus d'une dizaine). De plus, les satellites scientifiques ne sont pas produits en série, chacun de ces satellites est donc un prototype. Ces contraintes ont amené les industriels du spatial ainsi que les agences d'États (le CNES en France) à mettre au point un processus de développement standardisé [fSS96] [fSS91]. Ce processus se base sur un cycle de développement en V classique. Il définit quelles sont les entrées et les sorties de chaque phases. Les documents produits servent à la fois de documentation et de contrats entre les différents partenaires. On veut montrer dans cette thèse, comment on peut accompagner ce processus à l'aide d'un langage de description d'architecture. L'utilisation de tels langages permet d'utiliser des méthodes formelles et les outils de vérification associés.

Les langages de decription d'architecture

L'apparition d'UML [Alh98] et son utilisation de plus en plus importante dans l'industrie, nous montre que les approches d'ingénierie guidées par les modèles sont une solution permettant de mieux appréhender la complexité croissante des systèmes informatiques. L'avantage principal de ce type d'approche est de prendre en compte cette complexité dès le niveau de

conception. De plus, l'utilisation de méthodes semi-formelles permet d'introduire progressivement des outils d'analyse et de vérification. Enfin, elles permettent de créer un support de communication entre le monde des développeurs et celui des méthodes formelles.

Jusqu'à présent, ce type de méthodes a, dans la plupart des cas, abordé la notion d'architecture selon un aspect statique dans lequel il s'agissait principalement de vérifier que les interfaces statiques des composants de l'architecture étaient respectées. Les aspects dynamiques (propriétés temporelles, dimensionnement, performances, fiabilité) n'étaient généralement pas adressés dans les méthodes usuelles de conception. Les langages de description d'architecture (ADL), en plein essor, ont pour principale vocation d'intégrer ces aspects très tôt dans le processus de développement. D'ailleurs, le profil MARTE [DTA+08] d'UML 2.0 [OMG07] a été développé pour répondre à ces préoccupations.

Parmi les ADL, AADL [Aer04] (Architecture Analysis and Design Language) a été conçu comme un langage destiné à faciliter la conception et l'analyse de systèmes complexes, critiques, et temps réel comme l'aéronautique, l'automobile et le spatial. Il est issu du langage Meta-H [Ves98] développé par Honeywell, et est standardisé par le SAE (Society of Automotive Engineers). AADL dispose de nombreux avantages qui justifient son intérêt croissant dans l'industrie de l'embarqué. Ce langage a été conçu pour être extensible, soit à l'aide de propriétés soit par définition d'annexes qui permettent de l'étendre en fonction de ses besoins. De plus, les outils du logiciel libre développés pour le langage (OSATE [OSA], TOPCASED [TOP]) facilitent la création de nouveaux outils d'analyse. Enfin, le standard propose une sémantique d'exécution précise, elle permet de définir le comportement d'un système modélisé en AADL, et une base sémantique commune pour les outils d'analyse.

Méthodes formelles

Les méthodes formelles font référence à des techniques et des outils mathématiques permettant de raisonner de manière rigoureuse. Elles permettent de spécifier, concevoir et valider des systèmes logiciels et matériels. On classe souvent les méthodes formelles en deux grandes catégories : les méthodes déductives et les méthodes basées sur les modèles [Mon00].

Dans le cadre des méthodes déductives, un problème de vérification de propriétés est interprété comme un théorème de la forme :

$$\text{Système} \models \text{Propriété}$$

On a donc besoin de modéliser dans une théorie mathématique à la fois le système étudié et la propriété à valider. Des langages comme B [Abr96] ou Z [Spi89] utilisent pour cela la théorie des ensembles, et la logique du premier ordre, HOL (High Order Logic) [GM94] la logique d'ordre supérieur, et Coq [BBC+97] le calcul des constructions inductives. Afin de vérifier que la propriété est vraie pour le système considéré, on doit prouver que le théorème présenté ci-dessus est vrai à partir des axiomes (propriétés de base supposées vraies) et des règles d'inférence du cadre logique choisi. Il existe de nombreux outils permettant d'être assisté dans la réalisation de cette preuve (HOL, Coq, PVS(Prototype Verification System) [COR+95]). L'inconvénient majeur de cette approche est qu'elle nécessite une grande maîtrise des outils mathématiques employés afin de guider l'assistant de preuve dans la démonstration.

Dans les méthodes de vérifications par évaluation sur un modèle (model checking), on doit modéliser le système et la propriété à vérifier (comme dans les méthodes déductives). D'ailleurs, pour la spécification, on peut utiliser les mêmes formalismes mathématiques (on utilise habituellement B avec un assistant de preuve, mais il existe aussi un model checker). L'idée de base est que le "model checker" va générer le modèle de Kripke du système, c'est-à-dire le graphe des états atteignables et de toutes les transitions possibles, il évalue ensuite la valeur de vérité de la propriété sur tous les états. Ceci est possible à la condition que le graphe soit fini. Des techniques d'abstraction complexes permettent cependant de ramener un problème infini à un espace d'états fini. Les problèmes d'espace d'états infinis apparaissent dès que l'on modélise un problème mettant en jeu des données non bornées tel que des compteurs pour les problèmes de synchronisations et plus généralement des horloges pour le problèmes de temps réel. C'est par exemple le cas des automates temporisés [AD94] qui peuvent être abstraits en automates de régions. Le principal avantage de cette technique est qu'elle ne demande aucune interaction avec l'utilisateur. De plus, dans le cas où la propriété est violée dans un état particulier, le "model checker" exhibe une trace menant à cet état. Son principal inconvénient est qu'elle nécessite en général la génération d'un espace d'état qui explose très vite.

Dans cette thèse, nous utilisons TLA (Temporal Logic of Actions) comme langage de spécification de systèmes et plus généralement comme cadre logique. Ce langage permet de décrire à la fois le comportement d'un système et ses propriété dans un seul formalisme. On a choisi ce langage principalement pour :
- sa puissance d'expression basée sur la théorie des ensembles,
- la disponibilité d'outils open source : des analyseurs syntaxique, sémantique, un "pretty printer" et un "model-checker",

– l'expérience d'expression de spécification de mécanismes systèmes comme par exemple : une formalisation de l'API `Win32 Threads` [Lam96], et de manière plus générale l'expression de protocoles centralisés et répartis.

Les deux méthodes présentées précédemment peuvent être utilisées : TLA possède un système déductif et des outils de preuve pour TLA font aujourd'hui l'objet de recherches. TLA possède aussi un model checker ; dans le cas où l'espace d'état est fini, le model checker permet d'analyser des propriétés exprimées dans la logique temporelle de TLA.

Objectif de la thèse et plan du document

L'objectif principal de cette thèse est d'étudier la possibilité d'effectuer des vérifications de propriétés dynamiques sur des modèles de logiciels de vol. Afin de pouvoir vérifier de telles propriétés il est nécessaire de pouvoir décrire le comportement temporel des modèles étudiés. L'objectif préalable à la vérification devient alors la description formelle de la sémantique d'exécution du langage utilisé. Cette thèse a débuté simultanément à une étude ASTRIUM destiné à analyser l'utilisation d'AADL dans leur processus de développement. Cette étude nous a servi de cas d'étude.

Cette thèse est organisée de la manière suivante :

Le chapitre deux présente les éléments bibliographiques, ce chapitre se décompose en deux grandes parties. On commence par montrer quelles sont les difficultés liées à la conception des systèmes réactifs. On introduit ensuite les langages de description d'architecture permettant de maîtriser la complexité liée à ce type de système. On présente dans cette partie les concepts communs à cette classe de langage. Puis, on montre qu'il est nécessaire d'accompagner ce type de langage d'un modèle d'exécution si on veut pouvoir étudier la dynamique des systèmes à analyser. On expose différents modèles d'exécution selon plusieurs niveaux d'abstractions. Enfin pour terminer cette partie, on décrit différents noyaux d'exécutions temps réel. Dans une seconde partie, on présente les méthodes formelles utilisées au cours de cette thèse.

Le chapitre trois est une présentation non exhaustive, mais assez complète du langage AADL. Une première partie présente le langage, ses capacités d'extension et son évolution. La seconde partie présente un ensemble de mécanismes qui définit le modèle d'exécution d'AADL.

Le chapitre quatre se concentre sur l'étude du processus de développement des logiciels de vol satellite. On présente en détail les fonctions et spécificités de ce type de logiciels. Puis on donne une vue globale du processus de

développement employé pour les réaliser. On montre ensuite comment un langage comme AADL peut être utilisé dans un tel cadre.

Dans le chapitre cinq, on définit un mini AADL, il s'agit d'un sous-ensemble d'AADL adapté à nos besoins d'expression et suffisamment simple pour pouvoir être étudié à l'aide de méthodes formelles. On présente ensuite une formalisation du noyau de ce langage en TLA. La gestion de modes étant un problème complexe, on présente cet aspect de manière séparé avant de l'insérer au modèle complet. Enfin, on montre quels types de propriétés sont vérifiables dans le cadre présenté. On propose un prototype d'outil permettant d'utiliser ces résultats.

On reprend dans le chapitre six une partie des modèles développés dans le chapitre quatre. On montre comment ils peuvent être adaptés au mini AADL. Enfin on donne des exemple de vérification de propriétés a l'aide du model-checker TLC.

Dans le chapitre sept, nous concluons la thèse et discutons de quelques perspectives de ces travaux.

Indications bibliographiques et contributions

Différentes parties des travaux présentés dans cette thèse ont été publiées dans plusieurs communications. Une partie de la formalisation TLA de AADL a été présentée dans [RBC+07]. La partie présentant le protocole de changement de modes a été partiellement publiée dans [RBF+08b] et [RBF+08a]. Enfin, l'étude présentant l'utilisation d'AADL dans le processus de développement d'un logiciel de vol a été présentée dans [RDC07], et les évolutions de l'annexe comportementale issues de cette étude ont été présentées dans [FRBF07].

La contribution principale de cette thèse est présentée dans le chapitre cinq. L'utilisation d'un langage formel tel que TLA oblige à être très rigoureux lors de la description d'un système. La description d'une partie modèle d'exécution d'AADL était un objectif. Elle nous a permis de relever des imprécisions dans le standard AADL V1. Cette activité de formalisation ayant eut lieu pendant la phase de définition de AADL V2, j'ai participé au développement du langage notamment sur les parties concernant la gestion des communications par port et la gestion des modes systèmes.

Chapitre 2

État de l'art

Les logiciels de vol sont des systèmes réactifs critiques. Dans cette bibliographie on présente une étude des systèmes réactifs concernant leur description, leur modélisation et leur support d'exécution. Dans la seconde partie, nous présenterons plusieurs méthodes formelles dans le but de prendre en compte l'aspect critique de ces systèmes.

2.1 Les systèmes réactifs

Un système réactif [HP85] réagit de manière continue avec son environnement au fur et à mesure que celui-ci évolue. Il assure que les contraintes définissant le bon fonctionnement du système sont maintenues entre les entrées et les sorties. On les différencie des systèmes transformationnels et des systèmes interactifs : un système transformationnel dispose de toutes ses données au début de l'exécution, il s'arrête à la fin du traitement des données ; un système interactif réagit avec son environnement mais à son propre rythme. Les systèmes réactifs sont en général déterministes, les sorties du système sont entièrement définies par la séquence des événements qu'il a reçue. Il s'agit d'une propriété fondamentale sur laquelle on reviendra dans la présentation des langages synchrones et dans le chapitre 3. Ces systèmes ont en général des contraintes temporelles très fortes, ils doivent réagir au rythme de leur environnement. Un système réactif doit pouvoir enregistrer et traiter un événement avant la prochaine occurrence de cet événement. La conception de ce genre de systèmes se fait très souvent par composition de modules parallèles. Chaque module assure le maintien d'une partie des propriétés entre les entrées et les sorties. De plus, le monde extérieur peut être considéré comme un processus concurrent qui évolue selon ses propres lois. Ainsi, le système réactif le plus simple peut être représenté par deux composants : le système étudié et son environnement. L'étude d'un tel sstème met déjà en jeu du parallélisme.

Deux approches classiques permettent de modéliser des systèmes réactifs, les automates déterministes [Har87], et les langages de haut niveau intégrant des notions de parallélisme [SVNY02]. Dans un automate déterministe, le système est décrit sous la forme d'états et de transitions, l'arrivée d'un événement déclenche une transition. Elle fait changer le système d'état et peut être associée à un comportement. L'avantage de cette approche est qu'elle est très simple, la traduction vers du code peut être assez directe. Ce genre d'automate est bien connu, il permet de faire des vérifications assez facilement. Mais la taille d'un tel automate augmente très rapidement du fait que le langage d'expression est relativement pauvre. Ce type de langage s'adapte très bien à la description d'un système événementiel, un état de l'automate correspond à un état du système et une transition correspond au traitement d'un événement. Mais la représentation et la manipulation des données ne sont pas prises en compte. L'approche par langages de haut niveau nous permet de définir des processus parallèles très facilement, de plus ces langages mettent à notre disposition des mécanismes de synchronisation évolués (sémaphore, moniteurs, conditions). Il est à noté que ces mécanismes ont été définis pour implémenter des systèmes et non pour les valider. Le problème majeur de ces langages se pose lorsqu'ils sont non déterministes, le comportement du programme ne dépendra pas que du code source. Les instants où ont lieu les points de synchronisation sont déterminés à l'exécution en fonction des arrivées de messages.

Dans les parties suivantes, on présente d'une part les langages d'architecture, et d'autre part différents modèles et noyaux d'exécution. Les langages d'architectures permettent de décrire une abstraction des systèmes réactifs, principalement dans le but de modéliser et d'effectuer des vérifications. Les modèles d'exécution définissent les sémantiques sous-jacentes aux langages précédemment présentés. Quant aux noyaux d'exécution, ils définissent des supports d'exécution pour ces langages.

2.1.1 Les langages de description d'architecture

Une architecture est une spécification décrivant les composants d'un système et leurs interactions[GS93]. Une architecture doit permettre d'exprimer les propriétés comportementales d'un système. Un langage de description d'architecture est donc un outil permettant de :

- raisonner sur un système ;
- accompagner le processus de développement.

Les langages orientés objet peuvent être vus comme un premier niveau de langage de description d'architecture. Les objets sont des composants, leurs interfaces sont décrites en termes d'ensemble de méthodes, et les connexions entre objets sont matérialisées par des appels de méthodes. Mais ce formalisme apparaît très vite limité dans le sens où les interfaces ne sont décrites qu'en terme de services offerts, de plus on ne peut pas décrire les propriétés

non fonctionnelles d'un composant. Les langages de description d'architecture permettent de décrire des interfaces et des interactions entre composants de manières plus riches.

Concepts communs aux ADLs

Afin de permettre la description d'architecture plus précise et plus structurée, une nouvelle classe de langages a été développée au début des années 90, les langages de description d'architectures. Le but de ces langages est de permettre la description haut niveau d'une application complète du point de vue de l'analyse, de l'implantation et du déploiement, de centraliser les différentes propriétés, fonctionnelles et non fonctionnelles, des composants et ainsi de faciliter l'analyse des modèles décrits. La définition de langage d'architecture peut être très large, mais on peut citer celle donnée par Garlan et Shaw [GS93] :

"[Software architecture is a level of design that] goes beyond the algorithms and data structures of the computation : designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure ; protocols for communication, synchronization, and data access ; assignment of functionality to design elements ; physical distribution ; composition of design elements ; scaling and performance ; and selection among design alternatives."[1]

Depuis, de nombreux ADLs sont apparus, bien que tous ces langages aient leurs spécificités propres, ils sont tous basés sur des concepts communs, ces concepts permettent de comparer les différents ADLs. Une étude comparée de tous ces langages a été faite dans l'article de Medvidovic [MT97].

Les blocs de base d'un ADL sont les composants (les boites), les connecteurs (les fils), et les configurations d'architectures (le graphe des composants/connecteurs). Les composants sont des entités logicielles ou matérielles disposant d'une interface et d'un comportement. Les connecteurs définissent les interactions possibles entre les différents composants. Et une configuration décrit la coopération entre les différents composants. De plus, un ADL doit permettre de modéliser les interfaces des composants. Enfin, il est important qu'un tel langage soit accompagné d'outils permettant l'édition, l'analyse... Dans ce qui suit, on présente chacun de ces aspects. Leur illustration effective sera donnée ultérieurement à l'aide du langage AADL.

[1] L'architecture logicielle est un élément de conception qui va au-delà des algorithmes et des structures de données : la conception et la spécification de la structure globale du système émerge comme une nouvelle problématique. Les préoccupations structurelles comprennent l'organisation et la structure globale du contrôle, les protocoles de communication, de synchronisation et l'accès aux données ; l'affectation de fonctionnalités aux éléments de conception ; la distribution physique ; la composition des éléments de conception ; la mise à l'échelle et les performances, et la sélection parmi les alternatives de conception.

Les composants

Un composant est une unité de calcul ou de stockage, il peut aussi bien représenter un processus qu'une procédure ou une zone de données partagée. Un composant a une interface, cette interface définit l'ensemble de ses points d'interactions avec l'extérieur, i.e. les services qu'il propose et ceux qu'il requiert. Ces points d'interactions peuvent être hiérarchiques ; AADL définit la notion de groupe de ports. De manière plus générale, la description d'une interface à l'aide de protocoles permet de raisonner sur les comportements des composants. Certains ADLs permettent de modéliser le comportement des composants, Wright [AG97] en CSP [Hoa78], Darwin [MDEK95] en π-calcul [Mil99], Meta-H [Ves98] dans un langage dédié (Control-H), ou par un ensemble de propriétés (UniCon [SDK+95]). Des contraintes ou propriétés sont généralement associées aux composants afin de préciser leur comportement et d'établir des dépendances entre les différentes parties internes du composant. Ainsi, le type d'un composant est généralement défini par son interface et un ensemble de propriétés. Un type de composant peut être implémenté par plusieurs composants différents. Enfin, un composant est amené à évoluer, un ADL doit permettre de maîtriser ces changements. Les mécanismes de sous typage et de raffinement permettent dans une certaine mesure de contrôler cette mise à jour. Intuitivement, ces mécanismes vont assurer que le nouveau composant va posséder au mois les propriétés du composant remplacé.

Les connecteurs

Les connecteurs servent à modéliser les interactions entre composants. Parmi les différents ADLs cités, certains les modélisent explicitement (C2 [MORT96], Wright [AG97], UniCon [SDK+95], ACME [GMW00], SADL [MQR95]), d'autres les restreignent à des connexions élémentaires entre composants, dans ce cas la description des interactions entre composants peut être faite au niveau de l'interface du composant. Les connecteurs sont définis suivant les mêmes caractéristiques que les composants. Ils disposent d'une interface permettant de valider la composition des composants et de définir le rôle des participants à l'interaction. Le type d'un connecteur correspond à une abstraction des mécanismes de communication et de coordination. Comme pour les composants, certains langages permettent de décrire le comportement des connecteurs, généralement dans le même formalisme que pour les composants ; Wright par exemple utilise CSP. Des contraintes peuvent être définies afin de limiter l'utilisation des connecteurs. Enfin, certains ADLs permettent de faire évoluer les connecteurs par conformance de type (Wright [AG97]), par sous typage avec préservation du comportement (Aesop [GAO94]), ou par raffinement (SADL [MQR95]).

Un connecteur définit la composition entre différents composants. Dans

sa forme la plus simple, ce peut être un canal de communication avec un protocole élémentaire d'échange, e.g., un canal CSP. La notion de connecteur permet aussi d'élaborer des compositions plus complexes de composants.

Les configurations

La topologie, ou graphe connecté, des composants forme une configuration de l'architecture. Une configuration permet d'établir que toutes les connexions entre composants respectent leurs interfaces respectives. De plus, la configuration définit le comportement global du système. Ce niveau de conception permet aussi de vérifier que des propriétés non fonctionnelles globales sont bien respectées. La capacité à décrire une configuration dans un ADL peut être évaluée par différents critères.

Un ADL doit fournir une syntaxe simple et intuitive pour décrire la configuration. On doit pouvoir appréhender la structure globale d'une architecture en ne regardant que la configuration. Afin de faciliter la lecture, certains ADLs proposent une vue graphique des configurations.

Lorsqu'on décrit une architecture, on est amené à spécifier des systèmes à différents niveaux de détails, on doit pouvoir décomposer hiérarchiquement une spécification. On distingue deux catégories de composants, les composants primitifs, non décomposable et les composants composites formés d'un ensemble de sous composants.

Une configuration définit l'instanciation des composants et de leurs connecteurs.

Spécificités des différents langages

Bien que tous ces langages cités soient des ADLs, ils ont tous leurs spécificités. Ainsi, Aesop [GAO94] met l'accent sur la réutilisation de composants et introduit la notion de " style " d'architecture. Un style est un motif d'architecture fréquemment utilisé, l'utilisation de styles permet de guider le processus de développement en apportant des solutions génériques pour certaines classes de problèmes. Adage [CS93] est un langage dédié à la description d'architectures avioniques. C2 [MORT96] est aussi un langage dédié qui se concentre sur les besoins particuliers des applications possédant un aspect interface avec les utilisateurs importants. Darwin [MDEK95] est une notation permettant de spécifier des architectures distribuées. SADL [MQR95] met l'accent sur le raffinement entre différents niveaux d'architectures. UniCon [SDK+95] permet de spécifier des architectures logicielles très hétérogènes. Meta-H [Ves98] est un langage développé par Honeywell, il a été utilisé dans de nombreux développements industriels d'applications embarquées avioniques. Wright [AG97] utilise CSP [Hoa78] pour spécifier et analyser les interactions entre composants. Rapide [LKA+95] est accompagné d'outils permettant la simulation d'exécution des architectures décrites. Enfin,

ACME [GMW00] a pour but de définir un langage d'architecture générique, une sorte de langage pivot permettant d'utiliser les spécificités de plusieurs ADLs.

Les outils

Afin d'être réellement utilisé, un langage de description d'architecture doit être accompagné d'environnements aidant au développement. Le premier outil nécessaire est un éditeur permettant la validation syntaxique du code. Par extension, on peut proposer un éditeur graphique lorsque le langage propose une syntaxe graphique. Le développement des ADLs est en partie motivé par le besoin de rassembler de nombreuses informations au sein d'un même modèle. Afin d'exploiter correctement un tel modèle, il faut pouvoir en donner des vues partielles, et permettre ainsi d'extraire aisément l'information souhaitée. Le support du raffinement et de la génération de code sont aussi des aspects qui doivent apparaître dans de tels environnements de développement. Enfin, des outils permettant d'analyser une spécification et de l'animer sont souhaitables. La correction de ces outils repose sur l'existence d'une sémantique bien définie de ces langages. L'utilisation des méthodes formelles permet de définir un cadre sémantique qui établit par construction cette correction. Dans la partie 2.2, nous présentons quelques outils formels que nous avons été amenés à utiliser dans le cadre de cette thèse. La sémantique dynamique d'un langage de description d'architecture définit un modèle d'exécution, dans la partie suivante, on présente quelques-uns de ces modèles.

2.1.2 Modèles d'exécutions

Dans la partie précédente, on a présenté une classe de langages de modélisation, les langages de description d'architecture. L'utilisation de tels langages dans le cadre de développement de logiciel temps réel est motivée par la capacité de détecter des erreurs de conception au plus tôt. En effet, les ADL vont permettre des analyses sur les modèles de conception. La validation d'un ensemble de propriétés dynamiques nécessite de définir avec précision la sémantique d'exécution de ces modèles. Pour cela, on doit définir un modèle d'exécution : une abstraction de la plateforme sur laquelle sera exécuté le logiciel. Un tel modèle nous permettra aussi de générer des simulations d'exécutions. Dans cette partie, on présente différents modèles d'exécutions couramment utilisés dans le domaine de l'embarqué. Nous considérons successivement le modèle d'exécution synchrone, les langages time driven et enfin les langages parallèles classiques.

L'approche synchrone

Le but des langages synchrones est d'arriver à maîtriser la complexité des systèmes réactifs [BB02] [BCE+03]. Pour cela, la modélisation d'un tel système est déterministe : pour une séquence d'entrée donnée, une séquence unique de sortie est générée. Les langages synchrones se basent sur deux hypothèses très restrictives pour atteindre ce but. Les sorties sont instantanées : elles sont synchrones avec les entrées. Il n'y a pas de temps de propagation des messages ; dans la littérature [BG92] [LGLBLM91] [HCRP91], on parle de contrôle immédiat ou de paradigme zéro délai. Ces hypothèses permettent de simplifier l'expression du parallélisme : deux messages peuvent arriver en même temps sur deux composants différents, ils sont alors traités simultanément et leurs résultats sont aussi produits simultanément. Il s'agit là d'une simplification par rapport au modèle usuel d'entrelacement du parallélisme [And92]. De plus, l'évolution du système n'est plus liée au temps physique, mais à la vitesse d'arrivée des évènements. Dans ce genre de langages, on considère que les ressources du système sont infinies. On vérifie a posteriori que l'hypothèse de synchronisme est vérifiée c'est-à-dire que le système est ordonnançable : au sens où tout bloc d'instruction associé au traitement d'un événement pourra être exécuté avant l'arrivée du prochain événement. Le modèle synchrone est à la base de langages ayant une sémantique précise permettant d'envisager la validation de programme [BPPS00] [HR99] ainsi que la génération automatique de code certifié [ITPS08].

Différents langages synchrones ont été développés, ils partagent tous les mêmes hypothèses citées précédemment, mais ont des styles de programmation différents. Ainsi, Esterel [BG92] est un langage impératif alors que Signal [LGLBLM91] et Lustre [HCRP91] sont déclaratifs. Des formalismes graphiques, basés sur les statecharts, tels qu'Argos [Mar91] ou les SyncCharts [And96] ont été développés à partir de ces langages. Des formalismes incluant la notion de mode ont aussi été étudiés [MR98]. L'utilisation de modes permet de décrire des systèmes ou des ensembles de tâches ne se déroulant pas en parallèle, mais séquentiellement. Les développements plus récents des langages synchrones essaient d'assouplir les contraintes du synchrone en définissant des architectures globalement asynchrones et localement synchrones (GALS) [BCG99] [GGpT+03].

Les langages Time driven

Dans les langages time driven, le flot de contrôle est dirigé par une horloge et non plus comme dans le synchrone par des événements. Le programme est divisé en tâches activées périodiquement. Toutes les communications se font en début et en fin de période. L'environnement d'une tâche est gelé au début de sa période et elle communique ses résultats à la fin de la période. Ces contraintes permettent de définir une sémantique d'exécution déterministe

puisque toutes les exécutions suivent le même motif.

Dans le langage Giotto [HHK01], le composant de base est la tâche ; un ensemble de tâches périodiques et concurrentes forment un mode. Un programme Giotto est un ensemble de modes. Les tâches communiquent entre elles par des ports. Les ports sont mis à jour au début (consommation) et à la fin de chaque période (production). À l'intérieur d'un mode, les tâches ne peuvent pas se synchroniser : elles peuvent uniquement communiquer via leurs ports. Chaque mode a aussi une période propre, c'est un multiple de l'hyperpériode des tâches qui le composent. À chaque fin de période de mode, le système peut changer de mode. Lors du changement de mode, des données peuvent être communiquées entre les deux modes, par l'intermédiaire de ports particuliers. Giotto permet de représenter une architecture logicielle et de définir son comportement temporel. La partie fonctionnelle peut être décrite dans un langage tiers, le langage c par exemple. Ce code doit être associé à un des différents éléments du langage :

- tâche : description du traitement des données,
- ports : comportement du port
- mode : condition du changement de mode, gestion du changement de contexte.

Un successeur de Giotto, HTL [GSVK+06] reprend les bases de Giotto en ajoutant une vision hiérarchique. Une tâche peut être raffinée en un ensemble de sous tâches devant respecter un ensemble de contraintes afin que les propriétés temporelles vérifiées au niveau supérieur restent valides.

Il existe pour ce langage un compilateur [HK02]. La compilation se déroule en deux phases. Dans un premier temps, le compilateur génère un code intermédiaire, le E code, indépendant de la plateforme. Le code produit est portable et exhibe, pour un ensemble d'entrées donné, un " minutage " et un ensemble de résultats déterministes. Dans un second temps, le compilateur vérifie que les caractéristiques de la plateforme garantissent la conservation des propriétés temporelles du E code, constituées essentiellement par la périodicité des tâches.

Approche parallèle classique

Cette approche est mise en œuvre par des langages impératifs classiques proposant des mécanismes permettant de prendre en compte les problèmes liés au temps réel. Dans cette classe de langage on peut citer Ada [SVNY02] et son profil Ravenscar [AD03], RTSJ [Wel04]... Il s'agit de l'approche la plus classique, c'est aussi la plus concrète. Un programme est vu comme un ensemble de tâches devant partager des ressources communes (partageables ou non, préemptibles ou non), par exemple le processeur, la mémoire, les périphériques, ... Dans ce modèle une tâche est notamment caractérisée par son mode d'activation :

- périodique : la tâche doit être exécutée à une fréquence fixe ;

– apériodique : la tâche est déclenchée par la réception d'un message ;
– sporadique : la tâche à un comportement similaire à une tâche apériodique mais un délai minimal entre deux activations doit être respecté ;
– tâche de fond : la tâche est exécutée lorsqu'aucune autre tâche n'occupe la ressource processeur.

L'accès au processeur dépend du mode d'activation de chaque tâche.

Les tâches communiquent entre elles principalement par rendez-vous ou par des zones de mémoire partagée. L'accès aux ressources partagées peut être géré par de nombreux protocoles [SRL90]. Un tel cadre permet beaucoup plus de flexibilité que les modèles synchrones ou time driven mais en contrepartie les analyses possibles restent limitées. Le comportement temporel d'une application peut en partie être validé en employant les travaux de la théorie de l'ordonnancement [AB90]. Les résultats de cette théorie permettent de tester l'ordonnançabilité d'une application en fonction du jeu de tâches à exécuter, de leurs caractéristiques temporelles et de l'algorithme d'ordonnancement choisi. L'article fondateur de Liu et Layland [LL73] présente un test d'ordonnançabilité pour un système de plusieurs tâches indépendantes se partageant un seul processeur. Des extensions ont depuis été proposées dans le but de rendre ces tests applicables à des contextes de plus en plus réalistes [KRP+93]. Ainsi, des problèmes comme la dépendance de tâches [SSNB95], la gestion des ressources partagées [SRL90], la gestion des changements de modes [TBW92], [RC04] ont été abordés. Des outils comme Cheddar [SLNM04], times [AFM+03] permettent aujourd'hui de décrire et de valider des applications temps réel en exploitant ces résultats.

2.1.3 Noyaux d'exécution temps réel

Les noyaux d'exécution temps réel définissent une couche d'abstraction entre le matériel et l'application. Ce sont des OS spécialisés, qui implantent les mécanismes de base permettant d'assurer le cycle de vie (création, activation, communication, terminaison) des différentes tâches de l'application. On présente ici les services proposés par les noyaux les plus utilisés dans le domaine du spatial et de l'aéronautique : RTEMS [On-03] et VxWorks [Win93]. L'ordonnancement des différentes tâches est géré par un ordonnanceur préemptif à priorité fixe. Dans le cas où plusieurs tâches de même priorité sont actives au même moment, le noyau peut partager le temps processeur entre ces différentes tâches. La communication peut se faire par variables partagées, par file de messages, par signaux ou événement. VxWorks [Win93] propose aussi d'utiliser des tubes, des sockets ou des appels RPC, notamment pour communiquer via un réseau. La synchronisation et la protection des ressources partagées sont assurées par des sémaphores. Afin d'éviter les problèmes d'inversion de priorités, ces noyaux implantent les protocoles PIP et PCP [SRL90]. Ces protocoles permettent à une tâche qui accède à une ressource partagée d'élever sa priorité si elle bloque ou peut

bloquer une tâche plus prioritaire. Le temps d'attente de la tâche prioritaire est donc ainsi réduit.

Dans ces noyaux, des mécanismes de synchronisation plus évolués sont aussi mis à disposition : les signaux et les événements. Les signaux modifient de manière asynchrone le flot de contrôle d'une tâche. Lorsqu'un signal est reçu, la tâche se suspend immédiatement. À sa prochaine exécution elle exécutera le handler correspondant au signal. Les événements, contrairement aux signaux, sont synchrones. Une tâche est en attente d'un événement, elle se suspend jusqu'à la réception de cet événement. Lorsqu'elle le reçoit elle poursuit son exécution. Les événements permettent de décrire des synchronisations complexes : une tâche peut attendre une combinaison d'événements, par exemple, il est possible de se mettre en attente de l'occurrence de plusieurs événements (alors que classiquement on se met en attente de un événement parmi plusieurs). Enfin, on dispose de timers pour déclencher les tâches. Ces noyaux d'exécutions permettent aussi de gérer les interruptions matérielles et les accès à la mémoire. Des mécanismes de protection ou d'isolation sont aussi proposés par de tels noyaux. La notion de partition permet d'assurer de telles propriétés. Chaque partition définit un espace de mémoire propre (isolation spatiale) et une partie du temps processeur (isolation temporelle). La communication entre partitions est généralement possible mais très limitée.

Bien que chaque noyau d'exécution possède une API propre, ils implantent aussi des API standards comme POSIX [IEE93] ou ARINC [Air97].

2.2 Methodes formelles

Dans les parties précédentes, on a présenté les langages d'architectures comme une technique permettant d'aider au développement de logiciels embarqués. On a ensuite exposé différents types de modèles d'exécutions. On a souligné que la définition précise d'un modèle d'exécution était nécessaire afin d'appréhender correctement le comportement d'une application. On présente ici des outils permettant de définir la sémantique d'un modèle d'exécution ainsi que d'établir ses propriétés.

On présentera tout d'abord le langage B-événementiel [JRCL05] issu de la méthode B [Abr96]. On donnera ensuite un aperçu du langage TLA+ [Lam02], destiné à représenter des système de transition. Enfin, on présentera UPPAAL [LPY95] et les automates temporisés [AD94].

2.2.1 Le B-événementiel

Comme pour la méthode B, le modèle mathématique sous-jacent au B-événementiel est la théorie des ensembles et le calcul des prédicats du premier ordre. Un développement B débute par la construction d'un modèle abstrait qui reprend les spécifications des besoins. Un modèle est caractérisé

par quatre éléments : un nom, une liste de variables d'états, un ensemble de prédicats représentant des propriétés invariantes, et un ensemble de transitions appelées ici événements. Un événement se décompose en trois parties : un nom, un ensemble de prédicats, la garde, et une substitution généralisée[2]. La garde définit la condition nécessaire pour que la transition ait lieu. La transition d'état associée à un évènement est définie par une substitution généralisée. Afin de définir l'état de départ du système, il existe un événement spécial non gardé baptisé `Initialization`. Trois types de substitutions existent afin de représenter la transition associée à un événement : déterministe, non déterministe et vide(`skip`). La transition `skip` ne fait rien mais se termine. Cette transition permet essentiellement de modéliser une transition interne (non observable de l'extérieur) qui sera raffinée ultérieurement. Les substitutions généralisées sont des raccourcis permettant de décrire une relation entre deux états du système sans avoir recours à une notation mathématique. Afin de pouvoir structurer la spécification, on peut décomposer un modèle en sous modèles.

En plus de la description de la partie modèle, afin de définir un système complet on doit préciser un contexte. Un contexte définit un modèle de donnée. Un contexte est défini par un nom, une liste d'ensembles porteurs (ensembles abstraits en B), une liste de constantes et un ensemble de propriétés. Les ensembles porteurs sont caractérisés par leurs noms, ils doivent être indépendants et sont considérés non vides. Les constantes sont définies à l'aide des propriétés qui sont de simples prédicats.

```
MODEL prodcons

SETS

  DATA
; STATE = {empty, full}

VARIABLES

  buffer, bufferstate, bufferc

INVARIANT

  bufferstate : STATE
& buffer : DATA
& bufferc : DATA

INITIALISATION
```

[2]En B, la notion de substitution généralisée formalise et généralise le concept d'instruction classique des langages de programmation

```
   bufferstate := empty
|| buffer :: DATA
|| bufferc :: DATA

EVENTS

produce = /*when buffer is empty*/
  ANY dd WHERE
      dd : DATA
    & bufferstate = empty
  THEN
       buffer := dd
    || bufferstate := full
END

; consume = /*when buffer is full*/
  SELECT bufferstate = full
  THEN
       bufferc := buffer
    || bufferstate := empty
END

END
```

Comme dans la méthode B, le raffinement tient une place centrale dans ce langage. Un modèle abstrait et son contexte doivent être raffinés en modèles plus concrets. Il peut y avoir plusieurs étapes de raffinement jusqu'à arriver à un niveau implémentation. Plusieurs raffinements peuvent satisfaire une spécification. Le raffinement d'un contexte consiste à ajouter des ensembles porteurs et des constantes. Un raffinement de modèle repose sur un ensemble de variables complètement distinct de l'ensemble de variables de l'abstraction. Seul l'invariant de collage [Abr96] peut référencer à la fois les variables de l'abstraction et celle du raffinement. Cet invariant définit la relation entre les deux niveaux d'abstractions. Chaque événement abstrait doit être raffiné par au moins un événement concret. Il est possible que deux évènements abstraits soient raffinés par le même évènement concret. Dans ce cas, la garde de l'événement concret est la disjonction des gardes des événements abstraits et les substitutions généralisées doivent être identiques. D'autre part, Il est possible d'introduire de nouveaux événements en raffinant l'évènement implicite `skip`.

Le langage B événementiel propose un cadre de travail satisfaisant nos besoins de plus il dispose outils de preuves avancés. Mais on a préféré le langage TLA+ en particulier à cause de la liberté d'expression qu'il nous

offre. De plus, dans l'optique de validation automatique de propriétés on préfère l'approche par model checking de TLA+.

2.2.2 Le langage TLA+

TLA+ (Temporal Logic of Action) est un langage de spécification basé sur la théorie des ensembles et la logique temporelle linéaire (LTL) [SC85]. Les principaux opérateurs utilisés sont

- des opérateurs booléens et arithmétiques classiques,
- des opérateurs ensemblistes,
- et des opérateurs temporels.

L'état d'une spécification est défini par la valeur de toutes les variables de la spécification. L'espace d'état d'une spécification est l'ensemble de tous les états possibles de cette spécification. Un prédicat d'état est une simple fonction booléenne sur un état de la spécification. Un prédicat peut être paramétré par une liste de variables. Une action est un prédicat portant sur deux états successifs (courant et suivant). Il s'agit d'une formule décrivant l'état courant du système ainsi que l'état suivant à l'aide de variables "primées". Soit x une variable du programme, x' désigne la valeur de la variable x dans l'état suivant. De manière plus générale une expression entière peut être primée. Une action est donc un prédicat décrivant une relation entre deux états de la spécification. Un comportement est défini comme une séquence d'états telle qu'il y a toujours une action valide entre deux états successifs : ces deux états valident la relation associée à cette action.

Une spécification contient toujours un état initial et une action `next` définissant toutes les transitions possibles, cette action `next` est généralement la disjonction de toutes les transitions atomiques définies. Il s'agit là de la spécification de la partie sûreté qui définie les propriétés toujours vraies. Quant à la vivacité, il est aussi possible de la spécifier en indiquant l'équité forte ou faible de chaque action [Lam02]. Informellement, la notion d'équité permet de spécifier un ordre sur le déclenchement des actions.

TLA+ introduit la notion de modules afin de structurer les spécifications. Un module contient des déclarations de constantes, variables, prémisses, et enfin de prédicats. Les prémisses sont principalement utilisées pour typer les constantes. On définira toujours au moins trois prédicats, un prédicat définissant le type des variables manipulées généralement appelé `TypeInvariant`[3], un prédicat définissant l'état du module lors de l'initialisation généralement appelé `Init`, et un prédicat décrivant les transitions possibles généralement appelé `Next`.

[3] On distingue en général deux types de propriétés dans cet invariant. Les propriétés relevant du typage statique et les propriétés d'états qui résultent de la dynamique du système.

remarque : L'invariant de type en TLA+ est considéré comme une propriété de sûreté qui devra être vérifiée au même titre que n'importe quelle autre propriété. Le typage est donc vérifié dynamiquement à l'exécution (lors de la phase de model-checking). On ne peut pas faire de vérification statique de type lors de la compilation comme dans une approche typée.

```
---------------- MODULE NomModule ----------------
CONSTANTS
  liste_des_constants
ASSUME
  typage_des_constantes
VARIABLES
  liste_des_varaibles
TypeInvariant ≜
  Invariant_de_typage
Init ≜
  Initialisation
Next ≜
  Transitions_possibles
==================================================
```

L'assemblage de modules ce fait à l'aide de deux types de relations existent entre deux modules, l'extension et l'instanciation.

Le mot clef `EXTENDS` est suivi de la liste de modules étendus par le module courant. Une extension permet d'accéder aux variables, constantes et prédicats définis dans le module étendu.

L'instanciation permet de définir plusieurs instances d'un même module. Chaque instance a un comportement identique, mais possède un état propre. Les variables de chaque instance doivent être définies dans le module où a lieu l'instanciation. Au moment de l'instanciation, on assigne des variables du module courant aux variables du module instancié. Cet assignement peut être implicite si le nom de la variable est le même dans les deux modules. Lorsque deux modules différents doivent communiquer, on les instancie avec des variables communes. Les prédicats définis dans une instance peuvent être appelés en utilisant la syntaxe `instance !predicat()`.

Le temps réel en TLA Le principe utilisé par Lamport[Lam05] pour décrire une modélisation temps réel dans un langage standard est extrêmement simple. On va ajouter une variable dont l'évolution représente l'écoulement

du temps, Lamport nomme cette variable *now*. Une opération appelée tick fait évoluer cette variable. Afin de pouvoir exprimer des contraintes sur l'écoulement du temps trois types de timers sont décrits :

- expiration timer : le tick ne change pas la valeur du timer. Il est positionné à une valeur $now + \tau$ et le timeout intervient lorsque $now = timer$.
- count down timer : le tick fait décroître le timer, le timeout a lieu lorsque $timer = 0$
- count up timer : le tick augmente la valeur du timer. Le timeout a lieu lorsqu'il devient égal à une valeur constante prédéfinie.

Dans ses exemples, Lamport utilise principalement des count down timers. Afin de faciliter la validation des spécifications on préfère utiliser de count up ou des count down timers car on peut facilement fixer leurs limites, en effet ils ne peuvent évoluer que entre zéro et la valeur de leur borne. L'évolution de *now* peut se faire de différentes manières, classiquement on la fait évoluer pas à pas par incrément d'une unité. Mais une approche alternative est présentée, en étudiant le système spécifié, on peut calculer les dates auxquelles auront lieu les prochaines actions. On peut donc prédire quand la prochaine action doit avoir lieu. Au lieu de faire avancer *now* d'une valeur prédéfinie fixe, on peut faire des “ sauts ” jusqu'à la prochaine action possible. Cette technique permet de générer moins d'états lors du model-checking. Les timers vont nous permettre d'exprimer des gardes pour le déclenchement des opérations.

Les timers évoluent en même que la variable *now*, ils sont donc modifiés par l'opération `tick`, ils sont réinitialisés dans le `Next`.

Dans notre cas, on va associer deux timers décroissants à chaque thread de l'ensemble `Thread`, l'un `exec_timer` représentera le temps nécessaire à l'exécution du thread, l'autre `deadline_timer` représentera l'échéance. Lorsqu'un thread sera activé, on initialisera le premier avec la valeur du WCET du thread et le second avec sa deadline. L'opération tick en plus de faire avancer *now*, décrémentera tous les timers associés à la deadline des threads dipatchés (`Ready`) et le timer du thread en cours d'exécution : `computing_tread`.

$$
\begin{array}{l}
tick \triangleq \\
\quad \land\ now' = now + 1 \\
\quad \land\ deadline_timer' = [t \in Thread \mapsto \\
\qquad \text{IF } t = computing_thread \lor t \in Ready \text{ THEN} \\
\qquad deadline_timer[t] - 1 \\
\qquad\ \text{ELSE } deadline_timer[t]] \\
\quad \land\ exec_timer' = [t \in Thread \mapsto \\
\qquad \text{IF } t = computing_thread \text{ THEN} \\
\qquad exec_timer[t] - 1 \\
\qquad\ \text{ELSE } exec_timer[t]]
\end{array}
$$

Remarque La variable *now* n'est pas bornée, cependant si le code utilisateur ne référence pas cette variable, on notera que les timers sont bornés. Si on exclut la variable *now* de l'analyse de l'espace d'état [4], celui est borné. Par suite, une vérification de modèle exhaustive est (théoriquement) possible.

2.2.3 UPPAAL et les automates temporisés

Les automates temporisés ont été introduits par Alur et Dill [AD94] [AD96] pour modéliser explicitement le temps. Cette modélisation est effectuée par l'intermédiaire de variables horloges à valeurs réelles (temps dense) dont la progression est implicite. Un automate temporisé avec invariants [HNSY94] est constitué de deux parties principales :
- Un automate fini qui décrit les états de contrôle du système, et les transitions supposées (sans écoulement du temps) instantanées entre les états.
- Un nombre fini d'horloges utilisées pour spécifier les contraintes de temps associées aux transitions de l'automate fini.

Initialement, les horloges ont des valeurs nulles puis elles évoluent à la même vitesse d'une façon synchrone avec le temps. Chaque état est étiqueté par une expression logique, un invariant, bâti sur les variables d'horloges. L'invariant est appelé invariant de place et il doit être vérifié tant que l'automate ne change pas d'état. Il sert en particulier à assurer que l'on ne reste pas indéfiniment dans un état donné. Chaque transition est étiquetée par un triplet composé :
- d'une expression logique portant sur les valeurs des horloges, appelée garde ou condition de franchissement ;
- d'une étiquette ;
- de remises à zéro d'horloges.

UPPAAL est un outil de vérification de systèmes temps réel. Il se base sur la théorie des automates temporisés afin de décrire des systèmes temps réel. Un système UPPAAL est un ensemble d'automates temporisés qui communiquent. La communication peut se faire soit par variables partagées, soit par synchronisation pure (sans échange de variables) sur des canaux. Les variables, partagées ou locales à un automate peuvent être de type horloge, entier borné, booléen. À partir de ces types primitifs, on peut en créer de nouveaux à l'aide de structures. De plus, on peut utiliser des tableaux de longueur fixe. Les déclarations de variables peuvent être accompagnées de définitions de fonctions pures. Ces fonctions sont décrites dans un langage très proche du c et peuvent faire partie d'une garde, ou être utilisées

[4] En TLA, il est possible d'exclure une variable de l'espace d'état à l'aide du mécanisme de vue.

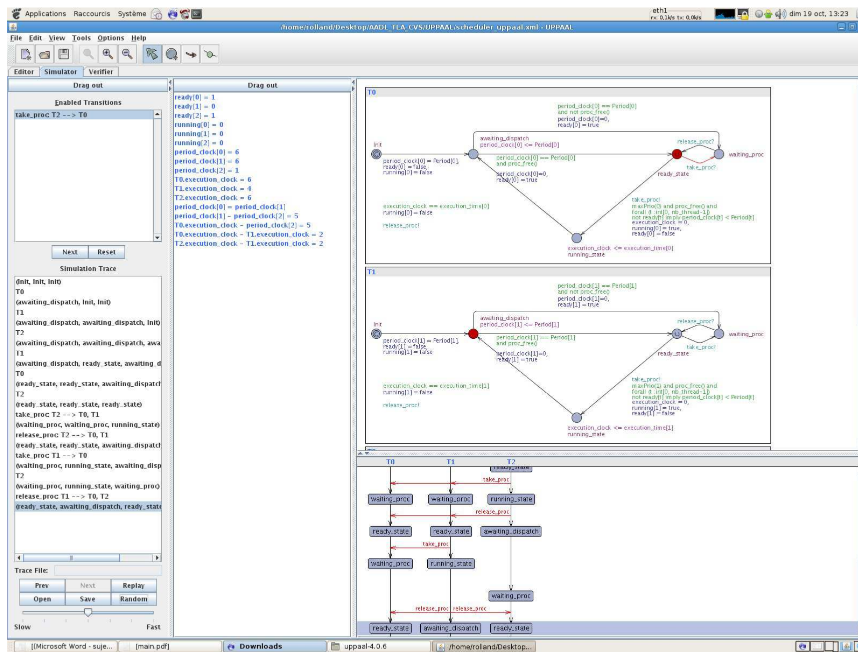

FIG. 2.1 – Capture d'écran du simulateur UPPAAL

pour mettre à jour des variables lors d'une transition. Les états d'un automate peuvent être qualifiés d'`urgent` ou de `commited`. Dans le premier cas, lorsqu'un automate entre dans un tel état, le temps ne peut plus s'écouler. Plusieurs transitions peuvent être tirées lorsque l'automate est dans cet état. Dans le second cas, dès que l'automate entre dans un tel état le temps ne peut plus s'écouler et seules les transitions permettant de sortir de cet état peuvent être tirées.

Chaque automate UPPAAL peut être paramétré, ceci permet de construire facilement des systèmes composés de plusieurs automates indépendants ayant le même comportement. Enfin, UPPAAL est un outil disposant d'un environnement graphique permettant d'éditer les automates, et de voir l'évolution d'un système sous la forme de MSC [IT96] (message sequence charts). Cet environnement permet aussi d'animer le système.

UPPAAL utilise un fragment temporisé de CTL [Eme90] comme langage de spécification de propriétés. On peut donc décrire des propriétés temporelles (sûreté, vivacité, absence d'interblocage) devant être vérifiées par le système modélisé ; le model-checker teste ensuite la validité de la propriété ; dans le cas contraire, il donne une trace permettant d'aboutir depuis l'état initial à un état qui viole la propriété.

2.3 Synthèse

Dans cette première partie, on a commencé par présenter les problématiques liées à l'étude des systèmes réactifs. On a proposé d'utiliser des langages de description d'architecture afin de spécifier de tels systèmes. Puis, nous avons souligné la nécessité d'accompagner de tels langages d'un cadre permettant de définir le comportement des systèmes décrits. On satisfait ce besoin en définissant un modèle d'exécution.

De tels modèles peuvent se situer à différents niveaux d'abstraction. Plus ce niveau d'abstraction est élevé, plus il est simple de donner une sémantique lisible de ces modèles, et plus la validation de propriétés est aisée. Nous avons vu en particulier que les modèles synchrones ou time driven ont pour propriété de base d'être déterministe. Il s'agit d'une propriété importante dès que l'on envisage de faire des vérifications. Nous verront que le langage AADL essaie de conserver une telle propriété. Par contre, plus le modèle d'exécution se rapproche des OS temps réel, plus les modèles seront proches du code final. Dans ce cas, la génération de code est plus aisée.

Enfin, on a présenté différents langages formels comme les outils qui nous serviront à étudier le langage AADL. Nous présentons plus en détails ce dernier dans la partie suivante.

Chapitre 3

Le langage AADL et son modèle d'exécution

AADL [Aer04] est un langage de description d'architectures standardisé par le SAE (Society of Automotive Engineers). AADL a été conçu pour permettre de décrire et d'analyser des modèles de systèmes temps réel [FGHL05]. Il est issu des travaux réalisés sur Meta-H [Ves98], un langage de description d'architectures dédié aux systèmes avioniques et développé par Honeywell. AADL peut être utilisé pour modéliser et analyser les parties logicielles et matérielles de systèmes temps réel critiques. Actuellement (octobre 2008), la version 2 d'AADL est en cours de standardisation auprès du SAE. Dans ce chapitre, on présente dans un premier temps le langage d'un point de vue syntaxique, puis son modèle d'exécution.

3.1 Présentation du langage

Une architecture est décrite comme un ensemble de composants logiciels (process, thread, thread group, data subprogram) qui s'exécute sur une plateforme d'exécution décrite par des composants matériels (processors, memories, devices, buses). Les interfaces des composants sont modélisées par des connecteurs (features), on décrit ensuite les connexions entre les différents composants et l'allocation des composants logiciels sur les composants de la plateforme. Une des particularités d'AADL est de permettre, par l'utilisation de modes, de décrire un système comme un ensemble fini d'architectures qui représenteront des configurations successives dans le temps. On modélise des modifications dynamiques, mais prédéterminées, de l'architecture de l'application. Ce langage possède un modèle d'exécution précis, et est conçu pour être compatible avec les normes ARINC 653 [Air97] et POSIX [IEE93]. Enfin, AADL propose des mécanismes d'extensions permettant de l'adapter à des besoins particuliers.

3.1.1 Structure générale d'un composant

AADL définit trois catégories principales de composants, les composants logiciels (data, subprogram, thread, thread group, process), les composants matériels (memory bus, processor, device), et le composant system permettant de composer les deux catégories précédentes. Cette partie décrit les aspects communs à toutes les catégories de composants AADL. On décrit aussi ici l'organisation d'une application en termes de package. Un composant représente une entité matérielle ou logicielle qui fait partie du système modélisé.

Les packages permettent d'organiser une spécification en groupe de composants séparés en introduisant des espaces de noms séparés. Ils disposent de parties publiques et privées qui permettent de rendre visibles seulement les interfaces d'une application et non son implantation.

Chaque composant a un type qui spécifie son interface externe que ses implantations doivent satisfaire. Il contient l'interface du composant et une liste de propriétés. L'interface est décrite en termes de ports et d'accès proposés ou requis à un sous composant (data, bus, subprogram). De plus, on peut décrire les flots d'informations entre les entrées et les sorties du composant. Un type peut étendre un autre type, il hérite de toutes ses déclarations et propriétés. Il peut compléter et raffiner les déclarations du type dont il hérite. Les composants ainsi définis forment une hiérarchie d'extension.

Une ou plusieurs implantations sont associées à chaque type, elles représentent des variantes du composant se conformant au même type. Une instanciation est déterminée par un type et une implantation. Un type peut éventuellement ne pas avoir d'implantation, dans ce cas une implantation implicite existe. Un composant data peut ainsi avoir seulement un type, on n'a pas nécessairement besoin de connaître son implantation. La réalisation d'un composant est décrite par des sous composants, leurs connexions, des propriétés et éventuellement des modes. Les connexions sont définies entre deux sous composants ou entre l'interface du composant et un sous composant. Les flots décrits au niveau type sont ici précisés par des séquences de flots entre les différents sous composants. Une implantation peut aussi étendre une autre implantation, elle reprend toutes ses caractéristiques et les étend ou les modifie.

La notion de composant est hiérarchique. Une implantation peut contenir des sous composants. Un sous composant est une instance d'une implantation.

L'implantation d'un composant peut contenir une déclaration de modes, chaque mode représente une configuration alternatives de sous composants.

La figure 3.1 représente l'implantation d'un `process`. L'interface de ce `process` est constitué de deux ports `p1` et `p2`. Il contient deux instances de threads, chacun de ces threads dispose aussi de deux ports. Les lignes entre les ports représentent les connexions entre les différentes interfaces.

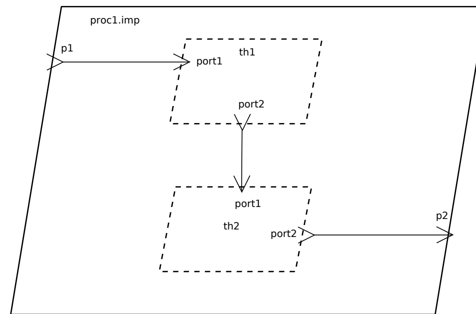

FIG. 3.1 – Représentation graphique d'une implantation de processus

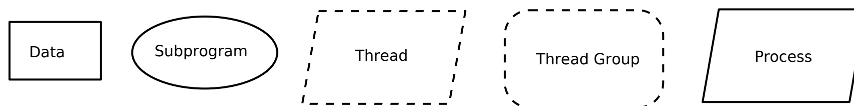

FIG. 3.2 – Représentation graphique des éléments logiciels

```
process proc1
features
  p1: in event port;
  p2: out event port;
end proc1;

process implementation proc1.imp
subcomponents
  th1: thread th1.imp;
  th2: thread th2.imp;
connections
  connection1: event port p1 -> th1.port1;
  connection2: event port th1.port2 -> th2.port1;
  connection2: event port th2.port2 -> p2;
end proc1.imp
```

3.1.2 Les composants logiciels

On présente ici les composants logiciels : data, subprogram, threads, threads group, et process. AADL permet d'associer au composant son code sous la forme d'un fichier externe. Ce fichier externe peut contenir un programme exprimé dans un langage de programmation traditionnel, un langage spécifique au domaine (DSL), ou même un fichier binaire objet. Chaque objet du langage possède une représentation graphique, la figure 3.2 présente les représentations graphiques des éléments logiciels.

Les données (data)

Un composant `data` peut être utilisé pour définir un nouveau type de donnée. La structure interne d'une donnée (champs d'un enregistrement par exemple) peut être représentée par des sous composants data dans l'implantation du composant. Au niveau du type on peut déclarer des sous programmes permettant la manipulation de cette donnée. Dans ce cas, les instances de cette donnée ne peuvent être manipulées que par ces sous programmes. AADL offre ainsi la possibilité d'encapsuler des données.

Un sous composant data représente une donnée statique de l'application. Différents composants peuvent partager l'accès à cette donnée. En AADL le partage de données est possible, mais doit être explicite, le composant contenant la donnée doit en proposer l'accès et un composant désirant partager cette donnée doit en requérir l'accès. On a alors éventuellement besoin de préciser par l'intermédiaire d'une propriété le mécanisme d'exclusion mutuelle utilisé.

Les sous programmes (subprogram)

Un composant `subprogram` représente un texte source appelé avec des paramètres. Dans son type, on trouve ses paramètres, les accès requis à des données externes, et la liste des événements qu'il est susceptible d'émettre. Un sous programme n'a pas d'état rémanent entre deux appels. Des composants `data` peuvent représenter des variables locales dans leur implantation. Un sous programme est appelé depuis un thread ou un autre sous programme. En AADL on ne dispose pas de notion d'exception, tous les traitements liés à une détection d'erreur se font par transmission de messages.

Les threads

Un thread modélise une tâche concurrente, une unité exécutable. Chaque thread représente un flot de contrôle séquentiel. L'interface du thread, définie dans son type, contient une liste de ports, et une liste des accès requis ou proposés à des données ou des sous programmes. Les sous composants d'un thread peuvent être des data ou des sous programmes. Les sous programmes peuvent être partagés par plusieurs threads au sein du même processus. Lors d'un appel à un sous programme le thread exécute le code associé. Dans le cas où le thread propose l'accès à un sous programme comme un service, lorsqu'un second thread appelle ce sous programme, ce thread (appelant) est suspendu ; il reprendra la main lorsque le premier thread (appelé) aura terminé d'exécuter le sous programme. AADL permet ainsi de définir la notion de serveur de sous programmes.

Toutes les caractéristiques liées à l'ordonnancement du thread sont décrites en utilisant des propriétés. On peut ainsi fixer pour chaque thread

son protocole d'activation (périodique, apériodique, sporadique, ou tâche de fond), sa période, sa priorité, sa date limite d'échéance et son temps d'exécution.

```
thread PF
  features
    TC_port : in event data port;
end PF;


thread implementation PF.Impl
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_time => 4 ms .. 4 ms;
    SEI::Priority => 185;
    Deadline => 125 ms;
    Period => 125 ms;
end PF.Impl;
```

Tous les threads ont un port prédéclaré appelé `dispatch`, si ce port est connecté, alors la réception d'un message sur ce port déclenchera l'activation du thread. Dans ce cas, lorsque les autres ports du thread reçoivent des messages, ils sont stockés dans une file et deviennent disponibles lorsque le thread est activé. Si ce port n'est pas connecté, la réception de n'importe quel message déclenche l'activation du thread. Pour les threads périodiques, l'arrivée de messages ne déclenche pas d'activation. Tous les threads ont un port `complete`, si celui-ci est connecté, un événement est envoyé sur ce port lorsque le thread a terminé son exécution.

Les thread groups

Les threads peuvent être logiquement regroupés au sein d'un composant appelé thread group. Il permet de factoriser des éléments de l'interface des threads ou des propriétés. Un thread group peut contenir d'autres thread group, ce composant peut donc servir à créer une hiérarchie de threads.

Les processus

Un processus représente un espace d'adressage virtuel. Cet espace d'adressage contient le code et les données associé aux threads et aux sous programmes du processus. Un processus contient au moins un thread. L'interface du processus est similaire à celle du thread ; elle comporte une liste de ports ainsi qu'une liste de services requis ou proposés. L'implantation d'un processus peut contenir des data, des sous programmes, des threads ou des groupes de threads.

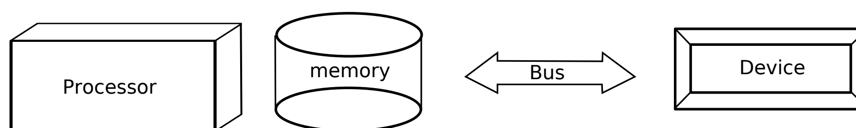

FIG. 3.3 – Représentation graphique des éléments matériels

3.1.3 Les composants de la plateforme d'exécution

Dans cette partie, on présente les composants matériels utilisés pour décrire la plateforme matérielle et l'environnement physique. Comme pour les éléments logiciels, une syntaxe graphique existe pour les composants matériels, cette syntaxe est présentée dans la figure 3.3.

Le processeur

Un processeur est une vue abstraite du matériel et du logiciel responsable de l'ordonnancement et de l'exécution des threads. À un autre niveau d'abstraction, un processeur peut représenter l'OS responsable de l'exécution de l'application. Si le comportement interne du processeur doit être représenté, on peut définir un système modélisant ce comportement et le rattacher au composant processeur. Un processeur exécute les threads auxquels il est associé.

Les mémoires

Un composant mémoire représente une partie de la plateforme d'exécution qui stocke le code associé à l'application et les données. Ce composant peut représenter n'importe quels composants physiques de stockage comme la RAM ou un disque dur. Les différents composants logiciels doivent être associés à un composant mémoire. De plus, le bus d'accès à la mémoire doit être représenté.

Les bus

Un bus est un composant permettant d'échanger des données entre les mémoires, les processeurs, et les périphériques. C'est donc un canal de communication matériel associé à un protocole de communication. Différents bus peuvent être interconnectés.

Les périphériques

Ce composant représente une entité de l'environnement ou une interface avec l'environnement. On peut aussi utiliser un device pour modéliser une partie du système dont on veut faire abstraction. Un device peut avoir un

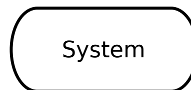

FIG. 3.4 – Représentation graphique d'un système

comportement complexe, et on peut avoir besoin de modéliser ce comportement par un système complet. Une propriété du device permet alors de désigner un autre système AADL représentant le comportement du périphérique. Un device est logiquement connecté aux entités logicielles du système et physiquement relié aux autres composants matériels. Un device peut donc avoir des ports connectés aux composants logiciels.

3.1.4 Les systèmes

Un système est un composant composite rassemblant les composants matériels et logiciels. Les systèmes, avec les thread group, sont les seuls composants AADL qui peuvent être hiérarchiquement récursifs, un système peut contenir un autre système. Deux systèmes peuvent être connectés, on doit représenter leurs interfaces. L'interface d'un système peut contenir des ports, et des listes de services permettant ou requérant l'accès à des bus des données ou à des sous programmes. À l'intérieur d'un système (figure 3.4), dans son implantation on définit comment les composants logiciels sont associés au matériel.

3.1.5 Les interfaces et les connections

Interfaces

Les interfaces des composants AADL sont décrites par des ports ou des services permettant ou requérant l'accès à un composant. On distingue trois catégories de ports, les ports data, les ports event, et les ports event data. Les ports sont directionnels, ils peuvent être en entrée, en sortie ou les deux. Les ports peuvent être regroupés en groupe de ports, lorsque l'interface d'un composant est définie par un groupe de ports le composant auquel il est connecté doit posséder le groupe de ports dual. Un groupe de ports dual est défini comme un ensemble de ports de mêmes types que le groupe de ports auquel il fait référence, avec des directions duales. Les ports sont des points de connexions logiques entre les composants. Ils peuvent être utilisés pour transférer des données (data) ou signaux (event). Les ports sont vus par un thread comme des tampons accessibles comme des variables locales. Les ports en entrée sont mis à jour lors de l'activation du thread. La définition d'instants précis où peuvent avoir lieu des communications est une des propriétés essentielles d'AADL. Elle permet de définir un comportement global

de l'application plus déterministe. On revient sur ces différents mécanismes dans la partie présentant le modèle d'exécution AADL.

- Les ports data servent à transmettre une donnée, ces ports n'ont donc pas de file d'attente, seule la dernière valeur est disponible. Un drapeau (`fresh`) permet au thread de savoir si la valeur du port a été modifiée depuis sa précédente activation. Si le port n'a pas reçu de nouvelle valeur, l'ancienne valeur reste disponible. Les données envoyées par un port data en sortie le sont lorsque le thread termine son exécution.
- Les ports event servent principalement à transmettre des signaux : la réception d'un événement peut déclencher l'activation d'un thread. Chaque port event dispose d'un compteur d'événements. Des propriétés permettent de définir la politique d'accès au port, lors de l'activation d'un thread le système peut mettre à jour un port event en copiant le contenu du compteur d'événement ou un seul. De plus, on peut aussi définir le comportement du port lorsque le compteur a atteint sa valeur maximale. Un port event en sortie peut envoyer un événement à n'importe quel instant de l'exécution du thread. Cet instant est déterminé par le code associé au thread.
- Les ports event data servent à transmettre et stocker des données. Ils disposent d'une file pour stocker les données arrivant et ont le même comportement que les ports event. Lorsque la file est pleine, il peut effacer la donnée la plus récente (queue de la file) ou la donnée la plus ancienne (tête de la file) et insérer dans la file la nouvelle donnée.

L'exemple suivant définit un thread `PF` et son interface composée de 6 ports, un de chaque type. La figure 3.5 montre la version graphique de cette spécification.

```
thread PF
  features
    Port1 : in data port;
    Port2 : in event port {
      Queue_Size => 5 ;
      Dequeue_Protocol => OneItem
    };
    Port3 : in event data port {
      Queue_Size => 10 ;
      Dequeue_Protocol => AllItems
    };
    Port4 : out data port;
    Port5 : out event port;
    Port6 : out event data port;
end PF;
```

L'interface sert aussi à spécifier qu'un composant propose (ou requiert) l'accès à un service (sous programme), ou un sous composant (data ou bus).

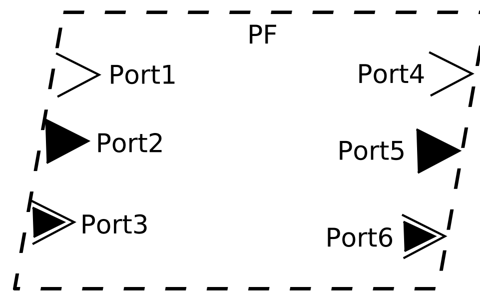

FIG. 3.5 – Définition d'un ensemble de ports

Un thread peut ainsi proposer à d'autres threads d'exécuter des sous programme ou d'accéder à une partie de ses données. Symétriquement, il peut avoir besoin d'accéder à une donnée située dans un autre composant. Dans le cas d'une donnée partagée, une propriété du connecteur permet de définir si le composant veut accéder à la donnée en lecture ou écriture seule ou en lecture et écriture.

Connexions

Une connexion entre deux ports représente un transfert de signaux ou de données entre deux composants s'exécutant en parallèle. Une telle connexion est en réalité une séquence de une ou plusieurs connexions élémentaires qui suivent la hiérarchie des composants AADL. La source et la destination finales d'une connexion sont soit un thread, un processeur, ou un périphérique. Dans le cas d'une connexion entre deux ports data ou deux ports event data, les connexions sont typées, on ne peut établir une connexion qu'entre ports transmettant un type de données compatibles[1]. Un port de data en entrée ne peut avoir qu'une seule source, les connexions de données sont de type 1-n. Les connexions entre ports event et entre ports event data sont de type n-n.

L'exemple suivant définit deux threads, `P` et `Q`, le premier possède un port en sortie et le second un port en entrée. Ces threads sont instanciés dans deux implantations de process : `A` et `B`. À leur tour ces process sont instanciés dans un système. Pour décrire une connexion entre ces deux threads on doit suivre toute la hiérarchie des composants. Une version graphique de cet exemple est représenté par la figure 3.6

```
thread P
  features
    PortThreadP : out data port;
end P;
```

[1]Le standard AADL ne définit pas de notion de compatibilité entre types de données telle qu'on peut la trouver dans des langages de programmation. Cependant AADL permet de définir des relations de compatibilité entre différents types de donnée par une propriété.

```
thread implementation P.imp
end P.imp;

thread Q
  features
    PortThreadQ : in data port;
end Q;

thread implementation Q.imp
end Q.imp;

process A
  features
    PortProcessA :  out data port;
end A;

process B
  features
    PortProcessB :  in data port;
end B;

process implementation A.imp
  subcomponents
   thp : thread P.imp ;
  connections
   cnx1 : data port thp.PortThreadP -> PortProcessA;
end A.imp;

process implementation B.imp
  subcomponents
   thq : thread Q.imp ;
  connections
   cnx2 : data port PortProcessB  -> thp.PortThreadQ;
end B.imp;

system simple
end simple;

system implementation simple.imp
  subcomponents
    procA : process A.imp;
    procB : process B.imp;
  connections
    cnx3 : data port procA.PortProcessA -> procB.PortProcessB;
end simple.imp;
```

Le transfert de donnée entre deux threads périodiques ayant une même période, ou l'une des périodes est un multiple de l'autre, peut être rendu déterministe. Une connexion entre deux ports data de threads périodiques

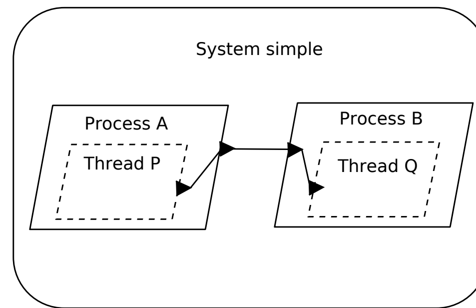

FIG. 3.6 – Hiérarchie des connexions

peut être immédiate ou retardée. Dans le cas de la connexion immédiate, le thread produisant la donnée l'envoie à sa complétion (lorsque le thread rend le contrôle au noyau ; ces aspects du modèle d'exécution seront présentés dans la section 3.2) et le thread destination la recevra au moment du début de son exécution. Si les deux threads sont de même période, la connexion est synchrone, à chaque cycle, un thread produit une donnée, l'envoie à sa destination qui la reçoit au début de son exécution. Si la période du thread récepteur est un multiple de celle de l'émetteur alors, le récepteur fait du sur échantillonnage , il réutilise plusieurs fois la même donnée. Dans le cas contraire, il fait du sous échantillonnage, il ne consomme pas toutes les données produites par l'émetteur. Dans le cas d'une connexion retardée, l'émetteur transmet sa donnée au moment de la fin de son échéance. La donnée produite dans un cycle est consommée au cycle suivant par le consommateur. Il ne peut pas y avoir de cycle de communications immédiates entre un ensemble de composants.

Les connexions servent aussi à relier les connecteurs représentant un service requis ou proposé au composant final proposant ce service (donnée partagée, sous programme). Comme pour les connexions entre ports, il faut parcourir toute la hiérarchie de composants AADL pour atteindre le composant souhaité ; on ne peut pas définir directement une connexion entre deux composants ne faisant pas partie du même composant.

3.1.6 Les modes

Un mode représente une configuration particulière d'une application. Les modes permettent de définir un ensemble prédéfini de configurations alternatives. En AADL, il représente un ensemble de composants actifs, une topologie des connexions, et des valeurs particulières pour les propriétés. Au niveau de la plateforme, on peut définir des ensembles de processeurs, mémoires, bus et devices différents en fonction du mode du système. Certains équipements par exemple peuvent n'être actifs que dans un mode particulier.

Au niveau de l'application, un mode est un ensemble de threads actifs avec une topologie particulière des connexions. Chaque composant a un ou plusieurs modes. Chaque mode est associé à un ensemble de sous composants et de connexions actifs, et a des valeurs particulières de propriétés. Dans chaque composant modal, un automate décrit les différentes transitions de modes possibles. Chaque état de cet automate correspond à un mode, et chaque transition est associée à un port event. La réception d'un événement sur ce port déclenchera le changement de mode. L'automate de mode est une notion opérationnelle ; il doit donc être déterministe. Si un composant est inactif dans un certain mode alors, tous ses sous composants le sont aussi.

La configuration générale de l'application n'est pas visible de manière globale, mais est divisée en plusieurs automates de modes répartis dans les différents composants AADL. Le mode courant du système est défini par un vecteur de modes appelé System Operational Mode (SOM). Chaque élément de ce vecteur représente le mode d'un composant.

Lorsqu'un changement de mode influe sur la configuration générale de l'application, une modification de l'ensemble des threads actifs par exemple, le changement de mode de l'application suit un protocole complexe décrit dans la partie modèle d'exécution. Ce protocole sert à préserver la synchronisation des différents threads entre les deux modes et à gérer les transferts de données entre les modes. Lorsque le changement de mode se situe au niveau du thread le changement de mode peut avoir lieu immédiatement. Il représente juste une alternative dans le code source associé au thread.

Un automate de mode peut être vu comme un statechart [Har87]. La hiérarchie des statecharts se retrouve au niveau de l'imbrication des composants et donc des modes qui peuvent leur être attachés. La concurrence des statecharts (états concurrents) se retrouve au niveau des différents composants actifs dans un même mode. Cependant, la sémantique associée au changement de mode dans AADL est très différente d'un changement d'état dans un statechart, et plus généralement dans une approche synchrone. En AADL le changement de mode ne peut se faire de manière immédiate, la transition n'est pas atomique et doit suivre un protocole complexe[RC04] [Aer04].

3.1.7 Propriétés et annexes

Les propriétés permettent d'associer des informations aux composants AADL. Elles permettent d'enrichir de manière spécifique la description d'un composant. À titre d'exemple toute les informations liées à l'ordonnancement (période, temps d'exécution, échéance) sont des propriétés associées aux threads.

Une propriété a un nom, un type et une valeur. La définition d'une propriété consiste à donner son nom et la liste des éléments auxquels elle s'applique. Le type de la propriété définit l'ensemble des valeurs que peut

prendre cette propriété. On peut définir une valeur par défaut pour une propriété. Les propriétés peuvent être regroupées dans un ensemble de propriétés. Le standard définit deux ensembles prédéclarés. Les propriétés standards couvrent une grande partie des informations nécessaires à l'analyse d'un système embarqué temps réel.

De plus, l'utilisateur peut introduire de nouveaux ensembles de propriétés afin de prendre en compte des problèmes spécifiques non couvert par AADL, par exemple la consommation électrique des différents élément matériels. La propriété d'un composant peut être déclarée dans un autre composant dans la hiérarchie AADL. Par exemple, la période d'un thread peut être définie dans le thread group auquel il appartient. Une propriété dont la valeur est fixée au niveau du type sera définie pour toutes les implantations de ce type. Il est toutefois possible de redéfinir la valeur de cette propriété dans les implantations.

Les annexes permettent d'attacher des annotations dans un autre langage aux composants AADL. Il existe des annexes standardisées comme l'annexe d'erreur et l'annexe comportementale. La principale utilisation de ce mécanisme et de permettre de développer facilement de nouvelles analyses basées sur des notations ou des langages spécifiques. Ce mécanisme permet de définir plus précisément les composants AADL à l'aide de langages spécifiques au domaine étudié (DSL).

3.1.8 Exemples d'annexes standard

L'annexe comportementale

Le but de cette annexe est de permettre la description du comportement interne de composants. AADL permet de décrire le comportement général d'une application, mais il ne permet pas d'exprimer le comportement local d'un composant. On ne peut pas établir de relation entre les entrées d'un thread et ses sorties. Le comportement est modélisé par un automate. Les états de cet automate peuvent être déclarés localement ou faire référence aux états de l'automate de mode du composant. L'état interne du composant peut aussi être décrit à l'aide de variables. Une transition entre deux états peut être gardée, et peut déclencher une action. La garde peut dépendre des variables locales, mais aussi des valeurs des différents ports du composant. Les actions peuvent être des appels à des sous programmes, des modifications des variables locales, ou des émissions des données ou d'événement sur les ports du composant. On peut exprimer que certaines actions prennent du temps. On remarquera que l'annexe comportementale prend en compte le modèle d'exécution sous-jacent à AADL.

L'annexe de modèles d'erreurs

Cette annexe permet de définir des bibliothèques de modèles d'erreurs. Les modèles d'erreurs représentent le comportement des composants auxquels ils sont associés en présence de détection de fautes, de réception d'événement de reconfiguration, ou de propagation d'erreurs. On associe un modèle d'erreur à chaque composant AADL, on donne ses caractéristiques et on décrit leurs relations (propagations d'erreurs...). L'utilisation de cette annexe permet d'avoir une vue centrée sur la fiabilité. Cette vue permet ensuite de générer des modèles de fiabilité basés sur des chaînes de Markov, ou des arbres de fautes. Dans sa thèse [Rug07] A.E. Rugina propose une traduction vers des réseaux de Petri stochastiques généralisés (GSPN) qui permettent d'effectuer une vérification structurelle avant de procéder à l'analyse de fiabilité.

3.1.9 Les additions de AADL V2

Une seconde version du standard AADL est en cours de validation. Elle introduit de nombreux mécanismes de modélisation tels que les composants génériques, les composants abstraits, les tableaux de composants, les processeurs virtuels.

Cette version introduit la possibilité de décrire des *composants génériques*, ces composants définissent des schémas réutilisables de définition qui devront être instanciés par la suite.

Une nouvelle catégorie de composant est introduite, les composants abstraits. Ces composants permettent de décrire des systèmes en faisant abstraction de leur nature logicielle, matérielle ou hybride. Cet aspect sera précisé ultérieurement dans un raffinement.

Afin d'augmenter la puissance d'expression du langage le nouveau standard introduit la notion de tableaux. Cette notion est notamment utile pour l'expression de la réplication (comme par exemple dans les architectures multi-cœurs). Afin de pouvoir gérer facilement les connexions entre des tableaux des sous composants ou de ports, le nouveau standard met à disposition des motifs de topologie de communications. Ces motifs permettent d'exprimer les connexions entre les différents composants d'un tableau. Par exemple, un motif élémentaire est celui associé à un anneau de composants.

Au niveau matériel, des processeurs virtuels ont été ajoutés, ils ont les mêmes caractéristiques qu'un processeur et doivent être associés à un processeur. Ils permettent notamment de décrire des ordonnanceurs hiérarchiques, chaque processeur virtuel constitue une unité d'ordonnancement et exécute un sous ensemble des threads, et le processeur physique répartit son temps entre les différents processeurs virtuels. Cette notion de processeur virtuel peut être considérée comme une abstraction dédiée à la notion de partition.

Le profil temporel des communications (suite des instants de communi-

cation) par ports a aussi été revu. Dans AADL V1 les instants des mises à jour des ports data étaient précis, mais dans certains cas, il pouvait manquer de flexibilité : la mise à jour des ports en entrée ne peut avoir lieu qu'à deux instants, l'activation et le début de l'exécution ; et l'envoie de donnée ne peut avoir lieu que lors de la complétion ou à l'échéance. Quant aux ports event et event data, le standard restait non déterministe pour l'envoi de message (le standard dit qu'un événement peut être émis à n'importe quel moment, pendant l'exécution du thread) et très strict pour la réception(réception lors de l'activation). Un envoi de message pouvait avoir lieu n'importe quand pendant l'exécution d'un thread, et les ports en entrées étaient rafraîchis seulement lors de l'activation du thread. Dans la nouvelle version, on pourra spécifier qu'un port envoie ou reçoit une donnée ou un événement à un instant précis de l'exécution du thread.

3.2 Le modèle d'exécution

On présente ici le modèle d'exécution défini dans le standard AADL. On commence par présenter le cycle de vie d'un thread. On décrit ensuite le fonctionnement des différents mécanismes de communications. Enfin, on termine par une présentation du protocole de changement de mode défini dans AADL.

3.2.1 Ordonnancement

Un thread commence son exécution dans un état en attente d'activation. Une propriété associe à chaque thread un protocole d'activation. Un thread périodique est activé à intervalle de temps régulier spécifié par sa période. Les threads apériodiques sont activés lorsqu'ils reçoivent un événement sur un de leurs ports ou qu'une requête d'exécution d'un de leurs sous programme arrive. Lorsqu'un thread apériodique activé reçoit une nouvelle requête d'activation, elle peut être soit stockée dans une file d'événements soit ignorée. Les threads sporadiques ont le même comportement, mais un temps minimum entre deux activations doit être respecté. Les threads background sont activés immédiatement et ne quittent pas l'état actif. AADL reprend ici les notions classiques des systèmes temps réel [AB90].

L'ordonnanceur gère l'accès des threads actif (figure 3.7) aux différentes ressources du système (processeur et ressources partagées). Une fois activé un thread entre dans un état prêt. L'ordonnanceur sélectionne ensuite le thread de plus haute priorité parmi les threads prêts pour être exécutés. La politique d'ordonnancement est paramétrée par une propriété du processeur (le processeur étant vu comme une abstraction de l'OS). Le thread choisi passe alors dans l'état en cours d'exécution, un seul thread par processeur peut être dans cet état. S'il n'y a aucun thread de prêt, le processeur exécute

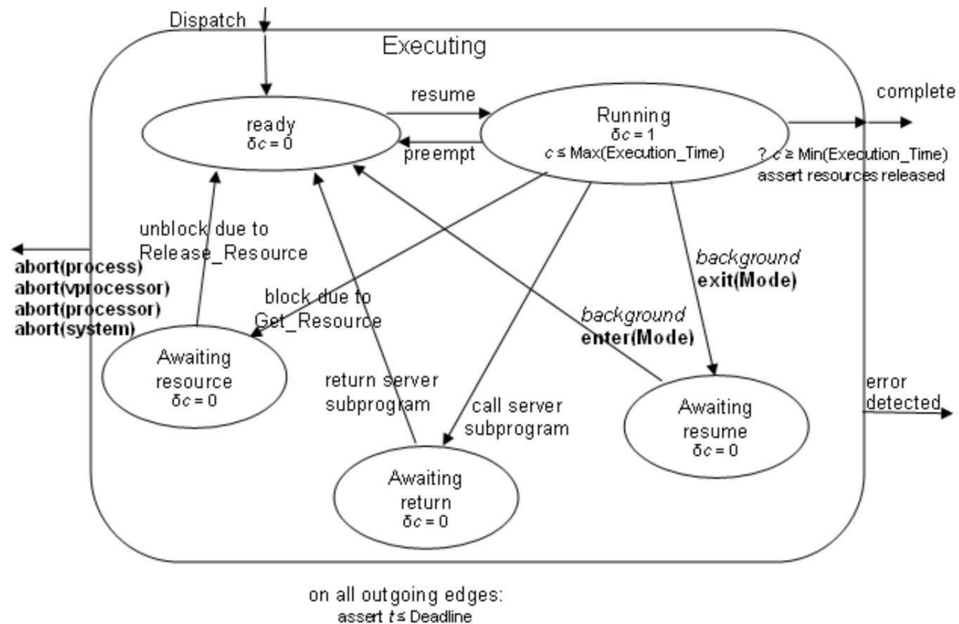

FIG. 3.7 – Les différents états d'un thread AADL en cours d'exécution

un thread background ou reste inactif. Lorsque le thread en cours d'exécution se termine, il retourne dans l'état en attente d'activation[2]. Il peut être interrompu si un thread de plus haute priorité est activé, dans le cas où le protocole d'ordonnancement supporte la préemption, dans ce cas il retourne dans l'état prêt. Il peut aussi être bloqué par l'accès à une ressource partagée, dans ce cas il passe dans un état attente de ressource. Comme pour le protocole d'ordonnancement, le protocole d'accès à une ressource partagée est spécifié par une propriété. Lorsque le thread en cours d'exécution fait un appel à un sous programme exécuté par un autre thread il est suspendu et revient dans l'état prêt lorsqu'il reçoit le résultat du sous programme.

3.2.2 Initialisation et finalisation d'un thread

AADL décrit en plus du comportement normal des threads, leur comportement lors de la phase d'initialisation, de terminaison et de gestion des erreurs. La figure 3.8 représente ces différents états. Lorsqu'un système est démarré, tous les threads sont dans un état arrêtés (`thread halted` cf. figure 3.8). Ils sont tous initialisés, puis ceux faisant partie du mode initial passent dans l'état en attente d'activation et les autres en attente de mode. De même lors de l'arrêt du système tous les threads passent par une phase

[2]En AADL tous les threads sont remis en attente d'activation après leur complétion. Les threads périodiques attendent le début de leur nouvelle période et les apériodique attendent un événement d'activation.

de finalisation. Enfin, lors d'un changement de mode les threads devant être arrêté exécutent une partie désactivation et ceux devant commencer leur exécution une partie activation.

3.2.3 Communications

Les communications par port dans AADL suivent des règles très précises. Cet ensemble de règles permet de définir un cadre de communications très déterministe : lors de l'activation l'environnement d'exécution du thread est gelé, il ne recevra pas de nouvelle donnée avant son activation suivante.

Tous les ports en entrées sont mis à jour au moment de l'activation du thread. Une fois que le thread est activé, son environnement d'exécution ne change plus. Dans le cas où une connexion immédiate relie deux ports data de deux threads périodiques le port destination est mis à jour au début de l'exécution du thread (figure 3.9).

Les données écrites par le thread dans les ports data ne sont envoyées qu'à la fin de l'exécution du thread. Dans le cas de connexions retardées entre threads périodiques, elles sont envoyées à la fin de la période du thread. Les ports event et event data peuvent servir à envoyer des événements ou des données, ils peuvent être utilisés à n'importe quel instant de l'exécution du thread. Les ports event data par défaut se comportent comme des ports data : si une donnée stockée dans un port event data n'est pas envoyée explicitement pendant l'exécution du thread, elle est transmise au moment de la complétion (cf. figure 3.7) du thread. La figure 3.10 montre à quels instants un thread peut communiquer.

Dans sa version 1.0, AADL permet de spécifier qu'un thread doit accéder à une donnée partagée, mais ne permet pas de définir précisément à quel moment il y accède. Lorsqu'un thread déclare accéder à une ressource partagée, on doit considérer qu'il peut y accéder à tout moment de son exécution.

Remarque On peut se rapprocher du modèle d'exécution synchrone en n'utilisant que des threads périodiques et des communications par ports de données connectés par des connexions immédiates ou retardées. En effet, les hypothèses du synchrone[3] sont validées par le comportement où la communication avec l'environnement est figée au début du cycle, et la communication avec les autres threads se fait à des instants précis (dans le même cycle pour les connexions immédiates, avec un retard de un pour les connexions retardées). Une étude de l'expression de la communication AADL à l'aide d'un formalisme synchrone a été faite dans [LMdS08].

[3] L'hypothèse du synchrone d'exécution en temps nul est valide dans le cas où toutes les exécutions se terminent avant l'arrivée de la prochaine interruption.

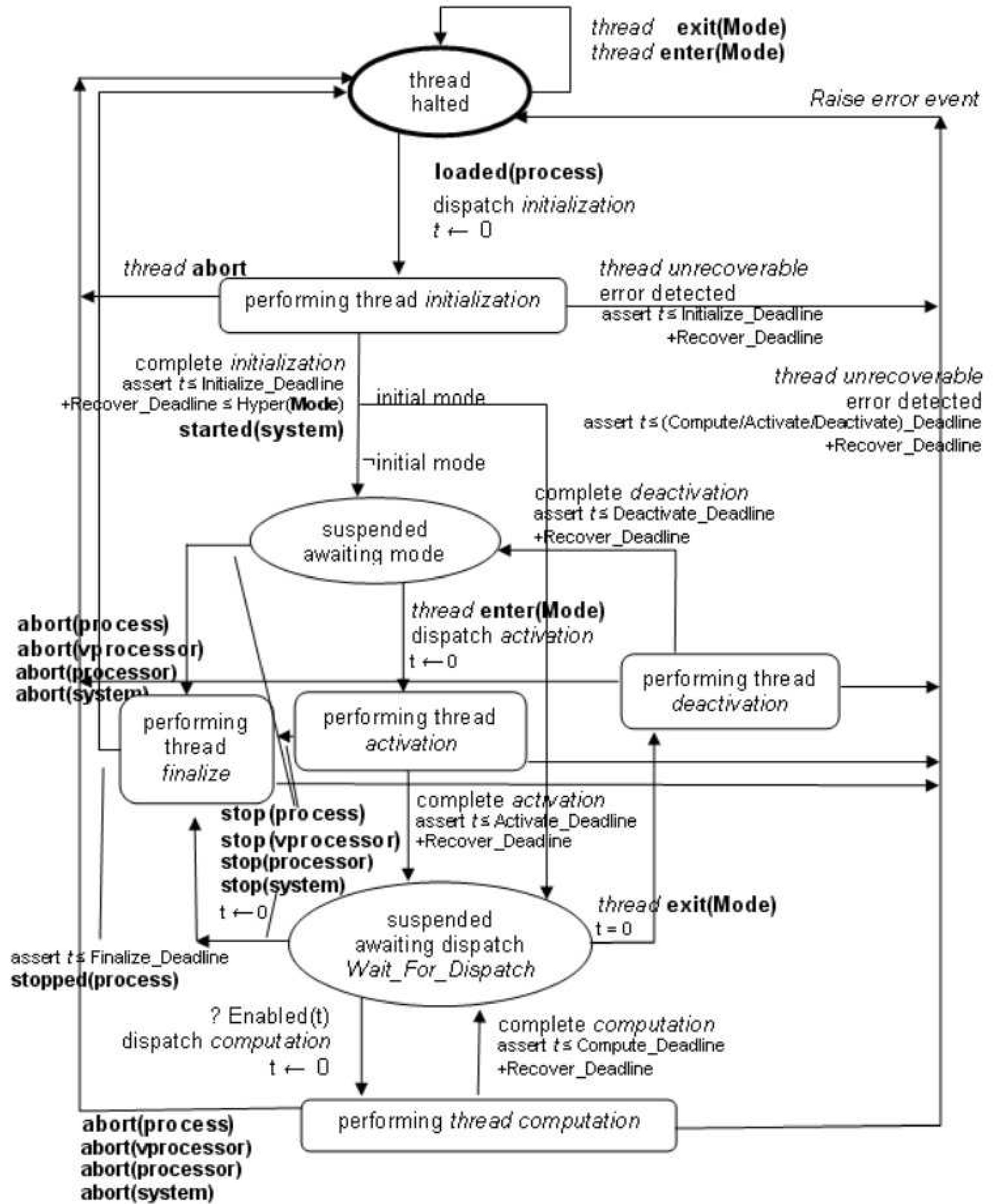

FIG. 3.8 – Cycle de vie du thread en AADL

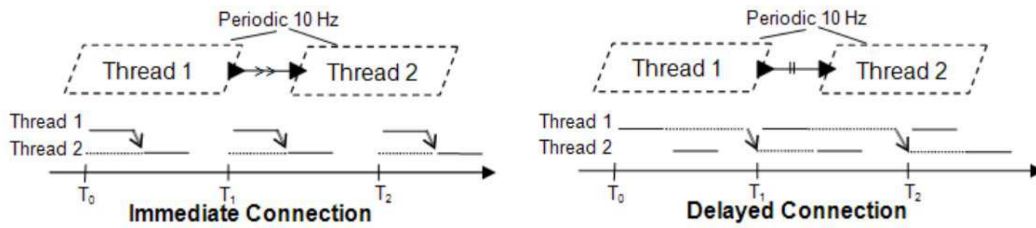

FIG. 3.9 – Connexions immédiates et retardées

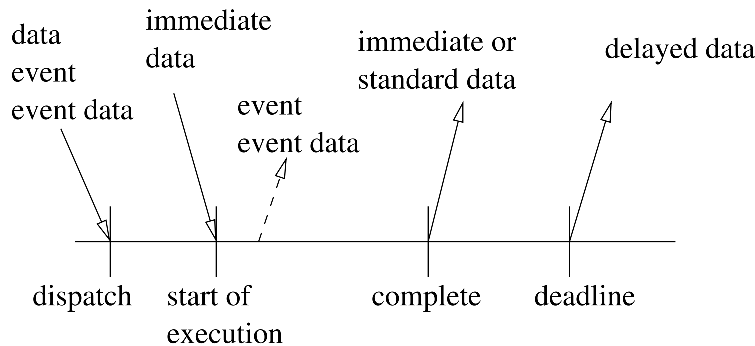

FIG. 3.10 – Instants de communications du thread

3.2.4 Le protocole de changement de mode

Un mode est une configuration de l'application, il est définit par un ensemble de threads présents et un ensemble de connexions. Il s'en suit qu'un changement de mode est un protocole relativement lourd et complexe. Lors du changement de mode, un sous ensemble des threads du mode courant doivent être arrêtés, d'autres threads doivent être réveillés. Ce changement doit être effectué en garantissant des propriétés relatives à la cohérence des données et à la synchronisation des threads.

Afin de garantir des propriétés de synchronisation ou de prédictibilité, AADL décrit un protocole de changement de mode.

Le changement de mode est initié par l'arrivée d'un événement sur un port. Dans un premier temps, le système va continuer à s'exécuter normalement, il attend qu'un sous ensemble de threads périodiques soient synchronisés. Ces threads correspondent à des threads critiques de l'application, ils doivent être stoppés au moment de leur hyperpériode[4]. L'instant où le système arrive à cette hyperpériode est désigné comme l'instant du changement de mode effectif. À cet instant, l'ensemble des threads de l'ancien

[4]L'hyperpériode est définie comme le plus petit commun multiple des périodes des threads. Intuitivement, elle correspond à l'instant où l'ensemble des threads est de nouveau prêt simultanément.

mode qui ne sont pas présents dans le nouveau mode sont arrêtés. Toutes les connexions de ces threads sont aussi désactivées. Dans ce nouvel état, seuls les threads communs aux deux modes continuent leur exécution. Le système reste dans cet état tant que tous les anciens threads n'ont pas été désactivés et tant que les nouveaux n'ont pas été activés. Les nouveaux threads commencent leur exécution lorsqu'ils sont tous activés et que le système arrive à l'hyperpériode des threads critiques communs aux deux modes. La seconde partie de la condition garantit que tous les threads critiques sont activés au même instant, leurs périodes sont alignées. Les nouvelles connexions sont activées à cet instant.

On a vu que seuls des threads périodiques peuvent faire partie des threads critiques, mais des threads apériodiques peuvent aussi avoir à traiter des tâches critiques. AADL donne la possibilité à l'utilisateur de spécifier que certains de ces threads pourront terminer leur exécution dans le nouveau mode. Un tel thread peut être autorisé à terminer seulement son exécution courante ou à terminer le traitement de tous les événements présents dans ses ports.

3.3 Synthèse

On a présenté le langage AADL. Ce langage regroupe les caractéristiques classiques d'un langage de description d'architecture. Ce langage propose la notion de modes permettant de décrire un ensemble de configurations dynamiques, bien que le système soit globalement statique. De plus, AADL étant un langage destiné à la description et à l'analyse de systèmes temps réel critiques et résulte de l'expérience acquise au cours de plusieurs projets dont Meta-H.

Le standard définit le modèle d'exécution sous-jacent au langage et permettant de définir le comportement des modèles AADL. Enfin, il dispose de mécanismes d'extension permettant de l'enrichir pour des besoins dédiés. Dans le chapitre suivant, on présente les problèmes liés au développement de logiciels de vol satellites, et le processus utilisé pour créer de tels logiciels. Dans le chapitre suivant, on présente une étude ASTRIUM destinée à évaluer l'utilisation d'AADL dans un tel cadre.

Chapitre 4

L'étude de cas ArchiDyn

Dans ce chapitre, on commence par présenter les particularités du développement des logiciel de vol. On montre ensuite qu'une solution utilisée pour maîtriser le développement est de suivre un processus standardisé par l'ESA. On présente ensuite l'étude ArchiDyn[LDL06], cette étude réalisée par ASTRIUM, a pour but d'évaluer l'utilisation du langage dans le cadre de ce processus.

4.1 Le développement de logiciels de vol

4.1.1 Fonctions et spécificités des logiciels de vols

Fonctions du logiciel de vol

Le logiciel de vol d'un satellite assure principalement toutes les fonctions de gestion du satellite, mais il peut aussi remplir des fonctions liées à la mission. Ses fonctions dépendent grandement de son autonomie. Les fonctions qui nécessitent des réactions rapides seront toujours réalisées à bord ; ce sont notamment le contrôle d'attitude et d'orbite, le contrôle thermique, la gestion de l'alimentation, et les fonctions de sécurité. On peut classer les fonctions d'un logiciel de vol en trois grandes catégories :

- gestion, surveillance et reconfiguration
- acquisition et traitement
- communication.

Ces activités peuvent cohabiter dans le logiciel de vol central ou être réalisées par des sous systèmes autonomes.

La partie gestion prend en charge principalement la partie contrôle d'attitude et d'orbite (AOCS), le contrôle thermique, la gestion de l'alimentation, la détection et le traitement des pannes (FDIR) et la gestion du plan de travail.

La partie acquisition et traitement concerne la mission du satellite. Celle-ci consiste souvent à acquérir des données et à les transmettre à des stations

au sol. Ces données peuvent être volumineuses et ne peuvent pas en général être transmises directement vers la Terre. Un premier traitement des données peut avoir lieu à bord.

Spécificités du logiciel de vol

Les spécificités d'un satellite imposent de nombreuses limitations au logiciel de vol. La première de ces limitations concerne la communication, la distance entre le satellite et la Terre ainsi que la puissance des émetteurs utilisés limitent la bande passante disponible. De plus, le satellite n'est visible que périodiquement par les stations sol (sauf s'il est géostationnaire), ceci réduit d'autant les possibilités de communication. Il doit donc pouvoir réagir de manière autonome à toutes les situations qu'il peut rencontrer. Enfin, la maintenance du logiciel en vol (remplacement ou modification) est une opération complexe et dangereuse. Le logiciel doit donc satisfaire un niveau de confiance élevé.

L'environnement spatial est très perturbateur, voire destructeur. Les différences de température à l'intérieur du satellite peuvent être très importantes, de plus l'électronique est très sensible au vent solaire et aux orages magnétiques, enfin durant la phase de décollage le matériel doit subir des contraintes physiques très importantes. Ces contraintes imposent l'utilisation de composants électroniques durcis, ce qui limite grandement le choix. Cette limitation impose l'utilisation de processeur peu puissant et de faible capacité de mémoire. Ceci conduit à rechercher une grande efficacité du logiciel de vol en termes de taille de code, de taille des données et de puissance de calcul nécessaire. De plus, ces limitations contraignent aussi le choix de l'environnement de développement. Et comme les processeurs employés sont souvent peu utilisés au sol, les outils d'analyse et de test restent peu évolués.

Satellite	Processeur	Puissance	Mémoire
Spot 4	F9450	0.7 MIPS	256 KB
Spot 5	MA3-1750	1 MIPS	512 KB
Pléiades	ERC32	13 MIPS	6 MB
Protéus	MA3-1750	1 MIPS	512 KB
Myriade	T805	4 MIPS	2 MB
téléphone		100-300 MIPS	3-6 MB

Enfin, le cycle de vie d'un satellite est très long (entre 5 et 10 ans de développement et plus de 5 ans d'exploitation pour un satellite d'observation) et implique de nombreux partenaires. Le développement représente une phase très importante de la vie du satellite, surtout dans le cas de missions scientifiques, ce sont souvent des prototypes produits en un seul exemplaire. Les satellites produits en série comme les satellites de télécommunications nécessitent un temps de développement moindre. Enfin, l'utilisation de ces

satellites pendant de longues périodes rend presque inévitables des phases de maintenance avec le besoin de modifier le logiciel du satellite. Ces contraintes de développement imposent la définition et le respect de processus de développement très stricts.

4.1.2 Processus de développement

Un processus de développement précis et complet est nécessaire afin de permettre à tous les acteurs impliqués dans la construction d'un satellite de communiquer. Ce processus a pour but de supporter le développement, mais aussi l'exploitation du satellite tout au long de sa vie. L'ESA a mis au point un processus standardisé afin d'uniformiser le développement de satellite. Une série de documents publiés par l'ECSS (European Cooperation for Space Standardization) décrit les différentes phases et le planning d'un projet. Les documents ECSS-M-30A [fSS96, fSS03] décrivent le processus à utiliser dans le cadre d'un développement de logiciel de vol. Dans le cadre de cette étude, le document [fSS96] a été effectivement utilisé.

Phase 0 : Identification des besoins Cette première phase a pour but d'analyser et de caractériser les besoins de la mission. On définit aussi les performances attendues et les buts en termes de sûreté et de fiabilité. On doit évaluer dans cette phase les contraintes dues aux particularités de la mission et de l'environnement. On essaie d'identifier les différents concepts de systèmes utilisables, pour cela on se base sur les données résultantes des projets actifs. L'accent doit être mis sur le degré d'innovation nécessaire. De premières évaluations de coûts, de planning et d'organisation doivent être faites.

Phase A : Faisabilité La seconde phase doit finaliser l'expression des besoins fonctionnels commencés dans la phase 0 et commencer à proposer des solutions. Cette phase permet de raffiner les besoins exprimés dans la phase précédente. On essaie de caractériser et de quantifier les éléments critiques (technique ou économique). On explore différentes solutions possibles pour répondre aux besoins. Cette phase a notamment pour but d'estimer la faisabilité d'un projet. Dans certains cas, on peut être amené à modifier la définition de la mission. Le résultat de cette phase est la publication d'une première spécification fonctionnelle.

Phase B : Définition préliminaire Cette phase consiste principalement à préciser les choix effectués précédemment. On s'attachera aussi à sélectionner des solutions parmi les différents choix proposés. On adopte dans cette partie les différentes solutions techniques permettant de remplir les fonctions définies dans la phase A. Les performances et les coûts de ces solutions sont ré-évalués avec plus de précision, notamment en utilisant le résultat des

projets précédents. La faisabilité du projet doit être confirmée. Cette phase conduit à exprimer les besoins en termes de spécification technique.

Phase C : Définition détaillée On raffine dans cette phase les choix faits précédemment. Le résultat est une conception détaillée du système. Ce modèle est utilisé pour étudier les performances globales du système. Les données utilisées en entrée sont des évaluations basées sur les projets antérieurs. On prépare dans cette phase les méthodes et les moyens de production et de vérification.

Phase D : Production et qualification au sol Durant la phase de production et de qualification, le système est entièrement développé. Les éléments théoriques ayant permis la validation des modèles de l'étape précédente sont vérifiés. Les phases de vérifications et de validations sont essentiellement basées sur des techniques de test et d'analyse statique de code. La qualification se décompose en deux parties, on vérifie le comportement des différents composants par rapport à leur spécification et leur aptitude à être utilisés pour répondre aux besoins.

Phases E et F : Utilisation et fin de vie Ces phases couvrent le cycle de vie du satellite depuis le lancement jusqu'à sa désactivation. La phase d'utilisation comprend une partie de tests après la mise à poste du satellite. Durant la phase d'utilisation, des modifications du logiciel peuvent avoir lieu. La dernière phase consiste à retirer le satellite de son orbite. Les satellites en orbite basse sont précipités vers la Terre et brûlent dans l'atmosphère. Les satellites géostationnaires sont envoyés sur des orbites légèrement plus hautes ou plus basses.

Les phases C et D couvrent l'essentiel du cycle de vie d'un satellite, autour de 75%, alors que la phase d'utilisation ne représente que 15%.

Le processus de développement présenté ici correspond à un cycle en V classique (figure 4.1). Les phases 0 et A correspondent à l'analyse des besoins. Les phases B et C sont des phases de design. Et enfin, la phase D correspond à la partie montante du V, production, intégration et validation. Ce processus n'est cependant pas suivi de manière itérative. Le développement se fait par incrément. À chaque pas, une partie des besoins est détaillée et analysée. Une solution est modélisée, validée puis codée. Chaque pas est réalisé en pensant et en préparant les raffinements suivants. Chaque incrément compose en général, un sous ensemble complet du logiciel, cela permet de disposer rapidement d'une partie du logiciel et de réaliser de premiers tests d'intégrations.

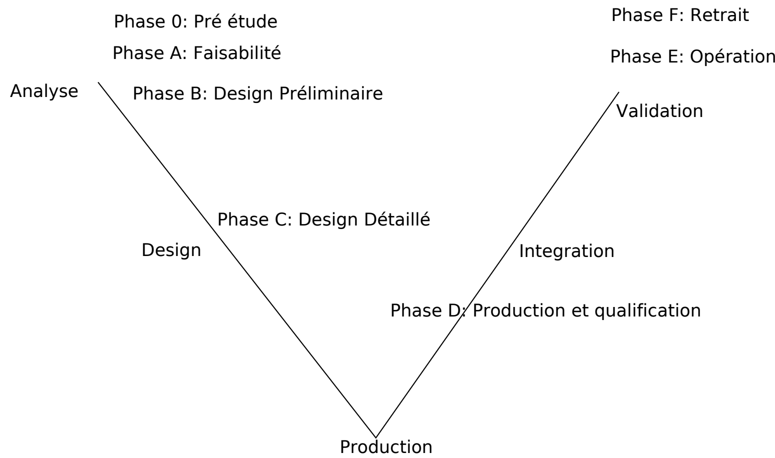

FIG. 4.1 – Cycle de développement du logiciel

Un ancien standard de l'ESA, le PSS-05-0 [fSS91], définit un processus très semblable à celui du standard ECSS-M-30A. Les noms des documents définis dans ce standard restent utilisés dans le processus actuel. La partie analyse est décomposée en deux séries de documents les " User Requirement Documents" et les "Interfaces Requirement Documents". Le modèle d'architecture général est décrit dans l'"Architecture Design Document". Enfin, les modèles de logiciel détaillés sont présentés dans les " Software Requirement Documents".

Tout le processus de développement est basé sur la production des différents documents produits dans les différentes phases. Ces documents contiennent toutes les données techniques et organisationnelles du projet. Ces documents sont validés au cours de revues rassemblant les différents participants au projet. Ils sont vus comme des contrats entre les différents partenaires. Au final, le développement d'un satellite produit des centaines de documents, chacun existant en de nombreuses versions.

Une partie du sujet de cette thèse consiste à étudier l'utilisation d'un langage de modélisation afin d'accompagner le processus présenté. Le but est de le faire évoluer vers un processus dirigé par les modèles.

4.2 L'étude de cas ArchiDyn

Le cas d'étude utilisé pour illustrer les contributions de cette thèse se base sur le modèle AADL d'un logiciel de vol réel, modèle élaboré dans le contexte de l'étude Archidyn, menée par Astrium pour le CNES [LDL06]. Dans ce chapitre, nous présentons les résultats de cette étude, en termes de modélisation et méthodologie, que nous utilisons comme points d'entrée à

notre démarche présentée dans le chapitre 5.

Dans cette partie, on commence par présenter le satellite Pléiades et son logiciel de vol qui nous servira de cas d'étude. On décrira ensuite les différents modèles définis au cours de cette étude. Puis, on explore les différentes relations entre ces modèles et la méthodologie de conception qui peut en découler.

Les résultats de cette étude, notamment les besoins d'expressivité, ont été utilisés dans le cadre de l'évolution de l'annexe comportementale. Les modèles proposés ont été modifiés afin d'être compatibles avec la nouvelle version de l'annexe comportementale qui devra être standardisée après AADL V2.

Le langage AADL ainsi que l'annexe comportementale ont beaucoup évolué depuis le début de cette étude. Les problèmes rencontrés lors du développement de ces modèles ne se posent plus forcément dans les mêmes termes. Ce travail a participé à l'évolution de ces formalismes.

4.2.1 Pléiades et son logiciel de vol

Le satellite Pléiades

Le satellite Pléiades est destiné à remplacer les satellites d'observation SPOT. La mission de ce satellite est de permettre une observation de la Terre journalière avec une définition d'image à résolution métrique. Il est plus simple que ses prédécesseurs, en effet il n'embarque qu'un seul instrument d'imagerie. Mais ses capacités d'observations en font un système plus complexe. La particularité de ce satellite est de pouvoir effectuer des missions d'observations dites agiles ; contrairement aux satellites SPOTs qui étaient limités à la capture d'image à la verticale de leur position, Pléiades peut pointer son instrument de part et d'autre de sa trace. Ceci permet une bien meilleure réactivité, on n'a pas besoin de passer à la verticale d'une région pour la photographier. Les caractéristiques principales de sa mission sont les suivantes :

- orbite basse (visibilité régulière, pas de latence sur les communications) ;
- une seule mission (plate-forme simple) ;
- autonomie limitée, FDIR (Failure Detection Identification and Recovery) simple ;
- grande agilité (calculs embarqués).

Le logiciel résultant de ces contraintes est relativement simple, il est organisé en quatre grandes applications : système, plate-forme, charge utile et AOCS (Système de Calcul d'Attitude et d'Orbite). Ces grandes tâches fonctionnent à une cadence de travail unique. Le plan de mission est établi à l'aide de macros commandes datées envoyées au satellite et interprétées à bord. Enfin, on notera que la fonction de calcul de profil de guidage et

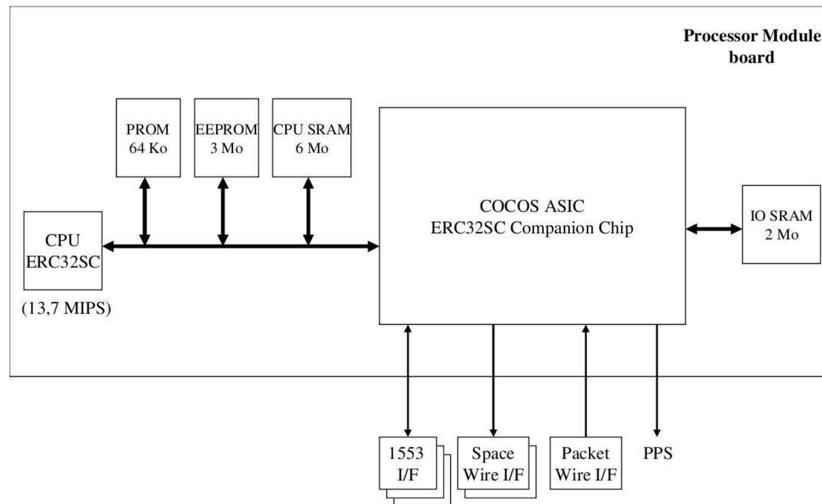

FIG. 4.2 – Architecture matérielle du calculateur

complexe et fortement consommatrice de CPU. L'architecture matérielle du calculateur se décompose principalement en un processeur ERC32 associé à sa mémoire pour le calcul et un ASIC, le CoCOS qui gère toutes les communications vers l'extérieur (instrument, liaisons vers la terre). La figure 4.2 représente cette architecture matérielle.

Présentation du logiciel de vol

Le logiciel est organisé autour de quatre tâches principales séquencées à une fréquence unique de 8Hz (figure 4.3). Chacune de ces tâches peut déléguer des traitements à des tâches apériodiques. Le regroupement d'une tâche périodique et de ses tâches apériodiques forme une application. Le logiciel a été conçu comme multitâches afin de faciliter le développement, mais techniquement une seule tâche aurait suffi. L'ordonnancement des différentes tâches est géré par une politique à priorités fixes. L'accès aux ressources partagées protégées est mis en œuvre par un mécanisme d'héritage de priorités. La partie logicielle communique avec l'extérieur par un bus 1553b.

La partie fonctionnelle du logiciel se décompose en quatre applications suivant un pattern générique :

- le AOCS s'occupe du positionnement du satellite ;
- la plateforme (tâche PF) prend en charge tous les aspects gestion d'énergie et de température ;
- la charge utile (tâche PL : PayLoad) remplit les fonctions associées à la mission du satellite ;
- et le système coordonne l'ensemble de ces applications et gère la partie détection et prise en compte des erreurs.

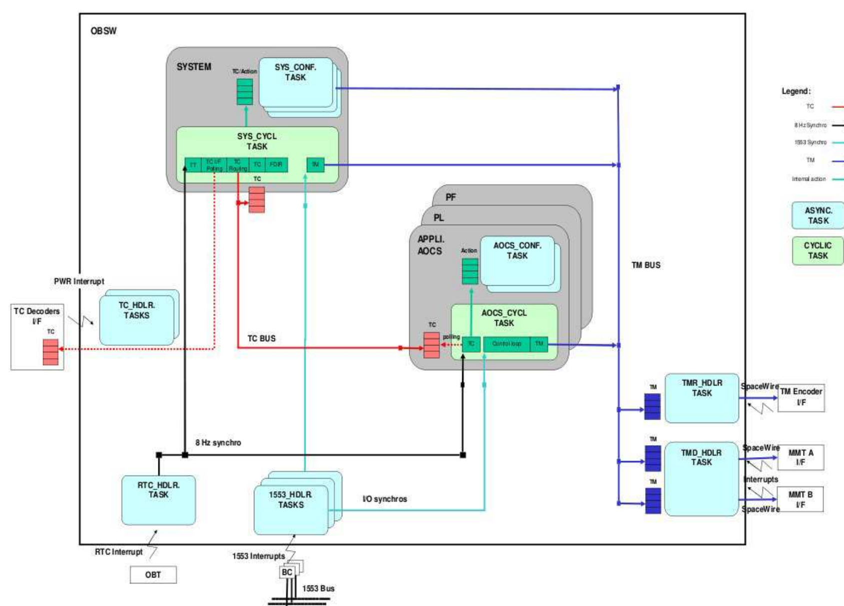

Fig. 4.3 – Architecture logicielle de Pléiades

Chaque application se décompose en une tâche périodique et un ensemble de tâches apériodiques, la figure 4.4 représente ce motif d'implantation. Lors de son activation, la tâche périodique va consulter un tampon de télécommandes (TC) et commence leur traitement. Une télécommande se décompose en une séquence d'actions à exécuter soit par la tâche périodique, soit par une des tâches apériodiques. Le traitement d'une TC par une tâche apériodique peut se dérouler sur plusieurs cycles. Les tâches périodiques, en plus de s'occuper du décodage des TCs et éventuellement de leur exécution. En fin de cycle, les tâches cycliques communiquent leur résultat par le bus 1553.

4.2.2 Modélisations AADL du logiciel de vol

Trois niveaux de modélisation ont été développés afin de décrire les différents aspects de l'architecture : aspect fonctionnel, aspect dynamique temps réel, aspect flot de données.

Le modèle fonctionnel

Ce modèle présente l'architecture fonctionnelle et le comportement attendu du logiciel de vol en faisant abstraction de la technologie employée. Chaque partie du logiciel de vol dispose de plusieurs modes de fonctionnement correspondant aux différentes phases de la mission, configurations, ou modes de défaillances. L'enchaînement de ces modes est généralement spécifié dans les URDs (User Requirement Documents) sous la forme d'automates

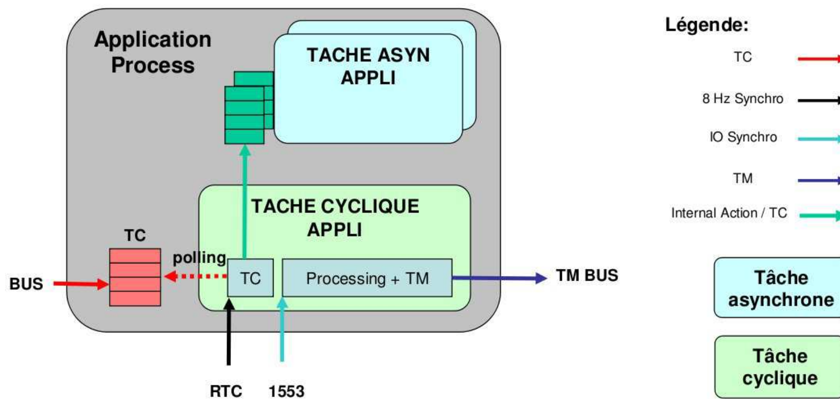

FIG. 4.4 – Pattern d'architecture dynamique

hiérarchiques. Chaque transition de mode peut à son tour déclencher une procédure de changement de mode dans une ou plusieurs de ses sous fonctions.

En AADL les automates de modes ne sont pas hiérarchiques et ne permettent pas de déclencher un traitement lors d'une transition. L'annexe comportementale permet de spécifier un traitement lors d'une transition, mais elle ne permet pas non plus de décrire des états hiérarchiques. Afin de représenter le comportement hiérarchique du satellite, nous avons utilisé une hiérarchie de composants `systems`, chaque composant contenant un automate décrit grâce à l'annexe comportementale.

Après avoir décrit le comportement haut niveau du logiciel de vol, on intègre au modèle une première spécification de l'architecture des communications. Pour cela, on suit un plan de description commun aux objets de l'URD. Chaque application dispose de l'interface suivante :

– un port pour les requêtes en entrée,
– un port pour les résultat (TM Télémesure),
– un port pour les émissions de requêtes FDIR.

Chaque application se décompose ensuite en :

– un composant `Req handler` qui reçoit les requêtes et les distribue aux autres composants ;
– un composant `Manager` qui gère les modes de l'application ;
– un composant `TM handling` qui génère les résultats ;
– un composant `FDIR strategy` qui s'occupe de la gestion des pannes ;
– et un ensemble de fonctions.

La figure4.5 représente cette décomposition.

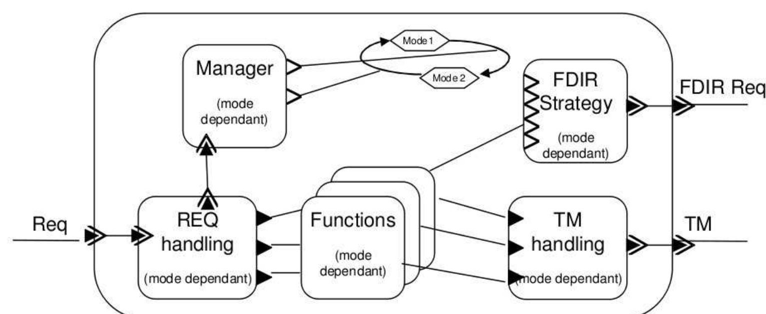

Fig. 4.5 – Architecture fonctionnelle

De nombreux problèmes de modélisation ont été rencontrés à ce niveau. La description d'un modèle haut niveau à l'aide d'un langage ayant une sémantique très forte a été problématique. Mais depuis la fin de cette étude, de nouvelles constructions ont été introduites dans le langage AADL telle que les composants abstraits. Ces nouveaux éléments du langage ont pour but de faciliter la description de système à un haut niveau d'abstraction. De même, l'annexe comportementale a été enrichie et une partie des problèmes rencontrés est maintenant aisément appréhendable.

Le modèle d'analyse temps réel

À ce niveau, on décrit le logiciel de vol comme un ensemble de tâches ayant des caractéristiques temporelles représentatives du comportement du logiciel à l'exécution.

Deux niveaux d'abstractions Dans un premier temps, on construit un modèle simplifié de tâches. Chaque tâche est modélisée par un thread, et est caractérisée par :

- son protocole d'activation (périodique ou apériodique),
- sa période,
- sa durée d'exécution pire cas,
- son échéance.

La construction de ce modèle se base principalement sur l'utilisation de patterns récurrents. Ainsi, chaque application est décomposée en une tâche périodique (de période 125ms dans notre cas) et d'un ensemble de tâches apériodiques commandées par cette tâche périodique. Les temps d'exécution utilisés sont estimés à partir des projets déjà développés.

Dans un second temps, on va introduire des mécanismes plus complexes. Les tâches périodiques notamment réalisent des fonctions de contrôle segmentées en trois parties : acquisition, contrôle, commande. Les échanges de

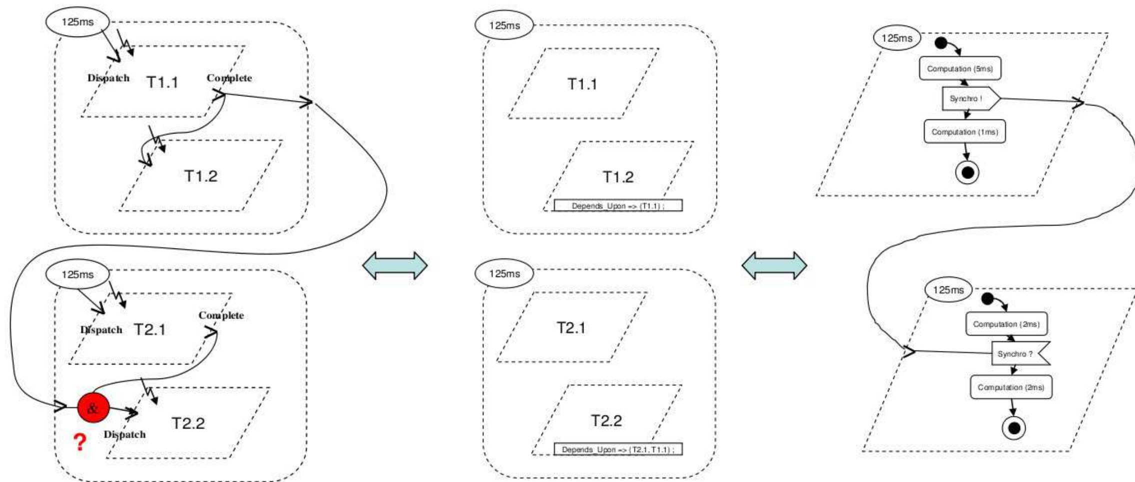

FIG. 4.6 – Description d'un tâche à suspension avec synchronisation

bus alors nécessaires sont réalisés en un temps non négligeable forçant ces tâches à s'endormir.

Différentes solutions ont été envisagées (figure 4.6) pour modéliser ce comportement. Dans la première, la tâche périodique est séparée en deux parties ; la première est périodique et la seconde apériodique et est déclenchée par la réception d'une combinaison d'événements, la fin de la première partie et la synchronisation du bus. Mais l'activation de thread par une combinaison d'événements n'était pas prévue dans AADL. Une seconde solution consiste à déclencher la seconde partie par l'arrivée de l'événement de synchronisation bus en ajoutant une propriété de dépendance spécifiant que cette tâche ne peut commencer son exécution que si la première partie a terminé la sienne. Une troisième solution consiste à utiliser l'annexe comportementale. Dans ce cas, on considère que la lecture sur un port event vide est bloquante. La tâche est débloquée lorsque l'événement arrive.

On a présenté ici un problème de synchronisation avec le bus 1553. Afin de fermer le système, on doit représenter ce bus.

Fonctionnement du bus 1553 Dans l'architecture Pléiades, les échanges avec le bus interne sont réalisés grâce à une trame définie statiquement et exécutée par le CoCOS. Elle est divisée en plusieurs `slots` et traduit l'ensemble des messages possibles. La position de chaque échange dans la trame est fixe, il est possible de connaître précisément la date d'envoi ou de réception d'un message, et par conséquent les dates de synchronisations.

Une première modélisation du bus est basée sur l'utilisation de l'annexe comportementale. Dans cette modélisation, on décrit le comportement des handlers 1553 par des automates. La temporisation de la trame 1553 est

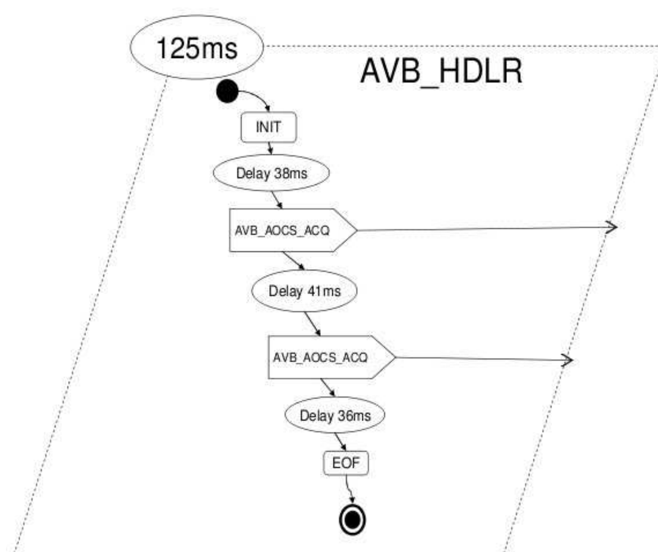

FIG. 4.7 – Modélisation des synchronisations avec le bus 1553

décrite à l'aide de la primitive `delay` de l'annexe comportementale (figure 4.7).

Une seconde solution consiste à considérer le bus 1553 comme un système séparé ayant des ressources communes avec le modèle étudié. Ainsi, le bus peut être modélisé par un processeur, une tâche unique et un ensemble de données partagées (figure 4.8). Il faut ensuite définir à quels instants la tâche accède aux données. Cela n'était pas prévu dans la version un d'AADL, une nouvelle propriété a donc dû être ajoutée. Cette solution s'éloigne de l'implantation réelle, mais reste cohérente d'un point de vue temporel. Cette solution a été utilisée dans le cadre de simulation et de validation d'ordonnancement avec l'outil Cheddar [SLNM04].

Le modèle physique

Ce modèle est le plus proche de la structure du logiciel réel. De plus, ce modèle détaille de manière plus fine le matériel associé au logiciel. Il se décompose hiérarchiquement en plusieurs systèmes. Au niveau le plus haut, on décrit les interfaces internes et externes du satellite : les bus 1553 connectant les différents équipements, les liens vers la Terre (space wire et packet wire). Ce niveau comprend aussi une instance du niveau inférieur, le processor module. Dans ce second niveau, on décrit l'architecture matérielle du calculateur : processeur, mémoires, bus interne, le CoCOS, et leur relation avec le système logiciel de niveau inférieur. Ce troisième niveau reprend globalement l'architecture du modèle d'analyse temps réel. Les flots de données présentés dans le niveau précédent peuvent changer de nature, par exemple

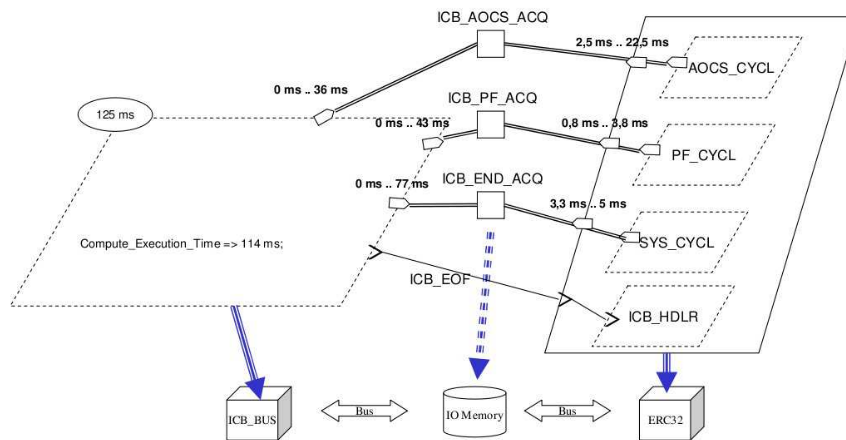

FIG. 4.8 – Modélisation du bus 1553 à l'aide de ressources partagées

une connexion event data entre deux threads peut être implémentée par des ressources partagées. Enfin, une grande partie des communications était occultée dans le modèle précédent, on ne gardait que la communication nécessaire à la description de l'ordonnancement. On ajoute ici toute la partie transmission de données.

4.2.3 Méthodologie et procédé

Dans la partie précédente, on a présenté différents modèles du logiciel de vol. Ces modèles correspondent à trois points de vue du logiciel, fonctionnel, flot de contrôle et flot de données et à trois niveaux du cycle de vie, spécification, conception, et conception détaillée). On présente ici les relations qui existent entre ces différents modèles.

Conception du modèle d'analyse temporelle

Utilisation de patrons La définition des tâches du logiciel de vol est justifiée par la réutilisation de bonnes pratiques acquises par l'expérience. Celles-ci sont capturées à travers la notion de patrons de conceptions. Elles sont rassemblées au sein de l'`application framework` qui définit une micro architecture générique de composants utilisée pour chaque application. Pour les satellites d'observation comme Pléiades, une solution mono fréquence est généralement adoptée. Ainsi, le patron utilisé est défini par une tâche périodique réalisant à la fois le traitement des requêtes, les boucles de contrôle et la génération de la télémesure et par au moins une tâche apériodique, activée sur demande de la tâche cyclique, utilisée pour les activités de plus longue durée (calcul de manœuvre par exemple). Pour des satellites plus complexes, ce patron se généralise à travers le cas multifréquences, l'application se décompose en plusieurs tâches périodiques ayant des périodes différentes et un

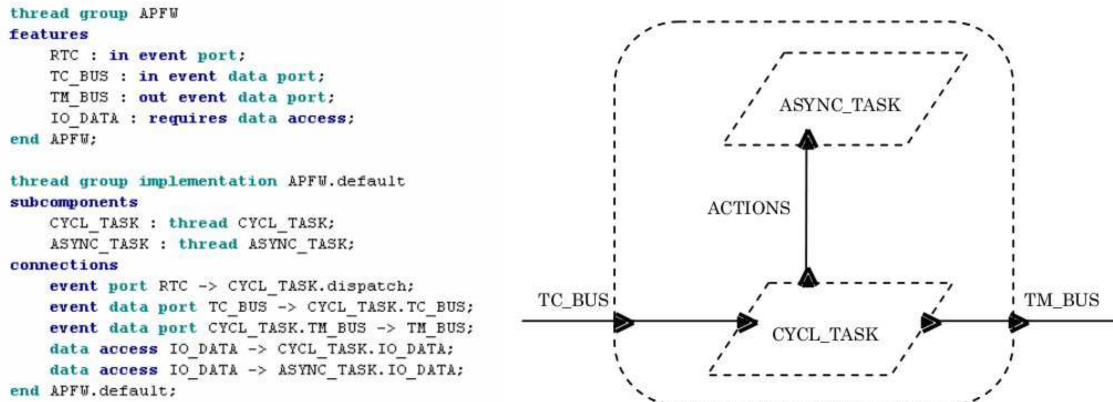

FIG. 4.9 – Patron d'application monofréquence

ensemble de tâches apériodiques. La figure 4.9 présente une vue AADL de ce patron.

Ajout des informations temporelles Le modèle obtenu à partir des patrons de conception décrit l'architecture d'un logiciel. L'analyse du comportement à l'exécution nécessite de compléter le modèle avec les caractéristiques temporelles des tâches : période, temps d'exécution, échéance, algorithme d'ordonnancement. À ce niveau, une première analyse de type pire cas peut être effectuée.

Le comportement des tâches est ensuite progressivement raffiné à l'aide d'automates pour préciser les synchronisations avec le bus. On ajoute aussi à ce niveau une abstraction du bus de communication. Une deuxième analyse d'ordonnancement, plus précise, peut alors être effectuée.

Construction du modèle d'implémentation

Le passage du modèle d'analyse temps réel au modèle d'implémentation consiste principalement à préciser les flots de données à l'aide des informations issues du modèle fonctionnel et de raffiner les moyens de communications et de synchronisations.

Allocation des fonctions définies dans le modèle fonctionnel Le modèle d'analyse temporelle est presque une implémentation du modèle fonctionnel. En effet, il détermine l'exécution des fonctions identifiées au niveau supérieur. Il est alors intéressant de lier explicitement ces deux modèles en indiquant par exemple quelle tâche du modèle concret exécute les fonctions définies au niveau plus abstrait. Cette allocation des fonctions sur l'architecture logique permet en outre d'éviter une redondance de description et facilite le transfert d'information d'un modèle vers l'autre. Cette

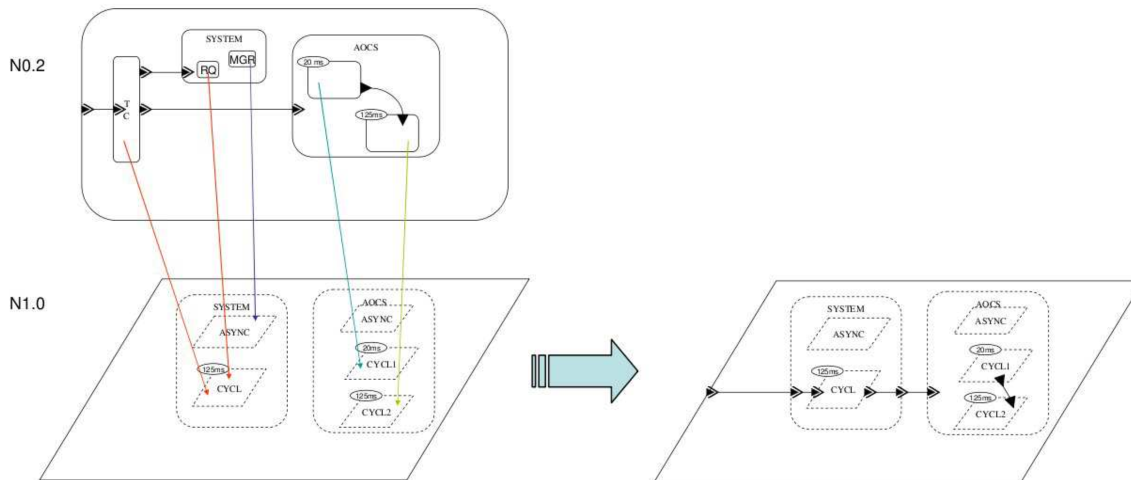

FIG. 4.10 – Tissage entre deux niveaux d'abstraction

technique permet d'identifier des flots de données ou des synchronisations entre les tâches. L'allocation de deux fonctions échangeant des données sur des tâches différentes implique un flot de données entre ces deux tâches (figure 4.10). De la même manière, lorsque deux fonctions devant être exécutés séquentiellement sont allouées sur deux tâches différentes, une synchronisation entre les deux tâches doit être définie.

Implémentations des moyens de communication et de synchronisation L'objectif ici est d'identifier les mécanismes ou les objets de la plateforme (service et primitives de l'OS par exemple) qui seront utilisés pour implémenter le modèle logique (figure 4.11). Par exemple, concernant les synchronisations et les communications, AADL définit trois types de connexions :

- connexion data : cette connexion correspond à un échange de donnée et peut être en général implémenté par une variable partagée protégée ou non. Peter Feiler propose une étude présentant de telle relation entre modèles dans [Fei08].
- connexion event data : on a ici à faire à un transfert de données entre deux tâches avec un tampon d'une taille précise. Elle peut être implémentée par une simple file.
- connexion event : il s'agit d'une synchronisation. Elle est généralement implantée par l'envoi / réception d'un événement ou par l'utilisation d'un sémaphore binaire ou plus simplement par polling d'une variable booléenne.

La figure 4.12 résume le processus de développement proposé.

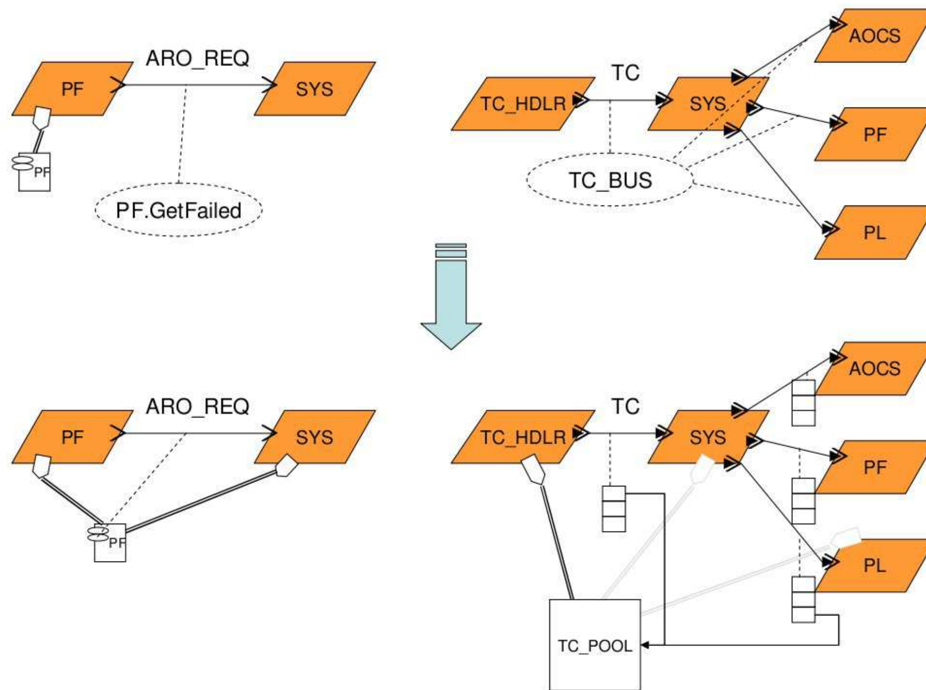

FIG. 4.11 – Implémentation de connexions par des composants data

4.3 Impact de cette étude sur le langage

On rassemble ici les principales critiques faites au langage AADL et à son annexe comportementale ; on étudie ensuite les solutions que proposent les évolutions de ces formalismes.

Un des premiers problèmes d'expression rencontrés a été l'impossibilité de définir des états composite, ce type d'état étant utilisés pour décrire progressivement un comportement. Aujourd'hui l'annexe comportementale permet de définir de tels états.

Les politiques d'accès au ports dans AADL sont très strictes. Pour les port `event` ou `event data`, la version un du langage ne permettait d'accéder qu'à un ou tous les éléments du ports. Dans AADL V2, une nouvelle politique, plus flexible, est apparue. Elle permet d'accéder à un nombre variable d'élément présent dans la liste des ports.

La description de tâches à suspensions qui était problématique est désormais beaucoup plus aisé grâce à l'introduction d'un nouveau protocole d'activation. Ce protocole permet de décrire des tâches hybrides à la fois activées périodiquement et pouvant être déclenchées par la réception d'un message.

Enfin, cette étude présente AADL comme un langage trop bas niveau ne permettant pas de décrire des systèmes abstrait. Cette critique a été prise en

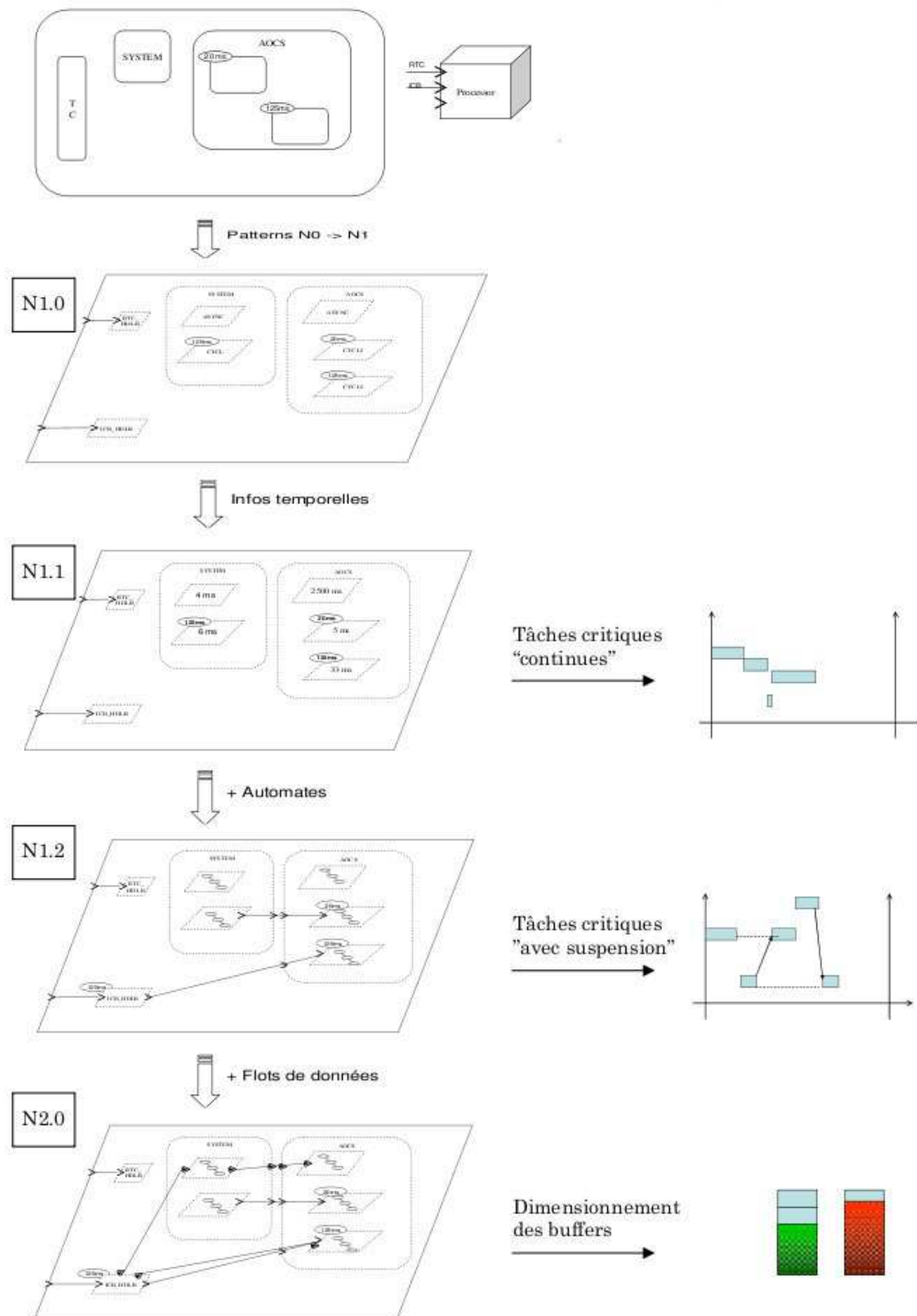

FIG. 4.12 – Processus de développement par raffinements successifs

compte et AADL V2 propose des composants abstraits et des mécanismes de généricité.

Certaine des critiques formulées dans ArchiDyn sont cependant toujours actuelles. Par exemple, on ne peut pas faire d'héritage multiple, il n'est pas non plus possible de supprimer des éléments hérités.

4.4 Synthèse

Dans ce chapitre, on a commencé par présenter les fonctions et les spécificités des logiciels de vol satellites. On a ensuite décrit le processus de développement utilisé pour prendre en compte la complexité lié à un tel développement. On a ensuite présenté l'étude ArchiDyn, ce travail a été à la source de certaines évolutions de AADL et de son annexe comportementale. Cette étude et plus particulièrement le modèle d'analyse temporelle nous a servi à définir le mini-AADL présenté dans le chapitre suivant.

Chapitre 5

Un mini AADL pour ArchiDyn

Dans le chapitre précédent, on a présenté le langage AADL et son modèle d'exécution : l'ensemble des mécanismes d'ordonnancement, de communication et de gestion des modes. On définit un mini AADL, on a développé ce langage afin qu'il réponde pleinement aux besoins de modélisation et de vérification des études de cas envisagées (cf. 4.1), mais aussi pour travailler sur un langage plus simple et plus réduit dans le but de n'aborder que les aspects qui nous semblent pertinents. Informellement, nous faisons abstraction des aspects concernant le génie logiciel tels que les `process`, `thread group` ainsi que des éléments propres à la plateforme matérielle.

Dans un premier temps, on décrira les différents composants du langage ainsi que leur comportement. Cette description est faite de manière progressive, on commence par décrire le modèle de thread, puis la communication et enfin la gestion des modes. On présentera ensuite une spécification formelle de son modèle d'exécution en TLA+. Dans une troisième partie, on étudie plus en détail le concept de modes système, on y présente deux abstractions, une en TLA+ et une en UPPAAL, du protocole de modes AADL. La spécification TLA+ donne une vue globale du changement de mode. Dans la partie UPPAAL on étudie plus en détail le coté temporisé de la transition de mode.

5.1 Le langage et son modèle d'exécution

Notre mini AADL repose sur l'utilisation de deux composants logiciels de base, les threads et les data. Par conséquent, nous ignorons toute la dimension hiérarchique d'AADL, on ne prend pas en compte les composants `system`, `process`, `thread group`, qui servent à structurer (dans le sens génie logiciel) une modélisation. De plus, on n'utilisera pas non plus les éléments de définition de la plateforme d'exécution. Les threads sont les principaux

composants actifs d'AADL, leur description définit toute la partie ordonnancement du modèle d'exécution. On conserve ici une grande partie de leur sémantique d'exécution, on ignore les processus d'activation et de désactivation ainsi que la gestion des erreurs. Les `data` représentent un composant passif permettant d'échanger des données, mais aussi de synchroniser les threads dans le cas de données protégées. Les communications entre threads peuvent aussi être définies grâce aux ports. Là aussi, on reste très proche de la définition des ports dans AADL : la plupart des politiques d'accès aux ports a été considérée. Enfin, la notion de mode dans le développement d'un logiciel de vol étant très structurante au niveau de la dynamique de l'application, nous l'avons conservée. Notons que le comportement interne des threads pourra être modélisé à l'aide d'une sous partie de l'annexe comportementale d'AADL.

5.1.1 Le modèle de threads

Un thread représente une unité de traitement. Ce composant est caractérisé par son interface et un comportement interne. De plus, un modèle d'ordonnancement décrit les différents états des threads en cours d'exécution.

L'interface d'un thread est décrite par un ensemble de ports en entrée, ou en sortie ainsi que par un ensemble de connecteurs devant être associés à des composants `data`. On détaille dans la partie suivante les caractéristiques et le comportement de ces ports.

Comportement interne du thread

Le comportement du thread est décrit par un sous ensemble de l'annexe comportementale. Le comportement est donc décrit par un automate. Les états de cet automate peuvent être qualifiés de `initial`, ou de `complete`. Il n'y a qu'un seul état initial par automate, c'est l'état de départ du thread lors de sa première exécution. Une exécution aboutit à un état terminé (`complete`). Lors de l'activation suivante, le thread recommencera son exécution dans ce dernier état. Les transitions entre deux états de l'automate peuvent être gardées par des expressions booléennes et déclencher des actions. Une garde est une expression booléenne pouvant porter sur le contenu des ports en entrée du thread ou sur les composants `data` auxquels il a accès[1]. Les actions sont des mises à jour des tampons de sorties associés aux ports ou des modifications des zones de données auxquelles le thread peut accéder. On accède ici aux variables associés aux ports et non aux tampons du ports. Rappelons qu'un port est constitué d'une variable accessible au thread et d'une zone tampon gérée par le système. Les valeurs de ports en

[1] Ce dernier point fait l'objet de discussions pour la version de l'annexe comportementale qui accompagnera AADL V2.

entrées sont mises à jour aux instants définis par le modèle de communication. De même, les données des ports en sortie sont envoyées à des instants fixes.

Dans l'exemple suivant, on déclare un thread avec deux ports `data` en entrée, et un port `data` en sortie. Lors de chaque exécution, le thread renvoie sur son port de sortie le maximum des deux valeurs contenues dans ses ports en entrée. La description d'une transition se fait en 4 parties. On commence par spécifier l'état de départ de la transition. On définit ensuite entre crochets la garde de la transition. L'accès au contenu d'un port se fait grâce à l'opérateur `?`. On donne ensuite l'état cible de la transition. Enfin on définit les actions de la transition par une expression entre accolade. L'envoi d'une donnée ou d'un événement par un port est spécifié par l'utilisation de l'opérateur `!`.

```
thread max
  features
    Port1 : in data port;
    Port2 : in data port;
    Port3 : out data port;
end max;

thread implementation max.imp
annex Behavioral_annex {**
  states
    s0 : initial complete state;
  transitions
    s0 -[Port1? <= Port2?]-> s0 {Port3!(Port2?)};
    s0 -[Port1? > Port2?]-> s0 {Port3!(Port1?)};
**};
end max.imp;
```

Ordonnancement et états du thread

Un thread est caractérisé par des propriétés classiques de la théorie de l'ordonnancement :

- protocole d'activation (périodique ou apériodique),
- période (pour les threads périodiques),
- temps d'exécution,
- date limite de fin d'exécution,
- priorité.

Au début de son exécution, un thread est en attente d'activation (dispatch). Les threads périodiques sont activés à intervalles réguliers définis par leur période. Les threads apériodiques sont activés lorsqu'un message arrive sur un de leurs ports "event". Tous les ports event ne déclenchent pas forcément une activation. On doit définir de manière explicite quels ports peuvent déclencher l'activation du thread en les connectant au port par-

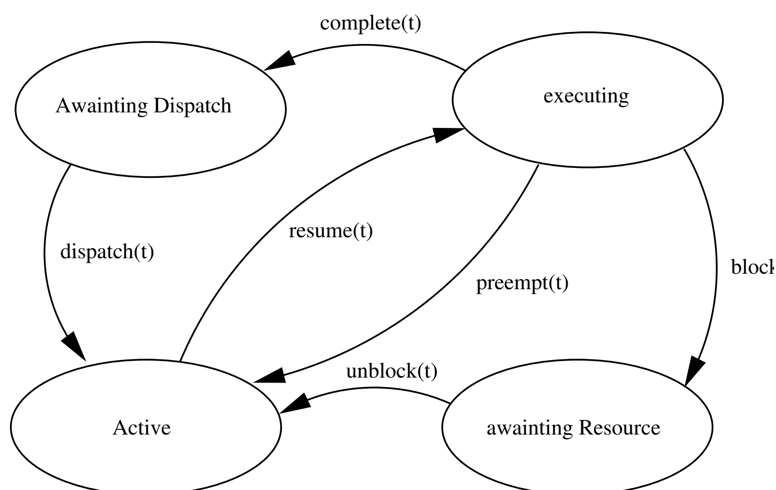

FIG. 5.1 – Ordonnancement des threads

ticulier “dispatch”. Une fois activé, un thread passe dans l'état “ en cours d'exécution ” s'il a la priorité maximum. Un thread en cours d'exécution peut être interrompu si un thread de plus haute priorité est activé ou s'il tente d'accéder à une ressource partagée verrouillée. Dans le premier cas, le thread revient dans l'état activé et attend d'être de nouveau le thread de plus haute priorité. Dans le second cas, le thread passe dans un état “ en attente de ressource ”, il revient dans l'état activé lorsque la ressource est relâchée. Lorsqu'un thread termine son exécution, il revient dans l'état en attente d'activation. L'automate présenté dans la figure 5.1 représente ce comportement.

5.1.2 Le modèle de communication

Dans le modèle présenté, la communication entre threads est assurée par deux mécanismes, les ports, et les données partagées. On présente ici les différents types de ports puis les variables partagées.

Les ports

On définit trois types de ports, les ports `data`, les ports `event` et les ports `event data`. Chacun de ces ports peut être soit en entrée soit en sortie. On décrira en premier lieu le comportement fonctionnel des ports puis leur comportement temporel.

Comportement des ports Un port `data` en entrée est constitué d'une zone tampon (non accessible au thread), d'une valeur disponible pour le thread et d'un marqueur permettant de tester la mise à jour de la donnée lors de la dernière activation. Le rôle de la zone tampon est de modéliser

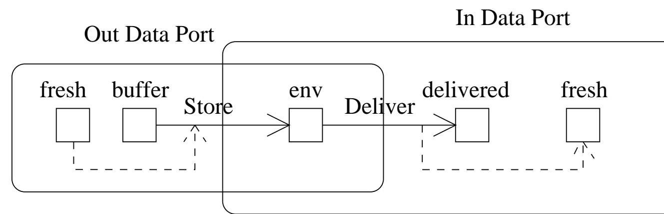

FIG. 5.2 – Structure d'un port data

l'aspect asynchrone des ports `data`. En effet, lorsqu'un producteur envoie une donnée, elle n'est pas disponible immédiatement chez le consommateur, mais après être passée par la zone tampon. Lors de la mise à jour du port le contenu de la zone tampon est recopié dans la variable accessible au thread, le marqueur passe à vrai et on efface le contenu de la zone tampon. Dans le cas où la zone tampon ne contient pas de valeur, on fait passer la valeur du marqueur à faux. Le thread peut continuer à accéder à l'ancienne valeur.

Un port `event` en entrée est constitué d'un compteur invisible au thread, d'une variable et d'un marqueur accessibles pour le thread. Le compteur associé au port est incrémenté à chaque fois qu'il reçoit un événement. On différencie deux politiques de mise à jour du port : `OneItem` et `AllItems`. Dans le premier cas on décrémente le compteur de un, on positionne la variable associée au thread à un et le marqueur à vrai. Dans le second cas la valeur du compteur est copiée dans la variable, et le marqueur passe aussi à vrai. Le compteur dispose d'une limite maximum ; si celle-ci est dépassée les nouveaux événements sont ignorés.

Un port `event data` en entrée a exactement le même comportement qu'un port event. Le compteur et la variable locale au thread sont remplacés par des files de données. Dans le cas `OneItem`, on gère la file comme une FIFO et on recopie les éléments un par un à chaque activation. Dans le cas `AllItems` on copie toute la file à chaque activation.

Les ports en sortie ont tous le même comportement, pendant son exécution le thread peut modifier le contenu d'une zone tampon et son marqueur associé. Au moment où la donnée ou l'événement doivent être envoyé, le système recopie le contenu de cette zone, si le marqueur est vrai, vers les ports destination, et positionne le marqueur à faux (figure 5.2).

Comportement temporel des ports Dans la version un d'AADL, les ports en entrée sont mis à jour à des dates très précises (lors de l'activation ou lors du début de l'exécution du thread). De même, l'envoi de données par un port data ne peut avoir lieu qu'à la complétion du thread. Le comportement des ports event et event data en sortie est plus libre, mais ne peut pas être précisé au niveau AADL. Un event ou event data peut être envoyé n'importe quand pendant l'exécution du thread, mais AADL ne permet pas de décrire à

quel instant avec précision. Dans sa version 2, AADL introduit des propriétés permettant de définir avec précision les instants de communications.

On associe à chaque port l'instant auquel il doit être mis à jour. Cet instant est caractérisé par un point de référence et éventuellement un délai. Pour les ports en entrée, le point de référence peut être l'activation ou le début de l'exécution du thread. Dans le second cas, on peut ajouter un décalage. Dans ce cas, le port sera mis à jour pendant l'exécution du thread. Pour les ports en sortie, le point de référence peut être le début de l'exécution, la complétion, et la date limite d'exécution. Dans le premier cas, un décalage doit être ajouté. On peut associer à un port une liste d'instants de mise à jour, ceci permet de décrire qu'un port est mis à jour plusieurs fois dans une même exécution.

Dans l'exemple suivant on décrit l'interface et les caractéristiques d'un thread. Chaque port est mis à jour à des instants particuliers. La figure 5.3 montre à quels instants de l'exécution du thread les ports reçoivent ou envoient des données.

```
thread th1
-- interface du thread et caractéristiques des ports
  features
    -- le port p1 est mis à jour à chaque activation
    p1 : in data port{
      Input_time => Dispatch;
      };
    -- le port p2 est mis à jour au début de chaque exécution
    p2 : in data port{
      Input_time => Execution;
      };
    -- le port p3 est mis à jour deux fois pendant l'exécution
    -- aux instants relatifs de date 3 et 5
    p3 : in data port{
      Input_time => Execution;
      offset_input_time => [3,5];
      };
    p4: out data port{
      Output_time => Execution;
      offset_output_time => [7];
      };
    -- un événement est envoyé sur le port p6 à la terminaison
    p5: out event port{
      Output_time => Complete;
      };
-- caractéristiques du thread
  properties
     Dispatch_Protocol => Periodic;
     Compute_Execution_time => 10 ;
     Deadline => 20;
     Period => 20;
```

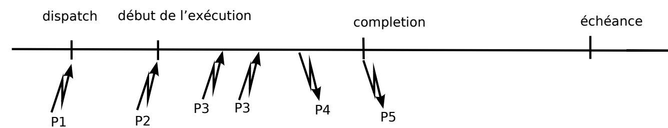

FIG. 5.3 – Exemple de mises à jour des ports

```
    Priority => 5;
end th1;
```

Dans le cas d'un thread apériodique, on a vu qu'un sous ensemble des ports event pouvait servir à déclencher l'activation du thread. À chacun de ces ports, on peut associer la liste des ports à mettre à jour. Par défaut, tous les ports qui ne font pas partie de cet ensemble sont mis à jour lors de l'activation du thread.

Les variables partagées

Les variables partagées sont modélisées par des data, on spécifie qu'un thread doit accéder à une data par un connecteur (`require data access`). Comme propriété de ce connecteur, on trouve l'intervalle de temps pendant lequel durant son exécution le thread doit accéder à la donnée. Le connecteur est relié à une data, cette data peut être protégée ou non. Dans le premier cas, on applique la politique PCP [SRL90] : lorsqu'un thread entre dans sa zone critique, il prend la priorité du thread de plus haute priorité pouvant accéder à la donnée. Dans le second cas, aucun mécanisme de protection n'est utilisé, on considère que l'ordonnancement des tâches garantit que la donnée n'est pas utilisée par deux threads en même temps[2]. Cette propriété sera à établir lors de l'étape de vérification.

5.1.3 Les modes

Le concept de mode dans AADL n'est pas tout à fait le même suivant les composants auxquels il s'applique. Dans le cas d'un thread, un mode peut être vu comme un état hiérarchique, les sous état étant définis à l'aide de l'annexe comportementale. Dans le cas d'un `system` ou d'un `process`, un mode représente une configuration de l'architecture du logiciel.

Au niveau du thread, les modes permettent de définir un premier niveau de comportement. Ils permettent aussi d'associer des propriétés à un état donné. À titre d'exemple, on peut associer un temps d'exécution en fonction du mode courant.

[2]Dans le cas des logiciels de vol satellite, l'ordonnancement statique des tâches peut garantir l'exclusion d'accès aux données partagées ; dans ce cas, la donnée n'est pas protégée.

Au niveau système, un mode représente une configuration du logiciel comportant notamment la définition d'un ensemble de threads et de connexions entre ces threads. Dans le cas d'un système embarqué, l'ensemble global des threads est statique, seul la présence effective de threads dans un mode donné varie ; il n'y a pas création dynamique de thread. Les modes introduisent donc une notion d'architecture "dynamique", et ce dans le cadre d'une configuration globalement statique.

Les modes threads

Un mode thread définit un état interne du thread. L'automate de mode du thread présente donc une vue abstraite du comportement du thread. Cet automate est défini par un ensemble d'états et un ensemble de transitions entre ces états. Une transition entre deux états peut être déclenchée soit par l'arrivée d'un événement sur un port, soit de manière interne au thread. Un thread peut changer de mode à deux instants de son cycle d'exécution, lors de son activation ou à la terminaison. Le changement de mode lors de l'activation est déclenché par l'arrivée d'un événement dans un des ports du thread. Le changement de mode à la terminaison du thread peut avoir été déclenché par un événement reçu au cours de l'exécution du thread ou par un événement interne. De nombreuses propriétés du thread peuvent être dépendantes de son mode. Le temps d'exécution du thread dépend du mode, le thread peut avoir un comportement différent par mode, donc un temps d'exécution particulier. De la même manière, l'accès aux ports et aux ressources partagées peut être différent suivant le mode courant du thread. Pour un thread apériodique, l'ensemble des ports capables de déclencher son activation peut aussi dépendre du mode.

Les modes systèmes

Un mode système est une configuration de l'architecture de l'application. Cette configuration comprend un ensemble de tâches actives et une topologie particulière des connexions. Les changements de modes sont contrôlés par un automate déterministe, chaque état de cet automate représente un mode. Les transitions entre deux modes sont déclenchées par la réception d'un événement.

La transition de mode Lorsque la transition de mode est déclenchée, le système commence par attendre la synchronisation d'un sous ensemble des threads périodiques. On appelle ces threads les threads critiques, dans AADL ils sont qualifiés de synchronisés. Lorsque ces threads sont synchronisés, le système arrête les threads de l'ancien mode ne faisant pas partie du nouveau mode et démarre les nouveaux threads. Les connexions sont aussi modifiées. Certains threads de l'ancien mode peuvent être autorisés à terminer leur

exécution dans le nouveau mode. Durant toute la durée de la transition, le système ne répond pas aux nouvelles requêtes de changement de mode.

5.2 Formalisation du modèle en TLA+

Dans cette partie, on présente la modélisation en TLA+ de notre AADL réduit. On commence par présenter les principaux choix de conception et l'architecture générale des modules TLA+ qui en découle. On présente ensuite les différentes facettes de notre modélisation : l'ordonnancement, les communications, les ressources partagées, le comportement interne des threads et enfin les modes.

5.2.1 Architecture générale et choix de conception

Deux types de modélisations ont été envisagés au début de cette étude. La première, la plus naturelle, consiste à représenter chaque thread par un automate avec son interface, ses propriétés, son comportement et des mécanismes de partage des ressources (processeurs ou données) et de se synchroniser avec les autres threads. Le comportement global de l'application est la composition des comportements de tous les threads. La seconde solution consiste à définir une machine virtuelle capable d'animer un modèle d'architecture. Cette machine virtuelle est un automate qui, paramétré par des constantes représentant un modèle d'application, peut simuler des exécutions de cette application. Le principal avantage de cette solution est de présenter de manière synthétique une formalisation du modèle d'exécution. De plus, la seconde solution semble plus adaptée à la génération automatique de configurations TLA à partir de modèles AADL. Le principal inconvénient de cette solution est de manipuler des variables plus complexes. Une simple variable dans la première solution devient un tableau [3] indexé par un ensemble d'objets du système qui associe à chacun de ces objets une valeur. Dans la première solution on décrit des composants autonomes ayant chacun leur relation `Next`, la composition de ces composants ainsi que leurs communications sont réalisées par la “ machine ” TLA sous-jacente. Dans la seconde solution, ces compositions et communications sont décrites explicitement. Il s'en suit la description explicite du modèle d'exécution en question.

L'architecture des modules TLA+ découle directement de ce choix de conception. On a, d'une part, un module `kernel` et un ensemble de modules ports qui modélisent le modèle d'exécution, et d'autre part, deux modules, `architecture model` et `thread behavior` qui sont une traduction de l'architecture de l'application décrite et du comportement des threads en TLA.

[3] En TLA, un tableau est représenté par une fonction, dans notre cas le domaine de cette fonction est l'ensemble des objets auxquels on associe une valeur.

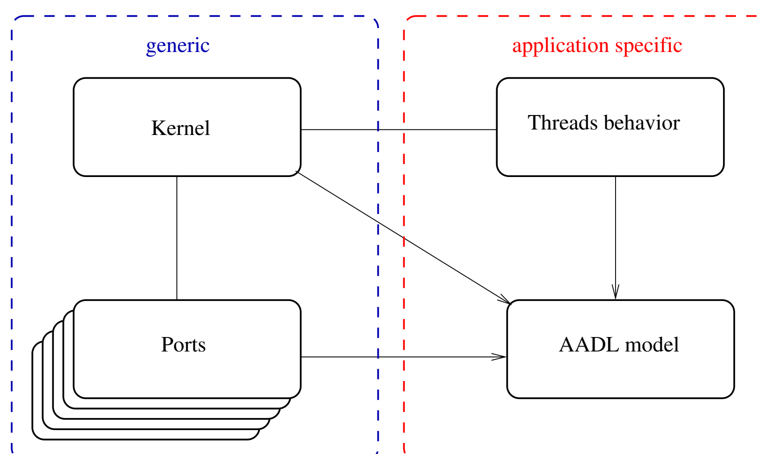

Fig. 5.4 – Architecture de la spécification

Le module kernel contient tous les mécanismes liés à l'ordonnancement, la gestion des ressources partagées, au comportement temporel des ports et à la gestion des modes. Les modules ports définissent le comportement fonctionnel des différents types de ports. Le module `architecture model` est un ensemble de constantes décrivant l'architecture de l'application modélisée. Enfin, le module `thread behavior` contient une traduction en TLA des automates associés aux threads et définis grâce à l'annexe comportementale. La figure 5.4 représente cette architecture. On présente le contenu de ces modules progressivement dans les parties suivantes. On commence par présenter la modélisation de l'ordonnanceur, on présente ensuite le modèle de communication par ports, puis par variables partagées. On décrit ensuite l'incidence de l'ajout des modes internes aux threads. Enfin, on montre comment peut être pris en compte le comportement interne des threads.

5.2.2 Définition d'un ordonnanceur simple

On commence par introduire les notions permettant de définir un ordonnanceur basique. L'ensemble des threads du système est représenté par la constante (`Thread`) définie dans le module `architecture model`. On considère qu'un thread peut se trouver dans trois états différents, en attente d'activation (`awaiting_dispatch`), prêt (`ready`), et en cours d'exécution (`running`). Pour représenter ces trois états, le module `kernel` contient trois variables permettant de représenter l'état courant des threads du système :

- `awaiting_dispatch` est un sous ensemble de `Thread` contenant les threads en attente d'activation ;
- `ready` un autre sous ensemble de thread, disjoint du premier, contient les threads prêts ;
- `computing_thread` est une variable simple prenant sa valeur dans l'en-

semble `Thread`, elle représente le seul thread éventuellement en cours d'exécution.

VARIABLES $awaiting_dispatch$, $ready$, $running$

$TypeInvariant \triangleq$

static type invariant

$\wedge\ awaiting_dispatch \in \text{SUBSET}\ Thread$
$\wedge\ ready \in \text{SUBSET}\ Thread$
$\wedge\ computing_thread \in Lifted_Thread$

dynamic type invariant

$\wedge\ awaiting_dispatch \cap ready = \{\}$
$\wedge\ awaiting_dispatch \cup ready = Thread$
$\wedge\ computing_thread \in ready \cup \{bot_thread\}$

L'ensemble `Lifted_Thread` correspond à l'ensemble `Thread` auquel on a ajouté la valeur `bot_thread`, cette valeur est utilisée dans le cas où aucun thread n'est en cours d'exécution, et représente donc à ce moment-là, la valeur de la variable `computing_thread`. On décrit ensuite les transitions entre ces états : `dispatch`, `resume`, `preempt`, et `complete`. On sépare toutes ces transitions en deux parties distinctes, la garde et les actions à effectuer.

Activation d'un thread

Un thread est activé s'il est périodique et qu'il arrive en début de période ou s'il est apériodique et qu'un de ses ports event pouvant déclencher l'activation contient un élément. Cette condition s'exprime en TLA de la manière suivante :

$$
\begin{aligned}
&can_dispatch(th) \triangleq \\
&\quad \vee\ th \in Periodic \wedge period_timer[th] = 0 \\
&\quad \vee\ th \in Aperiodic \wedge \exists\, p \in In_event_port : \\
&\quad\ \ p \in dispatch_ports[th] \wedge iep_event_cpt[p] > 0
\end{aligned}
$$

Dans cette expression `Aperiodic` et `Periodic` sont deux ensembles constants formant une partition de l'ensemble `Thread`. `In_event_port` représente l'ensemble des ports event du système. Enfin, `dispatch_ports` est une fonction qui associe à chaque thread apériodique l'ensemble de ses ports pouvant déclencher son activation. Toutes ces constantes sont définies dans le module `architecture model` et proviennent directement du modèle AADL. On trouve ensuite dans cette expression deux variables, `iep_event_cpt[p]` et `period_timer[th]`. La première est une fonction qui associe à chaque port event en entrée la valeur de son compteur d'événements. La seconde associe à chaque thread périodique une horloge qui compte le temps entre deux activations. Enfin, le paramètre `th` représente un thread. Cette expression retourne un booléen lors de son évaluation, vrai si le thread `th` est prêt à être activé, faux sinon.

Au moment de son activation, le thread quitte son état d'attente et passe dans l'état prêt, ses ports et ses horloges sont mis à jours. Si le thread est périodique, tous ses ports sont mis à jour. Mais si le thread est apériodique, seule une sous partie de ses ports est modifiée. L'ensemble des ports à modifier dépend de du port ayant déclenché le dispatch. On met ensuite à jour les différentes horloges associées au thread.

$$
\begin{array}{l}
\mathit{dispatch} \triangleq \\
\text{LET } \mathit{th} \triangleq \text{ CHOOSE } \mathit{th} \in \mathit{awaiting_dispatch} : \mathit{can_dispatch}(\mathit{th}) \text{IN} \\
\quad \land\, \mathit{awaiting_dispatch}' = \mathit{awaiting_dispatch} \setminus \{\mathit{th}\} \\
\quad \land\, \mathit{ready}' = \mathit{ready} \cup \{\mathit{th}\} \\
\quad \land\, \mathit{deadline_timer}' = [x \in \mathit{Thread} \mapsto \\
\qquad \text{IF } x = \mathit{th} \text{ THEN } \mathit{Deadline}[\mathit{th}] \text{ ELSE } \mathit{deadline_timer}[x]] \\
\quad \land\, \mathit{execution_timer}' = [x \in \mathit{Thread} \mapsto \\
\qquad \text{IF } x = \mathit{th} \text{ THEN } 0 \text{ ELSE } \mathit{execution_timer}[x]] \\
\quad \land\, \mathit{period_timer}' = [x \in \mathit{Periodic} \mapsto \\
\qquad \text{IF } x = \mathit{th} \text{ THEN } \mathit{Period}[\mathit{th}] \text{ ELSE } \mathit{period_timer}[x]]
\end{array}
$$

L'expression TLA `LET ... IN` nous permet de choisir un élément satisfaisant un prédicat dans un ensemble. Les expressions `set_port_dispatch_periodic(th)` et `set_port_dispatch_aperiodic(th,ep)` sont des actions servant à mettre à jour les ports. Les variables représentant les horloges (`deadline_timer`, `execution_timer`, `period_timer`) sont des fonctions qui associent une valeur d'horloge à chaque thread. Au moment de l'activation d'un thread, on veut réinitialiser seulement la valeur associée à ce thread. On décrit alors la nouvelle fonction comme étant modifiée pour le thread considéré et identique pour toutes les autres valeurs du domaine de la fonction. On peut apparenter cette mise à jour à l'utilisation de l'opérateur de surcharge d'une fonction en B. On montre plus tard dans cette partie comment évoluent ces horloges.

La description de cette action n'est pas complète, mais elle est suffisante pour la présentation du mécanisme d'ordonnancement. On complétera cette description par la suite 5.2.

Gestion de l'ordonnancement

Dans un premier temps, on doit être capable d'élire le thread prêt de plus haute priorité. L'expression TLA suivante est vraie si le thread passé en paramètre a une priorité supérieure à tous les autres threads prêts. On introduit ici une nouvelle constante du module `architecture model`, `Priority` est une fonction qui associe à chaque thread sa priorité.

$$
\begin{array}{l}
\mathit{maxPrio}(\mathit{th}) \triangleq \\
\quad \forall\, t \in \mathit{ready} : \mathit{Priority}[t] \leq \mathit{Priority}[\mathit{th}]
\end{array}
$$

On doit commencer l'exécution d'un nouveau thread si le thread en cours d'exécution n'a plus la priorité maximum. Cette condition est traduite par la condition suivante.

$$\begin{array}{l} canResume \triangleq \\ \quad \land \neg maxPrio(computing_thread) \end{array}$$

Si cette condition est vraie on doit élire et commencer à exécuter le nouveau thread de plus haute priorité. Lorsqu'un thread commence son exécution, il ne quitte pas l'ensemble des threads actifs. La variable `computing_thread` peut avoir une valeur particulière appelée `bot_thread`, cette valeur est utilisée lorsqu'aucun thread n'est actif. On associe à cette valeur une priorité minimale. Une précondition de cette opération est que l'ensemble `ready` est non vide.

$$\begin{array}{l} resume \triangleq \\ \quad \text{LET } mt \triangleq \text{ CHOOSE } mt \in ready : maxPrio(mt) \text{IN} \\ \qquad \land computing_thread' = mt \end{array}$$

La complétion

On considère dans notre modèle qu'un thread a toujours le même temps d'exécution. Dans la littérature, le temps d'exécution d'une tâche est souvent donné comme un intervalle entre un meilleur temps d'exécution et un pire temps d'exécution. Cette approche est difficile à utiliser lorsqu'on envisage d'étudier la validité de propriétés par model checking. L'utilisation d'intervalles fait exploser le nombre d'états générés par le model checker. De plus, dans la plupart des cas, on est intéressé par le pire cas d'exécution. On verra par la suite que l'on peut légèrement relâcher cette contrainte en utilisant la notion de mode interne aux threads. Un thread termine donc son exécution lorsque l'horloge qui compte son temps d'exécution atteint la valeur de son temps d'exécution total. La constante `Execution_time` appartient au module `architecture model`, et associe un temps d'exécution à chaque thread.

$$\begin{array}{l} hasComplete \triangleq \\ \quad \land computing_thread \neq bot_thread \quad \text{il y a bien un thread courant} \\ \quad \land execution_timer[computing_thread] \\ \qquad = Execution_time[computing_thread] \end{array}$$

On effectue très peu d'actions au moment de la complétion : le thread en cours d'exécution quitte l'état prêt et revient dans l'état en attente d'activation. Le processeur est libéré, il n'y a plus de threads en cours d'exécution. Et enfin, on envoie éventuellement des données ou des événements par les ports.

$$
\begin{aligned}
&complete \triangleq \\
&\quad \land\ awaiting_dispatch' = awaiting_dispatch \cup \{computing_thread\} \\
&\quad \land\ ready' = ready \setminus \{computing_thread\} \\
&\quad \land\ computing_thread' = bot_thread
\end{aligned}
$$

La gestion de l'écoulement du temps

Afin de modéliser l'écoulement du temps en TLA, on attache trois horloges à chaque thread, une associé à la période du thread, la seconde à son temps d'exécution et la troisième à son échéance, et on définit une opération, `tick`. Le modèle d'écoulement du temps adopté est celui où on distingue des opérations discrètes où seul l'espace d'état évolue et une opération de délai où seul le temps évolue. Les horloges associées aux threads ont déjà été présentées dans 2.2.2. Elles sont initialisées au moment de l'activation du thread. L'opération `tick` incrémente une variable `now`, généralement dite de temps absolu, cette variable est initialisée à zéro et augmente continuellement. Elle n'est ni modifiée ni accédée dans le modèle, elle ne fait pas partie des variables prises en compte pour le model checking. Cette variable est utile lorsqu'on doit lire le résultat de l'analyse du model checker. Elle permet de mieux suivre la chronologie des divers états. On utilise ici la méthode proposée par L.Lamport dans [Lam05].

$$
\begin{aligned}
&Tick \triangleq \\
&\quad \land\ now' = now + 1 \\
&\quad \land\ deadline_timer' = [t \in Thread \mapsto \\
&\quad\quad \textsc{if}\ \lor\ t \in ready \\
&\quad\quad\quad \lor\ (t \in awaiting_dispatch \land deadline_timer[t] > 0)\ \textsc{then} \\
&\quad\quad deadline_timer[t] - 1 \\
&\quad\quad\ \textsc{else}\ \ deadline_timer[t]] \\
&\quad \land\ execution_timer' = [t \in Thread \mapsto \\
&\quad\quad \textsc{if}\ t = computing_thread\ \textsc{then} \\
&\quad\quad execution_timer[t] + 1 \\
&\quad\quad\ \textsc{else}\ \ execution_timer[t]] \\
&\quad \land\ period_timer' = [t \in Periodic \mapsto \\
&\quad\quad period_timer[t] - 1]
\end{aligned}
$$

Dans le but de réduire le nombre d'états générés par le model checker on pourrait calculer dans cette opération l'instant du prochain événement à prendre en compte, et faire avancer directement le modèle jusqu'à cet instant.

Automate général de l'ordonnanceur

On présente ici l'opération `Next` qui décrit la transition globale du système. Dans cette expression, si une des conditions présentées est vraie on

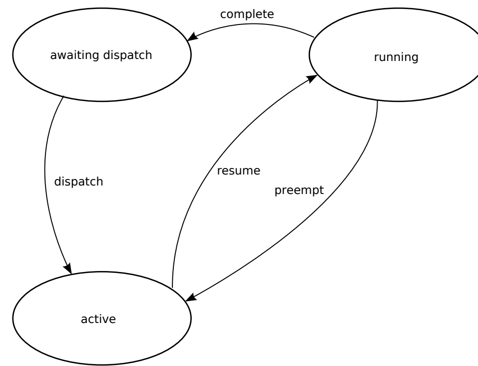

FIG. 5.5 – Automate de gestion simplifié des threads

exécute l'opération correspondante. L'ordre dans lequel on évalue les expressions est important. Un thread qui se termine peut devoir envoyer des données à un thread qui doit être activé. De la même manière, avant d'élire le thread de plus haute priorité, on veille à avoir activé tous les threads qui pouvaient l'être. Enfin, on fait avancer le temps seulement si aucune autre action n'est possible. On peut représenté le code TLA suivant par la figure 5.5.

$$
\begin{array}{l}
Next \triangleq \\
\quad \text{IF } hasComplete \text{ THEN} \\
\quad\quad complete \\
\quad \text{ELSE IF } \exists\, th \in awaiting_dispatch : can_dispatch(th) \text{ THEN} \\
\quad\quad dispatch \\
\quad \text{ELSE IF } canResume \text{ THEN} \\
\quad\quad resume \\
\quad \text{ELSE } Tick
\end{array}
$$

5.2.3 Spécification des ports

Spécification fonctionnelle des ports

Chaque type de ports (data, event, event data, en entrée ou en sortie) est décrit dans un module TLA séparé. Chaque module représente un ensemble de ports, ainsi le module `OutDataPort` décrit le comportement commun à tous les ports data du système. Dans chacun de ces modules, on retrouve les constantes permettant de configurer les ports, les variables représentant l'état des ports et des opérations de manipulation des ports. L'ensemble de ces modules sont instanciés dans un module `ports`, ce module est lui-même instancié dans le module noyau. À titre d'exemple, l'appel de l'opération `store` sur un ensemble de ports en sortie `ODP` depuis le module `kernel` est noté `ports!odp!store(ODP)`.

On décrit, dans les parties suivantes, en détail les modules concernant

les ports data. On indique ensuite les différences principales des ports event et event data.

Les ports data en sortie Dans ce module on décrit le comportement des ports data en sortie. On commence par décrire les constantes dont on va avoir besoin. L'ensemble `Data` contient les données utilisables. Les ensembles `Out_data_port` et `In_data_port` représentent les ports de données en sortie et en entrée du système. Enfin `bot_data` est une valeur spéciale de l'ensemble `Data`, cette constante représente une valeur non accessible à l'utilisateur. Les variables `buffer` et `fresh` sont accessibles au thread émetteur. Lorsqu'un thread doit envoyer une donnée, il met à jour ces variables. La fonction `data_connection` décrit les connexions entre les ports data en entrée et en sortie. Enfin, `env` est la fonction qui associe un tampon à chaque port en entrée. Lors de l'initialisation du système le contenu de tous les ports data en sortie est initialisé avec la valeur spéciale `bot_data`, de même le marqueur `fresh` est initialisé à faux.

On définit dans ce module une opération `store`. Cette opération lit le contenu des ports de sortie et le transmet aux ports en entrée. Cette opération prend en paramètre un ensemble de ports data en sortie. Lors de l'appel à cette opération on va modifier les ports data en entrée associés aux ports présents dans le paramètre de l'opération. On modifie la valeur associée à un port data en entrée par la fonction `env` si ce port est connecté à un des ports en sortie présent dans l'ensemble paramètre de l'opération et que le port en sortie à bien été modifié. Enfin, les marqueurs associés aux ports data en sortie présent dans l'ensemble `dp` passent à faux, i.e. la donnée qui leur est associée a bien été envoyée.

MODULE *OutDataPort*

CONSTANTS $Data$, Out_data_port, In_data_port, bot_data

ASSUME
$\wedge\ bot_data \in Data$

VARIABLES env, $data_connection$, $buffer$, $fresh$

$TypeInvariant \triangleq$
$\wedge\ env \in [In_data_port \rightarrow Data]$
$\wedge\ data_connection \in [In_data_port \rightarrow Out_data_port]$
$\wedge\ buffer \in [Out_data_port \rightarrow Data]$
$\wedge\ fresh \in [Out_data_port \rightarrow \text{BOOLEAN}]$

$Init \triangleq$
$\land\ buffer = [x \in Out_data_port \mapsto bot_data]$
$\land\ fresh = [x \in Out_data_port \mapsto \text{FALSE}]$

$store(dp) \triangleq$
$\land\ dp \in \text{SUBSET}\ Out_data_port$
$\land\ env' = [x \in In_data_port \mapsto$
$\text{IF}\ data_connection[x] \in dp \land fresh[data_connection[x]]$
$\text{THEN}\ buffer[data_connection[x]]$
$\text{ELSE}\ env[x]]$
$\land\ fresh' = [x \in Out_data_port \mapsto$
$\text{IF}\ x \in dp\ \text{THEN FALSE}$
$\text{ELSE}\ fresh[x]]$

Les ports data en entrée On retrouve ici une partie des constantes présentées dans la description de ports data en sortie, l'ensemble des données, la valeur spéciale `bot_data` et l'ensemble des ports data en entrée. De plus, `env` est la même variable que dans le module précédent. Deux nouvelles variables apparaissent, `delivered`, la valeur du port auquel le thread destination a accès, et `fresh` un marqueur associé à cette valeur. Ces variables sont toutes des tableaux qui associent une valeur à chaque port data en entrée. Ils sont initialisés avec la valeur `bot_data` pour les variables `env` et `delivered` et faux pour `fresh`.

L'unique opération de ce module copie le contenu du tampon dans la variable accessible au thread et met à jour le marqueur correspondant pour un ensemble de ports data en entrée donné. Cet ensemble est déterminé par le paramètre de l'opération. La variable `delivered` associée à un port est mise à jour si le port est dans l'ensemble paramètre et si le tampon qui lui est associé contient une donnée valide. De la même manière le marqueur passe à vrai si `delivered` a été modifié et à faux sinon. Ainsi, un thread peut accéder à l'ancienne valeur s'il n'a rien reçu, mais il sait que la variable qu'il consulte n'a pas été mise à jour. Enfin, le contenu des tampons utilisés est effacé (ils prennent la valeur `bot_data`) afin de pouvoir détecter l'arrivée d'une nouvelle valeur.

MODULE *InDataPort*

CONSTANTS bot_data, $Data$, In_data_port

ASSUME
$\land\ bot_data \in Data$

VARIABLES
$delivered$, $fresh$, env

$TypeInvariant \triangleq$
$\land\ delivered \in [In_data_port \rightarrow Data]$
$\land\ fresh \in [In_data_port \rightarrow \text{BOOLEAN}]$
$\land\ env \in [In_data_port \rightarrow Data]$

$Init \triangleq$
$\land\ delivered = [x \in In_data_port \mapsto bot_data]$
$\land\ fresh = [x \in In_data_port \mapsto \text{FALSE}]$
$\land\ env = [x \in In_data_port \mapsto bot_data]$

$deliver(P) \triangleq$
$\land\ P \subseteq In_data_port$
$\land\ delivered' = [x \in In_data_port \mapsto \text{IF } x \in P$
$\land\ env[x] \neq bot_data \text{ THEN } env[x] \text{ ELSE } delivered[x]]$
$\land\ fresh' = [x \in In_data_port \mapsto$
$\text{IF } x \in P \text{ THEN } env[x] \neq bot_data \text{ ELSE } fresh[x]]$
$\land\ env' = [x \in In_data_port \mapsto$
$\text{IF } x \in P \text{ THEN } bot_data \text{ ELSE } env[x]]$

Les autres types de ports Les différences entre le fonctionnement (transfert des données) des ports data et des autres types de ports sont minimes. La principale différence est la nature du tampon, dans le cas des ports event il s'agit d'un compteur, pour les ports event data ce sont des files de données. Ces tampons ont une taille maximum définie par une constante. Si la file et pleine, lors de l'arrivée d'un nouvel élément, le comportement standard et d'ignorer les nouveaux message. Enfin, la mise à jour d'un port en entrée peut se faire suivant deux politiques. Soit on ne délivre que l'élément le plus ancien de la file soit on recopie toute la file.

Spécification temporelle des ports

Après avoir présenté des opérations permettant de manipuler les ports, on définit à quels instants ces opérations vont être appelées. Pour cela, on doit en premier lieu donner les moyens de décrire l'instant où l'interaction a lieu. On ajoute deux constantes `input_time` et `output_time` à notre modèle. Ces constantes associent à chaque port l'instant du cycle de vie du

thread auquel le port doit être mis à jour. Pour les ports en entrée, ce peut être au moment de l'activation (`Dispatch`), lors du début de l'exécution (`Execution`) ou pas du tout (`None`). Le dernier cas est utilisé pour spécifier que dans le mode courant seul une partie des ports du thread sont mis à jour. . Pour les ports en sorties, ces instants de communication possibles peuvent être le début de l'exécution, la complétion, à la deadline, ou pas du tout. Dans le cas où le point de référence est le début de l'exécution, deux nouvelles constantes `offset_input_time` et `offset_output_time` permettent de définir à quel instant de l'exécution du thread les ports doivent être mis à jour. Dans le cas où un port doit être mis à jour plusieurs fois pendant son exécution, ces propriétés permettent de lui associer une liste de valeurs représentant la liste des instants de communication. Les décalages définis par ces constantes sont relatifs au temps d'exécution du thread, pas à l'horloge globale du système.

On se base ici sur les propriétés définies dans AADL V2. Dans la version un d'AADL, les possibilités de description temporelle du comportement des ports étaient plus limitées. Les mises à jour des ports en entrée ne pouvaient avoir lieu que lors du `dispatch` et au début de l'exécution dans le cas de connexions immédiates (cf. 3.2.3). L'envoi de donnée par des ports data n'était possible qu'à la fin de l'exécution des threads. Et on ne pouvait pas spécifier précisément l'instant auquel un event ou event data devait être envoyé.

Dans la suite de cette partie, on présente les opérations de mise à jour des ports. Ces opérations seront par la suite ajoutées à la relation `Next` de base. Cet ajout se fait de deux manières différentes :

– soit il s'agit d'une opération à synchroniser avec une opération existante, dans ce cas on fera la conjonction entre l'ancienne et l'opération de mise à jour ;
– soit il s'agit d'une nouvelle opération indépendante, et dans ce cas on ajoutera une alternative à la relation `Next`.

Envoi à la complétion Dans cette opération, on veut calculer un ensemble de ports du thread et envoyer des données ou des événements par ces ports. On définit `ODP` comme étant l'ensemble des ports appartenant à `th` et dont la donnée doit être envoyée à la complétion du thread. On calcule de la même manière un ensemble de ports event et event data. Ensuite, il ne reste plus qu'à appeler les opérations de mises à jour définies dans les modules ports en les paramétrant par les ensembles que l'on vient de calculer. Afin de mettre à jour ces ports à l'instant voulu, il suffit d'ajouter cette action à la conjonction d'actions de l'opération `complete`.

$send_at_completion(th) \triangleq$
  LET $ODP \triangleq \{p \in thread_out_data_port[th] :$
    $output_time[p] = \text{“completion”}\}$IN
  LET $OEP \triangleq \{p \in thread_out_event_port[th] :$
    $output_time[p] = \text{“completion”}\}$IN
  LET $OEDP \triangleq \{p \in thread_out_event_data_port[th] :$
    $output_time[p] = \text{“completion”}\}$IN
  $\wedge \quad ports!odp!store(ODP)$
  $\wedge \quad ports!oep!RaiseEvent(OEP)$
  $\wedge \quad ports!oedp!RaiseEvent(OEDP)$

Envoi à l'échéance (deadline) Contrairement à l'envoi à la complétion, on n'a défini jusqu'à présent aucune action à effectuer au moment de la deadline d'un thread. On va donc devoir ajouter une alternative à notre automate global (le `Next`). Comme pour les autres transitions on définit dans un premier temps la garde puis les actions à effectuer. La condition `must_send_at_deadline` est vraie si l'horloge `deadline_timer` associée au thread paramètre `th` est arrivée à zéro et si un des ports de ce thread doit envoyer une donnée à cet instant.

L'action à effectuer est très proche de celle de l'envoi à la complétion. La principale différence est que l'on peut avoir plusieurs threads qui arrivent en même temps à leur deadline. Dans l'opération `send_at_deadline`, on choisit un de ces threads, comme dans l'opération `send_at_completion` on calcule trois ensembles de ports à mettre à jour et on appelle les opérations des modules ports permettant d'envoyer des données ou des événements.

$must_send_at_deadline(th) \triangleq$
  $\wedge\ deadline_timer[th] = 0$
  $\wedge\ \exists\, p \in thread_out_data_port[th]$
    $\cup\ thread_out_event_port[th]$
    $\cup\ thread_out_event_data_port[th] :$
      $output_time[p] = \text{“deadline”}$

$send_at_deadline \triangleq$
LET $this \triangleq$ CHOOSE $this \in awaiting_dispatch$
  $: must_send_at_deadline(this)$IN
  LET $ODP \triangleq \{dp \in thread_out_data_port[this] :$
    $\wedge\ output_time[dp] = \text{“deadline”}\}$IN
  LET $OEP \triangleq \{ep \in thread_out_event_port[this] :$
    $\wedge\ output_time[ep] = \text{“deadline”}\}$IN
  LET $OEDP \triangleq \{edp \in thread_out_event_data_port[this] :$
    $\wedge\ output_time[edp] = \text{“deadline”}\}$IN
  $\wedge\ ports!odp!store(ODP)$
  $\wedge\ ports!oep!RaiseEvent(OEP)$
  $\wedge\ ports!oedp!RaiseEvent(OEDP)$

Mise à jour de l'environnement d'un thread à l'activation En fonction de la nature du thread, périodique ou apériodique, la mise à jour des ports lors de son activation ne se fait pas tout à fait de la même manière. Pour un thread périodique, le cas le plus simple, on met à jour tous les ports du thread. Comme dans les précédentes opérations, on commence par calculer les ensembles de ports sur lesquels on veut agir. Chaque ensemble est défini comme les ports du thread ayant pour `input_time` la valeur `dispatch`. On appelle ensuite les opérations de mise à jour de ces ports (`deliver`).

Dans le cas d'un thread apériodique, on ne va mettre à jour que les ports dépendants du port qui a déclenché l'activation. La fonction `ports_to_deliver` définit, pour chaque port capable de déclencher l'activation du thread, l'ensemble des ports à mettre à jour.

$$
\begin{array}{l}
\textit{set_port_dispatch_periodic}(\textit{this}) \triangleq \\
\quad \text{LET } \textit{IDP} \triangleq \{p \in \textit{thread_in_data_port}[\textit{this}] \\
\qquad : \textit{input_time}[p] = \text{“dispatch”}\} \text{IN} \\
\quad \text{LET } \textit{IEP} \triangleq \{p \in \textit{thread_in_event_port}[\textit{this}] \\
\qquad : \textit{input_time}[p] = \text{“dispatch”}\} \text{IN} \\
\quad \text{LET } \textit{IEDP} \triangleq \{p \in \textit{thread_in_event_data_port}[\textit{this}] \\
\qquad : \textit{input_time}[p] = \text{“dispatch”}\} \text{IN} \\
\qquad \land \textit{ports}!\textit{idp}!\textit{deliver}(\textit{IDP}) \\
\qquad \land \textit{ports}!\textit{iep}!\textit{deliver}(\textit{IEP}) \\
\qquad \land \textit{ports}!\textit{iedp}!\textit{deliver}(\textit{IEDP})
\end{array}
$$

$$
\begin{array}{l}
\textit{set_port_dispatch_aperiodic}(\textit{this}, \textit{ep}) \triangleq \\
\qquad \land \textit{ports}!\textit{idp}!\textit{deliver}(\textit{In_data_port} \cap \textit{ports_to_deliver}[\textit{ep}]) \\
\qquad \land \textit{ports}!\textit{iep}!\textit{deliver}(\textit{In_event_port} \cap \textit{ports_to_deliver}[\textit{ep}]) \\
\qquad \land \textit{ports}!\textit{iedp}!\textit{deliver}(\textit{In_event_data_port} \cap \textit{ports_to_deliver}[\textit{ep}])
\end{array}
$$

Mise à jour d'un port au cours de l'exécution Les opérations d'envoi ou de réception au cours de l'exécution du thread sont très similaires, on ne présente ici que la partie émission. Comme pour l'envoi de message à la deadline, on va devoir ajouter une transition à l'automate global. La garde de cette transition est simple. La condition `must_send` exprime qu'il existe un port appartenant au thread en cours d'exécution dont la propriété `output_time` vaut `execution`. De plus, la valeur de l'horloge associée au temps d'exécution du thread doit apparaître dans la liste des instants auxquels ce port doit envoyer une donnée. Cette liste d'instants est définie par la constante `offsets_output_time`.Enfin le port ne doit pas déjà avoir été mis à jour.

L'opération d'envoi ressemble énormément à celle de l'envoi à la complétion. La principale différence est que le calcul des ensembles de ports est un peu plus complexe. Ici, on doit vérifier que l'exécution du thread a suffisamment progressé, c'est à dire que la valeur du compteur représentant le

temps d'exécution du thread apparaît bien dans la liste des instant où doit communiquer le thread.

$$
\begin{array}{l}
\mathit{must_send} \triangleq \\
\land\ \mathit{computing_thread} \neq \mathit{bot_thread} \\
\land\ \exists\, p \in \mathit{thread_out_data_port}[\mathit{computing_thread}] \\
\quad \cup\ \mathit{thread_out_event_port}[\mathit{computing_thread}] \\
\quad \cup\ \mathit{thread_out_event_data_port}[\mathit{computing_thread}] : \\
\qquad \land\ \mathit{output_time}[p] = \text{“execution”} \\
\qquad \land\ \mathit{execution_timer}[\mathit{computing_thread}] \in \mathit{offsets_output_time}[p] \\
\qquad \land\ (\mathit{oep_fresh}[p] \lor \mathit{odp_fresh}[p] \lor \mathit{oedp_fresh}[p])
\end{array}
$$

$$
\begin{array}{l}
\mathit{send_port} \triangleq \\
\quad \text{LET } \mathit{ODP} \triangleq \{p \in \mathit{thread_out_data_port}[\mathit{computing_thread}] : \\
\qquad \land\ \mathit{output_time}[p] = \text{“execution”} \\
\qquad \land\ \mathit{execution_timer}[\mathit{computing_thread}] \in \mathit{offsets_output_time}[p] \text{IN} \\
\quad \text{LET } \mathit{OEP} \triangleq \{p \in \mathit{thread_out_event_port}[\mathit{computing_thread}] : \\
\qquad \land\ \mathit{output_time}[p] = \text{“execution”} \\
\qquad \land\ \mathit{execution_timer}[\mathit{computing_thread}] \in \mathit{offsets_output_time}[p] \text{IN} \\
\quad \text{LET } \mathit{OEDP} \triangleq \{p \in \mathit{thread_out_event_data_port}[\mathit{computing_thread}] : \\
\qquad \land\ \mathit{output_time}[p] = \text{“execution”} \\
\qquad \land\ \mathit{execution_timer}[\mathit{computing_thread}] \in \mathit{offsets_output_time}[p] \text{IN} \\
\qquad\ \land\ \mathit{ports}!\mathit{odp}!\mathit{store}(\mathit{ODP}) \\
\qquad\ \land\ \mathit{ports}!\mathit{oep}!\mathit{RaiseEvent}(\mathit{OEP}) \\
\qquad\ \land\ \mathit{ports}!\mathit{oedp}!\mathit{RaiseEvent}(\mathit{OEDP})
\end{array}
$$

Les opérations que l'on a définies ici vont être introduites dans le modèle de manière différentes. Pour les mises à jour de ports en cours d'exécution, ou à l'échéance on va ajouter des alternatives à l'opération `Next`. Pour les mises à jour lors de l'activation, ou de la terminaison du thread, on ajoute les actions correspondantes aux opérations déjà définies.

$$
\begin{array}{l}
\mathit{Next} \triangleq \\
\quad \text{IF } \mathit{must_send} \text{ THEN} \\
\qquad \mathit{send_port} \\
\quad \text{ELSE IF } \mathit{hasComplete} \text{ THEN} \\
\qquad \mathit{complete} \\
\quad \text{ELSE IF } \exists\, th \in \mathit{awaiting_dispatch} : \mathit{must_send_at_deadline}(th) \text{ THEN} \\
\qquad \mathit{send_at_deadline} \\
\quad \text{ELSE IF } \exists\, th \in \mathit{awaiting_dispatch} : \mathit{can_dispatch}(th) \text{ THEN} \\
\qquad \mathit{dispatch} \\
\quad \text{ELSE IF } \mathit{canResume} \text{ THEN} \\
\qquad \mathit{resume} \\
\quad \text{ELSE IF } \mathit{must_set_port} \text{ THEN} \\
\qquad \mathit{set_port} \\
\quad \text{ELSE } \mathit{Tick}
\end{array}
$$

$$
\begin{array}{l}
complete \triangleq \\
\quad \land\ awaiting_dispatch' = awaiting_dispatch \cup \{computing_thread\} \\
\quad \land\ ready' = ready \setminus \{computing_thread\} \\
\quad \land\ computing_thread' = bot_thread \\
\quad \land\ send_at_completion(computing_thread)
\end{array}
$$

$$
\begin{array}{l}
dispatch \triangleq \\
\text{LET } th \triangleq \text{ CHOOSE } th \in awaiting_dispatch : can_dispatch(th) \text{IN} \\
\quad \land\ awaiting_dispatch' = awaiting_dispatch \setminus \{th\} \\
\quad \land\ ready' = ready \cup \{th\} \\
\quad \land \text{ IF } th \in Periodic \text{ THEN} \\
\qquad \land\ set_port_dispatch_periodic(th) \\
\quad\ \text{ELSE } \exists\, ep \in dispatch_ports[th] : \\
\qquad \land\ iep_event_cpt[ep] > 0 \\
\qquad \land\ set_port_dispatch_aperiodic(th,\ ep) \\
\quad \land\ deadline_timer' = [x \in Thread \mapsto \\
\qquad \text{IF } x = th \text{ THEN } Deadline[th] \text{ ELSE } deadline_timer[x]] \\
\quad \land\ execution_timer' = [x \in Thread \mapsto \\
\qquad \text{IF } x = th \text{ THEN } 0 \text{ ELSE } execution_timer[x]] \\
\quad \land\ period_timer' = [x \in Periodic \mapsto \\
\qquad \text{IF } x = th \text{ THEN } Period[th] \text{ ELSE } period_timer[x]]
\end{array}
$$

5.2.4 Les données partagées

Dans notre modèle, on utilise deux types de ressources partagées, des ressources protégées et non protégées. Les ressources protégées le sont par un mécanisme explicite d'exclusion mutuelle, le protocole que l'on a choisi d'implémenter est IPCP(`Immediate Priority Ceiling Protocol`[RC04]). Ce protocole a principalement été choisi pour sa simplicité d'implantation : lorsqu'une tâche accède à une ressource, elle change au plus une fois de priorité pour prendre la plus haute priorité. Dès qu'un thread entre dans une zone critique, il prend la plus haute priorité des threads pouvant accéder à cette ressource. Les ressources non protégées servent à modéliser des ressources partagées dont l'intégrité est censée être garantie par l'ordonnancement des threads. On vérifiera que ces données non protégées ne sont jamais utilisées par deux threads au même instant.

On introduit ici de nouvelles constantes permettant de décrire les ressources et leurs propriétés :

- `SharedData` : l'ensemble des ressources partagées,
- `Protected` : l'ensemble des ressources partagées protégées,
- `Unprotected` : l'ensemble des ressources partagées non protégées,
- `data_Priority` : la priorité d'une ressource (plus haute priorité des threads pouvant accéder à la ressource),
- `accessing_thread` : fonction qui associe à chaque donnée la liste des threads pouvant l'utiliser,

- `access_time` : fonction qui associe à un couple donnée, thread l'intervalle pendant lequel le thread accède à la ressource.

On ajoute aussi de nouvelles variables au modèle :

- `awaiting_resource` : l'ensemble des threads en attente d'une ressource,
- `AccessedBy` : une fonction qui associe à chaque donnée le thread qui l'utilise,
- `actual_Priority` : la priorité courante du thread.

Verrouiller et déverrouiller une ressource

On commence par définir deux opérations permettant de modifier l'état d'une ressource partagée. La variable `AccessedBy` associe à chaque ressource le nom du thread qui est en train de l'utiliser. On utilise la valeur spéciale `bot_thread` dans le cas où la ressource est libre. La première opération, `lock_resource` modifie cette variable, elle associe à la ressource passée en paramètre le thread en cours d'exécution. La seconde, `release_resource`, associe à la ressource la valeur `bot_thread` signifiant que la ressource est libre.

$$
\begin{array}{l}
\mathit{lock_resource}(sd) \triangleq \\
\quad \land\ \mathit{AccessedBy}' = [d \in \mathit{SharedData} \mapsto \\
\qquad \textsc{if}\ d = sd \\
\qquad\quad \textsc{then}\ \mathit{computing_thread}\ \textsc{else}\ \mathit{AccessedBy}[d]]
\end{array}
$$

$$
\begin{array}{l}
\mathit{release_resource}(sd) \triangleq \\
\quad \land\ \mathit{AccessedBy}' = [d \in \mathit{SharedData} \mapsto \\
\qquad \textsc{if}\ d = sd \\
\qquad\quad \textsc{then}\ \mathit{bot_thread}\ \textsc{else}\ \mathit{AccessedBy}[d]]
\end{array}
$$

Accès à une ressource

On va de nouveau enrichir notre automate d'une nouvelle transition, la garde de cette transition devient vraie lorsque le thread en cours d'exécution entre dans sa section critique. L'action associée à cette transition verrouille la ressource si elle est libre. Si la ressource est déjà verrouillée et que la ressource est protégée alors, le thread suspend son exécution et passe dans un état où il attend que la ressource redevienne libre. Dans le cas où la ressource n'est pas protégée, on considère que l'accès par deux threads en même temps à une ressource est une erreur de conception, on va donc faire passer notre automate dans un état puits. Lors de la phase de vérification, si le système arrive dans cet état, le model checker considérera qu'il a trouvé un chemin menant à un interblocage et affichera la trace correspondante.

La garde de la transition est vraie si trois conditions sont remplies :

– le thread en cours d'exécution fait partie des threads pouvant accéder à la ressource passée en paramètre ,
– ce thread entre dans sa section critique ,
– et que ce thread n'a pas déjà verrouillé cette ressource.

Dans l'opération associée à cette transition, on formalise les trois cas présentés plus haut :

– la ressource n'est pas protégée, mais un autre thread y accède, on bloque l'exécution,
– si elle est protégée et verrouillée le thread en cours d'exécution passe dans un état où il attend la ressource,
– sinon il la verrouille et prend sa priorité si celle-ci est supérieure à la sienne.

Afin de rendre plus lisible la spécification, on a défini trois opérations, `protected_locked(sd)`, `unprotected_locked(sd)` et `block(mt)`. Les deux premières permettent de tester si la donnée passée en paramètre est verrouillée par un thread, la troisième fait passer le thread en paramètre dans l'état en attente de ressource.

Les expressions `block(mt)`, `protected_locked(sd)`, et `unprotected_locked(sd)` ont été introduites pour segmenter le code et le rendre plus lisible.

$$
\begin{array}{l}
\mathit{must_access}(d) \triangleq \\
\quad \land\ \mathit{computing_thread} \neq \mathit{bot_thread} \\
\quad \land\ d \in \mathit{SharedData} \\
\quad \land\ \mathit{computing_thread} \in \mathit{accessing_thread}[d] \\
\quad \land\ \mathit{access_time}[d,\ \mathit{computing_thread}][1] \\
\qquad\qquad = \mathit{execution_timer}[\mathit{computing_thread}] \\
\quad \land\ \mathit{AccessedBy}[d] \neq \mathit{computing_thread}
\end{array}
$$

$$
\begin{array}{l}
\mathit{access} \triangleq \\
\quad \text{LET } \mathit{sd} \triangleq \text{ CHOOSE } \mathit{sd} \in \mathit{SharedData} : \mathit{must_access}(\mathit{sd}) \text{IN} \\
\qquad \text{IF } \mathit{unprotected_locked}(\mathit{sd}) \text{ THEN} \\
\qquad \text{FALSE} \\
\qquad\ \text{ELSE IF } \mathit{protected_locked}(\mathit{sd}) \text{ THEN} \\
\qquad\qquad \land\ \mathit{block}(\mathit{computing_thread}) \\
\qquad\qquad \land\ \mathit{computing_thread}' = \mathit{bot_thread} \\
\qquad\ \text{ELSE } \land\ \mathit{lock_resource}(\mathit{sd}) \\
\qquad \land\ \mathit{actual_Priority}' = [t \in \mathit{Lifted_Thread} \mapsto \\
\qquad\qquad \text{IF} \quad t = \mathit{computing_thread} \\
\qquad\qquad\qquad \land\ \mathit{sd} \in \mathit{Protected} \\
\qquad\qquad\qquad \land\ \mathit{data_Priority}[\mathit{sd}] > \mathit{actual_Priority}[t] \\
\qquad\qquad \text{THEN } \mathit{data_Priority}[\mathit{sd}] \\
\qquad\qquad \text{ELSE } \mathit{actual_Priority}[t]]
\end{array}
$$

$$
\begin{aligned}
&block(mt) \triangleq \\
&\quad \land awaiting_resource' = awaiting_resource \cup \{mt\} \\
&\quad \land ready' = ready \setminus \{mt\}
\end{aligned}
$$

$$
\begin{aligned}
&protected_locked(sd) \triangleq \\
&\ sd \in Protected \land AccessedBy[sd] \neq bot_thread
\end{aligned}
$$

$$
\begin{aligned}
&unprotected_locked(sd) \triangleq \\
&\ sd \in Unprotected \land AccessedBy[sd] \neq bot_thread
\end{aligned}
$$

Libérer une ressource

On définit de manière très similaire à la transition précédente une transition pour la sortie de section critique. La garde indique simplement que la ressource est bien verrouillée par le thread en cours d'exécution et que celui-ci arrive à la fin de sa section critique.

Lorsque cette garde est vraie, on libère la ressource (appel à l'opération `release_resource`) et la priorité du thread redescend soit à la priorité du thread, soit à la plus haute priorité des ressources qu'il verrouille encore.

$$
\begin{aligned}
&must_release(d) \triangleq \\
&\quad \land d \in SharedData \\
&\quad \land computing_thread \neq bot_thread \\
&\quad \land internal_mode[computing_thread] \in accessing_thread[d] \\
&\quad \land access_time[d,\ computing_thread][2] \\
&\qquad\qquad = execution_timer[computing_thread] \\
&\quad \land AccessedBy[d] = computing_thread
\end{aligned}
$$

$$
\begin{aligned}
&accessed_data \triangleq \\
&\quad \{d \in SharedData : \\
&\quad AccessedBy[d] = computing_thread\}
\end{aligned}
$$

$$
\begin{aligned}
&max_prio_data \triangleq \\
&\ \textsc{choose}\ \ d \in accessed_data : \\
&\quad \forall\, od \in accessed_data : data_Priority[od] \leq data_Priority[d]
\end{aligned}
$$

$$
\begin{aligned}
&release \triangleq \\
&\quad \textsc{let}\ sd \triangleq \textsc{choose}\ sd \in SharedData : must_release(sd)\textsc{in} \\
&\qquad \land release_resource(sd) \\
&\qquad \land actual_Priority' = [t \in Lifted_Thread \mapsto \\
&\qquad\qquad \textsc{if}\ t = computing_thread\ \textsc{then} \\
&\qquad\qquad \textsc{if}\ accessed_data \neq \{\}\ \textsc{then} \\
&\qquad\qquad\quad data_Priority[max_prio_data] \\
&\qquad\qquad\ \textsc{else}\ \ Priority[t] \\
&\qquad\qquad\ \textsc{else}\ \ actual_Priority[t]]
\end{aligned}
$$

Sortir de l'état " en attente de ressource "

On ajoute enfin une dernière transition permettant aux threads en attente d'une ressource critique de retourner dans l'état `ready` dès que la ressource se libère.

La garde de cette opération est vraie si au moins un thread de l'ensemble `awaiting_resource` n'est plus bloqué par une resource partagée verrouillée. Dans ce cas, on calcule l'ensemble des threads à libérer et cet ensemble de threads sort de l'état `awaiting_resource` et revient dans l'état `ready`.

$$canUnblock \triangleq \{x \in awaiting_resource : \neg protected_locked(x)\} \neq \{\}$$

$$\begin{array}{l} unblock \triangleq \\ \quad \text{LET } free \triangleq \{x \in awaiting_resource : \neg protected_locked(x)\} \text{ IN} \\ \quad\quad \land\ awaiting_resource' = awaiting_resource \setminus free \\ \quad\quad \land\ ready' = ready \cup free \end{array}$$

5.2.5 Comportement interne des threads

On définit ici le comportement d'un thread comme une succession de pas d'exécution. On montre ensuite comment on peut utiliser l'annexe comportementale pour décrire le comportement du thread.

Segmentation de l'exécution

L'exécution d'un thread doit être segmentée en fonction de ses différents instants de communications. On découpe le comportement du thread en une succession de pas. Le premier pas commence au début de l'exécution et se termine lors de la première interaction du thread (émission ou réception de donnée ou d'événement sur un port, accès à une ressource partagée). Le pas suivant a lieu lors de l'interaction suivante. Le dernier pas est calculé à la complétion du thread. Un pas est atomique, il s'agit d'une simple relation entre les entrées et les sorties du thread. On a au final une série de relations qui représentent les différents pas successifs possibles du thread.

On ajoute au module `kernel` une transition permettant d'exécuter un pas. La garde de cette transition est simple, le thread en cours d'exécution doit avoir une interaction à effectuer ou doit avoir terminé son exécution, et le pas ne doit pas avoir déjà été exécuté. Afin de vérifier cette partie de la condition on introduit une nouvelle variable, `last_step` qui stocke la valeur de du compteur de durée d'exécution à chaque pas. L'exécution d'un pas est très simple, on demande au module `thread behavior` d'exécuter le prochain pas du thread en cours d'exécution. Et on utilise la fonction `last_step` afin de noter que l'exécution de ce pas à bien était effectuée.[4]

[4] D'un point de vu logique, la quantification existentielle sur l'ensemble des threads dans l'opération `can_make_step` ne se justifie pas. Mais TLC, nous oblige à adopter une telle construction.

$$
\begin{array}{l}
can_make_step \triangleq \\
\land \quad \lor\ must_send \\
\quad \lor\ must_receive \\
\quad \lor\ \exists\, d \in Data : must_access[d] \lor must_release[d] \\
\quad \lor\ hasComplete \\
\land\ last_step[computing_thread] \neq execution_timer[computing_thread]
\end{array}
$$

$$
\begin{array}{l}
make_step \triangleq \\
\quad \land\ \exists\, th \in Thread : th = computing_thread \\
\quad\quad \land\ thr(th)!atomic_step \\
\quad \land\ last_step' = [t \in Thread \mapsto \\
\quad\quad \text{IF } t = computing_thread \text{ THEN } execution_timer[computing_thread] \\
\quad\quad\quad \text{ELSE } last_step[t]]
\end{array}
$$

Définition du comportement

Dans cette partie, on décrit comment on peut générer les différents pas d'exécution d'un thread à partir de la description de son comportement exprimé grâce à l'annexe comportementale. On peut dans l'annexe comportementale directement accéder au contenu d'un port en entrée en utilisant l'opérateur `?`, on peut de la même manière utiliser un opérateur pour accéder au marqueur `fresh` associé à ce port. Dans notre modèle TLA, nous avons défini des fonctions permettant d'associer à un nom de port son contenu ou la valeur du marqueur de `fresh`. Pour un port data en entrée, cela consiste à utiliser les fonctions `data_input_buffer` et `data_input_fresh`. La référence au contenu d'un port `d1` se traduit par l'expression `data_input_buffer[d1]`. De la même manière lorsqu'on veut envoyer une donnée par un port data (`op !(data)`), on modifie les tampons associés au port `op` :

$$
\begin{array}{l}
\land\ data_output_buffer' = [x \in Out_data_port \mapsto \text{IF } x = op \text{ THEN } data \\
\quad \text{ELSE } data_output_buffer[x]] \\
\land\ data_output_fresh' = [x \in Out_data_port \mapsto \text{IF } x = op \text{ THEN TRUE} \\
\quad \text{ELSE } data_output_fresh[x]]
\end{array}
$$

Le comportement du thread est défini grâce à l'annexe comportementale par un automate. Afin de pouvoir synchroniser la mise à jour des données par le noyau d'exécution et leur utilisation par le thread, on va introduire un nouveau type d'états : les états de synchronisation. Toutes les transitions entre deux états de synchronisation forment un pas d'exécution, ils correspondent à une seule transition TLA. L'état `initial` et les états `complete` sont considérés comme des états de synchronisation. Une transition simple entre deux états de synchronisation de la forme :

```
st1 -[condition]-> st2 exec ; ;
```

se traduit en TLA par l'expression suivante :

$\wedge$ $internal_state[th] = st1$
$\wedge$ $condition$
$\wedge$ $exec$
$\wedge$ $internal_state' = [x \in Thread \mapsto \text{IF } x = th \text{ THEN } st2$
$\text{ELSE } internal_state[th]]$

La traduction d'une séquence de transition sans choix est tout aussi simple, on décrit une conjonction des conditions et une conjonction des différentes actions. L'état interne final est l'état final de la séquence de transitions.

Si un état a plusieurs transitions en sorties, on traduira chaque chemin en sortie de cet état par une alternative d'une disjonction. Chaque chemin de sortie est traduit en suivant ces règles de traductions. Ils permettent chacun de définir une expression TLA. Une fois que les différentes expressions ont été définies la transition globale est une disjonction de ces différentes expressions.

5.2.6 Les modes au niveau du thread

On reprend ici la notion de mode présenté dans 5.1.3 ; on montre comment on peut modéliser une telle notion en TLA. Un mode, pour un thread, est un état local de ce thread. L'automate de mode associé à un thread est une première abstraction du comportement du thread. En fonction du mode courant, un thread pourra utiliser seulement un sous ensemble de ses ports où accéder à une partie des ressources partagées auxquelles il a accès. Dans la partie concernant l'ordonnancement, on a défini le temps d'exécution comme une constante fixe pour toutes les exécutions du thread. Grâce aux modes, on peut légèrement affaiblir cette contrainte. Maintenant `Execution_time` ne sera plus une simple fonction qui associe à chaque thread son temps d'exécution, mais une fonction dont le domaine sera un ensemble de couples threads, modes. Un thread aura donc plusieurs temps d'exécutions possibles en fonction de ses modes. De nombreuses autres constantes présentées dans les parties précédentes vont être impactées par l'utilisation des modes. Par exemple la constante `input_time` qui définit l'instant des communications pour les ports en entrées, doit prendre en compte cette modification. Son domaine qui était simplement l'ensemble des ports en entrées devient un ensemble de couples mode, port.

Les différents modes et transitions de modes sont représentés par un automate. La constante `Internal_mode` définit pour chaque thread l'ensemble de ses modes. Si un thread n'a pas de modes, on lui associe un mode générique permettant de conserver des types et des opérations similaires dans les deux cas. La constante `ModeTrans` définit les différentes transitions de modes possibles. Elle associe à un couple événement, mode un autre mode. On ajoute une nouvelle variable au noyau, `internal_mode`, elle représente

le mode courant des threads du système. Une nouvelle constante définit le mode initial de chaque thread.

Une transition de mode peut avoir lieu à deux moments dans le cycle d'exécution d'un thread, à l'activation (`dispatch`) ou à la completion. Au moment du dispatch, la mise à jour d'un port event peut déclencher un changement de mode. Ce comportement modélise la sélection d'un chemin d'exécution en fonction d'un événement reçu. Par exemple, un thread apériodique peut avoir plusieurs comportements, en fonction du port qui va déclencher l'activation, le thread va sélectionner un chemin d'exécution particulier. Chaque chemin d'exécution n'a pas forcément le même temps d'exécution, n'utilise pas forcément les mêmes ports, etc...

L'ensemble des ports mis à jour lors de l'activation est différent pour un thread périodique et un thread apériodique. Les procédures de changement de mode sont donc très légèrement différentes pour ces deux types de threads. On a découpé cette procédure en trois parties :

- La première calcule simplement l'ensemble des ports `event` à mettre à jour.
- La seconde est une condition qui vaut vraie si un changement de mode est possible pour un thread considéré.
- Enfin dans la dernière, si un changement de mode est possible, on détermine le nouveau mode et on modifie la variable `internal_mode`.

$$\begin{aligned}
&\mathit{to_deliver_periodic}(th) \triangleq \\
&\quad \{x \in \mathit{In_event_port} : \mathit{Input_time}[\mathit{internal_mode}[th], x] = \text{“Dispatch”} \\
&\quad \land\ \mathit{iep_event_cpt}[x] > 0\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{to_deliver_aperiodic}(th, ep) \triangleq \\
&\quad \{x \in \mathit{ports_to_deliver}[ep] \cap \mathit{In_event_port} : \mathit{iep_event_cpt}[x] > 0\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{enable_mode_switch_periodic}(th) \triangleq \\
&\quad \exists\, p \in \mathit{to_deliver_periodic}(th) : \\
&\quad \mathit{ModeTrans}[p, \mathit{internal_mode}[th]] \in \mathit{Internal_mode}
\end{aligned}$$

$$\begin{aligned}
&\mathit{enable_mode_switch_aperiodic}(th, ep) \triangleq \\
&\quad \exists\, p \in \mathit{to_deliver_aperiodic}(th, ep) : \\
&\quad \mathit{ModeTrans}[p, \mathit{internal_mode}[th]] \in \mathit{Internal_mode}
\end{aligned}$$

$$\begin{aligned}
&\mathit{switch_mode_periodic}(th) \triangleq \\
&\quad \text{IF } \mathit{enable_mode_switch_periodic}(th) \text{ THEN} \\
&\qquad \exists\, p \in \mathit{to_deliver_periodic}(th) : \\
&\qquad\quad \land\ \mathit{ModeTrans}[p, \mathit{internal_mode}[th]] \in \mathit{Internal_mode} \\
&\qquad\quad \land\ \mathit{internal_mode}' = [x \in \mathit{Thread} \mapsto \\
&\qquad\qquad\qquad \text{IF } x = th \text{ THEN } \mathit{ModeTrans}[p, \mathit{internal_mode}[th]] \\
&\qquad\qquad\qquad \text{ELSE } \mathit{internal_mode}[x]] \\
&\quad \text{ELSE UNCHANGED } \mathit{internal_mode}
\end{aligned}$$

$$
\begin{array}{l}
\mathit{switch_mode_aperiodic}(th,\ ep) \triangleq \\
\quad \text{IF } \mathit{enable_mode_switch_aperiodic}(th,\ ep) \text{ THEN} \\
\qquad \exists\, p \in \mathit{to_deliver_aperiodic}(th,\ ep) : \\
\qquad\quad \wedge\ \mathit{ModeTrans}[p,\ \mathit{internal_mode}[th]] \in \mathit{Internal_mode} \\
\qquad\quad \wedge\ \mathit{internal_mode}' = [x \in \mathit{Thread} \mapsto \\
\qquad\qquad\qquad \text{IF } x = th \text{ THEN } \mathit{ModeTrans}[p,\ \mathit{internal_mode}[th]] \\
\qquad\qquad\qquad\ \text{ELSE } \mathit{internal_mode}[x]] \\
\quad \text{ELSE UNCHANGED } \mathit{internal_mode}
\end{array}
$$

Un changement de mode peut aussi avoir lieu lors de la complétion du thread. Pendant toute la durée de l'exécution, on mémorise tous les événements susceptibles de déclencher une transition de mode dans une variable appelée `triggering_events`. En plus des événements reçus par le thread pendant son exécution, cet ensemble peut contenir des événements internes du thread. Un événement interne n'est pas reçu par un port, il est directement généré par le comportement du thread. Au moment de la complétion, si cet ensemble n'est pas vide, on choisit un événement et on applique la transition correspondante.

Dans le cas d'un thread apériodique, ce mécanisme peut servir à définir l'ensemble des ports actifs pour le prochain dispatch, et donc l'ensemble des événements auxquels ce thread sera capable de réagir.

$$
\begin{array}{l}
\mathit{complete} \triangleq \\
\quad \wedge\ \mathit{awaiting_dispatch}' = \mathit{awaiting_dispatch} \cup \{\mathit{computing_thread}\} \\
\quad \wedge\ \mathit{computing_thread}' = \mathit{bot_thread} \\
\quad \wedge\ \mathit{ready}' = \mathit{ready} \setminus \{\mathit{computing_thread}\} \\
\quad \wedge\ \mathit{send_at_completion}(\mathit{computing_thread}) \\
\quad \wedge\ \text{IF } \mathit{triggering_events}[\mathit{computing_thread}] \neq \{\} \text{ THEN} \\
\quad \exists\, ev \in \mathit{triggering_events}[\mathit{computing_thread}] : \\
\qquad \mathit{internal_mode}' = [x \in \mathit{Thread} \mapsto \\
\qquad\quad \text{IF } x = \mathit{computing_thread} \text{ THEN } \mathit{ModeTrans}[ev,\ x] \\
\qquad\quad\ \text{ELSE } \mathit{internal_mode}[x]] \\
\quad \text{ELSE UNCHANGED } \mathit{internal_mode}
\end{array}
$$

5.3 Les modes systèmes

Dans la partie 5.1.3, on a présenté de manière informelle la notion de mode système. Dans cette partie, on se propose d'étudier ce mécanisme en détail. On commence par donner une spécification abstraite de la transition de mode en TLA. On introduit ensuite la notion de temps grâce à un modèle UPPAAL. Enfin, on intégrera le protocole de mode à la spécification TLA du modèle d'exécution présenté dans 5.2.

5.3.1 Une abstraction des modes systèmes

On commence par donner une spécification des modes systèmes et plus particulièrement le protocole de changement de modes de manière indépendante. Cette approche nous permet de nous focaliser sur les problèmes liés au changement de mode de manière indépendante aux problèmes d'ordonnancement et de communications.

Transition de mode atomique

On présente à ce niveau une simple transition de mode atomique. L'automate de changement de mode est défini par un ensemble de constantes : l'ensemble de mode `Mode`, son mode de départ `InitialMode`, et un ensemble de transition, la fonction `NextMode`. L'ensemble `Event` définit l'ensemble des événements susceptibles de provoquer un changement de mode. L'ensemble de couples `domNextMode` représente le domaine de la fonction `NextMode`. Enfin, la fonction `ModeThreads` associe à chaque mode l'ensemble de ses threads actifs.

On utilise seulement deux variables, la première définit le mode courant (`currentMode`), et la seconde représente l'ensemble des threads en cours d'exécution (currentThreads). Lors de l'initialisation, ces variables sont initialisées respectivement à la valeur `InitialMode` et à un sous ensemble des threads de ce mode initial.

Dans ce modèle, la transition de mode est atomique, sur la réception d'un événement déclenchant une transition, on passe dans le mode suivant. L'ensemble des threads en cours d'exécution est aussi modifié, ils doivent faire partie de l'ensemble des threads du nouveau mode.

La transition `ThreadTransition` est une abstraction de l'ordonnancement des threads. On considère que les threads du mode courant peuvent être dans deux états, prêt ou inactif. Les threads prêts correspondent au thread en cours d'exécution, en attente de donnée partagée ou en attente de la ressource processeur dans le modèle complet. Les threads du mode courant non présent dans cet ensemble sont en attente d'activation.

MODULE *atomicModes*

CONSTANTS
$Mode, InitialMode, Event, domNextMode, NextMode,$
$Thread, ModeThreads$

ASSUME
$\land\ InitialMode \in Mode$
$\land\ domNextMode \subseteq Mode \times Event$
$\land\ NextMode \in [domNextMode \rightarrow Mode]$
$\land\ ModeThreads \in [Mode \rightarrow \text{SUBSET}\ Thread]$

VARIABLES $currentMode, currentThreads$

$TypeInvariant \triangleq$
$\quad \land\ currentMode \in Mode$
$\quad \land\ currentThreads \in \text{SUBSET}\ Thread$

$Invariant \triangleq currentThreads \in \text{SUBSET}\ ModeThreads[currentMode]$

$Init \triangleq$
$\quad \land\ currentMode = InitialMode$
$\quad \land\ currentThreads \in \text{SUBSET}\ ModeThreads[currentMode]$

$ModeTransition(evt) \triangleq$
$\quad \land\ \langle currentMode, evt \rangle \in domNextMode$
$\quad \land\ currentMode' = NextMode[currentMode, evt]$
$\quad \land\ currentThreads' \in \text{SUBSET}\ ModeThreads[currentMode']$

$ThreadTransition \triangleq$
$\quad \land\ currentThreads' \in \text{SUBSET}\ ModeThreads[currentMode]$
$\quad \land\ \text{UNCHANGED}\ \langle currentMode \rangle$

Les threads critiques

Dans un mode on peut trouver un ensemble de threads que l'on ne doit pas interrompre à n'importe quel moment. On doit attendre que ces threads aient tous terminé leur exécution. On définit pour chaque mode un ensemble de threads critique grâce à la fonction `CriticalThreads`. À la réception d'un événement, on commence la transition de mode. Dans un premier temps, on mémorise juste l'événement qui a déclenché la transition. Puis lorsque tous les threads critiques ont terminé leur exécution, on procède au changement de mode proprement dit.

On ajoute une variable `currentEvent` afin de conserver le nom de l'événement qui a déclenché la transition. Le début de la transition de mode ne peut avoir lieu que si aucun événement n'a été reçu. À la fin de la transition, le système redevient sensible aux requêtes de changement de mode.

MODULE *Critical*

CONSTANTS
$\quad Mode, InitialMode, Event, Thread, NextMode, domNextMode,$
$\quad ModeThreads, Critical$

ASSUME
$\quad \land\ InitialMode \in Mode$
$\quad \land\ domNextMode \subseteq Mode \times Event$
$\quad \land\ NextMode \in [domNextMode \rightarrow Mode]$
$\quad \land\ ModeThreads \in [Mode \rightarrow \text{SUBSET}\ Thread]$
$\quad \land\ \text{"NoEvent"} \notin Event$
$\quad \land\ Critical \in [Mode \rightarrow \text{SUBSET}\ Thread]$
$\quad \land\ \forall\, m \in Mode : Critical[m] \subseteq ModeThreads[m]$

$AllEvent \triangleq Event \cup \{\text{"NoEvent"}\}$

VARIABLES $currentMode$, $currentThreads$, $currentEvent$

$$
\begin{array}{l}
TypeInvariant \triangleq \\
\quad \land\ currentMode \in Mode \\
\quad \land\ currentThreads \in \text{SUBSET}\ Thread \\
\quad \land\ currentEvent \in AllEvent
\end{array}
$$

$Invariant \triangleq currentThreads \in \text{SUBSET}\ ModeThreads[currentMode]$

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ currentMode = InitialMode \\
\quad \land\ currentThreads \in \text{SUBSET}\ ModeThreads[currentMode] \\
\quad \land\ currentEvent = \text{"NoEvent"}
\end{array}
$$

$$
\begin{array}{l}
StartModeTransition(evt) \triangleq \\
\quad \land\ \langle currentMode, evt \rangle \in domNextMode \\
\quad \land\ currentEvent = \text{"NoEvent"} \\
\quad \land\ currentEvent' = evt \\
\quad \land\ \text{UNCHANGED}\ \langle currentMode, currentThreads \rangle
\end{array}
$$

$$
\begin{array}{l}
EndModeTransition \triangleq \\
\quad \land\ currentEvent \neq \text{"NoEvent"} \\
\quad \land\ currentEvent' = \text{"NoEvent"} \\
\quad \land\ currentMode' = NextMode[currentMode, currentEvent] \\
\quad \land\ currentThreads' \in \text{SUBSET}\ ModeThreads[currentMode'] \\
\quad \land\ currentThreads \cap Critical[currentMode] \subseteq currentThreads'
\end{array}
$$

$$
\begin{array}{l}
ThreadTransition \triangleq \\
\quad \land\ currentThreads' \in \text{SUBSET}\ ModeThreads[currentMode] \\
\quad \land\ \text{UNCHANGED}\ \langle currentMode, currentEvent \rangle
\end{array}
$$

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor\ \exists\, e \in Event : StartModeTransition(e) \\
\quad \lor\ EndModeTransition \\
\quad \lor\ ThreadTransition
\end{array}
$$

Les threads zombies

Certaines activités ne peuvent être arrêtées instantanément. À titre d'exemple, une telle activité devra arrêter un équipement avant de disparaître. Pour cela, on introduit un nouveau type de threads, les zombies, ces threads ne sont pas critiques, mais ne doivent pas être arrêtés brutalement. On va donc leur permettre de terminer leur exécution dans le nouveau mode. Au moment de la transition de mode, on calcule l'ensemble des zombies actifs.

L'ensemble des threads actif dans le nouveau mode sera un sous ensemble des threads du nouveau mode plus l'ensemble des zombies actifs. Enfin, on considère que la transition de mode est finie lorsque tous les zombies ont été supprimés.

MODULE *zombies*

CONSTANTS
$Mode, InitialMode, Event, Thread, NextMode, domNextMode,$
$ModeThreads, Critical, Zombies$

ASSUME
$\land\ InitialMode \in Mode$
$\land\ domNextMode \subseteq Mode \times Event$
$\land\ NextMode \in [domNextMode \rightarrow Mode]$ a mode transition is deterministic
$\land\ ModeThreads \in [Mode \rightarrow \text{SUBSET } Thread]$
$\land\ \text{“NoEvent”} \notin Event$
$\land\ Critical \in [Mode \rightarrow \text{SUBSET } Thread]$
$\land\ Zombies \in [Mode \rightarrow \text{SUBSET } Thread]$
$\land\ \forall\, m \in Mode : Zombies[m] \cap ModeThreads[m] = \{\}$

$AllEvent \triangleq Event \cup \{\text{“NoEvent”}\}$

VARIABLES
$currentMode, currentThreads, currentEvent, zombies$

$TypeInvariant \triangleq$
$\land\ currentMode \in Mode$
$\land\ currentEvent \in AllEvent$
$\land\ currentThreads \in \text{SUBSET } ModeThreads[currentMode] \cup zombies$
$\land\ zombies \in \text{SUBSET } Zombies[currentMode]$

ASSUME
$\land\ currentEvent \neq \text{“NoEvent”} \leadsto currentThreads \cap Critical[currentMode] = \{\}$

$Init \triangleq$
$\land\ currentMode = InitialMode$
$\land\ currentThreads \in \text{SUBSET } ModeThreads[currentMode]$
$\land\ zombies = \{\}$
$\land\ currentEvent = \text{“NoEvent”}$

$StartModeTransition(evt) \triangleq$
$\land\ \langle currentMode, evt \rangle \in domNextMode$
$\land\ currentEvent = \text{“NoEvent”}$
$\land\ currentEvent' = evt$
$\land\ zombies = \{\}$
$\land\ \text{UNCHANGED } \langle zombies, currentMode, currentThreads \rangle$

$EndModeTransition \triangleq$

$\land$ *currentEvent* $\neq$ "NoEvent"
$\land$ *currentEvent'* $=$ "NoEvent"
$\land$ *currentMode'* $=$ *NextMode*[*currentMode*, *currentEvent*]
$\land$ *currentThreads* $\cap$ *Critical*[*currentMode*] $\in$ SUBSET *ModeThreads*[*currentMode'*]
$\land$ *zombies'* $=$ *currentThreads* $\cap$ *Zombies*[*currentMode'*]
$\land$ *currentThreads'* $\in$ SUBSET *ModeThreads*[*currentMode'*] $\cup$ *zombies'*

ThreadTransition $\triangleq$
$\land$ *currentThreads'* $\in$ SUBSET *ModeThreads*[*currentMode*]
$\land$ *zombies* $\neq \{\} \Rightarrow$ *zombies'* $\in$ SUBSET *zombies*
$\land$ *currentThreads'* $\in$ SUBSET *ModeThreads*[*currentMode*] $\cup$ *zombies'*
$\land$ UNCHANGED $\langle$*currentMode*, *currentEvent*$\rangle$

Préemptions et priorités

On veut rendre le protocole présenté plus flexible :

- les transitions de mode ne sont plus considérées comme atomiques. On introduit la possibilité d'interrompre un changement de mode durant la prise en compte d'une transition de mode ;
- on introduit un ordre sur les transitions de façon à pouvoir prendre en compte leur importance relative ;
- enfin le traitement associé à un changement de mode ne s'exprime plus uniquement en fonction du mode courant mais en fonction du mode courant et de la transition en cours.

Dans un premier temps, on va modifier le type de la fonction `Critical`. Dans la première version de cette fonction, l'ensemble des threads critiques ne dépend que du mode courant. Or il nous parait utile de définir cet ensemble en fonction de l'événement qui déclenche la transition. La taille de cet ensemble dimensionne directement le temps d'attente avant le changement effectif de mode. Plus cet ensemble est réduit, plus le changement de mode sera rapide. Or certains changements de modes peuvent être plus urgents que d'autres. Par exemple, une requête de changement de mode résultant d'une détection de panne matérielle devra être traitée plus rapidement qu'un changement de mode prévu dans l'exécution du système.

Enfin, on introduit des priorités sur les changements de mode. Ainsi lorsqu'une transition de mode a débuté, que le système attend la fin des threads critiques, si une requête de changement de mode plus prioritaire arrive la transition de mode est interrompue.

——— MODULE *advanced_modes* ———

EXTENDS *Naturals*
CONSTANTS
Mode, *InitialMode*, *Event*, *Thread*, *NextMode*, *domNextMode*,

$\quad ModeThreads, Critical, Zombies, Priority$

$AllEvent \triangleq Event \cup \{\text{"NoEvent"}\}$

ASSUME
$\quad \land InitialMode \in Mode$
$\quad \land domNextMode \subseteq Mode \times Event$
$\quad \land NextMode \in [domNextMode \rightarrow Mode]$ a mode transition is deterministic
$\quad \land ModeThreads \in [Mode \rightarrow \text{SUBSET } Thread]$
$\quad \land \text{"NoEvent"} \notin Event$
$\quad \land Critical \in [domNextMode \rightarrow \text{SUBSET } Thread]$
$\quad \land Zombies \in [Mode \rightarrow \text{SUBSET } Thread]$
$\quad \land \forall m \in Mode : Zombies[m] \cap ModeThreads[m] = \{\}$
$\quad \land Priority \in [AllEvent \rightarrow Nat]$
$\quad \land Priority[\text{"NoEvent"}] = 0$
$\quad \land \forall e \in Event : Priority[e] > 0$

VARIABLES
$\quad currentMode, currentThreads, currentEvent, zombies$

$TypeInvariant \triangleq$
$\quad \land currentMode \in Mode$
$\quad \land currentThreads \in \text{SUBSET } Thread$
$\quad \land zombies \in \text{SUBSET } Thread$
$\quad \land currentEvent \in AllEvent$

$Invariant \triangleq$
$\quad \land currentThreads \subseteq (ModeThreads[currentMode] \cup zombies)$
$\quad \land zombies \cap ModeThreads[currentMode] = \{\}$
$\quad \land zombies \subseteq Zombies[currentMode]$
$\quad \land currentEvent \neq \text{"NoEvent"} \Rightarrow zombies = \{\}$

$Init \triangleq$
$\quad \land currentMode = InitialMode$
$\quad \land currentThreads \in \text{SUBSET } ModeThreads[currentMode]$
$\quad \land zombies = \{\}$
$\quad \land currentEvent = \text{"NoEvent"}$

$StartModeTransition(evt) \triangleq$
$\quad \land \langle currentMode, evt \rangle \in domNextMode$
$\quad \land Priority[evt] > Priority[currentEvent]$
$\quad \land currentEvent' = evt$
$\quad \land zombies' = \{\}$
$\quad \land currentThreads' = currentThreads \setminus zombies$
$\quad \land \text{UNCHANGED } \langle currentMode \rangle$

$ModeTransition \triangleq$

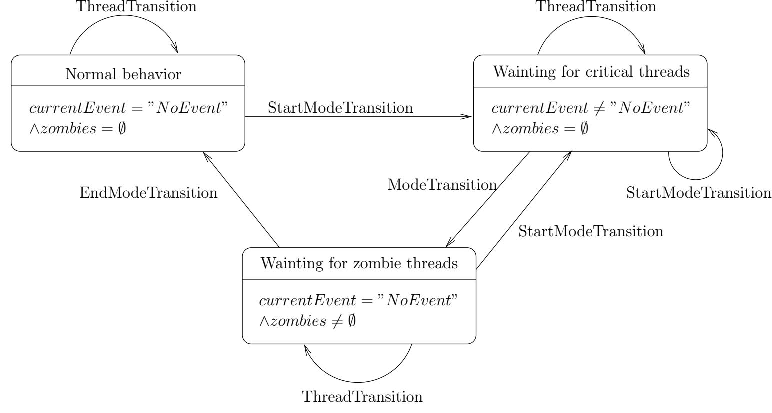

Fig. 5.6 – Automate de gestion des modes

$\land\ currentEvent \neq$ "NoEvent"
$\land\ currentEvent' =$ "NoEvent"
$\land\ currentMode' = NextMode[currentMode, currentEvent]$
$\land\ currentThreads \cap Critical[currentMode, currentEvent]$
$\quad \in \text{SUBSET } ModeThreads[currentMode']$
$\land\ zombies' = currentThreads \cap Zombies[currentMode']$
$\land\ currentThreads' \in \text{SUBSET } (ModeThreads[currentMode'] \cup zombies')$
$\land\ currentThreads \cap Critical[currentMode, currentEvent] \subseteq currentThreads'$

$EndModeTransition \triangleq$
$\land\ zombies = \{\}$
$\land\ \text{UNCHANGED } \langle currentMode, currentThreads, zombies, currentEvent\rangle$

$ThreadTransition \triangleq$
$\land\ currentThreads' \in \text{SUBSET } ModeThreads[currentMode]$
$\land\ \text{IF } zombies \neq \{\} \text{ THEN}$
$\land\ zombies' \in \text{SUBSET } zombies$
$\land\ currentThreads' \in \text{SUBSET } (ModeThreads[currentMode] \cup zombies')$
ELSE
$\quad \land\ currentThreads' \in \text{SUBSET } ModeThreads[currentMode]$
$\quad \land\ \text{UNCHANGED } \langle zombies\rangle$
$\land\ \text{UNCHANGED } \langle currentMode, currentEvent\rangle$

La figure 5.6 représente cette spécification TLA sous la forme d'un automate. Sur cette figure on ne représente que le nom des transitions TLA.

5.3.2 Les modes en UPPAAL

Dans la partie précédente, on a présenté la procédure de changement de mode de manière tout à fait indépendante de l'ordonnancement. On va

se concentrer sur la dimension temporelle du protocole. On désire ici se rapprocher de la nature continue de l'écoulement du temps. Pour cela nous considérons les horloges du formalisme des automates temporisés. On a donc choisi d'utiliser UPPAAL [LPY95] pour cette partie de l'étude. Ce choix nous a aussi apporté des restrictions, et cette spécification peut être moins précise que la précédente sur certains points.

Comportement simple d'un thread

On ne considère ici que des threads périodiques sans préemption. On décrit le comportement d'un thread générique, un système est ensuite composé de plusieurs instances de ce thread. Un thread est caractérisé par sa période, sa priorité et son temps d'exécution. Un thread peut être dans différents états :
- `Init` : l'état de départ,
- `awaiting_dispatch` : en attente d'activation,
- `ready_state` : prêt à être exécuté,
- `waiting_proc` : en attente de la libération du processeur,
- `running_state` : en cours d'exécution.

On associe à chaque thread deux horloges, une pour compter sa période et la seconde son temps d'exécution. Enfin, on utilise deux variables globales `ready` et `running` qui permettent à chaque thread de connaître l'état des autres threads. On utilise aussi deux canaux de communications broadcast `take_proc` et `release_proc`. Le premier sert à signifier aux threads qu'un thread vient de prendre la ressource processeur, et le second qu'il vient de la libérer.

Un thread commence son exécution par initialiser son horloge de période à la valeur de sa période afin d'être activé immédiatement. Les variables `ready` et `running` sont initialisées à faux.

Lorsqu'un thread l'horloge de période d'un thread devient égale à sa période, le thread passe soit dans l'état `ready_state` si le processeur est libre soit dans l'état `waiting_proc`.

L'état `ready_state` est urgent (section : 2.2.3), lorsqu'un thread est dans cet état, le temps ne peut pas s'écouler. Le thread doit sortir de cet état afin que le système continue sa progression. Si le thread est le thread prêt de plus haute priorité que le processeur est libre, et qu'il n'y a pas de threads prêt à être activé, il peut passer dans l'état `running_state`. À ce moment, son horloge d'exécution est remise à zéro et ses variables d'état sont modifiées afin de rendre compte de son changement d'état. À ce moment, tous les autres threads présents dans l'état `ready_state` reçoivent le message `take_proc`, ils passent alors dans l'état `waiting_proc`.

Le thread reste dans l'état `running_state` tant qu'il n'a pas fini son exécution (dans ce modèle, on considère un ordonnancement non préemptif et sans ressources partagées). Lorsqu'il quitte cet état, il modifie sa variable

running et envoie le message release_proc. Lorsque les threads en attente du processeur reçoivent ce message, ils passent dans l'état prêt et un nouveau thread est choisi pour commencer son exécution. La figure 5.7 montre l'automate Uppaal décrit ici.

Le changement de mode dans UPPAAL

Afin de faciliter l'expression de l'automate de mode en UPPAAL, on restreint l'expressivité des automates de modes AADL : un événement ne pourra déclencher qu'une seule transition, quelque soit le mode courant. Ceci nous permet de représenter la relation de transition de mode comme une fonction ne dépendant que de l'événement reçu et non d'un couple événement, mode courant. L'automate de mode en UPPAAL est défini par son nombre de modes (nb_modes), un nombre d'événements (nb_event) et un tableau (Next_Mode). Ce tableau définit les transitions possibles de l'automate, il est indexé par le nombre d'événements et associe à chaque événement le mode suivant de la transition.

La configuration de chaque mode est définie par trois tableaux de booléen à deux dimensions, ModeThread, Critical, et Zombie. Chaque ligne représente un mode et chaque colonne un thread.

On modifie l'automate présenté dans la section précédente afin d'intégrer le protocole de changement de mode. La principale modification est l'ajout d'un état awaiting_mode. Pour des raisons de synchronisation, on a dû séparer la transition d'activation du thread en deux transitions séparées par un état committed, lorsqu'un thread entre dans un tel état, une de ses transitions de sortie doit être tirée immédiatement. La transition entre les états awaiting_dispatch et ready_state reste donc atomique : le thread ne peux pas rester dans l'état intermédiaire. On ajoute au modèle une variable partagée awaiting qui permet à chaque thread de connaître l'état des autres threads. Enfin deux canaux start et abort sont ajoutés.

La transition d'initialisation est séparée en deux transitions. Si le thread fait partie du mode initial, il commence son exécution dans l'état awaiting_dispatch sinon il passe dans l'état awaiting_mode. Lorsqu'un thread reçoit le message abort, quelque soit son état, s'il ne fait pas partie du mode suivant il doit être arrêté. Il passe donc dans l'état awaiting_mode. S'il est dans l'état ready_state, waiting_proc, ou running_state et qu'il fait partie des threads autorisés à se terminer dans le nouveau mode il reste dans son état. Les threads "zombies" sont éliminés lorsqu'ils terminent leur exécution. Un thread qui termine son exécution et qui ne fait pas partie du mode courant retourne dans l'état awaiting_mode.

Les threads appartenant au nouveau mode sont activés par la réception du message start. Ils passent dans l'état awaiting_dispatch et sont initialisés. Ce comportement est représenté par l'automate présenté par la figure 5.8.

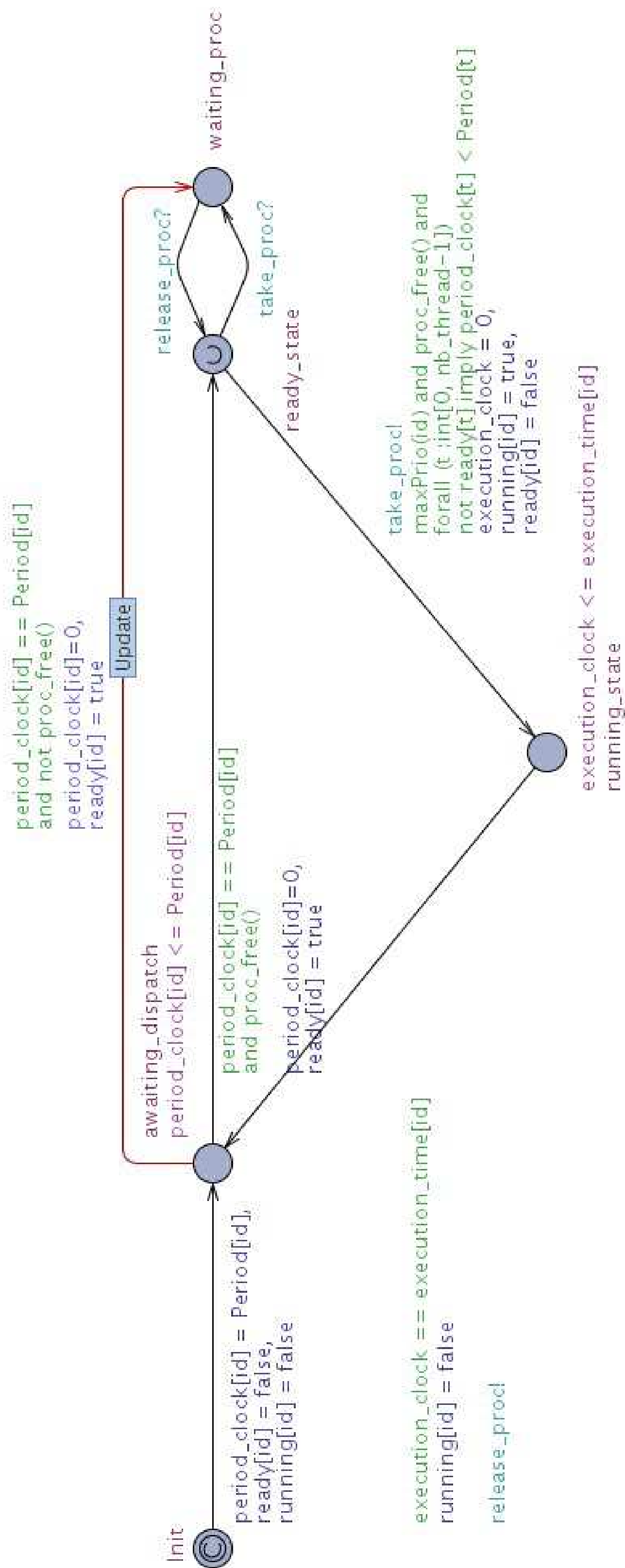

FIG. 5.7 – Ordonnancement de threads périodiques en UPPAAL

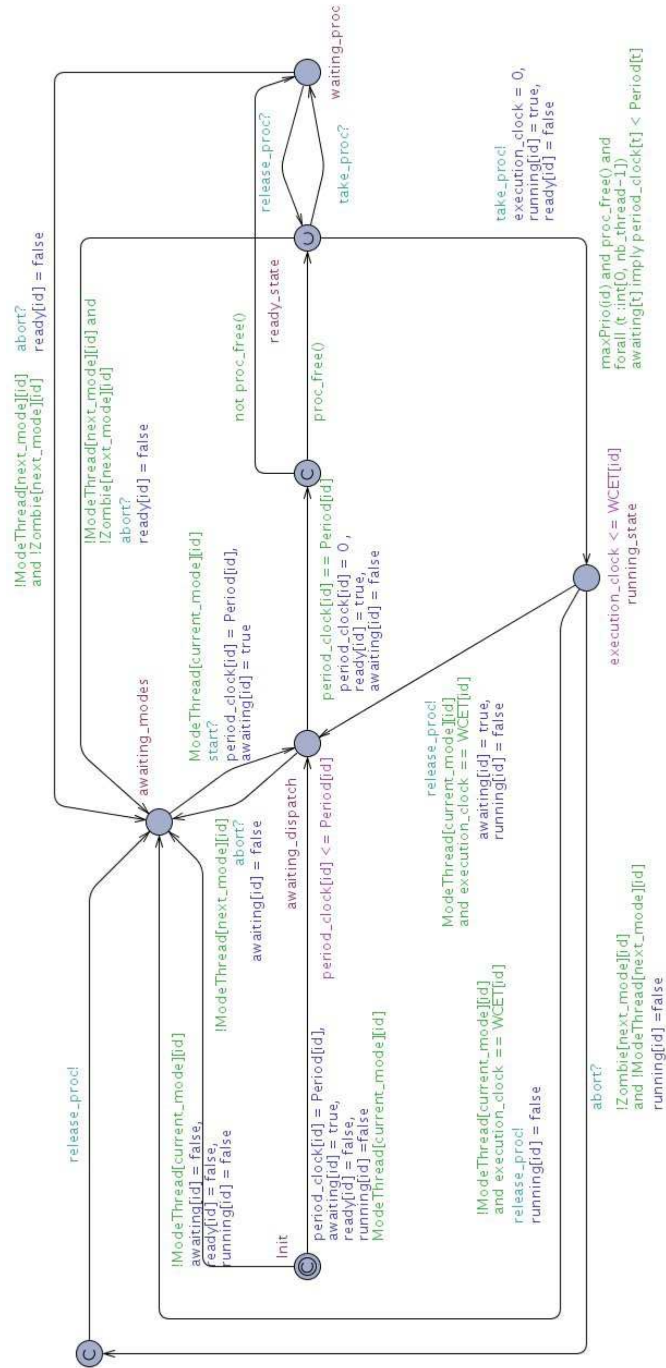

FIG. 5.8 – Ordonnancement de threads périodiques en UPPAAL avec gestion des modes

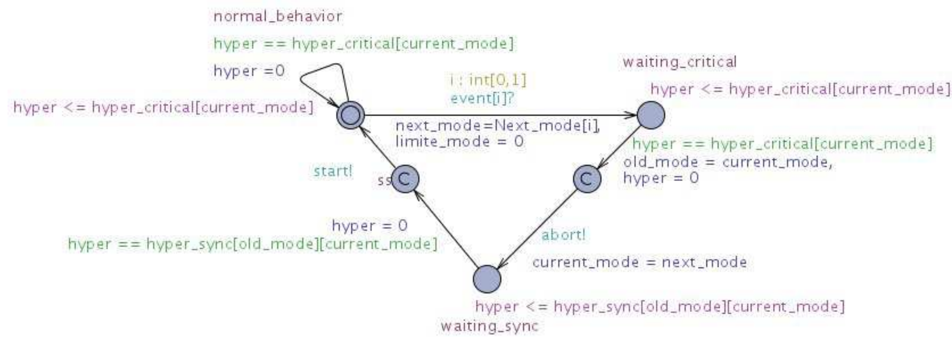

FIG. 5.9 – Automate de gestion des modes

Le changement de mode est contrôlé par un autre automate (figure 5.9), cet automate est composé de trois états principaux :

- `normal_behavior` : le système évolue normalement,
- `waiting_critical` : le système continue son exécution normale, on attend la synchronisation des threads critiques,
- `waiting_sync` : on attend que les threads communs aux deux modes soient synchronisés.

Toute la temporisation des transitions est effectuée grâce à une horloge `hyper`. Lorsque le système est dans l'état normal, cette horloge est remise à zéro à chaque hyperpériode du mode. On ajoute une constante `hyper_critical` qui à chaque mode fait correspondre son hyperpériode. Lorsque cet automate reçoit un événement de changement de mode, il passe dans l'état `waiting_critical`, en fonction de l'événement reçu on calcule le prochain mode. À l'hyperpériode du mode, la garde de la transition suivante devient vraie, on passe dans le nouveau mode, on envoie le signal `abort` afin d'arrêter les threads de l'ancien mode, en on mémorise l'ancien mode. Cette transition est séparée en deux parties car UPPAAL ne permet pas de faire un test sur une horloge et d'envoyer un message sur la même transition.

On attend dans l'état suivant que les threads communs aux deux modes soient synchronisés avant de démarrer les threads du nouveau mode. Cette hyperpériode de synchronisation dépend donc du nouveau et de l'ancien mode. Lorsque le système est synchronisé, le signal `start` est envoyé aux threads.

On a besoin de modéliser l'environnement en faisant des hypothèses sur les séquences d'arrivée des événements de changement de mode. Pour cela on peut utiliser toute l'expressivité des automates temporisés. La seule contrainte imposée par UPPAAL est de séparer les conditions sur les horloges et les synchronisations par canaux. Chaque transition doit donc être divisée en deux, séparée par un état `committed`(2.2.3). C'est cet état `committed` qui garantit que les deux transitions auront lieu immédiatement l'une après l'autre. La

première partie exprime la garde de la transition et la seconde l'envoi du message de changement de mode.

Afin de vérifier des propriétés dynamiques on peut utiliser le model checker de UPPAAL. Ces propriétés dans des cas simples peuvent être aisément exprimées en CTL. On peut exprimer et vérifier des propriétés telles que l'absence de d'interblocage, le respect des échéances, le fait qu'un seul thread puisse être en cours d'exécutions...

Dans des cas plus complexes, comme vérifier qu'au moment du changement de mode toutes les horloges de période des threads actifs sont à zéro, on à recourt à un automate observateur. Un tel automate suit le comportement du système, et si un comportement non valide est détecté, il passera dans un état puits. On doit ensuite vérifier qu'un tel état puits ne sera jamais atteint.

5.3.3 Integration des modes dans le système global

Dans cette partie, on modifie le modèle d'exécution présenté dans 5.2 afin de prendre en compte le mécanisme de changement de mode.

Définition de l'automate de mode

L'automate de mode AADL est défini par deux constantes `SystemMode`, l'ensemble des modes du système et `modeTransition` la fonction qui définit les transitions de l'automate. Cette fonction a pour domaine un ensemble de couples mode, événement, elle associe à chacun de ces couples le nom de la transition à laquelle elle correspond. Le domaine de la fonction est défini par la constante `SystemModeTransitionDomain`. Le nom des fonctions est défini par la constante `ModeTransition`. Enfin, la fonction `NewMode` associe à chaque transition le nouveau mode du système.

Les configurations de chaque mode sont définies par quatre fonctions :

- `ModeThread` : associe à chaque mode l'ensemble des threads actifs,
- `Synchronized` : définit l'ensemble des threads critiques pour chaque mode,
- `OneExec` : l'ensemble des threads zombies autorisés à terminer leur exécution courante,
- `AllExec` : l'ensemble des threads zombies autorisés à terminer le traitement de toutes les données contenues dans leurs ports.

Début de la transition

Une transition de mode peut débuter lorsqu'un événement est reçu par un port. Cette réception correspond toujours à l'envoi d'un message par un autre thread. Lors de chaque envoi de message, on va tester si ce message peut déclencher une transition de mode. Si c'est le cas et qu'aucune transition de mode n'est commencée, la transition de mode débute.

L'expression `triggering_Events(P)` calcule le sous ensemble de l'ensemble d'événements P pouvant déclencher une transition de mode. La condition `enable_mode_transition(P)` est vraie si un élément de P peut déclencher la transition et qu'aucune transition de mode n'a commencé. Enfin, l'opération `StartTransition(P)` correspond au début de la transition de mode. On se contente de mémoriser le nom de la transition en cours. Le système continuera à fonctionner normalement jusqu'à ce que les threads critiques soient synchronisés.

$$
\begin{aligned}
&\mathit{triggering_Events}(P) \triangleq \{ep \in P : \exists\, \mathit{transName} \in \mathit{ModeTransition} : \\
&\qquad \mathit{modeTransition}[ep,\ \mathit{currentMode}] = \mathit{transName}\} \\
&\mathit{enable_mode_transition}(P) \triangleq \\
&\quad \mathit{triggering_Events}(P) \neq \{\} \land \mathit{currentTransition} = \mathit{bot_trans} \\
&\mathit{StartTransition}(P) \triangleq \\
&\quad \exists\, ep \in \mathit{triggering_Events}(P) : \\
&\quad \mathit{currentTransition}' = \mathit{modeTransition}[ep,\ \mathit{currentMode}]
\end{aligned}
$$

Changement de mode

Le changement de mode ne peut avoir lieu après le début de la transition que lorsque tous les threads critiques sont synchronisés, i.e. à l'hyperpériode du système. Cette condition est explicitée par l'expression `enable_actual_mode_switch`. L'opération `activation_deactivation` décrit les changements d'états des threads. On calcule l'ensemble des threads à démarrer, l'ensemble des threads à arrêter et on applique ces changements. L'opération `actual_mode_switch` correspond au changement de mode effectif. La variable `currentMode` prend la valeur du nouveau mode, on effectue les activations et désactivations des threads, et enfin on modifie la topologie des connexions. La figure 5.10 montre l'automate complet du système.

$$
\begin{aligned}
&\mathit{enable_actual_mode_switch} \triangleq \\
&\quad \land\ \mathit{currentTransition} \neq \mathit{bot_trans} \\
&\quad \land\ \forall\, th \in \mathit{Synchronized}[\mathit{currentTransition}] : \\
&\qquad \mathit{period_timer}[th] = 0
\end{aligned}
$$

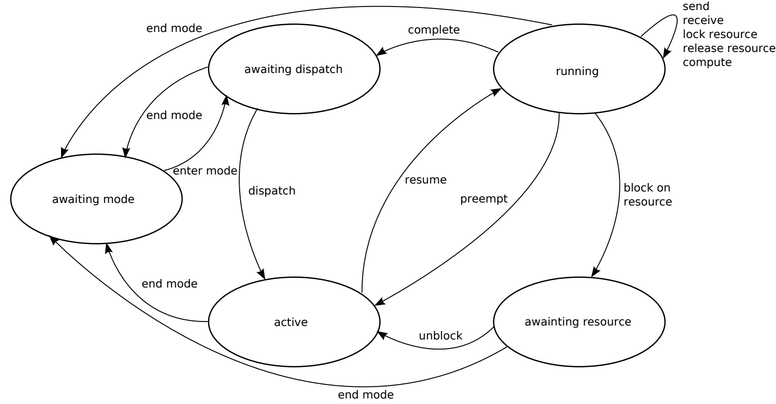

FIG. 5.10 – Automate complet du système TLA

$activation_deactivation \triangleq$
$\quad$ LET $thread_to_stop \triangleq$
$\quad\quad ModeThreads[currentMode] \setminus ModeThreads[currentMode']$IN
$\quad$ LET $thread_to_start \triangleq$
$\quad\quad ModeThreads[currentMode'] \setminus ModeThreads[currentMode]$IN
$\quad$ LET $thread_to_stop_now \triangleq$
$\quad thread_to_stop \setminus ((ready \cup awaiting_resource)$
$\quad\quad \cap (OneExec[currentTransition]$
$\quad\quad \cup AllExec[currentTransition]))$
IN
$\quad \wedge awaiting_dispatch' =$
$\quad (awaiting_dispatch \setminus thread_to_stop_now) \cup thread_to_start$
$\quad \wedge ready' = ready \setminus thread_to_stop_now$
$\quad \wedge awaiting_resource' = awaiting_resource \setminus thread_to_stop_now$
$\quad \wedge computing_thread' =$ IF $computing_thread \in thread_to_stop_now$ THEN
$\quad bot_thread$ ELSE $computing_thread$
$\quad \wedge awaiting_mode' = (awaiting_mode \cup thread_to_stop_now) \setminus thread_to_start$

$actual_mode_switch \triangleq$
$\quad \wedge currentMode' = NewMode[currentTransition]$
$\quad \wedge activation_deactivation$
$\quad \wedge ports!SetConnections(data_connection_topology[currentMode'],$
$\quad event_connection_topology[currentMode'],$
$\quad event_data_connection_topology[currentMode'])$

Éliminer les threads zombies

L'élimination de threads zombies se fait de manière très naturelle en modifiant l'opération de complétion. Lorsqu'un thread se termine, on va

juste tester son appartenance au mode courant. S'il n'en fait pas partie au lieu de retourner dans l'état attente d'activation, il passe dans l'état en attente de changement de mode.

5.4 Vérification et Prototype

Dans cette partie, on expose les possibilités de vérifications qu'apporte la formalisation du modèle d'exécution en TLA. On commence par présenter des outils d'analyse de modèles AADL existants. On présente ensuite les différentes vérifications que l'on propose. Enfin, on montre comment on peut intégrer ce type d'outils à l'atelier TOPCASED.

5.4.1 Vérification

Outils d'analyse de modèles AADL

On présente ici quatre outils permettant d'analyser des modèles AADL. On commence par Osate [OSA] (Open Source SAE AADL Toolset), il s'agit d'un plugin Eclipse développé par l'équipe du SEI en charge du développement d'AADL. Osate définit le métamodèle d'AADL en EMF (Eclipse Modeling Framework). Osate propose un éditeur textuel basé sur ce métamodèle. Il peut aussi présenter un modèle sous la forme d'un arbre XML et permet d'exporter les modèles dans ce format. Bien sûr, il permet de vérifier que la syntaxe AADL est bien respectée et que les modèles sont cohérents, c'est à dire que les règles du standard sont bien respectées. Osate propose aussi une API java permettant de manipuler les modèles AADL, et donc de définir ses propres extensions à l'outil. Enfin, il propose quelques analyses statiques simples, comme la charge d'un bus, ou d'un processeur, la latence des communications, ou encore la consommation électrique du système. Dans le cadre du projet Topcased, un éditeur graphique a été bâti au-dessus d'Osate.

ADeS [AG] est un simulateur d'exécution de modèles AADL écrit en java. Il se base sur Osate, et se présente comme un greffon Eclipse. ADeS a pour but d'implanter complètement le standard AADL. Le simulateur prend en compte une partie de l'annexe comportementale afin de spécifier le comportement des composants. Le simulateur est construit pour être extensible et ainsi prendre en compte les annexes AADL.

Cheddar [SLNM04] est un outil de simulation permettant de calculer différents critères de performance (contraintes temporelles, dimensionnement de ressources). L'outil permet, entre autres, de tester le respect des contraintes temporelles d'un jeu de tâches modélisant un système temps réel. Cheddar utilise Ocarina [HZPK08] pour l'analyse de performance de modèle AADL.

Ocarina est un compilateur AADL développé par Télécom-Paris-Tech. Cheddar peut aussi être employé comme un greffon de Topcased. Cheddar est principalement constitué de deux composants logiciels :

- Un éditeur qui permet à l'utilisateur de décrire l'application à analyser et sur lequel, les résultats de simulation seront présentés.
- Une bibliothèque comportant les principaux résultats de la théorie de l'ordonnancement temps réel ainsi que quelques outils basés sur la théorie des files d'attente.

Le Furness Toolset [Ass] est un logiciel d'analyse dédié à AADL. Il se présente comme un greffon Eclipse et est basé sur une approche à base d'algèbre de processus [BW90]. Il utilise ce modèle sémantique pour faire de l'analyse d'ordonnancement.

Vérifications proposées

Le langage TLA+ est accompagné d'un model checker, TLC[Lam02]. Cet outil, bien que moins performant que certains models checkers(vérificateurs de modèles) spécialisés, nous permet d'effectuer des vérifications de propriétés. Le model checking du modèle proposé permet de vérifier principalement trois types de propriétés :

- l'ordonnançabilité ;
- la taille des tampons ;
- et la protection des ressources partagées non protégées.

Le principal avantage d'utiliser un model checker afin de valider l'ordonnançabilité d'un système est que l'on peut prendre en compte de manière plus fine les dépendances entre les tâches. Par exemple, l'activation d'un thread apériodique dépend du comportement des threads qui peuvent le déclencher. De la même manière, on peut appréhender de manière plus rigoureuse l'impact des ressources partagées sur l'ordonnancement. Mais la possibilité de décrire le comportement d'un système de manière assez fine implique une explosion du nombre d'états générés lors de la vérification.

De la même manière, on peut prendre en compte le comportement des threads afin de vérifier que l'on ne dépasse jamais la capacité des tampons associées aux ports. Ces deux types de propriétés sont vérifiés par l'invariant de type. En effet, on a associé une horloge à l'échéance de chaque thread. Cette horloge est initialisée lors de l'activation et ne peut progresser que lorsque le thread est actif. À chaque `tick`, ce compteur est décrémenté et s'il passe en négatif, le thread ne peut satisfaire son échéance. Or ce compteur est un naturel, donc si on essaie de le rendre négatif son invariant de type sera violé.

En cas d'accès simultanés par deux threads à une ressource partagée non protégée, on fait passer le système dans un état d'erreur (puits). Si le

système arrive dans cet état, le model checker va détecter un deadlock, et nous signaler la violation de la propriété.

Les propriétés que l'on se propose de vérifier ici sont très simples. On présentera dans la partie perspectives différents types de propriétés plus complexes susceptibles d'être vérifiées par ce modèle.

5.4.2 Prototype

On a présenté dans les parties précédentes des éditeurs AADL permettant de construire des modèles et un outil de vérification, TLC, permettant de valider des propriétés de notre modèle. On se propose ici de montrer comment on peut faire le lien entre ces deux mondes. Notre prototype se base sur Topcased, Osate, Acceleo et TLC.

Topcased est un logiciel d'ingénierie assistée par ordinateur, il est basé sur le framework de la plateforme de développement Eclipse. Il s'appuie principalement sur des langages standardisés pour la modélisation du logiciel (UML, SysML, AADL...). Pour la gestion des modèles AADL il repose sur Osate. Il propose notamment un éditeur graphique de modèles AADL. Il se base aussi sur des outils de transformations de modèles ou de génération de code. Dans notre cas, nous avons utilisé le générateur de code Acceleo.

Acceleo [OBE] est un générateur de code qui se présente sous la forme d'un greffon Eclipse. Il peut être paramétré par le métamodèle du langage source de la traduction, dans notre cas AADL. Il accepte sans problèmes notables de traiter des modèles AADL une fois paramétré par le métamodèle défini dans Osate. On peut alors définir des modèles de traductions spécifiant le code à générer en fonction d'un modèle AADL. Une des particularités de ce générateur est de permettre la définition de service java que l'on peut ensuite utiliser dans les modèles de transformation. Or Osate propose de nombreuses méthodes permettant de parcourir efficacement un modèle AADL. On peut donc faire appel à ces méthodes dans le traducteur. On a notamment utilisé ce procédé pour retrouver les propriétés des éléments AADL. En effet, dans un modèle AADL une propriété peut être déclarée dans l'implémentation d'un composant, son type, éventuellement dans son super type ou même dans un autre composant. Osate propose des méthodes permettant de retrouver immédiatement une telle propriété quelle que soit sa position dans la hiérarchie des composants AADL.

Dans le prototype que l'on propose (figure 5.11), Topcased est utilisé pour éditer les modèles AADL. On utilise ensuite un modèle de traduction Acceleo pour générer des fichiers TLA. Ces fichiers TLA correspondent aux paramètres de notre modèle d'exécution. On peut alors utiliser TLC pour vérifier les propriétés présentées dans la partie précédente. Lorsque la vérification échoue, TLC donne une trace de l'exécution sous forme textuelle permettant d'arriver à l'état qui viole une des propriétés à vérifier. Le traducteur défini pendant cette thèse n'a pas évolué en même temps que le modèle

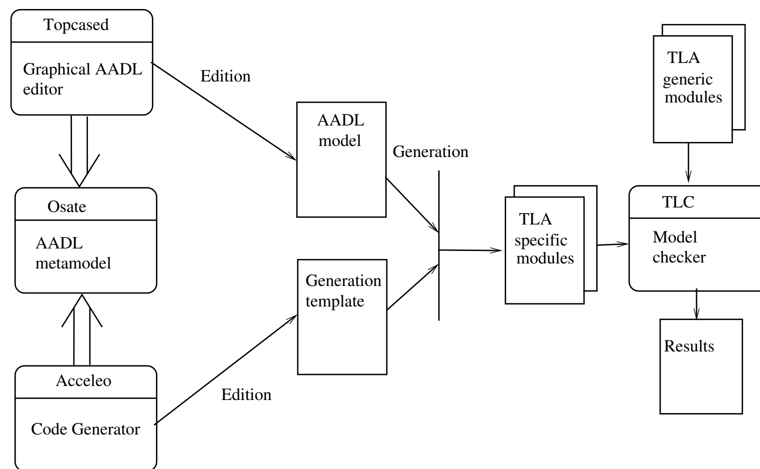

FIG. 5.11 – Architecture du prototype

d'exécution. Il permet de générer le code TLA adapté à une ancienne version de la spécification. Il est prévu de le mettre à jour.

5.5 Synthèse

Ce chapitre présente la contribution majeure de cette thèse. On a commencé par présenté notre mini-AADL. Ce langage est à la fois issu de AADL (V1 et V2), des besoins exprimés par ASTRIUM dans l'étude ArchiDyn et de l'annexe comportementale. On a ensuite proposé une formalisation en TLA du modèle d'exécution de notre mini-AADL. Cette formalisation a deux but :

– définir de manière non ambigüe la sémantique associée aux différents composants AADL ;
– et de valider des propriétés dynamiques sur des modèles AADL.

On a ensuite proposé un prototype mettant en jeu différents outils :

– Topcased et Osate pour l'édition des modèles,
– Acceleo pour la transformation modèle-source (AADL vers TLA),
– et TLC pour la vérification.

Dans le chapitre suivant, on reprend les modèles ArchiDyn et on montre comment on peut utiliser TLC pour vérifier des propriétés d'ordonnancement.

Chapitre 6

Les modèles ArchiDyn en mini-AADL

Afin de pouvoir utiliser TLC pour valider des propriétés sur les modèles définis dans l'étude ArchiDyn, on a besoin de les rendre compatibles avec le fragment d'AADL utilisé. En effet le comportement décrit dans le rapport ArchiDyn n'est pas toujours compatible avec la sémantique proposée par AADL. On a pris en compte les modèles d'analyses temporelles, car ils nous semblaient les plus intéressant à analyser. Dans un premier temps, on montre un modèle simple sans synchronisation. On présente ensuite un second modèle plus complexe prenant en compte ces synchronisations.

6.1 Modèle d'analyse d'ordonnancement simple

6.1.1 Définition du modèle étudié

On se base ici sur le modèle d'analyse temps réel sans suspensions défini dans la partie 4.2.2 (figure 6.1). Nous avons au total sept tâches :

- quatre tâches périodiques de période 125ms (`syst, pf, pl, aocs`) ;
- et trois tâches apériodiques (`aocs_man, acc, comp`) déclenchées par les tâches périodiques. Deux de ces tâches partagent une ressource protégée.

Les ports `event` entre les tâches périodiques et les tâches apériodiques sont utilisés pour déclencher ces dernières. Les connexions `event data` symbolisent la transmission des télécommandes du système vers les autres tâches périodiques. Elles n'ont pas d'influence sur le comportement du système. Ce modèle a principalement été utilisé pour tester la partie ordonnancement du modèle TLA. En jouant sur les valeurs de période, de temps d'exécution et d'échéance, on a volontairement introduit des erreurs afin de vérifier qu'elles étaient bien détectées par TLC. On montre dans la partie suivante comment on peut analyser un tel modèle.

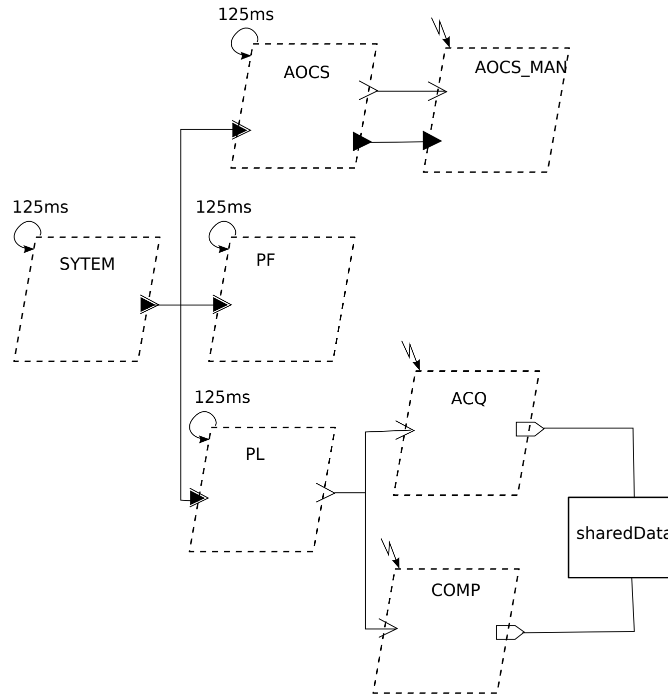

FIG. 6.1 – Architecture du modèle d'ordonnancement

6.1.2 Analyse du modèle

On se propose ici de montrer comment on peut utiliser notre modèle TLA et le model-checker TLC pour étudier l'ordonnançabilité du système. Les différentes tâches du modèle AADL présenté ont les propriétés temporelles suivantes :

	Syst	PL	PF	AOCS	AOCS _MAN	ACC	COMP
Type	Pério	Pério	Pério	Pério	Apér	Apér	Apér
Période	125	125	125	125			
Temps d'exécution	25	18	12	35	12	10	8
Priorité	205	195	185	55	35	40	41
Échéance	100	100	100	100	80	80	80

Le système présenté est ordonnançable. Normalement le model-checker ne donne pas d'informations lorsque les propriétés d'un système sont validées. Il indique juste que la vérification est réussie et le nombre d'états générés (310 dans notre cas). Afin de pouvoir observer le comportement du système, on a ajouté au code TLA des directives permettant de suivre

l'évolution de l'évaluation de la vérification. Ainsi, à chaque activation de tâche, début ou reprise d'exécution, verrouillage ou libération d'une ressource, et fin d'exécution, on demande à TLC de nous afficher une partie de l'état du système. L'opération permettant de demander à TLC d'afficher une chaine de caractère se nomme `PrintT`. On ajoute a l'opération `Dispatch` l'action suivante : `PrintT(■dispatch of the task■, th,■date■,now)`. Lors de chaque activation de tâche TLC affichera donc la ligne correspondante ; la variable `th` représente le thread activé, la variable `now` nous permet de suivre l'enchaînement temporel des actions. Lorsque une tâche prend la ressource processeur, on affiche le message `start of the task`, suivi du nom de la tâche et de la date. À la terminaison d'une tâche, on donne le nom de la tâche, la date ansi que le temps restant avant son échéance. Enfin, lorsqu'une resource est vérouillée ou relachée on affiche le nom de la ressource, le nom de la tâche qui y accède et la date.

```
<< "dispatch of the task", "aocs", "date", 0 >>
<< "dispatch of the task", "pf", "date", 0 >>
<< "dispatch of the task", "pl", "date", 0 >>
<< "dispatch of the task", "syst", "date", 0 >>
<< "start of the task", "syst", "date", 0 >>
<< "end of the task", "syst", "date", 25, "laxity", 75 >>
<< "start of the task", "pl", "date", 25 >>
<< "end of the task", "pl", "date", 43, "laxity", 57 >>
<< "dispatch of the task", "comp", "date", 43 >>
<< "dispatch of the task", "acc", "date", 43 >>
<< "start of the task", "pf", "date", 43 >>
<< "end of the task", "pf", "date", 55, "laxity", 45 >>
<< "start of the task", "aocs", "date", 55 >>
<< "end of the task", "aocs", "date", 90, "laxity", 10 >>
<< "dispatch of the task", "aocs_man", "date", 90 >>
<< "start of the task", "comp", "date", 90 >>
<< "lock resource", "sd", "comp", "date", 92 >>
<< "release resource", "sd", "comp", "date", 94 >>
<< "end of the task", "comp", "date", 100, "laxity", 23 >>
<< "start of the task", "acc", "date", 100 >>
<< "lock resource", "sd", "acc", "date", 102 >>
<< "release resource", "sd", "acc", "date", 104 >>
<< "end of the task", "acc", "date", 108, "laxity", 15 >>
<< "start of the task", "aocs_man", "date", 108 >>
<< "end of the task", "aocs_man", "date", 120, "laxity", 50 >>
<< "dispatch of the task", "aocs", "date", 125 >>
<< "dispatch of the task", "pf", "date", 125 >>
<< "dispatch of the task", "pl", "date", 125 >>
<< "dispatch of the task", "syst", "date", 125 >>
```

```
<< "start of the task", "syst", "date", 125 >>
<< "end of the task", "syst", "date", 150, "laxity", 75 >>
<< "start of the task", "pl", "date", 150 >>
<< "end of the task", "pl", "date", 168, "laxity", 57 >>
<< "dispatch of the task", "comp", "date", 168 >>
<< "dispatch of the task", "acc", "date", 168 >>
<< "start of the task", "pf", "date", 168 >>
<< "end of the task", "pf", "date", 180, "laxity", 45 >>
<< "start of the task", "aocs", "date", 180 >>
<< "end of the task", "aocs", "date", 215, "laxity", 10 >>
<< "dispatch of the task", "aocs_man", "date", 215 >>
<< "start of the task", "comp", "date", 215 >>
<< "lock resource", "sd", "comp", "date", 217 >>
<< "release resource", "sd", "comp", "date", 219 >>
<< "end of the task", "comp", "date", 225, "laxity", 23 >>
<< "start of the task", "acc", "date", 225 >>
<< "lock resource", "sd", "acc", "date", 227 >>
<< "release resource", "sd", "acc", "date", 229 >>
<< "end of the task", "acc", "date", 233, "laxity", 15 >>
<< "start of the task", "aocs_man", "date", 233 >>
<< "end of the task", "aocs_man", "date", 245, "laxity", 50 >>
<< "dispatch of the task", "aocs", "date", 250 >>
<< "dispatch of the task", "pf", "date", 250 >>
<< "dispatch of the task", "pl", "date", 250 >>
```

Ces informations nous permettent de définir la figure 6.2. Cette figure est créée manuellement mais on peut très bien imaginer une génération automatique de cette figure à partir de la trace TLC. Dans cette figure on voit que les tâches périodiques sont exécutées en premier. À la fin de la tâche `PL`, les tâches `ACC` et `COMP` sont activées. De même, à la fin de la tâche `AOCS`, on active la tâche `AOCS_MAN`. Ces tâches sont apériodiques, elles sont activées par une simple réception d'événement.

On peut rendre ce système non ordonnançable, il suffit par exemple d'augmenter la priorité des tâches `ACC` et `COMP` pour qu'elles soient exécutées avant la tâche `AOCS`. Dans ce cas, TLC produit une trace dont le dernier état est reproduit ici. TLC affiche l'ensemble des variables du système sous la forme d'une conjonction. On va par exemple retrouver les variables définissants l'état de chaque thread :

- $ready = \{"aocs"\}$,
- $computing_thread = aocs$,
- $awaiting_resource = \{\}$,
- $awaiting_dispatch = \{"pf", "pl", "syst", \ldots\}$.

On peut regarder le contenu des ports du système à cet instant, ainsi l'expression :

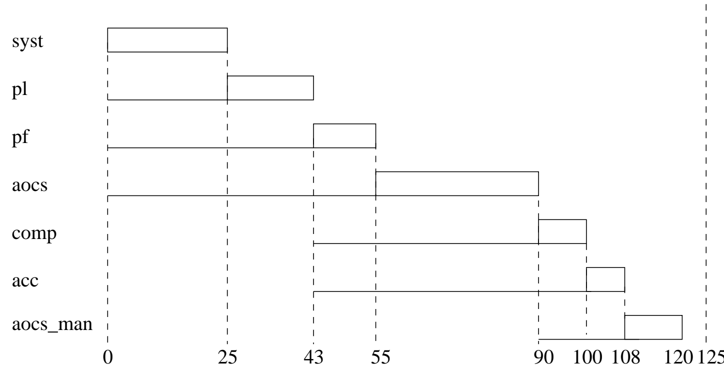

FIG. 6.2 – Ordonnancement du système généré par TLC.

$iep_delivered = [aocs_man_calc \mapsto 0, comp_comp \mapsto 1, acc_acc \mapsto 1]$
nous indique que le port `calc` ne contient aucun élément, et que les ports `comp` et `acc` en contiennent un. On voit sur la dernière ligne que l'horloge associée à l'échéance de la tâche `AOCS` est passée à -1. Ceci signifie dans notre modèle que cette tâche n'a pas pu respecter son échéance.

$STATE125 : < Actionline659, col3toline681, col69ofmoduleThreads >$
$\wedge\ currentMode =$ "mode1"
$\wedge\ last_output_port_time = [aocs \mapsto -1, pf \mapsto -1, pl \mapsto -1, syst \mapsto -1,$
$aocs_man \mapsto -1, comp \mapsto -1, acc \mapsto -1]$
$\wedge\ now = 101$
$\wedge\ out_event_buffer = [aocs_calc \mapsto \text{FALSE}, pl_acc \mapsto \text{TRUE}, pl_comp \mapsto \text{TRUE}]$
$\wedge\ event_data_connection = [syst_TC_port \mapsto \{$"aocs_TC_port",
"pf_TC_port", "pl_TC_port"$\}]$
$\wedge\ awaiting_mode = \{\}$
$\wedge\ ready = \{$"aocs"$\}$
$\wedge\ awaiting_dispatch = \{$"pf", "pl", "syst", "aocs_man", "comp", "acc"$\}$
$\wedge\ out_data_fresh = [aocs_dt \mapsto \text{FALSE}]$
$\wedge\ triggering_events = [aocs \mapsto \{\}, pf \mapsto \{\}, pl \mapsto \{\}, syst \mapsto \{\},$
$aocs_man \mapsto \{\}, comp \mapsto \{\}, acc \mapsto \{\}]$
$\wedge\ internal_mode = [aocs \mapsto$ "mode_aocs", $pf \mapsto$ "mode_pf", $pl \mapsto$ "mode_pl",
$syst \mapsto$ "mode_syst", $aocs_man \mapsto$ "mode_aocs_man",
$comp \mapsto$ "mode_comp", $acc \mapsto$ "mode_acc"$]$
$\wedge\ event_connection = [aocs_calc \mapsto \{$"aocs_man_calc"$\}, pl_acc \mapsto \{$"acc_acc"$\},$
$pl_comp \mapsto \{$"comp_comp"$\}]$
$\wedge\ internal_state = [aocs \mapsto 0, pf \mapsto 0, pl \mapsto 18, syst \mapsto 25,$
$aocs_man \mapsto 0, comp \mapsto 0, acc \mapsto 0]$
$\wedge\ last_access_resource = (\langle$"sd", "comp"$\rangle :> 4\ @@\ \langle$"sd", "acc"$\rangle :> 4)$
$\wedge\ iedp_delivered = [aocs_TC_port \mapsto \langle\ \rangle, pf_TC_port \mapsto \langle\ \rangle,$
$pl_TC_port \mapsto \langle\ \rangle]$
$\wedge\ computing_thread =$ "aocs"
$\wedge\ awaiting_resource = \{\}$

$\wedge$ $out_data_buffer = [aocs_dt \mapsto 11]$
$\wedge$ $out_event_fresh = [aocs_calc \mapsto \text{FALSE},\ pl_acc \mapsto \text{TRUE},\ pl_comp \mapsto \text{TRUE}]$
$\wedge$ $AccessedBy = [sd \mapsto \text{"bot_thread"}]$
$\wedge$ $out_event_data_buffer = [syst_TC_port \mapsto 7]$
$\wedge$ $iep_event_cpt = [aocs_man_calc \mapsto 0,\ comp_comp \mapsto 0,\ acc_acc \mapsto 0]$
$\wedge$ $last_input_port_time = [aocs \mapsto -1,\ pf \mapsto -1,\ pl \mapsto -1,\ syst \mapsto -1,$
$aocs_man \mapsto -1,\ comp \mapsto -1,\ acc \mapsto -1]$
$\wedge$ $iep_delivered = [aocs_man_calc \mapsto 0,\ comp_comp \mapsto 1,\ acc_acc \mapsto 1]$
$\wedge$ $idp_fresh = [aocs_man_dt \mapsto \text{FALSE}]$
$\wedge$ $idp_delivered = [aocs_man_dt \mapsto 11]$
$\wedge$ $out_event_data_fresh = [syst_TC_port \mapsto \text{FALSE}]$
$\wedge$ $iedp_fresh = [aocs_TC_port \mapsto \text{FALSE},\ pf_TC_port \mapsto \text{FALSE},$
$pl_TC_port \mapsto \text{FALSE}]$
$\wedge$ $idp_env = [aocs_man_dt \mapsto 11]$
$\wedge$ $actual_Priority = [bot_thread \mapsto 0,\ aocs \mapsto 55,\ pf \mapsto 185,\ pl \mapsto 195,$
$syst \mapsto 205,\ aocs_man \mapsto 0,\ comp \mapsto 60,\ acc \mapsto 61]$
$\wedge$ $currentTransition = \text{"bot_trans"}$
$\wedge$ $iep_fresh = [aocs_man_calc \mapsto \text{FALSE},\ comp_comp \mapsto \text{TRUE},\ acc_acc \mapsto \text{TRUE}]$
$\wedge$ $period_timer = [aocs \mapsto 24,\ pf \mapsto 24,\ pl \mapsto 24,\ syst \mapsto 24]$
$\wedge$ $data_connection = [aocs_man_dt \mapsto \text{"aocs_dt"}]$
$\wedge$ $iedp_event_data_queue = [aocs_TC_port \mapsto \langle 7 \rangle,\ pf_TC_port \mapsto \langle 7 \rangle,$
$pl_TC_port \mapsto \langle 7 \rangle]$
$\wedge$ $execution_timer = [aocs \mapsto 28,\ pf \mapsto 12,\ pl \mapsto 18,\ syst \mapsto 25,$
$aocs_man \mapsto 0,\ comp \mapsto 10,\ acc \mapsto 8]$
$\wedge$ $deadline_timer = [aocs \mapsto -1,\ pf \mapsto 0,\ pl \mapsto 0,\ syst \mapsto 0,$
$aocs_man \mapsto 0,\ comp \mapsto 22,\ acc \mapsto 22]$

La trace complète de l'exécution amenant à cet état est constitué par 125 états similaires à celui-ci. Il est toutefois possible de demander à TLC de n'afficher que les variable modifiées entre deux états. Une trace complète donne suffisamment d'informations pour déduire le comportement du modèle AADL. Dans l'optique de proposer un outil plus complet, on peut envisager d'utiliser ces informations pour présenter ce comportement graphiquement.

6.2 Introduction des synchronisations

La solution retenue dans Archidyn pour représenter les synchronisations entre les tâches et le bus n'est pas directement utilisable dans notre modèle. En effet, AADL ne permet pas de décrire la suspension d'une tâche périodique. De plus, on a choisi de se limiter aux protocoles d'activation périodique et apériodiques. On présente ici le comportement de la tâche AOCS tel que défini dans Archidyn, puis on propose une nouvelle modélisation de la tâche. La figure 6.3 représente le comportement de la tâche AOCS. Lorsqu'elle est activée, cette tâche commence par exécuter un traitement de requêtes pendant 2,5ms ; elle passe ensuite dans un état d'attente. Elle attend l'arrivée de deux messages différents (AVB : `Avionic Bus` et ICB :

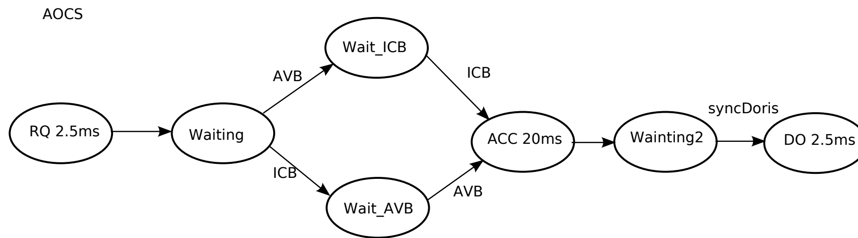

FIG. 6.3 – Comportement de la tâche AOCS

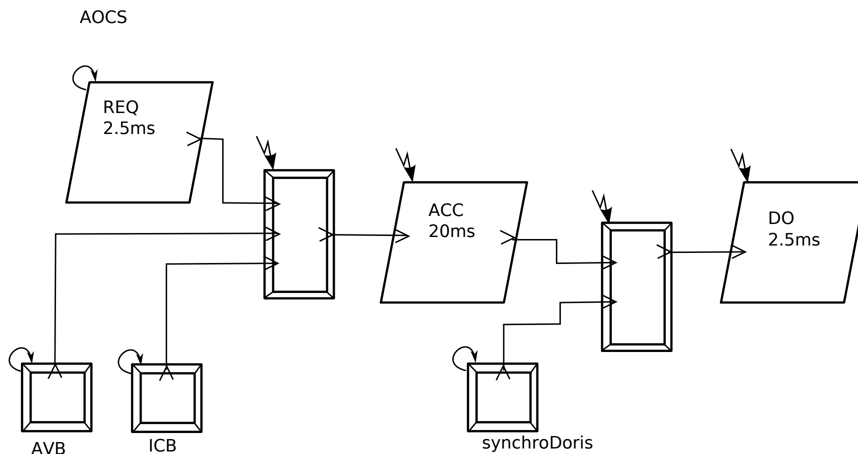

FIG. 6.4 – Nouvelle modélisation de la tâche AOCS

`Internal communication Bus`). Lorsque ces messages sont reçus, une seconde phase de calcul de 20ms commence. À la fin de ce calcul, elle se met de nouveau en attente d'un message de synchronisation (syncDoris), après l'avoir reçu, elle termine son exécution par une opération de 2,5ms.

Dans le modèle présenté ici, on a cherché à utiliser seulement les moyens à notre disposition pour représenter les synchronisations complexes de tâches. Ce travail nous a aussi amené à envisager la modélisation de l'environnement. On a utilisé des `device` afin de représenter des blocs de synchronisations de messages ainsi que des interfaces avec l'extérieur. Un `device` sera traduit dans notre modèle par un thread ayant un temps d'exécution nul et une priorité maximum. On profite ainsi de toutes les capacités du thread (comportement interne, activation périodique ou apériodique, etc..). Une interface avec l'extérieur est ainsi représentée par un `device` périodique produisant une donnée ou un événement régulièrement. Des `devices` apériodiques peuvent être utilisés pour décrire des synchronisations de messages, ils attendent une série de messages, lorsque tous les messages ont été reçus ils envoient un événement à un thread.

L'application AOCS est divisée en trois threads (figure 6.4), un par activité. Le premier est périodique de période 125ms, il exécute le traitement

des requêtes pendant 2,5ms. À la fin de son exécution, il envoie un signal vers un `device` de synchronisation. Cet élément attend aussi les deux messages AVB et ICB envoyés par des `devices` périodiques. Lorsque les trois messages sont arrivés, un premier thread apériodique est déclenché. Après une exécution de 20ms il envoie un message vers un second `device` de synchronisation. Celui-ci déclenchera la dernière partie de la tâche une fois qu'il aura reçu l'événement `syncDoris`.

Il est à noter que dans sa nouvelle version, AADL prend en compte ce type de tâche. Il définit un protocole d'activation mixte permettant à une tâche d'être déclenchée sur sa période ou sur la réception d'un événement.

6.3 Un exemple d'illustration

Dans cet exemple, on veut présenter plus particulièrement la notion de donnée partagée protégée et non protégée. On définit un modèle AADL composé simplement de deux tâches périodique. Ces deux tâches accèdent durant toute leur exécution à une donnée partagée protégée. Les caractéristique de ces tâches sont définies dans le tableau suivant :

	Période	Temps d'exécution	Priorité	Échéance
one	10	3	3	4
two	6	3	1	6

Afin de vérifier un tel système, TLC génère 93 états. TLC nous donne la trace d'exécution suivante, qui correspond à la figure 6.5 :

```
  << "dispatch of the task", "one", "date", 0 >>
<< "dispatch of the task", "two", "date", 0 >>
<< "start of the task", "one", "date", 0 >>
<< "lock resource", "sd", "one", "date", 0 >>
<< "release resource", "sd", "one", "date", 3 >>
<< "end of the task", "one", "date", 3, "laxity", 1 >>
<< "start of the task", "two", "date", 3 >>
<< "lock resource", "sd", "two", "date", 3 >>
<< "release resource", "sd", "two", "date", 5 >>
<< "end of the task", "two", "date", 6, "laxity", 0 >>
<< "dispatch of the task", "two", "date", 6 >>
<< "start of the task", "two", "date", 6 >>
<< "lock resource", "sd", "two", "date", 6 >>
<< "release resource", "sd", "two", "date", 8 >>
<< "end of the task", "two", "date", 9, "laxity", 3 >>
<< "dispatch of the task", "one", "date", 10 >>
<< "start of the task", "one", "date", 10 >>
<< "lock resource", "sd", "one", "date", 10 >>
```

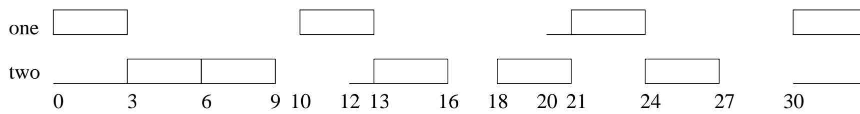

FIG. 6.5 – Ordonnancement de deux tâches se partageant une ressource

```
<< "dispatch of the task", "two", "date", 12 >>
<< "release resource", "sd", "one", "date", 13 >>
<< "end of the task", "one", "date", 13, "laxity", 1 >>
<< "start of the task", "two", "date", 13 >>
<< "lock resource", "sd", "two", "date", 13 >>
<< "release resource", "sd", "two", "date", 15 >>
<< "end of the task", "two", "date", 16, "laxity", 2 >>
<< "dispatch of the task", "two", "date", 18 >>
<< "start of the task", "two", "date", 18 >>
<< "lock resource", "sd", "two", "date", 18 >>
<< "release resource", "sd", "two", "date", 20 >>
<< "dispatch of the task", "one", "date", 20 >>
<< "start of the task", "one", "date", 20 >>
<< "lock resource", "sd", "one", "date", 20 >>
<< "release resource", "sd", "one", "date", 23 >>
<< "end of the task", "one", "date", 23, "laxity", 1 >>
<< "start of the task", "two", "date", 23 >>
<< "end of the task", "two", "date", 24, "laxity", 0 >>
<< "dispatch of the task", "two", "date", 24 >>
<< "start of the task", "two", "date", 24 >>
<< "lock resource", "sd", "two", "date", 24 >>
<< "release resource", "sd", "two", "date", 26 >>
<< "end of the task", "two", "date", 27, "laxity", 3 >>
<< "dispatch of the task", "one", "date", 30 >>
<< "dispatch of the task", "two", "date", 30 >>
<< "start of the task", "one", "date", 30 >>
<< "lock resource", "sd", "one", "date", 30 >>
<< "release resource", "sd", "one", "date", 33 >>
<< "end of the task", "one", "date", 33, "laxity", 1 >>
<< "start of the task", "two", "date", 33 >>
<< "lock resource", "sd", "two", "date", 33 >>
<< "release resource", "sd", "two", "date", 35 >>
<< "end of the task", "two", "date", 36, "laxity", 0 >>
<< "dispatch of the task", "two", "date", 36 >>
```

Dans la figure 6.5, on voit que la tâche `one` est activée à la date 20 alors que la tâche `two` est toujours active et a verrouillé la ressource. Si on modifie notre système et que l'on déclare que la ressource est non protégée,

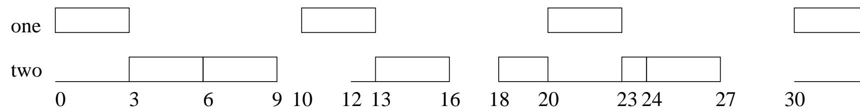

FIG. 6.6 – Ordonnancement de deux tâches se partageant une ressource non protégée

TLC va faire évoluer le système jusqu'à cet instant et va ensuite détecter un interblocage. Une possibilité pour corriger ce problème est de limiter la plage de temps pendant laquelle la tâche `two` accède à la ressource partagée. Il suffit en effet que cette tâche relâche la ressource à la date 20 pour que le système soit de nouveau ordonnançable. Notre modèle a alors le comportement décrit par la figure 6.6.

6.4 Synthèse

On a essayé de montrer ici comment on peut vérifier des propriétés d'ordonnancement par model-checking. De la même manière on peut vérifier que la tailles des files des ports n'est jamais dépassée. La version courante du modèle se contente d'afficher une alarme lorsque cela arrive et poursuit l'exécution. Mais on peut aussi considérer que c'est une erreur et bloquer l'exécution lorsque cela arrive. Les modèles présentés ici sont très simples et n'utilisent pas toutes les possibilités d'expression de notre mini-AADL. Le principal avantage, par rapport aux méthodes d'analyse d'ordonnançabilité classiques est qu'il permet de prendre en compte le comportement des threads.

Chapitre 7

Conclusion et Perspectives

Bilan

L'objectif de cette thèse était d'étudier la possibilité d'utiliser un ADL pour supporter le processus de développement d'un logiciel de vol satellite et d'intégrer dans ce processus l'utilisation de méthodes formelles. On a montré que l'utilisation d'AADL dans un tel cadre est possible. Dans ce bilan nous développons deux conclusions, une concernant le processus de développement, et l'autre la formalisation du modèle d'exécution AADL.

Processus de développement

Parmi les notions de base qui interviennent dans le processus de développement, nous pouvons citer la réutilisation, la composition, et le raffinement. On a vu que l'expression de patterns et leur héritage en AADL permettaient la réutilisation telle qu'on la trouve dans le domaine du spatial. Il est à noter que cette réutilisation sera encore plus facile grâce au concept de composant abstrait introduit dans AADL V2. La composition d'entités peut être gérée en AADL par différents moyens, au niveau statique, par la définition d'interfaces et au niveau dynamique par une liste de propriétés que l'on peut associer à ces interfaces. L'ensemble de ces propriétés témoigne d'une expérience métier dans le domaine du temps réel en général. Cependant, la notion de raffinement dans AADL aussi bien au niveau des composants que des types est traitée seulement de manière syntaxique. Nous reviendrons sur ce point dans la partie perspective.

Formalisation du modèle d'exécution AADL

On a proposé dans cette thèse une formalisation d'une partie du modèle d'exécution AADL. Cette formalisation a été construite pour être aisément extensible (au moins pour son auteur) et abordable. Elle permet ainsi de donner une vision du modèle d'exécution AADL précise et non sujette à

plusieurs interprétations. Cette formalisation peut ainsi servir de base pour le développement d'outils d'analyses et de vérification. Un des avantages majeurs d'utiliser un langage formel pour aborder un problème est qu'il oblige à être rigoureux.

Cette approche nous a permis de participer à l'évolution d'AADL. Ainsi, une collaboration avec Peter Feiler, un des principaux architectes d'AADL (avec Bruce Lewis et Steve Vestal), a pu s'instaurer après un séjour au SEI. Une partie des résultats de cette thèse tels que le timing des communications ou la gestion des modes systèmes ont été abordés lors de cette coopération.

La couverture des concepts AADL par cette formalisation a dans une certaine mesure démontré qu'il devenait pertinent d'utiliser des outils formels. L'utilisation effective d'une notation formelle pour décrire la sémantique d'un langage d'architecture nous semble aujourd'hui envisageable.

Enfin, on a proposé un prototype basé sur la formalisation proposée. Ce prototype permet la vérification de propriétés dynamiques sur un modèle AADL en réutilisant un vérificateur de modèle.

Ce travail a aussi permis à l'équipe d'acquérir une compétence en langages d'architectures en général, et plus particulièrement sur AADL. Cette expérience de spécification formelle des mécanismes AADL a effectivement été utilisée dans le cadre d'autres projets tels que Spacify [Spa], et Spices [Spb].

Perspectives

De nombreuses pistes de recherches peuvent être envisagées pour poursuivre ce travail.

Dans le cadre du processus de développement, on a présenté des raffinements entre différents niveaux d'abstractions. Si aujourd'hui des outils comme Topcased permettent de tracer les exigences entre différents niveaux, il est encore difficile de valider ces raffinements, i.e. de montrer qu'une propriété validée à un niveau abstrait reste établie par le raffinement. Au niveau fonctionnel, de telles preuves sont aujourd'hui prises en charge par des outils tels que l'atelier B. Cependant, pour ce qui est des propriétés non fonctionnelles, seules des approches automatiques semblent raisonnables ; procéder à la preuve manuelle de formules faisant intervenir le temps est en général très complexe.

Une seconde piste de travail serait d'étendre les différents types de propriétés vérifiables avec notre modèle. On peut vérifier que le traitement d'une requête ne dépasse pas un délai fixé. AADL utilise la notion de flot pour décrire ce type de propriétés. Un flot est le chemin que parcourt une donnée entre un port d'entrée et un port de sortie. Une propriété des flots est le temps maximum de parcours. On peut sur notre modèle utiliser des horloges associées aux flots afin de vérifier qu'ils ne dépassent pas le temps de

parcours maximum qui leur est associé.

On souhaite aussi vérifier des propriétés plus spécifiques. Par exemple, vérifier qu'un certain comportement apparaît (ou n'apparaît pas) dans l'ensemble des comportements possibles. Pour cela, il faut arriver à décrire ces comportements dans la logique temporelle définie par TLA. L'expression de ce type de propriétés dans un langage de plus haut niveau (à l'aide des MSC [IT96] par exemple) pourrait rendre plus accessible ce type de spécification. Il n'y aurait alors plus besoin de maîtriser le langage TLA pour exprimer les propriétés à vérifier.

Un dernier axe de poursuite serait de continuer à étendre le modèle TLA afin de couvrir une plus ample partie d'AADL. En effet, AADL comprend de nombreux mécanismes qui n'ont pas été jugés nécessaires dans cette étude, e.g., les notions relatives à la hiérarchie des composants (`process, thread group...`). Ces mécanismes sont néanmoins intéressants et il serait intéressant de les intégrer à notre modèle d'exécution.

De plus, dans notre cas, on considère que notre système ne comporte qu'un seul processeur. Or avec la généralisation de l'utilisation des processeurs multi-cœurs, il serait souhaitable de pouvoir prendre en compte ce type d'architecture.

Les modèles sont destinés à être finalement exécutés sur un système temps réel disposant de son propre noyau d'exécution. Il serait intéressant d'étudier la compatibilité entre ces noyaux et le modèle présenté ici, et le cas échéant de définir des raffinements permettant de garantir que les propriétés vérifiées au niveau modèle restent établies. Ceci nous amène à nous poser la question de la pertinence des abstractions choisies pour le modèle d'exécution et de manière plus générale à nous interroger sur la vision de AADL comme une bonne abstraction des mécanismes temps réel implantés par les différents noyaux d'exécutions (Rtems, VxWorks...).

Enfin, on a évoqué la possibilité d'étendre le modèle d'exécution présenté, mais au risque d'augmenter sa complexité. Afin de pouvoir raisonner plus facilement sur ce modèle d'exécution, il serait peut être plus facile d'essayer d'identifier un noyau primitif de mécanismes. Et ensuite, on pourrait définir les autres par traduction vers ce noyau.

Annexe A

Description des constantes du modèle TLA

Constante	Description
Thread	Ensemble des threads du modèle
bot_thread	Thread indéfini
Internal_mode	Ensemble des modes internes
mode_thread	Fonction qui associe chaque mode interne à son thread
thread_mode	Fonction inverse de mode_thread
mode_trans_domain	ensemble des couples mode interne, événement pouvant déclencher une transition
ModeTrans	Fonction de transition des modes internes
initial_internal_mode	Fonction qui définit pour chaque thread son mode initial
Internal_events	Liste des événements internes pouvant être générés par les threads
SystemMode	Ensemble des modes systèmes
InitialSystemMode	Mode initial
SystemModeTransitionDomain	ensemble des couples mode système, événement pouvant déclencher une transition de mode
ModeTransition	ensemble des noms des transitions de modes
bot_trans	transition indéfinie
modeTransition	fonction associant à chaque couple mode système, événement la transition correspondante
NewMode	Fonction donnant le nouveau mode à partir du nom de la transition
ModeThread	Fonction définissant pour chaque mode sa configuration de threads
Synchronized	Ensemble des threads synchronisés
OneExec	Ensemble des threads autorisés à terminer leur exécution courante pour chaque mode
AllExec	Ensemble des threads autorisés à terminer toutes leurs exécutions pour chaque mode
In_data_port	Ensemble des ports data en entrée
in_data_port	Fonction associant chaque port data à un thread
thread_in_data_port	Fonction inverse de la précédente
....	On a les même types de définition pour chaque type de port
protocol_event_data	Définition du protocole d'accès pour chaque port event data
protocol_event	Définition du protocole d'accès pour chaque port event
max_event	Nombre de message maximum pour chaque port event
max_event_data	Nombre de message maximum pour chaque port event data

Output_time	Liste des points de référence temporels possibles permettant de décrire la dynamique temporelle des ports en sortie
Input_time	Liste des points de référence temporels possibles permettant de décrire la dynamique temporelle des ports en entrée
Input_time_domain	ensemble de couple mode interne, port en entrée
Output_time_domain	ensemble de couple mode interne, port en sortie
input_time	On associe à chaque couple mode interne, port en entrée l'instant où le port doit être mis à jour
offsets_input_time	Liste des instant de communication au cours de l'exécution du thread pour chaque port en entrée
offsets_output_time	Liste des instant de communication au cours de l'exécution du thread pour chaque port en sortie
connection_topology_domain	Chaque mode et chaque transition de mode a une topologie de connexions propre.
data_connection_topology	À chaque mode ou transition de mode on associe une configuration des connexions. Une configuration des connexions est une fonction associant à chaque port en entrée un port en sortie.
event_connection_topology	À chaque mode ou transition de mode on associe une configuration des connexions. Une configuration des connexions est une fonction associant à chaque port en sortie un ensemble de ports en entrée.
Periodic	Ensemble des threads périodiques.
Aperiodic	Ensemble des threads apériodiques.
Period	Fonction définissant la période de chaque thread périodique
WCET	Pour chaque mode interne de thread, on donne son temps d'exécution
Priority	Fonction définissant la priorité de chaque thread, `bot_thread` à une priorité de zéro.
Deadline	Fonction définissant l'échéance de chaque thread.
dispatch_ports	Pour chaque mode interne de threads on définit l'ensemble des ports pouvant déclencher l'activation du thread.
ports_to_deliver	Pour chaque port déclenchant une activation on définit l'ensemble des ports à mettre à jour lors de cette activation.
SharedData	Ensemble des ressources partagées.
Protected	Ressources partagées protégées.
Unprotected	Ressources partagées non protégées.
accessing_thread	Fonction associant à chaque ressource partagée la liste des modes internes des threads pouvant y accéder.
acces_time_domain	Ensemble des couples ressource, thread pouvant accéder à cette ressource.
access_time	On associe à chaque couple ressource partagée mode interne de thread un couple de naturels. Ce couple définit l'intervalle de temps pendant lequel le thread accède à la ressource.
data_Priority	Priorité de chaque ressource partagée.

Annexe B

Code TLA

```
────────────────────── MODULE Kernel ──────────────────────
EXTENDS Sequences, FiniteSets, Naturals, TLC, aadl_model, Integers

CONSTANTS Data, bot_data, Access, bot_dport

VARIABLES   connections
  data_connection, event_connection, event_data_connection
connection_vars ≜ ⟨data_connection, event_connection,
event_data_connection⟩

VARIABLES   thread states
  computing_thread, ready, awaiting_dispatch, awaiting_resource,
  internal_mode, awaiting_mode
threads_state ≜ ⟨computing_thread, ready, awaiting_dispatch,
 awaiting_resource, internal_mode, awaiting_mode⟩

VARIABLES
  currentMode, currentTransition

VARIABLES
  internal_state

VARIABLES   timers
  deadline_timer, execution_timer, period_timer
timers ≜ ⟨deadline_timer, execution_timer, period_timer⟩

VARIABLES   now
  now

VARIABLES   Shared Data

  AccessedBy, last_access_resource, actual_Priority
shared_resources_vars ≜ ⟨AccessedBy, actual_Priority,
```

```
last_access_resource⟩

VARIABLES  ports
  idp_delivered, idp_fresh, idp_env,              data ports variables
  iep_delivered, iep_fresh, iep_event_cpt,              event ports variables
  iedp_delivered, iedp_fresh, iedp_event_data_queue,    event data ports variables
  last_input_port_time, last_output_port_time,
  out_data_buffer, out_event_buffer, out_event_data_buffer,
  out_data_fresh, out_event_fresh, out_event_data_fresh,
  triggering_events

input_buffers ≜ ⟨idp_env, iep_event_cpt, iedp_event_data_queue⟩

output_buffers ≜ ⟨out_data_fresh, out_event_fresh, out_event_data_fresh,
out_data_buffer, out_event_buffer, out_event_data_buffer,
last_output_port_time⟩

thread_input_vars ≜ ⟨idp_delivered, idp_fresh
 , iep_delivered, iep_fresh
 , iedp_delivered, iedp_fresh
 , last_input_port_time, triggering_events⟩

total_vars ≜ ⟨idp_delivered, idp_fresh, idp_env
, iep_delivered, iep_fresh, iep_event_cpt
, iedp_delivered, iedp_fresh, iedp_event_data_queue
, data_connection, event_connection, event_data_connection
, now, deadline_timer, execution_timer, period_timer
, computing_thread, ready, awaiting_dispatch, awaiting_resource
, AccessedBy, internal_state, actual_Priority, last_access_resource
, last_input_port_time, last_output_port_time
, out_data_buffer, out_event_buffer, out_event_data_buffer
, out_data_fresh, out_event_fresh, out_event_data_fresh
, internal_mode, triggering_events, currentMode, currentTransition
, awaiting_mode⟩


 View represent the set of variables that must be taken into account
 by the model checker. View is the set of all the system's variable
 except now, so if two state differ only by the value of now they will
 be considered by TLC as the same state.
view ≜ ⟨idp_delivered, idp_fresh, idp_env
, iep_delivered, iep_fresh, iep_event_cpt
, iedp_delivered, iedp_fresh, iedp_event_data_queue
, data_connection, event_connection, event_data_connection
, deadline_timer, execution_timer, period_timer
```

$, computing_thread, ready, awaiting_dispatch, awaiting_resource$
$, AccessedBy, internal_state, actual_Priority, last_access_resource$
$, last_input_port_time, last_output_port_time$
$, out_data_buffer, out_event_buffer, out_event_data_buffer$
$, out_data_fresh, out_event_fresh, out_event_data_fresh$
$, triggering_events, internal_mode, currentMode, currentTransition, awaiting_mode\rangle$

$ports \triangleq \text{INSTANCE } Ports$

$TypeInvariant \triangleq$
$\quad \land ports!TypeInvariant$
$\quad \land computing_thread \in Lifted_Thread$
$\quad \land ready \subseteq Thread$
$\quad \land awaiting_resource \subseteq Thread$
$\quad \land awaiting_dispatch \subseteq Thread$
$\quad \land deadline_timer \in [Thread \rightarrow Nat]$
$\quad \land execution_timer \in [Thread \rightarrow Nat]$
$\quad \land period_timer \in [Periodic \rightarrow Nat]$
$\quad \land now \in Nat$
$\quad \land AccessedBy \in [SharedData \rightarrow Lifted_Thread]$
$\quad \land last_input_port_time \in [Thread \rightarrow Int]$
$\quad \land last_output_port_time \in [Thread \rightarrow Int]$
$\quad \land last_access_resource \in [last_access_resource_domain \rightarrow Int]$
$\quad \land internal_state \in [Thread \rightarrow Nat]$
$\quad \land actual_Priority \in [Lifted_Thread \rightarrow Nat]$
$\quad \land triggering_events \in [Thread \rightarrow \text{SUBSET } (In_event_port \cup Internal_events)]$
$\quad \land internal_mode \in [Thread \rightarrow Internal_mode]$
$\quad \land currentMode \in SystemMode$
$\quad \land currentTransition \in Lifted_ModeTransition$
$\quad \land awaiting_mode \in \text{SUBSET } Thread$

$thr(this) \triangleq \text{INSTANCE } thread_behavior$

$Init \triangleq$
$\quad \land ports!Init$
$\quad \land computing_thread = bot_thread$
$\quad \land ready = \{\}$
$\quad \land awaiting_resource = \{\}$
$\quad \land awaiting_dispatch = ModeThreads[InitialSystemMode]$
$\quad \land deadline_timer = [x \in Thread \mapsto 0]$
$\quad \land execution_timer = [x \in Thread \mapsto 0]$
$\quad \land period_timer = [x \in Periodic \mapsto 0]$
$\quad \land now = 0$
$\quad \land AccessedBy = [x \in SharedData \mapsto bot_thread]$
$\quad \land last_input_port_time = [x \in Thread \mapsto -1]$

$\wedge$ *last_output_port_time* $= [x \in Thread \mapsto -1]$
$\wedge$ *internal_state* $= [x \in Thread \mapsto 0]$
$\wedge$ *last_access_resource* $= [x \in last_access_resource_domain \mapsto -1]$
$\wedge$ *actual_Priority* $= [x \in Lifted_Thread \mapsto 0]$
$\wedge$ *triggering_events* $= [x \in Thread \mapsto \{\}]$
$\wedge$ *internal_mode* $=$ *initial_internal_mode*
$\wedge$ *currentMode* $=$ *InitialSystemMode*
$\wedge$ *currentTransition* $=$ *bot_trans*
$\wedge$ *awaiting_mode* $= Thread \setminus ModeThreads[InitialSystemMode]$

——————-System modes——————————-

$triggering_Events(P) \triangleq \{ep \in P : \exists\, transName \in ModeTransition :$
$\quad modeTransition[ep, currentMode] = transName\}$

$enable_mode_transition(P) \triangleq$
$\quad triggering_Events(P) \neq \{\} \wedge currentTransition \neq bot_trans$

$StartTransition(P) \triangleq$
$\quad \exists\, ep \in triggering_Events(P) :$
$\quad currentTransition' = modeTransition[ep, currentMode]$

$enable_actual_mode_switch \triangleq$
$\quad \wedge\ currentTransition \neq bot_trans$
$\quad \wedge\ \forall\, th \in Synchronized[currentTransition] :$
$\quad\quad period_timer[th] = 0$

$send_at_mode_transition \triangleq$
LET $ODP \triangleq \{x \in Out_data_port :$
$\quad \exists\, idp \in In_data_port : data_connection_topology[currentTransition][idp] = x\}$ IN
LET $OEP \triangleq$ DOMAIN $(event_connection_topology[currentTransition])$ IN
LET $OEDP \triangleq$ DOMAIN $(event_data_connection_topology[currentTransition])$ IN
$\quad \wedge\ ports!odp!ForceStore(ODP, data_connection_topology[currentTransition])$
$\quad \wedge\ ports!oep!ForceRaiseEvent(OEP, data_connection_topology[currentTransition])$
$\quad \wedge\ ports!oedp!ForceRaiseEvent(OEDP, data_connection_topology[currentTransition])$

$activation_deactivation \triangleq$
LET $thread_to_stop \triangleq ModeThreads[currentMode] \setminus ModeThreads[currentMode']$ IN
LET $thread_to_start \triangleq ModeThreads[currentMode'] \setminus ModeThreads[currentMode]$ IN
LET $thread_to_stop_now \triangleq$
$\quad thread_to_stop \setminus ((ready \cup awaiting_resource) \cap$
$\quad (OneExec[currentTransition] \cup AllExec[currentTransition]))$
IN
$\quad \wedge\ awaiting_dispatch' = (awaiting_dispatch \setminus thread_to_stop_now) \cup thread_to_start$
$\quad \wedge\ ready' = ready \setminus thread_to_stop_now$
$\quad \wedge\ awaiting_resource' = awaiting_resource \setminus thread_to_stop_now$

$\land$ $computing_thread' = $ IF $computing_thread \in thread_to_stop_now$ THEN
bot_thread ELSE $computing_thread$
$\land$ $awaiting_mode' = (awaiting_mode \cup thread_to_stop_now) \setminus thread_to_start$

$actual_mode_switch \triangleq$
$\land$ $currentMode' = NewMode[currentTransition]$
$\land$ $send_at_mode_transition$
$\land$ $activation_deactivation$
$\land$ $ports!SetConnections(data_connection_topology[currentMode'],$
$event_connection_topology[currentMode'], event_data_connection_topology[currentMode'])$
$\land$ UNCHANGED $\langle internal_mode, currentTransition, internal_state, now \rangle$
$\land$ UNCHANGED $timers$
$\land$ UNCHANGED $shared_resources_vars$
$\land$ UNCHANGED $output_buffers$
$\land$ UNCHANGED $thread_input_vars$

——————deliver ports ——————————-

$set_port_dispatch_periodic(this) \triangleq$
LET $IDP \triangleq \{p \in thread_in_data_port[this]:$
$input_time[internal_mode[this], p] =$ "dispatch"$\}$ IN
LET $IEP \triangleq \{p \in thread_in_event_port[this]:$
$input_time[internal_mode[this], p] =$ "dispatch"$\}$ IN
LET $IEDP \triangleq \{p \in thread_in_event_data_port[this]:$
$input_time[internal_mode[this], p] =$ "dispatch"$\}$ IN
$\land$ $ports!idp!deliver(IDP)$
$\land$ $ports!iep!deliver(IEP)$
$\land$ $ports!iedp!deliver(IEDP)$

$set_port_dispatch_aperiodic(this, ep) \triangleq$
$\land$ $ports!idp!deliver(In_data_port \cap ports_to_deliver[ep])$
$\land$ $ports!iep!deliver(In_event_port \cap ports_to_deliver[ep])$
$\land$ $ports!iedp!deliver(In_event_data_port \cap ports_to_deliver[ep])$

$must_set_port \triangleq$
$\land$ $computing_thread \neq bot_thread$
$\land$ $\exists\, p \in thread_in_data_port[computing_thread]$
$\cup\, thread_in_event_port[computing_thread]$
$\cup\, thread_in_event_data_port[computing_thread]:$
$\land$ $input_time[internal_mode[computing_thread], p] =$ "execution"
$\land$ $execution_timer[computing_thread] \in offsets_input_time[internal_mode[computing_thread], p]$
$\land$ $last_input_port_time[computing_thread] < execution_timer[computing_thread]$

$set_port \triangleq$
LET $IDP \triangleq \{dp \in thread_in_data_port[computing_thread]:$

$\land$ $input_time[internal_mode[computing_thread], dp] =$ "execution"
$\land$ $execution_timer[computing_thread]$
$\in offsets_input_time[internal_mode[computing_thread], dp]$
$\land$ $last_input_port_time[computing_thread] < execution_timer[computing_thread]\}$ IN
LET $IEP \triangleq \{ep \in thread_in_event_port[computing_thread]:$
$\land$ $input_time[internal_mode[computing_thread], ep] =$ "execution"
$\land$ $execution_timer[computing_thread]$
$\in offsets_input_time[internal_mode[computing_thread], ep]$
$\land$ $last_input_port_time[computing_thread] < execution_timer[computing_thread]\}$ IN
LET $IEDP \triangleq \{edp \in thread_in_event_data_port[computing_thread]:$
$\land$ $input_time[internal_mode[computing_thread], edp] =$ "execution"
$\land$ $execution_timer[computing_thread]$
$\in offsets_input_time[internal_mode[computing_thread], edp]$
$\land$ $last_input_port_time[computing_thread] < execution_timer[computing_thread]\}$ IN
$\land$ $ports!idp!deliver(IDP)$
$\land$ $ports!iep!deliver(IEP)$
$\land$ $ports!iedp!deliver(IEDP)$
$\land$ $last_input_port_time' = [x \in Thread \mapsto$
IF $x = computing_thread$ THEN $execution_timer$
ELSE $last_input_port_time[x]]$
$\land$ $triggering_events' = [x \in Thread \mapsto$
IF $x = computing_thread$ THEN
$triggering_events[x] \cup$
$\{e \in IEP : \exists\, m \in Internal_mode : ModeTrans[e, internal_mode[x]] = m\}$
ELSE $triggering_events[x]]$
$\land$ UNCHANGED $connection_vars$
$\land$ UNCHANGED $timers$
$\land$ UNCHANGED $threads_state$
$\land$ UNCHANGED $shared_resources_vars$
$\land$ UNCHANGED $output_buffers$
$\land$ UNCHANGED $\langle now, internal_state, currentMode, currentTransition\rangle$

——————-Output ports——————————

$must_send_at_deadline(th) \triangleq$
$\land$ $deadline_timer[th] = 0$
$\land$ $\exists\, p \in thread_out_data_port[th]$
$\cup$ $thread_out_event_port[th]$
$\cup$ $thread_out_event_data_port[th]:$
$output_time[internal_mode[th], p] =$ "deadline"

$send_at_deadline \triangleq$
LET $this \triangleq$ CHOOSE $this \in awaiting_dispatch : must_send_at_deadline(this)$ IN
LET $ODP \triangleq \{dp \in thread_out_data_port[this]:$

$\land$ *output_time*[*internal_mode*[*this*], *dp*] = "deadline"}IN
LET *OEP* $\triangleq$ {*ep* $\in$ *thread_out_event_port*[*this*] :
$\land$ *output_time*[*internal_mode*[*this*], *ep*] = "deadline"}IN
LET *OEDP* $\triangleq$ {*edp* $\in$ *thread_out_event_data_port*[*this*] :
$\land$ *output_time*[*internal_mode*[*this*], *edp*] = "deadline"}IN
$\land$ *ports*!*odp*!*store*(*ODP*)
$\land$ *ports*!*oep*!*RaiseEvent*(*OEP*)
$\land$ *ports*!*oedp*!*RaiseEvent*(*OEDP*)
$\land$ IF *enable_mode_transition*(*OEP*) THEN
StartTransition(*OEP*)
ELSE UNCHANGED *currentTransition*
$\land$ UNCHANGED *thread_input_vars*
$\land$ UNCHANGED *connection_vars*
$\land$ UNCHANGED *timers*
$\land$ UNCHANGED *threads_state*
$\land$ UNCHANGED *shared_resources_vars*
$\land$ UNCHANGED $\langle$*now*, *internal_state*, *last_output_port_time*, *currentMode*$\rangle$

must_send $\triangleq$
$\land$ *computing_thread* $\neq$ *bot_thread*
$\land$ $\exists$ *p* $\in$ *thread_out_data_port*[*computing_thread*]
$\cup$ *thread_out_event_port*[*computing_thread*]
$\cup$ *thread_out_event_data_port*[*computing_thread*] :
$\land$ *output_time*[*internal_mode*[*computing_thread*], *p*] = "execution"
$\land$ *execution_timer*[*computing_thread*] $\in$ *offsets_output_time*[*internal_mode*[*computing_thread*], *p*]
$\land$ *last_output_port_time*[*computing_thread*] < *execution_timer*[*computing_thread*]

send_port $\triangleq$
LET *ODP* $\triangleq$ {*p* $\in$ *thread_out_data_port*[*computing_thread*] :
$\land$ *output_time*[*internal_mode*[*computing_thread*], *p*] = "execution"
$\land$ *execution_timer*[*computing_thread*] $\in$ *offsets_output_time*[*internal_mode*[*computing_thread*], *p*]
$\land$ *last_output_port_time*[*computing_thread*] < *execution_timer*[*computing_thread*]}IN
LET *OEP* $\triangleq$ {*p* $\in$ *thread_out_event_port*[*computing_thread*] :
$\land$ *output_time*[*internal_mode*[*computing_thread*], *p*] = "execution"
$\land$ *execution_timer*[*computing_thread*] $\in$ *offsets_output_time*[*internal_mode*[*computing_thread*], *p*]
$\land$ *last_output_port_time*[*computing_thread*] < *execution_timer*[*computing_thread*]}IN
LET *OEDP* $\triangleq$ {*p* $\in$ *thread_out_event_data_port*[*computing_thread*] :
$\land$ *output_time*[*internal_mode*[*computing_thread*], *p*] = "execution"
$\land$ *execution_timer*[*computing_thread*] $\in$ *offsets_output_time*[*internal_mode*[*computing_thread*], *p*]
$\land$ *last_output_port_time*[*computing_thread*] < *execution_timer*[*computing_thread*]}IN
$\land$ *ports*!*odp*!*store*(*ODP*)
$\land$ *ports*!*oep*!*RaiseEvent*(*OEP*)
$\land$ *ports*!*oedp*!*RaiseEvent*(*OEDP*)
$\land$ *last_output_port_time*$'$ = [*x* $\in$ *Thread* $\mapsto$

```
IF x = computing_thread THEN execution_timer
 ELSE last_output_port_time[x]]
 ∧ IF enable_mode_transition(OEP) THEN
StartTransition(OEP)
 ELSE UNCHANGED currentTransition
 ∧ UNCHANGED thread_input_vars
 ∧ UNCHANGED connection_vars
 ∧ UNCHANGED timers
 ∧ UNCHANGED threads_state
 ∧ UNCHANGED shared_resources_vars
 ∧ UNCHANGED ⟨now, internal_state, currentMode⟩

send_at_completion(th) ≜
  LET ODP ≜ {p ∈ thread_out_data_port[th] :
    ∧ output_time[internal_mode[th], p] = "completion"}IN
  LET OEP ≜ {p ∈ thread_out_event_port[th] :
    ∧ output_time[internal_mode[th], p] = "completion"}IN
  LET OEDP ≜ {p ∈ thread_out_event_data_port[th] :
    ∧ output_time[internal_mode[th], p] = "completion"}IN
  ∧ ports!odp!store(ODP)
  ∧ ports!oep!RaiseEvent(OEP)
  ∧ ports!oedp!RaiseEvent(OEDP)
  ∧ IF enable_mode_transition(OEP) THEN
  StartTransition(OEP)
   ELSE UNCHANGED currentTransition

------------------atomic step--------------------------
can_make_step ≜
∧ computing_thread ≠ bot_thread
∧ ∨ ∃ p ∈ thread_out_data_port[computing_thread]
    ∪ thread_out_event_port[computing_thread]
    ∪ thread_out_event_data_port[computing_thread] :
      ∧ output_time[internal_mode[computing_thread], p] = "execution"
      ∧ execution_timer[computing_thread] ∈
       offsets_output_time[internal_mode[computing_thread], p]
      ∧ last_output_port_time[computing_thread] < execution_timer[computing_thread]
      ∧ internal_state[computing_thread] ≠ execution_timer[computing_thread]
 ∨ ∧ execution_timer[computing_thread] = WCET[internal_mode[computing_thread]]
 ∧ internal_state[computing_thread] ≠ execution_timer[computing_thread]

make_step ≜
  ∧ ∃ th ∈ Thread : th = computing_thread
   ∧ thr(th)!atomic_step(execution_timer[th])
  ∧ internal_state′ = [t ∈ Thread ↦
```

$\quad$ IF $t = computing_thread$ THEN $execution_timer[computing_thread]$
$\quad$ ELSE $internal_state[t]]$
$\land$ UNCHANGED $thread_input_vars$
$\land$ UNCHANGED $input_buffers$
$\land$ UNCHANGED $connection_vars$
$\land$ UNCHANGED $timers$
$\land$ UNCHANGED $threads_state$
$\land$ UNCHANGED $shared_resources_vars$
$\land$ UNCHANGED $\langle now, last_output_port_time, currentMode, currentTransition \rangle$

——————-shared resource——————

$protected_locked(sd) \triangleq$
$sd \in Protected \land AccessedBy[sd] \neq bot_thread$

$unprotected_locked(sd) \triangleq$
$sd \in Unprotected \land AccessedBy[sd] \neq bot_thread$

$lock_resource(sd) \triangleq$
$\land AccessedBy' = [d \in SharedData \mapsto$
$\quad$ IF $d = sd \land internal_mode[computing_thread] \in accessing_thread[d]$
$\quad$ THEN $computing_thread$ ELSE $AccessedBy[d]]$
$\land PrintT(\langle$"lock resource", $sd, computing_thread,$ "date", $now\rangle)$

$release_resource(sd) \triangleq$
$\land AccessedBy' = [d \in SharedData \mapsto$
$\quad$ IF $d = sd \land internal_mode[computing_thread] \in accessing_thread[d]$
$\quad$ THEN bot_thread ELSE $AccessedBy[d]]$
$\land PrintT(\langle$"release resource", $sd, computing_thread,$ "date", $now\rangle)$

$block(mt) \triangleq$
$\land awaiting_resource' = awaiting_resource \cup \{mt\}$
$\land ready' = ready \setminus \{mt\}$
$\land PrintT(\langle$"task locked by a shared data", $mt\rangle)$

$must_access(d) \triangleq$
$\land computing_thread \neq bot_thread$
$\land d \in SharedData$
$\land internal_mode[computing_thread] \in accessing_thread[d]$
$\land access_time[d, internal_mode[computing_thread]][1]$
$\qquad = execution_timer[computing_thread]$
$\land last_access_resource[d, computing_thread] < execution_timer[computing_thread]$

$acccess \triangleq$
LET $sd \triangleq$ CHOOSE $sd \in SharedData : must_access(sd)$IN
$\quad$ IF $unprotected_locked(sd)$ THEN
$\quad$ FALSE $\land$ UNCHANGED $total_vars$

$\text{ELSE IF } protected_locked(sd) \text{ THEN}$
$\land\ block(computing_thread)$
$\land\ computing_thread' = bot_thread$
$\land$ UNCHANGED $thread_input_vars$
$\land$ UNCHANGED $input_buffers$
$\land$ UNCHANGED $connection_vars$
$\land$ UNCHANGED $timers$
$\land$ UNCHANGED $shared_resources_vars$
$\land$ UNCHANGED $output_buffers$
$\land$ UNCHANGED $\langle now, internal_state, awaiting_dispatch, internal_mode, currentMode, currentTransition, awaiting_mode\rangle$
ELSE $\land\ lock_resource(sd)$
$\land\ last_access_resource' = [x \in last_access_resource_domain \mapsto$
IF $x = \langle sd, computing_thread\rangle$ THEN $execution_timer[computing_thread]$
ELSE $last_access_resource[x]]$
$\land\ actual_Priority' = [t \in Lifted_Thread \mapsto$
IF $t = computing_thread \land sd \in Protected \land data_Priority[sd] > actual_Priority[t]$
THEN $data_Priority[sd]$
ELSE $actual_Priority[t]]$
$\land$ UNCHANGED $thread_input_vars$
$\land$ UNCHANGED $input_buffers$
$\land$ UNCHANGED $connection_vars$
$\land$ UNCHANGED $timers$
$\land$ UNCHANGED $threads_state$
$\land$ UNCHANGED $output_buffers$
$\land$ UNCHANGED $\langle now, internal_state, currentMode, currentTransition\rangle$

$must_release(d) \triangleq$
$\land\ d \in SharedData$
$\land\ computing_thread \neq bot_thread$
$\land\ internal_mode[computing_thread] \in accessing_thread[d]$
$\land\ access_time[d, internal_mode[computing_thread]][2]$
$= execution_timer[computing_thread]$
$\land\ last_access_resource[d, computing_thread] < execution_timer[computing_thread]$

$accessed_data(sd) \triangleq$
$\{d \in SharedData :$
$AccessedBy[d] = computing_thread \land d \neq sd\}$

$max_prio_data(sd) \triangleq$
CHOOSE $d \in accessed_data(sd) :$
$\forall\, od \in accessed_data(sd) : data_Priority[od] \leq data_Priority[d]$

$release \triangleq$
LET $sd \triangleq$ CHOOSE $sd \in SharedData : must_release(sd)$ IN

$\wedge$ $release_resource(sd)$
$\wedge$ $last_access_resource' = [x \in last_access_resource_domain \mapsto$
IF $x = \langle sd, computing_thread\rangle$ THEN
$execution_timer[computing_thread]$
ELSE $last_access_resource[x]]$
$\wedge$ $actual_Priority' = [t \in Lifted_Thread \mapsto$
IF $t = computing_thread \wedge sd \in Protected$ THEN
IF $accessed_data(sd) \neq \{\}$ THEN
$data_Priority[max_prio_data(sd)]$
ELSE $Priority[t]$
ELSE $actual_Priority[t]]$
$\wedge$ UNCHANGED $thread_input_vars$
$\wedge$ UNCHANGED $input_buffers$
$\wedge$ UNCHANGED $connection_vars$
$\wedge$ UNCHANGED $timers$
$\wedge$ UNCHANGED $threads_state$
$\wedge$ UNCHANGED $output_buffers$
$\wedge$ UNCHANGED $\langle now, internal_state, currentMode, currentTransition\rangle$

—————-Unblock thread——————————————-

$canUnblock \triangleq \{x \in awaiting_resource : \neg protected_locked(x)\} \neq \{\}$

$unblock \triangleq$
LET $free \triangleq \{x \in awaiting_resource : \neg protected_locked(x)\}$IN
$\wedge$ $awaiting_resource' = awaiting_resource \setminus free$
$\wedge$ $ready' = ready \cup free$
$\wedge$ $PrintT(\langle$"tasks unlocked :", $free\rangle)$
$\wedge$ UNCHANGED $thread_input_vars$
$\wedge$ UNCHANGED $input_buffers$
$\wedge$ UNCHANGED $connection_vars$
$\wedge$ UNCHANGED $timers$
$\wedge$ UNCHANGED $shared_resources_vars$
$\wedge$ UNCHANGED $output_buffers$
$\wedge$ UNCHANGED $\langle now, internal_state, currentMode, currentTransition,$
$awaiting_dispatch, awaiting_mode, computing_thread\rangle$

——————Dispatch———————————————

$can_dispatch(th) \triangleq$
$\vee$ $th \in Aperiodic \wedge \exists\, p \in In_event_port :$
$p \in dispatch_ports[internal_mode[th]] \wedge iep_event_cpt[p] > 0$
$\vee$ $th \in Periodic \wedge period_timer[th] = 0$

```
to_deliver_periodic(th) ≜
  {x ∈ thread_in_event_port[th] : input_time[internal_mode[th], x] = "Dispatch"
                   ∧ iep_event_cpt[x] > 0}

to_deliver_aperiodic(th, ep) ≜
    {x ∈  ports_to_deliver[ep] ∩ In_event_port : iep_event_cpt[x] > 0}

enable_mode_switch_periodic(th) ≜
    ∃ p ∈ to_deliver_periodic(th) :
    ⟨p, internal_mode[th]⟩ ∈ mode_trans_domain

enable_mode_switch_aperiodic(th, ep) ≜
    ∃ p ∈ to_deliver_aperiodic(th, ep) :
    ⟨p, internal_mode[th]⟩ ∈ mode_trans_domain

switch_mode_periodic(th) ≜
  IF enable_mode_switch_periodic(th) THEN
      ∃ p ∈ to_deliver_periodic(th) :
        ∧ ModeTrans[p, internal_mode[th]] ∈ Internal_mode
        ∧ internal_mode' = [x ∈ Thread ↦
              IF x = th THEN ModeTrans[p, internal_mode[th]]
               ELSE  internal_mode[x]]
   ELSE  UNCHANGED internal_mode

switch_mode_aperiodic(th, ep) ≜
  IF enable_mode_switch_aperiodic(th, ep) THEN
      ∃ p ∈ to_deliver_aperiodic(th, ep) :
        ∧ ModeTrans[p, internal_mode[th]] ∈ Internal_mode
        ∧ internal_mode' = [x ∈ Thread ↦
              IF x = th THEN ModeTrans[p, internal_mode[th]]
               ELSE  internal_mode[x]]
   ELSE  UNCHANGED internal_mode


dispatch ≜
LET th ≜ CHOOSE th ∈ awaiting_dispatch : can_dispatch(th)IN
  ∧ awaiting_dispatch' = awaiting_dispatch \ {th}
  ∧ ready' = ready ∪ {th}
  ∧ IF th ∈ Periodic THEN
     ∧ set_port_dispatch_periodic(th)
     ∧ switch_mode_periodic(th)
   ELSE ∃ ep ∈ dispatch_ports[internal_mode[th]] :
      ∧ iep_event_cpt[ep] > 0
      ∧ set_port_dispatch_aperiodic(th, ep)
      ∧ switch_mode_aperiodic(th, ep)
  ∧ triggering_events' = [x ∈ Thread ↦
```

$\text{IF } x = th \text{ THEN } \{\} \text{ ELSE } triggering_events[x]]$
$\land\ deadline_timer' = [x \in Thread \mapsto \text{IF } x = th \text{ THEN } Deadline[th] \text{ ELSE } deadline_timer[x]]$
$\land\ execution_timer' = [x \in Thread \mapsto \text{IF } x = th \text{ THEN } 0 \text{ ELSE } execution_timer[x]]$
$\land\ period_timer' = [x \in Periodic \mapsto \text{IF } x = th \text{ THEN } Period[th] \text{ ELSE } period_timer[x]]$
$\land\ last_input_port_time' = [x \in Thread \mapsto$
$\text{IF } x = th \text{ THEN } -1$
$\text{ELSE } last_input_port_time[x]]$
$\land\ last_output_port_time' = [x \in Thread \mapsto$
$\text{IF } x = th \text{ THEN } -1$
$\text{ELSE } last_output_port_time[x]]$
$\land\ internal_state' = [x \in Thread \mapsto$
$\text{IF } x = th \text{ THEN } 0$
$\text{ELSE } internal_state[x]]$
$\land\ last_access_resource' = [x \in last_access_resource_domain \mapsto$
$\text{IF } x[2] = th \text{ THEN } -1$
$\text{ELSE } last_access_resource[x]]$
$\land\ actual_Priority' = [x \in Lifted_Thread \mapsto$
$\text{IF } x = th \text{ THEN } Priority[x]$
$\text{ELSE } actual_Priority[x]]$
$\land\ \text{UNCHANGED } connection_vars$
$\land\ \text{UNCHANGED } output_buffers$
$\land\ \text{UNCHANGED } \langle now, computing_thread, awaiting_resource, AccessedBy,$
$currentMode, currentTransition, awaiting_mode\rangle$
$\land\ PrintT(\langle \text{"dispatch of the task"}, th, \text{"date"}, now\rangle)$

complete

$hasComplete \triangleq$
$\land\ computing_thread \neq bot_thread$
$\land\ execution_timer[computing_thread] = WCET[internal_mode[computing_thread]]$

$empty_buffers(th) \triangleq$
$\forall\, ep \in thread_in_event_port[th]:$
$iep_event_cpt[ep] = 0$

$complete \triangleq$
$\land\ awaiting_dispatch' = awaiting_dispatch \cup \{computing_thread\}$
$\land\ computing_thread' = bot_thread$
$\land\ ready' = ready \setminus \{computing_thread\}$
$\land\ send_at_completion(computing_thread)$
$\land\ \text{IF } triggering_events[computing_thread] \neq \{\} \text{ THEN}$
$\exists\, ev \in triggering_events[computing_thread]:$
$internal_mode' = [x \in Thread \mapsto$

```
      IF x = computing_thread THEN ModeTrans[ev, x]
       ELSE  internal_mode[x]]
  ELSE  UNCHANGED internal_mode
  ∧ IF  currentTransition  ≠ bot_trans
   ∧ computing_thread ∉ ModeThreads[currentMode]
   ∧ (computing_thread ∈ OneExec
   ∨ (computing_thread ∈ AllExec ∧ empty_buffers(computing_thread)))
  THEN  awaiting_mode' = awaiting_mode ∪ {computing_thread}
  ∧ UNCHANGED  awaiting_dispatch
  ELSE  awaiting_dispatch' = awaiting_dispatch ∪ {computing_thread}
      ∧ UNCHANGED  awaiting_mode
  ∧ IF  currentTransition ≠ bot_trans
    ∧ ∀ th ∈ (ready ∪ awaiting_resource ∪ awaiting_dispatch) :
    th ∈ ModeThreads[currentMode] THEN
    currentTransition' = bot_trans
   ELSE  UNCHANGED  currentTransition
  ∧ UNCHANGED  thread_input_vars
  ∧ UNCHANGED  connection_vars
  ∧ UNCHANGED  timers
  ∧ UNCHANGED  shared_resources_vars
  ∧ UNCHANGED ⟨now, internal_state, awaiting_resource,
  last_output_port_time, currentMode⟩
```

———————————Resume———————————-

```
maxPrio(th) ≜
  ∀ t ∈ ready : actual_Priority[t] ≤ actual_Priority[th]

canResume ≜
  ∧ ¬maxPrio(computing_thread)

resume ≜
 LET mt ≜ CHOOSE mt ∈ ready : maxPrio(mt)IN
     ∧ computing_thread' = mt
     ∧ UNCHANGED  thread_input_vars
     ∧ UNCHANGED  input_buffers
     ∧ UNCHANGED  connection_vars
     ∧ UNCHANGED  timers
     ∧ UNCHANGED  shared_resources_vars
     ∧ UNCHANGED  output_buffers
     ∧ UNCHANGED ⟨now, internal_state, awaiting_resource,
     ready, awaiting_dispatch, internal_mode,
     currentMode, currentTransition, awaiting_mode⟩
```

$\land$ $PrintT$(⟨"start of the task", mt, "date", now⟩)

———————Tick————————————

$Tick \triangleq$

$\land\ now' = now + 1$

$\land\ deadline_timer' = [t \in Thread \mapsto$

IF $\lor (t \in ready \cup awaiting_resource)$

$\lor (t \in awaiting_dispatch \land deadline_timer[t] > 0)$ THEN

$deadline_timer[t] - 1$

ELSE $deadline_timer[t]]$

$\land\ execution_timer' = [t \in Thread \mapsto$

IF $t = computing_thread$ THEN

$execution_timer[t] + 1$

ELSE $execution_timer[t]]$

$\land\ period_timer' = [t \in Periodic \mapsto$

$period_timer[t] - 1]$

$\land$ UNCHANGED $thread_input_vars$

$\land$ UNCHANGED $input_buffers$

$\land$ UNCHANGED $output_buffers$

$\land$ UNCHANGED $connection_vars$

$\land$ UNCHANGED $threads_state$

$\land$ UNCHANGED $shared_resources_vars$

$\land$ UNCHANGED $\langle internal_state, currentMode, currentTransition\rangle$

——————————-Next————————————-

$Next \triangleq$

IF can_make_step THEN

$make_step$

ELSE IF $must_send$ THEN

$send_port$

ELSE IF $\exists\, d \in SharedData : must_release(d)$ THEN

$release$

ELSE IF $hasComplete$ THEN

$complete$

ELSE IF $canUnblock$ THEN

$unblock$

ELSE IF $\exists\, th \in awaiting_dispatch : must_send_at_deadline(th)$ THEN

$send_at_deadline$

ELSE IF $enable_actual_mode_switch$ THEN

$actual_mode_switch$

ELSE IF $\exists\, th \in awaiting_dispatch : can_dispatch(th)$ THEN

$dispatch$
ELSE IF $canResume$ THEN
$resume$
ELSE IF $\exists\, d \in SharedData : must_access(d)$ THEN
$acccess$
ELSE IF $must_set_port$ THEN
set_port
ELSE $Tick$

$Spec \triangleq Init \land \Box[Next]_{total_vars}$

MODULE *Ports*

EXTENDS $Naturals$, $aadl_model$, $connections$, TLC

CONSTANTS $Data$, bot_data

$invPort(tab) \triangleq [x \in Thread \mapsto \{p \in \text{DOMAIN } (tab) : tab[p] = x\}]$

$Check_ports_disjoint \triangleq$
$\land\ Out_event_port \cap In_event_port = \{\}$
$\land\ Out_event_port \cap In_data_port = \{\}$
$\land\ Out_event_port \cap In_event_data_port = \{\}$
$\land\ Out_event_port \cap Out_data_port = \{\}$
$\land\ Out_event_port \cap Out_event_data_port = \{\}$
$\land\ In_event_port \cap Out_data_port = \{\}$
$\land\ In_event_port \cap In_data_port = \{\}$
$\land\ In_event_port \cap Out_event_data_port = \{\}$
$\land\ In_event_port \cap In_event_data_port = \{\}$
$\land\ Out_data_port \cap In_data_port = \{\}$
$\land\ Out_data_port \cap In_event_data_port = \{\}$
$\land\ Out_data_port \cap Out_event_data_port = \{\}$
$\land\ In_data_port \cap In_event_data_port = \{\}$
$\land\ In_data_port \cap Out_event_data_port = \{\}$
$\land\ Out_event_data_port \cap In_event_data_port = \{\}$

$Check_ports_protocols \triangleq$
$\land\ protocol_event \in [In_event_port \to \{\text{“AllItems”}, \text{“OneItem”}\}]$
$\land\ protocol_event_data \in [In_event_data_port \to \{\text{“AllItems”}, \text{“OneItem”}\}]$

$Check_ports_attachments \triangleq$
Ports are attached to threads, threads are considered as opaque types

$\land$ $in_data_port \in [In_data_port \rightarrow Thread]$
$\land$ $thread_in_data_port \in [Thread \rightarrow \textsc{subset}\ In_data_port]$
$\land$ $thread_in_data_port = invPort(in_data_port)$
$\land$ $out_data_port \in [Out_data_port \rightarrow Thread]$
$\land$ $thread_out_data_port \in [Thread \rightarrow \textsc{subset}\ Out_data_port]$
$\land$ $thread_out_data_port = invPort(out_data_port)$
$\land$ $in_event_port \in [In_event_port \rightarrow Thread]$
$\land$ $thread_in_event_port \in [Thread \rightarrow \textsc{subset}\ In_event_port]$
$\land$ $thread_in_event_port = invPort(in_event_port)$
$\land$ $out_event_port \in [Out_event_port \rightarrow Thread]$
$\land$ $thread_out_event_port \in [Thread \rightarrow \textsc{subset}\ Out_event_port]$
$\land$ $thread_out_event_port = invPort(out_event_port)$
$\land$ $in_event_data_port \in [In_event_data_port \rightarrow Thread]$
$\land$ $thread_in_event_data_port \in [Thread \rightarrow \textsc{subset}\ In_event_data_port]$
$\land$ $thread_in_event_data_port = invPort(in_event_data_port)$
$\land$ $out_event_data_port \in [Out_event_data_port \rightarrow Thread]$
$\land$ $thread_out_event_data_port \in [Thread \rightarrow \textsc{subset}\ Out_event_data_port]$
$\land$ $thread_out_event_data_port = invPort(out_event_data_port)$

$Check_setup \triangleq$
$\land$ $Check_ports_disjoint$
$\land$ $Check_ports_protocols$
$\land$ $Check_ports_attachments$
$\land$ $data_connection \in [In_data_port \rightarrow Out_data_port]$
$\land$ $event_connection \in [Out_event_port \rightarrow \textsc{subset}\ (In_event_port)]$
$\land$ $event_data_connection \in [Out_event_data_port \rightarrow \textsc{subset}\ (In_event_data_port)]$

VARIABLES
$idp_delivered, idp_fresh, idp_env,$ data ports variables
$iep_delivered, iep_fresh, iep_event_cpt,$ event ports variables
$iedp_delivered, iedp_fresh, iedp_event_data_queue,$ event data ports variables
$out_data_buffer, out_event_data_buffer, out_event_buffer,$
$out_data_fresh, out_event_fresh, out_event_data_fresh$

$idp \triangleq$ INSTANCE $InDataPort$ WITH
$Data \leftarrow Data,$
$bot_data \leftarrow bot_data,$
$In_data_port \leftarrow In_data_port,$
$delivered \leftarrow idp_delivered,$
$fresh \leftarrow idp_fresh,$
$env \leftarrow idp_env$

$iep \triangleq$ INSTANCE $InEventPort$ WITH
$protocol \leftarrow protocol_event,$
$In_event_port \leftarrow In_event_port,$

$delivered \leftarrow iep_delivered,$
$fresh \leftarrow iep_fresh,$
$event_cpt \leftarrow iep_event_cpt$

$iedp \triangleq$ INSTANCE $InEventDataPort$ WITH
$protocol \leftarrow protocol_event_data,$
$Data \leftarrow Data,$
$In_event_data_port \leftarrow In_event_data_port,$
$delivered \leftarrow iedp_delivered,$
$fresh \leftarrow iedp_fresh,$
$event_data_queue \leftarrow iedp_event_data_queue$

$odp \triangleq$ INSTANCE $OutDataPort$ WITH
$Data \leftarrow Data,$
$bot_data \leftarrow bot_data,$
$data_connection \leftarrow data_connection,$
$In_data_port \leftarrow In_data_port,$
$env \leftarrow idp_env,$
$buffer \leftarrow out_data_buffer,$
$fresh \leftarrow out_data_fresh$

$oep \triangleq$ INSTANCE $OutEventPort$ WITH
$Out_event_port \leftarrow Out_event_port,$
$event_connection \leftarrow event_connection,$
$In_event_port \leftarrow In_event_port,$
$max_event \leftarrow max_event,$
$event_cpt \leftarrow iep_event_cpt,$
$buffer \leftarrow out_event_buffer,$
$fresh \leftarrow out_event_fresh$

$oedp \triangleq$ INSTANCE $OutEventDataPort$ WITH
$Data \leftarrow Data,$
$bot_data \leftarrow bot_data,$
$Out_event_data_port \leftarrow Out_event_data_port,$
$In_event_data_port \leftarrow In_event_data_port,$
$event_data_connection \leftarrow event_data_connection,$
$event_data_queue \leftarrow iedp_event_data_queue,$
$buffer \leftarrow out_event_data_buffer,$
$fresh \leftarrow out_event_data_fresh$

$TypeInvariant \triangleq$
$\land\ idp!TypeInvariant$
$\land\ iep!TypeInvariant$
$\land\ iedp!TypeInvariant$
$\land\ odp!TypeInvariant$
$\land\ oep!TypeInvariant$

$\land$ $oedp!TypeInvariant$
$\land$ $connectionsInvariant$

$Init \triangleq$
$\land$ $connectionsInit$
$\land$ IF $Check_setup$ THEN
$\land$ $idp!Init$
$\land$ $iep!Init$
$\land$ $iedp!Init$
$\land$ $odp!Init$
$\land$ $oep!Init$
$\land$ $oedp!Init$
ELSE
$\land$ $PrintT$("ports setup failed")
$\land \neg Check_ports_disjoint \Rightarrow PrintT$("port confusion")
$\land \neg Check_ports_protocols \Rightarrow PrintT$("illegal protocol")
$\land \neg Check_ports_attachments \Rightarrow PrintT$("illegal attachment")

MODULE $OutDataPort$

CONSTANTS $Data$, Out_data_port, In_data_port, bot_data

AS 5506 p.122 Data connections are restricted to 1-n connectivity, i.e., a data port can have multiple outgoing connections, but only one incoming connection.

ASSUME
$\land$ $bot_data \in Data$

VARIABLES env, $data_connection$, $buffer$, $fresh$

$TypeInvariant \triangleq$
$\land$ $env \in [In_data_port \rightarrow Data]$
$\land$ $data_connection \in [In_data_port \rightarrow Out_data_port]$
$\land$ $buffer \in [Out_data_port \rightarrow Data]$
$\land$ $fresh \in [Out_data_port \rightarrow$ BOOLEAN $]$

$Init \triangleq$
$\land$ $buffer = [x \in Out_data_port \mapsto bot_data]$
$\land$ $fresh = [x \in Out_data_port \mapsto$ FALSE$]$

$store(dp) \triangleq$
$\land$ $dp \in$ SUBSET Out_data_port
$\land$ $env' = [x \in In_data_port \mapsto$
IF $data_connection[x] \in dp \land fresh[data_connection[x]]$

THEN $buffer[data_connection[x]]$
ELSE $env[x]]$
$\wedge$ $fresh' = [x \in Out_data_port \mapsto$
IF $x \in dp$ THEN FALSE
ELSE $fresh[x]]$
$\wedge$ UNCHANGED $buffer$

MODULE *OutEventPort*

This module is concerned by outeventports.
It makes also the link between outeventports and their
corresponding ineventports.
For model checking purposes we set a number maximal of events that can be raised.

EXTENDS $Naturals$, $FiniteSets$, TLC

CONSTANT Out_event_port, In_event_port, max_event

ASSUME
For model checking purposes we set a maximal number of events that can be raised.
Remark : the in event ports linked to out event ports are in fact
shared variables of the module InEventPort. In this module
the variable cpt_event[x] is incremented only. The event
is considered from now on in the environment.
It will be later delivered and the corresponding counter will be
decremented in the module InEvent by the operation Deliver.
$\wedge$ $max_event \in [In_event_port \rightarrow Nat]$ this max is set for model checking purposes only.

VARIABLES $event_cpt$, $event_connection$, $buffer$, $fresh$

The connection between outeventports and ineventports
an out event port can be connected to more than obe in event port.
$TypeInvariant \triangleq$
$\wedge$ $event_cpt \in [In_event_port \rightarrow Nat]$
$\wedge$ $event_connection \in [Out_event_port \rightarrow$ SUBSET $(In_event_port)]$
$\wedge$ $buffer \in [Out_event_port \rightarrow$ BOOLEAN $]$
$\wedge$ $fresh \in [Out_event_port \rightarrow$ BOOLEAN $]$

$Init \triangleq$
$\wedge$ $buffer = [x \in Out_event_port \mapsto$ FALSE$]$
$\wedge$ $fresh = [x \in Out_event_port \mapsto$ FALSE$]$

$RaiseEvent(P) \triangleq$
$\wedge$ $P \in$ SUBSET (Out_event_port)
$\wedge$ $event_cpt' =$
$[x \in In_event_port \mapsto$

```
LET nb_sent ≜ Cardinality({oep ∈ P : x ∈ event_connection[oep] ∧ fresh[oep]})IN
  IF event_cpt[x] + nb_sent < max_event[x] THEN event_cpt[x] + nb_sent
   ELSE event_cpt[x]]
∧ fresh' = [x ∈ Out_event_port ↦
   IF x ∈ P THEN FALSE
    ELSE buffer[x]]
∧ ∀ x ∈ In_event_port :
 LET nb_sent ≜ Cardinality({oep ∈ P : x ∈ event_connection[oep] ∧ buffer[oep]})IN
   (event_cpt[x] + nb_sent ≥ max_event[x]) ⇒ PrintT(⟨"max_event on out event port ", x, " reached"⟩)
∧ UNCHANGED buffer
```

```
MODULE OutEventDataPort
```

Semantics : SAE AS5506 p. 122

```
EXTENDS Sequences, Naturals, FiniteSets

CONSTANTS Data, bot_data, Out_event_data_port,
In_event_data_port

ASSUME
  ∧ bot_data ∈ Data

VARIABLES event_data_queue, event_data_connection, buffer, fresh

TypeInvariant ≜
  ∧ event_data_queue ∈ [In_event_data_port → Seq(Data)]
  ∧ event_data_connection ∈ [Out_event_data_port → SUBSET (In_event_data_port)]
  ∧ buffer ∈ [Out_event_data_port → Data]
  ∧ fresh ∈ [Out_event_data_port → BOOLEAN ]

Init ≜
  ∧ buffer = [x ∈ Out_event_data_port ↦ bot_data]
  ∧ fresh = [x ∈ Out_event_data_port ↦ FALSE]

RaiseEvent(P) ≜
  ∧ P ∈ SUBSET Out_event_data_port
  ∧ event_data_queue' =
    [iedp ∈ In_event_data_port ↦
     IF {oep ∈ P : fresh[oep] ∧ iedp ∈ event_data_connection[oep]} ≠ {} THEN
       LET oep ≜ CHOOSE oep ∈ P : fresh[oep] ∧ iedp ∈ event_data_connection[oep]IN
         Append(event_data_queue[iedp], buffer[oep])
      ELSE event_data_queue[iedp]]
  ∧ fresh' = [x ∈ Out_event_data_port ↦
  IF x ∈ P THEN FALSE
   ELSE fresh[x]]
```

$\land$ UNCHANGED *buffer*

MODULE *InDataPort*

SAE AS 5506 p. 99
Data ports are intended for transmission os state data such as signals.
Therefore, no queuing is supported for data ports. A thread can
determine whether the input buffer of an in data port has new
data at his dispatch by checking the port status, which is accessible
through the port variable.
p.98 Data and event data ports are used to transmit data between threads.
They appear to the thread as input and output buffers, accessible
in source text as port variables.

CONSTANTS *bot_data*, *Data*, *In_data_port*

delivered and frsh are the port variables
env in the environment not yet delivered

VARIABLES
delivered, port variable
fresh, port variable
env environment variable to be delivered

$TypeInvariant \triangleq$
$\land\ delivered \in [In_data_port \rightarrow Data]$
$\land\ fresh \in [In_data_port \rightarrow \text{BOOLEAN}]$
$\land\ env \in [In_data_port \rightarrow Data]$

$Init \triangleq$
$\land\ delivered = [x \in In_data_port \mapsto bot_data]$
$\land\ fresh = [x \in In_data_port \mapsto \text{FALSE}]$
$\land\ env = [x \in In_data_port \mapsto bot_data]$

$get(d, p, f) \triangleq$
$\land\ d = delivered[p]$
$\land\ f = fresh[p]$
$\land$ UNCHANGED $\langle delivered, fresh, env \rangle$

$deliver(P) \triangleq$
delivery from the environment to the thread data ports in P.
$\land\ P \subseteq In_data_port$
$\land\ delivered' = [x \in In_data_port \mapsto \text{IF } x \in P$
$\land\ env[x] \neq bot_data \text{ THEN } env[x] \text{ ELSE } delivered[x]]$
$\land\ fresh' = [x \in In_data_port \mapsto$
$\text{IF } x \in P \text{ THEN } env[x] \neq bot_data \text{ ELSE } fresh[x]]$
$\land\ env' = [x \in In_data_port \mapsto$

```
    IF x ∈ P THEN bot_data ELSE env[x]]
─────────────────────────────────────────────────────────────

──────────────────── MODULE InEventPort ────────────────────
EXTENDS Naturals, TLC

CONSTANT protocol, In_event_port

  This module describes when and how many messges
  are delivered from the environment to the port.
  to each apport is assigned a protocol wich either
  OneItem : one message from the environment is delivered
  AllItem : all messages from the environment are delivered

ASSUME protocol ∈ [In_event_port → {"OneItem", "AllItems"}]

  for each in event port iep
  delivered[iep] is the number of messages that have been actually
  delivered to the port.
  fresh[iep] tells if some messages have been actually delivered.
  event_cpt[iep] is the number of messages that have been sent to the port
  but not yet delivered. They are considered to be in the environment.
VARIABLES delivered, fresh, event_cpt

TypeInvariant ≜
  ∧ delivered ∈ [In_event_port → Nat]
  ∧ fresh ∈ [In_event_port → BOOLEAN ]
  ∧ event_cpt ∈ [In_event_port → Nat]

Init ≜
  ∧ delivered = [iep ∈ In_event_port ↦ 0]
  ∧ fresh = [iep ∈ In_event_port ↦ FALSE]
  ∧ event_cpt = [iep ∈ In_event_port ↦ 0]

  for each protocol, we specify how port variables delivered, fresh and event_cpt
  are updated at delivery
  One item protocol
cptOneItem(iep) ≜ IF event_cpt[iep] > 0 THEN event_cpt[iep] − 1 ELSE event_cpt[iep]
freshOneItem(iep) ≜ event_cpt[iep] > 0
deliveredOneItem(iep) ≜ IF event_cpt[iep] > 0 THEN 1 ELSE event_cpt[iep]
  All items protocol
cptAllItems(iep) ≜ 0
freshAllItems(iep) ≜ event_cpt[iep] > 0
deliveredAllItems(iep) ≜ event_cpt[iep]

deliver(P) ≜
  P is a set of in event ports
```

```
∧ delivered' =
  [iep ∈ In_event_port ↦ IF iep ∈ P ∧ protocol[iep] = "OneItem" THEN deliveredOneItem(iep)
   ELSE IF iep ∈ P ∧ protocol[iep] = "AllItems" THEN deliveredAllItems(iep)
   ELSE delivered[iep]]
∧ fresh' =
  [iep ∈ In_event_port ↦ IF iep ∈ P ∧ protocol[iep] = "OneItem" THEN freshOneItem(iep)
   ELSE IF iep ∈ P ∧ protocol[iep] = "AllItems" THEN freshAllItems(iep)
   ELSE fresh[iep]]
∧ event_cpt' =
  [iep ∈ In_event_port ↦ IF iep ∈ P ∧ protocol[iep] = "OneItem" THEN cptOneItem(iep)
   ELSE IF iep ∈ P ∧ protocol[iep] = "AllItems" THEN cptAllItems(iep)
   ELSE event_cpt[iep]]

get(p, c, f) ≜
 this operation has a read purpose only
 It tells for the port p, if it is fresh (f) and
 how many messages have been delivered (c)
 ∧ c = delivered[p]
 ∧ f = fresh[p]
```

```
──────────── MODULE InEventDataPort ────────────

EXTENDS Naturals, Sequences, TLC

CONSTANT protocol, Data, In_event_data_port

ASSUME protocol ∈ [In_event_data_port → {"OneItem", "AllItems"}]

VARIABLES delivered, fresh, event_data_queue

TypeInvariant ≜
 ∧ delivered ∈ [In_event_data_port → Seq(Data)]
 ∧ fresh ∈ [In_event_data_port → BOOLEAN ]
 ∧ event_data_queue ∈ [In_event_data_port → Seq(Data)]

Init ≜
 ∧ delivered = [x ∈ In_event_data_port ↦ ⟨⟩]
 ∧ fresh = [x ∈ In_event_data_port ↦ FALSE]
 ∧ event_data_queue = [x ∈ In_event_data_port ↦ ⟨⟩]

deliver(p) ≜
 ∧ delivered' = [x ∈ In_event_data_port ↦
  IF x ∈ p ∧ event_data_queue[x] ≠ ⟨⟩ THEN
    IF protocol[x] = "OneItem" THEN ⟨Head(event_data_queue[x])⟩
     ELSE event_data_queue[x]
   ELSE delivered[x]]
 ∧ fresh' = [x ∈ In_event_data_port ↦
```

```
   IF x ∈ p THEN event_data_queue[x] ≠ ⟨⟩ ELSE fresh[x]]
  ∧ event_data_queue' = [x ∈ In_event_data_port ↦
   IF x ∈ p THEN
    IF event_data_queue[x] ≠ ⟨⟩ ∧ protocol[x] = "OneItem" THEN
      Tail(event_data_queue[x])
     ELSE ⟨⟩
    ELSE event_data_queue[x]]

get(p, d, f) ≜
  ∧ d = delivered[p]
  ∧ f = fresh[p]
  ∧ UNCHANGED ⟨event_data_queue, delivered, fresh⟩
```

```
---------------- MODULE connections ----------------

EXTENDS aadl_model, TLC

CONSTANT bot_dport

VARIABLES data_connection, event_connection, event_data_connection

connectionsInvariant ≜
  ∧ data_connection ∈ [In_data_port → Out_data_port]
  ∧ event_connection ∈ [Out_event_port → SUBSET In_event_port]
  ∧ event_data_connection ∈ [Out_event_data_port → SUBSET In_event_data_port]

SetConnections(new_data_connection, new_event_connection, new_event_data_connection) ≜
  ∧ new_data_connection ∈ [In_data_port → Out_data_port]
  ∧ new_event_connection ∈ [Out_event_port → SUBSET In_event_port]
  ∧ new_event_data_connection ∈ [Out_event_data_port → SUBSET In_event_data_port]
  ∧ data_connection' = new_data_connection
  ∧ event_connection' = new_event_connection
  ∧ event_data_connection' = new_event_data_connection

connectionsInit ≜
  ∧ data_connection = data_connection_topology[InitialSystemMode]
  ∧ event_connection = event_connection_topology[InitialSystemMode]
  ∧ event_data_connection = event_data_connection_topology[InitialSystemMode]
```

```
---------------- MODULE aadl_model ----------------

threads

Thread ≜ {"aocs", "pf", "pl", "syst", "aocs_man", "comp", "acc"}

Lifted_Thread ≜ Thread ∪ {"bot_thread"}

inv(tab) ≜ [x ∈ Thread ↦ {p ∈ DOMAIN (tab) : tab[p] = x}]
```

$bot_thread \triangleq$ "bot_thread"

Internal Modes

$Internal_mode \triangleq$ {"mode_aocs", "mode_pf", "mode_pl", "mode_syst", "mode_aocs_man", "mode_comp", "mode_acc"}

$mode_thread \triangleq [x \in Internal_mode \mapsto$ IF $x =$ "mode_aocs" THEN "aocs"
ELSE IF $x =$ "mode_pf" THEN "pf"
ELSE IF $x =$ "mode_pl" THEN "pl"
ELSE IF $x =$ "mode_syst" THEN "syst"
ELSE IF $x =$ "mode_aocs_man" THEN "aocs_man"
ELSE IF $x =$ "mode_comp" THEN "comp"
ELSE "acc"]

$thread_mode \triangleq inv(mode_thread)$

$mode_trans_domain \triangleq \{\}$

$ModeTrans \triangleq [x \in mode_trans_domain \mapsto$ "mode_one"]

$initial_internal_mode \triangleq [x \in Thread \mapsto$ IF $x =$ "aocs" THEN "mode_aocs"
ELSE IF $x =$ "pf" THEN "mode_pf"
ELSE IF $x =$ "pl" THEN "mode_pl"
ELSE IF $x =$ "syst" THEN "mode_syst"
ELSE IF $x =$ "aocs_man" THEN "mode_aocs_man"
ELSE IF $x =$ "comp" THEN "mode_comp"
ELSE "mode_acc"]

$Internal_events \triangleq \{\}$

System modes

$SystemMode \triangleq$ {"mode1"}

$InitialSystemMode \triangleq$ "mode1"

$SystemModeTransitionDomain \triangleq \{\}$

$ModeTransition \triangleq \{\}$

$bot_trans \triangleq$ "bot_trans"

$Lifted_ModeTransition \triangleq ModeTransition \cup \{bot_trans\}$

$modeTransition \triangleq \{\}$

$NewMode \triangleq \{\}$

$ModeThreads \triangleq [x \in SystemMode \mapsto$ {"aocs", "pf", "pl", "syst",

$\text{“aocs_man”}, \text{“comp”}, \text{“acc”}\}]$

$Synchronized \triangleq \{\}$

$OneExec \triangleq [x \in SystemMode \mapsto \{\}]$

$AllExec \triangleq [x \in SystemMode \mapsto \{\}]$

Ports attachment to threads

$In_data_port \triangleq \{\text{“aocs_man_dt”}\}$
$in_data_port \triangleq [x \in In_data_port \mapsto \text{“aocs_man”}]$
$thread_in_data_port \triangleq inv(in_data_port)$

$Out_data_port \triangleq \{\text{“aocs_dt”}\}$
$out_data_port \triangleq [x \in Out_data_port \mapsto \text{“aocs”}]$
$thread_out_data_port \triangleq inv(out_data_port)$

$In_event_port \triangleq \{\text{“aocs_man_calc”}, \text{“comp_comp”}, \text{“acc_acc”}\}$
$in_event_port \triangleq [x \in In_event_port \mapsto \text{ IF } x = \text{“aocs_man_calc”} \text{ THEN } \text{“aocs_man”}$
$\text{ ELSE } \text{ IF } x = \text{“comp_comp”} \text{ THEN } \text{“comp”} \text{ ELSE } \text{“acc”}]$
$thread_in_event_port \triangleq inv(in_event_port)$

$Out_event_port \triangleq \{\text{“aocs_calc”}, \text{“pl_acc”}, \text{“pl_comp”}\}$
$out_event_port \triangleq [x \in Out_event_port \mapsto \text{ IF } x = \text{“aocs_calc”} \text{ THEN } \text{“aocs”}$
$\text{ ELSE } \text{“pl”}]$
$thread_out_event_port \triangleq inv(out_event_port)$

$In_event_data_port \triangleq \{\text{“aocs_TC_port”}, \text{“pf_TC_port”}, \text{“pl_TC_port”}\}$
$in_event_data_port \triangleq [x \in In_event_data_port \mapsto \text{ IF } x = \text{“aocs_TC_port”} \text{ THEN } \text{“aocs”}$
$\text{ ELSE } \text{ IF } x = \text{“pf_TC_port”} \text{ THEN } \text{“pf”} \text{ ELSE } \text{“pl”}]$
$thread_in_event_data_port \triangleq inv(in_event_data_port)$

$Out_event_data_port \triangleq \{\text{“syst_TC_port”}\}$
$out_event_data_port \triangleq [x \in Out_event_data_port \mapsto \text{“syst”}]$
$thread_out_event_data_port \triangleq inv(out_event_data_port)$

$Output_port \triangleq Out_data_port \cup Out_event_port \cup Out_event_data_port$
$Input_port \triangleq In_data_port \cup In_event_port \cup In_event_data_port$
$Ports \triangleq Output_port \cup Input_port$

Ports protocols

$protocol_event_data \triangleq$

$[x \in In_event_data_port \mapsto$ "OneItem"$]$
$protocol_event \triangleq$
$[x \in In_event_port \mapsto$ "OneItem"$]$
$max_event \triangleq$
$[x \in In_event_port \mapsto 10]$
$max_event_data \triangleq$
$[x \in In_event_data_port \mapsto 10]$

————————
Ports timing
————————

$Output_time \triangleq$ {"completion", "deadline", "execution", "none"}
$Input_time \triangleq$ {"dispatch", "execution", "none"}

$Input_time_domain \triangleq$ {⟨"mode_aocs_man", "aocs_man_dt"⟩,
⟨"mode_aocs_man", "aocs_man_calc"⟩,
⟨"mode_comp", "comp_comp"⟩, ⟨"mode_acc", "acc_acc"⟩,
⟨"mode_aocs", "aocs_TC_port"⟩, ⟨"mode_pf", "pf_TC_port"⟩,
⟨"mode_pl", "pl_TC_port"⟩}
$Output_time_domain \triangleq$ {⟨"mode_aocs", "aocs_dt"⟩, ⟨"mode_aocs", "aocs_calc"⟩,
⟨"mode_pl", "pl_acc"⟩, ⟨"mode_pl", "pl_comp"⟩,
⟨"mode_syst", "syst_TC_port"⟩}

$input_time \triangleq$
$[x \in Input_time_domain \mapsto$ "dispatch"$]$

$offsets_input_time \triangleq$
$[x \in Input_time_domain \mapsto \{\}]$

$output_time \triangleq$
$[x \in Output_time_domain \mapsto$ "completion"$]$

$offsets_output_time \triangleq$
$[x \in Output_time_domain \mapsto \{\}]$

Connection topology

$connection_topology_domain \triangleq SystemMode \cup ModeTransition$

$data_connection_topology \triangleq$
$[d \in connection_topology_domain \mapsto [x \in In_data_port \mapsto$ "aocs_dt"$]]$

$event_connection_topology \triangleq$
$[d \in connection_topology_domain \mapsto [x \in Out_event_port \mapsto$
IF $x =$ "aocs_calc" THEN {"aocs_man_calc"} ELSE
IF $x =$ "pl_acc" THEN {"acc_acc"} ELSE {"comp_comp"}$]]$

$event_data_connection_topology \triangleq$
$[d \in data_connection_topology_domain \mapsto [x \in Out_event_data_port$
$\mapsto \{\text{"aocs_TC_port"}, \text{"pf_TC_port"}, \text{"pl_TC_port"}\}]]$

timing information

$Periodic \triangleq \{\text{"aocs"}, \text{"pf"}, \text{"pl"}, \text{"syst"}\}$
$Aperiodic \triangleq \{\text{"aocs_man"}, \text{"comp"}, \text{"acc"}\}$

$Period \triangleq [x \in Periodic \mapsto 125]$
$WCET \triangleq [x \in Internal_mode \mapsto \text{IF } x = \text{"mode_aocs"} \text{ THEN } 35 \text{ ELSE}$
$\text{IF } x = \text{"mode_pf"} \text{ THEN } 12 \text{ ELSE}$
$\text{IF } x = \text{"mode_pl"} \text{ THEN } 18 \text{ ELSE}$
$\text{IF } x = \text{"mode_syst"} \text{ THEN } 25 \text{ ELSE}$
$\text{IF } x = \text{"mode_aocs_man"} \text{ THEN } 21 \text{ ELSE}$
$\text{IF } x = \text{"mode_comp"} \text{ THEN } 10 \text{ ELSE } 8]$

$Priority \triangleq [x \in Lifted_Thread \mapsto \text{IF } x = \text{"aocs"} \text{ THEN } 55 \text{ ELSE}$
$\text{IF } x = \text{"pf"} \text{ THEN } 185 \text{ ELSE}$
$\text{IF } x = \text{"pl"} \text{ THEN } 195 \text{ ELSE}$
$\text{IF } x = \text{"syst"} \text{ THEN } 205 \text{ ELSE}$
$\text{IF } x = \text{"aocs_man"} \text{ THEN } 35 \text{ ELSE}$
$\text{IF } x = \text{"comp"} \text{ THEN } 200 \text{ ELSE}$
$\text{IF } x = \text{"acc"} \text{ THEN } 200 \text{ ELSE } 0]$

$Deadline \triangleq [x \in Thread \mapsto \text{IF } x = \text{"aocs"} \text{ THEN } 100 \text{ ELSE}$
$\text{IF } x = \text{"pf"} \text{ THEN } 100 \text{ ELSE}$
$\text{IF } x = \text{"pl"} \text{ THEN } 100 \text{ ELSE}$
$\text{IF } x = \text{"syst"} \text{ THEN } 100 \text{ ELSE}$
$\text{IF } x = \text{"aocs_man"} \text{ THEN } 125 \text{ ELSE}$
$\text{IF } x = \text{"comp"} \text{ THEN } 125 \text{ ELSE } 125]$

dispatch information

$dispatch_ports \triangleq$
$[x \in Internal_mode \mapsto \text{IF } x = \text{"mode_aocs_man"} \text{ THEN } \{\text{"aocs_man_calc"}\} \text{ ELSE}$
$\text{IF } x = \text{"mode_comp"} \text{ THEN } \{\text{"comp_comp"}\} \text{ ELSE } \{\text{"acc_acc"}\}]$

$ports_to_deliver \triangleq$
$[x \in Ports \mapsto \text{IF } x = \text{"aocs_man_calc"} \text{ THEN } \{\text{"aocs_man_calc"}, \text{"aocs_man_dt"}\} \text{ ELSE}$
$\text{IF } x = \text{"comp_comp"} \text{ THEN } \{\text{"comp_comp"}\} \text{ ELSE } \{\text{"acc_acc"}\}]$

shared data information

$SharedData \triangleq \{\text{"sd"}\}$
$Protected \triangleq \{\text{"sd"}\}$

```
Unprotected ≜ {}

accessing_thread ≜ [x ∈ SharedData ↦ {"mode_comp", "mode_acc"}]

access_time_domain ≜ {⟨"sd", "mode_comp"⟩, ⟨"sd", "mode_acc"⟩}

last_access_resource_domain ≜ {⟨"sd", "comp"⟩, ⟨"sd", "acc"⟩}

access_time ≜
 [x ∈ access_time_domain ↦
  IF x = ⟨"sd", "mode_comp"⟩ THEN ⟨2, 4⟩
   ELSE ⟨2, 4⟩]

data_Priority ≜
  [x ∈ SharedData ↦ 200]
```

```
──────────── MODULE thread_behavior ────────────

EXTENDS Naturals, TLC, aadl_model

CONSTANTS Data, this, bot_data

VARIABLES  ports level
  idp_delivered, idp_fresh,        data ports variables
  iep_delivered, iep_fresh,        event ports variables
  iedp_delivered, iedp_fresh,      event data ports variables
  out_data_buffer, out_event_buffer, out_event_data_buffer,
  out_data_fresh, out_event_fresh, out_event_data_fresh

 port indexes of this
this_IDP ≜ thread_in_data_port[this]
this_IEP ≜ thread_in_event_port[this]
this_IEDP ≜ thread_in_event_data_port[this]
this_ODP ≜ thread_out_data_port[this]
this_OEP ≜ thread_out_event_port[this]
this_OEDP ≜ thread_out_event_data_port[this]

 ports of this
this_in_data_port_delivered ≜ [p ∈ this_IDP ↦ idp_delivered[p]]
this_in_data_port_fresh ≜ [p ∈ this_IDP ↦ idp_fresh[p]]
this_in_event_port_delivered ≜ [p ∈ this_IEP ↦ iep_delivered[p]]
this_in_event_port_fresh ≜ [p ∈ this_IEP ↦ iep_fresh[p]]
this_in_event_data_port_delivered ≜ [p ∈ this_IEDP ↦ iedp_delivered[p]]
this_in_event_data_port_fresh ≜ [p ∈ this_IEDP ↦ iedp_fresh[p]]

atomic_aocs(exec_timer) ≜
  ∧ out_data_buffer' = [x ∈ Out_data_port ↦
```

IF $x =$ "aocs_dt" THEN 7 ELSE $out_data_buffer[x]]$
$\land\ out_data_fresh' = [x \in Out_data_port \mapsto$
IF $x =$ "aocs_dt" THEN TRUE ELSE $out_data_fresh[x]]$
$\land\ out_event_buffer' = [x \in Out_event_port \mapsto$
IF $x =$ "aocs_calc" THEN TRUE ELSE $out_event_buffer[x]]$
$\land\ out_event_fresh' = [x \in Out_event_port \mapsto$
IF $x =$ "aocs_calc" THEN TRUE ELSE $out_event_fresh[x]]$
$\land$ UNCHANGED $\langle out_event_data_buffer, out_event_data_fresh\rangle$

$atomic_pf(exec_timer) \triangleq$
$\land$ TRUE
$\land$ UNCHANGED $\langle out_data_buffer, out_event_buffer, out_event_data_buffer,$
$out_data_fresh, out_event_fresh, out_event_data_fresh\rangle$

$atomic_pl(exec_timer) \triangleq$
$\land\ out_event_buffer' = [x \in Out_event_port \mapsto$
IF $(x =$ "pl_acc" $\lor\ x =$ "pl_comp"$)$ THEN TRUE ELSE $out_event_buffer[x]]$
$\land\ out_event_fresh' = [x \in Out_event_port \mapsto$
IF $(x =$ "pl_acc" $\lor\ x =$ "pl_comp"$)$ THEN TRUE ELSE $out_event_fresh[x]]$
$\land$ UNCHANGED $\langle out_data_buffer, out_event_data_buffer, out_data_fresh,$
$out_event_data_fresh\rangle$

$atomic_syst(exec_timer) \triangleq$
$\land\ out_event_data_buffer' = [x \in Out_event_data_port \mapsto$
IF $x =$ "syst_TC_port" THEN 7 ELSE $out_event_data_buffer[x]]$
$\land\ out_event_data_fresh' = [x \in Out_event_data_port \mapsto$
IF $x =$ "syst_TC_port" THEN TRUE ELSE $out_event_data_fresh[x]]$
$\land$ UNCHANGED $\langle out_data_buffer, out_event_buffer, out_data_fresh, out_event_fresh\rangle$

$out_data_fresh, out_event_fresh\rangle$

$atomic_aocs_man(exec_timer) \triangleq$
$\land$ TRUE
$\land$ UNCHANGED $\langle out_data_buffer, out_event_buffer, out_event_data_buffer,$
$out_data_fresh, out_event_fresh, out_event_data_fresh\rangle$

$atomic_comp(exec_timer) \triangleq$
$\land$ TRUE
$\land$ UNCHANGED $\langle out_data_buffer, out_event_buffer, out_event_data_buffer,$
$out_data_fresh, out_event_fresh, out_event_data_fresh\rangle$

$atomic_acc(exec_timer) \triangleq$
$\land$ TRUE
$\land$ UNCHANGED $\langle out_data_buffer, out_event_buffer, out_event_data_buffer,$
$out_data_fresh, out_event_fresh, out_event_data_fresh\rangle$

$atomic_step(exec_timer) \triangleq$
IF $this =$ "aocs" THEN $atomic_aocs(exec_timer)$
ELSE IF $this =$ "pf" THEN $atomic_pf(exec_timer)$
ELSE IF $this =$ "pl" THEN $atomic_pl(exec_timer)$
ELSE IF $this =$ "syst" THEN $atomic_syst(exec_timer)$
ELSE IF $this =$ "aocs_man" THEN $atomic_aocs_man(exec_timer)$
ELSE IF $this =$ "comp" THEN $atomic_comp(exec_timer)$
ELSE $atomic_acc(exec_timer)$

Annexe C

Exemple ArchiDyn en AADL

```
thread AOCS
        features
                TC_port : in event data port{
      Dequeue_Protocol => OneItem;
      Queue_Size => 10;
      Input_time => Dispatch;
      };
                calc: out event port{
      Output_time => Complete;
      };
                dt: out data port{
      Output_time => Complete;
      };
end AOCS;

thread implementation AOCS.Impl
        properties
                Dispatch_Protocol => Periodic;
                Compute_Execution_time =>  25 ms ..  25 ms;
                Deadline  =>  125 ms;
                Period =>  125 ms;
            SEI::Priority => 55;
end AOCS.Impl;

thread AOCS_MAN
        features
                calc: in event port{
      Dequeue_Protocol => OneItem;
      Queue_Size => 10;
      Input_time => Dispatch;
      };
                dt: in data port{
      Input_time => Dispatch;
      };
```

```
end AOCS_MAN;

thread implementation AOCS_MAN.Impl
        properties
                Dispatch_Protocol => Aperiodic;
                Compute_Execution_time => 11 ms .. 11 ms;
            SEI::Priority => 300;
                Deadline => 125 ms;
end AOCS_MAN.Impl;

thread PF
        features
                TC_port : in event data port{
      Dequeue_Protocol => OneItem;
      Queue_Size => 10;
      Input_time => Dispatch;
      };
end PF;

thread implementation PF.Impl
        properties
                Dispatch_Protocol => Periodic;
                Compute_Execution_time => 4 ms .. 4 ms;
                SEI::Priority => 185;
                Deadline => 125 ms;
                Period => 125 ms;
end PF.Impl;

thread PL
        features
                TC_port : in event data port{
      Dequeue_Protocol => OneItem;
      Queue_Size => 10;
      Input_time => Dispatch;
      };
                Acc: out event port{
      Output_time => Complete;
      };
                Comp: out event port{
      Output_time => Complete;
      };
end PL;

thread implementation PL.Impl
        properties
                Dispatch_Protocol => Periodic;
                Compute_Execution_time => 8 ms .. 8 ms;
                SEI::Priority => 195;
                Deadline => 125 ms;
```

```
                Period =>  125 ms;
end PL.Impl;

thread Acc
        features
                Acc: in event port{
      Dequeue_Protocol => OneItem;
      Queue_Size => 10;
      Input_time => Dispatch;
      };
                data1: requires data access sharedData {
                Required_Access => access read_write;
                Input_time => 2 ms .. 4 ms;
      };
end Acc;

thread implementation Acc.Impl
        properties
                Dispatch_Protocol => Aperiodic;
                Compute_Execution_time =>  8 ms ..  8 ms;
                SEI::Priority =>  200;
                Deadline  =>  125 ms;
end Acc.Impl;

thread Comp
        features
                Comp: in event port{
      Dequeue_Protocol => OneItem;
      Queue_Size => 10;
      Input_time => Dispatch;
      };
                data1: requires data access sharedData {
                Required_Access => access read_write;
                Input_time => 2 ms .. 4 ms;
      };
end Comp;

thread implementation Comp.Impl
        properties
                Dispatch_Protocol => Aperiodic;
                Compute_Execution_time =>  8 ms ..  8 ms;
                SEI::Priority =>  199;
                Deadline  =>  125 ms;
end Comp.Impl;

data sharedData
end sharedData;

data implementation sharedData.imp
```

```
properties
        Concurrency_Control_Protocol => none;
end sharedData.imp;


thread SYST
        features
                TC : out event data port{
      Output_time => Complete;
      };
end SYST;

thread implementation SYST.Impl
        properties
                Dispatch_Protocol => Periodic;
                Compute_Execution_time =>  5 ms ..  5 ms;
                SEI::Priority =>  205;
                Deadline  =>  125 ms;
                Period =>  125 ms;
end SYST.Impl;

system sys
end sys;

system implementation sys.sysimp
  subcomponents
    process1: process proc.processimp;
end sys.sysimp;

process proc
end proc;

process implementation proc.processimp
        subcomponents
                aocs: thread AOCS.Impl;
                pf: thread PF.Impl;
                pl: thread PL.Impl;
                syst: thread SYST.Impl;
                aocs_man: thread AOCS_MAN.Impl;
                comp: thread Comp.Impl;
                acc: thread Acc.Impl;
                shd: data sharedData.imp;
        connections
                dataAccess1 : data access shd -> comp.data1;
                dataAccess2 : data access shd -> acc.data1;
                EventConnection1: event port pl.Acc -> acc.Acc;
                EventConnection2: event port pl.Comp -> comp.Comp;
                EventConnection3: event port aocs.calc -> aocs_man.calc;
                EventData1 : event data port syst.tc -> aocs.TC_port;
```

```
                EventData1 : event data port syst.tc -> pf.TC_port;
                EventData1 : event data port syst.tc -> pl.TC_port;
end proc.processimp;

system sys
end sys;

system implementation sys.sysimp
  subcomponents
    process1: process proc.processimp;
end sys.sysimp;
```

Annexe D

Modèle de traduction Acceleo

```
<%
metamodel http:///AADL/core
import service.Wrapper
%>

<%script type="core.AadlSpec" name="evtcnx"%>
<%for (processImpl.connections.eventConnection.select("src.name == args(0)"))
{%><%if (current() ==
 current().eContainer().eventConnection.select("src.name == args(0)").nLast())
   {%><%dst.name%> <%}else{%> <%dst.name%>, <%}%><%}%>

<%script type="core.AadlSpec" name="evtdatacnx"%>
<%for (processImpl.connections.eventDataConnection.select("src.name == args(0)"))
{%><%if (current() ==
 current().eContainer().eventDataConnection.select("src.name == args(0)").nLast())
   {%><%dst.name%> <%}else{%> <%dst.name%>, <%}%><%}%>

<%script type="core.AadlSpec" name="shdw"%>
<%processImpl.connections.dataAccessConnection[src.name == args(0) &&
   ! (dst.getRequired_Access() == "read_only")].dstContext.name%>

<%script type="core.AadlSpec" name="shdr"%>
<%processImpl.connections.dataAccessConnection[src.name == args(0) &&
   ! (dst.getRequired_Access() == "write_only")].dstContext.name%>

<%script type="core.AadlSpec" name="aadl2tla" file="port_setup.tla"%>
```

```
------------------- MODULE ports_setup ---------------------------------

\***************
\* threads
\***************


Thread ==
 {"<%processImpl.subcomponents.threadSubcomponent.name.sep("\",\"")%>"}


bot_thread == "bot_thread"

Lifted_Thread == Thread \union {bot_thread}


inv(tab) == [x \in Thread  |-> {p \in DOMAIN(tab) : tab[p] = x}]

\* --------------------------
\* Ports attachment to threads
\* --------------------------


In_data_port ==
  {"<%processImpl.connections.dataConnection.dst.name.sep("\",\"")%>"}
in_data_port == [x \in In_data_port |->
 <%if (processImpl.connections.dataConnection.nSize() == 0){%> bot_thread ]
<%}else{%>
<%for (processImpl.connections.dataConnection){%>
<%if (current() == current().parent().dataConnection.nLast()){%>
          "<%dstContext.name%>" ]
<%}else{%>

          IF x = "<%dst.name%>" THEN "<%dstContext.name%>" ELSE
<%}%><%}%><%}%>
thread_in_data_port == inv(in_data_port)

Out_data_port ==
  {"<%processImpl.connections.dataConnection.src.name.sep("\",\"")%>"}
out_data_port == [x \in Out_data_port |->
 <%if (processImpl.connections.dataConnection.nSize() == 0){%> bot_thread ]
<%}else{%>
<%for (processImpl.connections.dataConnection){%>
<%if (current() == current().parent().dataConnection.nLast()){%>
```

```
          "<%srcContext.name%>" ]
<%}else{%>

          IF x = "<%src.name%>" THEN "<%srcContext.name%>" ELSE
<%}%><%}%><%}%>
thread_out_data_port == inv(out_data_port)


In_event_port ==
 {"<%processImpl.connections.eventConnection.dst.name.sep("\",\"")%>"}
in_event_port ==
 [x \in In_event_port |->
 <%if (processImpl.connections.eventConnection.nSize() == 0){%> bot_thread ]
<%}else{%>
<%for (processImpl.connections.eventConnection){%>
<%if (current() == current().parent().eventConnection.nLast()){%>
          "<%dstContext.name%>" ]
<%}else{%>

          IF x = "<%dst.name%>" THEN "<%dstContext.name%>" ELSE
<%}%><%}%><%}%>
thread_in_event_port == inv(in_event_port)

Out_event_port ==
 {"<%processImpl.connections.eventConnection.src.name.sep("\",\"")%>"}
out_event_port == [x \in Out_event_port |->
 <%if (processImpl.connections.eventConnection.nSize() == 0)
 {%> bot_thread ] <%}else{%>
<%for (processImpl.connections.eventConnection){%>
<%if (current() == current().parent().eventConnection.nLast()){%>
          "<%srcContext.name%>" ]
<%}else{%>

          IF x = "<%src.name%>" THEN "<%srcContext.name%>" ELSE
<%}%><%}%><%}%>
thread_out_event_port == inv(out_event_port)

In_eventData_port ==
 {"<%processImpl.connections.eventDataConnection.dst.name.sep("\",\"")%>"}
in_eventData_port == [x \in In_eventData_port |->
 <%if (processImpl.connections.eventDataConnection.nSize() == 0){%> bot_thread ]
<%}else{%>
<%for (processImpl.connections.eventDataConnection){%>
<%if (current() == current().parent().eventDataConnection.nLast()){%>
```

```
          "<%dstContext.name%>" ]
<%}else{%>

          IF x = "<%dst.name%>" THEN "<%dstContext.name%>" ELSE
<%}%><%}%><%}%>
thread_in_eventData_port == inv(in_eventData_port)

Out_eventData_port ==
 {"<%processImpl.connections.eventDataConnection.src.name.sep("\",\"")%>"}
out_eventData_port == [x \in Out_eventData_port |->
 <%if (processImpl.connections.eventDataConnection.nSize() == 0)
  {%> bot_thread ] <%}else{%>
<%for (processImpl.connections.eventDataConnection){%>
<%if (current() == current().parent().eventDataConnection.nLast()){%>
          "<%srcContext.name%>" ]
<%}else{%>

          IF x = "<%src.name%>" THEN "<%srcContext.name%>" ELSE
<%}%><%}%><%}%>
thread_out_eventData_port == inv(out_eventData_port)


\* ---------------
\* Ports protocols
\* ---------------

protocol_event_data ==
  [x \in In_event_data_port |->
 <%for (processImpl.connections.eventDataConnection){%>
  IF x = "<%dst.name%>" THEN "<%dst.getDequeue_Protocol()%>" ELSE
<%}%> "AllItems"]

protocol_event ==
  [x \in In_event_port |->  <%for (processImpl.connections.eventConnection)
 {%>
  IF x = "<%dst.name%>" THEN "<%dst.getDequeue_Protocol()%>" ELSE
<%}%> "AllItems"]

max_event ==
  [x \in In_event_port |-> <%for (processImpl.connections.eventConnection)
 {%>
  IF x = "<%dst.name%>" THEN "<%dst.getQueueSize()%>" ELSE
<%}%> 0]
```

```
\*-------------------
\* Connection topology
\*-------------------

data_connection_topology ==
  [x \in In_data_port |-> <%for (processImpl.connections.dataConnection){%>
<%if (current() ==  current().eContainer().dataConnection.nLast() ){%>
 <%src.name%>
  <%}else{%>
 IF x = <%dst.name%> THEN <%src.name%> ELSE
  <%}%>
<%}%>]

event_connection_topology ==
  [x \in Out_event_port |->
 <%for (threadType.features.eventPort.select("direction == "out"")){%>
 IF x = <%name%> THEN {<%getRootContainer().evtcnx(current().name)%>} ELSE  <%}%>{}]

event_data_connection_topology ==
  [x \in Out_event_data_port |->
 <%for (threadType.features.eventDataPort.select("direction == "out"")){%>
 IF x = <%name%> THEN
 {<%getRootContainer().evtdatacnx(current().name)%>} ELSE  <%}%>{}]


 <%processImpl.subcomponents.threadSubcomponent.getDispatchProtocol()%>
\*------------------------
\*timing information
\*------------------------



Periodic ==
{"<%processImpl.subcomponents.threadSubcomponent[current().getDispatchProtocol()
      == "Periodic"].name.sep("\",\"")%>"}
Aperiodic ==
 {"<%processImpl.subcomponents.threadSubcomponent[current().getDispatchProtocol()
  == "Aperiodic"].name.sep("\",\"")%>"}

Period == [ x \in Periodic |->
 <%for (processImpl.subcomponents.threadSubcomponent[current().getDispatchProtocol()
   == "Periodic"]){%>
         IF x = "<%name%>" THEN <%current().getPeriod()%> ELSE<%}%> 0 ]

WCET == [x \in Thread |->
```

```
 [ x \in Periodic |-> <%for (processImpl.subcomponents.threadSubcomponent){%>

            IF x = "<%name%>" THEN <%current().getWCET()%> ELSE<%}%> 0 ]

Priority == [x \in Lifted_Thread |->
 <%for (processImpl.subcomponents.threadSubcomponent){%>

            IF x = "<%name%>" THEN
  <%current().properties.propertyAssociation[current().propertyDefinition.name
  == "Priority"].propertyValue.valueString%> ELSE<%}%> 0 ]

Deadline == [x \in Thread |->
 <%for (processImpl.subcomponents.threadSubcomponent)
  {%>
            IF x = "<%name%>" THEN <%current().getDeadline()%> ELSE<%}%> 0 ]



\*--------------------------
\*shared data information
\*--------------------------



SharedData ==
  {"<%processImpl.subcomponents.dataSubcomponent.name.sep("\",\"")%>"}

Protected ==
 {"<%processImpl.subcomponents.dataSubcomponent
  [current().getConcurrencyControlProtocol() ==
  "Interrupt_Masking"].name.sep("\",\"")%>"}

Unprotected ==
  {"<%processImpl.subcomponents.dataSubcomponent
  [current().getConcurrencyControlProtocol() ==
   "None"].name.sep("\",\"")%>"}

Writer == [x \in SharedData |->
 <%for (processImpl.subcomponents.dataSubcomponent){%>
IF x = "<%name%>" THEN
 {"<%getRootContainer().shdw(current().name).sep("\",\"")%>"} ELSE
<%}%>{}]

Reader == [x \in SharedData |->
 <%for (processImpl.subcomponents.dataSubcomponent){%>
```

```
IF x = "<%name%>" THEN
 {"<%getRootContainer().shdr(current().name).sep("\",\"")%>"} ELSE
<%}%>{}]
================================================================================
```

Bibliographie

[AB90] N. Audsley and A. Burns. Real-time System Scheduling. Technical Report YCS 134, University of York, 1990.

[Abr96] Jean-Raymond Abrial. *The B-Book : Assigning programs to meanings*. Cambridge University Press, 1996.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[AD96] R. Alur and D.L. Dill. Automata-theoretic verification of real-time systems. *Formal Methods for Real-Time Computing*, Trends in Software Series :55–82, 1996.

[AD03] Peter Amey and Brian Dobbing. High Integrity Ravenscar. In *In 8th International Conference on Reliable Software Technologies – Ada-Europe 2003 (AE03*, 2003.

[Aer04] SAE Aerospace. *SAE AS5506 : ARCHITECTURE ANALYSIS and DESIGN LANGUAGE (AADL)*. SAE International, 2004.

[AFM+03] "T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi". "times : a tool for schedulability analysis and code generation of real-time systems". In *1st International Workshop on Formal Modeling and Analysis of Timed Systems*. Springer–Verlag, 2003.

[AG] Axlog-Geensyde. Ades. http ://www.axlog.fr/aadl/ades_fr.html.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[Air97] Airlines electronic engineering committee, 2551, Riva road, Annapolis, Maryland 21401. *Avionics Application Software Standard Interface, ARINC Specification 653*, january 1997.

[Alh98] Sinan S. Alhir. *UML in a Nutshell*. O'Reilly, September 1998.

[And92] Arnold André. *Systèmes de transitions finis et sémantique des processus communicants*. Masson, 1992.

[And96] Charles André. Representation and analysis of reactive behaviors : A synchronous approach. In *in Proc. CESA 201996*, 1996.

[Ass] Fremont Associates. Furness toolset. http ://www.furnesstoolset.com/index.htm.

[BB02] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. In *Readings in hardware/software co-design*, pages 147–159, Norwell, MA, USA, 2002. Kluwer Academic Publishers.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. http ://coq.inria.fr.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1) :64–83, Jan 2003.

[BCG99] Albert Benveniste, Benoit Caillaud, and Paul Le Guernic. From synchrony to asynchrony. In *International Conference on Concurrency Theory*, pages 162–177, 1999.

[BG92] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. *Sci. Comput. Program.*, 19(2) :87–152, 1992.

[BPPS00] ”V. Bertin, M. Poize, J. Pulou, and J. Sifakis”. ”towards validated real-time software”. *12 th Euromicro Conference on Real-Time Systems*, ”2000”.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process algebra*. Cambridge University Press, New York, NY, USA, 1990.

[COR+95] S. Crow, S. Owre, J. Rushby, N. Shankar, and S. Mandayam. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton*, http ://www.csl.sri.com/pvs, April 1995.

[CS93] L Coglianese and R Szymanski. Dssa-adage : An environment for architecture-based avionics development. In *Proc. AGARD*, 1993.

[DTA+08] Sébastien Demathieu, Frédéric Thomas, Charles André, Sébastien Gérard, and François Terrier. First experiments using the UML profile for MARTE. In *ISORC '08 : Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 50–57, Washington, DC, USA, 2008. IEEE Computer Society.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1990.

[Fei08] Peter H. Feiler. Efficient embedded runtime systems through port communication optimization. In *ICECCS '08 : Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pages 294–300, Washington, DC, USA, 2008. IEEE Computer Society.

[FGHL05] Peter H. Feiler, David P. Gluch, John J. Hudak, and Bruce A. Lewis. Pattern-based analysis of an embedded real-time system architecture. In *Architecture Description Languages*, volume 176/2005, pages 51–65. Springer Boston, October 2005.

[FRBF07] Ricardo Bedin França, Jean-François Rolland, Jean-Paul Bodeveix, and Mamoun Filali. Assessment of AADL's behavioral annex. In *FAC Formalisation des Activités Concurrentes, Toulouse, 15/03/2007-16/03/2007*, 2007.

[fSS91] European Cooperation for Space Standardization. Software engineering standards, February 1991. PSS-05-0.

[fSS96] European Cooperation for Space Standardization. Space project management, April 1996. ECSS-M-30A.

[fSS03] European Cooperation for Space Standardization. Space engineering - software - part 1 : Principles and requirements, November 2003. ECSS-E-40 Part 1B.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94 : The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, December 1994.

[GGpT+03] Paul Le Guernic, Paul Le Guernic, Jean pierre Talpin, Jean pierre Talpin, Jean christophe Le Lann, Jean christophe Le Lann, and Projet Espresso. Polychrony for system design. *Journal of Circuits, Systems and Computers. World Scientific*, 12, 2003.

[GM94] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1994.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme : Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[GSVK+06] Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT '06 : Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 132–141, New York, NY, USA, 2006. ACM.

[Har87] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[HHK01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto : a time-triggered language for embedded programming. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2001.

[HK02] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine : predictable, portable real-time code. In *PLDI '02 : Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 315–326, New York, NY, USA, 2002. ACM.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111 :394–406, 1994.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21 :666–677, 1978.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.

[HR99] Nicolas Halbwachs and Pascal Raymond. Validation of synchronous reactive systems : From formal verification to automatic testing. In *Asian Computing Science Conference*, pages 1–12, 1999.

[HZPK08] Jérome Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. From the prototype to the final embedded system using the Ocarina AADL tool suite. *Trans. on Embedded Computing Sys.*, 7(4) :1–25, 2008.

[IEE93] IEEE/ISO/IEC. *ISO/IEC IEEE Std. 1003.1b-1993 realtime extensions*, 1993.

[IT96] ITU-T. Message sequence charts (msc). ITU Recommendation Z120, octobre 1996.

[ITPS08] Nassima Izerrouken, Xavier Thirioux, Marc Pantel, and Martin Strecker. Certifying an Automated Code Generator Using Formal Tools : Preliminary Experiments in the GeneAuto Project. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*, page (electronic medium), http ://www.sia.fr, 2008. Société des Ingénieurs de l'Automobile.

[JRCL05] Abrial J.-R., Metayer C., and Voisin L. Event-B Language. Technical report, Information Society Technologies, 2005.

[KRP+93] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Lam96] Leslie Lamport. The Windows Win32 threads API specification, May 1996.

[Lam02] Leslie Lamport. *Specifying Systems :The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lam05] Leslie Lamport. Real time is really simple. Technical report, Microsoft Research, 2005.

[LDL06] Maillet Luc, Thomas Dave, and Planche Luc. Architecture dynamique des logiciels enfouis. Technical report, ASTRIUM, 2006.

[LGLBLM91] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9) :1321–1336, Sep 1991.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Trans. Softw. Eng.*, 21(4) :336–355, 1995.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, 1973.

[LMdS08] Su-Young Lee, Frédéric Mallet, and Robert de Simone. Dealing with aadl end-to-end flow latency with uml marte. In *ICECCS '08 : Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems*

(iceccs 2008), pages 228–233, Washington, DC, USA, 2008. IEEE Computer Society.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.

[Mar91] F. Maraninchi. The Argos language : graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, 1991.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[Mil99] Robin Milner. *Communicating and mobile systems : the π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[Mon00] J.F. Monin. *Introduction aux méthodes formelles.* Hermès, 2000.

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the c2 style. *SIGSOFT Softw. Eng. Notes*, 21(6) :24–32, 1996.

[MQR95] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4) :356–372, 1995.

[MR98] Florence Maraninchi and Yann Rémond. Mode-automata : About modes and states for reactive systems. *Lecture Notes in Computer Science*, 1381 :185–200, 1998.

[MT97] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer–Verlag, 1997.

[OBE] OBEO. Acceleo. http ://www.acceleo.org.

[OMG07] OMG. *OMG Unified Modeling Language V2.1.2*, 11 2007.

[On-03] On-Line Applications Research Corporation. *RTEMS applications Ada user's guide*, 2003.

[OSA] OSATE. Open source aadl tool environment. http ://la.sei.cmu.edu/aadl/currentsite/releasenotes.html.

[RBC+07] Jean-François Rolland, Jean-Paul Bodeveix, David Chemouil, Mamoun Filali, and Dave Thomas. Towards a formal semantics for AADL execution model. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/08-01/02/08*, 2007.

[RBF+08a] Jean-François Rolland, Jean-Paul Bodeveix, Mamoun Filali, David Chemouil, and Thomas Dave. AADL modes for space software. In *Data Systems In Aerospace (DASIA), Palma de Majorca-Spain, 27/05/08-30/05/08*, page (electronic medium), http ://www.esa.int/publications, mai 2008. European Space Agency (ESA Publications).

[RBF+08b] Jean-François Rolland, Jean-Paul Bodeveix, Mamoun Filali, David Chemouil, and Thomas Dave. Modes in asynchronous systems. In *13th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2008), Belfast Ireland, 31/03/08-04/04/08*, page (electronic medium), http ://www.ieee.org/, avril 2008. IEEE.

[RC04] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems : A survey and a new proposal. *Real-Time Syst.*, 26(2) :161–197, 2004.

[RDC07] Jean-François Rolland, Thomas Dave, and David Chemouil. Utilisation d'AADL pour la conception de logiciels de vol satellite. *Génie Logiciel*, 80 :41–44, mars 2007.

[Rug07] A.E. Rugina. *Dependability modeling and evaluation - From AADL to stochastic Petri nets.* PhD thesis, Institut National Polytechnique de Toulouse, 2007.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3) :733–749, 1985.

[SDK+95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering, IEEE Transactions on*, 21(4) :314–335, Apr 1995.

[SLNM04] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a flexible real time scheduling framework. *Ada Lett.*, XXIV(4) :1–8, 2004.

[Spa] The Spacify project. Model-driven engineering and formal methods for critical embedded software. http ://spacify.gforge.enseeiht.fr/.

[Spb] The Spices projetc. Support for predictable integration of mission critical embedded. http ://www.spices-itea.org.

[Spi89] J. M. Spivey. *The Z notation : a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols : an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9) :1175–1185, Sep 1990.

[SSNB95] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6) :16–25, 1995.

[SVNY02] Inc. Springer-Verlag New York, editor. *Consolidated Ada reference manual : language and standard libraries*. New York, NY, USA, 2002.

[TBW92] Ken Tindell, Alan Burns, and Andy J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *IEEE Real-Time Systems Symposium*, pages 100–109, 1992.

[TOP] TOPCASED. Toolkit in open-source for critical applications and systems development. http ://www.topcased.org.

[Ves98] S. Vestal. *MetaH User's Manual*. Honewell Technology Drive, 1998. http ://www.htc.honeywell.com/metah/uguide.pdf.

[Wel04] Andy Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.

[Win93] Wind River Systems, Inc. *VxWorks Programmer's Guide*, 1993.