Table des matières

Table des figures

Chapitre 1

Introduction générale

Contents

1.1 Contexte de l'étude et contributions

Les systèmes temps-réel, répartis et embarqués (TR^{2}E) réagissent à des stimuli provenant de capteurs spécifiques. Ils fournissent la possibilité d'interagir avec leur environnement en collectant des données, ou en le contrôlant au travers de périphériques dédiés. Ces systèmes sont largement utilisés de nos jours dans des domaines d'activité variés, comme l'avionique [BW98], l'automobile [TYZP05], le militaire [PPSS00], l'ubiquitaire [NTI$^+$04] ou encore la robotique [MNL08].

Selon le contexte d'exécution de tels systèmes, une faute, une panne ou une erreur d'exécution peut se traduire en pertes civiles ou financières. Ils doivent donc pouvoir être évalués selon différents critères afin de s'assurer qu'ils ne produiront pas d'erreur, ou dans le but de prévoir leurs réactions si l'environnement d'exécution change de façon inattendue.

Ces critères sont liés à la performance, au comportement, ou encore à la gestion de ressources. Ces systèmes doivent respecter des contraintes industrielles qui sont souvent relatives à des standards. Ces derniers peuvent être spécifiques au domaine d'application du système TR^{2}E et couvrent tous les aspects allant du processus d'ingénierie (D0178B [Sta93]) à la réalisation (ARINC653 [ari07]).

Nous nous intéressons dans le cadre de nos travaux à cette phase de réalisation, quand l'architecture de ces systèmes a déjà été établie. Nous étudions les propriétés amenées par les différents aspects des systèmes TR^{2}E, que nous présentons ci-après.

1.1.1 Définitions générales

Nous donnons ici une définition des systèmes qui nous intéressent dans le cadre de notre étude et qui définissent différents aspects des systèmes TR^{2}E : les systèmes répartis, les systèmes embarqués, les systèmes temps-réel et les systèmes critiques.

Définition 1.1 ***(Systèmes répartis)*** *[Gos07]*

Un système réparti est un essaim d'unités de calcul échangeant des informations et travaillant de concert pour exécuter une tâche.

Ils nécessitent la mise en œuvre de mécanismes de répartition. Des infrastructures dédiées (les intergiciels) permettent de les masquer en proposant des abstractions adéquates. Ils s'appuient sur des modèles de répartition, comme les objets répartis (CORBA [OMG06a]), les appels de procédures distants (RPC [BN84]), le passage de messages (DDS [OMG07]) ou encore la mémoire partagée distribuée (TreadMarks [KCD+94]).

Définition 1.2 ***(Systèmes embarqués)*** *[Bar07]*

Un système embarqué est un sous-système dédié à une tâche particulière. Le nombre de tâches qu'il est en mesure d'effectuer est limité.

Il n'existe pas de modèle canonique de système embarqué : ils sont pensés, conçus et implantés pour répondre à des besoins spécifiques. Ils doivent cependant être :

- *Adaptables* car ils peuvent être déployés sur des éléments matériels variés ou même exotiques (processeurs spécifiques par exemple).
- *Peu exigeants* car leur environnement d'exécution peut être limité : contraintes économiques (répercussion sur le matériel sous-jacent), et ressources limitées.
- *Fiables*, car ils doivent fonctionner sans fautes ou pannes le plus longtemps possible, ou être capables de fonctionner avec des performances raisonnables en mode dégradé.

Définition 1.3 ***(Systèmes temps-réel)*** *[BW01]*

Un système temps-réel est un système qui doit fonctionner correctement dans un intervalle temporel spécifique en réponse à des stimuli extérieurs.

Ces systèmes doivent être déterministes et prédictibles (accès aux ressources matérielles en temps borné par exemple), de façon à pouvoir s'assurer *a priori* qu'ils respectent les exigences temporelles spécifiées. On distingue les échéances temporelles «dures», dont le respect est crucial pour une application, des échéances «souples» (ou «molles»).

Définition 1.4 ***(Systèmes critiques)*** *[Rus94]*

Un système critique est un système dont l'échec ou la panne peut avoir de lourdes conséquences financières, environnementales, ou humaines.

Ces systèmes sont évalués en fonction de leur criticité. Il en existe différents niveaux, relatifs à l'impact possible des dysfonctionnements. On distingue les catégories suivantes :

- *Business critical* : provoque des pertes coûteuses ;
- *Mission critical* : provoque l'échec de la mission ;
- *Life critical* : provoque des pertes humaines ou environnementales graves.

Ils doivent respecter des contraintes particulières tout au long de leur développement pour offrir des garanties sur leurs aspects critiques. Ils doivent être documentés, suivis depuis leur conception jusqu'à leur implantation et leur déploiement, testés et validés.

Nous sommes donc en mesure de définir la catégorie de systèmes que nous allons considérer dans la suite de ce manuscrit :

Définition 1.5 ***(Systèmes TR^{2}E)*** *[Hug05]*
Un système TR^{2}E est un système qui supporte une application répartie, devant respecter des contraintes temps-réel, et s'exécutant dans un environnement contraint. De tels systèmes peuvent être critiques.

Ils doivent donc être validés et vérifiés pour s'assurer de leur adéquation aux exigences et aux contraintes de leur environnement d'exécution. La notion de vérification est souvent associée à celle de validation. Les deux termes décrivent un processus assurant qu'un système correspond à des spécifications, mais avec un angle de vue différent :

Définition 1.6 *(Validation et Vérification) [FL00]*
- *La vérification peut être associée à un processus interne (respect de normes par exemple), résumé par cette question : «Sommes-nous en train de construire correctement le système ?»*
- *La validation peut être associée à un processus externe (respect de contrats, comme un cahier des charges), résumé par cette question : «Sommes-nous en train de construire un système correct ?»*

1.1.2 Complexité d'analyse des systèmes TR^{2}E

Les définitions précédentes ont mis en valeur les différentes contraintes qu'un système TR^{2}E est susceptible de subir. Nous précisons ici les relations entre ces contraintes et leur impact sur les analyses qu'il est nécessaire d'effectuer.

Criticité et temps-réel Les systèmes critiques et temps-réels doivent être déterministes sur des aspects que les méthodes formelles sont à même de garantir : leur comportement doit être prédictible, reproductible, simulable et analysable (par exemple, s'il est possible de modéliser un système avec des automates déterministes, alors il est possible de garantir ce déterminisme par tautologie).

Répartition Les systèmes répartis mettent en œuvre des canaux de communication entre les différents nœuds de l'application. La gestion de ces canaux rend complexe l'analyse du système, et concerne par exemple l'autorisation d'accès aux média de transfert, ou la gestion du multiplexage de requêtes.

Ces caractéristiques ont un impact fort sur les aspects temps-réels de l'application, via les délais de transmission ou les politiques de gestion des messages. Il est donc nécessaire de prendre en compte ces informations lors de l'analyse des aspects temps-réel d'un système TR^{2}E.

Embarqué Les systèmes embarqués peuvent subir des contraintes matérielles fortes ayant une influence sur le fonctionnement de l'application : temps d'accès aux périphériques, gestion des interruptions. Ces aspects impactent les éléments temps-réel d'un système TR^{2}E et doivent être pris en compte dans l'analyse d'icelui.

En conclusion, ces systèmes TR^{2}E sont complexes à analyser car ils amènent différents types de contraintes propres à chacun de leurs aspects (embarqués, temps-réel, répartis, critiques). Il est cependant nécessaire de les valider et de les vérifier avant de les déployer pour offrir des garanties quant à leur fonctionnement.

1.2 Validation et vérification des systèmes TR^{2}E

Valider et vérifier de tels systèmes nécessite de pouvoir raisonner sur des informations qui leurs sont relatives, et qui doivent être structurées. La modélisation de ces systèmes le permet, et nécessite le cadre d'un processus définissant de manière rigoureuse les informations à capturer et la façon de les structurer.

Ces processus mettent en œuvre des notations standardisées, spécifiques au domaine d'application.

Modélisation des systèmes TR^{2}E : Cette étape est essentielle pour l'analyse. Traditionnellement, des approches comme l'IDM («Ingénierie Dirigée par les Modèles») centrent le processus de développement autour de la modélisation du système, traditionnellement réalisée à l'aide de langages de description d'architecture (ADL, pour *Architecture Description Language*) [Med00].

Ils sont dans la plupart des cas considérés comme semi-formels :

Définition 1.7 *(Semi-formel)*
Langage : langage standardisé dont la sémantique est définie informellement en langage naturel, et donc soumise à interprétation.

Dans notre contexte, ils permettent de spécifier des systèmes TR^{2}E, structurant les informations, facilitant le partage ou encore l'intégration de travaux provenant de différentes équipes. Celles-ci traitent différents aspects d'un système à l'aide de spécifications séparées : spécifications fonctionnelles, matérielles ou encore temporelles.

Au sein d'une même spécification, les ADLs permettent d'exprimer les contraintes du système sous forme d'attributs spécifiques. Le langage AADL [SAE08] est un ADL dédié à la modélisation de systèmes TR^{2}E, dans le but de procéder à leur analyse. AADL aide l'ingénieur à séparer ses préoccupations, et facilite l'analyse du système en différenciant les informations temps-réel des informations embarquées ou réparties.

Cependant, *si, dans le domaine des systèmes TR^{2}E, une approche dirigée par les modèles facilite le processus de développement pour les ingénieurs, elle ne permet pas de garantir, seule, le respect des contraintes énoncées par le client*.

En effet, ces approches n'indiquent alors pas *comment* exploiter ces modèles dans l'optique de valider ou de vérifier le système considéré.

Analyse des systèmes TR^{2}E : L'analyse de ces systèmes doit garantir que des aspects comme leur ordonnancement, le dimensionnement de leurs ressources, ou encore leur comportement (détection d'interblocage, de famine, équité) sont corrects.

Il existe des méthodes spécifiques pour effectuer ces analyses, qui peuvent être classées en différentes catégories : la simulation, le test, la vérification par preuves, ou encore la validation de modèles par exploration (*model-checking*).

Les techniques de simulation ou de tests arrivent tard dans le processus de développement, car elles nécessitent des informations détaillées sur le système. En effet, dans un processus de développement itératif, les informations sur le matériel ou sur des contraintes particulières peuvent être connues seulement après de nombreuses itérations. De ce fait, les inadéquations du système peuvent être détectées tardivement, et leur correction être coûteuse.

Nous nous focalisons sur les méthodes d'analyse dites «formelles», qui peuvent apparaître tôt dans le processus de développement car elles permettent de raisonner sur des abstractions du système. Ces dernières sont en effet connues tôt dans le cycle de développement.

Ces méthodes sont dites formelles car elles s'articulent autour d'un formalisme de spécification ayant des bases mathématiques fixant leur sémantique. Différentes sortes de formalismes apparaissent dans l'utilisation des méthodes formelles :

- Le formalisme de spécification du système (réseaux de Petri, Z, B, VDM, Fiacre) [1] ;
- Le formalisme de spécification des contraintes du système (HOL, OCL, prédicats) : échéances temporelles, conditions d'activation (*trigger*) d'éléments du système, politique d'accès aux ressources, etc. ;
- Le formalisme de spécification des propriétés que l'on souhaite analyser au travers des outils (LTL, CTL, TLTL) : validation d'ordonnancement, vérification de causalité entre deux événements, inexistence de comportements fautifs, etc. Ce formalisme peut-être différent de celui utilisé pour spécifier les contraintes du système.

La base mathématique des formalismes permet aux outils de procéder à des analyses, et de produire des résultats garantissant (ou au contraire invalidant) le respect de contraintes.

Cependant, *si les méthodes formelles permettent à l'ingénieur d'avoir des garanties quant au respect de certaines contraintes, leur utilisation requiert souvent l'intervention d'experts dès lors que l'on s'intéresse à des aspects pointus du système*.

Typiquement, valider le comportement d'une application peut par exemple nécessiter l'utilisation d'automates exprimés dans des formalismes spécifiques, comme les réseaux de Petri. Si certaines propriétés, comme l'absence d'interblocage, peuvent être vérifiées automatiquement, analyser des liens de causalité entre événements peut nécessiter l'expression de formules de logique complexes permettant d'interroger les outils (*model-checker*).

De ce fait, malgré leur pertinence pour vérifier et valider les systèmes TR²E, l'intégration des méthodes formelles dans le cycle de développement reste limitée : l'enthousiasme des développeurs reste faible, et leur difficulté de prise en main rebute ceux qui seraient intéressés [Sin96]. Cette difficulté est liée à l'établissement d'un *bon* niveau d'abstraction pour spécifier le système, puis pour l'analyser.

Définition 1.8 *Le niveau d'abstraction d'une modélisation définit à quelle granularité le modèle créé modélise le système réel :*

- *Trop élevée, la vision du système devient généraliste, et la quantité d'information utilisable pour procéder à des analyses est trop faible ;*
- *Trop fine, la vision du système est trop détaillée : des informations peuvent parasiter le modèle. Les outils manipulant ces modèles sont souvent plus efficaces avec des modèles de taille raisonnable : trop d'informations peut augmenter ces modèles, sans pour autant être utile à l'analyse en cours.*

Il est à noter qu'au cours du processus de développement, ce niveau d'abstraction peut être amené à évoluer en fonction des analyses que l'on souhaite effectuer.

Intégration dans un processus de développement standardisé : Si l'utilité des modèles pour exprimer les contraintes d'un système, en avoir une vue globale et partager des informations entre équipes de développeurs n'est pas à remettre en cause, ceux-ci ne sont pas suffisants pour vérifier et valider les systèmes TR²E. L'utilisation de méthodes d'analyses (potentiellement formelles) est alors nécessaire.

1. Nous détaillerons ce type de formalisme plus avant dans ce document (section 2.2).

Ces systèmes doivent par ailleurs respecter des standards de développement, qui offrent des garanties sur le produit final. Ils sont contraignants, au vu des risques que les systèmes TR^{2}E peuvent engendrer. On peut citer, pour le domaine de l'avionique, le standard DO178-B [Sta93], ou l'IEC61508 [Bel06] pour le ferroviaire. Ils imposent des contraintes à tous les niveaux du processus de développement : enchaînement des phases (spécification, validation, implantation, etc.), restriction pour l'implantation (profil Ravenscar, interdisant l'allocation dynamique de tâches par exemple), documents et livrables à fournir, etc.

Cependant ces standards ne mentionnent rien quant à l'utilisation de méthodes formelles permettant d'effectuer des analyses. Leur mise en œuvre dans un processus de développement respectant ces standards soulève différentes questions clefs, auxquelles nous nous intéressons dans le cadre de nos travaux visant à aider à la construction de ces systèmes :

1. Au sein du processus de développement du système, à quel moment doit-on (ou peut-on) utiliser les méthodes formelles pour analyser le système ? Si elles interviennent trop tôt dans le processus, elles ne disposeront pas forcément de suffisamment d'informations pour avoir des résultats significatifs. A l'inverse, si elles sont utilisées trop tard, les résultats obtenus risquent d'exhiber des problèmes apparus très en amont, remettant en cause la viabilité du projet.

   → *Quand utiliser les méthodes formelles ?*

2. Selon le moment où l'on souhaite utiliser les méthodes formelles, quelle sera la plus adaptée au contexte de vérification ou de validation ?

   → *Quelle méthode choisir, selon quels critères ?*

3. Une fois décidé le moment de procéder à une analyse formelle dans le processus, quelles informations doivent être considérées ? Quels sont les outils à utiliser ?

   → *Comment utiliser les méthodes formelles ?*

4. L'analyse effectuée, il est nécessaire de prendre du recul quant aux résultats obtenus : puisque l'analyse du système va dépendre du niveau d'abstraction choisi, il est nécessaire d'interpréter les résultats en fonction de ce niveau d'abstraction. Par ailleurs, les hypothèses posées, soit par le concepteur soit par l'outil, ne sont-elles pas trop restrictives ?

   → *Quelle valeur accorder au(x) résultat(s) obtenu(s) ?*

5. Dans le cas où l'analyse produit des résultats pertinents pour le système, comment injecter ces résultat ou corriger la spécification analysée : quels mécanismes doivent être mis en place ?

   → *Comment intégrer ce(s) résultat(s) dans le processus de développement ?*

1.3 Contributions

Pour répondre aux problématiques soulevées, nous proposons de présenter, dans ce mémoire, les contributions suivantes :

1. *Quand utiliser les méthodes formelles ?*

   ⇒ Suivre un processus de développement dirigé par les analyses

   Nous proposons un processus de développement non plus basé sur une démarche de type IDM, mais sur une démarche de type IDV^2 (pour Ingénierie dirigée par les Vérifications et les Validations) [KHR08]. Ainsi, la question que nous posons n'est pas «Que dois-je modéliser ?», mais plutôt «Que puis-je vérifier ou valider sur mon système ?». Le processus s'articule donc autour des analyses à effectuer sur le système.

2. *Quelle méthode choisir, selon quels critères ?*

⇒ Sélection des méthodes formelles

Le choix d'une méthode formelle pour analyser un aspect du système considéré dépend du type de propriétés considéré. Le choix d'un outil se fait en fonction de ses performances pour le type de propriété, et en fonction de sa capacité à supporter le type d'analyse que l'on souhaite effectuer. Nous proposons donc des guides permettant de sélectionner les outils adaptés à l'analyse des propriétés à vérifier sur un système.

Les guides que nous proposons permettent de lier :

(a) Les propriétés à vérifier ou valider

(b) Les techniques à mettre en œuvre

(c) Les outils (en fonction des techniques choisies) à utiliser

3. *Comment utiliser les méthodes formelles ?*
   Quelle valeur accorder au(x) résultat(s) obtenu(s) ?
   ⇒ Guides de transformation et intégration dans le processus de développement
   L'utilisation de méthodes formelles ne doit pas être un frein dans le processus de développement : différentes méthodes, utilisant différents formalismes, doivent pouvoir être utilisées avec un coût réduit [RKH09a, RKH09b, RHK08].
   Nous proposons donc, à partir d'un langage pivot de spécification du système qui est standard, de l'exploiter (avec) ou de le transformer vers différents formalismes utilisés par les méthodes formelles, en proposant des règles applicables de façon systématique.
   Ce langage pivot doit capturer le plus d'informations possible pour la spécification des système TR^{2}E, afin de maximiser les possibilités d'analyses.
   Dans l'optique de réduire les coûts liés à l'utilisation de différentes méthodes formelles, nous proposons autant que faire se peut une automatisation des processus d'analyse.
4. *Comment intégrer ce(s) résultat(s) dans le processus de développement ?*
   ⇒ Traçabilité
   Nous proposons dans nos règles de transformation des mécanismes permettant de localiser un composant fautif lorsqu'une erreur ou une anomalie est détectée.

La figure 1 présente de façon générale le processus mettant en œuvre les propositions que nous avons faites à la section 1.3. Les chiffres dans les cercles de couleurs permettent de lier cette figure à nos propositions. La figure représente une itération du processus. Les certangles représentent une étape de cette itération. Le losange indique les actions qui permettent de passer d'une étape à une autre.

En partant d'une spécification du système dans une notation semi-formelle, et selon le type d'analyse que nous souhaitons effectuer, nous procédons à des transformations de modèle pour passer dans une notation formelle les supportant.

Les erreurs détectées sont alors signalées à l'utilisateur à deux niveaux : tout d'abord au niveau de la spécification formelle du système, puis à la spécification de haut niveau en langage de description d'architecture à l'aide des mécanismes de traçabilité mis en place.

1.4 Organisation du mémoire

Le chapitre 2 justifie notre approche : nous y présentons les principales notations semi-formelles du domaine permettant de satisfaire les contraintes de modélisation évoquées précédemment. Nous y présentons également les méthodes formelles existantes et les propriétés qu'elles sont les plus à même de traiter. Enfin, nous présentons et analysons différents cycles de développement susceptibles de supporter notre approche.

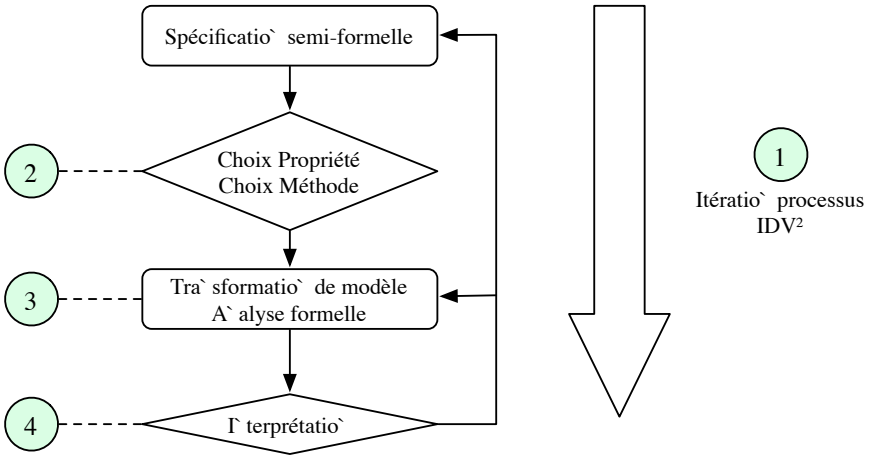


FIGURE 1 – Vue d'ensemble d'une itération du processus IDV^2

Dans le chapitre 3, nous présentons comment le langage AADL (*Architecture Analysis and Design Language*) peut servir de support au processus IDV^2 décrit à la figure 1. Ce langage y est analysé de façon à extraire et classer les informations qu'il est susceptible d'apporter pour effectuer des analyses sur le système.

Nous proposons la mise en œuvre de deux itérations de ce processus dans les chapitres 4 et 5, nous permettant d'éprouver la viabilité de notre approche. Nous y effectuons des analyses utilisant des méthodes formelles très différentes, afin d'étudier des aspects complémentaires des systèmes développés : nous mettons tout d'abord en œuvre une approche de validation par *model-checking* sur des modèles formels en réseaux de Petri, pour des analyses comportementales de l'applicatif ; puis nous mettons en œuvre une approche de vérification par *preuve de théorèmes* reposant sur une notation formelle (la notation Z), dans le but d'analyser l'architecture de l'exécutif supportant l'applicatif.

Nous présentons dans le chapitre 4 les règles de transformation d'AADL vers les réseaux de Petri pour la validation comportementale (détection d'interblocage, de famine). Nous montrons comment il est possible de factoriser ces règles lorsque l'on souhaite ajouter un nouveau formalisme dans le processus, en exhibant les règles de transformations pour deux variantes de réseaux de Petri : les colorés permettant de faire des analyses qualitatives (analyse des flots de messages dans le système), et les temporels permettant de faire des analyses quantitatives (ordonnancement, dimensionnement de ressources).

Nous présentons ensuite dans le chapitre 5 comment procéder à la vérification de l'exécutif supportant l'application décrite en AADL, dans le cadre d'une deuxième itération du processus IDV^2. Nous y présentons la spécification formelle en Z de PolyORB, l'exécutif que nous avons retenu. Cette spécification a été validée syntaxiquement et a permis de prouver différents théorèmes présentés en annexe C. Nous détaillons également comment utiliser cette spécification pour procéder à des analyses plus poussées, via des opérations que nous avons formellement spécifiées, permettant par exemple de la configurer pour une instance précise d'exécutif.

Nous présentons, dans le chapitre 6, la chaîne d'outils que nous avons mis en place au cours de ce travail de thèse. Nous y motivons le choix de nos outils, et expliquons dans quel ordre les opérations sont effectuées.

Enfin, dans le chapitre 7 nous montrons comment utiliser sur un cas d'étude notre processus de Vérification et de Validation. Nous y indiquons les interactions que notre chaîne d'outils doit effectuer avec l'utilisateur final. Nous précisons également dans quelle mesure cette chaîne est automatisée. Elle met en œuvre les outils adaptés aux analyses que nous effectuons, sélectionnés en accord avec les propriétés

que nous souhaitons analyser.

La figure 2 reprend la figure 1, en indiquant pour chaque étape quelles sont les notations, les formalismes et les outils utilisés. Elle montre comment, dans ce mémoire, nous proposons d'effectuer deux itérations successives du processus IDV^2. Les chapitres concernés sont indiqués dans des carrés de couleur.

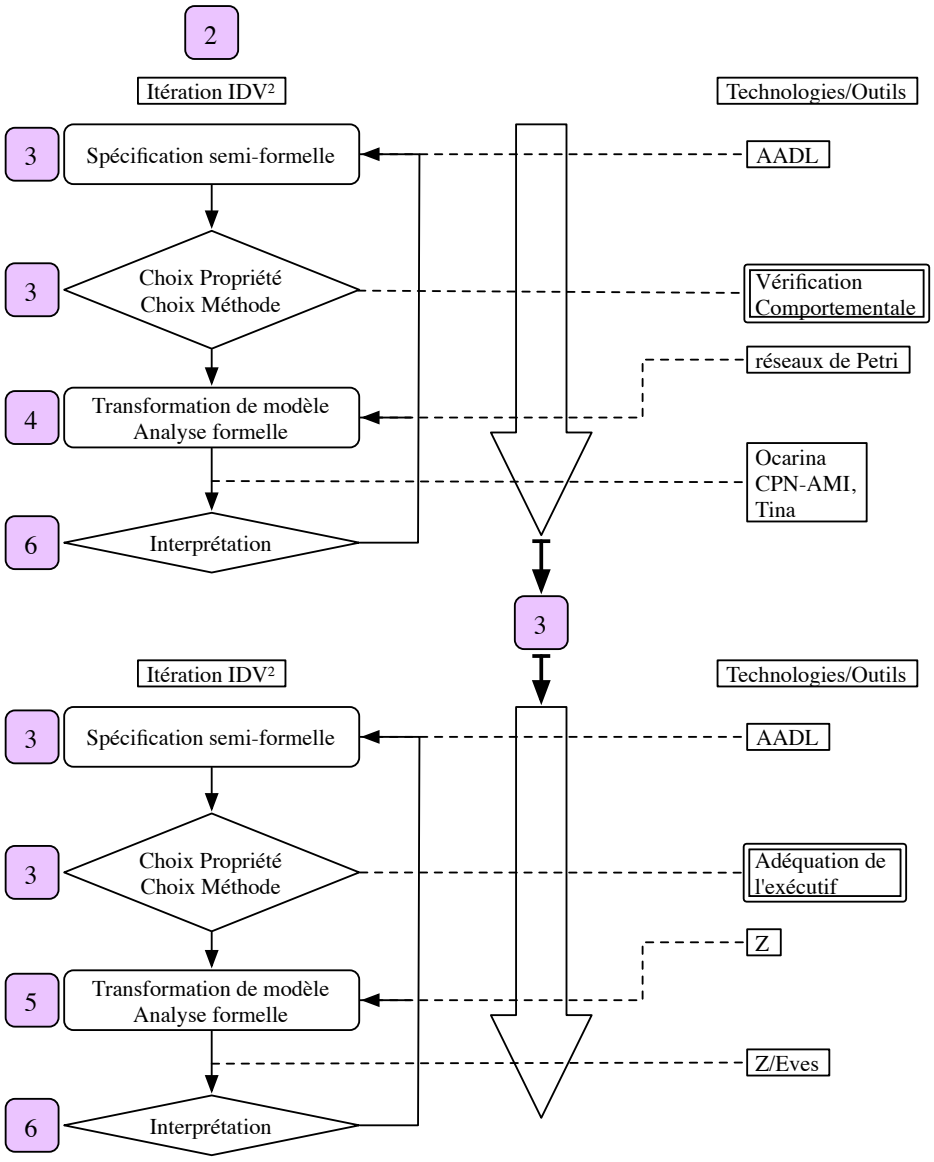


FIGURE 2 – Itération du processus IDV^2 instancié selon les technologies présentées dans ce mémoire

Chapitre 2

Etat de l'art et positionnement

Contents

La validation et la vérification de systèmes TR^{2}E s'effectue au travers de différentes analyses, chacune reposant sur une abstraction structurée et complète. Ces abstractions modélisent un aspect particulier du système, lié à certaines contraintes :

- Les contraintes fonctionnelles et non fonctionnelles : comportement cohérent du système en réponses à des stimuli, empreinte mémoire, respect d'échéances temporelles ;
- Les contraintes structurelles : préparation de l'application en vue de la distribuer, utilisation de composants existants, hiérarchie entre les composants ;
- Les contraintes de déploiement : choix de l'exécutif sous-jacent assurant l'exécution du système, description du matériel utilisé (type de processeur, de mémoire, de bus), communication et interaction entre les différents nœuds du système.

Les langages de description d'architecture (ADL, pour *Architecture Description Langage*) permettent de les exprimer de façon condensée et structurée. Ils servent de points d'entrée pour procéder à l'analyse du système, ou encore de notation pivot pour produire un modèle dans un autre formalisme.

Dans le cadre de nos travaux de thèse, nous souhaitons utiliser une notation pivot capturant le maximum d'informations sur les systèmes TR^{2}E : elle nous permettra de produire des modèles formels sur lesquels nous pourrons effectuer des analyses.

Dans un premier temps, nous exposons nos critères d'analyse des ADLs existants, puis nous les étudions pour identifier le plus adapté à nos objectifs. Comme nous souhaitons utiliser les méthodes formelles au cours du processus de développement, nous catégorisons également les propriétés que ces méthodes permettent de vérifier sur un système en vue d'établir un guide de sélection les concernant. Enfin, nous nous intéressons aux principaux processus de développement existants (processus en cascade, en V, en spirale, ...) dans le but de déterminer le plus adapté à l'intégration notre démarche de validation et de vérification.

2.1 Langages de description d'architecture

La description des éléments sur lesquels le système est construit, leurs interactions, les patrons guidant leurs compositions, ou encore les contraintes sur ces patrons sont des informations nécessaires à l'établissement d'une architecture logicielle. Un ADL est un langage qui répond à ces besoins pour spécifier (pas nécessairement pour implanter), l'architecture d'un système.

Avant d'analyser ces ADLs, nous présentons notre grille d'évaluation (section 2.1.1) des caractéristiques de différents ADLs de la littérature (Fractal ADL, UML/MARTE, AADL, LwCCM, Wright).

2.1.1 Éléments caractéristiques

Nous présentons dans cette section un ensemble d'éléments qui caractérisent les ADLs, pour les comparer afin de savoir quelle catégorie d'ADL peut au mieux servir nos objectifs. Pour cela, nous nous sommes appuyés sur les travaux de [Med00], ainsi que sur les résultats de [RKMS07].

Les composants Un composant peut être assimilé à une unité de calcul ou de stockage de données élémentaires. Cela peut aller de la simple procédure à tout ou partie d'une application. Un composant peut fournir ou requérir un ensemble de service, via ses interfaces. Par exemple, dans le modèle de composants de CORBA [OMG06a], ou encore dans le modèle de composants de Flex-eWare (FCM) [RKMS07], on retrouve la notion d'interfaces requises ou fournies.

Les composants permettent de définir un élément du système et de lui attacher spécifiquement un ensemble de propriétés utiles pour l'analyse.

Les connecteurs Ils symbolisent les canaux de communication entre les composants, ou plus généralement, leurs interactions (appels de procédure distants, échanges de messages). A l'inverse des composants, les connecteurs peuvent ne pas correspondre à une unité compilable du système, mais renvoyer diverses informations pour l'éditeur de lien, ou encore représenter des séquences d'appels de procédure.

Les connecteurs permettent de caractériser les interactions entre composants, influant leurs comportements et donc incidemment leur analyse.

La configuration Elle définit un assemblage de composants interagissant via des connecteurs. Elle contient les informations nécessaires au déploiement de l'application et à la génération de code.

La configuration permet de cibler une instance précise du système lors de son analyse, ainsi affinée par la prise en compte de propriétés spécifiques.

La standardisation Elle permet de déterminer les interactions entre les développeurs, fixe la sémantique des modèles, permet de faire adopter l'ADL dans les milieux industriels, et offre des garanties à l'utilisateur final.

Enfin, elle permet d'améliorer l'interopérabilité des outils manipulant une spécification ADL dans le but d'en faire des analyses.

Capacité d'analyse La capacité d'analyse liée à un ADL est cruciale. Plus elle est élevée (informations disponibles, outils existants), plus il sera facile d'exploiter l'ADL considéré pour vérifier ou valider le système spécifié.

Nous présentons maintenant les principaux ADLs de la littérature pouvant correspondre à nos besoins.

2.1.2 Fractal et Fractal ADL

Fractal est un modèle de composant ouvert qui peut être implanté dans différents langages de programmation. Il a été utilisé pour construire différents types de systèmes : noyaux de systèmes d'exploitation, serveurs d'application, applications multimédia, etc.

Le développement de Fractal a été motivé par l'idée que les modèles de composant et ADLs existants ne fournissaient qu'un support limité pour les mécanismes d'extension et d'adaptation, comme en témoigne de récents travaux sur les composants aspects [DEMS02, POM03, PAG03].

Le modèle de composant Fractal introduit la notion de composant doté d'un ensemble d'interfaces de contrôle. En d'autres termes, les composants Fractal sont réflexifs, dans le sens où leur exécution et leur structure interne peuvent être explicitée, et contrôlée par des interfaces bien définies.

La première version de Fractal a été distribuée en 2002. Une deuxième version a été produite en 2003. Cette dernière n'a jamais été modifiée.

Composants Le modèle de composant Fractal [BCS03] est un modèle de composant généraliste qui tend à implanter, déployer et gérer (*i.e.* surveiller, contrôler et reconfigurer dynamiquement) des systèmes complexes, et plus particulièrement les systèmes d'exploitation et les intergiciels. On y trouve donc :

- *Des composants composites* (contenant des sous-composants), dans le but d'avoir une vision uniforme des applications à différents niveaux d'abstraction ;
- *Des composants partagés* (sous-composants appartenant à différents composants), dans le but de modéliser les ressources et les ressources partagées ;
- *Une capacité d'introspection*, pour surveiller et contrôler l'exécution du système ;
- *Des capacités de reconfiguration*, pour déployer et dynamiquement reconfigurer le système.

Un composant Fractal est une entité d'exécution encapsulée, a une identité distincte, et propose une ou plusieurs interfaces. Une *interface* est un point d'accès au composant (semblable aux «ports» dans un autre modèle de composant, comme AADL ou UML), qui implante un *type d'interface*. Il s'agit d'un type spécifiant les opérations supportées par le point d'accès.

On identifie deux types d'interfaces :

- Les interfaces serveurs, qui correspondent aux points d'accès acceptant les requêtes d'invocation d'opération entrantes ;
- Les interfaces clientes, qui correspondent aux points d'accès émettant ces mêmes requêtes.

Un composant Fractal, comme le montre la figure 3, est composé d'une *membrane*, qui supporte les interfaces pour introspecter et reconfigurer ses propres attributs, et d'un *contenu*, qui correspond à un ensemble fini d'autres composants (des *sous-composants*).

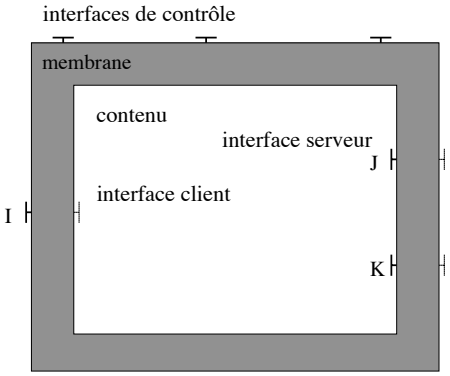


FIGURE 3 – Un composant Fractal

La membrane d'un composant peut avoir des interfaces internes et externes : les interfaces externes sont accessibles depuis l'extérieur du composant, tandis que les interfaces internes sont accessibles uniquement par les sous-composants propres au composant.

Connecteurs Le modèle Fractal fournit deux mécanismes pour définir l'architecture d'une application : la hiérarchie de composants et les liaisons. La communication entre les composants Fractal est possible uniquement si leurs interfaces sont liées. Fractal supporte à la fois les liaisons primitives (simples) et les liaisons composites.

- Une *liaison primitive* est une liaison entre une interface cliente et une interface serveur au sein du même espace d'adressage (qui peut être vu comme un composant). L'invocation de primitives par l'interface cliente doit être acceptée par l'interface serveur. Ces liaisons peuvent être implantées par des pointeurs ou des références (comme en Java).
- Une *liaison composite* correspond à un chemin de communication entre deux interfaces de composants arbitraires.

Ces liaisons sont construites à partir d'un ensemble de primitives de liaisons et de composants de liaisons : souches, squelettes, adaptateurs, etc. Une liaison est un composant Fracal à part entière.

Configuration Le FDF (*Fractal Deployment Framework*) permet de gérer les composants de l'application, ainsi que de les déployer. Il exécute un ensemble d'instructions spécifiées pour le déploiement, selon quatre types de procédure : *install, start, stop, et uninstall*. Ces procédures permettent de configurer différentes propriétés (comme les ports des machines hébergeant l'application, l'adresse des serveurs, ou encore les dépendances entre composants) sur les composants présentés précédemment.

Standardisation Fractal est supervisé par ObjectWeb, un consortium pour le logiciel libre créé en 1999 par Bull, France Telecom et l'INRIA. Si Fractal est utilisé par de nombreux industriels, il n'existe cependant pas de standard le définissant.

Capacité d'analyse Fractal ADL se voulant un langage de description d'architecture général, il ne permet pas de disposer d'informations suffisamment précises pour procéder à des analyses sur le système. Il se focalise en effet sur les descriptions hiérarchiques entre composants, ainsi que sur leurs interfaces. C'est pourquoi il existe des démarches comme [BABC+09] qui se basent sur Fractal ADL pour faire de l'analyse comportementale via des annotations supplémentaires.

Discussion Fractal ADL propose différents mécanismes permettant de gérer l'intégration puis le déploiement de différents composants d'une application. Cependant, Fractal ADL ne vise pas spécifiquement le domaine du temps-réel et de l'embarqué. Par ailleurs, les informations relatives au comportement de l'application ne sont pas non plus disponibles.

2.1.3 UML/MARTE

UML (pour *Unified Modeling Language*) est une spécification de l'OMG (*Object Management Group*), et a pour but de fournir un langage de modélisation standard. Il cherche à unifier les précédentes approches de langage de modélisation, comme Booch, OMT ou encore Jacobson's Use Cases, et couvre toutes les phases de développement logiciel, depuis les analyses en amont jusqu'à l'implantation. Différents aspects sont capturés dans différentes vues.

Depuis sa première spécification en 1995, il a été constamment raffiné et étendu. Une révision majeure est la version 2.0 qui améliore la clarté d'une spécification par un nouveau langage qui est utilisé pour définir UML même (MOF2).

Composants Une classe peut explicitement déclarer les interfaces qu'elle requiert ou fournit par rapport à son environnement.

Une *interface fournie* spécifie les opérations et les signaux auxquels une instance typée par sa classe devra répondre. Le fait de proposer (ou exposer) une interface fournie est réalisée via la relation de Réalisation depuis la classe à l'interface ciblée. Une *interface requise* spécifie les opération et les signaux qu'une instance typée par sa classe peut invoquer.

Les classes peuvent également exposer (*i.e.* rendre accessibles) des interfaces au travers de points d'encapsulation dédiés : il s'agit du concept de *Ports*. Un port est un élément nommé (graphiquement représenté par une petite boîte en bordure de la classe propriétaire), qui expose des interfaces requises ou fournies.

Les ports sont utilisés comme des sources (ou cibles) explicites de connexion entre différentes instances. Ce faisant, il devient possible en faisant varier le comportement associé à la classe, de faire varier la façon dont le port doit réagir sur réception d'un événement.

La spécification UML introduit par ailleurs une notation pour représenter les parties internes d'une classe. Cette notation est nommée «diagramme des structures composites». Une classe peut donc être composée de différents éléments qui prennent le nom de «part». Une part est la matérialisation d'une propriété de la classe : un attribut ou la terminaison d'une association.

Connecteurs Une fois que les parties internes d'une classe sont décrites à l'aide du diagrame de structures composites, il est possible de les interconnecter à l'aide des *connecteurs*.

Un connecteur est exprimé entre deux parts, pour mettre en avant le fait que les instances qui vont jouer le rôle des parts à l'exécution peuvent potentiellement communiquer ou collaborer (via l'appel d'opérations par exemple). La notion de connecteur est utile pour spécifier dès l'étape de conception qu'un chemin de communication est susceptible d'exister entre deux composants.

La notion de «points de variations sémantiques» a été introduite afin de permettre de spécifier de façon précise, au travers de *profils UML*, la sémantique d'un modèle particulier. En effet, en l'état, UML possède une sémantique faible, liée au fait qu'on ne peut à la fois couvrir un large domaine d'application et être sémantiquement précis (puisqu'il faut alors définir des domaines de valeurs spécifiques par exemple).

Configuration La configuration est spécifiée au travers de différents diagrammes modélisant les interactions entre composants selon différents points de vue [Amb04] : le diagramme de déploiement (relatif au matériel implantant le système, et aux informations de déploiement du logiciel sur celui-ci), de communication (relatif aux flots de messages entre les objets du système) ou encore de package (relatif à l'organisation hiérarchique des éléments et à leurs dépendances).

Standardisation La standardisation d'UML et du profil MARTE est gérée par l'OMG (*Object Management Group*), un consortium international non commercial existant depuis 1989. L'OMG se concentre sur l'intégration de standards pour un large panel de technologies et de domaines, dont : le temps-réel, les systèmes embarqués, les intergiciels ; de façon encore plus générale, on trouve, entre autre : la finance, la santé, la robotique, ou encore l'aérospatial.

Éléments spécifiques Il existe différents profils UML dédiés au domaine des systèmes TR^{2}E. On note parmi eux le profil «UML Profile for Schedulability, Performance and Time Constraints» [BFW04].

Le profil MARTE [DTA$^+$08], pour «Modeling and Analysis of Real-Time Embedded systems», vient en remplacement de ce précédent profil, en abordant les problématiques liées à l'embarqué, proposant des étapes de validations et de vérifications supplémentaires. MARTE définit une interprétation temporisée de modèles UML, forçant une sémantique plutôt que de laisser libre cours à différents outils d'analyse. Ainsi, pour analyser des modèles UML ciblant le domaine des systèmes embarqués temps réels, les outils prenant en entrée des modèles respectant le profil MARTE n'auront pas de divergence d'interprétation.

La notion de temps au sein du profil UML/MARTE repose sur la définition d'un langage spécifique, nommé CCSL [Mal08], pour «Clock Constraint Specification Language». A l'inverse du profil MARTE en lui-même, CCSL n'est pas normatif.

Capacité d'analyse En passant par les profils UML spécifiques, il est possible de procéder à des analyses sur le système : en utilisant le profil MARTE par exemple, il est possible de disposer d'informations permettant de faire des analyses temporelles [Mal08].

Discussion UML est un ADL complet, qui propose des mécanismes pour aider à la transformation de modèle, à la spécialisation pour un domaine d'application particulier, et enfin à l'analyse via des outils tiers utilisant cette notation standard disposant d'une communauté active. Il permet de cibler les systèmes TR^{2}E au travers de profils spécifiques.

Il n'y a cependant pas nativement de spécification exécutable formelle unique, permettant d'établir sans ambiguïté des règles de transformation vers des formalismes permettant de mettre en œuvre des méthodes formelles. Il est cependant possible de définir le cadre de ces transformations en utilisant des profils spécifiques spécialisés, à la manière du langage AADL [2].

2.1.4 AADL

MetaH [LCV00] est un langage de description d'architecture textuel. Il existe une notation graphique sémantiquement équivalente. De nombreux outils existent pour traiter des modèles exprimés dans ce langage. Ce langage a été développé par Honeywell pour le compte de l'armée américaine. Il permet de décrire à la fois le matériel et le logiciel mis en œuvre.

Le langage AADL (pour «Architecture Analysis and Design Language») est un ADL basé sur la notation textuelle de MetaH. Il s'agit d'un langage standardisé, dont la dernière version a été produite

2. Dans la section suivante, nous fournissons une première définition du langage AADL, complétée dans le chapitre 3

en 2009 [Sub09] (venue en remplacement de la version antérieure datant de 2004 [Sub06]). Ce langage cible le domaine des systèmes embarqués et temps-réels, et plus particulièrement celui de l'avionique.

AADL a été développé en gardant en tête les problèmes d'analyse de système : il facilite de fait l'interopérabilité d'outils. En conséquence de quoi, le standard AADL définit différents types de syntaxe :

- Une syntaxe *textuelle*, semblable à un langage de programmation ;
- Une syntaxe *graphique*, pour faciliter (du point de vue utilisateur) la description d'architecture (mais cela reste moins expressif que la syntaxe textuelle) ;
- Une *représentation XML*, pour faciliter la création d'outils d'analyse.

Composants Un modèle AADL repose sur une hiérarchie de composants interconnectés. Chaque composant dispose d'*interfaces* liées à une ou plusieurs *implantations*.

Il y a trois types de composants :

- **Les composants logiciels** (ou éléments applicatifs). Y sont représentés :
  1. Les *threads* (équivalent à un processus léger sur un système d'exploitation) ;
  2. Les *groupes* de thread (définissant une hiérarchie) ;
  3. Les *processus* (définissant un espace mémoire pour un thread ; chaque processus est lié à au moins un thread) ;
  4. Les *sous-programmes* (spécifiant les fonctions et les méthodes ; modélise les flots de données à l'aide de paramètres en entrée (*in*) ou en sortie (*out*)) ;
  5. Les *groupes de sous-programme* (représentant une bibliothèque de sous-programmes, pouvant accéder à d'autres sous-programmes) ;
  6. Les *données* (échangées entre les composants).
- **Les composants d'exécution** (ou éléments matériels). On y trouve :
  1. Les *processeurs* (modélisant la liaison entre un micro-processeur et un ordonnanceur (impliquant la présence d'un OS minimal)) ;
  2. Les *mémoires* (éléments de stockage, comme la ROM ou la RAM) ;
  3. Les *périphériques* (considérés comme des boîtes noires avec des interfaces bien définies) ;
  4. Les *bus* (représentant un lien physique entre des composants comme les processeurs, les mémoires ou encore les périphériques ; ils couvrent des liens allant de bus locaux à des communications internet).
- **Les composants logiques**
  1. Les *abstraits* (composants pouvant être redéfinis plus tard dans le processus de développement, contenant ou pouvant être contenus par n'importe quel autre type de composant) ;
  2. Les *systèmes* (une combinaison groupée de n'importe quel composant présenté précédemment) ;
  3. Les *bus virtuels* (abstraction logique d'un bus pour représenter un protocole et des canaux de communication virtuels) ;
  4. Les *processeurs virtuels* (machine virtuelle, comme pour Java, ou ordonnanceur hiérarchique).

Le type d'un composant définit ses interfaces, tandis que l'*implantation* décrit sa structure interne. Une implantation donnée réfère toujours à une interface donnée.

Connecteurs Une interface de composants peut contenir différents éléments (nommés *features*). Une feature peut être :

- Un *port* (ou interface de communication). Un port peut être en entrée (*in*), en sortie (*out*), ou bidirectionnel (*inout*). Il existe différents types de ports :
  1. Les ports d'événements (pour des notifications, comme les événements POSIX.4 [pos04]). Ils peuvent changer le comportement du récepteur.
  2. Les ports de donnée (permettant d'échanger les données entre composants). Ils ne permettent pas de générer des notifications.
  3. Les ports d'événement-donnée (permettent de lier des données à des notifications, représentant ainsi habituellement un message : la notification indique alors qu'un message est en attente de traitement).
- Les *groupes* de ports ;
- Des informations indiquant si un composant requiert ou fournit des accès spécifiques (bus par exemple).

Configuration AADL s'appuie sur la notion de *modes* pour proposer différentes configurations d'un même système. Un mode AADL modélise un mode opérationnel sous la forme d'un ensemble alternatif de threads actifs, de connexions et de comportement dans les threads. Cela est réalisé à l'aide d'attributs spécifiques attachés à chaque composant. Les modes permettent de reconfigurer dynamiquement l'application : il est en effet possible de spécifier les conditions permettant de passer d'un mode à un autre.

Standardisation La standardisation d'AADL est gérée par la SAE International (*Society of Automotive Engineers*). Cette organisation a pour but de développer des standards depuis 1905. AADL est donc standardisé au niveau international.

Capacité d'analyse AADL est un langage qui se prête particulièrement bien à l'analyse : il est possible de disposer d'informations relatives :

- aux communications,
- au déploiement,
- aux temps,
- aux threads,
- à la mémoire,
- à l'ordonnancement,
- aux appels de programme.

La modélisation de ces propriétés est intégrée dans le standard même.

Discussion AADL est un langage qui permet de spécifier l'ensemble d'un système TR^2E, depuis le matériel jusqu'au logiciel. Comme il a été créé dans une optique d'analyse, la littérature regorge de passerelles vers des modèles formels (BIP [CRBS09], TLA+ [Rol09]), ou non formels (Cheddar [SP07]).

Cela montre qu'il été éprouvé en tant que notation pivot, en faisant un bon candidat comme base pour l'utilisation complémentaire de méthodes formelles.

2.1.5 LwCCM

CCM [OMG06b] (*CORBA Component Model*), de concert avec le D&C [OMG06c] (*Deployment and Configuration of Component-based Distributed Applications*), définit un modèle de composant et de

déploiement sur cible. Lightweight CCM (LwCCM) est un profil spécifique de CCM, à la manière de MARTE pour UML, dédié aux systèmes embarqués distribués.

LwCCM met en avant :

– La séparation des préoccupation (code technique, aspects fonctionnels) ;
– La composition et l'assemblage de composants pour construire l'application finale ;
– Le support d'un cycle de vie pour le composant.

Les deux premiers aspects sont réalisés à l'aide d'un «container» au travers duquel toutes les interactions avec le monde extérieur sont gérées. Un container fournit :

– Une isolation de l'environnement d'exécution ;
– Le support du cycle de vie d'un composant ;
– Une interface avec l'environnement d'exécution sous-jacent pour assurer la communication entre composants par exemple.

Composants La représentation interne et l'implantation d'un composant sont définies par un ensemble de propriétés nommées. Une extension du langage IDL [OMG97] fournit les constructions pour définir le type d'un composant et ses ports de communication. La figure 4 montre une vue conceptuelle de la structure d'un composant CCM. Il interagit via des interfaces requises (*réceptacles*) ou fournies (*facettes*). Il peut émettre (*source*) ou recevoir (*puits*) des événements. Il possède également un ensemble d'attributs.

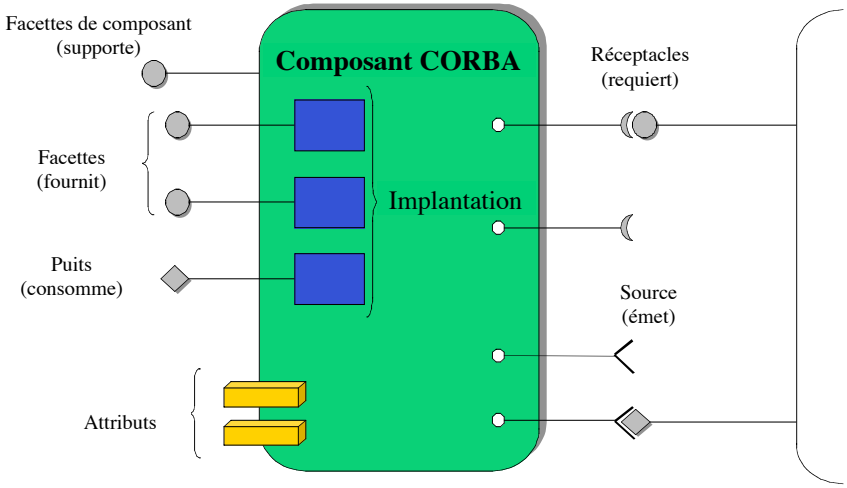


FIGURE 4 – Vue conceptuelle de la structure d'un composant CCM

Connecteurs La notion de connexion entre les composants est concrétisée par les concepts de *facettes* et de *réceptacles*.

Les facettes Un composant peut fournir plusieurs références d'objets CORBA, appelées facettes, qui sont capables de supporter des interfaces distinctes (*i.e.* ne partageant pas de relation d'héritage). Le composant possède une seule référence dont l'interface correspond à la définition du composant. Cette

référence supporte une interface nommée *interface équivalente du composant*. Elle définit les propriétés publiques du composant pour les clients.

L'interface équivalente permet aux clients de naviguer entre les facettes du composant et de se connecter aux ports de celui-ci.

Les réceptacles Ce sont les points de connexion pour communiquer avec d'autres facettes de composant. Réceptacles et facettes doivent avoir le même type pour communiquer. Un réceptacle est *multiple* s'il peut être connecté à différentes facettes ; il est dit *simple* dans le cas contraire. A titre de comparaison avec UML, les facettes et les réceptacles sont toujours typés par une unique interface (requise ou fournie, selon le cas de figure).

Configuration La configuration (ainsi que le déploiement) de composants LwCCM est basée sur le standard OMG «D&C» [OMG06c].

Le but de ce standard est d'unifier les modèles de déploiement pour les applications à base de composants. Il présente un PIM (*Platform Independent Model*) et une projection vers un PSM (*Platform Specific Model*) spécifique : CCM. Les notions de PIM et PSM sont décrites en détail dans [OMG03].

Le PIM est divisé en trois catégories de modèles :

1. Le modèle de composant, qui décrit les composants, leur implantation, ainsi que les exigences de configuration et de déploiement. L'implantation d'un composant peut être monolithique ou composite.
2. Le modèle cible qui décrit la plate-forme d'exécution ainsi que les ressources sur lesquelles les composants seront déployés.
3. Le modèle d'exécution qui permet d'appliquer le plan de déploiement sur une plate-forme, tout en vérifiant la disponibilité des ressources en accord avec les besoins des composants.

Standardisation De la même façon qu'UML, aussi bien CCM que le D&C sont des standards gérés par l'OMG. Il s'agit donc de standards de portée internationale, disposant d'une communauté active.

Capacité d'analyse De façon native, CCM ne permet pas de procéder à des analyses sur le système considéré, car cet ADL ne prend pas en compte des propriétés essentielles comme le temps-réel et ne propose pas de spécification du comportement du système (qui se limite à son implantation). Il existe cependant des extensions et des frameworks pour aller dans ce sens [HDD+03a, HDD+03b].

Discussion CCM, couplé au D&C, permet de gérer, modéliser des systèmes ainsi que d'en décider le déploiement. Cependant, CCM ne s'intéresse pas directement aux systèmes TR^{2}E, et, si des analyses sont possibles, des analyses formelles plus poussées ne le sont pas sans passer par des extensions spécifiques.

2.1.6 Wright

Wright [All97] est un ADL qui intègre nativement des mécanismes permettant de procéder à la vérification formelle du système spécifié. Il a été développé durant le projet ABLE du Carnegie Mellon University.

Il s'agit d'un ADL qui n'est pas dédié à la génération de code ou le déploiement, mais qui se focalise sur les interactions entre composants, permettant de faire des analyses en utilisant les *Communicating Sequential Processes* [Hoa78] («CSP»).

Composants La description d'un composant en Wright se scinde en deux parties : la partie *interface*, et la partie *calcul*. L'interface décrit un ensemble de ports, qui modélisent une interaction à laquelle le composant prend part. La description d'un port spécifie le protocole d'interaction qu'un composant utilise pour interagir via ce port. La partie calcul modélise le comportement global du composant en décrivant les relation entre les différents ports et leur implication dans les actions du composant.

Connecteurs Les connecteurs modélisent comment deux composants sont assemblés au travers de leurs ports. La description d'un connecteur en Wright est formée d'un ensemble de *rôles* et de *glues*.

Un connecteur possède deux rôles : un rôle «requis» et un rôle «fourni». Ces rôles définissent les protocoles d'interaction que les ports des composants liés au connecteur doivent respecter.

La glue d'un connecteur décrit les interactions entre les différents rôles de ce connecteur (séquence, envoi ou réception de messages, etc.).

Configuration L'architecture d'une application décrite en Wright est écrite comme une configuration : il s'agit d'un ensemble de composants et de connecteurs liés entre eux. La modélisation d'une configuration se fait en deux étapes :

1. La définition d'un ensemble de types de composants et de connecteurs utiles pour la configuration, ainsi que leurs contraintes. Ces informations donnent ce que l'on nomme un *style d'architecture*.
2. La définition des instances de composants et connecteurs, ainsi que la réalisation des interconnexions (via les connecteurs).

Standardisation Wright n'est pas un ADL standardisé.

Capacité d'analyse Wright permet de réaliser deux types de vérifications :

- La compatibilité des ports : Wright assure que les connexions entre composants sont correctes. Une connexion est correcte si chaque port interagissant dans la connexion respecte le rôle attendu par le connecteur.
- L'absence de blocages : Wright vérifie qu'aucun composant ne restera bloqué par des l'inaction d'autres composants.

Discussion En terme d'analyse, l'intérêt de Wright est qu'il intègre directement des mécanismes de vérification dans la spécification. Cependant, il ne prend pas en compte les problématiques de l'embarqué ou du temps réel : il manque les notions de temps ou encore de gestion ou de partage de ressources.

2.1.7 Synthèse

Nous avons présenté différents ADL, chacun lié à un modèle de composants particulier. Comme l'a montré [Med00], ils ont beaucoup de points communs. Le tableau 2.1 présente un résumé de notre étude.

	Composants composites	Standard	TR^{2}E	Analyse
Fractal	oui	INRIA - FT	-	via extensions
UML/MARTE	oui	International	oui (profil)	temps-réel
AADL	oui	International	oui	comportement, ordonnancement, dimensionnement
CCM	oui	International	oui (profil)	via extensions
Wright	oui	-	-	comportement, cohérence ports

TABLE 2.1 – Récapitulatif des caractéristiques des différents ADL étudiés

Pour faciliter l'intégration d'outils dans une chaîne de vérification sur le système à partir de la même notation pivot, il est essentiel que celle-ci soit standardisée : cela facilite l'interopérabilité des outils manipulant les modèles. De plus, il est nécessaire que l'ADL cible le domaine des systèmes TR^{2}E. Enfin, il est intéressant de cibler des ADLs qui ont déjà fait leurs preuves en terme d'analyse. Pour toutes ces raisons, notre choix se restreint de facto à AADL et UML/MARTE.

Si ces ADLs nous permettent de spécifier différents aspects d'un système TR^{2}E, il est nécessaire, pour le valider et le vérifier, de les utiliser conjointement avec des méthodes d'analyse. Nous présentons donc un aperçu des différentes méthodes formelles existantes.

2.2 Méthodes formelles

Les méthodes formelles permettent de spécifier rigoureusement un système à l'aide d'un modèle, et d'en décrire précisément les propriétés. Elles permettent également de les analyser précisément.

Il existe pléthore de méthodes formelles, chacune ayant ses points forts. Nous étudions les principales méthodes formelles de la littérature, en fonction de certaines caractéristiques exposées ci-après.

Cela nous permet de cibler les méthodes à utiliser selon le type d'analyse que l'on souhaite effectuer, et selon le moment où la méthode formelle intervient dans le cycle de développement.

2.2.1 Éléments caractéristiques

Capacité de raffinement La modélisation d'un système demande du temps et une certaine expertise. Cependant, pour procéder à des analyses très tôt sur un système, le modèle créé ne reflètera pas le système final : il est courant de raffiner les exigences ou les propriétés d'un système tout au long de son cycle de développement.

De ce fait, il est important de pouvoir ajouter à moindre coût des informations sur le modèle formel, tout en pouvant réutiliser les analyses effectuées sur un modèle moins restreint.

Composition Le développement d'un système met en œuvre différentes équipes travaillant séparément, dont les travaux sont ensuite assemblés pour construire le système final.

Pouvoir créer de façon indépendante (avec des interfaces bien définies), les modèles des différents composants du système permet de pouvoir les assembler quand on souhaite les analyser.

Méthode d'analyse D'après [CGP00], il existe quatre catégories de méthodes d'analyse : la simulation, le test, la vérification par inférence, le model-checking.

Faire de la simulation revient à faire des expérimentations sur un modèle du système. Cela implique d'avoir à disposition des informations sur le matériel en jeu, ce qui peut arriver tard dans le processus de développement de l'application répartie.

Nous ne considérerons pas les approches par simulation ou test puisque nous voulons utiliser des méthodes d'analyse de systèmes TR^2E qui soient basées sur la spécification de l'application (donc apparaissant en début ou milieu de cycle de développement).

Nous nous intéressons donc aux deux autres méthodes dites «formelles» : la vérification par inférence et le model-checking.

Elles permettent de réaliser des analyses de différents types :

- L'analyse «statique» permet de vérifier la cohérence d'une description en terme de typage et de composition d'interface ;
- L'analyse «dynamique» traite plus spécifiquement de l'évolution comportementale du système au cours du temps.

Automatisation Certaines méthodes formelles offrent la possibilité de procéder à des analyses de façon automatique (par model-checking par exemple) ; d'autres nécessitent l'intervention d'experts.

Dans l'optique de la réalisation d'un guide de sélection des méthodes formelles dans un processus de développement, il est utile de comparer ces méthodes sur cette caractéristique, puisqu'elle aura un impact fort sur le coût de mise en œuvre.

Types de propriétés Les méthodes d'analyse ciblent des propriétés différentes. De façon non exhaustive, s'y trouvent :

- des analyses d'ordonnancement,
- des analyses comportementales,
- des analyses temporelles (plus générales que les analyses d'ordonnancement),
- des analyses de dimensionnement de ressources.

Insertion dans le cycle de développement Enfin, il est important de pouvoir statuer sur la place que l'utilisation d'une méthode formelle peut occuper dans un processus de développement :

- En amont : ces méthodes sont utiles pour clarifier les exigences et les contraintes, pour spécifier l'architecture du système ;
- Tout au long du processus : ces méthodes seront utiles pour s'assurer de la non régression du système au fur et à mesure de son développement (typiquement en permettant facilement d'intégrer de nouvelles informations dans le modèle formel) ;
- En aval : elles permettent de s'assurer que l'on respecte bien les contraintes émises en amont du processus.

Nous étudions maintenant différentes méthodes formelles et les évaluons en fonction des critères précédents.

2.2.2 Approches par preuve de théorèmes

Nous abordons dans cette section les approches par preuve de théorèmes.

Méthode d'analyse Comme l'indique le titre de cette section, la méthode mise en œuvre ici est celle de la résolution de preuves.

Les théorèmes à prouver sont exprimées à l'aide d'éléments de logique (souvent du premier ordre). En combinant les différents prédicats exprimés dans le formalisme considéré, l'utilisateur doit alors être en mesure de prouver des expressions qui garantissent le respect de propriétés dans le système.

L'utilisateur écrit donc une *spécification formelle*, modélisant le système ainsi que ses contraintes. C'est cette spécification qui sera analysée par les outils.

Automatisation La résolution de preuves est un problème NP-complet [Coo71] qu'il n'est pas possible d'automatiser complètement. C'est pourquoi dans le meilleur des cas, l'outil d'analyse utilisé, un *prouveur*, est semi-automatique : l'avantage par rapport à la preuve manuelle est que le prouveur semi-automatique dispose de bibliothèques de théorèmes, de lemmes ou d'axiomes qui peuvent être intégrés à la résolution de la preuve en cours. Il propose aussi des méthodes de réduction de prédicats pour arriver au but, comme la réduction *one-point rule* qui permet de traiter les quantificateurs dans les prédicats.

Cela reste de l'aide à la preuve, car si les preuves triviales sont résolues automatiquement, il est nécessaire de guider le prouveur tout au long de la résolution des plus complexes : quels théorèmes, axiomes ou lemmes doit-il utiliser, quelles techniques de réduction doit-il appliquer, dans quel ordre (en effet, l'application d'algorithmes de réduction dans un certain ordre peut «bloquer» le prouveur, alors qu'un autre ordre terminera la preuve) ?

Ce sont des approches qui nécessitent une expertise pour être utilisées à leur plein potentiel [Sip96, BCP88].

Capacité de raffinement La modification d'un élément existant de la spécification du système implique qu'il faille :

1. Prouver que le nouvel élément ne présente pas d'erreur ;
2. Prouver qu'il s'intègre bien dans la spécification (cohérence) ;
3. Prouver que tous les éléments de preuve utilisant (dépendant de) ce nouvel élément (modifié) restent cohérents et corrects.

L'ajout d'éléments dans la spécification ne remet pas en cause les résultats déjà obtenus.

Composition Il est tout à fait possible, avec ce type d'approches, de spécifier dans des fichiers séparés les éléments du système. Chaque fichier peut contenir les preuves relatives à chaque élément.

Ces éléments peuvent être regroupés pour produire une spécification globale, où il sera cependant nécessaire de prouver que les éléments ajoutés n'ont pas d'impacts fautifs sur d'autres. En outre, les contraintes peuvent être exprimées au niveau global de la spécification, puis analysées.

Types de propriétés Les approches par preuves de théorèmes permettent d'analyser différents types de propriétés sur une spécification donnée :

- Vérification de type,
- Validité du domaine d'application de fonctions ou d'opérations,
- Cohérence des interactions entre composants (typage, paramètres),
- Animation selon des scenarii.

Le point fort de ces approches est qu'elles sont en mesure de traiter et d'analyser des systèmes susceptible de générer une infinité de situations (états) d'exécution.

Insertion dans le cycle de développement Bien que ces méthodes puissent être utilisées de façon ponctuelle à différents moments dans le processus de développement, elles donnent leur pleine mesure dans les premières heures de ce processus : elles permettent de clarifier les exigences, de fixer les interfaces, d'établir une architecture cohérente, voire même de servir de base d'implantation (via des raffinements et des transformations, comme la méthode B [Abr96]). Cela permet de réaliser un système «correct par construction».

Exemples d'utilisation

- **CICS** L'université d'Oxford et IBM ont collaboré en 1980 pour formaliser en Z une partie du CICS (*Customer Information Control System*), un système de gestion de transactions en ligne. Les mesures prises par IBM ont montré une amélioration de la qualité du produit, une réduction du nombre d'erreurs découvertes, et la détection en amont de problèmes.
  Z est un langage de spécification formel qui permet de faciliter l'écriture mathématique de systèmes complexes. Elle a été créée par J-R. Abrial et son équipe du «Programming Research Group», au laboratoire d'informatique de l'université d'Oxford (OUCL), dans les années 70. Elle repose sur la notion de «schémas». Ils représentent soit un type de composant (incluant ses contraintes), soit une opération modifiant l'espace d'état (spécifiant les entrées et les sorties de l'opération, ainsi que les états «avant» et «après» de l'espace d'état).
  Différents outils sont disponibles pour analyser des spécifications Z : Z-Eves [MS97], ProofPower [OCW06], HOL-Z [LEK+98] (repose sur Isabelle [Pau94]), ou encore FuZZ [Spi].
  Z est une notation formelle standardisée au niveau international [13502].
- **CIDS** En 1992, Praxis a fourni à l'Autorité d'aviation civile britannique le CIDS (*CCF Display Information System*), une partie d'un système de gestion de trafic aérien. VDM [BJ78] (*Vienna Development Method*) a été utilisé pour spécifier en amont les interfaces et les structures du système. La spécification VDM a ensuite été raffinée en module plus concrets, pour faciliter le passage à l'implantation respectant la spécification.
  VDM repose sur un langage formel, VDM-SL, qui a été créé dans les années 1970 au laboratoire d'IBM de Laxenburg-Vienne (Autriche). VDM est souvent comparée à Z [HHJ+93, Hay92], dont le système de spécification est assez semblable (modules contenant les contraintes exprimées, etc.). VDM permet de raffiner la spécification jusqu'à obtenir un modèle d'implantation respectant les contraintes exprimées.
  VDM-SL est une notation formelle standardisée au niveau international [PL92].
  Les outils disponibles pour VDM sont VDMTools [13608], ou encore SpecBox [FM89].
- **Metro Ligne 14 et navette-Roissy** La méthode B a été utilisée pour développer les parties critiques de la ligne 14 du métro parisien, ainsi que pour la navette menant à l'aéroport de Roissy.
  La méthode B a été créée par J-R. Abrial [Abr96], à partir des travaux de E.W. Dijkstra et C.A.R Hoare. B est une évolution du langage Z, adapté à une utilisation industrielle et utilisée tout au long du processus de développement, depuis la spécification du système jusqu'à son implantation (par raffinement de modèle). Le principal outil permettant de traiter une spécification B est l'Atelier-B [Ate08].

2.2.3 Approches par model-checking

Nous allons aborder dans cette section les méthodes dont l'analyse repose sur des approches par model-cheking.

Méthode d'analyse Dans ce type d'approche, particulièrement intéressante dans le cadre de l'analyse de systèmes concurrents, un système de transition représentant le système est donné en entrée à l'outil (model-checker).

Etant donnés M qui décrit un ensemble S d'états et les transitions entre les états, et une formule f d'une logique, l'algorithme de model-cheking recherche le sous-ensemble des états de S qui satisfont la propriété $S_f = \{s \in S \mid M, s \vDash f\}$

Les approches par model-checking soulevent cependant deux types de problèmes :

- Un premier problème provient des systèmes avec un nombre infini d'états. Dans ce cas, des abstractions finies sont nécessaires pour représenter le système.

– Un second problème provient de l'explosion combinatoire de la représentation des états, qui peuvent être très nombreux

Pour répondre à ces problèmes, différentes approches sont mises en œuvre. Il existe ainsi des techniques de compacité [TMPHK09] pour limiter l'impact de l'explosion combinatoire. Par ailleurs, afin d'éviter la représentation explicite de tous les états, des techniques de vérification à la volée ont été proposées [Hol96].

Automatisation Vérifier une propriété à l'aide du model-checking est un processus qui peut entièrement être automatisée.

Capacité de raffinement Selon le formalisme choisi, il peut être très facile d'ajouter des informations ponctuelles dans le modèle (comme la modification ou l'ajout d'une garde sur la transition d'un automate), mais l'ajout de composants complexes peut se révéler ardu : la modification d'un patron de communication entre deux automates peut entraîner des changements de modélisation de ceux-ci pour ne pas changer la sémantique du modèle ou pour éviter d'introduire des biais de modélisation (introduction d'un canal FIFO par exemple).

L'ensemble des propriétés à vérifier sur le système est exprimé une fois pour toutes à l'aide de formules de logique : il est donc possible, en raffinant le modèle, de savoir instantanément si les propriétés sont toujours vraies ou non, en appliquant de façon automatique l'algorithme de model-checking.

Composition S'il est possible de composer des éléments spécifiés séparément (modularité du modèle), il est extrêmement difficile de réutiliser toute propriété vérifiée sur un composant spécifique puisqu'il est alors nécessaire de s'assurer que les propriétés d'un composant sont toujours valides une fois celui-ci intégré dans le système global.

Types de propriétés Ces méthodes permettent de vérifier des propriétés relatives au :
– comportement (interblocage, famine, perte d'informations),
– respect d'invariants,
– dimensionnement (ressources générées à l'infini, ou au contraire non exploitées),
– temps (ordonnancement, échéances).

Insertion dans le cycle de développement Ces approches peuvent être utilisées très tôt dans le processus de développement, car même avec une quantité d'informations limitées, il est possible de vérifier un certain nombre de propriétés sur le système.

La possibilité d'ajouter des informations au modèle facilement dans la plupart des cas de figures (voir la discussion ci-après), ainsi que l'automatisation des analyses permet cependant d'utiliser ces méthodes tout au long du processus de développement.

Exemples d'utilisation

– **CSS** En 2002, Praxis a utilisé CSP (*Communicating Sequential Process*) pour modéliser et analyser le design d'une *smart-card* (environ 100 000 lignes de code), pour une autorité de certification. CSP a été décrit pour la première fois en 1978 par C. A. Hoare [Hoa85], et permet d'effectuer la description de systèmes en termes de processus (*i.e.* composants). Ils évoluent indépendamment, et interagissent uniquement via l'envoi et la réception de messages (pas d'appels de procédures par exemple). A noter que la partie «séquentielle» du nom de CSP est tombée en désuétude, puisque grâce à de constantes recherches sur CSP, les dernières versions de CSP permettent à la fois de

spécifier un processus comme étant séquentiel, ou comme la composition concurrente de processus plus primitifs.
Les relations entre les processus, et leur interactions avec leur environnement sont décrites à l'aide d'opérateurs algébriques (préfixe, choix déterministe, etc.).

– **BART** Il s'agit d'un système de train autour de la baie de San-Francisco [WKL04]. Pour résoudre les problèmes liés à l'atteinte des limites de la capacité du système, il a fallu travailler sur une nouvelle solution. La solution adoptée (moins coûteuse que de rajouter des voies), fut de réduire l'intervalle entre deux trains (passant de 2 mn 30 à 2 mn). Le contrôleur du système est considéré comme critique. La stratégie de contrôle de la vitesse a été vérifiée à l'aide des réseaux de Petri [Val03].
Un réseau de Petri est un graphe bipartite, dont les nœuds sont soit des places (cercles modélisant la présence ou l'absence de ressources), soit des transitions (rectangles modélisant la consommation ou la production de ressources).
Il existe de nombreuses variantes de réseaux de Petri, allant de la plus simple (P/T pour place/transition), aux plus complexes (temporels, stochastiques). Ces variantes permettent de mettre l'accent soit sur une analyse qualitative (interblocages, famine), soit une analyse quantitative (respect d'échéances, bornes des places).
Il existe un standard international en cours de réalisation sur les réseaux de Petri [BCvH$^+$03].
– **UTRA-LTE** Dans le cadre du projet UTRA-LTE (*Universal Terrestrial Radio Access - Long Term Evolution*) initié par la SAE, l'institut national de technologie de Corée a créé un système LTE de troisième génération (3GPP-LTE) en utilisant des spécifications SDL [THPT06]. Cela leur a permis d'évaluer leur technique de gestion de la mobilité.
SDL (*Specification and Description Language*) est un langage formel orienté objet, défini par le comité consultatif international télégraphique et téléphonique (CCITT, devenu ensuite ITU-T, pour *International Telecommunications Union - Telecommunication Standardization Sector*). Ce langage est dédié à la spécification de systèmes complexes, dirigés par la réception d'événements, temps-réel et interactifs, impliquant des entités concurrentes qui communiquent par envoi de signaux discrets.
Un système est spécifié comme un ensemble de machines à états finies (FSM pour *Finite State Machines*) étendues, indépendantes les unes des autres, et communiquant avec des signaux.
Un système SDL est décrit en spécifiant : des aspects structurels (composants systèmes, processus, blocs) ; des aspects liés aux communications (signaux éventuellement paramétrés, canaux) ; des aspects comportementaux (dans les processus) ; et enfin des aspects liés aux données (typage). Les composants SDL peuvent hériter les uns des autres.
Des outils comme Jade [PDCD$^+$00] ou PragmaDev [Pra08a] permettent de manipuler de tels modèles.

2.2.4 Synthèse

Nous avons présenté différentes méthodes de vérification, permettant d'analyser des propriétés sur des modèles exprimés dans un formalisme spécifique. Ces méthodes ont chacune leurs avantages et leurs inconvénients : automatisation de l'analyse, facilité de raffinement, gestion de l'explosion combinatoire des états (approches par preuve de théorème), ou encore production de contre-exemple lors de la détection d'erreurs (model-checking).

Chacune de ces méthodes peut être utilisée dans un processus de développement de système TR^{2}E : en fonction des propriétés à vérifier, ainsi que de la quantité d'informations disponibles, elles n'apparaîtront pas au même moment.

Le tableau 2.2 présente, en fonction d'un éventail de méthodes d'analyse, quelques outils disponibles.

Type d'analyse	Framework
Analyse sémantique	Cheddar [Sin07], MAST [CTR], SPARK [Pra08b], TRAIAN [VAS08]
Vérification de type	EiffelStudio [Eif08], FuZZ [Spi], Z/EVES [MS97]
Preuve de théorèmes	Atelier B [Ate08], Coq [CoQ], Z/EVES [MS97], PVS [SRI08]
Model Checking...	CHARON [Upe], CPN-AMI [LIP], FAST [Lab06], SMV [McM], SPIN [Hol07], SCADE [Est08], SPOT [DLP]
...temporel	CADP [VAS05], Kronos [DOTY02], TINA [LAA], UPPAAL [UPP]
...stochastique	GreatSPN [Gre], PRISM [PRI08], QPME [OPE07]

TABLE 2.2 – Succédané d'outils disponibles selon la méthode d'analyse ciblée

Si les méthodes formelles sont d'un réel apport pour vérifier les modèles représentant des systèmes, TR^{2}E de surcroît, elles doivent être intégrées dans un processus de développement pour répercuter de façon cohérente les résultats d'analyse. Nous allons donc présenter les différents types de processus existants, afin d'évaluer les plus à même d'accueillir les méthodes formelles.

2.3 Cycles de développement

Les cycles de développement imposent une structuration du développement logiciel : ils décrivent quelles tâches doivent être effectuées, dans quel ordre, voire même dans quel délai.

Nous allons présenter les principaux cycles existants (cycles séquentiels, cycles itératifs) et en analyser les principales étapes. Comme nous cherchons à pouvoir intégrer les résultats des analyses du système dans le processus de développement, nous nous intéressons particulièrement aux mécanismes prévus lors de la détection d'une erreur au cours du développement (échec du projet ou au contraire intégration rapide et transparente).

2.3.1 Cycle séquentiel

Le cycle dit «en cascade» est historiquement le premier processus de développement mis en place.

Les étapes qu'il doit respecter sont illustrées par la figure 5 :

1. Expression des exigences du systèmes : contraintes temporelles, matérielles, coût, etc. ;
2. Conception : langages de modélisation, utilisation de patrons de conception ;
3. Implantation : codage des différents éléments du système ;
4. Validation : processus d'évaluation du système à la fin du développement pour déterminer s'il satisfait les exigences exprimées ;
5. Maintenance : déploiement du système, gestion des erreurs et des mises à jour.

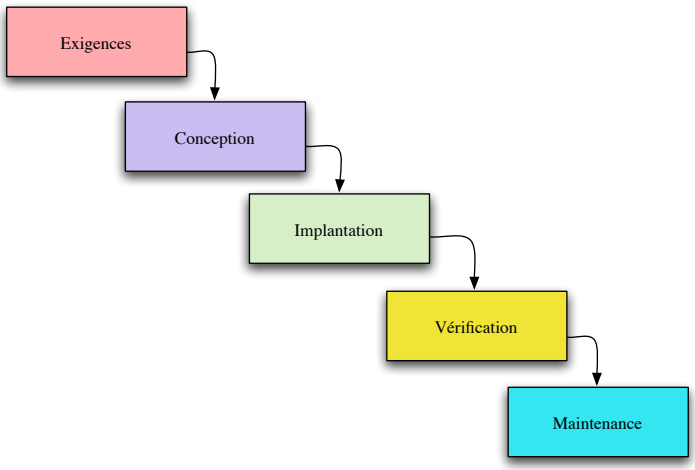


FIGURE 5 – Cycle en cascade

Ce cycle est opposé aux cycles itératifs et incrémentaux, qui permettent de réagir si une erreur est détectée au cours du processus (il n'y a pas de retours en arrière dans un cycle en cascade !). Pour contrebalancer ces effets, les cycles en *V* permettant des retours ont été introduits.

2.3.2 Cycles itératifs et incrémentaux

Prototypage Ce modèle de développement a été pensé sous l'hypothèse qu'il est souvent difficile de connaître toutes les exigences et contraintes au début d'un projet. Typiquement, les utilisateurs connaissent les objectifs qu'ils souhaitent atteindre avec leur système, mais ne connaissent pas les variations de données, ou encore les détails et les services du système. Le prototypage permet et offre une approche de développement qui mène aux résultats sans nécessiter toutes les informations dès le commencement.

Pour suivre ce processus, le développeur construit une version simplifiée du système ciblé, et le présente aux clients pour prendre en compte au plus tôt leurs retours. Ces retours, pris en compte dans le processus, permettent au développeur de les intégrer dans des étapes de raffinement du système.

Cycle en spirale Le cycle en spirale a été conçu pour prendre en compte les meilleurs aspects du processus en cascade et du prototypage. Il y ajoute une nouvelle étape : l'évaluation des risques.

A l'instar du prototypage, une version initiale du système est produite, puis modifiée à plusieurs reprises selon les retours des clients. Par contre, le développement de chaque version du système est attentivement produite en suivant les étapes d'un processus en cascade.

A chaque itération de la «spirale», une version plus complète du système est produite.

L'évaluation des risques est incluse dans chaque étape du processus de développement comme un moyen d'évaluer chaque version du système. Cela est fait dans l'optique de décider si le développement doit ou non continuer.

En effet, si le client estime qu'un risque, identifié, est trop important, alors le projet peut être abandonné. Par exemple, si un accroissement substantiel du coût, ou du temps de réalisation est détecté pendant une phase d'évaluation des risques, alors le client ou le développeur peuvent décider de stopper le projet.

La figure 6 illustre le modèle de cycle en spirale.

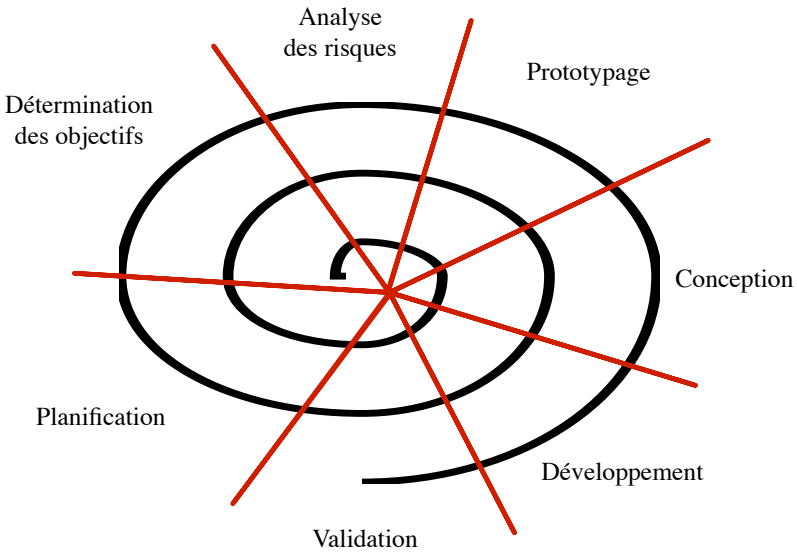


FIGURE 6 – Cycle en spirale

2.3.3 Discussion

Le processus en cascade est un cycle lent, qui requiert beaucoup d'informations dès le début du projet, et qui reste peu flexible. C'est pour faire face à ces problèmes que les cycles itératifs ont été développés.

Dans les cycles itératifs, le projet est divisé en petites étapes, permettant à l'équipe de développement de montrer au plus tôt des résultats, et d'obtenir des retours de la part des utilisateurs très rapidement.

Si les cycles itératifs font face aux problèmes soulevés par le processus en cascade, ils nécessitent cependant des contreparties pour être efficaces :

- Les utilisateurs finaux doivent être activement impliqués tout au long du projet. Si cette implication est bénéfique pour le projet, elle demande du temps à l'équipe et peut entraîner des retards de développement.
- Des exigences et des contraintes exprimées de façon informelles à chaque phase peuvent mener à des confusions. Un moyen de gérer ces requêtes doit être mis en place.
- Un cycle itératif peut mener un projet hors champs : puisque les retours utilisateur sont pris en compte à chaque phase, cela peut induire un accroissement de ses demandes. Alors qu'il voit le système prendre forme, le client peut réaliser que des caractéristiques supplémentaires pourraient améliorer grandement le résultat final.

2.4 Positionnement

L'intégration des méthodes formelles dans un processus de développement est un enjeu crucial pour la production de systèmes de confiance. Cette problématique d'intégration a été abordée dans le cadre de différents travaux comme [MH09, Rol09]. Ces derniers mettent en œuvre des mécanismes de transformation à partir d'ADLs comme UML (resp. AADL) vers les réseaux de Petri (resp. TLA+). Ils permettent cependant de n'analyser qu'un nombre restreint de propriétés sur le système, en mettant en œuvre un seul formalisme.

Les travaux de L. Hillah [MH09] ont été réalisés dans notre laboratoire d'accueil, et sous l'angle de vue suivant : des modèles en réseaux de Petri (IPN, pour *Instantiable Petri Nets*) sont construits à partir de diagrammes d'activité d'un modèle UML. Cela permet de vérifier l'absence d'interbloquage sur le système considéré. Cependant, le contexte de ces travaux est défini par les besoins de l'industriel Cofiroute pour le projet SafeSPOT (concernant les systèmes de transports intelligents). Ils ne s'attachent par exemple pas à prendre en compte les aspects temporels d'un système TR^{2}E. De plus, ces travaux ne prennent en compte qu'un seul formalisme de vérification, et ne permettent pas d'effectuer des analyses complémentaires tirant parti de différentes méthodes formelles. Enfin, l'intégration des méthodes formelles dans la méthodologie dégagée est faite dans un cycle en V, à des étapes bien précises du processus de développement.

Les travaux de J.F. Rolland [Rol09] mettent l'accent sur la vérification de propriétés quantitatives sur le système TR^{2}E : ordonnancement, taille des tampons, etc. Cependant, seul un sous-ensemble du langage AADL est utilisé pour la transformation vers TLA+, permettant de procéder à des analyses par *model-checking*. Enfin, ces travaux ne ciblent que des architectures mono-processeurs.

Nous nous différencions de ces travaux en cherchant un processus plus générique, qui permette d'inclure différents types de méthodes formelles pour l'analyse des systèmes TR^{2}E : une coordination de celles-ci permet en effet de vérifier des aspects complémentaires du système, à différentes étapes du processus de développement.

L'une des difficultés consiste donc à employer la bonne méthode au bon moment pendant le processus de développement.

Une abstraction du système considéré est nécessaire pour procéder à des analyses. Cela peut se faire en utilisant des ADLs, qui permettent d'en produire un modèle semi-formel. Ce dernier peut alors être dérivé vers un modèle formel (chapitre 4) qui sera analysé.

Nous avons étudié, au travers de leurs modèles de composants respectifs, différents ADLs de la littérature. L'ADL que nous devons choisir doit cibler les systèmes TR^{2}E et offrir des capacités d'analyse variées, nécessaires à l'intégration des méthodes formelles dans le processus de développement. En terme de modélisation, notre choix s'est restreint à AADL et UML/MARTE.

Dans la suite de ce mémoire, nous utilisons AADL. Plusieurs points motivent en effet ce choix technologique :

- La sémantique d'AADL est fixe, évitant les ambiguïtés lors de la transformation vers des modèles formels. A contrario, UML/MARTE, reposant sur UML, a une sémantique plus libre.
- AADL, utilisé depuis plus longtemps qu'UML/MARTE, permet de mieux maîtriser les patrons de spécification, en cernant leurs points cruciaux (AADL offre plus de recul qu'UML/MARTE).
- AADL est un sous-ensemble d'UML/MARTE, qui se veut plus généraliste qu'AADL. Ce dernier est défini dans la spécification UML/MARTE ; nos travaux portent donc sur un ensemble restreint de concepts et de propriétés.
  Travailler avec AADL permet d'avoir un retour plus rapide et plus efficace sur nos travaux, de pouvoir raffiner nos patrons de transformations et d'éprouver notre approche.
  Une piste de travaux futur sera donc d'étendre nos résultats à UML/MARTE.

Les travaux de D. de Niz [dN07] proposent par ailleurs une comparaison entre UML et AADL, étayant notre position : UML est un langage basé sur la notion de diagrammes, qui permettent de décrire la structure de logiciels. Cependant, comme les diagrammes sont spécifiés comme étant différentes vues d'un même système, UML peine à définir de façon complète les liens entre eux. Il est donc laissé à l'ingénieur le soin de s'assurer de la cohérence des différents diagrammes de son application. L'utilisation de méthodes comme RUP [Kru98] ou encore Fusion [CAB$^+$94] permettent d'aider l'ingénieur sur ces aspects du développement. Ils ne précisent cependant pas comment procéder à des analyses formelles

sur le système considéré.

AADL, héritier de Meta-H, a une philosophie proche des langages de programmation, que l'on peut traiter avec des méthodes classiques de compilation. AADL fut développé dans l'optique d'être un langage de description textuel de spécification d'architecture, et surtout d'avoir une sémantique cohérente (y compris en ayant une vue globale du système, là où UML lie difficilement la sémantique de différents diagrammes combinés) et une syntaxe fixe. La cohérence syntaxique des modèles AADL est facilement validée par des techniques de compilation.

Enfin, AADL permet une vérification automatisée de ses modèles, et cible nativement davantage le domaine des systèmes embarqués temps réels (champs d'attributs spécifiques définis).

Nous nous proposons donc par la suite de présenter comment transformer (ou exploiter) un modèle AADL semi-formel vers des formalismes spécifiques dédiés à l'analyse de propriétés variées. Cela sera réalisé à l'aide de guides, présentés au chapitre 3.

En terme de méthodes formelles, notre choix se porte sur deux méthodes distinctes, permettant d'éprouver notre approche sur des techniques très différentes :

- Les approches par *model-checking*, avec les réseaux de Petri, permettant de procéder à des analyses comportementales sur l'application décrite en AADL (chap. 4) ;
- Les approches par *preuve de théorèmes*, avec la notation Z, permettant de procéder à des analyses sur l'architecture de l'exécutif supportant l'application TR^{2}E (respect d'invariant, compositions d'interfaces possibles) (chap. 5).

Pour cela, nous intégrons l'utilisation de nos guides de transformations dans un processus de développement, afin de cadrer l'application des méthodes formelles. Les résultats des analyses sont également pris en compte rapidement dans le processus : nous privilégions un cycle itératif, de type cycle en spirale, qui est aujourd'hui plus proche des enjeux industriels comme le montre par exemple le projet Flex-eWare [fle09] (besoin d'un cycle souple, d'itérations successives de spécification, d'analyse ou encore d'implantation).

L'application des méthodes formelles peut être intégrée dans la phase d'évaluation des risques du cycle en spirale. A chaque itération, une phase d'analyse formelle peut facilement être ajoutée afin de valider ou non une étape de développement.

Chapitre 3

AADL : un langage pivot comme support pour les analyses formelles

Contents

Le langage de description d'architecture AADL, basé sur la description de composants, est standardisé par la SAE (*Society of Automotive Engineers*). Il permet de modéliser à la fois les aspects logiciels et matériels d'un système TR^{2}E. Lors de leur spécification, l'ingénieur définit avant tout les blocs d'interfaces de chaque composant. Leur implantation, séparée, permet de proposer différentes variations pour une même interface.

Nous étudions les différentes constructions et les différents mécanismes de ce langage, puis analysons leur utilité pour l'analyse formelle. Enfin, nous exposons les actions à effectuer pour mettre en place une approche IDV^2.

Les premiers éléments d'un ADL à étudier sont les composants AADL, qui sont des «boîtes» autour desquelles s'articule une spécification AADL. Les seconds sont les propriétés attachées à ces composants qui apportent différentes informations sur le système décrit (ressources, échéances, politiques de communication, etc.).

Avant d'aller plus avant, il est nécessaire de clarifier un point de sémantique : nous distinguons les «propriétés» au sens AADL du terme, des «propriétés» au sens des méthodes formelles.

Définition 3.1
- *Propriété AADL : élément caractéristique par sa valeur d'un composant AADL : priorité d'un thread, politique d'ordonnancement, taille mémoire, politique de déclenchement de thread, etc.*
  *Nous utiliserons le terme de **caractéristiques** ou d'**attributs** pour parler de ce type de propriété dans la suite de ce mémoire.*
- *Propriété formelle : but à atteindre lors d'une analyse formelle : détection d'interblocage, respect d'échéance, tampons bornés, etc.*
  *Nous utiliserons le terme de **propriété** ou de **but** pour parler de ce type de propriété dans la suite de ce mémoire.*

3.1 Eléments de langage : composants

Nous présentons maintenant les différents types de composants mis en jeu dans une spécification AADL, ainsi que la façon dont s'articulent leurs interactions. On différencie deux types de composants dans une spécification AADL : les composants «logiciels» et les composants «matériels».

Les premiers permettent de décrire les éléments applicatifs d'une architecture. Les seconds décrivent les éléments sur lesquels les éléments applicatifs seront déployés. Ces composants sont organisés de façon hiérarchique dans des super-composants dits «systèmes», qui peuvent eux-mêmes en contenir d'autres.

3.1.1 Composants logiciels

On distingue plusieurs types de composants logiciels :
- Les tâches légères (threads) : les threads modélisent une tâche concurrente ou un objet actif. Il s'agit d'un élément ordonnançable, pouvant s'exécuter de façon concurrente avec d'autre threads. Il s'exécute toujours dans l'espace virtuel d'un processus et se déclenche soit de façon périodique via une horloge, soit sur l'arrivée de données ou d'évènements sur leurs interfaces (*ports*). Ceux-ci sont gelés une fois l'exécution déclenchée : aucun événement ou donnée ne peut être pris en compte jusqu'au prochain déclenchement.
- Les groupes de threads : ils permettent de grouper logiquement les threads d'un système (*pool* de threads par exemple). Un groupe de threads est contenu dans un composant système et peuvent contenir, outre les threads, des composants de données (variables partagées par les threads), ou encore des composants représentant des sous-programmes (appelés par les threads du groupe, lors du déclenchement de leur activation).
- Les processus représentent un espace d'adressage virtuel et peuvent contenir des composants de données, des sous-programmes, ou encore des threads ou des groupes de threads.
- Les sous-programmes : un composant de ce type représente une séquence d'exécution appelée avec des paramètres et n'a pas d'état propre (pas de données statiques par exemple). Les sous-programmes, à la manière des threads, peuvent être organisés en groupes.

– Les données représentent des données statiques. Des composants peuvent partager la même ressource, leur exclusion mutuelle faisant alors partie des exigences.

La représentation graphique de chacun de ces composants est présentée figure 7.

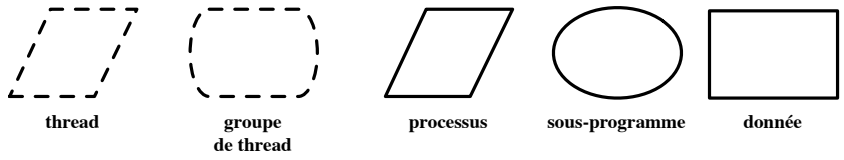


FIGURE 7 – Syntaxe graphique des différents composants logiciels d'une spécification AADL

3.1.2 Composants matériels

Les composants matériels d'une spécification AADL permettent de préciser les éléments sur lesquels l'application sera déployée. On distingue :

– Les processeurs, qui sont les unités matérielles permettant aux threads de s'exécuter. Ils peuvent contenir et accéder à de la mémoire, communiquer avec des périphériques ou avec d'autres processeurs via des bus.
– Les mémoires représentent des éléments permettant d'enregistrer du code ou des données binaires. Il peut aussi bien s'agir d'un composant de RAM que d'un composant matériel plus complexe, comme un disque. Leurs attributs typiques sont le nombre et la taille de leur espace d'adressage.
– Les bus sont les éléments permettant d'échanger des flots de contrôle ou de données entre les processeurs, les mémoires et les périphériques. Typiquement, ce sont des canaux de communication auxquels sont associés des protocoles. Ils peuvent se connecter directement entre eux afin de créer des réseaux complexes.
– Les périphériques (*devices*) représentent des éléments particuliers du matériel, des éléments extérieurs au système, ou encore des éléments interagissant avec l'extérieur depuis le système. Ces périphériques peuvent avoir leur propre processeur, mémoire et composants logiciels, qui peuvent être spécifiés séparément.

La syntaxe graphique de chacun de ces composants est présentée figure 8.

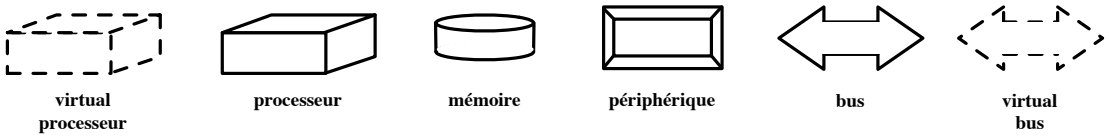


FIGURE 8 – Syntaxe graphique des différents composants matériels d'une spécification AADL

3.1.3 Hiérarchisation via les systèmes

Les composants «systèmes» sont organisés selon une hiérarchie représentant la structure globale du modèle. Il s'agit d'un ensemble de composants logiciels, matériels, et d'autres systèmes interagissant ensemble. La syntaxe graphique d'un composant système est présentée figure 9.

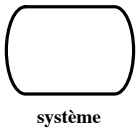


FIGURE 9 – Syntaxe graphique des composants systèmes d'une spécification AADL

3.1.4 Interfaces entre composants

Nous avons présenté les différents types de composants présents dans une spécification AADL. Ils interagissent via leurs éléments d'interfaces, que nous présentons maintenant.

Les caractéristiques (*features*) d'un composant spécifient la manière dont celui-ci interagit avec le reste du système, via :

- Des ports, qui sont les interfaces permettant aux composants d'échanger des événements ou des données. On les classe en trois catégories :
  - les ports de données (*data ports*)
  - les ports d'événements (*event ports*)
  - les ports de données et d'événements (*event data ports*)

  Les ports sont directionnels : un port de sortie sera connecté à un port d'entrée. On distingue les ports en entrée (*in ports*), en sortie (*out ports*) et bidirectionnels (*inout ports*).
- Des accès à des sous-programmes, modélisant les appels depuis des composants extérieurs d'une instance d'un sous-programme.
- Des paramètres, qui sont des valeurs de données qui sont passés en entrée (ou en sortie) des appels à des sous-programmes.
- Des accès à des données, spécifiant des accès à une variable partagée.
- Des accès à des bus qui représentent la connectivité physique des composants processeurs, mémoires, et autres composants via des bus.

Certaines caractéristiques d'un composant peuvent être requises (*requires*) ou fournies (*provides*).

La figure 10 décrit la syntaxe graphique de ces différents éléments d'interface.

FIGURE 10 – Syntaxe graphique des éléments d'interface d'une spécification AADL

La figure 11 montre un exemple de système AADL en utilisant la syntaxe graphique.

3.2 Eléments de langage : extension de la spécification

Le langage AADL s'intègre particulièrement bien dans un cycle de développement itératif puisqu'il offre différents mécanismes permettant d'étendre une spécification existante.

Tout d'abord, AADL prévoit dans le standard même, les mécanismes pour étendre le langage : il offre la possibilité de définir des annexes pour compléter la description d'éléments de différents types non pris en compte par ceux propres au cœur du langage. Ces annexes peuvent faire référence à des constructions

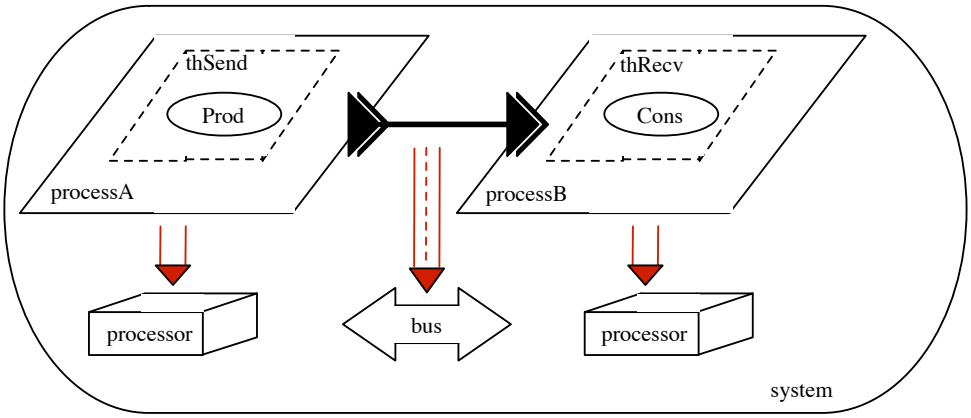


FIGURE 11 – Exemple de système AADL en utilisant la syntaxe graphique du langage : 2 threads communiquant via des port événements/données

de base du langage, mais nécessitent néanmoins de compléter un compilateur pouvant les prendre en compte (puisqu'il s'agit d'éléments de langage). Le processus de définition de nouvelles annexes est standardisé pour assurer la cohérence des nouvelles spécifications.

En parallèle de ces annexes, AADL permet d'étendre des ensembles de propriétés (au sens AADL). Il s'agit d'un équivalent des valeurs *tagguées* en UML : des attributs supplémentaires sont définis pour certains types d'éléments (par exemple, la «Période» d'un thread étend la propriété de «Temps» du langage).

Il est enfin possible de définir directement dans la spécification des composants qui en étendent d'autres, à l'aide du mot-clef **extends** (le mécanisme existait dans la première version du standard, via le mot-clef «refines», mais a été ensuite supprimé). De façon équivalente aux langages orientés objets comme C++ ou Java, il permet de définir un composant par héritage (héritage des types, des espaces de nommage, etc.).

3.3 Eléments de langage : attributs

Les attributs AADL, tels que présentés dans la deuxième version du standard [SAE08], se divisent en plusieurs catégories :

- Les attributs de déploiement : composants attachés à un processeur, une mémoire ;
- Les attributs liés aux éléments de communication : tailles de files de messages, politiques de gestion de file (pleine) ;
- Les attributs liés aux éléments de stockage : taille mémoire, tas, pile ;
- Les attributs de programmation : appels de procédures, priorité ;
- Les attributs liés aux threads : priorité, traitement des événements entrants, politique de diffusion de messages ;
- Les attributs temporels : dates d'échéances, délais de communication.

Nous allons explorer ces attributs, afin de proposer un guide de lecture des méthodes formelles susceptibles de les exploiter, en regard du type d'analyse à effectuer.

3.3.1 Attributs de déploiement

Ces attributs permettent de spécifier les contraintes de liaison qu'il peut y avoir entre un composant logiciel et un composant matériel (ou une catégorie de matériel).

Pour chaque attribut matériel considéré, il y a trois variations autour d'un mot clef : soit un tel mot clef, χ ; alors il y aura

→ `Allowed_`χ`_Binding_Class` : permet de spécifier sous forme d'ensemble les classes d'éléments susceptibles d'être liées au composant logiciel ;

→ `Allowed_`χ`_Binding` : ensemble de composants susceptibles d'être liés au composant logiciel ;

→ `Actual_`χ`_Binding` : désigne l'élément matériel effectivement lié au composant logiciel.

Les mots-clefs effectivement pris en compte dans le standard sont :

- `Processor` : si cet attribut est spécifié pour un thread, alors ce dernier peut se lier à n'importe quel élément de la liste. Si cet attribut n'est pas renseigné, alors un thread peut se lier à n'importe quel processeur de la spécification.
- `Memory` : indique quels sont les composants de données qui sont susceptibles d'être liés au composant concerné. En cas de non-spécification, n'importe quel élément de données de la spécification est un bon candidat.
- `Connection` : contraint, via des composants matériels (bus, processeur), les connexions entre composants. Si un port possède une telle caractéristique, alors toutes les connexions passant par ce port sont contraintes (protocole, etc.).
- `Subprogram_Call` : spécifie quelle instance de sous-programme caractérise un accès à un sous-programme.

Outre ces attributs, se trouvent aussi

- Ceux permettant d'indiquer si des composants sont déployés et liés sur le même matériel (regroupement logique des composants) ;
- Ceux qui permettant de spécifier la qualité de service attendue par les connecteurs ;
- Ceux indiquant quelles politiques sont mises en œuvre (déclenchement pour les processeurs, mode de lecture pour la mémoire, ordonnancement des tâches) ;
- Ceux exprimant des limites, comme le nombre de threads maximal géré par un processeur, ainsi que les bornes des priorités qu'on peut leur attacher.

Type d'analyse Les attributs relatifs au déploiement peuvent être exploités de différentes façons pour l'analyse du système :

- Validation de l'adéquation des composants déployés par rapport au matériel sous-jacent (respect des contraintes exprimées) ;
- Validation de la sécurité du système : flots de communications dans le système ;
- Informations pour architecturer un modèle formel (interactions entre composants logiciels et composants de ressources, partagées ou non).

3.3.2 Attributs liés aux communication

Ils permettent de caractériser une communication qui a lieu entre différents composants. On y distingue :

- Des politiques
  - liées à l'émission de message (`Fan_Out_Policy`) : lorsqu'un message est envoyé, il peut être diffusé à tout le monde (*broadcast*), choisir un destinataire à tour de rôle (*round robin*), ou encore envoyer le message à la demande.

  - liées à l'arrivée de nouveaux messages dans la file : en cas de file pleine (`Overflow_Handling_Protocol`) : il permet de supprimer de la file soit le message le plus récent, soit le plus ancien, ou encore lève une erreur.
  - liées à la paramétrisation de la connexion : taille de file (`Queue_Size`), ou encore le type de transmission (*push, pull*).
- Des informations quant aux délais et aux taux d'envoi ou de réception de messages :
  - d'émission (`Output_Rate`, `Input_Rate`) : nombre de messages envoyés ou reçus par seconde (ou par déclenchement de thread) ; (`Output_Time`, `Input_Time`) : moment où les messages sont envoyés ou reçus (avant le calcul du thread, après, pendant, à la date d'échéance).
  - de transmission (`Transmission_Time`) : temps passé par le message dans le bus.
- Des attributs caractérisant la latence du système (d'un bout à l'autre du système, pour les flots de données).

Type d'analyse Ces attributs offrent la possibilité d'analyser les communications du système : sûreté des communications en termes de perte de message, de temps d'acheminement, respect des échéances.

3.3.3 Attributs relatifs aux composants de stockage ou de transfert

Ils permettent de dimensionner le système, en indiquant, pour les threads :
- la taille du tas (`Source_Heap_Size`),
- la taille de la pile (`Source_Stack_Size`),
- la taille des données (`Source_Data_Size`).

Sont classés comme tels les attributs caractérisant les accès physiques à ces composants :
- Permissions d'accès en lecture et/ou écriture en mémoire ou sur le bus (`Acces_Right`) : lecture ou écriture seule, lecture et écriture, ou encore accès uniquement via des appels de méthodes spécifiques ;
- Temps d'accès physique à la mémoire en écriture (`Write_Time`) ;
- Adresses délimitant l'espace mémoire.

Type d'analyse Ces attributs permettent d'effectuer des analyses sur la sécurité des interactions entre composants, ainsi que sur le fait que les espaces mémoire prévus sont suffisants pour le système considéré (analyse des ressources)

3.3.4 Attributs de programmation

Ils offrent la possibilité de spécifier le comportement des threads sur la réception d'événements extérieurs, provoquant le déclenchement de leur exécution. Ce sont eux qui font le lien avec l'implantation des composants d'un système.

Il existe trois variations d'attributs autour d'un mot clef : il y aura, considérant un mot clef χ :

→ χ`_Entrypoint` : spécifie l'appel vers un sous-programme, qui doit être visible et accessible pour être appelé. Il est possible de préciser dans quel langage de programmation cette implantation a été réalisée.

→ χ`_Entrypoint_Call_Sequence` : permet de nommer une séquence de sous-programme qui sera exécutée une fois le thread déclenché.

→ χ`_Entrypoint_Source_Text` : permet de nommer un code source implantant le sous-programme ciblé.

Les mots-clefs concernés sont les suivants :
- `Activate` : tout ce qui concerne le thread lors de sa sélection pour un mode d'exécution.

- `Compute` : concerne l'exécution d'un thread une fois son déclenchement effectué. S'il s'agit de l'attribut d'un port, alors il peut être dissocié de l'attribut similaire du thread possédant le port (déclenchement sur réception d'un événement par exemple).
- `Deactivate` : programmes susceptibles d'être exécutés lors de la dé-sélection du thread d'un mode en cours.
- `Finalize` : code exécuté par un thread lorsque celui-ci termine son exécution.
- `Initialize` : ensemble des codes désignés pour s'exécuter lors de l'initialisation d'un thread dans le système.
- `Recover` : code exécuté par le thread dans le cas où celui-ci passe en mode de gestion d'erreur (*recovery*).

Type d'analyse Il est possible de faire des analyses sur le respect des interfaces, des paramètres, des invariants susceptibles d'être maintenus avant, pendant et après l'exécution d'un sous-programme.

3.3.5 Attributs liés aux threads

Ils ont vocation à spécifier des informations relatives aux composants actifs de la spécification : ils permettent de renseigner les politiques de déclenchement des threads, la concurrence, et le passage d'un mode AADL à un autre.

On y trouve :

- Les informations liées au déclenchement de threads : `Dispatch_Protocol (apériodique, sporadique, périodique, etc.)`, `Dispatch_Trigger` (spécifie la liste des points d'entrées susceptibles de déclencher le thread sur la réception d'un message).
- Les informations d'ordonnancement : `POSIX_Scheduling_Policy` : indique si le système est ordonnancé avec une politique FIFO, RR (préemptif), ou autre. En complément, viennent les attributs `Piority`, `Criticality`, `Urgency` et `Time_Slot`. Un attribut supplémentaire permet d'indiquer quelle politique appliquer pour l'exclusion mutuelle (`Concurrency_Control_Protocol`).
- Les informations liées aux politiques de traitement des messages reçus : `Dequeue_Protocol` (traite un, plusieurs ou tous les messages).
- Les informations liées à l'activation ou la désactivation d'un thread par rapport à un mode donné.

Type d'analyse Ces informations permettent d'analyser le comportement de l'application, les bornes des files de messages (tous les messages sont-ils traités ?), ou encore l'ordonnancement du système.

3.3.6 Attributs temporels

La dernière catégorie d'attributs à présenter est celle des attributs spécifiant les temps d'exécutions liés aux composants comme les threads, les périphériques ou encore l'exécutif sous-jacent.

Il existe deux variations d'attributs autour d'un mot clef : soit un tel mot clef, χ ; alors il y aura

$\rightarrow$ χ`_Deadline` : délai maximal autorisé pour effectuer l'opération χ ;

$\rightarrow$ χ`_Execution_Time` : temps estimé pour exécuter l'opération χ.

Les mots-clef autour desquels ont lieu ces variations :

- `Activate` : délais associés à l'activation d'un thread pour un mode d'exécution ;
- `Compute` : délais liés à l'exécution d'un thread une fois son déclenchement effectué ;
- `Deactivate` : programmes susceptibles d'être exécutés lors de la dé-sélection du thread d'un mode en cours ;
- `Finalize` : délai pour un thread terminant son exécution ;

- `Initialize` : délai maximal pour effectuer de l'initialisation d'un thread dans le système ;
- `Recover` : délai maximal entre la détection d'une erreur et le moment où le thread est en attente de déclenchement ;
- `Startup` : concerne l'initialisation des éléments matériels (processeurs, bus).

En sus, nous avons d'autres attributs :

- Ceux concernant le temps de chargement d'un thread dans le système ;
- `Period`, qui donne le délai minimal entre deux déclenchements d'un thread ;
- Ceux qui concernent la notion de temps au sein de la plate-forme d'exécution : le `Jitter` de l'horloge matérielle, le temps pour l'exécutif d'effectuer un changement de contexte pour les processus, le temps mis pour effectuer un changement de contexte pour les threads d'un même processus, etc.

Type d'analyse Ces attributs permettent d'effectuer de nombreuses analyses temporelles sur l'ensemble du système, voire même d'affiner des analyses comportementales en éliminant, à l'aide des contraintes qu'ils expriment, des comportements fautifs qui ne peuvent se produire du fait de l'ordonnanceur mis en place, comme par exemple les politiques FIFO_Within_Priorites ou PCP qui interdisent les interblocages dans le cas où il n'y a qu'un seul processeur.

3.3.7 Synthèse

Nous avons présenté les différentes catégories d'attributs AADL dans l'optique de les classer afin d'en faciliter l'analyse formelle. Nous allons indiquer, pour chacun de ces ensembles d'attributs, s'ils contiennent des informations permettant de caractériser :

- des propriétés structurelles,
- des propriétés qualitatives,
- des propriétés quantitatives.

Ceci est résumé dans le tableau 3.1.

Propriétés structurelles : elles sont liées à la structure du système, comme :

- Les connexions et la cohérence entre les interfaces des composants du système ;
- Les invariants à maintenir dans le système ;
- *Fault-tree analysis* (dépendance entre les composants quand l'un deux tombe en panne).

La plupart de ces propriétés doivent être établies très tôt dans le processus de développement, souvent à faible granularité. Elles peuvent être raffinées ou enrichies quand la conception du système évolue.

Propriétés qualitatives : elles sont relatives au comportement du système, c'est-à-dire quant à son ordonnancement, la détection de famine, d'interblocage, ou encore les liens de causalité entre composants.

Pour traiter de telles propriétés, le comportement du système doit être défini. Elles sont traditionnellement décrites plus tard dans le processus de développement, quand les informations relatives au comportement des composants deviennent accessibles.

Si on passe trop rapidement de l'étape de spécification à celle d'implantation (par exemple, en passant d'un diagramme de classes UML au codage), alors il est plus difficile d'établir ces propriétés (cela est complexe à partir d'un langage de programmation). A noter que des travaux vont en ce sens [HJ04, Hol00, HS02, Hol07, EKP+05]

Propriétés quantitatives : elles sont utilisées pour évaluer les performances du système ou pour évaluer son comportement selon des critères probabilistiques, ou de temps d'exécution. Pour établir ce type de propriétés, des informations sur les temps d'exécution sont requises.

Impact / Attributs AADL	Structurel	Qualitatif	Quantitatif
Déploiement	×		×
Communication	×	×	×
Programmation		×	×
Threads	×	×	×
Temps			×

TABLE 3.1 – Impact des attributs AADL sur les propriétés d'analyse

3.3.8 Guide de lecture des éléments AADL en vue d'analyses

Nous avons étudié les deux types d'éléments présents dans une spécification AADL : les composants et leurs attributs.

Nous allons maintenant présenter, pour chacun de ces type d'éléments, un récapitulatif des informations ou des facilités qu'ils apportent selon le type d'analyse visée.

Le tableau de la figure 3.2 le propose. Il présente deux niveaux de lecture.

Les colonnes catégorisent différents types d'analyses qu'il est possible d'effectuer sur un système.

Les lignes sont divisées en deux catégories :

– Les attributs AADL : pour chaque élément AADL, le tableau indique sur quel type d'analyse la valeur de l'attribut aura un impact ;
– Les méthodes d'analyse : le tableau indique quels types d'analyses sont susceptibles d'être effectués avec chacune d'entre elles.

	Cohérence d'interfaces	**Invariants Système**	**Fault-Tree Analysis**	**Ordonnancement**	**Vivacité**	**Causalité Interblocages**	**Analyse de performances**
Eléments AADL							
Déploiement	×		×			×	×
Temps		×		×			×
Communication	×					×	×
Programmation	×	×	×			×	
Threads		×	×	×	×	×	×
Méthodes d'analyse							
Simulation				×			
Analyse sémantique	×			×			×
Vérification de type	×						
Preuve de théorème	×	×	×		×	×	
Model Checking...	×	×		×	×	×	
...temporel							×
...stochastique							×

TABLE 3.2 – Croisement des attributs et de leur impact en terme d'analyse

Nous sommes maintenant en mesure de répondre aux questions suivantes :

– Quel type de validation ou de vérification souhaite-t-on effectuer ?
– Comment peut-on l'effectuer ?
– Quelles sont les informations qui lui sont utiles ?

Il est également possible de naviguer dans le tableau de différentes façons : pour quelles méthodes d'analyse tel attribut sera-t-il utile ? Que peut-on vérifier ou valider si l'on souhaite utiliser une technique particulière, et que l'on souhaite se focaliser sur certains attributs ?

Ce tableau est une brique essentielle dans un processus de vérification et de validation formelle autour d'une notation pivot.

3.4 AADL, un support pour une ingénierie dirigée par les Vérifications et les Validations

AADL est un langage de spécification qui se prête particulièrement bien à des analyses variées pour les systèmes TR^{2}E.

Dans une ingénierie classique dirigée par les modèles, l'ingénieur est en mesure, très rapidement et très facilement, de spécifier et d'annoter différentes vues de son système, séparant ses préoccupations. En procédant ainsi, lorsque viennent les étapes de validation et de vérification, l'ingénieur doit alors récupérer les informations disséminées dans toutes les vues de son système. Elles ne sont pas nécessairement orthogonales et peuvent être utiles pour effectuer plusieurs types d'analyses : bien souvent, il ne peut se contenter d'utiliser une vue particulière de son système, mais a besoin d'une composition de vues ou un sous-ensemble de celles-ci.

Se pose alors la question suivante : «Une fois le modèle utile produit, comment et pour quel(s) type(s) d'analyse(s) l'utiliser ? Avec quels outils ?» Nous avons commencé à répondre à cette question à la section précédente. Pour compléter ces résultats, nous allons expliquer comment il est possible de centrer l'ingénierie de tels systèmes non plus sur les modèles mêmes, mais sur les analyses que l'on souhaite effectuer.

Nous présentons maintenant comment AADL peut être utilisé comme un langage pivot pour une démarche IDV^2.

3.4.1 Application de méthodes formelles à partir d'AADL

AADL est un langage de description d'architecture standardisé qui a prouvé qu'il était utilisable pour procéder à différentes analyses sur les systèmes TR^{2}E : de nombreux travaux exploitant AADL dans cette optique ont été réalisés autour de BIP [CRBS09], de TLA+ [Rol09], d'UPPAAL [PC07], de LOTOS [HN07], de Lustre [JHR$^+$07], de Cheddar [Sin07] ou d'Archeopteryx [ABGM09]

3.4.2 Insertion dans un cycle de développement

Si une ingénierie dirigée par les modèles permet de spécifier précisément un système, l'exploitation de ces derniers pour l'analyse reste souvent un processus ad hoc, difficilement intégré aux processus de développement classiques que nous avons évoqué en section 2.4.

Pour répondre à la question posée en 1.2, «Quand utiliser les méthodes formelles ?», nous proposons l'approche suivante :

1. Prendre un cycle de développement classique de type itératif, couramment utilisé dans des projets industriels (comme le projet Flex-eWare [fle09]).
2. Insérer l'approche IDV^2 au niveau d'étapes stratégiques.

En effet, les cycles de développement présentés plus tôt, et couramment utilisés dans la littérature, ont en commun des étapes (ou phases) types. Ce qui diffère ces différents cycles, c'est l'enchaînement ces étapes (ou phases).

Nous privilégions les cycles itératifs qui évitent

- D'avoir à spécifier l'ensemble du système à un moment donné : il est possible d'ajouter des informations ou de contraindre certains aspects du système au cours du processus de développement ;
- De faire des retours en arrière coûteux en cas de modification des exigences du client.

Ces cycles itératifs ont en commun deux catégories d'étapes, qui, de par l'essence même du cycle de développement, reviennent régulièrement :

- Les phases de spécification : y sont précisées les contraintes du système, les informations de dimensionnement, etc. Chaque phase de spécification du cycle prend en compte les résultats (retours) des phases du cycle précédent.
- Les phases de validation : test de l'adéquation de l'implantation (ou du prototype si on considère un processus de développement de prototypage) par rapport à la spécification (de la phase de spécification).

A chaque étape de spécification ou d'intégration de contraintes, il faut enrichir le modèle dans la notation pivot, ici AADL.

L'approche est en effet plus directe si la spécification du cycle de développement est directement dans la notation pivot : l'intégration de nouvelles informations se fait de façon transparente. Dans le cas où ce n'est pas la notation pivot (permettant donc de dériver des modèles formels) qui est utilisée au cœur du processus de développement, il est alors nécessaire de pouvoir être en mesure de transformer la spécification du processus de développement vers cette notation pivot. Cela se fait en utilisant des techniques de transformation de modèle.

3.4.3 Articulation des étapes d'analyse

Nous avons présenté les points d'insertion de notre approche dans un processus de développement de systèmes TR^{2}E. Nous détaillons maintenant les différentes étapes mises en œuvre à chaque itération :

1. Définir un ensemble de vérifications ou de validations à effectuer : dans une approche itérative, on ne peut effectuer qu'un nombre restreint d'analyses pertinentes. A chaque itération successive du processus, cet ensemble pourra être étoffé avec des vérifications ou des validations supplémentaires. Il est caractérisé par les attributs de la spécification : selon le niveau de détail de celle-ci, de nouvelles analyses peuvent être effectuées.
2. Pour chaque type d'analyse de cet ensemble, sélectionner la (ou les) méthode(s) formelle(s) susceptible(s) d'exploiter les informations disponibles.
3. Utiliser des mécanismes de transformation pour passer de la notation pivot au formalisme dédié.
4. Procéder aux analyses, étudier les résultats.
5. Répercuter sur la spécification (en notation pivot) des éventuelles erreurs détectées.
6. Retourner à l'étape 3 jusqu'à obtention de résultats satisfaisants.

Ainsi, à chaque phase d'analyse du processus de développement, la spécification du système est corrigée et vérifiée : il n'est possible de passer à la phase suivante que si les analyses sont positives.

3.5 Synthèse

L'approche d'ingénierie dirigée par les vérification et les validations pour les systèmes TR^{2}E que nous proposons repose sur l'utilisation d'un langage de description d'architecture standardisé, exploité

pour procéder à des analyses formelles au sein d'un processus de développement adapté.

Les actions suivantes doivent être effectuées pour mettre en place cette approche IDV^2 :

- Sélectionner un processus de développement adapté aux systèmes TR^{2}E : un processus itératif comme le cycle en spirale est un bon choix, car il permet d'intégrer rapidement des retours d'analyses, validant les propriétés du système (et le corrigeant si nécessaire) jusqu'à satisfaction du client.
- Sélectionner une notation pivot (comme AADL) qui servira à spécifier le système dans le processus de développement : cette notation doit être standardisée pour pallier les problèmes d'interopérabilité des outils la manipulant. Elle doit par ailleurs capturer le plus d'informations liées au domaine TR^{2}E, afin d'augmenter la confiance placée dans le système en multipliant les analyses possibles.
- Proposer (et/ou utiliser) une classification des analyses possibles du système en fonction des informations disponibles dans une spécification en langage pivot. Cela permet de savoir quelles analyses effectuer, en utilisant le formalisme le plus adapté, à chaque itération du processus de développement.
- Proposer des patrons de transformation pour passer de la notation pivot à l'un des formalismes identifiés pour utiliser les techniques d'analyse ciblées. Cela permet de réduire le coût de la transformation en l'automatisant (une fois les patrons écrits). Des points d'extensions sont à prévoir pour intégrer de nouvelles propriétés ou de nouveaux formalismes.

A chaque itération du processus de développement a donc lieu une phase d'analyse. Cette phase d'analyse est elle-même un processus itératif où la spécification du système est analysée, raffinée puis corrigée jusqu'à obtenir des résultats validant ou vérifiant le système. Ces modifications (s'il y en a) sont alors répercutées dans le cycle principal de développement.

La suite de ce mémoire se focalise sur la validation comportementale de l'application ainsi que sur la vérification structurelle de l'intergiciel sous-jacent. Le chapitre 4 propose des guides pour extraire le comportement d'une application à partir de sa spécification AADL. Des patrons de transformations sont alors proposés pour spécifier ce comportement en réseaux de Petri. Le chapitre 5 présente l'exécutif AADL que nous considérons (PolyORB), et expose les patrons et mécanismes nécessaire à sa validation formelle en utilisant la notation Z. Les guides sont mis en œuvre et les patrons sont implantés dans la chaîne d'outil que nous proposons au chapitre 6 et qui est appliquée sur le cas d'étude du chapitre 7.

Chapitre 4

Guides et patrons de transformation depuis AADL vers les réseaux de Petri

Contents

Nous proposons dans ce chapitre des patrons de transformation d'AADL vers les réseaux de Petri pour faire de l'analyse comportementale, ainsi que des guides pour leur utilisation et leur composition. Comme le montre la figure 12, nous nous intéressons aux étapes de transformation de modèle et d'intégration des résultats de l'analyse formelle. Cette figure montre que pour analyser complètement le comportement d'une seule application décrite en AADL, nous utilisons deux formalismes traitant des propriétés qui leurs sont spécifiques.

Nous étudions les transformations d'AADL vers les réseaux de Petri colorés et les réseaux de Petri temporels à priorité. Nous verrons qu'ils peuvent être exploités de façon complémentaire [RKH09a, RKH09b], selon le type de propriétés (qualitatives ou quantitatives) à analyser.

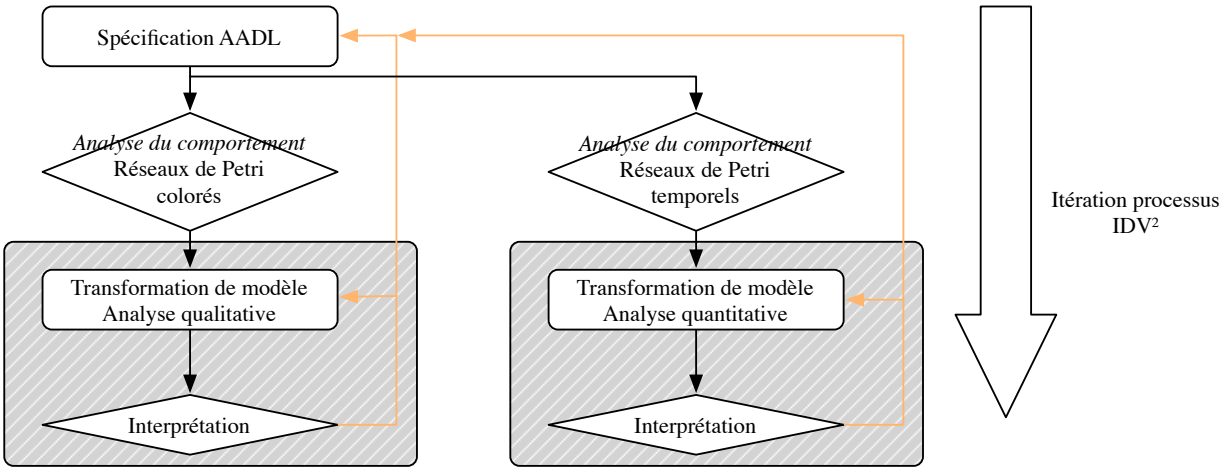


FIGURE 12 – Une itération du processus IDV^2 pour l'analyse comportementale en utilisant deux formalismes : les réseaux de Petri colorés et les réseaux de Petri temporels

En nous basant sur le standard, nous analysons les informations pouvant être apportées par les composants AADL ainsi que par leurs différents attributs, que nous exploiterons pour l'analyse comportementale en proposant des patrons ajustés.

Les deux formalismes choisis permettent d'effectuer des analyses comportementales : détection d'interblocage ou de famine, vérification des propriétés de vivacité ou encore d'équité. Cependant, chacun d'eux permet également de faire des analyses spécifiques, comme des analyses qualitatives pour les réseaux de Petri colorés (analyse des flots de messages pour tester des scenarii d'exécution), et quantitatives pour les réseaux de Petri temporels (ordonnancement, bornes de tampons).

Nous proposons une méthodologie d'analyse de la spécification AADL en vue de faire de la vérification comportementale (section 4.1). Nous en présenterons une mise en œuvre pour chacun des formalismes (définis dans la section 4.2) choisis dans la section 4.3.

4.1 Méthodologie pour la vérification comportementale

Cette section présente la méthodologie que nous appliquons à différentes variantes de réseaux de Petri pour effectuer de la vérification comportementale à partir de spécifications AADL. Cette méthodologie n'est pas dépendante du formalisme basé sur des automates états-transitions choisi.

4.1.1 Éléments d'AADL pour la vérification comportementale

Dans la section 3.3.8, nous avons effectué une classification des attributs AADL en fonction de leur thématique : nous détaillons ici ceux que nous avons utilisé pour établir des patrons et des règles de transformation depuis AADL vers les réseaux de Petri.

Les composants d'une spécification AADL qui concentrent le comportement de l'application sont les threads. Nous commençons donc par étudier le cycle de vie de ces threads pour en extraire des patrons reflétant la sémantique du standard.

La figure 13 présente l'automate du standard [Sub09], exhibant le cycle de vie d'une tâche.

Analyse de l'automate du standard

1. L'état de départ d'un thread est «arrêté» (*halted*).
2. Lorsqu'il est chargé, il passe par une phase d'initialisation. Celle-ci peut échouer (faute, échéance manquée), faisant retourner le thread dans l'état «arrêté».
3. Le thread est attaché à un *mode* du système. S'il n'appartient pas au mode initial du système, il attend que son mode soit activé.
4. Il passe dans un état d'attente de déclenchement de son exécution.
5. Si les conditions spécifiées par la fonction *Enabled* sont remplies, le thread est déclenché et passe dans un état où il effectue son exécution, ainsi que toutes les actions qui y sont associées.
6. Une fois son exécution terminée, il retourne dans l'état d'attente de déclenchement de son exécution.

Dans tous les autres cas, soit sur l'apparition d'une erreur (faute du thread, échéance manquée), soit sur la réception de commandes d'arrêt, le thread retourne dans l'état «arrêté».

Remarque : La notion de mode se résume à différentes combinaisons de configurations du système. Analyser chacun de ces modes de façon indépendante permet d'analyser la globalité du système, en réunissant ces analyses en fin de chaîne.

Nous ne considérerons pas directement la notion de modes dans la suite de ce manuscrit, car cela peut être ramené à des analyses menées séparément. Nous ne nous attacherons pas non plus au traitement des erreurs du système : il s'agit d'analyses complémentaire à celles que nous allons effectuer dans les chapitres suivants. Elles peuvent cependant s'intégrer dans le processus que nous proposons. Enfin, nous ne modéliserons pas dans le détail les bus et les protocoles de communications qui leur sont associés, car dans le cadre de cette thèse nous nous intéressons avant tout à la mise en œuvre de notre démarche IDV^2. La prise en compte de toutes ces informations sera l'objet de travaux futurs.

A partir des threads et de leurs interactions via les ports, nous sommes en mesure d'extraire les informations nécessaires à une analyse comportementale du système.

Nous présentons le sous-ensemble d'attributs des composants threads et des composants ports que nous avons pris en compte et qui permettront de raffiner et de paramétrer l'analyse. En effet, les threads et les sous-programmes concentrent le comportement de l'application finale : interactions via les ports, appels de sous-programmes, échanges de données et d'événements.

Attributs AADL des threads à considérer : Nous nous intéressons aux attributs des threads contenant de l'information sur leur comportement et sur les différentes échéances temporelles qu'ils doivent respecter, afin de pouvoir ultérieurement proposer des patrons de transformation vers des notations formelles comme les réseaux de Petri (colorés ou temporels).

- `Period` : spécifie un intervalle de temps entre deux déclenchements sucessifs d'un thread.
- `Dispatch_Protocol` : détermine les caractéristiques de déclenchement d'exécution d'un thread. Cet attribut est représenté dans l'automate de thread du standard par la fonction *Enabled* (t) décrite dans le tableau 4.1. On y note :
  - p un port en entrée ou un sous-programme accessible,
  - E l'ensemble des caractéristiques d'un thread permettant de déclencher son exécution,
  - t le temps écoulé depuis le dernier déclenchement. On y fait référence par sa dérivée δt, qui indique par sa valeur que du temps s'est effectivement écoulé ($\delta t = 1$) ou non ($\delta t = 0$).

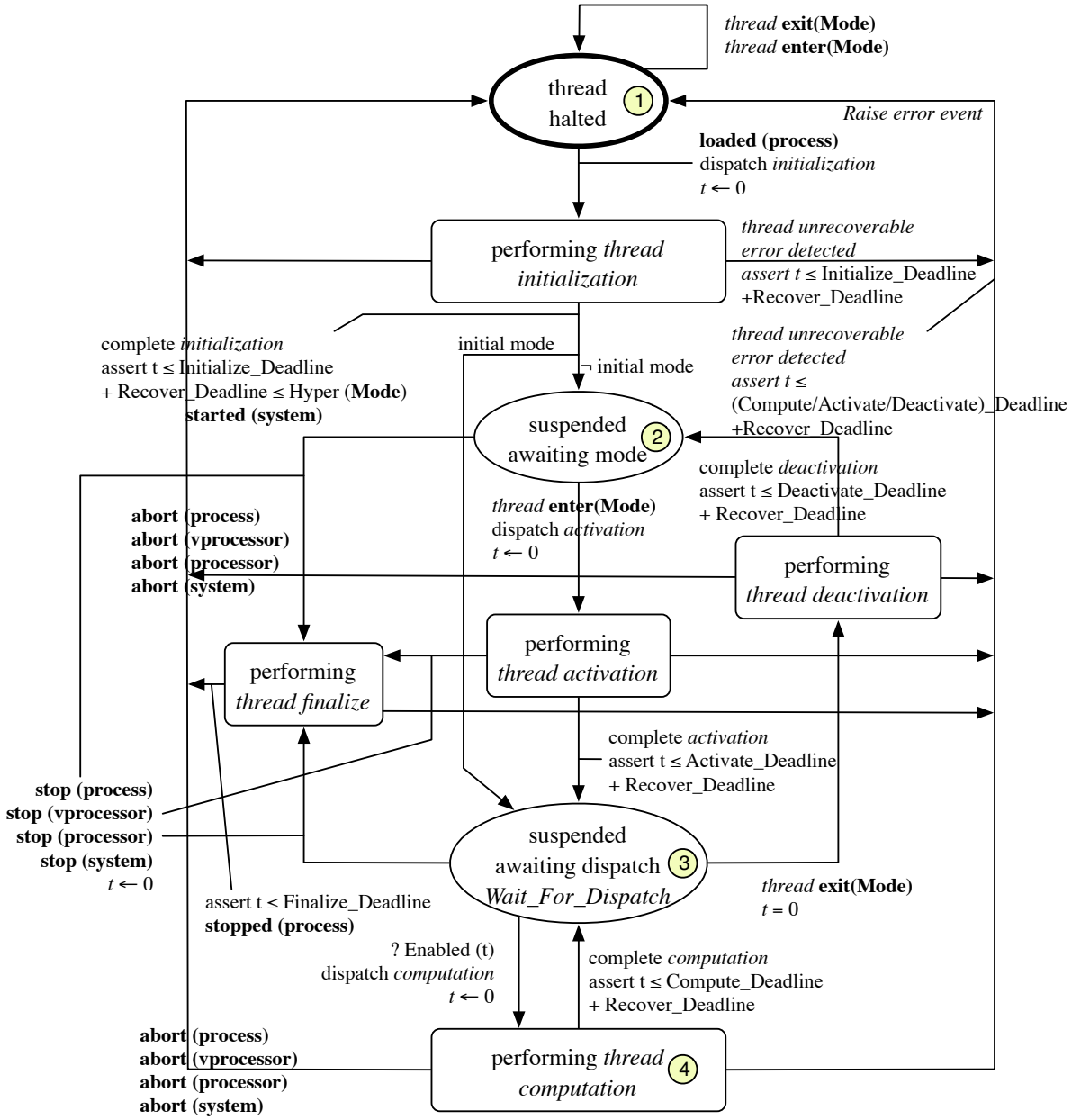


FIGURE 13 – Extrait du standard AADL : cycle de vie d'un thread

Dispatch_Protocol	Enabled	Wait_For_Dispatch Invariant
Periodique	$t = Period$	$t \leq Period \wedge \delta t = 1$
Aperiodique	$\exists p \in E : p \neq \emptyset$	$\forall p \in E : p = \emptyset$
Sporadique	$t \geq Period \wedge \exists p \in E : p \neq \emptyset$	$t < Period$ $\vee (t > Period \wedge \forall p \in E : p = \emptyset)$
Temporisé	$\exists p \in E : p \neq \emptyset \vee t = Period$	$\forall p \in E : p = \emptyset \wedge t < Period$
Hybride	$t = Period \wedge \exists p \in E : p \neq \emptyset$	$\forall t \in E : p = \emptyset \wedge t \leq Period$
Tâche de fond	*true*	$t = 0$

TABLE 4.1 – Règles de déclenchement d'un thread

- `Deadline` : spécifie le temps maximum autorisé entre le déclenchement de l'exécution d'un thread et l'instant où ce thread attend pour un nouveau déclenchement. Il n'y a pas de deadline pour les threads dont le protocole de déclenchement est «tâche de fond».
- `Priority` : spécifie la priorité d'un thread que doit prendre en compte un ordonnanceur.
- `Compute_Execution_Time` : spécifie le temps qu'un thread va mettre à retourner dans un état où il attend un nouveau déclenchement d'exécution.
- `Compute_Deadline` : spécifie l'intervalle de temps maximum autorisé pour qu'un thread exécute sa séquence d'action.
- `Actual_Processor_Binding` : indique si un processus est attaché à un processeur en particulier. Les threads hébergés par ce processus sont donc par voie de conséquence attachés à ce processeur.
- `Actual_Memory_Binding` : spécifie un ensemble de composants mémoire.

Attributs AADL des ports à considérer : Les ports permettant de modéliser les interactions entre les composants, et particulièrement entre les threads, nous nous intéressons à leurs attributs impactant les communications (et donc le comportement de l'application) :

- `Compute_Entrypoint` (pour les ports événements[données]) : spécifie le nom d'un sous-programme, appelé lors du déclenchement de l'exécution d'un thread.
- `Overflow_Handling_Protocol` : spécifie le comportement de l'exécutif sous-jacent lorsqu'un événement arrive et que la queue est pleine.
  * *DropOldest* supprime le plus vieil élément de la file et insère le nouveau.
  * *DropNewest* ignore le nouvel événement arrivant.
  * *Error* provoque une erreur dans le thread. La valeur par défaut est DropOldest.
- `Queue_Size` : spécifie la taille de la queue pour un port.
- Queue_Processing_Protocol : spécifie le protocole à utiliser pour traiter les éléments de la queue. La valeur par défaut est FIFO.
- `Dequeue_Protocol` : spécifie différentes options de retrait d'éléments de la queue :
  * *OneItem* : un élément parmi ceux qui sont gelés de la queue est retiré et mis à disposition lors du traitement des entrées à moins que la queue ne soit vide.
  * *AllItems* : tous les éléments gelés de la queue sont retirés et sont mis à disposition pour le traitement.
  * *Multiple_Items* : plusieurs éléments parmi ceux gelés peuvent être retirés séquentiellement de la queue, et mis à disposition.

Nous utilisons également la notion de composant qui existe en AADL (en terme de hiérarchie) :

Les composants logiciels : Les composants systèmes et les processus sont des composants apportant des informations de hiérarchie dans la spécifications.

Les composants systèmes sont utiles pour hiérarchiser la spécification AADL. Lors d'une analyse comportementale, ce ne sont pas des éléments impactant l'analyse. Les processus définissent un espace d'adressage pour les threads. Ils servent, au même titre que les composants systèmes, à structurer la spécification.

Nous pouvons donc conserver les informations hiérarchiques fournies au travers des composants systèmes et des composants processus sous la forme d'espace de nommage, pour la traçabilité des résultats de la vérification : cela permet de retrouver les éventuels composants fautifs.

Les composants matériels Les processeurs et la mémoire sont les principaux composants matériels à prendre en compte pour une analyse comportementale du système. En effet, dans les deux cas, il s'agit de ressources nécessaire au bon déroulement de l'application, partagées entre les différents threads : le processeur est utilisé pour l'exécution des threads, et la mémoire pour mettre en place des variables partagées.

Les éléments cruciaux pour l'analyse comportementale d'un système TR^{2}Evia sa spécification AADL sont donc les suivants :

– les composants sous-programmes,
– les composants threads,
– les composants processeurs,
– les composants de stockage de donnée.

Le tableau 4.2 indique l'impact des différentes structures d'une spécification AADL sur le comportement du système décrit.

4.1.2 Patrons de transformation

La première étape de notre démarche consiste à construire des patrons de transformation de ces éléments vers le formalisme de vérification ciblé.

Nous rappelons que les modes AADL peuvent être traités séparément. Nous appliquerons donc sur les différentes configurations du système la méthodologie que nous allons présenter. De ce fait, l'étape 2 de l'automate du standard est écartée. Il en va de même pour la gestion de erreurs.

Squelette de thread

Nous débutons par la spécification du patron du cycle de vie d'un thread.

Un thread démarre dans un état stable, passe par une phase d'initialisation, puis attend le déclenchement de son exécution. Une fois déclenché, il effectue un certain nombre d'actions, puis revient dans cet état d'attente de déclenchement.

Il y a donc trois principaux états dans ce patron : l'état initial, l'état d'attente, et l'état de calcul. Pour passer d'un état à l'autre, différentes transitions sont mises en place, comme le montre la figure 14.

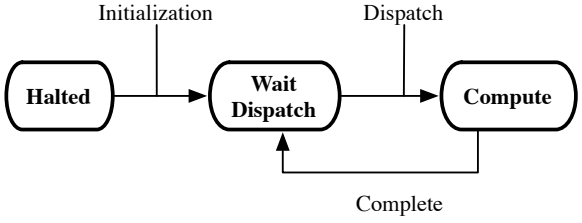


FIGURE 14 – Squelette d'un thread AADL

Elément de langage	**Rôle dans la spécification**	**Impact sur le comportement**	**Utilité pour l'analyse**
Systèmes	Hiérarchie	-	Espace de nommage Traçabilité
Processus	Hiérarchie	-	Espace de nommage Traçabilité
Groupes de threads	Hiérarchie	-	Espace de nommage Traçabilité
Threads	Reception Envoi Traitement messages	Source d'événements dans le système. Source potentielle d'interblocage	Valeurs d'attributs utiles pour ordonnancement, bornes tampons...
Groupes de sous-programes	Hiérarchie	-	Espace de nommage Traçabilité
Sous-programmes	Unité de calcul	Chemin d'exécution de threads	-
Données	Ressource	-	Typage de messages Analyse des flots de messages (dans le système, dans les séquences de sous-programmes)
Processeurs	Ressource d'exécution	Permet aux threads de s'exécuter	Analyse d'ordonnancement
Mémoires	Ressource	-	-
Bus	Canal de communication	-	Expression des flots d'information dans le système
Périphériques	Sources d'événements	Injection de messages	-
Ports	Interconnexion de composants	Taille tampon finie $\Rightarrow$ perte de message $\Rightarrow$ chemins d'exécution différents	Valeurs d'attributs : taille tampon, politique d'extraction de messages

TABLE 4.2 – Récapitulatif des éléments de langage AADL et analyse de leur impact comportemental

Déclenchement d'exécution

Afin de pouvoir procéder à des calculs, envoyer ou recevoir des données, un thread doit d'abord voir son exécution déclenchée. Cela dépend de la politique de déclenchement qui lui est attachée : il peut être périodique, sporadique ou apériodique.

Le patron suivant (figure 15) raffine la transition entre l'état «Wait_Dispatch» et «Compute» de la figure 14.

FIGURE 15 – Déclenchement d'exécution : plusieurs choix possibles, une interface

Les variantes de ce patron dépendent de l'attribut «Dispatch_Protocol» de la spécification AADL. Il peut prendre les valeurs définies par l'énumération «Supported_Dispatch_Protocol» (la sémantique est définie par le standard) :

- Périodique : tâches périodiques avec deadlines ;
- Sporadique : tâches à échéance molle dont le déclenchement de l'exécution survient sur la réception d'un événement. Il existe un délai minimal entre deux événements successifs ;
- Apériodique : tâches à échéance dure dont le déclenchement de l'exécution survient sur la réception d'un événement. Il n'y a pas de contraintes temporelles entre l'arrivée de deux événements successifs ;
- Temporisées (timed) : tâches dont le déclenchement de l'exécution survient au plus tard après un certain délai. Un événement externe peut cependant provoquer le déclenchement plus tôt ;
- Hybride : tâches dont le déclenchement de l'exécution est possible à la fois sous les conditions d'une tâche périodique et celles d'une tâche apériodique : l'arrivée d'une nouvelle période ou la réception d'un événement externe peut déclencher le thread ;
- Fond (background) : tâches dont le déclenchement de l'exécution survient une seule fois.

Suivant la valeur de cet attribut, le déclenchement d'un thread pourra être dû :

- uniquement à l'expiration d'une période ;
- à l'arrivée d'un événement ou d'un message ;
- à une combinaison des deux.

Réception - Calcul - Envoi

Une fois son exécution déclenchée, un thread est en mesure de recevoir des informations sous forme de données ou d'événements, d'agir en conséquence, puis d'envoyer de l'information, sous forme de données ou d'événements, à d'autres acteurs du système.

Pendant la phase de calcul, il peut appeler différentes séquences de sous-programmes, elles-même pouvant en contenir d'autres.

Nous proposons donc des règles permettant de traduire cet enchaînement de séquences d'appels dans le formalisme choisi. La figure 16 présente le patron de base utilisé pour la transformation.

Communication

Les mécanismes de communications permettent aux threads d'échanger de l'information, afin de procéder à des calculs ou tout simplement de déclencher leur exécution.

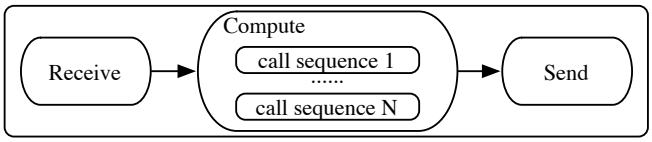


FIGURE 16 – Séquence de réception, traitement, et envoi : plusieurs séquences de traitement possibles. La réception et l'envoi sont facultatifs

Ces communications se font au travers des ports de données et/ou d'événements. Un port possède différents attributs (Queue_Processing_Protocol, Overflow_Handling_Protocol, QueueSize) ayant plusieurs valeurs possibles, provoquant des variantes dans les patrons (figure 17).

Un port peut transporter des données, des événements, ou encore une combinaisons des deux : *data* | *event* | *data event*.

Les ports étant directionnels, ils peuvent servir de points de réception et/ou d'envoi (*in* | *out* | *inout*).

Pour les ports de donnée, le standard édicte qu'il n'est pas possible d'avoir une file de données : si aucune nouvelle valeur n'est reçue, alors l'ancienne persiste ; si une nouvelle valeur arrive, l'ancienne est remplacée.

Le standard statue que les ports «event» ou «data event» peuvent avoir des files d'événements en attente. Par défaut, leur taille est fixée à 1, mais cela peut être modifié en spécifiant une valeur alternative pour l'attribut *Queue_Size* des ports.

Si un port disposant d'une file d'attente ne peut recevoir de nouveaux messages à cause d'une file pleine, alors l'attribut «Overflow_Policy» indique les actions effectuer.

Ainsi, le plus vieux message de la file peut être perdu («DropOldest»), ou encore le plus récent («DropNewest») ou enfin une erreur peut être levée («Error»).

Des règles doivent proposer de telles transformations dans le formalisme choisi.

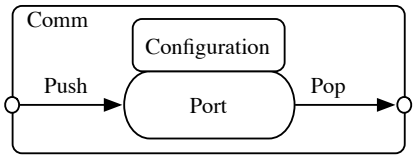


FIGURE 17 – Mécanisme de communication : deux interfaces, plusieurs variantes

4.1.3 Assemblage

Une fois les règles de transformations précisées pour chacun des points précédents, et après application de ces règles à la spécification d'une application, nous disposons de plusieurs boîtes à assembler.

Nous proposons des règles d'assemblage d'abord pour construire un thread dans sa globalité (squelette, déclenchement d'exécution, réception, calcul, envoi), puis un système complet.

Quand tous les threads du système sont assemblés, leurs interactions doivent être traduites, en liant entre eux les différents ports d'entrée ou de sortie des threads (multicast, communication point à point).

Les éléments matériels de l'application impactant le comportement de l'application, auxquels sont liés les threads, doivent être également retranscrits. Les ressources partagées (mémoire, processeur) font partie de ces éléments.

La figure 18 illustre comment doivent s'agencer les différents composants pour former un thread fonctionnel. La figure 19, elle, met en jeu les mécanismes de communication nécessaire à la construction complète du système.

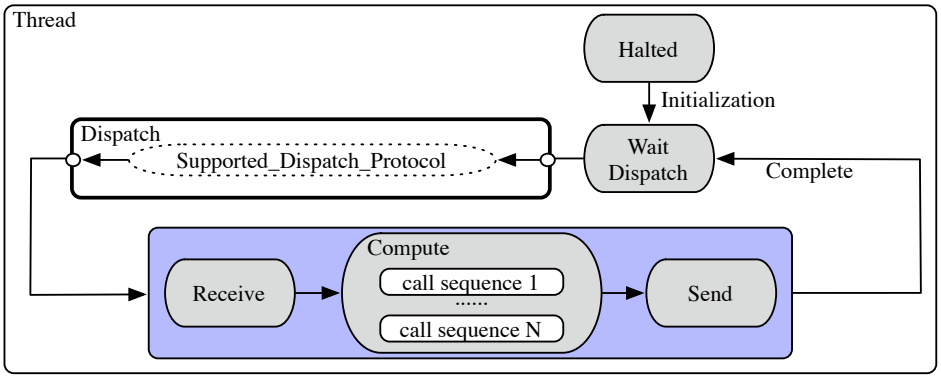


FIGURE 18 – Assemblage des composants pour former un thread complet

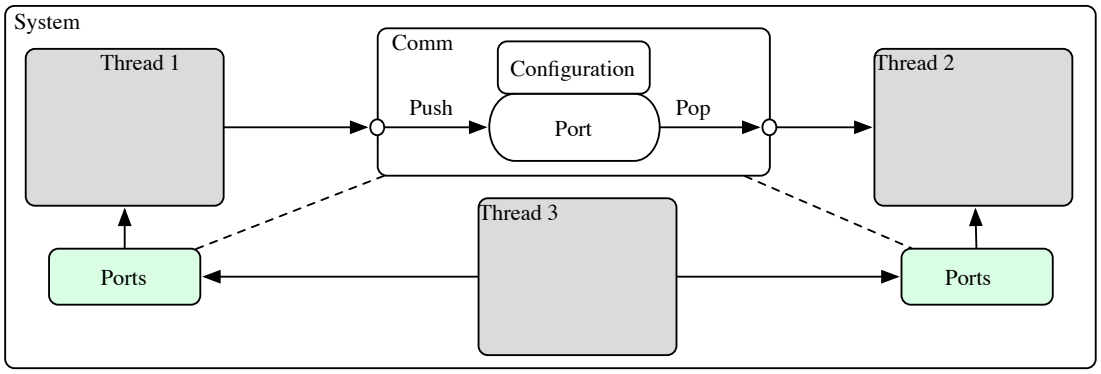


FIGURE 19 – Assemblage des threads pour former le système

4.1.4 Raffinement

Les règles précédemment proposées doivent pouvoir s'adapter en fonction des besoins de vérification de l'ingénieur. Le raffinement de tels patrons de transformation se fait en plusieurs étapes :

1. Propriété : une nouvelle propriété à vérifier est considérée, et les patrons actuels sont insuffisamment précis pour la vérifier.
2. Raffinement des patrons : afin d'effectuer ces vérifications, des informations sont ajoutées. Des observateurs dans les modèles, en fonction des propriétés, permettent de mettre cela en œuvre. Les règles spécifiant ces observateurs, et celles permettant de les ajouter dans le modèle formel doivent être proposées.

   Définition 4.1 *Un observateur est un mécanisme qui permet de savoir si des événements particuliers se sont produits dans le système, de façon non intrusive (i.e. ne modifie pas le comportement de l'application). Il s'agit de petits composants formel décrits dans le même formalisme que le modèle considéré.*

3. Analyse : les critères qui vont activer les observateurs sont spécifiés. Pendant la phase d'analyse, nous serons en mesure de savoir s'ils ont été déclenchés, par l'observation d'états incohérents, ou par la vérification d'invariants sur ces observateurs.

4.1.5 Traçabilité dans les modèles

En cas d'erreur, les causes ou les éléments concernés doivent être indiqués (et localisés) au développeur. Pour cela nous conservons la hiérarchie de la spécification AADL en la traduisant en espace de nommage pointé sur élément du modèle formel.

Si nous considérons un système «S», contenant un processus «P», contenant un thread «T», alors toutes les éléments formels ayant trait à ce thread seront préfixées par *S.P.T.*. De ce fait, en analysant les résultats produits par l'analyse, nous sommes en mesure de dire quels threads sont fautifs (interblocage, ordonnancement). Les éléments partagés par deux threads, comme les canaux de communications (via leurs ports), sont préfixé par le nom des deux threads en jeu : *S.P.T1.* et *S.P.T2.*.

4.1.6 Synthèse

L'analyse du standard AADL nous a permis d'établir sur quels éléments AADL il était nécessaire de se reposer pour procéder à des analyses comportementales.

Nous avons établi des patrons génériques sous forme de systèmes de transitions en présentant les points structurels clefs de chacun d'eux. Cela permet de les particulariser selon les propriétés que l'on souhaite analyser.

Ces patrons se focalisent sur les threads et leurs interactions, identifiés comme éléments comportementaux cruciaux dans une spécification AADL. Nous avons précisé les règles d'assemblage génériques de ces patrons pour construire un thread (élément central dans l'analyse comportemental) ainsi que pour construire un système complet (avec les threads et leurs interactions). Enfin, nous proposons des mécanismes permettant d'étendre les patrons existants dans le but de traiter de nouvelles propriétés sur le système.

Les réseaux de Petri, que nous avons utilisé pour procéder aux analyses comportementale dans le cadre de notre approche, permettent de procéder à des analyses très différentes et complémentaires :

- Les réseaux de Petri *Place - Transition* : les états sont représentés par des cercles contenant des jetons, les transitions par des rectangles. On passe d'un état à un autre via une transition qui «consomme» des jetons d'une place pour en «générer» de nouveaux dans une autre. On définit l'état global du système par le nombre et la localisation des jetons dans le modèle. Ce formalisme permet, entre autre, de vérifier l'absence d'interblocage dans le système.
- Les réseaux de Petri *Colorés* : les jetons contiennent de l'information (typage), et peuvent être combinés en tuples (types complexes). Des opérations de filtrage pour empêcher le franchissement de transition selon les informations contenues dans les jetons sont disponible. L'expressivité de ce formalisme est bien plus grande que pour les réseaux de Petri P/T. Ce formalisme permet, ou la vérification d'absence d'interblocage, de vérifier des propriété dépendant des informations des jetons (chemin d'un jeton dans le système, interblocage dû à la présence (ou l'absence) d'information dans un jeton).
- Les réseaux de Petri *Temporels* : ils héritent des réseaux de Petri P/T : les jetons marquent des places pour caractériser un état, mais ne contiennent pas d'informations. Par contre, une horloge est attachée à chaque transition du système, et, par un système d'intervalle attaché aux transitions, il est possible de déclarer des fenêtres temporelles pour autoriser le franchissement d'une de la transition. Ainsi, on peut détecter des interblocages dûs au fait que des jetons n'ont pus être transférés à temps.

– Les réseaux de Petri *Stochastiques* : ces réseaux de Petri ajoutent de l'indéterminisme dans le franchissement des transitions quand un choix se présente, en ajustant avec des poids les probabilités de franchissements transitions.

Pour effectuer des analyses comportementales complémentaires, nous nous sommes focalisés dans la suite de ce mémoire sur les réseaux de Petri colorés et les réseaux de Petri temporels à priorité.

4.2 Définition des formalismes

Nous présentons avant toute chose les définitions formelles des réseaux de Petri que nous allons utiliser.

4.2.1 Réseaux de Petri place-transition

Ils sont à la base de nombreuses variantes, en étant par exemple enrichis avec des informations de couleur ou de temps pour former les réseaux de Petri colorés ou temporels.

Les réseaux de Petri P-T ont été développés dans les années 60 par C.A. Petri [Pet62] pour modéliser des systèmes concurrents.

Ce sont des graphes bipartis. Les nœuds sont appelés «places», (représentées par des cercles) ou «transitions» (représentées par des rectangles) et sont connectés par des arcs valués. La valuation des arcs en entrée permet d'indiquer des conditions d'«activation» d'une transition alors que la valuation des arcs en sortie détermine l'effet de l'action représentée par la transition.

De façon synthétique, un réseau de Petri P-T est défini comme suit :

Définition 4.2 *Un réseau de Petri est un tuple* $N = \langle P, T, A, w \rangle$*, où :*

- P *est un ensemble fini de places ;*
- T *est un ensemble fini de transitions*
  - $P \cap T = \emptyset$ *;*
- A *est un ensemble d'arcs, tel que* $A \subseteq P \times T \cup T \times P$
  - *un arc connecte une place à une transition, ou inversement (un arc ne peut connecter deux places ou deux transitions ensembles).*
  - *pour une place donnée* $p \in P$ *on note*
    - $p^\bullet$ *les transitions en sortie de* p *:* $p^\bullet = \{t \mid (p,t) \in A\}$*,*
    - ${}^\bullet p$ *les transitions en entrée de* p *:* ${}^\bullet p = \{t \mid (t,p) \in A\}$*,*
    - *Note : il existe l'équivalent* $t^\bullet$ *et* ${}^\bullet t$ *pour les transitions* $t \in T$*.*
- w *est une fonction de poids (valuation) :* $w : P \times T \cup T \times P \to \mathbb{N}$ *;*
  - *les arcs d'un réseau de Petri P-T sont valués ;*
  - *tous les arcs d'un réseau de Petri P-T ont une valuation* ≥ 1*.*

Un réseau de Petri possède un «marquage» M, qui est définit comme une fonction $M : P \to \mathbb{N}$: si pour une place p, $M(p) = n$, alors on dit que la place p contient n jetons dans ce marquage.

Règles de tir Dans un marquage M, on dit qu'une transition peut être «franchie» (ou «tirée») si

$$\forall\, p \in^\bullet t \bullet M(p) \geq w(p,t)$$

soit, qu'une transition t est franchie seulement si toutes les places entrantes de t contiennent un nombre de jetons supérieur ou égal au poids de l'arc allant de p vers t.

Quand une transition est franchie dans un marquage M, $w(p,t)$ jetons sont retirés des places entrantes de t (donc des places $\in^\bullet t$), et un nouveau marquage M' est produit :

$$\forall p, M'(p) = M(p) - w(p,t) + w(t,p)$$

L'évolution d'un réseau de Petri est caractérisée par la succession des marquages obtenus en franchissant des transitions, à partir d'un marquage initial M_0.

A noter qu'il est requis qu'une seule transition puisse être franchie à la fois (pour passer d'un marquage M à M').

Graphe des marquages accessibles L'ensemble des marquages que peut atteindre un réseau de Petri au cours de son exécution est regroupé dans un graphe spécifique. Le graphe des marquages accessibles (GMA), est construit à partir du modèle formel, et utilisé pour effectuer des analyses.

Les model-checkers analysent ce GMA, à l'aide de formules et de prédicats : ils sont alors en mesure d'exhiber des marquages fautifs ou des chemins d'exécutions y menant.

4.2.2 Réseaux de Petri colorés

Les réseaux de Petri colorés sont dérivés des réseaux de Petri place-transtion.

Définition 4.3 *Un réseau de Petri coloré est un 5-uplet $\langle P, T, C, Pre, Post\rangle$ où :*
- *P est un ensemble de places ;*
- *T est un ensemble de transitions ;*
- *C est une famille indexée par P∪T d'ensembles C(x) ; C(x) est appelé domaine de couleurs associé à la place ou à la transition x ;*
- *Pré et Post sont deux familles indexées par P×T d'applications Pré(p,t) et Post(p,t) telles que pour tout (p,t) de P × T, Pré(p,t) et Post(p,t) aient pour domaine C(t) et co-domaine Bag(C(p))*[3]*. Pré et Post sont respectivement appelées fonction d'incidence avant et fonction d'incidence arrière.*

Les domaines de couleurs sont structurés :
- Ils sont nommés «classes» et représentent généralement des objets primitifs (tâche, ressources, etc.). Les classes sont des ensembles finis . Pour certains modèles, il peut être intéressant de définir un ordre (total) entre les couleurs de la classe : la classe est alors dite «ordonnée».
- Les couleurs d'une classe sont des objets d'un même type mais peuvent avoir des comportements différents : par exemple pour une classe définissant le type «tâche», les couleurs peuvent différencier les tâches intéractives des tâches de fond. La classe est alors dite partitionnée en «sous-classes statiques».

Un domaine de couleur est un produit cartésien des classes d'un réseau de Petri coloré. Dans un réseau de Petri coloré, un jeton est soit lié à une classe, soit à un domaine de couleur : il contient donc intrinsèquement des informations de type sur l'objet qu'il représente.

Dans cette variante des réseaux de Petri, les arcs peuvent être caractérisés par des fonctions de couleur qui permettent de sélectionner des jetons depuis les places adjacentes dans le but de franchir une transition. On distingue :

Définition 4.4
- *La fonction de projection, qui permet de sélectionner un élément d'un jeton.*
- *La fonction «successeur», notée ++, qui pour une valeur de jeton donnée, permet d'en donner le successeur dans une classe ordonnée.*
- *La fonction de sélection «globale» (ou fonction de diffusion [CDFH91]), notée all, qui pour une classe donnée, permet de produire un jeton par couleur de la classe.*

3. Bag(A) est un multi ensemble qui peut contenir plusieurs fois le même élément de A

On utilise des prédicats pour imposer des contraintes sur le type de jetons autorisés à franchir une transition. Ces prédicats sont nommés «gardes» de transition. Ce sont des expressions booléennes qui ne permettent de franchir une transition que si, par exemple, un jeton appartenant à une certaine classe est valué par la bonne couleur ; ou encore si deux jetons sont de la même couleur (ou de différentes couleurs), etc.

Classes d'équivalences Les réseaux de Petri colorés permettent la construction automatique du graphe des marquages symboliques. Il s'agit d'un espace d'états-quotient dans lequel les nœuds sont des classes d'équivalence entre états concrets, et les arcs des classes d'équivalence entre événements. Ce graphe est construit en exploitant les symétries du modèle, ce qui conduit à un graphe plus simple à manipuler, du fait de sa taille potentiellement exponentiellement plus petit qu'un graphe des marquages accessibles ordinaire.

Règles de tir Dans un réseau de Petri coloré, la règle de franchissement d'une transition est donc la suivante : une transition t est franchissable si

- La garde associée à la transition t est évaluée à vrai pour le marquage du réseau considéré ;
- $\forall p \in P, m(p) \geq Pre(p,t)(c_t)$, i.e. il existe suffisamment de jetons dans les places adjacentes pour remplir les conditions d'activation de la transition t.

4.2.3 Réseaux de Petri temporels à priorité

Les réseaux de Petri temporels [Zub91] sont une extension des réseaux de Petri classiques, où une horloge et un intervalle de temps sont associés à chaque transition du réseau. L'horloge mesure le temps écoulé depuis l'instant où la transition est franchissable. L'intervalle de temps intervient comme une condition de franchissement : la transition ne peut être tirée que si son horloge appartient à l'intervalle spécifié.

Définition 4.5 *Un réseau de Petri temporel est un tuple* $\langle P, T, Pre, Post, \alpha, \beta, M_0 \rangle$ *où :*

- *P est un ensemble de places ;*
- *T est un ensemble de transitions ;*
- *Pre(t) la fonction de pré-condition associée à la transition «t» ;*
- *Post(t) la fonction de post-condition associée à la transition «t» ;*
- $\alpha(t)$ *la fonction donnant l'instant de franchissement de la transition le plus tôt ;*
- $\beta(t)$ *la fonction donnant l'instant de franchissement de la transition le plus tard. Il peut s'agir de l'infini ;*
- M_0 *le marquage initial du réseau considéré.*

Il est à noter que nous allons plus particulièrement utiliser le formalisme des réseaux de Petri temporels *à priorité*, qui ajoutent une relation de priorité entre transitions lorsque plusieurs d'entre elles sont franchissables au même instant. Cette relation de priorité est transitive dans le sytème.

Règle de tir Une transition est dite franchissable si :

- Il existe suffisamment de jetons dans les places adjacentes pour activer la transition.
- En considérant l'intervalle de temps $[a,b]$attaché à la transition : la transition doit être activée depuis au moins a unités de temps (sans avoir été désactivée). Dans ce cas, la transition sera obligatoirement franchie au plus tard après qu'elle aie été activée depuis b unités de temps.

La notion de priorité dans la sous-catégorie de réseaux de Petri temporels que nous considérons permet de considérer une information supplémentaire lorsque deux transitions sont franchissables (puisque nous le rappelons, une seule transition peut être franchie à la fois). Dans ce cas, la transition avec la plus haute priorité sera franchie.

4.3 Application de la méthodologie aux réseaux de Petri : patrons et règles de transformation

Dans cette section, nous appliquons au formalisme des réseaux de Petri les guides et règles de transformation précédemment décrits. Nous utilisons les réseaux de Petri temporels à priorité (exploités avec l'outil Tina [LAA]) et les réseaux de Petri colorés (via la plate-forme CPN-AMI [LIP]).

Un thread AADL, traduit en réseau de Petri, est un assemblage de composants (eux-même en réseaux de Petri), correspondant à l'instanciation des variantes possible de configuration. Nous avons choisi d'assembler les composants réseaux de Petri par synchronisation de transition : cela génère moins de complexité qu'une composition par fusion de place où des ambiguïtés peuvent apparaître, comme le montre la figure 20. La fusion de places n'a pas la même sémantique que la synchronisation de transitions, et peut même générer des places ayant un marquage infini. Il est toujours possible de faire face à ces problèmes en ajoutant de nombreuses règles de fusion, au détriment de la simplicité de mise en œuvre.

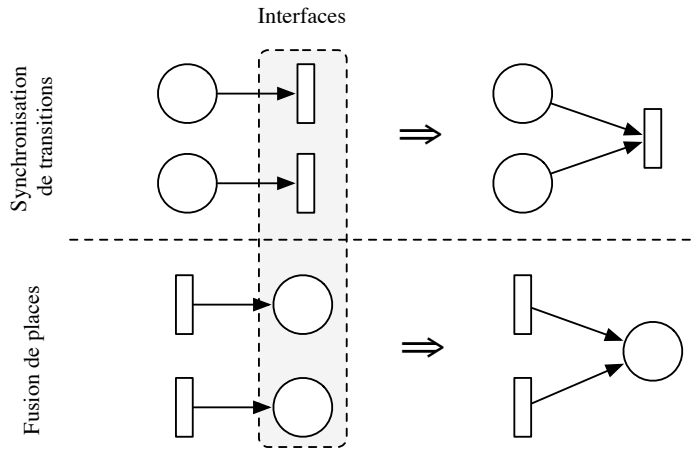


FIGURE 20 – Différence entre la synchronisation de transitions et la fusion de places pour la composition

Chaque composant possède donc un ensemble de transitions et de places locales, ainsi qu'un ensemble de transitions publiques servant d'interface avec d'autres composants.

Dans un composant isolé, les transitions d'interface peuvent être franchies indéfiniment (voir par exemple les patrons de port de données de la section 4.3.4). Cela peut induire un GMA de taille infinie (en cas de production de jetons), ou introduire des cycles d'exécution infinis dans celui-ci (par exemple bouclage de l'exécution de tâches périodiques). Il est donc nécessaire de composer entre elles, par fusion, toutes les interfaces de composants : cela permet de facto de contraindre leur franchissement.

Pour chaque patron identifié dans la section 4.1.2, nous présentons sa mise en œuvre sous la forme de réseaux de Petri. Dans un premier temps, nous présentons la structure des patrons (sans information temporelle ou colorée). Nous particularisons ensuite cette structure avec des annotations colorées (pour l'analyse qualitative) ou avec des annotations temporelles (pour l'analyse quantitative).

Le lecteur trouvera dans l'annexe A les différents algorithmes permettant la construction des patrons présentés dans ce chapitre.

4.3.1 Squelette de thread

Le patron correspondant à un squelette de thread a été décrit en section 4.1.2. L'algorithme 2 construisant ce patron est décrit dans l'annexe A.2.

Structure

D'après le standard, un thread commence dans un état dit «Halted». Cela est directement traduit en réseaux de Petri par une place du même nom.

La phase d'initialisation est représentée par le franchissement de la transition «Initialization_Dispatch» : le thread passe dans un état d'attente sur l'évènement qui va le déclencher. Cela peut être un événement externe dans le cas d'un thread apériodique, ou l'horloge dans le cas d'un thread périodique. Cet état est modélisé par la place «Wait_For_Dispatch». Il ne quitte cet état d'attente que sur la réception d'un événement de déclenchement (franchissement de la transition «Dispatch»), et n'y retourne que lorsqu'il a accompli sa tâche (transition «Complete»).

La structure du réseau de Petri résultant est présentée figure 21. Les transitions «Complete» et «Dispatch» sont des interfaces de composition, au travers desquelles interagirons les sous-composant de déclenchement et de traitement qui seront présentés ci-après.

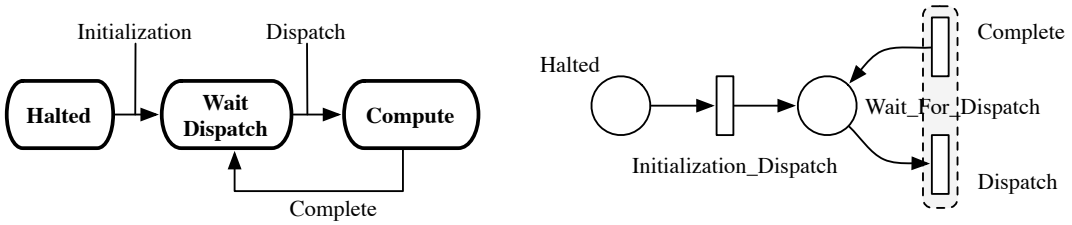


FIGURE 21 – Structure du squelette d'un thread en réseau de Petri

Réseau coloré

Nous pouvons spécialiser cette structure neutre à l'aide d'informations supplémentaires pour un type de réseau de Petri. Nous présentons les annotations dérivant la structure vers les réseaux de Petri colorés.

Nous introduisons la classe de couleur «thread_id» : chaque jeton de ce type transporte l'identité d'un thread particulier du système, identifiable dans le cadre de la vérification. Ces identifiants prennent leurs valeurs dans $\mathbb{N}$. Un identifiant ($\langle 0 \rangle$) est cependant réservé pour palier à certains biais de modélisation, comme l'initialisation du modèle pour éviter un réseau non vivace depuis l'état initial par exemple. Cet identifiant étant le plus petit, il définit la borne inférieure de «thread_id», et est noté «low».

L'identité des threads du système prend une valeur entre les bornes «low»$+1$ et «high», qui sont définies ainsi :

- «low»$+1$ est la plus petite valeur d'identifiant de thread dans le système,
- soit N le nombre d'instances de threads du système décrit par la spécification AADL : alors «high» vaut «low»$+\ 1\ +\ N\ -1$, soit «low»$+N$

Ces identifiants sont véhiculés entre chaque place et transition à l'aide d'une variable locale, «x».

Initialisation : La place «Halted» est initialisée par un jeton portant l'identité du thread modélisé, symbolisée par $\langle id \rangle$.

La figure 22 montre le sous-composant décrivant le squelette d'un thread en réseaux de Petri coloré.

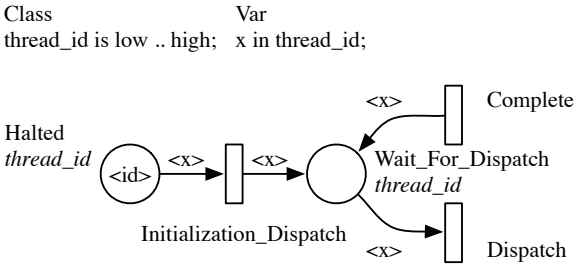


FIGURE 22 – Squelette d'un thread en réseau de Petri colorés

Réseau temporel

Nous présentons les annotations de la structure neutre la spécialisant vers les réseaux de Petri temporels à priorité, se traduisant dans le patron présenté figure 23.

Par défaut, les transitions des patrons sont des transitions dites immédiates : l'intervalle de temps équivalent est $[0, 0]$ (non reporté dans les figures suivantes). La transition est franchie dès que les préconditions nécessaires sont réunies. Graphiquement, ces transitions sont représentées comme des rectangles pleins noirs.

Dans le cas du squelette de base d'un thread, l'étape d'initialisation peut mettre un certain temps à être effectuée : pour cela, la transition «Initialization_Dispatch» possède un intervalle de temps de $[0, \infty[$: la sémantique est celle d'un réseau de Petri classique, la transition pouvant être franchie ne l'étant pas nécessairement. Ce délai peut être précisé comme le montre l'algorithme 7 de l'annexe A

Initialisation : La place «Halted» est initialisée avec un jeton, indiquant que le thread modélisé existe et est près à être activé.

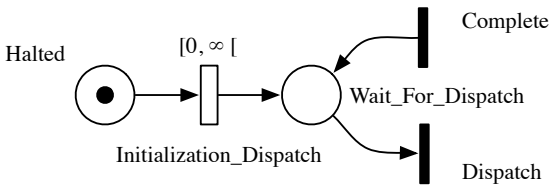


FIGURE 23 – Squelette d'un thread en réseau de Petri temporels

4.3.2 Déclenchement d'exécution

Le patron correspondant à un déclenchement de thread à été décrit en section 4.1.2. Les algorithmes 7 et 12 de l'annexe A décrivent la construction de ces patrons.

Structure

La structure de ce sous-composant est très différente en réseaux de Petri colorés et en réseaux de Petri temporels. Nous ne sommes pas en mesure d'exhiber un sous-ensemble structurel significatif. Nous rappelons que ce sous-composant permet de modéliser les différentes variantes de déclenchement d'un thread (figure 24).

FIGURE 24 – Patron de déclenchement des threads en réseaux de Petri colorés

Réseau coloré

En réseaux de Petri colorés, la notion de temps n'est pas considérée. En conséquence de quoi les notions de thread sporadique ou apériodique se confondent. Nous ne retenons donc que les patrons concernant les tâches périodiques et les tâches sporadiques. La notion de temps n'intervenant pas, il n'est possible de différencier un thread sporadique d'un thread périodique que de la façon suivante :

- un thread sporadique est déclenché dès qu'il reçoit un message sur l'un de ses ports en entrée. La transition permettant d'extraire un message de ces ports est composée avec la transition «Dispatch» pour réaliser cette modélisation
- un thread périodique pourra être déclenché dès que possible, i.e. dès qu'un thread a terminé son cycle d'exécution

Une tâche périodique est déclenchée depuis son état d'attente (Wait_For_Dispatch). Cela peut mettre en exergue certains comportements fautifs, comme la monopolisation du processeur par un thread : la ressource est indéfiniment prise par le même thread tout le temps. Ce comportement éliminé avec les réseaux de Petri temporels, où la notion de temps permet de réduire l'espace des états possibles.

La figure 25 présente le composant de déclenchement, aussi bien dans le cas d'un thread sporadique que d'un thread périodique. Il s'agit d'une place, marquée d'un jeton, et d'une transition, a priori toujours franchissable : c'est le cas pour les threads périodiques ; pour les sporadiques, une condition supplémentaire viendra de la fusion avec les ports en entrée.

Initialisation : La place de déclenchement est marquée d'un jeton.

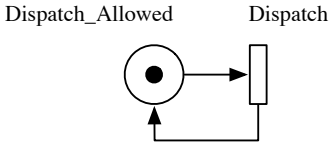


FIGURE 25 – Composant réseau de Petri coloré permettant à un thread de voir son exécution déclenchée

De façon pragmatique, nous ignorerons ce patron, implicitement invoqué lors de la construction de threads périodiques ou sporadiques.

Réseau temporel

Les réseaux de Petri temporels permettent d'exploiter la notion de temps dans les modèles, offrant des moyens de traduction directs des attributs d'ordonnancement et de déclenchement des tâches AADL.

Pour les tâches périodiques, nous proposons un patron créant un jeton de dispatch «horloge», à intervalles réguliers, permettant à la tâche d'être déclenchée. Ce dernier sera effectif quand l'intervalle minimal de temps correspondant à l'entre période sera écoulé.

Une transition, nommée «Period_Event» produit un jeton dans une place «Clock», tous les P intervalles de temps. Cette place est liée à une transition «Dispatch» qui sera fusionnée à celle du patron *squelette* (c'est pourquoi elle est indiquée comme étant une interface).

Initialisation (1) : La transition «Period_Event» possède une garde temporelle initialisée à $[P, P]$, où P est la période du thread, valuée par l'attribut `Period`.

Ce patron, seul et non contraint, induit des comportements fautifs (qui sont en réalité de faux positifs) : le graphe des marquages accessibles est infini, car un nouveau jeton de déclenchement pourra sans cesse être généré.

Pour limiter l'injection d'événements dans le système, afin de limiter l'explosion de l'espace d'états, un choix courant dans l'industrie est de définir une hyperpériode du système sur laquelle raisonner. Nous nous intéresserons donc dans la suite de ces travaux aux threads périodiques et sporadiques.

Nous contraignons donc ce patron en introduisant une place supplémentaire «Hyperperiod_Bound», bornant le nombre de tirage de la transition «Period_Event» (figure 26).

Initialisation (2) : Le marquage de cette place est tel qu'un thread sera déclenché au plus un nombre de fois permettant une analyse sur une hyperpériode «H» du système, calculée à partir de la période de tous les threads. Cette valeur est calculée par une analyse statique de la spécification AADL.

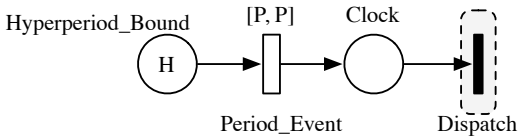


FIGURE 26 – Patron de déclenchement de thread périodiques en réseaux de Petri temporels.

Pour les threads sporadiques (ou apériodiques), le patron de communication présenté ci-après (section 4.3.3) est suffisant. Nous ne nous intéressons pas à l'ordre d'arrivée des événements, cela étant réservé à l'analyse utilisant les réseaux de Petri colorés.

4.3.3 Réception - Calcul - Envoi

Le patron correspondant au sous-composant de calcul d'un thread à été décrit en section 4.1.2. L'algorithme 5 de l'annexe A expose les constructions mises en œuvre pour ce patron.

Structure

Un thread, une fois déclenché, peut recevoir de données, les traiter, et envoyer un résultat. Les étapes de réception et d'envoi sont détectées lors de l'analyse de la spécification AADL et introduites dans le réseau de Petri si des ports en entrée ou en sortie sont présents.

La première partie de la figure 27 montre la structure basique du sous-composant modélisant la partie de calcul d'un thread :

- Deux interfaces avec le composant squelette des threads, présenté précédemment : les transitions «Dispatch» et «Complete». Franchir la première fait pénétrer le thread en phase de calcul, et la seconde l'en fait sortir
- Deux interfaces liées aux communications : une pour recevoir de nouvelles données, potentiellement intégrées dans les calcul (transition «Receive»), une autre pour diffuser des résultats ou des événements (transition «Send»)
- Une place modélisant la phase de calcul, «Compute»
- Des places intermédiaires nécessaires pour construire le composant sans forcer la synchronisation entre la réception de messages et le déclenchement des threads, ou entre l'envoi de messages et la complétion d'un thread. Cela reflète la position du standard quant aux moments où les threads peuvent recevoir ou envoyer des données (séparation d'avec le déclenchement ou la fin d'un thread)

La seconde partie de la figure 27 montre comment il est possible de raffiner ce sous-composant : une fois déclenché, un thread peut exécuter un ensemble de séquences d'appel de sous-programmes, qui viennent s'enchaîner à la place du triptyque $\{Receive, Compute, Send\}$. Si le thread doit exécuter une séquence d'appel, une fois dans l'état «Dispatched», la première transition franchie indiquant le début de la séquence sera «Receive», et la dernière «<Send».

Le patron pour les sous-programmes est l'ensemble de nœuds suivant :

1. une transition «Begin_Seq» et une transition «End_Seq», les transitions représentant les interfaces du composant, marquant l'entrée et la sortie du sous-programme. Ces transitions peuvent être fusionnées avec les interfaces de ports, indiquant que l'on reçoit ou envoie des valeurs de paramètres.
2. une place «SP_Work», indiquant que le thread considéré est en train de procéder à des calculs liés au sous-programme en cours.

Réseau coloré

La figure 28 présente la version colorée de ce patron. Les annotations apportées consistent à transmettre, d'un état à un autre, l'identifiant du thread en cours d'exécution.

Réseau temporel

Par défaut, les transitions permettant de passer d'un état à un autre sont des transitions immédiates.

Cependant, différents attributs AADL peuvent être utilisées pour changer les intervalles de temps des transitions, comme par exemple le temps de calcul (*Compute_Execution_Time*), ou encore le temps de chargement physique des données via un accès bus (*Transmission_Time, Output_Rate* : voir la figure 32, où la transition de réception de messages peut avoir une garde reflétant les délais de communication).

La figure 29 propose les patrons de transformations correspondant à la structure présentée précédemment.

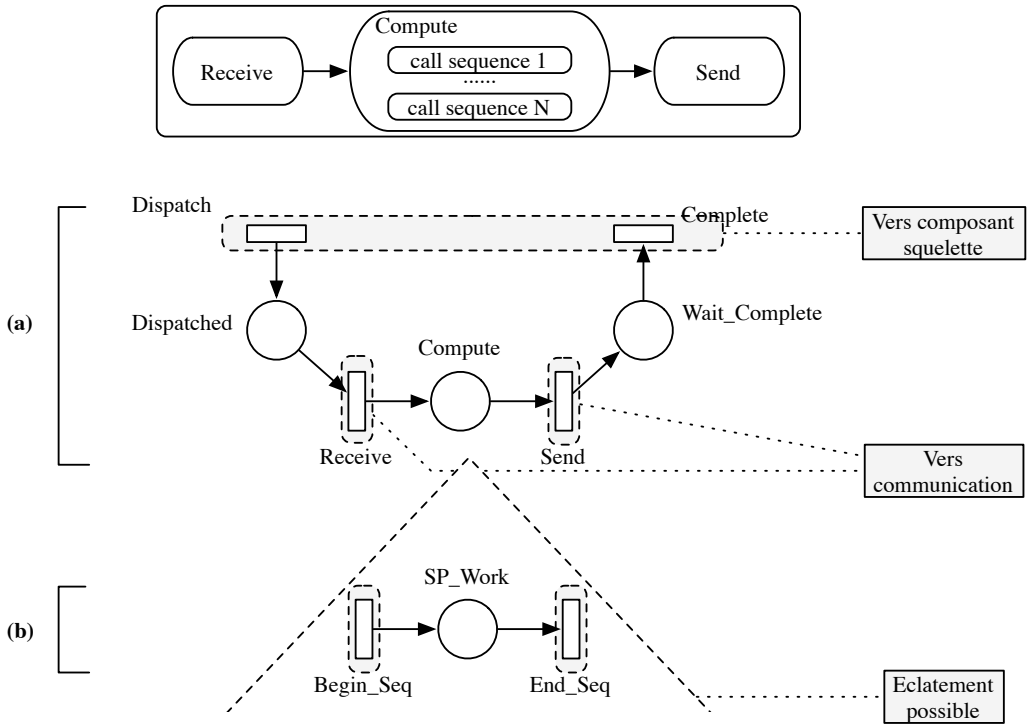


FIGURE 27 – (a) Structure de la partie exécutive d'un thread AADL en réseau de Petri. (b) Patron de sous-composant modélisant un sous-programme.

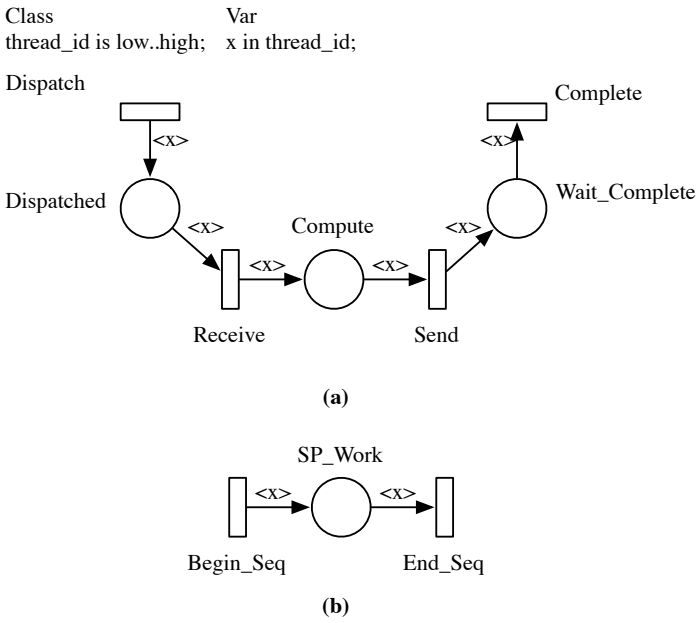


FIGURE 28 – (a) Partie exécutive d'un thread AADL en réseau de Petri colorés. (b) Patron de sous-composant modélisant un sous-programme.

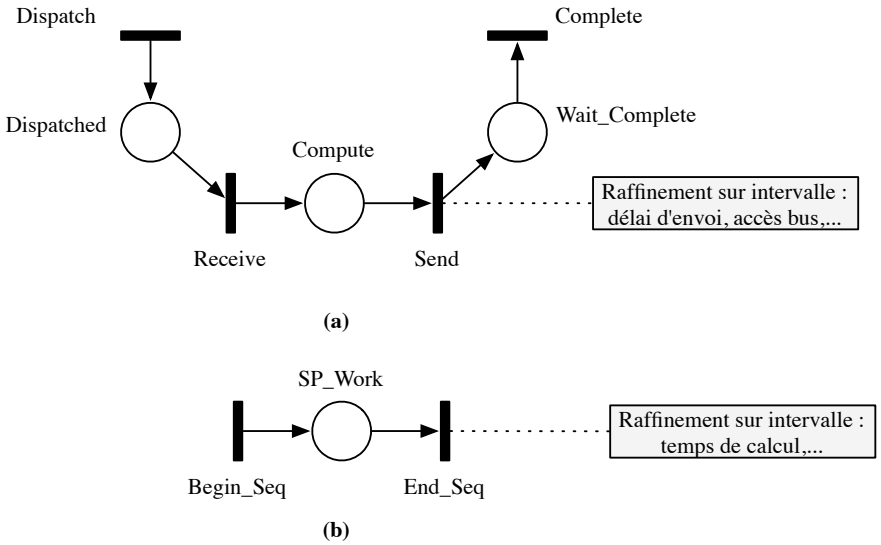


FIGURE 29 – (a) Partie exécutive d'un thread AADL en réseau de Petri temporels. (b) Patron de sous-composant modélisant un sous-programme.

4.3.4 Communication

Le patron correspondant aux communications de thread a été décrit en section 4.1.2. La construction des ports est mise en œuvre dans l'algorithme 8 de l'annexe A.

Il existe trois types de ports en AADL : les ports de donnée, les ports d'événement, et les ports de donnée et d'événement. Nous pouvons cependant les classer en deux catégories :

- Ceux qui transportent de l'information : port de donnée, port de donnée et d'événement
- Ceux qui peuvent intervenir sur le comportement d'un thread (déclenchement) : port d'événement, port de donnée et d'événement

Nous pouvons assimiler dans les analyses comportementales les ports de donnée et d'événement à des ports d'événement, et ne traiter que deux catégories de ports : les ports de donnée, et les ports d'événement.

Structure (port de donnée)

Suivant le patron évoqué dans la section 4.3.3, nous avons établi qu'un port doit posséder au moins deux interfaces : une pour mettre les nouveaux messages dans la file associée au port («Push»), et la seconde pour les en retirer («Pop»).

Pour les ports de données, il y a donc ces deux interfaces ainsi qu'une place permettant de stocker l'information qui transite («Storage»). La figure 33 montre cette structure.

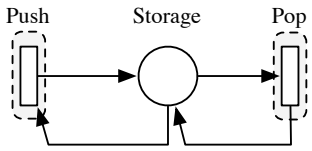


FIGURE 30 – Structure d'un port de données AADL traduit en réseaux de Petri

Ce patron reflète le comportement d'un port de donnée tel qu'exprimé dans le standard : une fois une donnée mise dans un port de donnée, elle y reste jusqu'à son remplacement par une valeur plus récente. Le franchissement de la transition «Pop» ne consomme donc pas seulement le contenu de la place «Storage», mais y remet la donnée. De façon symétrique, il est nécessaire de consommer la valeur déjà présente dans le port avant de la remplacer par une plus récente (connexions entre «Storage» et «Push»).

Réseau coloré (port de donnée)

Nous introduisons un domaine de couleur, correspondant à la définition d'un type de message, tel que ce type soit un tuple $\langle thread\_id, message\_type \rangle$. Cela permet de tracer les messages (et leurs types) d'un thread, et sera utilisé lors des analyses effectuées sur le modèle.

Le patron en réseau de Petri coloré d'un port de donnée est présenté figure 31, et l'algorithme 9 de A.5 décrit sa mise en œuvre.

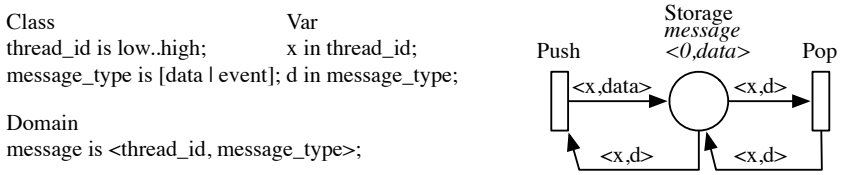


FIGURE 31 – Port de donnée AADL traduit en réseaux de Petri colorés

Initialisation : La place de stockage est initialisée avec un message fictif $\langle 0, data \rangle$, contenant un identifiant de thread réservé qui n'apparaît pas dans le système, de façon à permettre le franchissement, pour la première fois, de la transition «Push» (dans le cas contraire, notre patron provoquerait un interblocage dès l'état initial du réseau).

Réseau temporel (port de donnée)

L'utilisation des réseaux de Petri temporels se prête à des vérifications complémentaires à celles effectuées avec les réseaux de Petri colorés. Les patrons de transformation pour les communications ne sont pas les mêmes : nous utilisons les informations de temps, concernant les tâches, pour analyser le bon dimensionnement des buffers de communication.

Pour les ports de donnée, le patron est sensiblement le même que pour les réseaux colorés, comme le montre la figure 32. Ces patrons sont mis en œuvre dans l'algorithme 10 de l'annexe A.

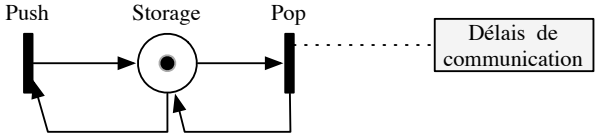


FIGURE 32 – Port de donnée en réseaux de Petri temporels

Initialisation : Comme pour le patron en réseaux de Petri coloré, il est nécessaire d'initialiser la place de stockage pour éviter un interblocage provoqué par la modélisation du système. A noter que les transitions *Push* ou *Pop* peuvent voir leur garde temporelle initialisé selon la valeur T d'attributs comme `Transmission_Time` $[T, T]$.

Structure (port d'événement)

Quel que soit le type du port considéré, nous avons établi qu'il doit posséder au moins deux interfaces (Push et Pop). Pour les ports d'événement, il est possible de considérer une troisième interface (Overflow) permettant de gérer les politiques décrites par l'attribut `Overflow_Policy` d'un port, comme le décrit la figure 33.

Par ailleurs, le contenu du patron entre les interfaces peut être personnalisé selon la valeur de l'attribut `Queue_Size`. Nous reviendrons en détail sur ces attributs par la suite.

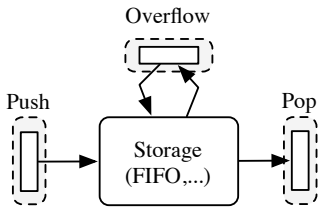


FIGURE 33 – Structure d'un port d'événement AADL traduit en réseaux de Petri

Réseau colorés (port d'événement)

Pour les ports d'événement, le patron se décline selon les différentes valeurs des différents attributs d'un port : taille de file d'attente et politique à appliquer en cas de file pleine.

Initialisation : Le nombre d'étage de la file (entre les interfaces Push et Pop) dépend de la valeur de la propriété «Queue_Size» (figure 34).

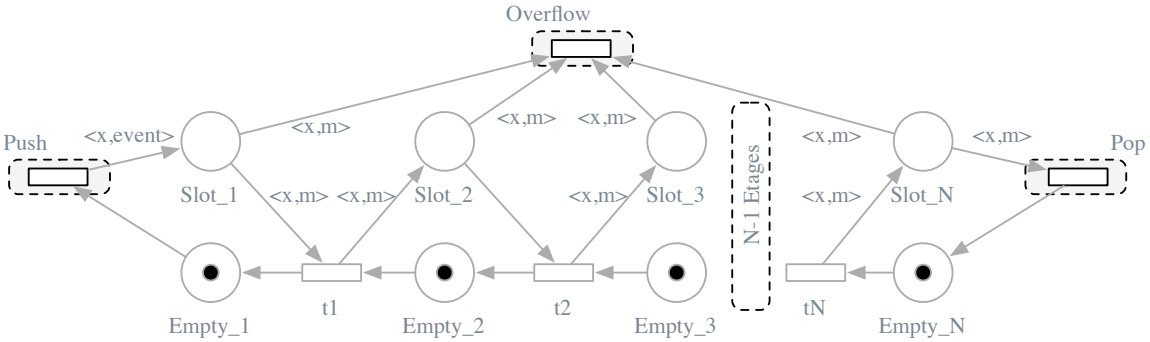


FIGURE 34 – Impact de l'attribut QueueSize sur le patron de port d'événement en réseaux de Petri colorés.

Un étage de file d'attente est composé d'une place stockant l'information en cours, et d'une place duale indiquant si la première contient de l'information, ou si elle est vide. On utilise la seconde place

pour savoir si l'on peut transmettre le message d'un étage à un autre de la file. Si cette place est marquée, la place contenant les informations est vide. Dans le cas contraire, l'étage est occupé, et cela inhibera le franchissement de la transition associée.

Le nombre d'éléments prélevés de la file (i.e. jusqu'à quel étage retire-t-on les éléments lors d'un Pop) est fixé par la propriété «Dequeue_Policy». On produira soit un élément («One_Item»), soit un tuple d'éléments de la file en sortie.

Par construction, comme le montre la figure 35, un élément ne pourra être inséré dans la file que si une case est libre (équivalent à la place «Empty» correspondante marquée d'un jeton incolore). Si aucune case n'est libre, une autre interface de composant intervient une nouvelle interface, complémentaire de *Push* : la transition *Overflow_Policy* (qui sera renommée en fonction de sa valeur : *DropOldest* | *DropNewest* | *Error*.

- DropOldest : un arc part de toutes les places de la file, et tous les éléments de la file sont re-routés une case en avant, le nouveau message étant inséré en dernier, le plus vieux message étant perdu.
- DropNewest : un arc part de toutes les places de la file, et tous les éléments de la file reviennent à leur place, le message arrivant étant perdu.
- Error : un arc part de toutes les places de la file, bloquant le modèle : rien n'est remis dans la file. On pourra observer le franchissement de cette transition en analysant le graphe des marquages accessibles.

L'utilisation de ce patron est particulière : lorsqu'un composant envoie un message via une transition Θ(typiquement la transition *Send* du patron de la section 4.3.3), et que ce message doit passer par un port d'événement, alors il faut

- dupliquer Θ : on obtient Θ_a et Θ_b, chacun ayant les mêmes arcs entrant que Θ
- fusionner Θ_a et «Push»
- fusionner Θ_b et la transition relative à la politique appliquée en cas de file pleine («DropOldest», «DropNewest», ou «Error»)

Cet algorithme permet de mettre en exclusion mutuelle le franchissement de la transition «Push» et de «DropOldest» (par exemple) :

- Franchir «Push» ne sera possible que si la file n'est pas pleine
- Franchir «DropOldest» ne sera possible que si la file est pleine
- Quelle que soit la situation, une et une seule des deux transitions sera franchie quand le thread sera dans un état menant à l'envoi d'un message

Réseau temporel (port d'événement)

Le patron en réseaux de Petri temporel pour les ports d'événement est exactement semblable à celui des réseaux de Petri colorés sans les annotations de couleur.

Cependant, nous proposons un patron différent permettant d'effectuer des analyses complémentaires de celles effectuées à l'aide des réseaux de Petri colorés, afin d'exploiter au mieux les spécificités de chaque formalisme.

Comme les réseaux colorés permettent d'effectuer des vérifications sur l'utilisation des politiques d'overflow, nous avons décidé de tirer parti des réseaux temporels pour analyser le nombre maximal de message pouvant transiter par un port d'événement (dimensionnement).

Nous ne nous baserons donc pas exactement sur la même structure pour présenter le patron : un port d'événement sera représenté comme tous les ports une place «Bus», avec deux interfaces *Push* et *Pop*.

Nous ne traitons pas les politiques d'overflow pour l'analyse.

Initialisation : Les interfaces sont des transitions immédiates par défaut ($[0, 0]$). Il est cependant possible d'y répercuter les attributs indiquant les temps de transfert de message dans le système, que nous

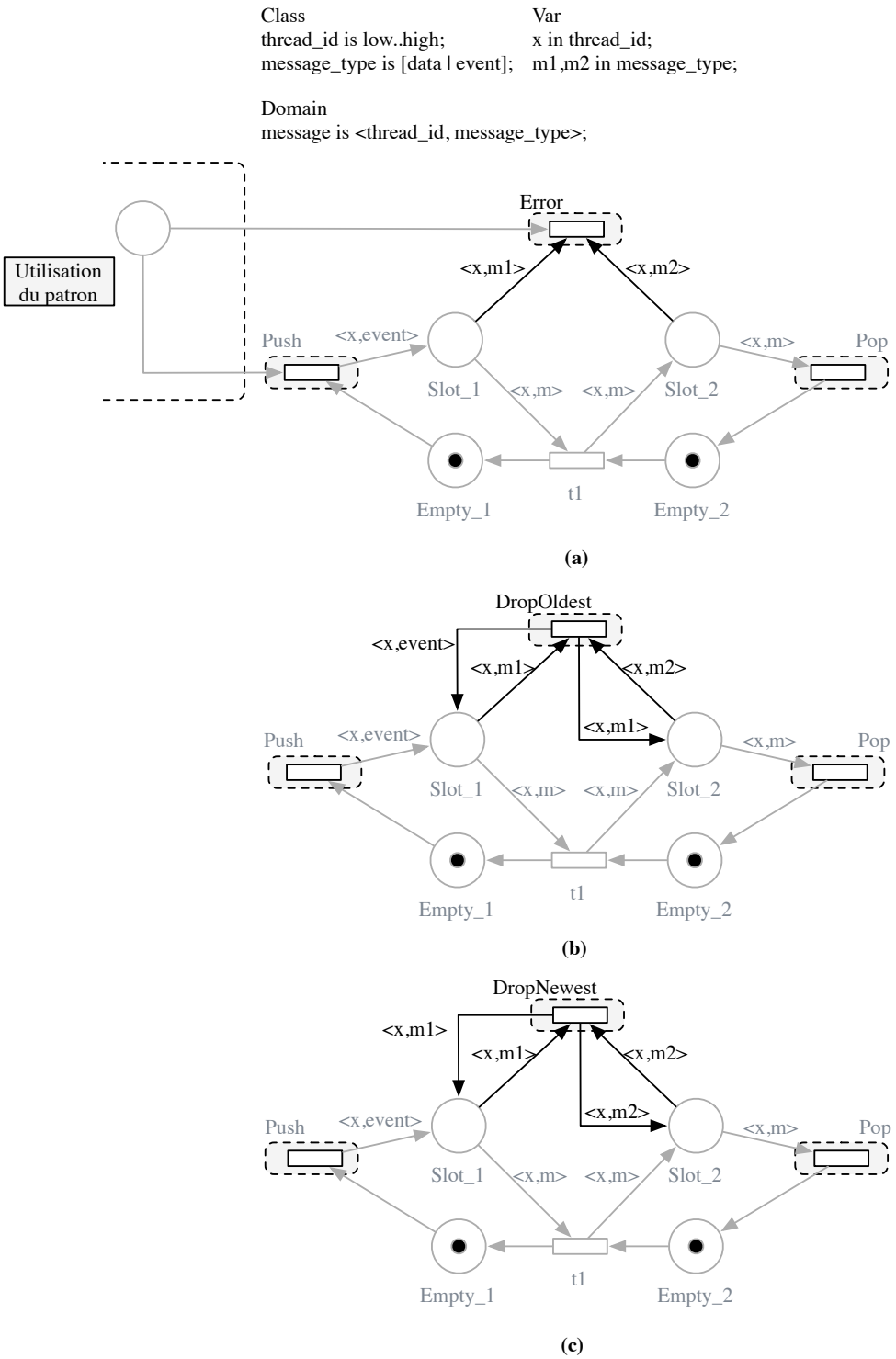


FIGURE 35 – Variations des patrons de communication pour les ports d'événements en réseau de Petri colorés : (a) Error, (b) DropOldest, (c) DropNewest. Pour chacun d'eux, politique de dequeue «One_Item»

avons vu plus tôt en section 3.3.

Nous pourrons par ailleurs vérifier, en utilisant les formules de logique temporelle adéquates, que le nombre maximum de jetons dans la place «Bus» reste inférieur à la valeur spécifiée en AADL à l'aide de la propriété *Queue_Size*.

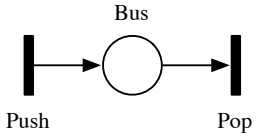


FIGURE 36 – Port de donnée et événement en réseaux de Petri temporels

4.3.5 Assemblage

Nous allons présenter ici les étapes et les règles à suivre pour produire un composant réseau de Petri correspondant à un thread. L'algorithme correspondant de l'annexe A est le numéro 12.

Construction d'un thread complet

Les politiques de déclenchement sont traitées lors de la construction complète d'un thread.

S'il s'agit d'un thread périodique (ou sporadique), il faut alors fusionner la transition «Dispatch», du composant de déclenchement, avec la transition «Dispatch» du squelette de thread.

Dans le cas de threads sporadiques, le déclenchement dépend de la présence ou non d'événements dans un port en entrée. Pour tous ses ports d'événement en entrée susceptibles de déclencher le thread, la fusion de leurs transitions d'interface «Pop» doit être effectuée, avec une copie de la transition «Dispatch» du thread (qui doit être dupliquée pour chaque port de déclenchement). La construction du thread s'achève en combinant aux précédents composants celui de calcul (section 4.3.3). L'analyse de la spécification AADL permet de savoir si le thread effectue des séquences d'appels, et s'il reçoit (ou envoie) des données.

Nous fusionnons donc les transitions «Dispatch» du composant de calcul avec la (ou les, si les ports de déclenchement ont leurs propres séquences d'appel) interface(s) du thread en cours d'assemblage. De la même façon, nous fusionnons la transition «Complete» du composant calcul avec la transition «Complete» du squelette.

La figure 37 présente un exemple d'assemblage pour un thread sporadique en réseaux de Petri colorés. La figure 38 présente l'assemblage pour un thread périodique de période P, en réseaux de Petri temporels.

Construction du système

Les interactions entre threads sont prises en compte lors de la construction du système. Comme elles se font par l'intermédiaire des ports, nous fusionnons directement les patrons correspondants : il n'existera qu'une seule instance de port entre deux thread (i.e. : il s'agira du même port entre celui en sortie pour un thread T_1 et celui en entrée pour un thread T_2).

Par ailleurs, le standard spécifie que tous les threads du système effectuent en même temps leur phase d'initialisation : de façon à traduire cette synchronisation, nous fusionnons alors toutes les transitions «Initialization_Dispatch» de tous les threads du système.

La figure 39 présente un exemple d'instanciation de système, en réseaux de Petri colorés. Elle y montre un thread périodique (en bas) qui produit des événements qui servent de déclenchement à

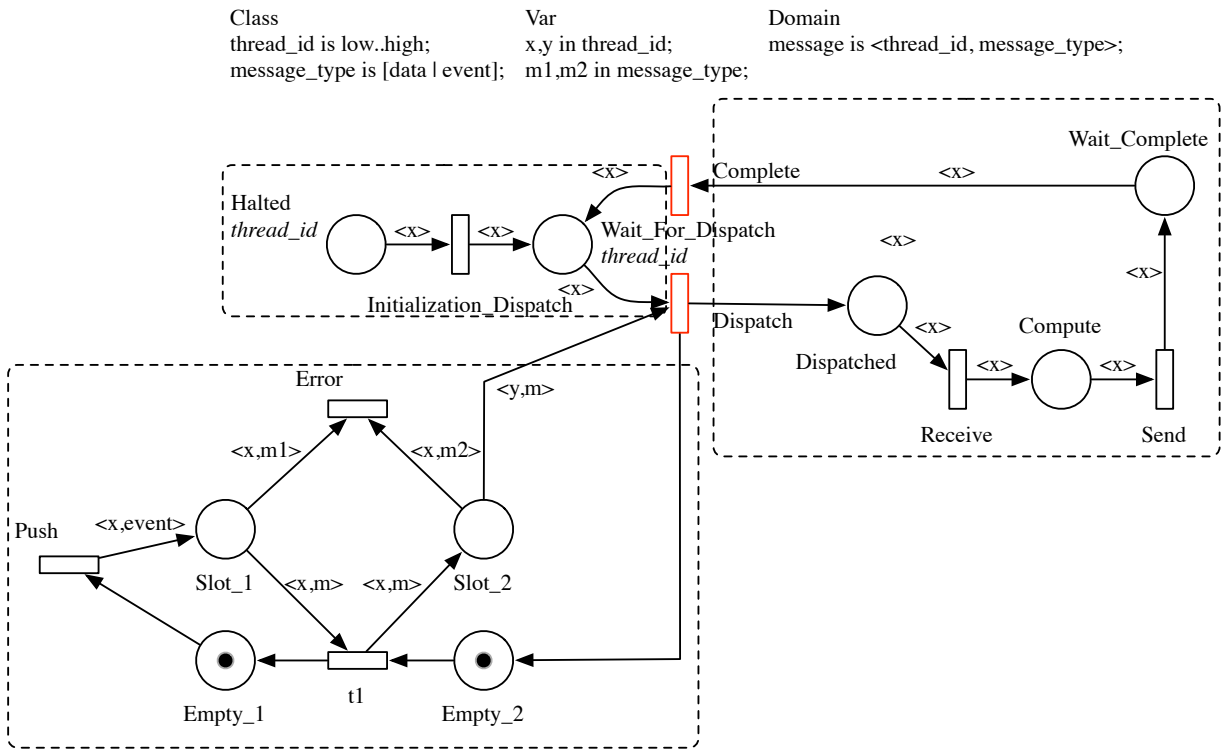


FIGURE 37 – Exemple d'assemblage de sous-composants de réseaux de Petri colorés pour construire un thread sporadique (déclenchement de l'exécution sur réception d'un événement).

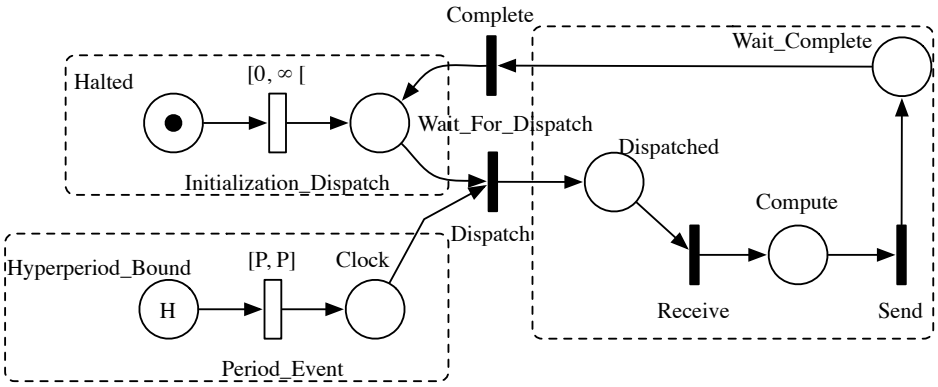


FIGURE 38 – Assemblage pour un thread périodique en réseaux de Petri temporels.

un thread sporadique (en haut). Par soucis de présentation, les transitions «Initialization_Dispatch» de chaque thread ne sont pas fusionnées.

Les règles d'assemblage édictées sont regroupées dans l'algorithme 11 de l'annexe A.

Particularité des réseaux de Petri temporels à priorité : Ils nous permettent à cette étape d'assemblage d'exploiter l'attribut *Priority* des threads AADL. Cela permet de restreindre les chemins d'exécution dans le graphe d'accessibilité en ne laissant que ceux effectivement calculés par l'ordonnanceur.

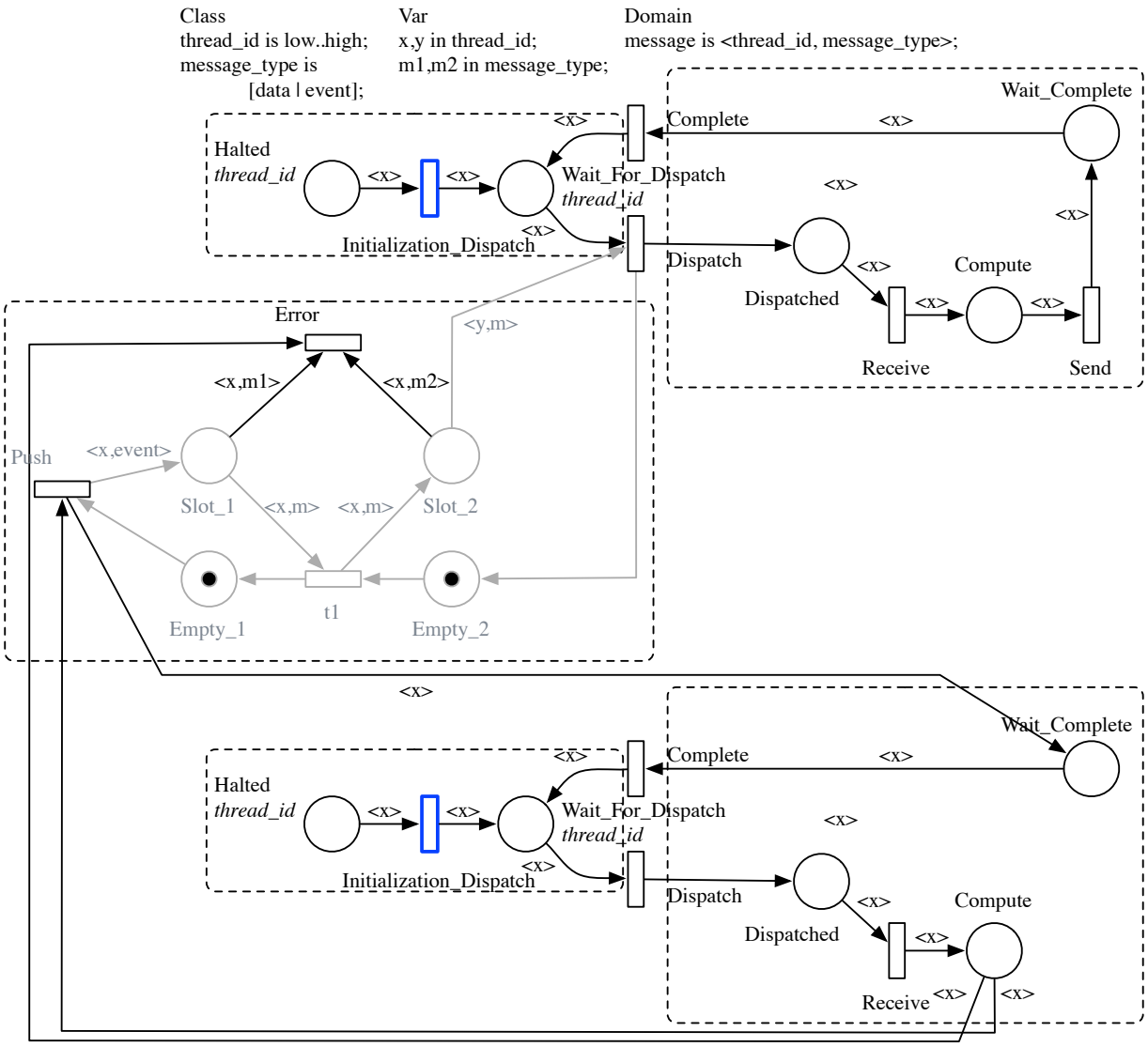


FIGURE 39 – Exemple d'assemblage d'un système AADL en réseaux de Petri colorés.

Si plusieurs transitions sont activées au même moment, au lieu d'en franchir une au hasard, celle appartenant au thread le plus prioritaire sera choisie. Cela est possible grâce au mécanisme de priorité entre transitions du formalisme.

Une tâche plus prioritaire aura sa transition «Dispatch» de plus haute priorité que les autres. Cela est traduit graphiquement par un arc partant de la transition avec la plus haute priorité vers celle ayant la plus basse.

L'algorithme 13 de l'annexe A montre comment la construction du système en est affectée.

La figure 40 présente un assemble de deux threads périodiques, s'échangeant des informations via un port de données.

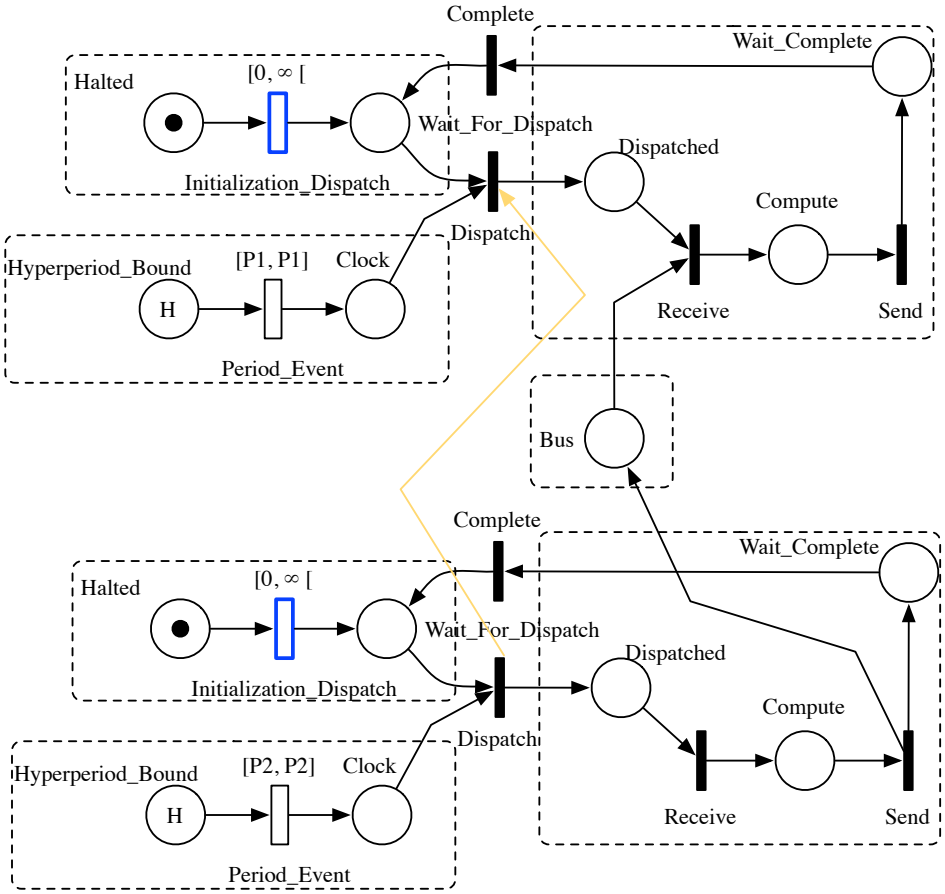


FIGURE 40 – Exemple d'assemblage pour un système complet en réseaux de Petri temporels. Les niveaux de priorités se traduisent avec les arcs jaunes.

4.3.6 Vérifications spécifiques

Nous avons présenté des patrons de transformation depuis AADL vers les réseaux de Petri colorés et vers les réseaux de Petri temporels à priorité. Nous avons exhibé la structure commune de ces patrons et spécifié quelles annotations permettaient de les spécialiser vers l'un ou l'autre de ces formalismes, dans le but de procéder à des analyses différentes sur le système.

Certains aspects du système nécessitent également des patrons dédiés à leur analyse : certains peuvent être exprimés dans les deux formalismes présentés (gestion de la ressource processeur), d'autres sont spécifiques à l'un deux : analyse des flots de messages pour les réseaux colorés, gestion des priorités et analyse d'ordonnancement pour les réseaux temporels.

Ressource processeur

Une spécification AADL permet de spécifier sur quel(s) processeur(s) les tâches sont exécutées. Intégrer cette information dans les patrons de transformation permet d'affiner l'analyse du système.

Une place «Processor» est introduite dans le composant de calcul d'un thread. Il existe une place «Processor» par processeur spécifié dans le modèle AADL. Elle est liée à chaque transition du patron RCE : le franchissement d'une transition nécessite la disponibilité du processeur, modélisant la nécessité

de la disponibilité du processeur pour l'exécution.

La place «Processor» est partagée par tous les composants de calcul des threads qui lui sont liés au travers des processus les englobant dans la spécification AADL.

Initialisation : La place Processor est marquée d'un jeton incolore. La consommation du jeton indique la prise du processeur. Aucun autre élément ne pourra l'utiliser tant que le jeton ne sera pas rendu.

Certaines transitions, comme «Dispatch» ou «Complete», prennent et rendent le jeton processeur instantanément, symbolisant le changement d'état du thread (inactif / actif). A l'inverse, la partie traitement du thread prend le jeton processeur au début de son exécution, et ne le rend qu'à la fin.

Ce raffinement de patron est proposé à la figure 41 et est mis en œuvre dans l'algorithme 11 de l'annexe A.

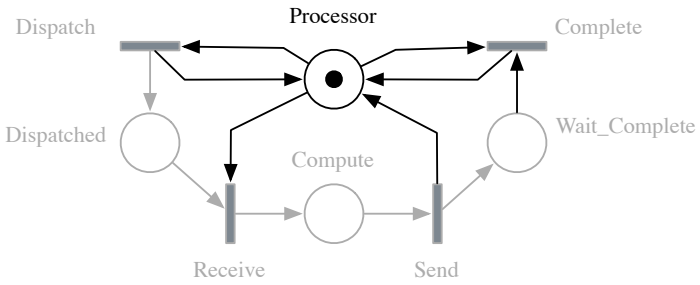


FIGURE 41 – Raffinement du patron de calcul pour la gestion de la ressource processeur.

Estampillage de messages (réseaux de Petri colorés)

Le patron de communication actuel ne permet pas de s'assurer que les messages émis par un thread sont bien reçus et traités par un autre thread. Il n'existe en effet aucun mécanisme différenciant le traitement d'un message par rapport à un autre. La solution retenue est l'estampillage des messages de chaque thread, en utilisant les réseaux de Petri colorés.

Le type de donnée *msg_id*, identifiant les messages, est introduit dans les classes de couleurs. Cela impacte la déclaration du domaine de couleur relatif aux messages, dont le tuple caractéristique est étendu : $\langle thread\_id, message\_type, msg\_id \rangle$.

Le patron de communication est raffiné pour mettre en place les mécanismes utilisant ces nouvelles donées. Une place «Stamp» est créée et liée à la transition *Push* du port ciblé (donnée ou donnée et événement).

Initialisation : Le marquage de la place *Stamp* contient tous les identificants de message possibles, permettant d'identifier chaque message lors du franchissement de la transition Push.

La figure 42 montre l'extension du patron de communication en ce sens. Ces mécanismes sont mis en valeur dans l'algorithme 14.

Analyse d'ordonnancement (réseaux de Petri temporels)

Les patron réseaux de Petri temporels sont utilisés pour analyser l'ordonnancement des différentes tâches du système. Avec les informations disponibles depuis le modèle AADL, nous pouvons vérifier si une tâche manque des instants cruciaux dans son cycle de vie, à l'aide d'observateurs. Ils permettent de

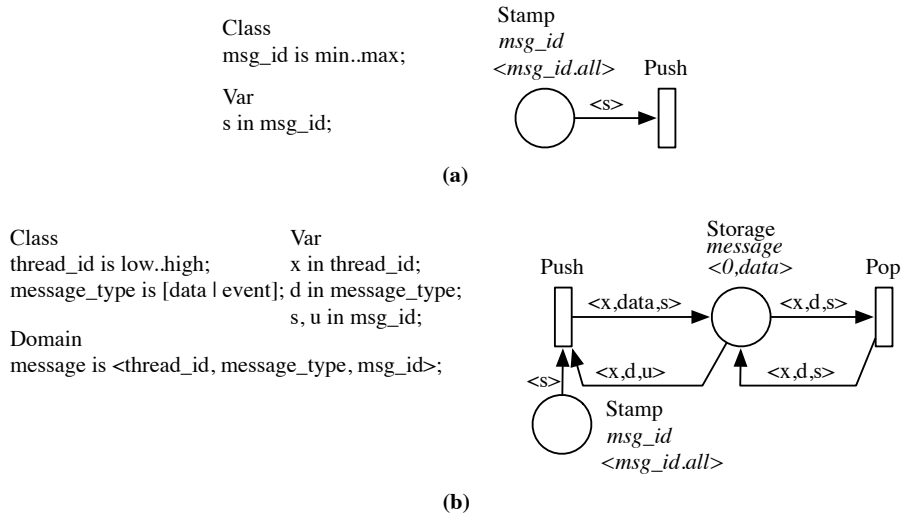


FIGURE 42 – Extension du patron de communication en réseaux de Petri colorés pour estampiller chaque message : (a) Composant réseau de Petri pour l'estampillage ; (b) Modification du patron «Port de donnée».

surveiller si un thread manque son activation, sa deadline, ou encore s'il ne respecte pas son pire temps d'éxécution (WCET pour *Worst Case Execution Time*).

La construction de ces observateurs est décrite par l'algorithme 14 de l'annexe A.

Pour l'activation, nous ajoutons un observateur dans le patron de déclenchement, comme le montre la figure 43. Une tâche manque son activation si un nouvel événement de déclenchement est arrivé alors que la tâche n'est pas dans l'état «Wait_For_Dispatch». On raffine donc également le patron de calcul d'un thread pour introduire une place miroir à «Wait_For_Dispatch», utilisée pour détecter l'événement.

Si le franchissement est possible pour deux transitions différentes, c'est la transition permettant un comportement légal qui doit être prioritaire. C'est pourquoi la transition «Dispatch» est plus prioritaire que celle permettant l'observation de l'événement.

Nous pouvons de la même façon s'assurer qu'une tâche n'excède pas son pire temps d'exécution, comme le montre la figure 44. *W* est la valeur de la propriété *Compute_Execution_Time* du thread. De même, la transition amenant vers un état légal «Complete» doit être plus prioritaire au franchissement que l'observateur, pour éviter les faux positifs lors de l'analyse.

Systèmes préemptifs

Conjointement à l'extension aux systèmes multi-processeurs (section 4.3.6), nous proposons une alternative au patron des sous-programmes, de façon à laisser la possibilité au système de préempter un traitement pour favoriser un thread plus prioritaire.

La figure 45 illustre la mise en œuvre de cette alternative.

Les appels de sous-programmes ne sont plus atomiques, et nécessitent en cours d'exécution de nouveau la ressource processeur. Cela augmente le nombre d'états possible du système, introduisant une difficulté supplémentaire de traitement par les outils. En revanche, cela permet de traiter un aspect supplémentaire de l'analyse d'ordonnancement, complétant les vérifications existantes en envisageant davantage de situations d'éxécution.

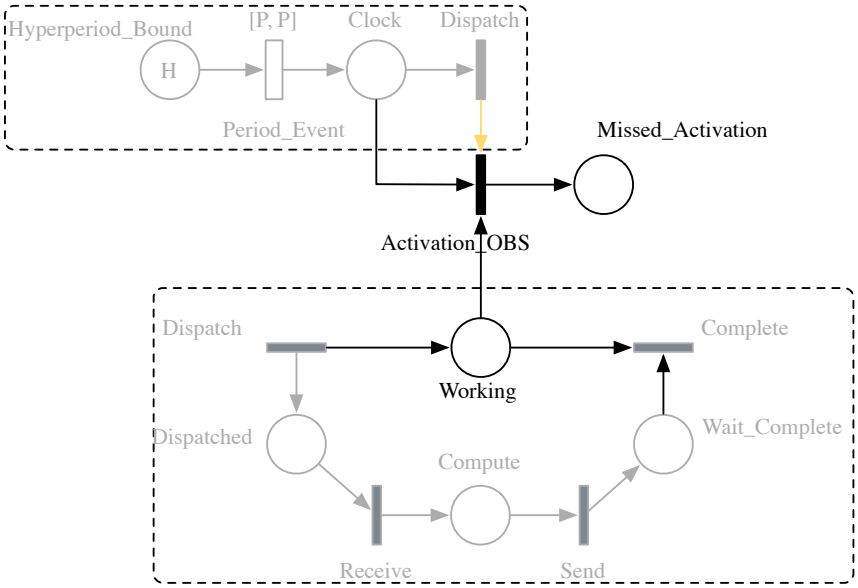


FIGURE 43 – Raffinement du patron de déclenchement et RCE pour vérifier le manquement d'activation d'une tâche.

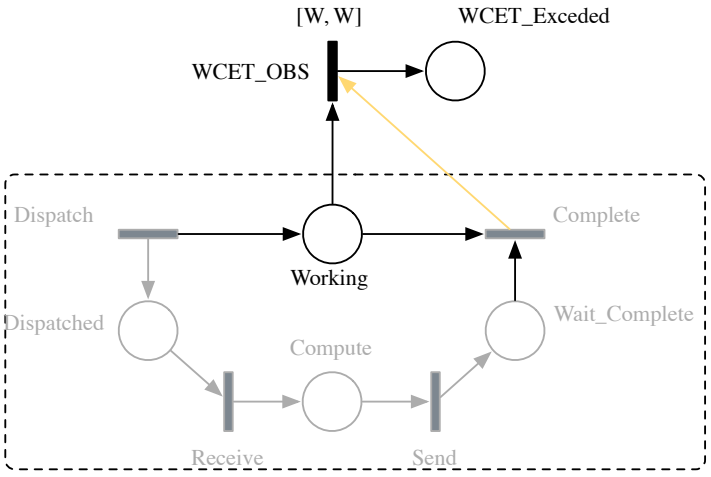


FIGURE 44 – Raffinement du patron RCE pour vérifier le respect du WCET d'une tâche.

4.4 Réduction de la taille des réseaux générés

Ces patrons peuvent mener à la construction d'un grand nombre de places et de transitions (voir par exemple la figure 1 de l'annexe B). Cela introduit de l'entrelacement qui accroît la taille de l'espace d'états à analyser.

Il est possible de réduire le nombre de nœuds du réseau généré en utilisant des techniques de réductions adaptées comme [Had88] (pour les réseaux de Petri colorés) ou [BR76] (pour les réseaux de Petri Place-Transition). Ces opérations de réductions interviennent une fois le réseau de Petri généré : elles opèrent

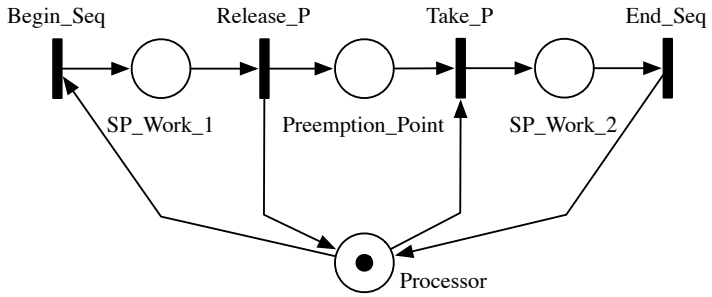


FIGURE 45 – Raffinement du patron de sous-programmes pour modéliser la possibilité de préemption.

en retirant du réseau les places ou les transitions qui ne modifient pas le comportement de l'application, en respectant un ensemble de propriétés comme les interblocages.

Par exemple, la post-agglomération permet de réduire, pour les réseaux de Petri Place-Transition, des séquences de transitions comme suit : si on considère une séquence de type :

$$\langle \textit{Transition}\, T_1,\ \textit{Place}\, P,\ \textit{Transition}\, T_2 \rangle,$$

où la seule transition en entrée de P est T_1, et la seule transition en sortie de P est T_2, il est alors possible d'agréger les transitions T_1 et T_2 en supprimant la place P.

Il est cependant nécessaire d'être précautionneux lors de l'application des techniques de réductions qui peuvent supprimer des places ou des transitions intéressantes à observer pour analyser une propriété. Pour l'instant nous n'appliquons pas ces techniques de réduction, les exemples traités ne le nécessitant pas. Nous citons cette technique comme un moyen complémentaire de réduire la taille des réseaux générés.

4.5 Synthèse

Nous avons présenté dans ce chapitre comment extraire le comportement d'un système à partir sa spécification AADL.

Pour cela, nous avons analysé et interprété le standard AADL pour identifier les éléments et leurs attributs ayant un impact sur le comportement de l'application spécifiée.

Les threads et les ports permettent de spécifier les éléments actifs d'une spécification et leurs interactions. Les éléments hiérarchiques d'une spécification AADL comme les systèmes ou les processus sont utilisés pour mettre en place des mécanismes de traçabilité, permettant de localiser les erreurs potentiellement détectées à l'analyse.

Cette analyse du standard nous a permis d'établir de façon générique des patrons de transformations d'AADL vers un formalisme basé sur des automates états-transitions. Nous avons par ailleurs pu exhiber les points structurels clefs de chacun de ces patrons, facilitant leur génération automatique pour des formalismes particuliers, comme nous le présentons pour les réseaux de Petri au chapitre 6.

L'analyse effectuée a permis de dégager une structure neutre pour chaque patron en réseau de Petri P/T, spécialisable vers deux variantes complémentaires, les réseaux de Petri colorés et les réseaux de Petri temporels.

Nous avons présenté comment personnaliser chacun de ces patrons dans le formalisme choisi, pour permettre de procéder à des analyses complémentaires, mise en œuvre dans le chapitre 7. Les réseaux de Petri colorés permettent de procéder à des analyses qualitatives, comme l'analyse des flots de messages dans le système. Les réseaux de Petri temporels permettent d'effectuer des analyses quantitatives, comme des analyses d'ordonnancement, ou la validation du bon dimensionnement des ressources mises en œuvre.

Nous avons montré qu'il est possible d'intégrer la transformation d'une spécification AADL vers des modèles formels dans un processus de fabrication logicielle pour procéder à des analyses comportementales.

Une application TR^{2}E décrite en AADL suppose qu'il existe un support d'exécution rendant un ensemble de services nécessaires à son bon fonctionnement. Nous allons donc nous intéresser dans le chapitre suivant à la vérification architecturale de PolyORB, un exécutif AADL, en utilisant la notation standardisée formelle Z.

Chapitre 5

Spécification en Z d'un exécutif AADL : application à PolyORB

Contents

Après avoir traité dans le chapitre précédent de l'analyse comportementale des spécifications AADL, nous montrons maintenant comment effectuer une phase de vérification sur l'exécutif servant de support à l'application modélisée en AADL [RHK08].

Une application décrite en AADL suppose que l'exécutif sous-jacent offre un certain nombre de services, comme la création de threads, la gestion de l'ordonnancement, de piles protocolaires, ou encore de l'allocation mémoire.

Plusieurs travaux se sont intéressés à la structure des patrons de génération et aux exécutifs pour AADL. Ainsi, dans notre équipe plusieurs architectures ont été proposées : PolyORB [VHPK04], PolyORB-HI [HZPK08a] et POK [DPF09].

Parmi ceux-ci, PolyORB, un intergiciel générique permettant de construire une large classe de systèmes. Il dispose de l'architecture la plus mature (*i.e.* stable depuis 2006). Les travaux de Th. Vergnaud [Ver06] ont montré que celui-ci peut fournir un support d'exécution complet pour AADL. Son architecture est par ailleurs apte à un effort de modélisation, comme cela a été établi dans les travaux de J. Hugues [HTMK$^+$04].

Nous présentons cet intergiciel en détail à la section 5.3. Il est configuré au travers de la spécification AADL de l'application, qui spécifie toutes les interfaces des composants, les aspects comportementaux, les aspects communications (intra ou inter processus), ainsi que les piles de protocoles.

Nous nous appuyons sur PolyORB pour procéder à la vérification formelle d'un exécutif AADL. Nous présentons donc tout d'abord comment nous procéderons à la vérification formelle de cet exécutif. Puis nous présentons la notation *Z* (section 5.2), qui est le formalisme que nous avons utilisé, puis PolyORB, l'exécutif que nous avons choisi (section 5.3). Nous l'analysons et en isolons alors les éléments d'architecture cruciaux pour la vérification formelle. Nous présentons ensuite les différents patrons paramétriques, en Z, de l'exécutif : nous y spécifions les différents éléments d'architecture et les opérations ou services qu'il est en mesure d'effectuer (sections 5.4 à 5.7). Enfin, nous proposons des guides pour établir des obligations de preuves, permettant par exemple de s'assurer que la combinaison des services de l'exécutif est possible et ne viole pas d'invariants (sections 5.8 et 5.9).

L'ensemble des éléments de spécification que nous présentons dans ce chapitre ont étés vérifiées syntaxiquement par l'outil Z/Eves [MS97], à l'aide duquel nous avons également prouvé des théorèmes relatifs aux opérations spécifiées. Ces théorèmes sont présentés dans l'annexe C.

5.1 Relations entre l'exécutif AADL et sa spécification formelle

Pour les applications temps-réel, réparties et embarquées, les intergiciels sont les supports d'exécution les plus courants. Ils mettent en œuvre divers modèles de répartition qui utilisent en apparence des mécanismes très différents pour véhiculer de l'information entre différents nœuds.

Les travaux de [PK04] ont cependant montré qu'ils partagent un ensemble de services communs : les services intergiciels canoniques. L'architecture schizophrène [Pau01], mise en œuvre dans PolyORB et sur laquelle nous reviendrons dans la section 5.3, est basée sur cette notion de services canoniques.

Ces services sont configurables pour être les plus adaptés aux besoins d'une application TR^{2}E.

Comme nous l'avons montré, AADL permet de spécifier de telles applications.

Les travaux de [Ver06, Zal08] ont montré quelles étaient les relations entre la spécification AADL d'une application et la configuration de l'intergiciel schizophrène servant de support d'exécution.

Comme le montre la figure 46, il existe un intergiciel «minimal» qui présente un ensemble d'interfaces qui permettent de mettre en œuvre l'application TR^{2}E.

Les travaux menés par [Zal08] ont montré que si l'architecture de l'intergiciel minimal était statique, une partie de celle-ci pouvait être configurée pour correspondre au mieux aux besoins de l'application en exploitant sa spécification AADL. En effet, si les protocoles de communications mis en œuvre par l'intergiciel sont eux complètement indépendants de l'application, ce n'est pas le cas des routines d'envoi de messages.

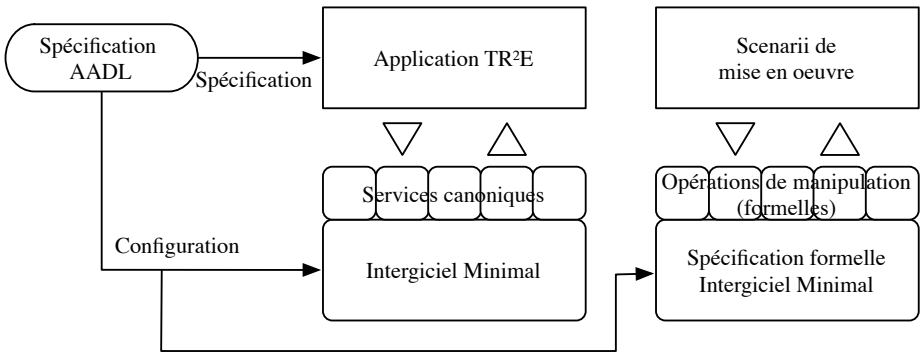


FIGURE 46 – Relations entre l'exécutif AADL, sa spécification formelle et la spécification de l'application TR^{2}E

L'intergiciel propose donc une bibliothèque de services mis en œuvre par l'application, ainsi que différents points de configuration pour répondre à ses besoins.

Nous proposons une approche identique pour la spécification formelle de l'exécutif AADL : elle propose une bibliothèque de service (ceux de l'intergiciel), ainsi que des points de configuration (semblables à ceux de l'intergiciel).

Procéder à la vérification formelle de l'exécutif se fera en plusieurs étapes, comme nous l'avons mentionné :

1. la spécification formelle de l'intergiciel minimal, proposant une bibliothèque de services, doit être cohérente.
2. la spécification formelle de l'intergiciel, *une fois configurée en accord avec la spécification AADL*, doit être cohérente : le respect d'invariants exprimés sur la spécification générique par la spécification configurée montre que la configuration élaborée par l'utilisateur peut être supportée par l'exécutif AADL.
3. des scenarii de mise en œuvre des services de la spécification formelle peuvent être élaborés, pour vérifier certaines contraintes dans le contexte particulier de l'application.

5.2 Z : une notation formelle pour la spécification

Cette notation a été proposée par Jean-Raymond Abrial et son équipe du «Programming Research Group», au laboratoire d'informatique de l'Université d'Oxford (OUCL), dans les années 70. Elle tire son nom de la théorie des ensembles de Zermelo-Frankel [SAEF68]

J-R. Abrial la fait apparaître pour la première fois dans «Data Semantics» [Abr74] en 1974. Elle est l'objet de travaux au cours des années suivantes, et il présente la notation Z avec l'aide de S. Schuman et de B. Meyer dans [ASM80].

Cette notation définit rigoureusement des opérateurs logiques, des opérations ensemblistes, des types complexes et leur combinaison, et la notion d'algèbre sur les schémas («schema-calculus»), sur lesquels nous allons revenir dans ce chapitre.

Un premier effort de standardisation a été mené par M. Spivey [Spi89a] en 1989. Il a ainsi proposé un manuel de référence, y présentant les notions mathématiques à la base de cette notation (et donc utile pour faire de la preuve), le langage en lui-même, ainsi que les outils mathématiques mis à la disposition de la notation. Cela a été fait en utilisant des exemples et des tutoriels, aidant à la compréhension du lecteur. Plus récemment, en 2002, la notation Z a été standardisée selon la norme ISO/IEC [13502].

Nous allons présenter les différents éléments d'une spécification Z, de façon à clarifier la lecture de la suite de ce chapitre.

5.2.1 Syntaxe

La syntaxe de la notation Z repose sur la notion de sections et de paragraphes. Une spécification peut contenir plusieurs sections (introduisant une hiérarchie nommée). Chaque section contient des paragraphes. On peut faire une spécification dite «anonyme» en utilisant directement un ensemble de paragraphes pour construire la spécification.

Chaque paragraphe représente une entité Z de la spécification, qui peut correspondre à :

- Des types de base,
- Un axiome,
- Un «schéma» (générique ou non),
- Une définition d'opérateur,
- Un type dit «libre» (ou *FreeType*, par opposition aux types de base),
- Une conjonction.

Types de base : On utilise la notation suivante pour définir un type de base :

$$[TYPE1, TYPE2]$$

Un type est, sémantiquement, un ensemble d'objets non vide. Cet ensemble peut être aussi bien fini qu'infini. La représentation réelle d'un objet de l'ensemble n'est pas considérée, car cela permet de spécifier rigoureusement les contraintes d'un système quelle que soit son implantation : un objet est donc vu comme une entité typée et abstraite.

La notion de type permet, lors de l'analyse d'une expression, de savoir si elle est cohérente. Ainsi, une expression de la forme :

$$(1, 2) = \{1, 2, 3\}$$

n'est pas cohérente, car le membre gauche est une paire ordonnée, tandis que le membre droit est un ensemble. Ce processus de vérification de type est appliqué pour toutes les expressions de la spécification.

Les axiomes : Ils sont notés de la façon suivante [4] :

Partie Declarative

Predicats

Cette construction permet d'introduire des variables globales dans la spécification. En utilisant la partie «Prédicats», on peut éventuellement en contraindre les valeurs (les prédicats étant optionnels).

4. Les accents français ne passent malheureusement pas dans les environnement LaTeXpour Z

Les schémas : Ils se représentent ainsi :

NomDuSchema
Declarations

Predicats

Ils permettent de regrouper structurellement des variables dans un espace d'état, à la manière des approches orientées objet où les classes permettent de regrouper les attributs d'un même élément.

Un schéma est défini comme l'ensemble des liaisons, ou *bindings* satisfaisant l'ensemble des prédicats de la partie «Prédicats». Il existe une relation de conjonction implicite entre tous les prédicats présents dans un schéma.

Définition 5.1 *(Binding) Fonction finie associant des valeurs à des identifiants (noms).*

Par exemple, considérant le schéma suivant :

*S*1
$a, b : \mathbb{Z}$

$a \geq 0$
$b \leq 4$

Le *binding* $\langle\!| a == 0, b == 0 |\!\rangle$ est un des *bindings* de *S1*.
Il est à noter qu'un schéma peut être paramétré (dépendant d'un ou plusieurs types) :

NomDuSchema[*Parametres*]
Declarations

Predicats

L'opérateur θ permet d'écrire des expressions de construction de *bindings*. L'expression θS, où S est nécessairement un schéma, produit les associations entre les variables de S et leur valeurs respectives.

On l'utilise classiquement dans des expressions comme $\theta S = \theta S'$, permettant de spécifier que dans les bindings θS et $\theta S'$:

- S et S' ont le même ensemble de variables ;
- quelles que soient les variable de S, celles de S' portent le même nom (à la décoration près) et sont associées à la même valeur.

Les types «libres» : Ils permettent de définir des types plus complexes, à la manière d'une BNF (*Backus-Naur Form* [Bac59]) :

$$NomDuType ::= Branche1 \mid ... \mid BrancheN$$

$$Branche1 ::= NomDeLaBranche[\langle\langle Expression \rangle\rangle]$$

On peut par exemple exprimer un type «arbre binaire dont les feuilles sont des entiers» comme suit :

$$ArbreBin ::= branche\langle\langle ArbreBin \times ArbreBin \rangle\rangle \mid feuille\langle\langle \mathbb{Z} \rangle\rangle$$

5.2.2 Systèmes séquentiels : opérations, compositions

On peut définir différentes opérations accédant à l'état des schémas précédemment décrits.
Nous introduisons la définition suivante :

Définition 5.2

(Précondition) Conditions à satisfaire pour que l'opération puisse être appliquée.
(Postcondition) Modifications apportées à l'espace d'états accédé par l'opération, résultant de son application.

Les opérations sur les schémas sont elles-mêmes définies comme des schémas, liant par des relations les états dits «pre» d'un schéma S (avant application de l'opération) à ses états «post» (après application de l'opération).

Un premier ensemble de prédicats définit les préconditions de l'opération (mettant en jeu les états «pre» de S), tandis qu'un second définit ses postconditions (mettant en jeu les états «post» de S). Par convention en Z, on note l'état «post» de S en décorant S avec une apostrophe : *S'*. Ce schéma S' est le schéma S dont les variables sont décorées (notées a' et b', toujours par convention).

Ces schémas-opération peuvent avoir des paramètres en entrée et en sortie. Les entrées sont des variables postfixées par un point d'interrogation «?», et les sorties sont notées comme des variables postfixées d'un point d'exclamation «!».

Le schéma suivant illustre cette notation :

Operation

ΔS

$i? : \mathbb{Z}$

$i? \in \mathbb{N}$ (1)

$a' = a + i?$ (2)

$b' = \textbf{if } i? \leq 4 \textbf{ then } i? \textbf{ else } b$ (3)

Δ (resp. Ξ) permet d'indiquer si l'opération modifie (resp. laisse inchangé) l'espace d'état défini par le schéma *S* : nous pouvons établir un parallèle avec les conventions de nommage des langages orientés objets, où les *setters* (resp. les *getters*) modifient (resp. consultent) la classe à laquelle ils accèdent.

Le prédicat (1) est la précondition de l'opération : elle n'est applicable que si $i? \in \mathbb{N}$.

Z permet d'accéder aux préconditions d'un schéma avec l'opérateur unaire pre .

Définition 5.3 *(Précondition) [Spi89b]*
Soit S un schéma, $x'_1, ..., x'_n$ les composants de S qui sont décorés d'une apostrophe ', et $y_1!, ..., y_n!$ les composants de S qui sont décorés d'un point d'exclamation !, alors le schéma pre *S est produit en cachant ces variables de S :*
$\mathrm{pre}\ S = S \setminus (x'_1, ..., x'_n, y_1!, ..., y_n!)$
Il contient uniquement les composants de S correspondant à l'état avant opération et aux variables d'entrée.

Ici, nous avons : pre *Operation* $= (i? \in \mathbb{N})$.

Les prédicats (2) et (3) sont les postconditions de l'opération, mettant en jeu les variables de l'état «post» de S (a', b').

Séquençage et composition : La notation Z permet le séquençage de schémas-opération, afin de définir des séquences d'opérations complexes. Ceci est réalisé en introduisant l'opérateur '$\fatsemi$', comparable à celui des langages de script comme Shell sous Unix :

$$OpSeq \mathrel{\widehat=} Op1 \fatsemi \ldots \fatsemi OpN$$

Imaginons que ces opérations modifient l'état d'un schéma S. Alors, l'état de S résultant de Op1 (donc «post») devient l'état avant modification de Op2 (donc son état «pre»). Ceci est masqué par un mécanisme de renommage ; pour deux opérations, par exemple, cela revient à :

$Op1$

$S(pre)$

$S''(post)$

$Op2$

$S''(pre)$

$S'(post)$

Alors

$OpSeq$

$S(pre)$

$S'(post)$

On voit que le mécanisme de renommage (S" au lieu de S' dans Op1, S" au lieu de S dans Op2) est transparent.

Comme pour les langages de script comme Shell, il existe un opérateur permettant de combiner les schémas opérations en indiquant que le résultat d'une commande est l'entrée de la suivante (*piping*). En Shell, cet opérateur est 'l', tandis qu'en Z, il s'agit de $\gg$:

$$OpPiping \mathrel{\widehat=} Op1 \gg \ldots \gg OpN$$

Dans ce cas, les mécanismes mis en place sont les mêmes que précédemment, sauf qu'en plus, si un schéma en amont dans la séquence a une variable en sortie ($x!$), et que le suivant a une variable en entrée de même nom ($x?$), alors ces deux variables doivent être de même type : la variable $x!$ sera en effet prise comme variable d'entrée dans le second schéma opération.

Le passage de paramètres en Z se fait en utilisant leur nom, et non leur position comme dans les approches classiques de langage de programmation.

Avec Z, nous ne pouvons donc en théorie que spécifier des systèmes séquentiels. Il est cependant possible de spécifier des systèmes concurrents (comme le montre [Eva94]). Dans la spécification en Z de tels systèmes, les canaux de communication, les processus, les ressources partagées sont spécifiés comme des espaces d'états, modifiés par les opérations y accédant (envoi ou réception de messages, écriture ou lecture dans un buffer).

Nous nous intéressons dans le cadre de nos travaux à la spécification de l'intergiciel PolyORB. Les travaux de [Hug05] ont montré que la quasi-totalité des éléments de PolyORB peuvent interagir en utilisant uniquement des mutex. Seule la partie noyau de l'intergiciel est plus complexe : comme elle a été fait l'objet d'analyses lors de sa modélisation en réseaux de Petri, nous sommes en mesure de dire que la notation Z offre donc tous les mécanismes dont nous avons besoin pour l'architecture que nous spécifions, ainsi que pour les propriétés que nous souhaitons vérifier (respect d'invariants, composition d'interfaces).

5.2.3 Outils

Il existe plusieurs outils permettant d'exploiter une spécification en notation Z. Les outils historiques sont FuZZ [Spi] et Z/Eves [MS97]. Ce sont ceux que nous avons utilisés dans le cadre de nos travaux. Nous allons les présenter ici.

Une spécification Z est la plupart du temps écrite en LaTeX : les outils permettant d'analyser une spécification prennent donc en entrée un fichier `.tex`.

Depuis le standard de fait écrit par J. M. Spivey jusqu'à la norme ISO/IEC, les commandes LaTeX sont standardisées, de façon à produire une spécification qui respecte la notation (symboles, schémas).

FuZZ : vérificateur de types et de cohérence

Introduit par J. M. Spivey, cet outil permet de vérifier la cohérence de typage et de syntaxe d'une spécification. Il est utile pour une première phase de vérification de la spécification. On peut y mélanger spécification formelle et texte explicatif en langage naturel : FuZZ ne prendra en compte que les éléments Z (à l'aide des commandes LaTeX déjà mentionnées).

Z/Eves : un prouveur semi-automatique

Z/Eves est en plus un prouveur : il permet d'écrire des théorèmes, sous forme de prédicats et portant sur tout ou partie de la spécification, et de les vérifier. Ce processus de vérification étant souvent complexe, Z/Eves n'est pas un prouveur complètement automatisé : il requiert l'intervention d'experts pour le guider dans le cheminement de la preuve (utilisation de théorèmes et axiomes annexes, désambiguïsation de certains prédicats, restriction de domaines de fonctions ou d'ensembles, etc.). Plusieurs mécanismes sont fournis, comme la réécriture sous forme canonique de prédicats et leur simplification par réduction, réécriture ou réarrangement de prédicats.

5.2.4 Discussion

Un des avantages de Z est la façon dont les invariants sont utilisés tout le long de la spécification. Typiquement, dans une spécification Z, l'auteur précise au départ un état abstrait, contraint par de forts invariants. Puis, il spécifie des opérations qui accèdent à cet état en le modifiant ou en le consultant. Elles doivent respecter les invariants spécifiés, et peuvent produire des effets de bord (paramètres en sortie).

L'utilisation de Z permet de faciliter la spécification d'un système en utilisant différentes vues : une vue locale à chaque acteur, une vue opératoire concernant deux acteurs : la notation permet de moduler et de combiner très facilement des éléments de spécification.

5.3 Eléments d'architecture cruciaux

Nous avons présenté le formalisme que nous allons utiliser pour procéder à la vérification architecturale de l'exécutif AADL.

Les travaux de T. Vergnaud [Ver06], ont mis en valeur le fait que l'exécutif AADL est matérialisé par la spécification AADL. Il gère l'ordonnancement des threads du système et les communications inter-nœuds. Ce type de services est fourni par les intergiciels de haut niveau.

Définition 5.4 *(Intergiciel de haut niveau) Il s'agit d'un intergiciel qui peut prendre en compte directement les modèles de distribution (passage de messages, objets répartis, etc.). A contrario, un intergiciel est dit de bas niveau s'il ne permet que d'envoyer et de recevoir des messages (typiquement une bibliothèque de sockets).*

Procéder à la vérification formelle de l'architecture d'un intergiciel implique d'en comprendre les mécanismes de façon détaillée. Pour cela, il faut étudier et analyser l'architecture ciblée et en isoler chaque élément d'importance, chaque service rendu.

PolyORB [5] a été catégorisé comme un exécutif AADL par les travaux de [Ver06].

Les travaux de T. Quinot [Qui03] ont en effet permis de constuire une architecture supportant plusieurs modèles de répartition. Ces travaux ont tout d'abord été poursuivis par J. Hugues [Hug05], permettant à cette architecture de supporter le temps-réel et l'embarqué, puis par K. Barbaria [Bar08], ajoutant le support de la tolérance aux pannes.

Ces travaux ont donc résulté en un intergiciel modulaire et paramétrique, basé sur des design patterns, permettant de construire une grande majorité des systèmes modélisés par le langage AADL.

Dans [Hug05], cet intergiciel a déjà fait l'objet de vérifications comportementales, facilitant son analyse pour en déduire une spécification formelle.

Nous connaissons donc bien son architecture, que nous allons présenter dans la section suivante.

5.3.1 Etude du Micro-Broker de PolyORB

PolyORB fournit les services nécessaires à la construction d'applications distribuées. Il se conforme à des standards industriels tels que CORBA [MC], Ada DSA («Distributed System Annex» pour Ada), ainsi qu'à des paradigmes de programmation comme les Web Services ou les *MOM* (Middleware orientés messages).

PolyORB est un middleware générique avec une instance par modèle de répartition. En sus d'une implantation traditionnelle de middleware, PolyORB propose une architecture permettant de faire interopérer différents modèles de répartition de façon uniforme.

Ces instances (ou personnalités) sont des vues des services de PolyORB, aussi bien au niveau applicatif («personnalités applicatives») que protocolaire («personnalités protocolaires»). Elles traduisent des vues mutuellement exclusive de l'architecture de PolyORB. La dissociation des personnalités protocolaires et applicatives, de concert avec un support simultané de différentes personnalités au sein d'un même exécutif sont les clefs de la construction d'applications distribuées interopérables.

De fait, PolyORB fournit une interopérabilité de middleware à middleware (*M2M*).

On isole trois principaux composants dans l'architecture de PolyORB (figure 47) :

– les personnalités applicatives
– le cœur «neutre» de l'intergiciel.
– les personnalités protocolaires

Les travaux de [Pau01] ont permis d'isoler les services et mécanismes permettant à l'ingénieur de développer un intergiciel supportant différents modèles de répartitions, différentes qualités de services (temps-réel, tolérance aux pannes). Ces services sont les suivants :

1. **Adressage** : contrôle l'espace d'adressage des objets applicatifs, de façon à les identifier au sein du système distribué. Il crée des références sur les servants de ces objets, qui sont ensuite échangées entre les acteurs du systèmes pour contacter et interagir avec ces objets.
2. **Binding** : regroupe tous les mécanismes permettant d'interagir avec un objet distant au travers d'un objet local. On y trouvera par exemple les piles protocolaires.
3. **Représentation** : transforme depuis (ou vers) une représentation homogène des données, en vue d'un échange à travers le réseau entre des machines et des nœuds hétérogènes. On y trouvera des mécanismes de sérialisation et dé-sérialisation par exemple.

5. http ://libre.adacore.com/polyorb

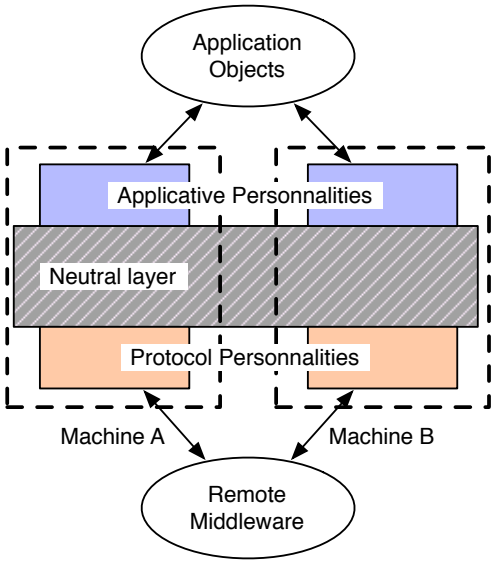


FIGURE 47 – Vue globale de l'architecture de PolyORB

4. **Protocole** : prend en charge tout ce qui est nécessaire à l'établissement d'une session entre deux nœuds. Il travaille de concert avec le service de représentation pour l'empaquetage et le dépaquetage des requêtes.
5. **Transport** : se charge de la réception des messages, de la détection d'évènements survenants sur les différents points d'entrées, à l'aide de moniteurs. Il se charge aussi de la création de points de terminaisons («endpoints» : représentent une connexion établie à la suite du contact entre deux points d'accès), et fournit les procédures pour lire et écrire des données dans ces points de terminaisons, à l'aide de tampons.
6. **Activation** : vérifie qu'il existe bien une correspondance, à la réception d'une requête, entre la référence produite par le service d'adressage et un objet serveur en local.
7. **Execution** : dégage les ressources nécessaires à l'exécution d'une requête reçue : threads, mémoire, etc.

L'architecture de cet intergiciel permet de construire des systèmes aux performances comparables à celles d'intergiciels de référence comme TAO [SGHP97] ou omniORB [Apa], comme les travaux de [Hug05] l'ont montré sur les critères suivants :

– Les performances brutes (temps de traitement des requêtes),
– L'utilisation de la bande passante (requêtes/s, bits/s),
– La latence de communication admissible,
– L'empreinte mémoire.

PolyORB est un intergiciel supporté et utilisé par de nombreux industriels comme AdaCore ou encore EuroControl.

5.3.2 Mise en œuvre du cœur neutre

Le cœur neutre est le chef d'orchestre de l'intergiciel : c'est lui qui va coordonner les services présentés et qui va gérer, au travers de l'adaptateur d'objet (*Object Adapter*), les différents servants (enregistrement, aiguillage de requêtes).

Les travaux de [Hug05] ont permis de réduire et de factoriser les services du cœur en vue de faire de la vérification comportementale. Leur rôle est donc clairement identifié.

La figure 48 résume ces mécanismes, en présentant le cheminement d'une requête au travers du système.

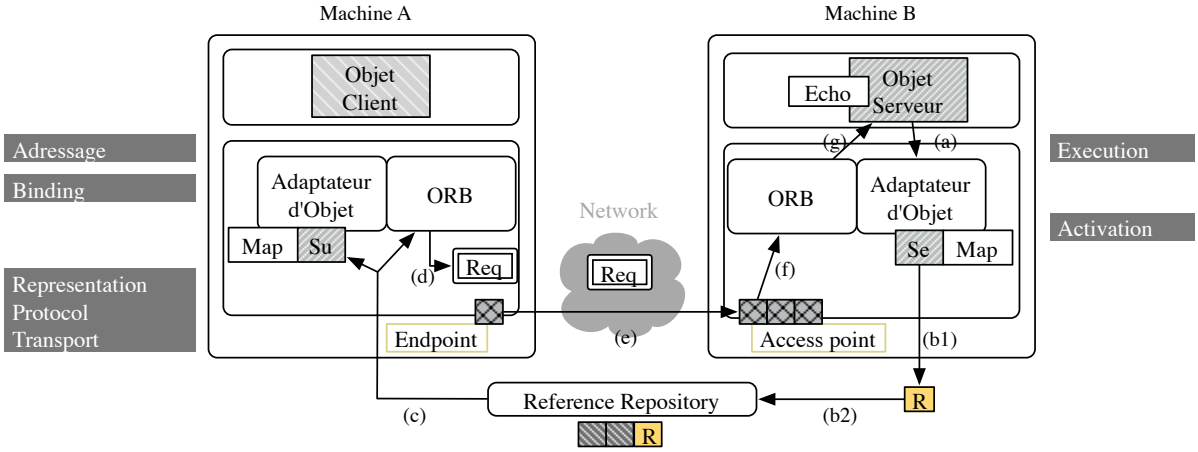


FIGURE 48 – Vue détaillée de l'architecture de PolyORB

Une application serveur, hébergée sur la machine B, possède un objet serveur. Il interagit avec le cœur de l'intergiciel pour qu'il enregistre un servant correspondant à cet objet (a). Ce servant est enregistré auprès de l'adaptateur d'objet. Une référence sur cet objet est alors créée, contenant les informations nécessaires au contact distant avec cet objet (b1) : elle contient un identifiant de nœud (quel nœud contacter), des informations protocolaires (comment contacter ce nœud), ainsi qu'un identifiant d'objet et de méthode (quelle méthode de quel objet sur ce nœud).

Cette référence est ensuite diffusée et partagée (via service de nommage ou par partage direct) avec d'autres nœuds (b2).

Pour cette première phase, les services d'adressage, de liaison et de représentation sont utilisés.

Envoyer une requête n'est possible qu'en possédant une référence sur un objet. L'émetteur de la requête doit donc récupérer une telle référence (c), puis, à partir de celle-ci, forger une requête (d), qui sera envoyée au bon destinataire (e).

Dans cette étape, les services d'adressage, de liaison, de transport et de protocole sont utilisés.

Finalement, à la réception d'une telle requête (f), du côté serveur, celle-ci est analysée, et aiguillée vers le bon servant (g). Puis, les ressources nécessaires à l'exécution sont dégagées.

Les services mis en œuvre sont ceux de représentation, de liaison, d'activation et d'exécution.

5.3.3 Synthèse

Nous avons présenté l'intergiciel PolyORB, identifié comme un exécutif AADL et dont nous avons détaillé l'architecture. Celle-ci s'articule autour d'un cœur neutre orchestrant des services permettant de construire une large classe de systèmes.

Nous allons aborder la modélisation en Z de ces mécanismes d'un point de vue développeur : nous commençons par la définition des types et des structures, puis continuons par la définition des opérations modifiant ces structures. Enfin, nous présentons la spécification des assemblages des différents modules produits ainsi que leur utilisation pour effectuer la vérification formelle.

A l'instar d'un intergiciel qui propose une interface pour être manipulé par le développeur, nous proposons des interfaces pour manipuler notre spécification formelle d'intergiciel.

Notre spécification formelle a en effet pour but d'être manipulée et configurée pour chaque spécification AADL. Nous proposons donc une spécification configurable (en accord avec la spécification AADL) par des mécanismes d'initialisation semblables à ceux du code de l'intergiciel PolyORB. Nous expliquons alors quelles obligations de preuve peuvent être produites et vérifiées, permettant de prouver que la configuration de PolyORB est possible pour l'application AADL et qu'elle ne viole pas les invariants exprimés sur la spécification formelle générique.

5.4 Patrons Z : ORB et types de base

Nous présentons les différents éléments Z de la spécification de l'intergiciel. Cela passe par la spécification des types de base, des types plus complexes et des schémas représentant des éléments cruciaux du système. Il est aussi nécessaire de spécifier les opérations modifiant l'état de ces éléments, ainsi que leurs combinaisons possibles.

Dans un premier temps, nous allons nous concentrer sur la partie dite «serveur» de l'intergiciel, et les mécanismes mis en œuvre pour l'enregistrement des servants, la création et la diffusion de références, la réception de requêtes et enfin leur traitement.

Par convention, nous préfixons les noms (de schémas ou non) introduisant un type Xyz par «T» (TXyz), un schéma opération Op par «op» (opOp).

Nous présentons l'élément central de l'intergiciel, l'ORB, ainsi que les différents éléments qui lui sont attachés, comme l'adaptateur d'objet (*Object Adapter*). Nous définissons chacun de ces composants, et précisons nos choix de modélisation ainsi que les propriétés que ces éléments doivent respecter. Ensuite, nous spécifions les opérations qui modifient l'état d'un ORB, via la création de servant ou encore l'envoi ou la réception de requêtes. Nous présentons ensuite la définition du système global comme étant un ensemble de nœuds sur chacun desquels fonctionne un ORB. Les opérations précédemment spécifiées ne sont pas suffisantes car elles ne permettent pas de cibler spécifiquement un ORB particulier au sein du système, afin d'en modifier l'espace d'états. Nous définissons donc de nouvelles opérations dont la portée est étendue à l'ensemble du système par différents mécanismes, et qui sont basées sur les opérations dont la portée est limitée à un ORB.

Nous terminons par la combinaison de tous ces modules pour construire la spécification finale, attendant d'être configurée par l'utilisateur. Nous abordons enfin les obligations de preuves de la spécification (pour s'assurer que le système respecte les propriétés évoquées précédemment), et les scenarii permettant de procéder à des vérifications plus approfondies du système configuré.

ORB
Dans notre spécification, un schéma représentera l'ORB.

TOrb

$$
\begin{array}{l}
OrbConfiguration \\
TObjectAdapter \\
tsapName : TTsapId \\
requestQueue : \mathrm{seq}\, TRequest \\
tapList : \mathbb{F}\, TTransportAccessPoint \\
memory : TMemory \\
\hline
\#managedServants \leq MaxServant \\
\#requestQueue \leq MaxRequestQueue
\end{array}
$$

Un ORB est assimilé à un ensemble contenant les variables [6] suivantes :

– OrbConfiguration : ce schéma définit un ensemble de variables limitant certains éléments de l'ORB, comme la taille maximale de mémoire à disposition, la capacité maximale en nombre de servant que peut gérer l'adaptateur d'objet, le nombre maximal de requêtes que peut recevoir l'ORB.

OrbConfiguration

$$
\begin{array}{l}
MaxRequestQueue : \mathbb{N} \\
MaxServant : \mathbb{N} \\
MaxMethod : \mathbb{N} \\
MaxMemory : \mathbb{N}
\end{array}
$$

Inclure le schéma OrbConfiguration dans celui de TOrb permet d'inclure ces variables dans tout élément manipulant TOrb. Le regroupement de ces variables dans un schéma distinct permet de limiter à celui-ci l'impact des modifications dues à la configuration de l'ORB.

– TObjectAdapter (OA) : l'adaptateur d'objet associé à l'ORB, qui va gérer les servants. Les travaux de [Ver06] ont montré que la notion d'OA non hiérarchique était suffisante pour décrire un exécutif AADL. Donc même si PolyORB supporte de tels OAs, comme le POA (*Portable Object Adapter*) de CORBA, nous définissons l'adaptateur d'objet comme un schéma contenant une unique variable, la table permettant d'enregistrer des servants :

TObjectAdapter

$$
\begin{array}{l}
managedServants : \mathrm{seq}\, TServant
\end{array}
$$

Les servants gérés sont représentés sous la forme d'une séquence (identifiée par managedServants). La place d'un servant dans la séquence permet de leur attribuer implicitement un identifiant unique. En Z, une séquence est définie formellement comme suit :

$$\mathrm{seq}\, X == \{f : \mathbb{N} \pfun X \mid \exists n : \mathbb{N} \bullet \mathrm{dom} f = 1..n\}$$

Les séquences d'éléments de type X sont des fonctions partielles ($\pfun$) ayant pour domaine de définition les entiers naturels, et pour domaine d'image l'ensemble X. Le domaine de définition de ces fonctions est borné.

6. Nous parlerons aussi d'attributs, à la manière des approches orientées objet

Chaque élément de la séquence est associé à un entier donnant sa position dans la séquence.
Un servant est associé à une paire (objet, identifiant) :

$$TServant == (TObject \times \mathbb{N})$$

Cela met en évidence le fait qu'un servant est une surcouche pour un objet applicatif, et qu'il est identifiable simplement.
On définit un objet comme suit :

$$TMethod == \mathbb{N}$$

$$TObject == \mathbb{F}\, TMethod$$

Un objet est ramené à un ensemble fini de méthodes, elles-mêmes ramenées à un simple entier (à la façon d'un pointeur sur une fonction, qui prend ses valeurs dans $\mathbb{N}$).

– tsapName : l'identifiant de l'ORB au sein du système. Il est utilisé pour contacter le nœud à distance. Nous introduisons le type de base TTsapId comme suit :

$$[TTsapId]$$

Il s'agit d'un ensemble non-vide d'éléments distincts.

– requestQueue : l'ORB reçoit des requêtes qu'il insère dans une file d'attente pour ensuite les traiter. Le type de cet attribut est «séquence» de requêtes.
Nous spécifions le type d'une requête comme suit :

$$TRequest == (TReference \times \mathbb{N})$$

Une requête est une paire contenant une référence sur un objet distant (de type TReference), et un identifiant de méthode (prenant ses valeurs dans $\mathbb{N}$). Une méthode est caractérisée par un identifiant et une signature. Pour les analyses que nous souhaitons effectuer, la signature de la méthode n'a pas d'impact. Elle pourra être prise en compte dans des travaux ultérieurs, par la spécification précise de ses paramètres par exemple.
Nous introduisons donc le type d'une référence et les types associés :

$$TReference == \mathbb{F}\, TProfile$$

Une référence est donc un ensemble fini de profils, dont le rôle est de rassembler toutes les informations nécessaires pour interagir avec un objet distant :

TProfile

$$orbId : TTsapId$$

$$servantId : \mathbb{N}$$

$$protocolStack : TProtocol$$

Les informations utiles que sont l'identifiant du nœud distant, l'identifiant du servant sur ce nœud ainsi que le protocole à utiliser sont présentes.

– tapList : l'ensemble des points d'accès de l'ORB, nécessaires à la réception de requêtes. Il s'agit d'un ensemble fini ($\mathbb{F}$) : par définition, un même point d'accès ne peut (et ne doit) pas être enregistré plus d'une fois auprès de l'ORB.

Un point d'accès est assimilé au protocole qu'il gère. Cette simplification, qui se traduit par un renommage du type *TProtocol* en *TTransportAccessPoint* (comme spécifié ci-dessous), pourra être levée dans le cadre de travaux futurs, qui affineront la spécification. Alors, nous serons en mesure de savoir exactement où ajouter de l'information, le renommage servant de point d'entrée.

$$TTransportAccessPoint == TProtocol$$

Nous définissons le type d'un protocole comme un type libre, où de l'information (décrivant des contraintes) pourra être ajoutée ultérieurement :

$$TProtocol ::= GIOP \mid SOAP \mid ...$$

Nous assimilons dans un premier temps au type TProtocol le protocole ainsi que la pile associée : cela pourra également être raffiné dans des travaux ultérieurs.

– memory : un espace mémoire à disposition de l'ORB, utilisé par le service d'exécution.
Nous assimilons la mémoire à un ensemble fini «d'objets» : nous pourrons ainsi, pour chaque requête, décrire la prise (puis la libération) de mémoire par l'insertion (puis le retrait) de l'objet la demandant dans cet ensemble.
Le type TMemory décrivant la mémoire est donc spécifié ainsi :

$$TMemory == \mathbb{F}\, TObject$$

– Deux prédicats qui sont les invariants que notre système doit respecter : ils indiquent la taille maximale que les séquences liées à l'adaptateur d'objet et à la file des requêtes pendantes ne doivent pas dépasser :
  – $\#managedServants \leq MaxServant$
  – $\#requestQueue \leq MaxRequestQueue$

5.5 Patrons Z : Opérations

Les types caractérisant les éléments de la spécification ayant été introduits, nous pouvons spécifier les opérations accédant à l'état de l'ORB.

5.5.1 Exporter un servant

La première opération que nous allons décrire consiste à exporter un servant : cela revient à enregistrer le servant auprès de l'adaptateur d'objet (étapes 1 et 2 de la figure 48).

La première version de cette opération que nous avons spécifiée était la suivante :

opExportServantOK

$\Delta TOrb$

$inObject? : TObject$

$outServant! : TServant$

$\#managedServants < MaxServant$ (1)

$outServant! = (inObject?, \#managedServants + 1)$ (2)

$managedServants' = managedServants \frown \langle outServant! \rangle$ (3)

L'opération opExportServantOK modifie l'état de l'adaptateur d'objet, et incidemment celui de l'ORB associé ($\Delta TOrb$).

Le paramètre d'entrée est inObject ?, qui est de type TObject : c'est l'objet applicatif que l'on souhaite enregistrer. Cela insère un servant dédié dans l'adaptateur d'objet.

Le paramètre en sortie est outServant !, qui est de type TServant, correspondant au servant créé. Il est utilisé dans la section 5.7 pour effectuer des combinaisons d'opérations.

Cependant, lors de l'utilisation de notre spécification avec un prouveur, nous nous sommes aperçus que si l'adaptateur d'objet est modifié par cette opération, le reste des variables de TOrb ne l'est pas. Cela introduit des ambiguïtés qu'il est trivial de résoudre en indiquant au prouveur, à l'aide des commandes dédiées, que les autres variables ne sont pas changées (par exemple, il suffit, pour une variable a, d'indiquer que $a' = a$).

Dans un soucis de clarté, nous avons pris le parti de spécifier in extenso les éléments de schéma qui restent inchangés : il devient alors inutile de le préciser au prouveur, et la reprise de nos travaux par un tiers en sera facilitée. Nous introduisons alors le schéma OrbOAChanges, inclus dans TOrb, et spécifié comme suit :

OrbOAChanges

$$\Delta TOrb$$

$$memory' = memory$$

$$requestQueue' = requestQueue$$

$$tsapName' = tsapName$$

$$tapList' = tapList$$

$$\theta OrbConfiguration' = \theta OrbConfiguration$$

Seul l'adaptateur d'objet peut être modifié : tout ce qui est explicitement mentionné dans ce schéma **ne le sera pas**.

Nous aboutissons donc à la version finale de notre opération :

opExportServantOK

$$OrbOAChanges$$

$$inObject? : TObject$$

$$outServant! : TServant$$

$$\#managedServants < MaxServant \quad (1)$$

$$managedServants' = managedServants \frown \langle outServant! \rangle \quad (2)$$

$$outServant! = (inObject?, \#managedServants + 1) \quad (3)$$

Analyse des prédicats

(1) Il s'agit d'une précondition. L'opération n'est possible que si la capacité maximale de servants gérés n'est pas atteinte. Le test à effectuer porte sur le cardinal de managedServants, qui est comparé aux valeurs limites spécifiées dans le schéma OrbConfiguration.

(2), (3) : Ce sont les post-conditions de l'opération. Un servant est un couple de type $(TObjet, \mathbb{N})$: l'objet passé en paramètre de l'opération correspond au premier élément du couple, tandis que la place du servant dans la séquence de l'adaptateur d'objet donne la valeur du second.

Le servant est inséré dans managedServants en utilisant l'opérateur $\frown$, qui permet de concaténer deux séquences de même type.

Ce schéma n'est cependant pas suffisant pour décrire l'intégralité de l'opération opExportServant qui peut ne pas être applicable si la précondition (1) n'est pas satisfaite.

Nous le précisons dans le schéma opExportServantERR, qui ne fait que consulter l'espace d'état TOrb (il n'y a pas d'ambiguïté, rendant inutile l'introduction d'un schéma spécifique comme OrbOA-Changes) :

opExportServantERR

$\Xi TOrb$

$inObject? : TObject$

$outStatus! : TStatus$

$\neg (\#managedServants < MaxServant)$

$outStatus! = FAILURE$

Les paramètres en entrée sont identiques au schéma précédent.

Les schémas inclus sont uniquement consultés et non plus modifiés. Le paramètre en sortie, outStatus !, diffère également du précédent schéma. Il est du type TStatus, défini de la façon suivante :

$$TStatus ::= SUCCESS \mid FAILURE$$

Il peut prendre deux valeurs possibles, indiquant si une opération s'est bien déroulée ou non. Traditionnellement, en Z, lorsque l'on décrit une opération, on scinde le schéma correspondant en deux parties, comme nous l'avons fait :

1. Une partie «OK», spécifiant les post-conditions de l'opération si les préconditions sont satisfaites,
2. Une partie «ERR», spécifiant les post-conditions de l'opération si les préconditions de la partie «OK» ne sont pas satisfaites. Il suffit alors d'indiquer comme préconditions de l'opération «ERR» la négation des préconditions de l'opération «OK» :
   $(\text{pre } opERR) = \neg (\text{pre } opOK)$

Ces deux parties sont combinées pour former un schéma opération dit *robuste*.

$$opROBUSTE \mathrel{\widehat=} opOK \lor opERR$$

Définition 5.5 *(Robustesse) Une opération est dite robuste si son comportement est spécifié alors même que toutes ses préconditions ne sont pas satisfaites (évitant les comportements indéfinis).*

Pour conserver une cohérence de paramètres entre la partie «OK» et la partie «ERR» de l'opération, le schéma suivant StSuccess est combiné à la partie «OK» : en effectuant une conjonction des prédicats elle possèdera un paramètre en sortie de type TStatus et de valeur SUCCESS.

StSuccess

$outStatus! : TStatus$

$outStatus! = SUCCESS$

Pour obtenir une opération robuste, les différentes parties de l'opération sont combinées de la façon suivante suivante :

$$opExportServant \mathrel{\widehat=} (opExportServantOK \land StSuccess) \lor opExportServantERR$$

Si les préconditions sont satisfaites, un servant sera créé et le statut final sera SUCCESS. Sinon, le statut final sera FAILURE. Un test de la valeur de la variable outStatus ! indique si l'opération a pu être appliquée ou non.

5.5.2 Créer une référence

Pour contacter le servant précédemment créé et enregistré dans l'adaptateur d'objet, il est nécessaire de construire une référence de type TReference.

L'opération suivante permet de la créer :

opCreateReferenceOK

$$\Xi TOrb$$

$$inServant? : TServant$$

$$outReference! : TReference$$

$$tapList \neq \emptyset \qquad (1)$$

$$outReference! = \{TProfile \mid orbId = tsapName \land servantId = second\ inServant? \land protocolStack \in tapList\} \qquad (2)$$

Cette opération consulte, sans modification, le schéma TOrb. A partir du paramètre d'entrée inServant ?, elle crée une référence outReference !.

Analyse des prédicats

(1) Précondition de l'opération : l'ORB possède des points d'accès permettant de le contacter ;

(2) Une référence est construite comme un ensemble spécifique respectant certaines conditions : on parle de «set comprehension». La référence est un ensemble de profils contenants, pour chaque point d'accès géré par l'ORB l'identifiant de l'ORB courant (c'est à cette fin que son schéma est inclus en consultation), l'identifiant du servant passé en paramètre (la fonction *second* permet de récupérer le second élément d'une paire), et le point d'accès considéré.

Cependant, cette opération peut échouer dans le cas où l'ensemble des points d'accès de l'ORB est vide. Il est nécessaire de spécifier le dual de cette opération, ainsi que la combinaison robuste des deux, afin d'avoir une spécification complète :

opCreateReferenceERR

$$\Xi TOrb$$

$$inServant? : TServant$$

$$outStatus! : TStatus$$

$$\neg\ (tapList \neq \emptyset)$$

$$outStatus! = FAILURE$$

$$opCreateReference \mathrel{\widehat=} (opCreateReferenceOK \land StSuccess) \lor opCreateReferenceERR$$

5.5.3 Réception d'une requête

Lorsqu'un message est reçu, il est analysé pour en extraire la requête correspondante. Cela met en jeu le service de représentation mentionné précédemment.

Dans notre spécification, nous introduisons le type TMessage comme suit :

$$TMessage == TRequest$$

La notion d'empaquetage et de dépaquetage d'une requête pour la transformer en message est introduite par le fait que le type TMessage est un synonyme pour TRequest. Cependant, implicitement, le fait de passer d'un type TMessage à un type TRequest indique que l'on use du service de représentation (prédicat (2) ci-dessous).

L'opération de réception d'une requête a des préconditions que nous allons présenter. Il est sera nécessaire de scinder sa spécification en deux parties, car les préconditions peuvent ne pas être satisfaites :

opReceiveMessageOK

$\Delta TOrb$

$inMessage? : TMessage$

$outRequest! : TRequest$

$\#requestQueue < MaxRequestQueue$ (1)

$outRequest! = inMessage?$ (2)

$requestQueue' = requestQueue \frown \langle outRequest! \rangle$ (3)

Cette opération consiste à introduire dans la séquence représentant la file d'attente de l'ORB la nouvelle requête, à partir du message reçu.

Il y a donc un paramètre en entrée inMessage ?, de type TMessage ; et un paramètre en sortie outRequest ! de type TRequest.

Analyse des prédicats

(1) : Pour que l'opération se déroule correctement, la file de requêtes ne doit pas être pleine.

(2) : Une requête est produite, en utilisant implicitement le service de représentation.

(3) : La requête en question est ajoutée dans la file requestQueue.

Cependant, les préconditions peuvent ne pas être satisfaites :

opReceiveMessageERR

$\Xi TOrb$

$inMessage? : TMessage$

$outStatus! : TStatus$

$\neg (\#requestQueue < MaxRequestQueue)$ (1)

$outStatus! = FAILURE$ (2)

Analyse des prédicats

(1) : La file est pleine. Il s'agit de la négation de la précondition de opReceiveMessageOK.

(2) : L'opération a échouée.

On obtient donc l'opération robuste finale :

$$opReceiveMessage \mathrel{\widehat=} (opReceiveMessageOK \land StSuccess) \lor opReceiveMessageERR$$

5.5.4 Service d'activation

L'ORB a une file de requêtes à traiter. Jusqu'à maintenant, aucune vérification n'a été faite quant à la présence effective dans l'adaptateur d'objet du servant mentionnée dans la requête. Cela est fait par le service d'activation, spécifié comme suit :

opActivationOK

$$
\begin{array}{l}
\Xi TOrb \\
inRequest? : TRequest \\
outObject! : TObject \\
outMethod! : TMethod \\
\hline
\mathrm{dom}\, managedServants \neq \emptyset \quad (1) \\
\land\ first\ inRequest? \neq \emptyset \quad (2) \\
\land\ (\exists p : first\ inRequest? \bullet (p.servantId \in \mathrm{dom}\, managedServants)) \quad (3) \\
\land\ (\exists p : first\ inRequest? \\
\quad \mid p.servantId \in \mathrm{dom}\, managedServants \\
\quad \bullet first\ (managedServants(p.servantId)) \neq \emptyset) \quad (4) \\
\land\ (\exists p : first\ inRequest? \\
\quad \mid p.servantId \in \mathrm{dom}\, managedServants \\
\quad \land first\ (managedServants(p.servantId)) \neq \emptyset \\
\quad \bullet second\ inRequest? \in (first\ (managedServants(p.servantId)))) \quad (5) \\
outMethod! = second\ inRequest? \quad (6) \\
(\exists p : first\ inRequest? \\
\quad \mid p.servantId \in \mathrm{dom}\, managedServants \\
\quad \land first\ (managedServants(p.servantId)) \neq \emptyset \\
\quad \land second\ inRequest? \in (first\ (managedServants(p.servantId))) \\
\quad \bullet outObject! = first\ (managedServants(p.servantId))) \quad (7)
\end{array}
$$

Cette opération consulte les états de l'ORB et donc de l'adaptateur d'objet. Elle prend en paramètre d'entrée une requête inRequest ? de type TRequest ; et produit deux paramètres en sortie, outObject ! et outMethod !, respectivement de type TObject et TMethod.

Analyse des prédicats

(1-5) : Les préconditions nécessaires à cette opération sont que

(1) L'ensemble de servants gérés par l'OA n'est pas vide.

(2) La requête entrante est bien formée et qu'elle contient bien des profils destinés à contacter un servant. L'opérateur *first* permet de sélectionner le premier élément d'un couple d'éléments : *first inRequest*? renvoie l'ensemble des profils de la requête.

(3) Il existe un profil de la requête faisant référence à l'identifiant d'un servant géré par l'OA.

(4) Pour ce servant, l'ensemble des méthodes qu'il propose n'est pas vide.

(5) Parmi ces méthodes, l'une d'entre elles est bien référencée par l'identifiant de méthode de la requête entrante. L'opérateur *second* permet de sélectionner le deuxième élément d'un tuple.

(6,7) : Les effets de bord de l'opération, exprimés par des paramètres en sortie, nommés outMethod ! et outObject !. Ce dernier prend pour valeur un servant référencé par les différents identifiants de la requête entrante. OutMethod ! permet d'exporter l'identifiant de la méthode invoquée chez outObject !.

Comme il y a certaines préconditions à l'exécution de cette opération, il est nécessaire d'introduire la partie «ERR» de celle-ci :

opActivationERR

$$
\begin{array}{l}
\Xi \mathit{TOrb} \\
\mathit{inRequest}? : \mathit{TRequest} \\
\mathit{outStatus}! : \mathit{TStatus} \\
\hline
\neg \; (\\
\mathrm{dom}\, \mathit{managedServants} \neq \emptyset \\
\land \mathit{first}\, \mathit{inRequest}? \neq \emptyset \\
\land \; (\exists p : \mathit{first}\, \mathit{inRequest}? \bullet (p.\mathit{servantId} \in \mathrm{dom}\, \mathit{managedServants})) \\
\land \; (\exists p : \mathit{first}\, \mathit{inRequest}? \\
\quad \mid p.\mathit{servantId} \in \mathrm{dom}\, \mathit{managedServants} \\
\quad \bullet \mathit{first}\, (\mathit{managedServants}(p.\mathit{servantId})) \neq \emptyset) \\
\land \; (\exists p : \mathit{first}\, \mathit{inRequest}? \\
\quad \mid p.\mathit{servantId} \in \mathrm{dom}\, \mathit{managedServants} \\
\quad \land \mathit{first}\, (\mathit{managedServants}(p.\mathit{servantId})) \neq \emptyset \\
\quad \bullet \mathit{second}\, \mathit{inRequest}? \in (\mathit{first}\, (\mathit{managedServants}(p.\mathit{servantId}))))) \qquad (1) \\
\mathit{outStatus}! = \mathit{FAILURE} \qquad (2)
\end{array}
$$

Analyse des prédicats

(1) : Il s'agit de la négation des préconditions de l'opération lorsqu'elle peut être appliquée avec succès.

(2) : Le statut final de l'opération.

On obtient donc au final l'opération robuste suivante :

$$\mathit{opActivation} \mathrel{\widehat{=}} (\mathit{opActivationOK} \land \mathit{StSuccess}) \lor \mathit{opActivationERR}$$

5.5.5 Exécution d'une requête

L'exécution d'une requête se fait en deux étapes : son début, consistant à réserver les ressources nécessaires et à les utiliser ; et sa fin, consistant à libérer ces ressources.

Cela se traduit dans notre spécification, en ne présentant pas une seule opération d'exécution, mais deux : opBeginExecution et opEndExecution.

Début d'exécution
Le début d'une exécution se fait donc pour un objet donné, et nécessite la prise de ressources :

opBeginExecutionOK

$\Delta TOrb$

$inObject? : TObject$

$outObject! : TObject$

$\#memory < MaxMemory$ (1)

$memory' = memory \cup \{inObject?\}$ (2)

$outObject! = inObject?$ (3)

Pour cela, il est nécessaire de consulter l'état de l'ORB (pour la mémoire), et de considérer en paramètre d'entrée un objet inObject ? de type TObjet. Le paramètre de sortie, outObject ! de type TObjet est essentiellement là pour simplifier la combinaison de cette opération avec d'autres dans la suite de la spécification. Il s'agit du même objet que celui passé en entrée.

Analyse des prédicats

(1) : La précondition de cette opération est qu'il faut avoir un espace mémoire disponible pour mener l'exécution correctement.

(2) : La mémoire est modifiée en ajoutant l'objet passé en paramètre. Il est important de noter que comme nous n'avons pas mentionné le fait que notre ORB était multi-tâches, il est implicitement mono tâche quant au traitement des requêtes. Cela se traduit dans le fait que la mémoire est un ensemble fini ($\mathbb{F}$), et que par construction, le même objet ne doit pas être en mémoire deux fois.

(3) : Par simplicité, on produit un objet en sortie identique à celui en entrée.

Comme il y a des préconditions à cette opération, elles peuvent ne pas être remplies :

opBeginExecutionERR

$\Xi TOrb$

$inObject? : TObject$

$outStatus! : TStatus$

$\#memory \geq MaxMemory$

$outStatus! = FAILURE$

Ainsi, si la mémoire est pleine, l'opération échoue.
On obtient donc l'opération robuste suivante :

$$opBeginExecution \mathrel{\widehat=} (opBeginExecutionOK \land StSuccess) \lor opBeginExecutionERR$$

Fin d'exécution
Finir une exécution consiste à libérer les ressources précédemment prises :

opEndExecutionOK

$\Delta TOrb$

$inObject? : TObject$

$inObject? \in memory$ (1)

$memory' = memory \setminus \{inObject?\}$ (2)

Analyse des prédicats

(1) : La précondition de l'opération est que l'objet dont on souhaite indiquer la fin d'exécution est bien présent en mémoire.

(2) : La mémoire est modifiée en retirant l'objet considéré de l'ensemble défini par la variable *memory*.

Les préconditions de l'opération peuvent ne pas être remplies :

opEndExecutionERR

$\Xi TOrb$

$inObject? : TObject$

$outStatus! : TStatus$

$inObject? \notin memory$

$outStatus! = FAILURE$

Cela se traduit par le fait que si l'objet n'est pas en mémoire : alors l'opération échoue.
On obtient au final l'opération robuste :

$$opEndExecution \mathrel{\widehat=} (opEndExecutionOK \land StSuccess) \lor opEndExecutionERR$$

Combinaison
Nous avons donc défini les opérations opBeginExecution et opEndExecution. Si l'on souhaite considérer cela comme une seule opération, on les regroupe en une séquence dans un même schéma. Nous procédons alors comme suit :

$$\begin{aligned} opExecution \mathrel{\widehat=} & opBeginExecution[workingObject!/outObject!] \\ & \gg opEndExecution[workingObject?/inObject?] \end{aligned}$$

On procède par renommage : le paramètre outObject !, du schéma opBeginExecution, est renommé en workingObject !. De même, le paramètre en entrée inObject ? du schéma opEndExecution est renommé en workingObject ?.

Ainsi, en utilisant l'opérateur $\gg$, les deux schémas sont combinés : le paramètre de sortie d'opBeginExecution sera pris comme paramètre d'entrée pour opEndExecution.

5.6 Patrons Z : Système d'ORBs

Pour les opérations modifiant l'état d'un ORB, il est nécessaire d'introduire le système contenant tous les ORBs.

Dans ce système, les ORBs sont identifiés par un nom (de type TTsapId). Il existe également une base de références, fonctionnant à la manière d'un annuaire, qui permet de contacter les objets que les ORBs ont déclarés, avec les protocoles correspondants.

System

$$objectsReferences : \mathbb{F}\, TReference \quad (1)$$

$$orbs : TTsapId \rightarrowtail TOrb \quad (2)$$

Analyse des prédicats

(1) : L'ensemble fini des références du système, dont l'accès permet d'échanger ces référence, renvoyant à la notion de dépôt de références, utilisé par le service de nommage par exemple.

(2) : La fonction injective partielle ($\rightarrowtail$) associant un identifiant TTsapId à un ORB (en effet, tous les identifiants ne sont pas nécessairement utilisé dans une session, et un identifiant renvoie à un seul ORB).

5.6.1 Exporter une référence

Nous avons déjà présenté l'opération permettant de créer une référence. Afin de la partager avec le reste du système, nous spécifions maintenant l'opération permettant d'exporter celle-ci vers l'annuaire accessible à tous et défini dans l'espace d'état du schéma System :

opExportReference

$$\Delta System$$

$$inReference? : TReference$$

$$objectsReferences' = objectsReferences \cup \{inReference?\} \quad (1)$$

$$orbs' = orbs \quad (2)$$

Cette opération modifie le système. Elle prend en paramètre la référence à partager, puis, selon (1), modifie le dépôt de références en y ajoutant celle passée en paramètre. Cependant, comme le précise (2), aucun ORB n'est ajouté ou retiré du système pendant cette opération.

Comme il n'y a pas de préconditions nécessaires au bon déroulement de cette opération, il n'est pas utile de la scinder en deux. Cette opération est robuste par définition.

5.6.2 Créer une requête

Première étape lorsqu'un client souhaite contacter un objet distant, cette opération permet de créer une requête :

opCreateRequest

$$inReference? : TReference$$

$$inMethodId? : \mathbb{N}$$

$$outRequest! : TRequest$$

$$outRequest! = (inReference?, inMethodId?)$$

Ce schéma opération peut être assimilé à une usine (factory) : nul n'est besoin de connaître l'état de l'ORB ou du système pour créer une requête ; si elle est invalide car la référence passée en paramètre est altérée, cela sera détecté par les services de l'ORB concerné.

Cette opération produit donc, à partir de la référence passée en paramètre d'entrée et de l'identifiant de méthode à invoquer, une requête comprenant ces informations. Cette opération est robuste par définition.

5.6.3 Envoi d'un message

Pour envoyer un message, il est nécessaire de le construire à partir d'une requête, puis de l'émettre :

opSendMessageOK

$$\Xi System$$

$$inRequest? : TRequest$$

$$outMessage! : TMessage$$

$$\exists i : \operatorname{dom} orbs \bullet$$
$$\quad \exists p : first\ inRequest? \bullet (p.orbId = (orbs(i)).tsapName)$$
$$\quad\quad \wedge (p.protocolStack \in (orbs(i)).tapList) \qquad (1)$$

$$outMessage! = inRequest? \qquad (2)$$

Cette opération prend en paramètre d'entrée une requête inRequest ? de type TRequest, et produit en sortie un message outMessage ! de type TMessage.

Analyse des prédicats

(1) : La précondition de cette opération est qu'il existe un profil dans la requête qui référence un ORB existant dans le système (via son TTsapId) et que cet ORB est bien joignable par un des protocoles supportés par le profil.

(2) : Implicitement, utilisation du service de représentation pour transformer la requête en message.

Les préconditions de cette opération peuvent ne pas être remplies :

opSendMessageERR

$$\Xi System$$

$$inRequest? : TRequest$$

$$outStatus! : TStatus$$

$$\neg (\exists i : \operatorname{dom} orbs \bullet$$
$$\quad \exists p : first\ inRequest? \bullet (p.orbId = (orbs(i)).tsapName)$$
$$\quad\quad \wedge (p.protocolStack \in (orbs(i)).tapList)) \qquad (1)$$

$$outStatus! = FAILURE \qquad (2)$$

En effet, l'opération peut échouer (2) si aucun profil de la requête ne référence un ORB existant dans le système, ou, s'il existe, elle échoue si l'ORB ciblé ne prend pas en charge le protocole associé (1).

On obtient donc au final l'opération robuste :

$$opSendMessage \mathrel{\widehat=} (opSendMessageOK \wedge StSuccess) \vee opSendMessageERR$$

5.6.4 Configuration de l'ORB

Pour utiliser la spécification dans le but de produire des obligations de preuve, il est nécessaire de la paramétrer. La configuration de l'ORB, qui dépendra des applications considérées, rend cela possible :

opConfigureOrb

$\Delta TOrb$

$inMaxRequestQueue? : \mathbb{N}$

$inMaxServant? : \mathbb{N}$

$inMaxMethod? : \mathbb{N}$

$inMaxMemory? : \mathbb{N}$

$MaxRequestQueue' = inMaxRequestQueue?$

$MaxServant' = inMaxServant?$

$MaxMethod' = inMaxMethod?$

$MaxMemory' = inMaxMemory?$

Ce schéma modifie l'état de l'ORB en affectant les valeurs passées en paramètre aux variables concernées. Il n'y a pas de préconditions pour cette opération, elle est robuste par définition.

5.7 Combinaison des briques Z

Nous avons spécifié les briques nécessaires à la construction d'un intergiciel. Afin de procéder à des vérifications, il est nécessaire de les combiner. Nous montrons comment les combiner, à la manière d'un intergiciel, de façon à proposer une interface de manipulation de notre spécification à l'ingénieur qui souhaite effectuer des analyses.

5.7.1 Contexte des opérations

Les opérations que nous avons précédemment spécifiées ont une portée locale à un ORB particulier. En effet, l'espace d'état auquel elles accèdent est celui de TOrb, et non pas System. Or en l'état, notre spécification ne permet pas de dire qu'une opération (opExportServant par exemple) s'applique à un ORB spécifique du système.

Nous devons définir de nouvelles opérations dont la portée est celle du système (accédant donc à l'espace d'état défini par System), et se basant sur les opérations «locales» précédemment spécifiées.

Ceci est rendu possible en introduisant le schéma Frame (opération robuste, spécifié ci-dessous en deux parties), qui lie le schéma System et le schéma TOrb.

FrameOK

$\Delta System$

$\Delta TOrb$

$i? : TTsapId$

$i? \in \mathrm{dom}\, orbs$ (1)

$orbs(i?) = \theta TOrb \land orbs'(i?) = \theta TOrb'$ (2)

$\forall j : \mathrm{dom}\, orbs \setminus \{i?\} \bullet orbs(j) = orbs'(j)$ (3)

Ce schéma, crucial, va lier les état «pre» et «post» du schéma TOrb et des ORBs effectivement contenus dans le domaine image de la fonction orbs.

Pour cela, ce schéma a un paramètre d'entrée i ?, de type TTsapId, qui permettra de savoir à quel TOrb est appliquée l'opération courante.

Analyse des prédicats

(1) : Précondition de l'opération : i ? doit appartenir au domaine de définition de *orbs*.

(2) : Indique pour l'ORB $orbs(i?)$, que son état «pre» sera celui spécifié par l'état «pre» du schéma TOrb dans les opérations. De la même façon, le «post» de cet ORB ($orbs'(i?)$) sera spécifié par le «post» du schéma TOrb dans ces mêmes opérations.

On lie ainsi les états «pre» (orbs) et «post» (orbs') du schéma System, pour chaque ORB, aux spécifications que nous avons effectuées jusqu'à maintenant et qui mettaient en jeu seulement le schéma TOrb.

(3) : Indique que tout ORB autre que celui passé en paramètre n'est pas affecté par l'opération en cours : son état «pre» est identique à son état «post».

Nous spécifions également l'opération correspondant au cas où les préconditions de Frame ne sont pas satisfaites :

FrameERR

$\Xi System$

$\Xi TOrb$

$i? : TTsapId$

$outStatus! : TStatus$

$\neg\, (i? \in \mathrm{dom}\, orbs)$

$outStatus! = FAILURE$

Et finalement, nous spécifions l'opération robuste correspondante :

$Frame \mathrel{\widehat=} (FrameOK \land StSuccess) \lor FrameERR$

5.7.2 Redéfinition des opérations

En prenant en compte le schéma précédent, il faut redéfinir les opérations qui portent sur un ORB particulier (opérations «locales»). Si cette redéfinition n'est pas effectuée, nous sommes dans l'incapacité

de savoir quel ORB sera modifié suite à l'application d'une opération (puisqu'elles font toutes références à un ORB de façon générique) :

$$Orbs\_opExportServant \mathrel{\widehat=} Frame \land opExportServant$$

$$Orbs\_opCreateReference \mathrel{\widehat=} Frame \land opCreateReference$$

$$Orbs\_opReceiveMessage \mathrel{\widehat=} Frame \land opReceiveMessage$$

$$Orbs\_opActivation \mathrel{\widehat=} Frame \land opActivation$$

$$Orbs\_opExecution \mathrel{\widehat=} Frame \land opExecution$$

$$Orbs\_opConfigureOrb \mathrel{\widehat=} Frame \land opConfigureOrb$$

De cette façon, toutes nos opérations précédemment définies ont un paramètre en entrée en plus, $i?$ qui permet de savoir quel ORB est ciblé par l'opération.

5.8 Obligations de preuves

Les briques de spécification formelles que nous proposons sont paramétriques : ainsi, on peut instancier celle-ci pour différents scenarii correspondants à des cas d'étude particuliers.

Classiquement, il faut s'assurer par exemple que les préconditions de nos opérations sont réalisables (afin que l'opération puisse toujours s'exécuter). On peut ensuite vérifier pour une configuration donnée de l'ORB que les invariants de sont pas violés.

Cette partie d'utilisation de la spécification ne peut être automatisée. Cependant, un ensemble d'obligation de preuves peut être déduit et préparé systématiquement. Le reste dépend intrinsèquement du cas d'étude : cela implique d'avoir de bonnes connaissances en spécification formelle et en preuves.

5.8.1 Syntaxe, typage et validation de domaine

De façon classique, une spécification Z doit être cohérente

1. Au niveau de la syntaxe ;
2. Au niveau du typage : cohérence des types pour les expressions de chaque élément de la spécification ;
3. Au niveau du domaine d'application des fonctions et opérations : pour chaque schéma de la spécification, il faut s'assurer que chaque expression :
   - n'applique une fonction avec un domaine de définition partiel à un élément qui ne serait pas dans ce domaine ;
   - n'utilise pas de façon impropre la définition d'une fonction.

Les deux premiers points sont automatiquement vérifiés par la plupart des outils traitant une spécification Z, en particulier par Z-Eves, que nous avons utilisé pour procéder à nos analyses.

Pour le troisième point, Z-Eves génère automatiquement, pour chaque paragraphe de la spécification, une «condition de domaine».

Nous avons prouvé la validité de chacune de ces conditions générées pour les paragraphes de notre spécification.

Notre intergiciel formel est donc syntaxiquement correct, correctement typé, et cohérent sur les domaines d'application des fonctions.

5.8.2 Préconditions

Les préconditions d'une opération sont l'ensemble des états pour lesquels le résultat de cette opération est défini et cohérent. On peut exprimer ces préconditions via un l'opérateur unaire Z dédié : $\mathrm{pre}\ Operation$(pour un schéma nommé *Operation*).

Nous les avons vérifié pour chaque opération de notre spécification : pour prouver les préconditions d'une opération robuste, nous vérifions les préconditions de la partie «OK» de l'opération, ainsi que celles de la partie «ERR». Nous utilisons ces résultats pour vérifier les préconditions de l'opération robuste.

Tous les théorèmes présentés dans l'annexe C ont été prouvés, et ne sont plus à la charge de l'utilisateur final, qui peut les intégrer dans des preuves plus complexes si nécessaire.

5.8.3 Initialisation des schémas

Nous avons décrit l'exécutif sous la forme d'états qui évoluent selon les opérations qui les manipulent, selon certains invariants qui doivent être respectés dans n'importe quel état valide dudit système.

Si l'état du système (via les schémas *System, TOrb, TObjectAdapter*) contient une incohérence, alors il est impossible de satisfaire les prérequis, et, a fortiori, cet état n'existe pas.

Il faut donc prouver que notre spécification est valide en ce sens qu'il existe au moins un état initial respectant l'ensemble des contraintes spécifiées. Traditionnellement, dans une spécification Z, cela se fait de la manière suivante :

1. On considère un schéma définissant un espace d'état «Etat».
2. On définit un schéma opération «EtatInit» qui caractérise l'état initial de «Etat».
3. On essaie de prouver que
   $\exists\, Etat' \bullet EtatInit$,
   où «Etat'» est l'état après l'initialisation.

Si l'on est en mesure de mener à bien chacune de ces étapes, alors nous sommes en mesure de prouver qu'un état initial existe.

Considérons le schéma *TObjectAdapter* :

TObjectAdapter

$$managedServants : \mathrm{seq}\ TServant$$

Un schéma initialisant TObjectAdapter est le suivant :

TObjectAdapterInit

$$TObjectAdapter'$$

$$managedServants' = \langle\,\rangle$$

qui spécifie qu'à l'initialisation, la liste des objets gérés est vide.

On cherche donc maintenant à prouver le théorème d'initialisation suivant :

theorem tTObjectAdapterInit
$\exists\, TObjectAdapter' \bullet TObjectAdapterInit$

qui est, après traitement par Z-Eves, vrai.

Il existe donc un état initial pour tout objet de type TObjectAdapter.

Ce processus doit être répété pour TOrb et System, nous assurant que des états initiaux valides existent.

5.8.4 Invariants du système

Soit I un invariant que l'on souhaite vérifier dans le système. I sera respecté par une opération O si, en considérant Init comme l'état initial de notre système, on a :

$$(Init \Rightarrow I) \land ((O \land I) \Rightarrow I')$$

Ainsi, de façon informelle, cela signifie qu'il faut s'assurer que

1. L'état initial du système respecte l'invariant ciblé ;
2. Si l'invariant est respecté avant application de O, alors il est encore respecté après (décoration des éléments de I indiquant une post-condition) ;
3. Les deux points précédents sont valides en même temps.

Nous proposons de telles obligations de preuves pour les invariants exprimés dans les schémas types (TOrb, TObjectAdapter) sur le système pour chaque opération. En effet, si les invariants inclus dans les schémas état permettent de s'assurer de façon quasi-automatique que les opérations respectent les invariants en entrée (i.e. avant d'être exécutées), rien n'indique dans la spécification que c'est toujours le cas une fois l'opération déroulée. C'est pourquoi il faut s'en assurer.

Un exemple de d'obligation de preuve est donné ci-après, considérant l'invariant ($I : \#managedServants \leq MaxServant$) :

theorem tInvariantCreateReference
$(InitOrb \Rightarrow (\#managedServants \leq MaxServant))$
$\land ((opCreateReference \land (\#managedServants \leq MaxServant))$
$\Rightarrow (\#managedServants' \leq MaxServant))$

Prouver ce théorème indique qu'après application de l'opération opCreateReference, l'invariant I est toujours respecté.

Les obligations de preuves peuvent être préparées à l'avance comme nous l'avons fait. Cependant leur résolution dépend du système considéré et de sa configuration initiale, impliquant l'utilisation de scenarii les mettant en oeuvre.

Nous présentons cette approche dans la section suivante.

5.9 Approche par scenarii

Il est en effet intuitif de combiner ces briques en suivant un scénario. Un scénario classique concerne l'envoi d'une requête par un ORB client vers un ORB serveur.

Cela se fait en plusieurs étapes :

Mise en place de l'ORB serveur

$$\begin{aligned} ServerSetup \mathrel{\widehat=} & \; Orbs\_opConfigureOrb[server?/i?] \\ & \fatsemi \; Orbs\_opExportServant[server?/i?] \\ & \gg Orbs\_opCreateReference[server?/i?, outServant?/inServant?] \\ & \gg opExportReference[outReference?/inReference?] \end{aligned}$$

Côté serveur, il faut tout d'abord configurer l'ORB. Puis, il faut enregistrer un servant auprès de l'adaptateur d'objet, créer la référence associée et la diffuser.

Alors le serveur est en mesure d'attendre des requêtes pour les traiter.

On note que le mécanisme de renommage est beaucoup utilisé : dans chacune des opérations invoquées, on renomme la variable $i?$, introduite par le schéma Frame, en $server?$. Un renommage est effectué entre les opérations de création et de diffusion d'une référence, afin de permettre l'utilisation de l'opérateur $\gg$.

Mise en place de l'ORB client

$$ClientSetup \mathrel{\widehat=} Orbs\_opConfigureOrb[client?/i?]$$

La mise en place d'un ORB client est beaucoup plus simple : il suffit de le configurer. On note le renommage de i ? en client ?.

Requête Oneway

Une requête dite «oneway» est une requête qui n'attend pas de réponse de la part du serveur. On peut la spécifier en utilisant les briques proposées de la façon suivante :

$$\begin{aligned} RequestLife \mathrel{\widehat=} & ServerSetup \\ & \fatsemi\ ClientSetup \\ & \fatsemi\ Connection \\ & \gg ServerProcess[outRequest?/inRequest?] \end{aligned}$$

Les deux ORBs sont configurés, puis le client envoie une requête qui sera ensuite reçue puis traitée par l'ORB serveur.

Connexion entre l'ORB client et l'ORB serveur

$$\begin{aligned} Connection \mathrel{\widehat=} & opSendMessage \\ & \gg Orbs\_opReceiveMessage[server?/i?, outMessage?/inMessage?] \end{aligned}$$

Il faut enfin connecter nos deux ORBS, en séquençant les opérations d'envoi et de réception d'un message. L'ORB qui reçoit le message étant l'ORB serveur, on renomme i ? en server ? dans le schéma Orbs_opReceiveMessage. On renomme aussi son paramètre d'entrée inMessage ? afin d'implicitement récupérer celui envoyé par le client.

Traitement par l'ORB serveur

$$\begin{aligned} ServerProcess \mathrel{\widehat=} & Orbs\_opActivation[server?/i?] \\ & \gg Orbs\_opExecution[server?/i?, outObject?/inObject?] \end{aligned}$$

On définit ici la séquence d'opérations que devra effectuer un serveur à la réception d'une requête : d'abord l'étape d'activation, puis celle de l'exécution. On renomme i ? en server ?, ainsi que le paramètre d'entrée de Orbs_opExecution afin de pouvoir lui affecter la valeur du paramètre de sortie de Orbs_opActivation.

Un tel scénario est alors analysable à l'aide d'un prouveur, garantissant que l'exécutif configuré est apte à rendre les services spécifiés dans les conditions spécifiées (initialisation, enchaînement d'opérations).

5.10 Synthèse

Dans ce chapitre, nous avons présenté notre approche pour vérifier l'exécutif servant de support à l'application spécifiée en AADL.

Nous avons pour cela sélectionné un exécutif AADL : PolyORB. Nous l'avons analysé, afin d'établir les éléments d'architecture cruciaux nécessitant d'être spécifiés. Nous avons proposé en utilisant la notation formelle Z une spécification formelle générique de cet exécutif, configurable selon les besoins de l'application. Cette spécification a été validée par l'outil Z/Eves, qui a également permis de prouver un ensemble de théorèmes sur cette spécification et présentés en annexe C.

Une première partie de cette spécification correspond aux éléments d'architecture identifiés, définissant les composants du système. La seconde partie de cette spécification formalise les services proposés par PolyORB sous forme d'opérations permettant de manipuler ces composants. Ces opérations sont spécifiées de façon robuste, pour éviter les comportements indéfinis lorsque certaines préconditions ne sont pas satisfaites.

La spécification proposée est instanciable selon la configuration décrite par la spécification AADL, comme nous l'exposons au chapitre 7 (section 7.4). Les interfaces de manipulation proposées permettent de mettre en oeuvre la spécification sur des scenarii issus du contexte de l'application.

Nous avons également établi les obligations de preuves qui doivent être satisfaites pour vérifier que la configuration requise par l'utilisateur est cohérente et ne viole pas les invariants d'architecture spécifiés.

Nous allons maintenant présenter la mise en œuvre technique des précédents chapitres dans une chaîne d'outils. Nous expérimenterons ensuite notre approche sur un cas d'étude permettant de mettre en exergue les différents points abordés tout au long de ce mémoire.

Chapitre 6

Chaîne d'outils mise en œuvre

Contents

Nous avons présenté dans les chapitres précédents qu'il était possible d'exploiter une spécification AADL d'un système TR^{2}E en cours de développement pour l'analyser, en utilisant des méthodes formelles.

Nous avons présenté des patrons de transformation depuis AADL vers les réseaux de Petri pour effectuer des analyses comportementales. Nous avons aussi spécifié un exécutif AADL en Z, afin de le vérifier en fonction de sa configuration.

L'intégration de ces patrons de transformation et de ces règles de vérification dans un processus de développement itératif est un de nos objectifs. Cette intégration passe par la proposition de chaînes d'outils, automatisant autant que faire se peut, les processus de transformation et d'analyse.

Pour l'analyse comportementale, nous nous sommes appuyés sur l'outil Ocarina [VZ06], sur lequel nous revenons à la section 6.1.1. Le fichier AADL au format textuel est analysé par cet outil, qui produit un réseau de Petri dans le formalisme ciblé (coloré ou temporel). Le réseau de Petri produit est ensuite pris en charge par un environnement dédié, selon son type : coloré (CPN-AMI) ou temporel (TINA).

Pour l'analyse de l'exécutif, nous utilisons Z-Eves [MS97]. Après avoir étudié de façon la spécification AADL considérée, nous sommes en mesure de configurer systématiquement et d'exploiter la spécification générique formelle Z proposée au chapitre 5.

La figure 49 propose une vue globale de la chaîne élaborée dans le cadre de cette thèse. En traits épais, les outils ou environnements utilisés. Les rectangles aux coins arrondis sont les formalismes mis en œuvre. Comme le montre cette figure, la spécification AADL du système est le point de départ de toute analyse.

Selon le type de propriétés que l'utilisateur souhaite vérifier, une branche d'outils sera activée :

– Pour l'analyse comportementale, la spécification AADL est traitée par Ocarina pour produire soit des réseaux de Petri temporels, soit des réseaux de Petri colorés.

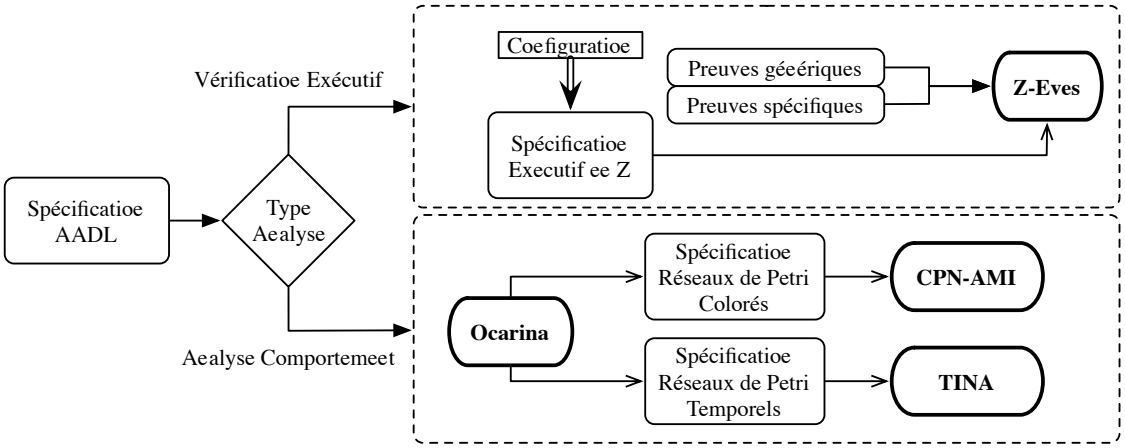


FIGURE 49 – Architecture générale de la solution technique mise en place.

– Pour vérifier l'adéquation de l'exécutif AADL, la spécification Z est paramétrée puis les obligations de preuves sont entrées dans Z-Eves.

Nous présentons dans la suite de ce chapitre chacune de ces branches d'outils.

6.1 Outils : analyse du comportement à l'aide des réseaux de Petri

Cette section est dédiée à la présentation des outils mis en œuvre lors de l'analyse comportementale d'une spécification AADL. La figure 50 indique en détail les outils utilisés.

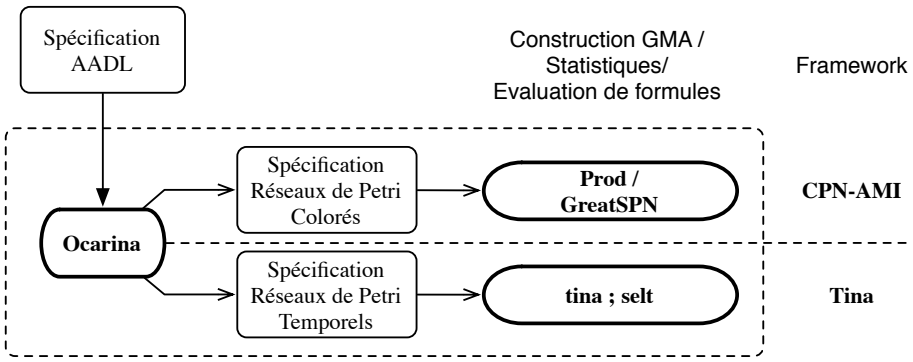


FIGURE 50 – Détail des outils mis en œuvre pour effectuer l'analyse comportementale d'un modèle AADL

6.1.1 Ocarina

Ocarina est un outil développé en Ada par TELECOM-Paristech [HZPK08b]. Il s'agit d'un outil dédié à la manipulation de modèles AADL. L'approche adoptée par Ocarina pour la manipulation d'une spécification AADL est de type compilateur.

Cet outil est en mesure de produire du code exécutable C ou Ada. Des outils tiers comme *Cheddar* permettent de procéder à des analyses d'ordonnancement.

La manipulation d'un modèle AADL avec Ocarina se fait en plusieurs étapes :

1. analyse syntaxique du modèle AADL et construction d'un arbre de syntaxe abstraite (AST) correspondant au modèle
2. passe de vérification sur cet arbre : déclaration des composants, typage, cohérence des attributs
3. instanciation de l'AST en fonction de la spécification AADL : production d'un arbre d'instance contenant toutes les instances de composants du modèle (attributs, connexions, référencement du matériel)
4. manipulation de l'arbre d'instance par les générateurs de code

Les étapes 1 à 3 sont les étapes communes à tous les outils se basant sur Ocarina pour traiter une spécification AADL. L'étape 4 est propre à chacun d'eux : génération de code, analyse de la spécification, analyse d'ordonnancement, etc.

L'arbre d'instance est une structure de données essentielle proposée par Ocarina. Il permet de manipuler les instances de composants d'une spécification. Dans le modèle présenté listing 6.1, un seul type de thread et un seul type de processus sont déclarés. Mais si l'on considère l'implantation du système «S», alors il y aura deux instances de processus «P.impl», chacun contenant deux instances de thread «T.impl».

Listing 6.1 – Exemple de spécification AADL mettant en jeu différentes instances de composants.

```
1  thread T
2  end T;
3
4  process P
5  end P;
6
7  process implementation P.impl
8  subcomponents
9          a : thread T.impl;
10         b : thread T.impl;
11 end P.impl;
12
13 system S
14 end S;
15
16 system implementation S.impl
17 subcomponents
18         p1 : process P.impl;
19         p2 : process P.impl;
20 end S.impl;
```

L'arbre de syntaxe d'Ocarina fera référence à chacun des types déclarés, dans un seul nœud. L'arbre d'instance aura un nœud pour chaque instance de type.

Pour procéder à l'analyse comportementale d'une spécification AADL, nous avons proposé au chapitre 4 des règles de transformation vers les réseaux de Petri. Nous avons donc implanté un générateur de réseaux de Petri dans la partie arrière (*backend*) d'Ocarina.

Notre générateur permet, à partir de l'arbre d'instance, de produire les modèles de réseaux de Petri dans le formalisme ciblé, selon le type de vérification que l'on souhaite effectuer : dans le format CAMI pour les réseaux de Petri colorés, dans le format adapté à TINA pour les temporels.

La figure 51 résume les différentes étapes de ce processus.

Quel que soit le type de réseau de Petri choisi pour procéder à l'analyse comportementale, les étapes sont les mêmes : analyse syntaxique du réseau de Petri considéré, construction et analyse du graphe des marquages accessibles (GMA) par des model-checkers.

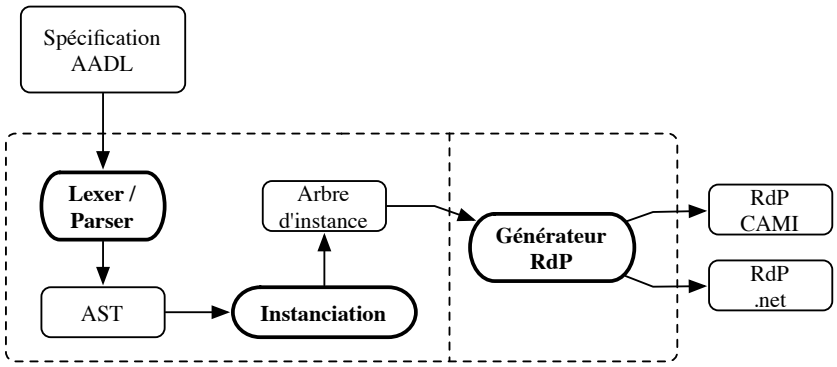


FIGURE 51 – Génération de réseaux de Petri à partir d'une spécification AADL en utilisant Ocarina.

6.1.2 CPN-AMI

CPN-AMI [HHK+06] est une plate-forme dédiée à la construction et à l'analyse des réseaux de Petri colorés.

Elle permet d'interagir avec de nombreux outils, allant de la vérification syntaxique et structurelle de réseaux de Petri, à l'analyse de leurs graphes d'états accessibles, en passant par la vérification structurelle du réseau. Les outils mis en œuvre dans nos travaux sont les suivants :

- *Borne_Place* permet de calculer les bornes d'un réseau de Petri, nous indiquant si des places ont un marquage infini, ce qui pose problème à la fois en terme d'analyse de graphe (pour les états accessibles), et en terme de spécification : le modèle AADL contient des erreurs si un tel cas vient à être rencontré.
- *Prod* nous permet d'effectuer du model-checking sur le graphe des marquages accessibles du réseau de Petri considéré. Cet outil est intéressant quand le GMA est de taille raisonnable, puisqu'il est nécessaire de construire tout le GMA pour ensuite faire de la vérification.
- *GreatSPN*, en combinaison avec d'autres outils du LIP6, permet également de procéder à du model-checking sur le GMA du réseau de Petri. Cependant, il ne lui est pas nécessaire de construire tout le GMA, la vérification peut-être faite à la volée. Par ailleurs, il permet d'utiliser des classes d'équivalence lors de la vérification, se basant sur un GMA symbolique, factorisant des états le permettant.

6.1.3 TINA

TINA est un outil développé au LAAS [LAA]. A l'instar de CPN-AMI, il s'agit d'un environnement dédié à la modélisation et à la vérification de réseau de Petri (place-transition et temporels).

Les outils suivants, intégrés dans cet environnement, ont été utilisés lors de notre mise en œuvre :

- *tina*, qui permet de construire le graphe des marquages accessibles. Il permet de produire différentes abstractions de l'espace d'états des réseaux de Petri temporels, préservant les marquages, les propriétés LTL ou CTL* de l'espace d'états concrets. Il nous donne aussi des statistiques sur le GMA produit.
- *selt*, un model-checker LTL «état/événement». Il raisonne sur des structures de Kripke.

6.1.4 Transformation de modèle : formats

Comme l'a montré la figure 51, notre mise en œuvre produit des réseaux de Petri dans des formats différents, selon le framework utilisé. Le choix du framework (Tina ou CPN-AMI) dépend des propriétés que l'on souhaite vérifier sur le modèle.

PNML Produire un réseau de Petri dans un format d'échange standardisé est une idée séduisante, puisqu'elle implique que nous serions en mesure, à moindre coûts, d'utiliser tous les outils supportant ce format, plutôt que d'utiliser des formats spécifiques.

PNML [BCvH+03], pour «Petri Net Markup Language», est une proposition, en voie de standardisation, d'un format d'échange de réseaux de Petri, basé sur XML. Cependant, si la représentation en PNML des réseaux de Petri colorés va vers une standardistion, la variante temporelle des réseaux de Petri n'est pas encore abordée.

De plus, le nombre d'outils supportant ce format reste réduit. Nous y trouvons Tina [LAA], PNML-Framework [Hil] et Coloane [Vor]. Ces outils permettent d'établir des passerelles vers des plate-formes de traitement et d'analyse des réseaux de Petri. Tina (via son module *ndrio*) propose une passerelle vers une plate-forme de traitement et d'analyse des réseaux de Petri temporels, tandis que Coloane (via PNMLFramework) ouvre vers la plate-forme CPN-AMI, qui est dédiée au traitement et à l'analyse des réseaux de Petri colorés.

Nous prévoyons donc, à terme, de proposer une exportation des réseaux de Petri, colorés ou temporels, vers le format PNML dès que celui-ci sera stable. Cela nous évitera de devoir gérer différents formats de réseaux de Petri, chose que nous avons été amenée à faire pendant ce travail de thèse.

En effet, si l'utilisation de PNML est un choix qui s'inscrit dans la logique de notre thèse (notation standardisée, intégration dans un processus de développement itératif, factorisation des efforts), ce langage reste inachevé. Dans l'attente de sa stabilisation et afin de mettre en œuvre nos règles de transformation, nous avons été dans l'obligation d'utiliser des formalismes dédiés à chaque framework.

CAMI CAMI est le format de représentation textuel de l'environnement CPN-AMI, reposant sur plusieurs mots-clef : ceux qui sont liés au format générique de stockage de données, et ceux spécifiques au formalisme des réseaux de Petri colorés.

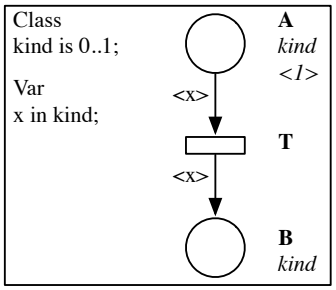
De façon générique, on a :

- *création d'un nœud* : longueur de chaîne de type, type, identifiant unique dans le réseau
  CN(`lng:type,id`**)**
- *définition des attributs d'un nœud* : longueur et nom de l'attibut, identifiant de nœud concerné, longeur et valeur de l'attribut
  CT(`lng_a:attr,id,lng_v:value`**)**
- *création d'un arc entre deux nœud* : type d'arc, identifiant unique, nœud de départ, nœud d'arrivée
  CA(`lng:type,id,from,to`**)**
- *informations liées au réseau* : nom, classes de couleurs, domaines, variables.
  CN(`3:net`**)**

La figure 52 présente à la fois la représentation graphique d'un réseau de Petri coloré, et sa représentation textuelle au format CAMI.

Les lignes 1, 3 et 6 (CN*) indiquent que l'on crée des nœuds : dans chacun des cas, il faut préciser si ce sont des places ou des transitions. Ces nœuds ont un type d'attribut en commun, leur noms qui sont respectivement indiqués lignes 2, 4 et 7. Enfin, des attributs spécifiques aux places (domaine de couleur, marquage initial) sont précisés aux lignes 5 (place B), 8 et 9 (place A).

Les lignes 10 à 13 indiquent comment ces nœuds sont connectés, et, pour chaque arc, quel est sa valuation (lignes 11 et 13).




```
1.  CN(10:transition,4)
2.  CT(4:name,4,1:T)
3.  CN(5:place,3)
4.  CT(4:name,3,1:B)
5.  CT(6:domain,3,4:kind)
6.  CN(5:place,2)
7.  CT(4:name,2,1:A)
8.  CT(6:domain,2,4:kind)
9.  CT(7:marking,2,3:<1>)
10. CA(3:arc,6,4,3)
11. CT(9:valuation,6,3:<x>)
12. CA(3:arc,5,2,4)
13. CT(9:valuation,5,3:<x>)
14. CN(3:net,1)
15. CM(11:declaration,1,1,-35,5:Class)
16. CM(11:declaration,1,2,2,13:kind is 0..1;)
17. CM(11:declaration,1,3,2,0:)
18. CM(11:declaration,1,4,2,3:Var)
19. CM(11:declaration,1,5,3,10:x in kind;)
```

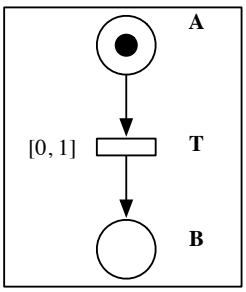
FIGURE 52 – Représentation graphique et textuelle au format CAMI d'un réseau de Petri coloré

Pour finir, les lignes 14 à 19 permettent de spécifier des informations liées au formalisme des réseaux de Petri colorés : déclaration des domaines de couleurs, de leurs champs de valeur, et des variables qui y sont liées.

.net Le format de représentation textuel et compact utilisé par TINA contient plusieurs mots-clef :

- *nom du réseau*
  net net_file
- *déclaration de transition* : nom, garde, liste de places en précondition, et en post-condition.
  tr transition_name **[**l,h**]** list_pre → list_post
- *déclaration de place* : nom, marquage initial
  pl place_name **(** X **)**, où X est un entier.
- *priorité entre transition* : indique si une transition est plus prioritaire qu'une autre en cas d'égalité lors de la possibilité de tirage
  pr T1 **<** T2, dans le cas où la priorité de T1 est plus faible que celle de T2.

La figure 53 présente à la fois la représentation graphique d'un réseau de Petri temporel, et sa représentation textuelle au format .net.




```
1. tr T [0,1] A -> B
2. pl A (1)
3. net buffer
```

FIGURE 53 – Représentation graphique et textuelle au format .net d'un réseau de Petri temporel

La ligne 1 indique la création d'une transition. On trouve sur la même ligne les informations relatives à cette transition (nom : T, intervalle de temps associé : $[0, 1]$), ainsi que les places auxquelles elle est connectée en entrée (B) et en sortie (A).

La ligne 2 indique qu'une place A de marquage 1 existe. On note que la déclaration de la place B n'est pas explicite : en effet, comme la seule déclaration qui lui est associée est son nom, son apparition dans la liste des places en sortie de T suffit à la déclarer dans le modèle. Elle aurait eu une ligne dédiée si nous avions eu besoin d'exprimer un marquage particulier pour cette place.

Enfin, la dernière ligne (3) indique qu'il s'agit du format textuel pris en charge par l'environnement Tina, dans un fichier du nom de *buffer*.

6.1.5 Meta-Modèle mis en œuvre

Ocarina est un outil qui a une approche par compilation pour le traitement des modèles AADL. Il définit le modèle syntaxique d'AADL, ainsi que le méta-modèle du modèle d'instance. Ces méta-modèles sont décrits en IDL [SS89]. Les différentes structures du modèle d'instance, leurs attributs ou listes d'attributs, leur sémantique et leur domaine de valeur sont également exprimés en IDL.

Pour effectuer une transformation de modèles (d'AADL vers un langage λ) dans Ocarina, il faut

1. disposer de l'arbre d'instance
2. avoir exprimé le méta-modèle de λ en IDL : cela permet de décrire la structure de donnée arborescente $\langle A\ \lambda \rangle$ de ce langage
3. parcourir l'arbre d'instance AADL
4. pour chaque élément de cet arbre, suivant des règles de transformation précises, insérer ou compléter un nœud dans l'arbre $\langle A\ \lambda \rangle$
5. parcourir $\langle A\ \lambda \rangle$ pour produire (par impression par exemple), un modèle dans le langage λ

Nous avons donc défini un méta-modèle, au format IDL, pour notre générateur de réseaux de Petri. Il se base sur une analyse des formats CAMI et .net présentés plus tôt. Nos objectifs lors de sa création étaient les suivants :

- la factorisation des structures de données
- la factorisation du code manipulant ces structures
- la possibilité d'étendre à moindre coût notre solution à un formalisme supplémentaire (réseaux de Petri stochastiques par exemple)

Nous avons spécifié des nœuds :

- liés à AADL, pour centraliser des informations particulières : attributs liés à un thread (ports, périodicité, séquences d'appel). Ces informations existent dans l'arbre d'instance, mais ne sont pas groupées selon cette logique
- indépendant de tout formalisme de réseaux de Petri : structure de base d'une place, d'une transition, d'un arc, etc.
- attachés à un formalisme de réseaux de Petri particulier (temporels, colorés) : déclaration de domaines de couleurs, intervalles de temps pour les transitions, etc.

Nous avons en outre pris une approche hiérarchique dans notre méta-modèle, de façon à garder le plus longtemps possible de telles informations sur des composants AADL dans leur version réseau de Petri : à l'heure actuelle, nos outils ne tirent pas complètement parti des réseaux de Petri hiérarchiques [Buc94], mais de récents travaux vont en ce sens [TMH08].

Nœuds liés aux structures AADL Pour chacun des éléments AADL ci-après, nous introduisons un type de nœud spécifique dans le méta-modèle dédié à la transformation d'AADL vers les réseaux de Petri :

- les threads : les informations à considérer sont la liste des ports AADL en entrée et en sortie, les séquences d'appels associées à ce thread, ainsi que l'identifiant du nœud correspondant dans l'arbre d'instance AADL : cela permet de pouvoir éventuellement retrouver des informations annexes qu'il est inutile d'avoir en doublon dans l'arbre de réseaux de Petri (comme le type de dispatch du thread par exemple)
- les ports : nous utilisons un composant contenant un couple d'identifiants de l'arbre d'instance AADL, correspondants aux identifiants des deux threads liés par ce port.
  Les ports sont spécialisés en ports de données et en ports de données-évenement.
- les séquences d'appels sont conservées sous forme de liste, contenant des listes de sous-programmes, ayant leurs propres listes de paramètres en entrée et en sortie

Le listing 6.2 présente les extraits du méta-modèle mettant en œuvre les éléments précédemment évoqués :

Listing 6.2 – Nœuds liés directement aux composants AADL dans le méta-modèle

```
interface Thread_Pattern : Pn_Component {
    List_Id     In_Ports;           /* give used ports and associated
                                       entrypoints */
    List_Id     Out_Ports;
    Value_Id    Hyperperiod;
    List_Id     Call_Seq;
    Node_Id     Th_Instance;        /* refers to aadl thread instance */
  };

  interface Port_Pattern : Pn_Component {
                                    /* local information */
    Node_Id     Port_Instance;
    Node_Id     Source_Instance;
    Node_Id     Target_Instance;
  };

  interface Data_Port_Pattern : Port_Pattern {
  };

  interface Data_Event_Port_Pattern : Port_Pattern {
    /* local information */
    Value_Id    Queue_Size;

    /* interfaces */
    boolean     Has_CEP;            /* has specific entrypoint */

    boolean     Dispatch_Port;      /* parameter if false */
  };

  interface Call_Sequence_Pattern : Pn_Component {
    List_Id     Spg_Call;
  };

  interface Subprogram_Call_Pattern : Pn_Component {
    List_Id     Param_In;           /* list of in spg_parameter */
    List_Id     Param_Out;          /* list of out spg_parameter */
  };

  interface Spg_Parameter_Pattern : Pn_Component {
    Node_Id     Par_Instance;       /* aadl instance of parameter :
                                       from/to [thread | another spg] */
  };
```

Dans ce listing (ainsi que dans les suivants), l'introduction d'un type de nœud dans l'arbre dédié aux réseaux de Petri se fait par l'utilisation du mot-clef «interface» : il permet de définir une nouvelle structure de données, regroupant logiquement des informations diverses.

Les types suivants ont donc été introduits :

- `Thread_Pattern` : structure de données liée au threads. Contient des références vers ses ports en entrée et en sortie, vers les séquences d'appel qu'ils prennent en charge, vers le nœud relatif dans l'arbre d'instance AADL d'Ocarina. L'attribut Hyperperiod permet d'optimiser le calcul de l'hyperpériode du système, et de mettre à jour sa valeur pour tous les nœuds «threads» dans l'arbre RdP.
- `Port_Pattern` : type de base d'un port AADL. Lie un thread source et un thread cible (nœuds de l'arbre RdP). Fais référence à son équivalent dans l'arbre d'instance AADL (Port_Instance). Il est spécialisé en port de données (Data_Port_Pattern), et en port de données et d'événement (Data_Event_Port_Pattern). Ces spécifications possèdent leurs propres attributs
- `Call_Sequence_Pattern` : la liste des séquences d'appel que peut effectuer un thread lors de son déclenchement. Une séquence met en jeu une succession de sous-programmes (Subprogram_Call_Pattern), qui eux-même nécessitent des paramètres (Spg_Parameter_Pattern)

Nœuds indépendants du formalisme de réseaux de Petri Nous définissons ici les éléments du méta-modèle qui ne seront pas impactés par le formalisme final considéré. Il s'agit de la structure même d'un réseau de Petri.

Le type Nœud (listing 6.3) contient un identifiant unique, et un attribut permettant de lui attribuer un nom respectant les espaces de nommage.

Listing 6.3 – Nœud de base du méta-modèle RdP

```
1  interface Pn_Node : Node_Id {
2    Node_Id      Identifier;      /* Identifier */
3    Node_Id      Scoped_Name;
4  };
```

A partir de ce type, nous pouvons proposer un type pour les trois éléments récurrents d'un réseau de Petri : les places, les transitions, et les arcs (listing 6.4) :

Listing 6.4 – Nœuds d'un réseau de Petri basique dans le méta-modèle

```
1  interface Pn_Arc : Pn_Node {
2    Node_Id      Pn_From;
3    Node_Id      Pn_To;
4  };
5
6  interface Pn_Place : Pn_Node {
7  };
8
9  interface Pn_Transition : Pn_Node {
10   List_Id      Pn_Arcs_In;
11   List_Id      Pn_Arcs_Out;
12 };
```

- `Pn_Arc` : un arc possède deux attributs : un identifiant vers chacun des nœuds à ses extrémités (Pn_From et Pn_To)
- `Pn_Place` : une place n'est à ce stade liée à aucun formalisme, il s'agit donc d'un simple renommage du type Pn_Node

– `Pn_Transition` : une transition est un nœud auquel on ajoute deux attributs, qui sont les listes d'identifiants d'arcs entrants et sortant de la transition considérée. Ce type de structure se retrouve ainsi par exemple dans le format .net de l'environnement TINA

Finalement, nous introduisons la notion de composant de réseaux de Petri, avec le type `Pn_Component` (listing 6.5).

Listing 6.5 – Définition d'un composant dans le méta-modèle RdP en vue d'introduire la hiérarchie

```
1  interface Pn_Component : Pn_Node {  /* has identifier and name */
2    List_Id      Public_Interfaces;    /* transitions to be merged */
3    List_Id      Internal_Transitions; /* private transitions */
4    List_Id      Internal_Places;      /* private places */
5  };
```

Conjointement aux travaux sur les IPN [TMH08] («Instantiable Petri Nets»), nous avons adopté les transitions comme élément d'interface dans un réseau de Petri.

Ainsi, un composant de réseaux de Petri aura, outre les attributs d'un nœud classique, une liste d'identifiants des transitions constituant ses interfaces, et deux listes d'identifiants des places et des transitions internes à ce composant.

Lors de leur composition , seule la liste des interfaces sera utile. Cela permet de proposer différentes implantations d'un même composant de réseau de Petri, tout en gardant les algorithmes d'assemblage et de génération intacts.

D'un point de vue plus général, nous définissons le type de `Pn_Box`, qui n'est autre qu'un ensemble de composants (listing 6.6).

Listing 6.6 – Nœud introduisant le type d'un composant composite dans le méta-modèle RdP

```
1  interface Pn_Box : Pn_Node {
2    List_Id      Pn_Subcomponents;     /* List of Pn_Component */
3    List_Id      Pn_Interconnections;
4  };
```

On y trouve donc la liste des composants d'un réseau de Petri, ainsi que les éléments connectant ces composants entre eux. Les interconnexions considérées peuvent être des composants de type `Pn_Component`, représentant par exemple des bus de la spécification AADL.

Pour terminer la structure de notre réseau de Petri, il convient de supporter différents formalismes. On définit donc le type final du réseau de Petri qui servira à faire de la génération : `Pn_Generated` (listing 6.7).

Listing 6.7 – Nœud racine d'un arbre RdP. Description dans le méta-modèle RdP

```
1  interface Pn_Generated : Node_Id {
2    Node_Id      Pn_Box;                                  /* The Petri Net */
3    Node_Id      Pn_Formalism_Specific_Informations;
4    Value_Id     Formalism;
5  };
```

Ce nœud racine contient :

– `Pn_Box` : un nœud composite étant racine de l'arbre RdP. C'est un nœud de type `Pn_Component`

– `Pn_Formalism_Specific_Informations` : structure contenant des informations propres à une variante de réseau de Petri (coloré, temporel)

– `Formalism` : un énumérateur, permettant de savoir comment interpréter le nœud contenant les informations propres à une variante de réseau de Petri

Nœuds propres aux réseaux de Petri colorés La première chose à définir est la structure contenant les informations propres au formalisme (à la manière du nœud «net» dans le format CAMI). Le listing 6.8 présente cette structure.

Listing 6.8 – Structure de données spécifiques aux réseaux de Petri colorés dans le méta-modèle

```
interface CPN_Specific_Informations : Node_Id {
   Node_Id      Classes;          /* formalism_classes */
   List_Id      Domains;          /* formalism_domains */
   List_Id      Variables;        /* formalism_variables */

   /* specific informations*/
   Value_Id     Threads_Count;
   List_Id      Threads_Ids;      /* aadl_id */
   List_Id      Ports_Ids;        /* aadl_id */
};
```

La première partie de cette structure contient des informations sur les couleurs définies dans le modèle (`Classes`), les tuples de couleurs utilisés (`Domains`), ainsi que les variables apparaissant dans le réseaux (`Variables`).

La seconde partie de cette structure permet, lors de la transformation du modèle AADL, de tenir à jour des variables d'état indispensables pour générer de nouvelles variables, ou encore pour établir les valeurs prises par celles-ci au sein d'une classe de couleur.

Enfin, le listing 6.9 montre les spécialisations des éléments du listing 6.4 dans le cadre des réseaux de Petri colorés.

Listing 6.9 – Spécialisation des structures de données de base pour les réseaux de Petri colorés

```
interface CPN_Place: Pn_Place {
  Node_Id      Domain;           /* class or domain */
  Node_Id      Marking;          /* cpn_marking */
  Value_Id     Nb_T;             /* simple tokens */
};
interface CPN_Transition : Pn_Transition {
  List_Id      Guards;
};
interface CPN_Arc : Pn_Arc {
  List_Id      Valuations;       /* CPN_Marking_Token */
};

/* specific attributes */

interface CPN_Marking_Token : Pn_Node {
};
interface CPN_Marking : Node_Id {
  List_Id      Tokens;           /* marking_token */
};
interface CPN_Transition_Guard : Pn_Node {
  Value_Id     Operator;         /* 0: < | 1: <= | 2: >= | 3: > | 4: = | 5: != */
  Value_Id     Left_Op;
  Value_Id     Right_Op;
};
interface CPN_Arc_Valuation : Pn_Node {
  boolean      Is_Colored;
};
```

Une place `CPN_Place` est définie comme une place de base, à laquelle on ajoute des informations sur son domaine de couleur. Dans les réseaux de Petri colorés, peuvent cohabiter des places ayant un

domaine de couleur, et d'autres n'en ayant pas : chacune d'elle a la possibilité d'avoir un marquage initial, respectivement coloré (`Marking`) ou non (`Nb_T`).

Les transitions des réseaux colorés peuvent avoir des gardes logiques de franchissement, faisant intervenir des variables et des opérateurs conditionnels, pour tester les valeurs des jetons entrants. Nous avons donc défini une telle structure (`CPN_Transition_Guard`), ajoutée comme attribut aux transitions.

Nœuds propre aux réseaux de Petri temporels De façon similaire aux réseaux colorés, nous proposons une structure de données propre à cette variante des réseaux de Petri (listing 6.10).

Listing 6.10 – Structure de données spécifiques aux réseaux de Petri temporels dans le méta-modèle

```
1  interface TPN_Specific_Informations : Node_Id {
2    Value_Id     Th_Number;
3    Value_Id     Hyperperiod;
4  };
```

On remarque que cette structure est plus simple que la précédente : les seules informations que nous gardons ici sont le nombre d'instances de thread dans le système (`Th_Number`), et la valeur courante de l'hyperpériode pour un système (`Hyperperiod`). Cette valeur est mise à jour à chaque traitement d'un nouveau thread.

Nous avons aussi étendu les types de base de réseaux de Petri aux réseaux temporels. Le listing 6.11 présente ces spécialisations.

Listing 6.11 – Spécialisation des structures de données de base pour les réseaux de Petri temporels

```
1  interface TPN_Place: Pn_Place {
2    Value_Id     Tokens_Number;
3  };
4
5  interface TPN_Transition : Pn_Transition {
6    Node_Id      Guard;
7    Value_Id     Priority;
8  };
9
10 interface TPN_Arc : Pn_Arc {
11   Value_Id     Valuation;
12   boolean      is_Priority;        /* yes if between two transitions */
13 };
14
15 /* specific attributes */
16
17 interface TPN_Guard : Node_Id {
18   Value_Id     Lower_Value;
19   Value_Id     Higher_Value;
20   Value_Id     Braces_Mode;        /* 0..3: [,] ; ],[ ; [,[ ; ],] */
21 };
```

Une place `TPN_Place` contient un certain nombre de jetons (`Tokens_Number`) ; les transitions `TPN_Transition` ont également des gardes de franchissement (`Guard`), qui sont liées à des intervalles de temps (`TPN_Guard`).

Puisque nous utilisons les réseaux de Petri temporels *à priorité*, il existe deux types d'arcs : ceux liant une place à une transition ; et ceux liant deux transitions pour indiquer la précédence de l'une sur l'autre. Un tel arc `TPN_Arc` peut également véhiculer plusieurs jetons, dont le nombre est défini par l'attribut `Valuation`.

6.1.6 Extension à d'autres formalismes

Le méta-modèle a été créé pour intégrer différentes variantes de réseaux de Petri. Nous avons déjà intégré les réseaux de Petri colorés et les réseaux de Petri temporels à priorité.

Dans l'hypothèse où nous souhaiterions intégrer un formalisme supplémentaire, comme les réseaux de Petri stochastiques (pour prendre en compte les taux de pannes d'un système par exemple), l'impact sur le méta-modèle serait le suivant :

- insertion d'une structure de donnée propre au formalisme des réseaux de Petri stochastiques
- spécialisation des types de base des réseaux de Petri (places, transitions, arcs), et introduction d'attributs spécifiques (gardes des transitions, valuation d'arcs, ...)

Notre générateur a été écrit également pour permettre l'intégration de nouveaux formalismes à moindre coûts. De ce fait, la création des patrons tel qu'exprimé au chapitre 4 est générique : charge à l'utilisateur de proposer les fonctions et procédures suivantes, spécifiques au formalisme ciblé :

- création de nœuds (place, transition, arc)
- initialisation des nœuds selon les patrons créés (marquage initial pour les ports, les threads, etc.)
- connexion de nœuds

Il doit par ailleurs préciser des procédures, appliquées selon un patron «Visiteur» [BWG08], permettant de faire une passe d'harmonisation sur l'ensemble du réseau, lors de l'assemblage final (ajout d'observateurs par exemple).

Nous avons donc circonscrit au plus près le code lié à un formalisme particulier, afin de maximiser la réutilisation du code existant.

6.2 Outils : analyse de l'exécutif à l'aide de la notation Z

Nous présentons ici les commandes utilisées lors de la manipulation de Z-Eves pour prouver les théorèmes de la spécification Z de l'exécutif AADL.

Z-Eves est un prouveur semi-automatique. Il est possible de le guider en lui envoyant des commandes spécifiques. Chacune d'entre elles a un effet particulier.

Principales commandes mises en œuvre

- Commandes de réduction

  La réduction de prédicats est un processus dans lequel Z-Eves explore chaque formule et sous-formule, et les remplace par des formule logiques équivalentes qu'il considère comme étant plus «simples».

  Il prend en compte dans ce processus toutes les définitions et tous les théorèmes définis comme activés (*enabled*) : afin de gagner en performances, il est en effet conseillé, lors de l'ajout de théorèmes ou de définitions, de ne pas les activer par défaut pour éviter à Z-Eves de devoir traiter un nombre trop élevé de prédicats en même temps.

  L'efficacité de la réduction dépend fortement de l'ordre d'écriture des prédicats, pour le but considéré.

  - `rearrange`
    Dans le cas d'une formule conjonctive ($P_1 \wedge P_2 \wedge ...P_N \Rightarrow G$), l'ordre des prédicats P_1 à P_N est modifié
  - `simplify`
    Permet à Z-Eves de raisonner sur les entiers (additions par exemple), et de vérifier s'il y a des tautologies dans le prédicat considéré
  - `rewrite`

Permet de remplacer tout ou partie de prédicats à l'aide de règles de «réécriture» : de telles règles sont des équivalence du prédicat considéré, mais dans lesquelles n'apparaissent pas de quantificateurs ou d'opérateurs de logique.

- `reduce`
  Cette commande applique successivement, et jusqu'à nullité des effets, les opérations de simplification (*simplify*) et de réécriture (*rewrite*). Si une sous-formule du but est un schéma, ou un élément générique, alors elle est remplacée par sa définition in extenso : la réduction est alors effectuée derechef sur ce résultat.

Il faut noter qu'il existe des commandes complémentaires (`with disabled`, `with enabled` ou encore `with normalization`), qui impactent sur le comportement des commandes de réduction. Elle permettent par exemple d'activer ou de désactiver temporairement certaines parties du prédicat

◦ Gestion des quantificateurs

- `instantiate`
  Permet d'instancier une ou plusieurs variables quantifiées, du type $\forall x, y \in \mathbb{Z}$. Les variables instanciées doivent apparaître à la suite du même quantificateur (*i.e.* dans $\forall x \in \mathbb{N} \bullet \forall y \in \mathbb{Z}$, on ne peut instancier x et y)
- `prenex`
  Cet opérateur permet de modifier un prédicat en retirant les quantificateurs présents ($\exists, \forall$). Les prédicats sont réécrits sous forme conjonctive : $(\exists x : \mathbb{N} \mid x \in S) \Rightarrow G$ devient $x \in \mathbb{N} \land x \in S \Rightarrow G$

◦ Utilisation de théorèmes ou de définitions

Dans sa spécification, l'utilisateur peut s'appuyer sur des théorèmes qu'il a déjà prouvé pour achever la preuve de nouveaux théorèmes (de façon incrémentale). Il peut aussi vouloir utiliser les théorèmes offerts pas la bibliothèque mathématique fournie avec Z-Eves [Saa97].
Les commandes suivantes sont alors utilisées :

- `apply`
  Permet d'appliquer un théorème à une formule spécifiée. Si cela est possible, le théorème sera appliqué à chaque occurrence de la formule
- `invoke`
  Pour un nom donné (de schéma par exemple), apparaissant dans le prédicat en cours de traitement, chaque occurrence de ce nom est remplacé par sa définition complète : dans le cas des schémas par exemple, leur nom est remplacé par l'ensemble des prédicats les constituants (préconditions, post-conditions, ...) sous forme conjonctive
- `use`
  Permet d'appliquer un théorème sur une formule. Les quantificateurs de cette dernière sont retirés, les variables qui leur étaient liées deviennent «libres» : il est alors nécessaire de leur attribuer une valeur par instanciation

◦ Division de la preuve en sous-buts

Dans les cas où le but à atteindre est très complexe, il est alors souvent possible d'atteindre ce but par résultats (preuves) successives, portant sur des sous-ensembles du but initial :

- `cases`
  Cette commande divise le but initial en sous-buts quand cela est possible : le but initial est une conjonction ou une disjonction, un prédicat conditionnel (de type *if ...then...else*), ou encore une implication dont la conclusion est un des cas précédents. Les cas sont numérotés, et on peut passer de l'un à l'autre en utilisant la commande `next`
- `split`
  Lorsqu'elle est appliquée à un prédicat, alors le but initial *G* est transformé en *if P then G else G*.

Cela permet d'analyser le but initial de deux façons différentes : un cas où le prédicat considéré est faux, un autre où il est vrai

- `conjunctive` et son dual `disjunctive`
  Pour ces deux commandes, lorsqu'elles sont invoquées, les implications, les équivalences, et les prédicats conditionnels sont réécrits en utilisant uniquement $\wedge$ et $\vee$. La portée des négations est réduite au minimum. Enfin, dans le cas de la commande `conjunctive`, l'opérateur $\vee$ est distribué selon $\wedge$ ($a \vee (b \wedge c)$ devient $(a \vee b) \wedge (a \vee c)$). `disjunctive` agit de façon duale

Il existe deux commandes particulières, `prove by reduce` et `prove by rewrite`, qui offrent le plus haut degré d'automatisation de preuve dans Z-Eves.

Ces deux commandes effectuent les actions suivantes :

1. gestion des quantificateurs (`prenex`)
2. ordonnancement des prédicats considérés (`rearrange`)
3. substitution des prédicats par leur équivalence (si elle existe)
4. réduction (`reduce`) ou réécriture (`rewrite`) selon la commande

Tactiques de preuve Il n'existe pas de guide de preuves pour traiter n'importe quel cas de figure : c'est en partie cela qui rend difficile l'utilisation des méthodes formelles basées sur la preuve de théorèmes.

Cependant, il existe des patrons de preuves que l'on peut mettre en œuvre afin de simplifier les buts initiaux plus rapidement.

Le problème des commandes automatiques `prove by ...` est que parfois, elles conduisent très vite à des prédicats où il est complexe de trouver un angle d'attaque. En effet, elles peuvent parfois supprimer rapidement un prédicat qui serait utile plus loin dans la preuve, et qui, au moment crucial, n'existe plus.

L'utilisation de ces commandes de façon répétitive pourrait s'écrire de la façon suivant :

$$\textit{while}\,(\textit{changement})\,\textit{do}\{\textit{invoke},\ \textit{rearrange}\ \ \textit{rearrange},\ \textit{equality subsitute},\ \textit{rearrange},\ \textit{rewrite}/\textit{reduce}\}$$

Afin d'être efficace dans nos preuves, nous avons suivi et mis en oeuvre dans la section 7.4 les recommandations suivantes (pour les cas complexes, où une simple application du patron précédent ne fonctionne pas) :

- ne pas utiliser les commandes à haut degré d'automatisation
  En les remplaçant par leurs équivalents de base, cela permet de pouvoir gérer et analyser à chaque étape les prédicats en présence : cela permet de contrôler finement les éléments de notre spécification qui peuvent faire défaut (imprécision de la spécification par exemple)
- limitation des commandes développant implicitement les prédicats (`reduce`, `invoke`).
  Cela permet d'éviter la disparition de prédicats utiles, comme mentionné précédemment. On remplacera avantageusement ces commandes par des utilisations successive de `rewrite` et `simplify`

Davantage de recommandations peuvent être trouvées dans les travaux de [Fre04].

6.3 Synthèse

Nous avons présenté dans ce chapitre les outils utilisés pour procéder à l'analyse comportementale d'une spécification AADL, et à la vérification que l'exécutif sous-jacent est compatible avec la configuration du système.

Dans le cas de l'analyse comportementale, nous passons par l'outil Ocarina, un compilateur AADL. Nous y avons ajouté une partie arrière générant des réseaux de Petri colorés et des réseaux de Petri temporels.

Cette transformation est complètement automatique : l'ingénieur n'a pas besoin de connaissances particulières pour générer ces modèles formels. Lors de cette transformation, des informations de nommage et de hiérarchie sont conservées pour permettre, en cas de détection d'erreur lors de l'analyse, de localiser les éléments introduisant un état fautif dans le système.

Ces réseaux de Petri générés sont ensuite traités par un environnement qui leur est spécifique (CPN-AMI pour les réseaux colorés, Tina pour les réseaux temporels). Une partie de ces traitements est automatisée, à l'aide de scripts : analyse syntaxique des réseaux générés par Ocarina, calcul des bornes du réseau en vu de générer le graphe des marquages accessibles, génération de statistiques sur le GMA (marquages morts par exemple).

Le reste de l'analyse est laissé à la discrétion de l'utilisateur final (formules LTL complexes ou études de points précis du GMA).

Dans le cas de la vérification de la compatibilité de l'exécutif, nous utilisons le prouveur semi-automatique Z-Eves. L'analyse de la spécification que nous proposons au chapitre 5, configurée en fonction de la spécification AADL, n'est pas automatique. Nous proposons cependant des tactiques de preuve permettant de simplifier dans une première approche les preuves à effectuer. Enfin, nous indiquons les possibilités de gestion fine de preuves complexes.

Ces points sont mis en œuvre dans le cadre d'un cas d'étude présenté au chapitre suivant. Nous y appliquons également les règles de transformation du chapitre 4. Les résultats présentés ont été obtenus avec les outils précédemment cités.

Chapitre 7

Application sur un cas d'étude

Contents

Dans ce chapitre, nous illustrons la mise en œuvre de notre processus IDV^2 sur une étude de cas. Nous exploitons les guides méthodologiques du chapitre 3 pour définir les propriétés que nous pouvons analyser à partir des informations d'un modèle AADL. Nous utilisons alors les outils appropriés pour valider le comportement du système à l'aide de réseaux de Petri colorés et temporels (construits suivant les règles présentées au chapitre 4). Ensuite, nous vérifions que l'intergiciel retenu satisfait les exigences de configuration et de services du modèle AADL en utilisant la spécification Z du chapitre 5.

Le chapitre est structuré comme suit : nous présentons tout d'abord le cas d'étude que nous avons choisi pour illustrer notre démarche, ainsi que les propriétés que nous souhaitons vérifier sur celui-ci. Puis nous décrivons ensuite la spécification AADL de ce cas d'étude : en effet, si la notation graphique d'AADL permet d'avoir une vue globale du système, sa notation textuelle permet de montrer précisément les valeurs des attributs des différents composants.

7.1 Description globale

Le cas d'étude que nous considérons est la description d'un robot suiveur de ligne. Il est composé de deux capteurs, d'un module de traitement des données (le contrôleur) et de servomoteurs (asservis par le contrôleur). Ces derniers permettent de contrôler les roues du robot : il existe un servomoteur pour les roues gauches, et un pour les droites.

Le fonctionnement du système est le suivant :

– dans le cas nominal, les deux servomoteurs sont actifs et les roues associées tournent, faisant avancer le robot en ligne droite
– tant que la ligne noire est droite, alors le robot suit celle-ci
– si la ligne noire bifurque sur la gauche ou sur la droite alors

1. un des capteurs passe au-dessus de la ligne noire (le capteur droit si la ligne noir bifurque vers la droite)
2. l'information est capturée
3. le contrôleur en prend connaissance
4. le servomoteur concerné est éteint, faisant pivoter de ce fait le robot (vers la droite si la ligne noire tourne vers la droite)
5. dès que le capteur repasse en zone «blanche», alors le servomoteur concerné est réactivé

La figure 54 présente les éléments du système.

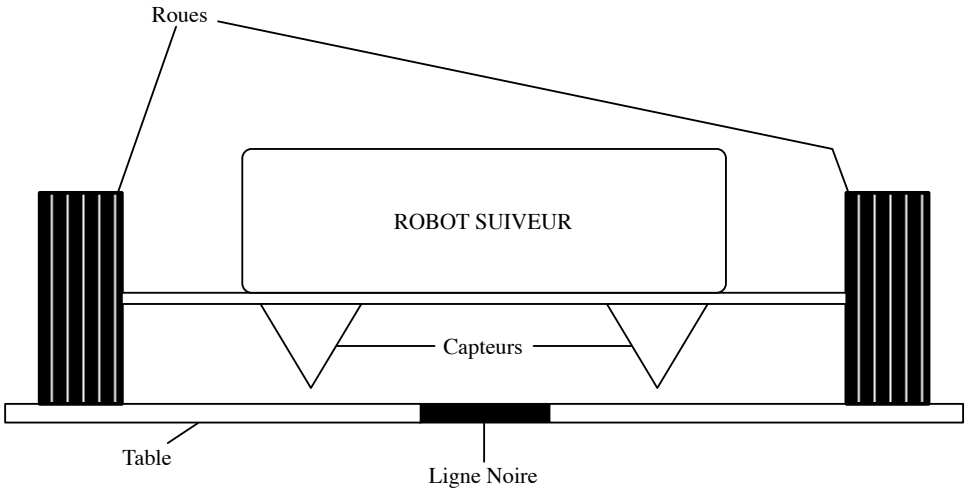


FIGURE 54 – Cas d'étude : robot suiveur (vue de face)

Les réactions du robot face à des événements inattendus de son environnement (une séparation de la ligne en deux par exemple) ne sont pas prises en compte dans le cadre de notre étude : la spécification AADL n'est pas impactée par des variations de réactions, qui sont implantées dans un code à part.

Dans la suite de ce chapitre, nous nous assurons que ce système robotique est correct en terme de comportement : nous validerons son ordonnançabilité, la bonne dimension des files de messages et des autres ressources, le bon fonctionnement des moteurs et du lien qui les lie aux capteurs. Nous vérifions également que la configuration de l'exécutif induite par ce système est bien supportée par PolyORB, et que celui-ci est alors bien en mesure de rendre les services qu'il propose.

7.2 Description semi-formelle : spécification AADL

Effectuer des analyses sur un système permet de le valider. Comme nous l'avons montré dans ce mémoire, il est pour cela nécessaire de passer par une notation pivot, capturant les informations de ce système.

Nous présentons dans cette section la spécification AADL de ce cas d'étude, qui se scinde en plusieurs parties :

- la spécification d'un capteur
- l'implantation d'un capteur
- la spécification d'un servomoteur
- l'implantation d'un servomoteur
- la spécification d'un contrôleur, faisant le lien entre les capteurs et les servomoteurs
- l'implantation du contrôleur
- l'implantation du système final

7.2.1 Les capteurs

Les capteurs sont vus comme des threads périodiques, envoyant des données au contrôleur. Les données permettent au contrôleur de savoir si le capteur survole la ligne noire ou non.

Le listing 7.1 présente les interfaces des composants concernés : le thread `capteur`, ainsi que le processus qui l'englobe. Le type des données échangées dans le système y est aussi introduit.

Listing 7.1 – Code source AADL du thread associé à un capteur du robot (interface)

```
1  -- aadlv2
2  package Robot
3  public
4  with Deployment;
5  with Data_Model;
6
7  ----------
8  -- Data --
9  ----------
10 data Alpha_Type
11 properties
12   Data_Model::Data_Representation => integer;
13 end Alpha_Type;
14
15 ---------------
16 -- Capteur --
17 ---------------
18 -- thread
19
20 thread capteur
21 features
22   evenement    : out event data port Alpha_Type;
23 end capteur;
24
25 -- process
26
27 process p_capteur
28 features
29   evenement    : out event data port Alpha_Type;
30 end p_capteur;
```

Le port événement-donnée du thread et du processus est donc utilisé à chaque fois que le capteur envoie des informations capturées.

L'implantation de ces interfaces est présentée au listing 7.2.

Les données sont collectées via l'exécution d'un sous-programme «collecte_donnee», qui envoie les données via le port associé.

Les données ainsi collectées sont envoyées au contrôleur qui prendra la décision d'arrêter le servomoteur concerné le cas échéant.

Listing 7.2 – Code source AADL du thread associé à un capteur du robot (implantation)

```
1  -- subprograms
2
3  subprogram collecte_donnee
4  features
5    d_source : out parameter Alpha_Type;
6  end collecte_donnee;
7
8  -- thread implementation
9
10 thread implementation capteur.i
11 calls main : {
12   D_Spg : subprogram collecte_donnee;
13 };
14 connections
15   parameter D_Spg.d_source -> evenement;
16 properties
17   Dispatch_Protocol => Periodic;
18   Period            => 110 ms;
19   Compute_Deadline  => 100 ms;
20 end capteur.i;
21
22 -- process implementation
23
24 process implementation p_capteur.i
25 subcomponents
26   th_c : thread capteur.i;
27 connections
28   port th_c.evenement -> evenement;
29 end p_capteur.i;
```

7.2.2 Les servomoteurs

Les servomoteurs sont vus comme des threads apériodiques, recevant des commandes de la part du contrôleur. La réception d'une commande indique qu'un capteur a prévenu le contrôleur du franchissement de la ligne.

Le listing 7.3 présente, de façon semblable aux capteur, les interfaces du composant thread `servomoteur`, ainsi que celles du processus qui le contiendra. Les servomoteurs reçoivent les commandes du contrôleur par leur port «ordre». La réception de ces commandes déclenche le thread concerné.

Listing 7.3 – Spécification AADL du thread rerpésentant les servomoteurs (interface)

```
1  ----------
2  -- Servo --
3  ----------
4
5  -- thread interface
```

```
6
7  thread servomoteur
8  features
9    ordre        : in event data port Alpha_Type;
10 end servomoteur;
11
12 -- process interface
13
14 process p_servomoteur
15 features
16   ordre              : in event data port Alpha_Type;
17 end p_servomoteur;
```

Une implantation de ces interfaces est proposée au listing 7.4.

Lors de la réception de commandes sur le port«ordre», le servomoteur exécute le sous-programme «action», qui arrête le moteur, ou au contraire le remet en marche.

Listing 7.4 – Implantation des composants servomoteurs du cas d'étude

```
1  -- subprogram
2
3  subprogram action
4  features
5    d_action : in parameter Alpha_Type;
6  end action;
7
8  -- thread implementation
9
10 thread implementation servomoteur.i
11 calls main : {
12   A_Spg : subprogram action;
13 };
14 connections
15   parameter ordre -> A_Spg.d_action;
16 properties
17   Dispatch_Protocol => Aperiodic;
18 end servomoteur.i;
19
20 -- process implementation
21
22 process implementation p_servomoteur.i
23 subcomponents
24   th_servomoteur : thread servomoteur.i;
25 connections
26   port ordre -> th_servomoteur.ordre;
27 end p_servomoteur.i;
```

7.2.3 Le contrôleur

Il transfère, en accord avec les informations qu'il reçoit d'un capteur, des commandes de changement d'état au servomoteur associé.

Le contrôleur est modélisé par deux threads sporadiques, dont le déclenchement se fait sur réception d'informations provenant des capteurs. Chaque thread est dédié au traitement des données d'un capteur, et associé au contrôle d'un servomoteur.

L'interface AADL du `contrôleur` est présentée dans le listing 7.5. Les données provenant des capteurs sont reçues par les ports «info_capteur» : il en existe un pour chaque moitié du robot. Les

ordres sont envoyés aux servomoteurs via les ports «comm_servo», eux aussi liés à une partie spécifique du système.

Listing 7.5 – Code source AADL du thread associé au contrôleur du robot (interface)

```
-- thread interface

thread controleur
features
  info_capteur : in event data port Alpha_Type;
  comm_servo   : out event data port Alpha_Type;
end controleur;

-- process interface

process p_controleur
features
  info_capteur_droit  : in event data port Alpha_Type;
  comm_servo_droit    : out event data port Alpha_Type;

  info_capteur_gauche : in event data port Alpha_Type;
  comm_servo_gauche   : out event data port Alpha_Type;
end p_controleur;
```

L'implantation du processus contenant deux instances des threads précédemment spécifiés est proposée aux listing 7.6 et 7.7.

Listing 7.6 – Implantation du thread contrôleur

```
-- thread implementation

thread implementation controleur.i
calls main : {
  T_Spg : subprogram traite;
};
connections
  parameter info_capteur  -> T_Spg.d_info;
  parameter T_Spg.d_ordre -> comm_servo;
properties
  Dispatch_Protocol => Sporadic;
  Period            => 110 ms;
  Compute_Deadline  => 100 ms;
end controleur.i;
```

Listing 7.7 – Implantation du processus contenant le thread contrôleur

```
-- process implementation

process implementation p_controleur.i
subcomponents
  th_ctrl_droit  : thread controleur.i;
  th_ctrl_gauche : thread controleur.i;
connections
  port info_capteur_droit        -> th_ctrl_droit.info_capteur;
  port th_ctrl_droit.comm_servo  -> comm_servo_droit;
  port info_capteur_gauche       -> th_ctrl_gauche.info_capteur;
  port th_ctrl_gauche.comm_servo -> comm_servo_gauche;
end p_controleur.i;
```

Chaque thread gérant un couple $\{interface\_controle, servomoteur\}$ est donc apériodique. Quad il est déclenché (sur la réception d'événements provenant du capteur), il appelle un sous-programme de traitement, qui émet la commande nécessaire. Cette dernière est transmise par un port en sortie.

Ces deux threads sont contenus dans un processus de contrôle. Il s'agit, comme le montre la spécification, de deux instances de la même implantation de thread (leur comportement étant identique).

7.2.4 Le système

Nous proposons maintenant la spécification complète de notre système de robot suiveur. Ce système contient donc deux capteurs (à droite et à gauche de la ligne noire à l'état initial), un contrôleur qui reçoit des données de ces capteurs, les traite, et envoie la commande adaptée aux servomoteurs. Il y a deux servomoteurs, chacun gérant un côté du robot (droite et gauche). Les listings 7.8 et 7.9 présentent la spécification associée.

Listing 7.8 – Spécification AADL finale du système de robot suiveur de ligne (interface)

```
1 -- system interface
2 system robot
3 end robot;
```

Listing 7.9 – Spécification AADL finale du système de robot suiveur de ligne (implantation)

```
1  -- system implementation
2
3  system implementation robot.i
4  subcomponents
5    proc_capteur_droit      : process p_capteur.i;
6    proc_capteur_gauche     : process p_capteur.i;
7
8    proc_controleur         : process p_controleur.i;
9
10   proc_servomoteur_droit  : process p_servomoteur.i;
11   proc_servomoteur_gauche : process p_servomoteur.i;
12
13   CPU1                    : processor the_processor;
14 connections
15   port proc_capteur_droit.evenement
16               -> proc_controleur.info_capteur_droit;
17   port proc_capteur_gauche.evenement
18               -> proc_controleur.info_capteur_gauche;
19
20   port proc_controleur.comm_servo_droit
21               -> proc_servomoteur_droit.ordre;
22   port proc_controleur.comm_servo_gauche
23               -> proc_servomoteur_gauche.ordre;
24 properties
25   actual_processor_binding => (reference (CPU1))
26                           applies to proc_capteur_droit;
27   actual_processor_binding => (reference (CPU1))
28                           applies to proc_capteur_gauche;
29
30   actual_processor_binding => (reference (CPU1))
31                           applies to proc_controleur;
32
33   actual_processor_binding => (reference (CPU1))
34                           applies to proc_servomoteur_droit;
35   actual_processor_binding => (reference (CPU1))
```

```
36                              applies to proc_servomoteur_gauche;
37  end robot.i;
38
39  -- aadlv2
40  end Robot;
```

Il n'y a qu'un seul processeur, «CPU1», dans ce système. Toutes les instances de threads spécifiées y sont liées pour l'exécution.

Nous allons maintenant présenter l'analyse de ce modèle AADL en suivant notre méthodologie. Nous allons aborder en premier lieu l'analyse comportementale, puis nous assurer que l'architecture sous-jacente supporte une telle configuration.

7.3 Vérification comportementale

Nous présentons ici les différentes analyses comportementales que nous sommes en mesure d'effectuer à partir de la spécification AADL du cas d'étude. Pour cela, nous mettons en œuvre les règles de transformations du chapitre 4, via les outils du chapitre 6.

Cette mise en œuvre permet de produire des modèles formels en réseaux de Petri colorés et en réseau de Petri temporels à priorité.

Nous présentons un ensemble de propriétés, exprimées en LTL, qui sont vérifiées pour le cas d'étude considéré. Nous exhibons celles d'entre elles qui peuvent être générées automatiquement.

Nous terminerons sur l'impact sur la spécification AADL des résultats obtenus.

7.3.1 Rappel sur la Logique temporelle linéaire - LTL

La logique *propositionnelle* peut être utilisée pour caractériser un instant.

Considérons deux variables propositionnelles, a et b. On peut alors caractériser les prédicats suivants :

1. $a \wedge b$
2. $a \wedge \neg\, b$
3. $\neg\, a \wedge b$
4. $\neg\, a \wedge \neg\, b$

Cependant, si l'on souhaite exprimer des propriétés de type «P1 précède P2» ou encore «Le système ne vérifiera pas toujours P3», il est nécessaire de faire apparaître la notion de causalité.

En utilisant la logique monadique du premier ordre, les variables propositionnelles sont transformées en des prédicats unaires, paramétrés par le temps. On a donc :

- $a(t), b(t)$: a et b sont définies pour un instant t,
- $t+1$: l'instant successeur immédiat,
- $t \leq u$: un ordre total sur les instants,
- $\exists\, t, \forall\, t$: des quantificateurs du premier ordre.

On peut donc exprimer la propriété «Le système ne vérifiera pas toujours P3» ainsi (avec P3 $= \neg\, a \wedge b$) :

$$\neg\, [\forall\, t \bullet (\neg\, a(t) \wedge b(t))]$$

La logique temporelle linéaire (LTL) est équivalente à la logique monadique du premier ordre à un successeur.

Elle introduit les opérateurs suivants :

Next	X f	f est vraie à l'instant suivant
Always	G f	f est vraie à tout instant
Eventually	F f	f sera vraie à un instant (présent ou futur)
Until	f U g	f est toujours vraie jusqu'à ce que g le soit

C'est cette logique que nous allons utiliser pour procéder à l'analyse de nos réseaux de Petri.

7.3.2 Propriétés comportementales à vérifier

Nous présentons ici les propriétés comportementales du cas d'étude qu'il faut être en mesure de valider :

(P_A) **Au moins un des servomoteurs est actif** Cette propriété permet de s'assurer que lorsque le système est en marche, alors le robot pourra toujours agir pour corriger sa trajectoire.

(P_B) **Lorsqu'un des capteurs détecte le survol de la ligne noire, le servomoteur associé est éteint** Cette propriété permet de s'assurer que le robot ne quittera pas la ligne lorsqu'elle dévie. Il s'agit donc de s'assurer qu'à la réception d'une information depuis un capteur, le contrôleur envoie bien un ordre au servomoteur concerné.

(P_C) **Les tampons de communication sont bien dimensionnés**, aucun message n'est perdu.

(P_D) **Le système est ordonnançable**.

Nous exprimerons dans la suite de ce chapitre ces propriétés en LTL, afin les vérifier sur les modèles formels en utilisant un model-checker.

7.3.3 Réseaux de Petri colorés

Nous abordons dans cette section les analyses liées à la transformation du modèle AADL en réseau de Petri coloré. Elles ont pu être menées grâce à l'environnement de vérification CPN-AMI.

Nous allons procéder aux analyses suivantes :

- Calcul des bornes du modèle. Il est impossible de construire le GMA (définition à la section 4.2.1) d'un réseau de Petri non borné.
- Détection des nœuds terminaux.
- Expression en LTL des propriétés comportementales du système et vérification sur le GMA.

Bornes

Une analyse structurelle peut directement être menée sur le réseau de Petri généré, sans avoir besoin de passer par la construction (totale ou partielle) du GMA.

Définition 7.1 *(Analyse structurelle) Elle repose sur l'algèbre linéaire et permet d'étudier des propriétés d'un réseau (comme les bornes) indépendamment d'un marquage initial.*

Elle se base sur l'équation de franchissement $M' = M + W^* \overline{S}$

Avec

- M' *le marquage résultant de la séquence de franchissement s.*
- M *le marquage du réseau avant la séquence de franchissement s.*
- W *la matrice d'incidence du réseau, avec* $W = W^+ - W^-$ *:*
  On appelle matrice d'incidence avant W^- *la matrice représentant pour chaque transition T, en fonction des places du réseau, le nombre de jetons qu'elle doit consommer pour être franchissable.*
  On appelle matrice d'incidence arrière W^+ *la matrice représentant pour chaque transition T, en fonction des places du réseau, le nombre de jetons qu'elle peut produire après franchissement.*

– *$\overline{S}$ le vecteur caractéristique d'une séquence de franchissement S : c'est un vecteur de dimension m égale au nombre de transition dans le réseau. Sa composante numéro j correspond au nombre de fois où la transition T_j est franchie dans la séquence s.*
Par exemple, si $S = T_2T_4T_1T_4T_2T_4$ alors $\overline{S} = [1, 2, 0, 3]^T$.

Cette analyse est à faire en premier lieu, car elle permet alors de déterminer si la construction du GMA est envisageable ou non. Une réponse négative (*i.e.* un GMA infini) empêche toute analyse plus fine du modèle.

Le calcul effectué reste indicatif : pour avoir les bornes exactes du système, il est nécessaire de procéder à l'exploration complète des états du système. Il donne cependant une première estimation exploitable.

Ces bornes indiquent, pour chaque place, le nombre minimal et le nombre maximal de jetons qu'elle peut contenir. Le nombre minimal est souvent 0. Par contre, selon les cas considérés, la borne supérieure peut être infinie. Dans un contexte de systèmes embarqués temps réels, une telle valeur pour la borne supérieure peut être interprétée comme le fait que dans notre système, un tampon (de requêtes par exemple) est utilisé en dehors de ses capacités. Dans ce cas, la construction du GMA est impossible.

Pour éliminer les faux positifs dus au fait qu'il s'agit d'une estimation, il convient de procéder de nouveau à cette analyse depuis le modèle en réseaux de Petri temporels : les informations temporelles contraignent les possibilités d'exécution du système, et les bornes sont affinées.

L'analyse du réseau de Petri du cas d'étude nous indique qu'il est borné.

La construction du GMA est donc possible, nous permettant de procéder à des analyses plus poussées.

Le tableau 7.1 présente des statistiques sur le nombre d'éléments du réseau de Petri coloré généré automatiquement à l'aide d'Ocarina. Il présente également des statistiques relatives au graphe des marquages accessible associé, généré soit à l'aide de l'outil Prod, soit à l'aide de l'outil GreatSPN. Prod calcule de façon exhaustive l'espace d'état du système, à l'inverse de GreatSPN qui travaille sur des états symboliques (graphe quotient, ou GQ) tels que présentés à la section 4.2.2.

	Réseau de Petri	GMA (Prod)	GQ (gSPN)
Nœuds	38 places 29 transitions	65537 états	1025 états symboliques
Arcs	92	425985	NC

TABLE 7.1 – Statistiques sur le réseau de Petri coloré «basique» et son GMA

Le réseau de Petri généré reste assez petit, mais le nombre d'état engendré est grand, considérant le fait que l'on cible un système où s'exécutent 6 threads. Cela reste cependant très raisonnable pour les model-checker explicites qui sont capables de gérer jusqu'à 10^6 états.

⇒ *Retour au développeur : le système est analysable.*

Nœuds terminaux

La présence d'un nœud terminal dans un GMA indique un blocage dans le système. Un nœud est dit «terminal» s'il représente un marquage du réseau de Petri qui ne peut plus évoluer. Cela traduit le fait qu'une ressource nécessaire à l'exécution du système n'est plus générée, ou bien qu'un interblocage est présent dans le système (attente mutuelle de plusieurs éléments actifs du système).

La détection de tels nœuds se fait par analyse du GMA associé au réseau de Petri. Le GMA montre en effet l'évolution du marquage du réseau de Petri selon les transitions qui sont franchies. Nous avons donc, de façon exhaustive, tous les états potentiellement accessibles par notre réseau de Petri.

Dans le cadre de notre cas d'étude, l'analyse du GMA nous indique qu'il n'existe pas de nœud terminal.

⇒ *Retour au développeur : le système n'est pas sujet à des interblocages.*

Expression des propriétés en LTL

L'analyse du GMA se fait en interrogeant un model-checker avec des formules précises, exprimées en LTL. Ces formules ne sont pas générées automatiquement et sont élaborées par le développeur.

Nous présentons les propriétés que nous souhaitons voir valider, et leur traduction en LTL. Pour valider les propriétés exprimées à la section 7.3.2, nous exprimons les propriétés connexes suivantes :

(P_1) Les capteurs produiront tout le temps des données : notre système sera ainsi tout le temps en mouvement.

(P_2) Lorsqu'un capteur produit une donnée, elle est analysée par le contrôleur.

(P_3) L'activation d'un controleur implique la génération d'une commande pour un servomoteur.

Le réseau de Petri généré (figure 1, annexe B) a pour destination une chaîne d'outils automatiques : il n'est pas construit pour être lisible facilement par un humain. Nous rappelons donc à la figure 55 les patrons de réseaux de Petri colorés mis en œuvre dans ce cas d'étude.

Nous allons exprimer ces propriétés en utilisant la logique LTL :

1. Soit *Push* et *Ovf* les transitions du patron d'un thread capteur permettant de produire une donnée (puisqu'il s'agit d'un port de donnée événement, ceux-ci possèdent deux interfaces en entrée ; et *Begin* la transition du patron d'un thread capteur indiquant que celui-ci a été déclenché :

   $$\mathtt{G}\,((\mathit{Begin} \rightarrow \mathtt{F}\,\mathit{Push}) \;||\; (\mathit{Begin} \rightarrow \mathtt{F}\,\mathit{Ovf}))$$

   soit, en langage naturel : «toujours, le déclenchement du capteur active la transition Push, OU, le déclenchement du capteur active la transition Ovf». En effet, si une de ces deux transitions est franchie, une donnée est produite.

   La validation de la formule $\mathtt{G}\,\mathtt{F}\,\mathit{Begin}$ nous assurera que notre capteur sera toujours déclenché .

2. Soit *Slot* la place contenant la donnée produite par le capteur, et *Work* la place de travail du contrôleur, indiquant qu'il a été déclenché et qu'il a lu la donnée du capteur (franchissement de transition prenant source dans deux places :*Slot* et *Wait_For_Dispatch*, ici *Pop*)

   $$\mathtt{G}\,(\mathit{Slot} \rightarrow \mathtt{F}\,\mathit{Work})$$

   soit, en langage naturel : «toujours, si Slot est marquée (et contient donc une donnée), alors le contrôleur associé passera dans l'état Work». On note cependant que la formule n'exprime rien quant à l'éventuelle perte de données ici.

3. Soit *Pop* la transition du patron d'un thread contrôleur indiquant par son franchissement que celui-ci a été déclenché. Soit *CPush* et *COvf* les interfaces (transitions) du port donnée-événement en sortie de ce thread, indiquant par leur franchissement qu'une donnée a été produite. On a alors :

   $$\mathtt{G}\,((\mathit{Pop} \rightarrow \mathtt{F}\,\mathit{CPush}) \;||\; (\mathit{Pop} \rightarrow \mathtt{F}\,\mathit{COvf}))$$

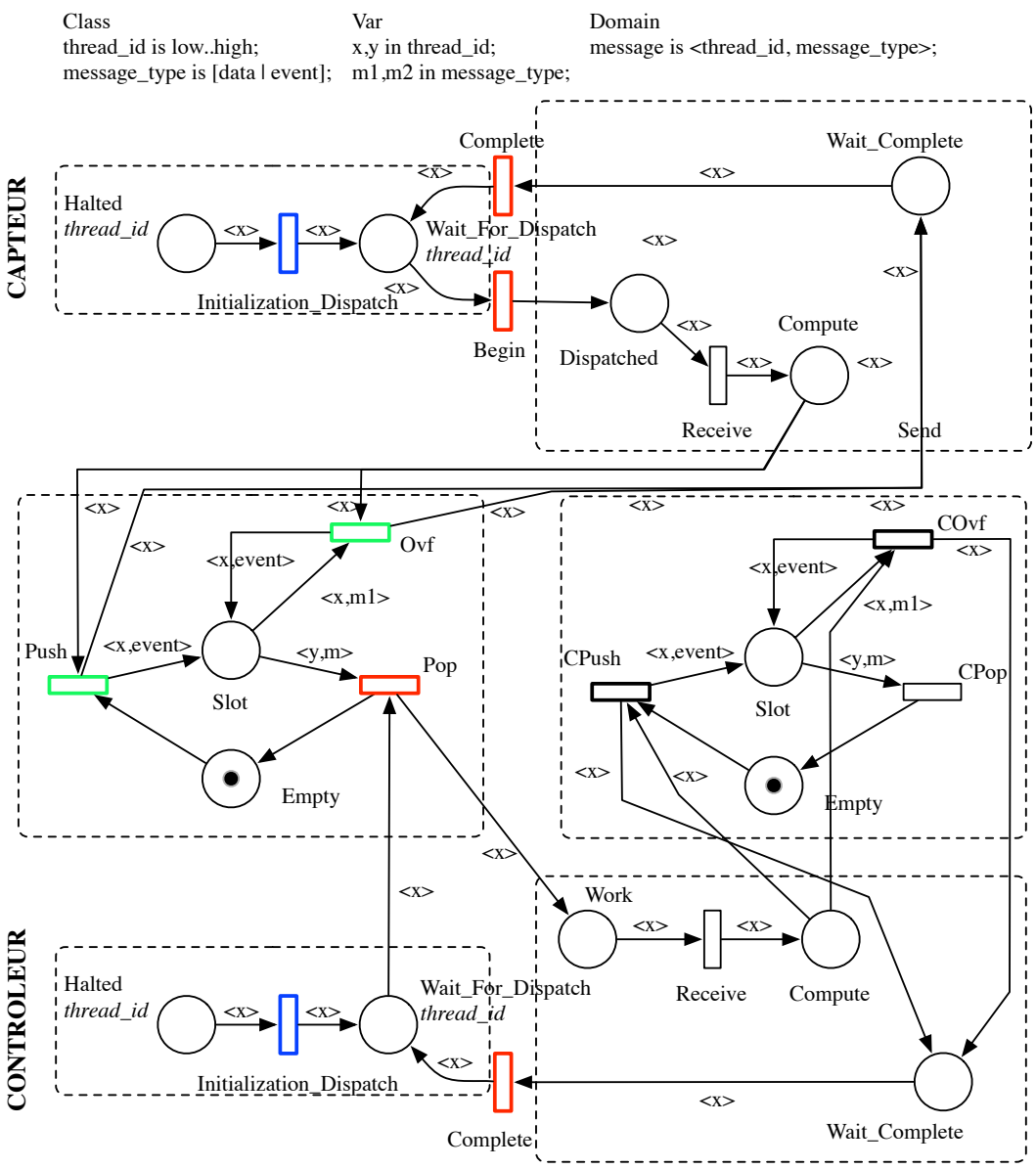


FIGURE 55 – Réseau de Petri coloré représentant l'interaction entre un thread `capteur` et un thread `contrôleur`

	P_1	P_2	P_3
Résultat	Faux	Vrai	Faux

TABLE 7.2 – Résultat des propriétés LTL exprimée sur le réseau de Petri coloré

Vérification des propriétés LTL

La validation des propriétés *PA* et *PB* se déduit des propriétés P_1, P_2 et P_3. Le tableau 7.2 présente les résultats de ces propriétés sur le réseau de Petri généré.

On s'aperçoit donc que les propriétés P_1 et P_3 (qui sont les mêmes, mais appliquées à deux entités différentes), ne sont pas validées.

⇒ *Retour au développeur (naïf) : Les capteurs produiront des données tout le temps, mais ces données peuvent ne pas être traitées par le contrôleur, et celui-ci peut ne pas générer de commande d'asservissement.*

Cela ne signifie pas que notre système n'est pas valide. En effet, la notion de temps ou de priorité n'intervenant pas en réseaux de Petri colorés, elles ne sont pas exprimées dans les patrons : le GMA contient alors des chemins a priori irréalisables qui peuvent donner lieu à des faux positifs lors de l'analyse.

L'analyse du GMA pour en extraire les marquages fautifs nous indique en effet que des contre-exemples sont induits par la modélisation et le formalisme.

Ils sont caractérisés par le fait que nous avons dans le système deux threads périodiques (les deux capteurs) qui, une fois traduits en réseaux de Petri, produisent deux P-Invariants ne faisant pas intervenir la place *Slot* du thread considéré.

Définition 7.2 *(P-Invariant) ou P-Semi-flot : le compte pondéré des jetons qui lui sont associés est constant quel que soit l'évolution du réseau marqué.*

Ces P-Invariants sont, pour chacun des capteurs :

$$Wait\_For\_Dispatch + Dispatched + Compute + Wait\_Complete = 1$$

Il existe donc deux boucles d'exécution indépendantes et infinies. Un chemin du GMA autorise l'exécution infinie d'un des capteurs au détriment de l'autre.

Pour faire face à ce problème, plusieurs solutions existent :

1. La synchronisation : en forçant l'exécution d'un capteur avant un autre, nous éliminons le facteur concurrentiel, et favorisons l'équité d'exécution des deux capteurs.
2. Réduire le GMA en supprimant certaines branches d'exécution.
3. Utiliser la symétrie du modèle initial pour n'analyser qu'une moitié de celui-ci (les deux moitiés sont indépendantes et identiques).

Quelle que soit la solution retenue pour éliminer ce biais, la propriété P_1 est validée pour un capteur.

Nous avons mis en œuvre la troisième solution, en réduisant le modèle initial (par le retrait d'un des capteurs) : nous avons effectué l'analyse (pour P_1 uniquement), sur un réseau où n'apparaît qu'un trio $\langle capteur, controleur, servomoteur \rangle$.

Nous avons donc P_1 et P_2 de vérifiées. Soit le fait que «le capteur produira toujours une donnée» et «si un capteur produit une donnée, alors son controleur l'analysera».

⇒ *Retour développeur : le système vérifie les propriétés P_1 et P_2*

Les prérequis de P_3 sont que le contrôleur doit être déclenché. D'après P_1 et P_2, ce sera toujours le cas.

La propriété P_3 ne peut être validée car le thread capteur provoque lui aussi un chemin d'exécution parasite dans le GMA. Pour y faire face, nous employons également une des méthodes évoquées précédemment :

1. La synchronisation (équité),
2. La réduction du GMA ,
3. L'exploitation de la symétrie du modèle initial.

La dernière boucle du GMA est alors éliminée, et la propriété P3 est validée.
La combinaison des propriétés P_1, P_2 et P_3 nous permet de valider la propriété P_A.

⇒ *Retour développeur : le système vérifie les propriétés P_1, P_2 et P_3, et donc la propriété P_A*

Pour valider la propriété P_B(«Une donnée émise par un capteur produit forcément la réaction d'un servomoteur»), il faut s'assurer qu'une information fournie par un capteur est bien traitée par un servomoteur (et que ce n'est pas une autre information qui provoque une réaction du servomoteur).

Nous exprimons cette propriété en LTL, en utilisant le patron présenté à la section 4.3.6. Ce patron met à profit la couleur pour estampiller les messages, afin de pouvoir les suivre dans le système.

$$\mathtt{G}\,(\mathit{Push}[tx = 1] \;\rightarrow \mathtt{F}\;\mathit{Begin}[tx = 1])$$

Il est à noter que nous avons exprimé la formule LTL en fonction d'un identifiant particulier de message. Un identifiant peut cependant être généralisé par une variable car il n'est fait aucune mention explicite (en particulier au niveau des gardes de transitions) d'un identifiant spécifique dans le modèle.

Ainsi, si cette propriété est vérifiée pour le message d'identifiant 1, alors elle sera vérifiée pour tous les autres identifiants de message.

$$\forall\, x \in \mathit{Msg\_Ids} \bullet P(x) \Rightarrow (\forall\, y \in \mathit{Msg\_Ids} \bullet P(y))$$

Si la propriété est vérifiée pour une instance de la classe Msg_Ids, elle l'est pour tous les autres.

⇒ *Retour développeur : le système vérifie la propriété P_B*

7.3.4 Réseaux de Petri Temporels

Si les propriétés qualitatives du système ont été vérifiées à l'aide du modèle en réseaux de Petri colorés, les réseaux de Petri temporels permettent de procéder à des analyses quantitatives.

A noter que l'on ne parle pas de graphe des marquages accessibles pour les réseaux de Petri temporels, mais d'une structure jouant le même rôle nommée «graphe des classes d'état» (GCE), qui est une abstraction finie du comportement des réseaux temporels bornés [BM83].

Comme pour les réseaux de Petri colorés, nous allons procéder aux analyses suivantes, en utilisant l'environnement Tina :

– Calcul des bornes du réseau (par approximation) ;
– Détection des nœuds terminaux ;
– Expression en LTL de propriétés comportementales du système et vérification sur le GCE.

Bornes

Le réseau de Petri temporel (figure 2 de l'annexe B) généré à l'aide d'Ocarina est borné, et se prête donc à l'analyse au travers de son GCE. Nous pouvons donc construire celui-ci.

Le tableau 7.3 présente donc le nombre d'éléments du réseau de Petri temporel, ainsi que du graphe des classes d'états associé, généré au moyen de l'outil tina. Celui-ci calcule de façon exhaustive l'espace d'état du système, avec des méthodes d'approximation.

	Réseau de Petri	GCE (tina)
Nœuds	51 places 37 transitions	628 états
Arcs	NC	1414

TABLE 7.3 – Statistiques sur le réseau de Petri temporel et son GCE

Alors que le réseau de Petri temporel reste de taille similaire au réseau de Petri coloré, le GCE associé est beaucoup plus petit que le GMA associé au réseau de Petri coloré.

En effet, de base, nous pouvons intégrer différentes informations dans la génération du réseau de Petri temporel qui réduisent, par ajout de contraintes, l'espace d'état accessible : on y trouve les attributs indiquant la période des threads, les priorités qu'ils ont entre eux, la notion de partage de ressource processeur (qui influe sur l'ensemble des priorités exprimées).

Toutes ces précisions réduisent donc significativement le GCE à analyser.

⇒ *Retour au développeur : le système est analysable.*

Nœuds terminaux

L'analyse du GCE révèle la présence d'un nœud terminal, ce qui implique un résultat négatif à première vue. Nous verrons cependant que ce nœud est attendu et justifié.

⇒ *Retour au développeur : le système est sujet aux interblocages.*

Par construction, le réseau de Petri temporel que nous générons produit un marquage correspondant à un état d'accueil : en effet, afin de justement être en mesure de construire le GCE pour l'analyser, nous nous limitons à l'étude de notre système sur une hyperpériode de celui-ci. Une fois cette hyperpériode révolue, le marquage de notre système n'évolue plus.

Cela est une conséquence du fait que lorsqu'il n'y a plus de jetons dans les places «Hyperperiod_Bound» de chaque thread, ceux-ci ne peuvent plus être déclenchés (la transition Dispatch ne sera jamais franchissable).

Le contre-exemple extrait du GCE caractérise cet «état d'accueil».

Il est donc normal d'avoir un nœud terminal dans notre GCE : il s'agit d'un état inévitable mais non inattendu. Cependant, celui-ci est et doit être vérifié : les threads du système doivent être revenus à un point d'exécution normal (en attente de déclenchement), et les ressources nécessaires (processeur par exemple) doivent être disponibles.

L'analyse de notre GCE nous indique que l'unique marquage mort est le suivant :

$$\langle CPU1\_Processor,$$
$$capteur\_droit\_Wait\_For\_Dispatch,$$

$$\textit{capteur\_gauche\_Wait\_For\_Dispatch},$$
$$\textit{controle\_droit\_Wait\_For\_Dispatch},$$
$$\textit{controle\_gauche\_Wait\_For\_Dispatch},$$
$$\textit{servomoteur\_droit\_Wait\_For\_Dispatch},$$
$$\textit{servomoteur\_gauche\_Wait\_For\_Dispatch}\rangle = \langle 1,1,1,1,1,1,1\rangle$$

Il s'agit donc du marquage mort attendu. Tout autre marquage traduirait un problème et nécessiterait davantage de réflexion pour isoler l'éventuel problème.

Il s'agit d'une règle LTL implicite à toujours vérifier.

⇒ *Retour au développeur : le système n'est pas sujet aux interblocages.*

Expression des propriétés en LTL

Les propriétés que nous souhaitons vérifier sur le système, en utilisant les réseaux de Petri temporels, sont les suivantes :

P_C les tampons de communication sont bien dimensionnés, aucun message n'est perdu ;

P_D le système est ordonnançable.

En accord avec l'attribut `Queue_Size` de la spécification AADL , la borne supérieure de la place de communication considérée doit être inférieure ou égale à l'attribut Queue_Size. Si cet attribut n'est pas spécifié, alors sa valeur par défaut est de 1.

Soit *Store* la place de communication considérée, nous pouvons exprimer la propriété avec la formule LTL suivante :

$$\mathtt{G}(\textit{Store} \leq \textit{Queue\_Size})$$

soit, en langage naturel, «toujours, le marquage de la place Store est inférieur ou égal à Queue_Size».

⇒ *Retour au développeur : P_C étant vérifiée, les tampons de communication du système sont bien dimensionnés.*

Sémantiquement, un seul message à la fois doit passer dans un bus : en avoir plusieurs n'aurait aucun sens. Nous avons vérifié avec la propriété précédente que les files d'attentes des ports ne conduiront pas à des pertes de messages. Nous vérifions maintenant à l'aide de la propriété LTL suivante que le marquage des places correspondant à un bus n'est pas supérieur à 1 (à répéter pour chaque place Bus du modèle) :

$$\mathtt{G}(\textit{Bus} \leq 1)$$

Cette propriété est vérifiée pour le cas d'étude considéré, pour chaque place de bus présente.

⇒ *Retour au développeur : au plus 1 message transite dans le canal de communication*

Afin de pousser plus avant l'analyse de notre système, nous avons généré les observateurs mentionnées au chapitre 4.3.6. Ils nous permettent de vérifier l'ordonnançabilité du système. Dans le cas d'étude considéré, nous disposons des informations relatives aux échéances des threads. Nous pouvons donc mettre en œuvre les mécanismes présentés par l'algorithme 14 de l'annexe A.

Pour s'assurer qu'un des threads de notre système ne manque pas son échéance, il faut s'assurer, en LTL, que la place de détection de l'observateur que l'on ajoute n'est jamais marquée. Soit *Deadline_Missed* la place d'accumulation de jeton dans le patron de l'observateur indiquant si un thread manque son échéance. Nous écrivons alors :

$$\mathbb{G}\,(Deadline\_Missed = 0)$$

soit, en langage naturel, «toujours, la place de marquage de l'observateur aura un marquage nul».

Cette formule est vérifiée pour le cas d'étude considéré.

⇒ *Retour au développeur : avec les informations temporelles disponibles, le système est ordonnançable.*

7.4 Connexion avec l'exécutif

La section précédente nous a permis de procéder à l'analyse comportementale de notre système, à partir de sa spécification en AADL. Cela a été fait en utilisant deux formalismes complémentaires, les réseaux de Petri colorés et les réseaux de Petri temporels.

Nous montrons maintenant comment, dans une seconde étape de vérification, nous pouvons vérifier que l'exécutif sous-jacent correspond bien à nos attentes, à partir de la spécification AADL du système.

7.4.1 Liaison entre la spécification Z et AADL

Nous rappelons le schéma décrivant un ORB, tel que spécifié dans le chapitre 5 (section 5.4) :

TOrb

OrbConfiguration

tsapName : *TTsapId*

requestQueue : $\mathrm{seq}\, TRequest$

tapList : $\mathbb{F}\, TTransportAccessPoint$

rootOA : *TObjectAdapter*

memory : *TMemory*

Il est utile aussi de se rappeler le schéma qui permet de configurer notre ORB :

OrbConfiguration

MaxRequestQueue : $\mathbb{N}$

MaxServant : $\mathbb{N}$

MaxMethod : $\mathbb{N}$

MaxMemory : $\mathbb{N}$

Tel que nous l'avons exprimé dans notre spécification, un «Servant» est un objet pouvant utiliser des ressources pour s'exécuter, afin de répondre à une demande de traitement. Par ailleurs, une méthode est un élément d'exécution, exécutée par l'objet lorsqu'il est activé. En d'autre termes, un servant dans notre spécification Z n'est rien d'autre qu'un thread dans une spécification AADL ; quant aux méthodes de la spécification Z, nous pouvons les assimiler à des appels de sous-programmes dans une spécification AADL.

Puisque nous sommes sur un robot, il n'y a qu'un seul nœud de déploiement pour notre intergiciel. Il n'y a donc qu'une seule instance d'ORB, par lequel tous les threads du système seront gérés. Par conséquent, l'envoi de requête d'un thread à un autre passera par la file de requête de l'ORB, «requestQueue».

La mémoire de l'ORB sera dimensionnée en accord avec le nombre d'instances de sous-programmes, puisqu'il s'agit, par assimilation, de l'unité de mémoire dans la spécification Z.

Nous avons vu quels éléments de la spécification AADL pouvaient être liés à la configuration de la spécification Z. Nous allons maintenant étudier quels sont les liens qui existent entre les modèles en réseau de Petri et la spécification en Z de l'exécutif. Nous serons ainsi en mesure de garantir la cohérence entre nos deux types de vérification.

7.4.2 Liaison entre la spécification Z et les modèles réseaux de Petri

A l'instar de la spécification AADL, certains éléments des modèles en réseaux de Petri se retrouvent dans la spécification Z.

Les threads sont les éléments centraux de la vérification comportementale effectuée grâce aux réseaux de Petri (colorés ou temporels).

Dans le cas des réseaux de Petri colorés, suite à la génération automatique depuis la spécification AADL, nous avons produit différentes classes et domaines de couleurs ainsi réparties :

- *Threads_Ids* : les identifiants de threads dans le système. Un identifiant par instance de thread, plus un identifiant spécial n'étant attaché à aucun thread (pour l'initialisation des place correspondants aux ports donnée par exemple).
- *Msg_Type* : une classe définissant le type d'un message envoyé dans le système.
- *mess* : un domaine de couleur définissant un tuple $\langle Threads\_Ids, Msg\_Type \rangle$: les messages circulant effectivement dans le système prendront leur valeur dans ce domaine de couleur.

La classe de couleur *Threads_Ids* est définie comme un intervalle : la valeur la plus petite *min* étant l'identifiant «réservé» aux marquages initiaux, la première valeur utile pour discriminer les threads étant $min + 1$. La dernière valeur de l'intervalle, *max*, définit la dernière valeur utile à l'identification d'un thread.

Nous pouvons donc en déduire le nombre de threads dans le système :

$$\begin{array}{c} max - (min + 1) + 1, \text{ où, comme } min = 0 \\ max \end{array}$$

Il y a donc *max* threads dans le système : ce nombre permet de configurer l'attribut *MaxServant* pour l'ORB.

Nous avons :

- établi le lien entre les threads AADL et les servants Z ;
- établi le lien entre le domaine de couleur des threads en réseaux de Petri colorés et le nombre d'instances de threads dans le modèle AADL.

Nous pouvons alors établir la correspondance entre la variable de configuration *MaxServant* et l'intervalle définissant la classe de couleur des identifiants de threads. De façon similaire, nous pouvons déduire, à partir du nombre d'instances de sous-programme générées dans le modèle réseau de Petri, la valeur de la variable de configuration *MaxMemory*, qui est utilisée pour charger en mémoire les sous-programmes. Enfin, la dernière variable de configuration de notre ORB est *MaxRequestQueue*. Celle-ci peut être déduite de notre modèle réseau de Petri en additionnant, pour chaque port de communication généré, la taille de leur file d'attente, définie par la propriété *Queue_Size* dans la spécification AADL.

Nous avons présenté quels étaient les liens entre les éléments générés des réseaux de Petri et les variables de configuration de notre spécification Z. Nous configurons maintenant celle-ci, afin de l'analyser pour savoir si l'exécutif reste cohérent et remplit ses obligations de preuves évoquées au chapitre 5.8.

7.4.3 Configuration de la spécification

En accord avec les deux sections précédentes, nous en déduisons, pour le cas d'étude considéré, les valeurs de nos variables de configuration :

- Il a 6 identifiants utiles de threads dans le domaine de couleur du réseau de Petri coloré généré ;
- Il y a 4 instances de ports générées dans les modèles en réseau de Petri, chacun ayant une file d'attente de taille 1 ;
- Il y a 6 instances de sous-programme générées dans les modèles en réseau de Petri.

Nous en déduisons le schéma d'initialisation suivant, pour *OrbConfiguration* :

InitOrbConfiguration

$$\begin{array}{|l} \hline OrbConfiguration' \\ \hline MaxRequestQueue' = 4 \\ MaxMemory' = 6 \\ MaxServant' = 6 \\ MaxMethod' = 6 \\ \hline \end{array}$$

Cela se répercute sur l'opération de configuration de *TOrb*, sur le schéma suivant :

InitOrb

$$\begin{array}{|l} \hline InitOrbConfiguration \\ TOrb' \\ \hline tsapName' = robot\_tsap \\ requestQueue' = \langle\,\rangle \\ tapList' = \emptyset \\ memory' = \emptyset \\ managedServants' = \langle\,\rangle \\ \hline \end{array}$$

Ce schéma initialise donc les bornes supérieures des différents éléments de l'ORB. On remarque que nous avons introduit l'entité *robot_tsap*, déclarée comme suit :

$$robot\_tsap : TTsapId$$

Il s'agit du nom utilisé pour déployer notre ORB sur le robot.

Nous pouvons maintenant procéder à l'analyse de la spécification de l'exécutif AADL.

7.4.4 Obligations de preuve et Analyse

Nous allons présenter les divers théorèmes et invariants qui doivent être satisfait pour l'ORB que nous considérons.

Pour cela, nous allons vérifier les éléments suivants :

- la syntaxe et la cohérence de types de la spécification,
- le domaine des schémas,
- les théorèmes liés à l'initialisation du système,
- les invariants du système.

Vérification de syntaxe et de typage

Il est tout d'abord nécessaire, pour chacun des éléments de notre spécification Z, de s'assurer de leur correction syntaxique. Ceci est vérifiable, et vérifié, automatiquement par Z/Eves dès que nous ajoutons un élément dans une spécification.

⇒ *Retour au développeur : la spécification est analysable.*

De la même façon, il faut s'assurer que la spécification est correcte en terme de typage : ce processus est lui aussi automatisé dans Z/Eves. Dans notre cas de figure, cela est vérifié.

⇒ *Retour au développeur : la spécification est cohérente en terme de typage.*

Vérification de domaine

Il est nécessaire de vérifier la cohérence de «domaine» d'une spécification. Ce procesus est appelé «domain check» : Z/Eves tente ainsi de s'assurer que les fonctions que nous utilisons sont appliquées correctement. Par exemple, l'opérateur de cardinalité # doit être appliqué sur un ensemble fini.

Z/Eves génère donc pour chaque élément E un théorème «E$domainCheck» qui, une fois prouvé, indique que l'élément considéré a un domaine valide.

Z/Eves tente de résoudre au maximum de façon automatique ces vérifications de domaine . Malgré tout, il est parfois nécessaire de guider l'outil.

Dans notre spécification, les vérifications de domaine suivantes n'ont pu être résolues automatiquement par Z/Eves ; elles ont cependant toutes été vérifiées suite à l'invocation de la commande `prove by reduce` :

- opExportServantOK$domainCheck,
- opExportServantERR$domainCheck,
- opReceiveMessageOK$domainCheck,
- opReceiveMessageERR$domainCheck,
- opActivationOK$domainCheck,
- opActivationERR$domainCheck,
- opBeginExecutionOK$domainCheck,
- opBeginExecutionERR$domainCheck.

⇒ *Retour au développeur : le domaine d'application des opérations de la spécification est correct.*

Les éléments de notre spécification sont donc cohérents en termes de syntaxe, de typage et de domaine.

Initialisation

Pour poursuivre l'analyse de la spécification de l'exécutif AADL, il faut considérer les aspects liés à l'initialisation du composant principal, TOrb.

Le théorème d'initialisation s'énonce comme suit :

theorem tInitTOrbOK
$\exists\, TOrb' \bullet InitOrb$

Ce théorème est vérifié.

⇒ *Retour au développeur : il existe donc un état initial pour le système configuré*

Invariants

Nous avons vu qu'il était possible d'invoquer les opérations modifiants les différents schéma d'états de notre spécification. Nous devons démontrer que l'application de ces opérations ne viole pas les invariants que nous avons spécifié.

Pour vérifier cela, nous devons exprimer des théorèmes de la forme suivante :

$$(Init \Rightarrow I) \wedge (Op1 \wedge I \Rightarrow I')$$

Avec *Init* le schéma d'initialisation de notre spécification, I le schéma spécifiant notre invariant, I' décrivant l'état de notre invariant après opération ; et finalement $Op1$ un schéma opération susceptible de violer l'invariant.

Si nous sommes en mesure de prouver le théorème exprimé ainsi, alors nous sommes capables de prouver qu'étant donné un état initial qui respecte l'invariant, alors l'application de l'opération ciblée ne modifie pas cet invariant.

Ces théorèmes ont été vérifiés pour les valeurs suivantes de I, et pour l'application de toutes les opérations apparaissant dans la spécification :

Invariant I	Traduction en langage naturel
$\#managedServants < MaxServant$	Le nombre de servants du système ne dépasse pas MaxServant
$\#requestQueue \leq MaxRequestQueue$	La taille de la file de réception des requêtes est inférieure à MaxRequestQueue
$\#memory \leq MaxMemory$	La taille de la mémoire allouée n'est pas supérieure à MaxMemory
$\#memory \geq 0$	La taille de la mémoire allouée n'est pas nulle

TABLE 7.4 – Invariants de la spécification respectés par l'application des opérations du système

⇒ *Retour au développeur : la configuration du système respecte les invariants à l'état initial, et, à partir de cet état, l'application des opérations les respecte aussi.*

En conclusion, les différents retours obtenus permettent de conclure que le système est correct :

- l'exécutif ciblé peut être configuré selon les informations disponibles (nous pouvons donc «démarrer» l'exécutif) ;
- l'exécutif est par ailleurs cohérent et correct (cf. chapitre 5).

7.5 Synthèse

Nous avons présenté dans ce chapitre un cas d'étude lié au domaine de la robotique. Ce cadre d'application de nos méthodes a permis de montrer comment utiliser notre chaîne d'outils sur un exemple concret, dans le cadre du processus de vérification que nous avons abordé au chapitre 3.

En nous appuyant sur la classification des propriétés AADL et des analyses qu'elles permettaient d'effectuer, nous avons exploité la spécification de ce cas d'étude pour procéder à deux itération de notre processus IDV^2, énumérées ci-après. Nous avons effectué ces deux itérations sur un système dont l'architecture était déjà établie, dans l'optique de valider la conception des développeurs. Les itérations effectuées sur le cas d'étude sont les suivantes :

1. Une itération de validation du comportement de l'application, uniquement basée sur sa spécification AADL. Cette itération peut prendre place dès la production des premières spécifications AADL dans le processus de développement ;
2. Une itération pour la vérification de l'exécutif, reposant sur la spécification formelle de PolyORB, et dépendant de la configuration demandée par la spécification AADL de l'application : cela nécessite d'avoir suffisamment d'informations dans la spécification AADL pour configurer entièrement l'exécutif.

Nous avons montré qu'il était possible d'appliquer les mécanismes présentés aux chapitres 4 et 5 pour effectuer l'analyse de différentes propriétés.

Pour le comportement de l'application, certaines d'entre elles sont vérifiables en utilisant les réseaux de Petri colorés, d'autres à l'aide des réseaux de Petri temporels :

- Absence d'interblocage ;
- Les capteurs génèrent des données indiquant si la ligne est survolée ou non ;
- Les données des capteurs sont bien analysées par le contrôleur ;
- Les commandes du contrôleur sont bien reçues et traitées par les servomoteurs ;
- Identification des liens de causalité entre l'émission d'une donnée d'un capteur et sa répercussion en commande adressée au servomoteur associé ;
- L'ordonnançabilité du système ;
- L'absence de perte de messages due à un mauvais dimensionnement du système (les files de messages ne sont jamais pleines).

Pour la vérification de l'exécutif, la spécification AADL nous a permis de configurer la spécification formelle de PolyORB dans l'optique de prouver différentes propriétés :

- La configuration de l'exécutif requise par l'application AADL est possible ;
- Les opérations manipulant l'exécutif configuré sont réalisables, notamment celles initialisant le système ;
- Les opérations manipulant l'exécutif ne violent pas d'invariant spécifiés par leur application.

La mise en œuvre de notre approche aux travers de ces deux itérations a permis de montrer qu'elle peut permettre à un développeur ne connaissant pas les méthodes formelles de valider et de vérifier différentes propriétés sur son système TR^{2}E.

Nous avons automatisé une grande partie des traitements à effectuer pour valider le comportement de l'application (génération automatique de modèles formels depuis une spécification AADL). Nous avons également préparé une spécification formelle «prête-à-l'emploi», en Z, de l'intergiciel PolyORB afin de vérifier des propriétés architecturales.

Il reste cependant des points qui nécessitent l'intervention du développeur pour procéder à des analyses complémentaires sur le système : au niveau comportemental, seules les propriétés classiques comme la détection d'interblocage sont automatiquement traitées. Pour la vérification de l'exécutif, il est nécessaire de guider le prouveur pour prouver des théorèmes sur la spécification Z.

Chapitre 8

Conclusion et perspectives

Contents

8.1 Processus de développement de systèmes TR^{2}E

Nos travaux ont pour cadre la conception des systèmes temps-réel, répartis et embarqués (TR^{2}E), qui ont une place prépondérante dans notre quotidien (avionique, téléphonie, automobile, ferroviaire, etc.). Ils nécessitent une attention particulière lors de leur développement. Il est en particulier nécessaire de pouvoir garantir ce qui a été spécifié ou de justifier les choix d'implantation.

L'analyse de tels systèmes est rendue possible par la création de modèles les représentant, sur lesquels il est possible de raisonner. Ces modèles permettent de modéliser les systèmes TR^{2}E en capturant et en structurant leurs caractéristiques (matériel, logiciel, contraintes, attributs, etc.).

Les méthodes formelles permettent d'accroître la fiabilité de ces systèmes. Leur intégration dans un cycle de développement pose cependant différents problèmes. Leur mise en œuvre requiert l'intervention d'experts, et leur intégration dans des processus industriels standardisés n'est pas aisée. Si une approche par modèle permet de structurer la vision du système et de diriger le développement, elle n'indique pas comment les utiliser pour valider le système.

Nous avons identifié plusieurs points liés à leur positionnement dans un cycle de développement :

– quel ensemble pertinent de propriétés établir pour étudier un aspect spécifique du système ?
– quand utiliser les méthodes formelles permettant d'exploiter cet ensemble ?
– quels outils mettre en œuvre pour appliquer ces méthodes formelles ?
– quel est l'impact des résultats d'analyse, comment les intégrer au processus de développement ?

L'étude de différentes méthodes de développement, de différentes techniques de modélisation et d'analyse a montré qu'elles ne permettaient d'aborder qu'une partie du problème. A partir de ces études, nous avons montré qu'il était possible d'établir une classification de ces techniques, selon leur pertinence dans l'analyse d'un système TR^{2}E.

8.2 Contributions

Nos travaux s'inscrivent dans l'optique d'une production automatique de systèmes TR^{2}E, les rendant corrects par construction. Nos contributions répondent à un besoin d'intégration des méthodes formelles dans un processus de développement itératif.

8.2.1 Guide de sélection des méthodes formelles

Nous avons constaté l'intérêt des langages de description d'architecture pour intégrer les méthodes formelles. Nous en avons étudié plusieurs (chap. 2) pour sélectionner AADL, un langage standardisé (par la *SAE International*) et adapté à la modélisation de systèmes TR^{2}E.

Nous avons analysé ce langage et classé les attributs existants selon leur impact sur la vérification et l'analyse du système (chap. 3).

Nous avons également étudié l'adéquation de ce langage avec des processus de développement itératifs qui facilitent une intégration efficace des méthodes formelles (sec. 2.3), en privilégiant des retours d'analyse réguliers tout au long du cycle. Enfin, nous avons présenté un processus de vérification et de validations'y insérant (sec. 3.4.3).

Ce processus repose sur le principe qu'AADL sert de notation pivot pour générer des modèles formels. Les formalismes cibles sont choisis en fonction de leur capacité d'analyse de la propriété sélectionnée. A chaque itération du processus de vérification et de validation, le modèle (pivot) sera raffiné et corrigé, de façon à correspondre aux attentes exprimées pendant les phases de spécification.

8.2.2 Orchestration des méthodes formelles pour l'analyse de systèmes TR^{2}E

Nous avons exploité les résultats précédents pour orchestrer l'analyse de systèmes TR^{2}E à l'aide de deux types de méthodes formelles.

Notre approche s'appuie sur des notations formelles et semi-formelles standardisées : AADL, Z, réseaux de Petri.

Guide de transformation pour la vérification comportementale

Nous avons proposé des patrons pour générer depuis AADL des modèles formels propices à l'analyse comportementale. Pour chaque élément du langage, nous avons résumé son impact sur le comportement de l'application (sec. 4.1). Nous avons identifié dans le standard AADL les éléments caractérisant le comportement de l'application.

Nous en avons déduit des patrons de transformation transposables vers tout formalisme basé sur des automates états-transitions, ainsi que des règles d'assemblage pour composer les éléments générés (sec. 4.1.2). Nous avons appliqué cette méthodologie à deux familles de réseaux de Petri : les réseaux colorés et les réseaux temporels (sec. 4.3). Ces deux formalismes nous permettent de procéder à des analyses complémentaires sur le système :

- détection d'interblocages,
- analyse d'ordonnancement,
- analyse des flots de messages ,
- dimensionnement des bornes de tampons.

Nous avons également proposé des algorithmes de génération de réseaux de Petri (annexe A) autorisant à la fois l'extension des patrons de transformation existants et l'ajout de nouveaux observateurs.

Formalisation d'un exécutif dédié au langage pivot (AADL)

Nous avons complété la vérification comportementale des applications par une analyse de l'exécutif pour une configuration donnée. Nous avons utilisé la notation formelle Z pour effectuer des vérification par preuves de théorèmes (sec. 5.2).

Nous avons étudié et analysé un exécutif AADL : PolyORB (5.3). Nous en avons extrait les composants essentiels et les points de configuration. Nous avons spécifié les invariants que chacun de ses composants doit respecter.

Nous avons proposé une spécification formelle configurable de cet exécutif : elle peut être manipulée au travers d'interfaces permettant de l'initialiser ou de construire des scenarii d'exécution, en accord avec la spécification AADL de l'application. Cela permet de vérifier la validité d'une configuration : respect des invariants et composition des opérations. L'utilisateur final est donc en mesure de vérifier son exécutif selon les propriétés attendues.

8.2.3 Vers une automatisation du processus de vérification

Nous proposons une chaîne d'outils mettant en application les résultats des chapitres 4 et 5, la validation comportementale de l'application et la vérification de l'exécutif.

Analyse comportementale

Nous avons implanté la transformation d'une spécification AADL vers un modèle en réseaux de Petri colorés et un modèle en réseaux de Petri temporels en vue de son analyse comportementale. Nous avons pour cela implanté un générateur dans la partie arrière (*backend*) de l'outil Ocarina (chap. 6). Ce générateur est générique afin de faciliter la production de spécification dans une nouvelle classe de réseaux de Petri. Pour cela, nous avons séparé les préoccupation propres à un formalisme (à l'aide d'annotations), des aspects structurels des patrons de transformation. Ainsi, pour gérer un nouveau formalisme, il faut définir dans une structure dédiée ses spécificités par rapport aux réseaux de Petri Place-Transition, et proposer une bibliothèque de gestion des nœuds (création, initialisation, connexion).

Nous avons mis en place un ensemble de script permettant de vérifier des propriétés récurrentes sur les systèmes TR^{2}E : absence d'interblocage, borne des tampons, ordonnancement. Cet outillage a été exploité avec succès sur un cas d'étude (sec. 7.3), démontrant la faisabilité de notre approche et la cohérence de nos règles de transformation.

Analyse de l'exécutif

Nous avons également mis en œuvre la spécification Z de l'exécutif AADL pour en vérifier des propriétés structurelles (sec. 7.4). Cela a permis de montrer les types de preuve attendus. Nous avons proposé des conseils méthodologiques pour la résolution de preuves (sec. 6.2).

Ce type de vérification ne peut être complètement automatisé, mais des patrons de commandes applicables au prouveur peuvent être utilisés de façon systématique pour faciliter la résolution des preuves.

8.3 Perspectives

La démarche proposée dans ce mémoire nous permet de vérifier pour un système TR^{2}E des propriétés comportementales et structurelles. Ces contributions ouvrent la voie à de nouveaux travaux, dont certains sont déjà en cours de réalisation : extension des guides de sélection, intégration de formalismes supplémentaires dans la démarche, amélioration de l'automatisation de notre processus.

Extension des guides de sélection

Si nos travaux rendent plus facile l'utilisation des méthodes formelles au sein d'un processus de développement, les guides fournis ne sont pas exhaustifs et gagneraient à être davantage affinés.

A ce stade, les guides que nous proposons couvrent principalement des propriétés «simples» : détection d'interblocages, famine, etc.

Il serait intéressant de proposer une classification par mots-clefs des attributs d'un système, en fonction des types de propriétés ciblées.

Il serait également possible de diviser plus finement les propriétés à vérifier sur un système et de rajouter un panel de propriétés plus important.

Intégration de nouvelles méthodes formelles dans notre processus itératif

Nous avons montré la faisabilité de l'approche pour deux classes de propriété. Dans chaque cas, nous proposons des règles de transformation et des patrons de conception facilitant le passage d'une notation pivot (AADL) à un formalisme choisi.

Il serait intéressant de proposer également des patrons et des règles de transformation pour chaque méthode d'analyse, en fonction des attributs du système à considérer.

Intégrer davantage de méthodes formelles dans notre cycle itératif étendra également les possibilités de vérification par un développeur tiers, et leur mise en œuvre dans les processus de développement industriels.

Cela va également enrichir le panel de propriété analysable depuis une spécification AADL.

Meilleure exploitation de la spécification AADL dans le processus de vérification

Nous avons proposé des règles automatisant la transformation d'une spécification AADL vers les réseaux de Petri. Nous avons également montré comment il était possible de dériver des obligations de preuve de façon systématique pour la spécification Z.

Il serait intéressant d'étendre cette notion d'obligations de preuve à la vérification comportementale en générant automatiquement les propriétés de logique temporelle à vérifier, à partir d'annotations AADL.

Nous pouvons pour cela exploiter certains mécanismes d'AADL : en utilisant des annexes comportementales, nous pouvons décrire des observateurs, ou encore spécifier des flots de données. Cela est un défi car nous avons montré que même sur un cas d'étude raisonnable, des difficultés lors de l'interprétation des résultats pouvaient apparaître.

Publications

Les travaux que nous avons présentés ont donné lieu à des publications dans diverses conférences internationales ainsi que dans différents workshop. Nous présentons ci-après la liste de ces publications.

2009

X. RENAULT, F. KORDON, J. HUGUES, « Adapting models to model checkers, a case study : Analysing AADL using Time or Colored Petri Nets ». In *Proceedings of the 20th International Symposium on Rapid System Prototyping (RSP'09)*, pages 26-33, Paris, France, June 2009. IEEE Computer Society.

X. RENAULT, F. KORDON and J. HUGUES, « From AADL Architectural Models to Petri Nets : Checking Model Viability, ». In *12th IEEE International Symposium on Object-oriented Real-Time distributed Computing (ISORC'09)*, pages 313-320, Tokyo, Japan, March 2009. IEEE Computer Society.

2008

X. RENAULT, J. HUGUES, F. KORDON, « Formal Modeling of a Generic Middleware to Ensure Invariant Properties ». In *10th Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, pages 185-200, Oslo, Norway, June 2008. Lecture Notes in Computer Science (LNCS), Springer-Verlag.

F. KORDON, J. HUGUES, X. RENAULT, « From Model Driven Engineering to Verification Driven Engineering ». In *6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008)*, pages 381-393, Capri Island, Italy, October 2008. Lecture Notes in Computer Science (LNCS), Springer-Verlag.

2007

F. BONNEFOI, L. M. HILLAH, F. KORDON, X. RENAULT, « Design, modeling and analysis of ITS using UML and Petri Nets ». In *10th International IEEE Conference on Intelligent Transportation Systems (ITSC'07)*, pages 314-319, Seattle, USA, September 2007. IEEE Computer Society.

2006

A. HAMEZ, L. M. HILLAH, F.KORDON. A.LINARD, E. PAVIOT-ADET, X. RENAULT, Y. THIERRY-MIEG, « New features in CPN-AMI 3 : Focusing on the analysis of complex distributed systems ». In *6th International Conference on Application of Concurrency to System Design (ACSD '06)*, pages 273–275, Turku, Finland, June 2006. IEEE Computer Society.

Bibliographie

[13502] ISO/IEC 13568. « Z formal specification notation — Syntax, type system and semantics », 2002.

[13608] « VDMTools : advances in support for formal modeling in VDM ». *SIGPLAN Not.*, 43(2) :3–11, 2008.

[ABGM09] A. ALETI, S. BJORNANDER, L. GRUNSKE and I. MEEDENIYA. « ArcheOpterix : An extendable tool for architecture optimization of AADL models ». *Model-Based Methodologies for Pervasive and Embedded Software, International Workshop on*, 0 :61–71, 2009.

[Abr74] J-R. ABRIAL. « Data Semantics ». In *IFIP Working Conference Data Base Management*, pages 1–60, 1974.

[Abr96] J-R. ABRIAL. *The B book - Assigning Programs to meanings*. Cambridge University Press, 1996.

[All97] R.J. ALLEN. « *A formal approach to software architecture* ». PhD thesis, Pittsburgh, PA, USA, 1997. Chair-Garlan, David.

[Amb04] S.W. AMBLER. *The Object Primer : Agile Model-Driven Development with UML 2.0*. Cambridge University Press, New York, NY, USA, 2004.

[Apa] APASPHERE LTD. « The omniORB home-page, `http://omniorb.sourceforge.net/` ».

[ari07] « ARINC 653 ». Technical Report, Aeronautical Radio Incorporated, 2007.

[ASM80] J-R. ABRIAL, S. A. SCHUMAN and B. MEYER. Specification Language. In *On the Construction of Programs*, pages 343–410. 1980.

[Ate08] ATELIER B. « Atelier B, the industrial tool to efficiently deploy the B Method, `http://www.atelierb.eu/index\_en.html` », 2008.

[BABC+09] T. BARROS, R. AMEUR-BOULIFA, A. CANSADO, L. HENRIO and E. MADELAINE. « Behavioural models for distributed Fractal components ». *Annales des Télécommunications*, 64(1-2) :25–43, 2009.

[Bac59] J. W. BACKUS. « The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference ». In *IFIP Congress*, pages 125–131, 1959.

[Bar07] M. BARR. « Embedded Systems Glossary », 2007.

[Bar08] K. BARBARIA. « *Architectures intergicielles pour la tolérance aux fautes et le consensus* ». PhD thesis, TELECOM-ParisTech, September 2008.

[BCP88] B. BROCK, S. COOPER and W. PIERCE. « Analogical Reasoning and Proof Discovery ». In *Proceedings of the 9th International Conference on Automated Deduction*, pages 454–468, London, UK, 1988. Springer-Verlag.

[BCS03] E. BRUNETON, T. COUPAYE and J.B. STEFANI. « The Fractal Component Model ». Technical Report, Specification v2, ObjectWeb Consortium, 2003.

[BCvH+03] J. BILLINGTON, S. CHRISTENSEN, K. van HEE, E. KINDLER, O. KUMMER, L. PETRUCCI, R. POST, C. STEHNO and M. WEBER. « The Petri Net Markup Language : Concepts, Technology, and Tools ». In *Applications and Theory of Petri Nets 2003 : 24th International Conference*, pages 1023–1024, Eindhoven, The Netherlands, June 2003.

[Bel06] R. BELL. « Introduction to IEC 61508 ». In *SCS '05 : Proceedings of the 10th Australian workshop on Safety critical systems and software*, pages 3–12, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[BFW04] A. J. BENNETT, A. J. FIELD and C. M. WOODSIDE. « Experimental Evaluation of the UML Profile for Schedulability, Performance and Time ». In Thomas BAAR, Alfred STROHMEIER, Ana MOREIRA and Stephen J. MELLOR, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 143–157. Springer, 2004.

[BJ78] D. BJØRNER and C. B. JONES, editors. *The Vienna Development Method : The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[BM83] B. BERTHOMIEU and M. MENASCHE. « An Enumerative Approach For Analyzing Time Petri Nets ». In *Proceedings IFIP*, pages 41–46. Elsevier Science Publishers, 1983.

[BN84] A. BIRRELL and B. J. NELSON. « Implementing Remote Procedure Calls ». *ACM Trans. Comput. Syst.*, 2(1) :39–59, 1984.

[BR76] G. BERTHELOT and G. ROUCAIROL. « Reduction of Petri-Nets ». In *MFCS*, pages 202–209, 1976.

[Buc94] P. BUCHHOLZ. « Hierarchical High Level Petri Nets for Complex System Analysis ». In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, pages 119–138, London, UK, 1994. Springer-Verlag.

[BW98] U. BROCKMEYER and G. WITTICH. « Case Study : Verification of an Embedded Fault-Tolerant Avionics System », 1998.

[BW01] A. BURNS and A. J. WELLINGS. *Real-Time Systems and Programming Languages : Ada 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[BWG08] C.D.S. O. BRUNO, M. WANG and J. GIBBONS. « The visitor pattern as a reusable, generic, type-safe component ». In *OOPSLA '08 : Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 439–456, New York, NY, USA, 2008. ACM.

[CAB+94] D. COLEMAN, P. ARNOLD, S. BDOFF, H. GILCHRIST, F. HAYES and P. JEREMAES. *Object-oriented Development : the Fusion method*. Englewood Cliffs, NJ, 1994.

[CDFH91] G. CHIOLA, C. DUTHEILLET, G. FRANCESCHINIS and S. HADDAD. Stochastic Well-Formed Coloured Nets and Multiprocessor Modelling Applications. In Kurt JENSEN and Grzegorz ROZENBERG, editors, *High-Level Petri Nets – Theory and Application*, pages 504–530. Springer, 1991.

[CGP00] E. CLARKE, O. GRUMBERG and D. PELED. *Model Checking*. MIT Press, 2000.

[Coo71] S. A. COOK. « The complexity of theorem-proving procedures ». In *STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[CoQ] COQ PROJECT AT INRIA. « The Coq proof assistant, http://coq.inria.fr/coq-eng.html ».

[CRBS09] M. Y. CHKOURI, A. ROBERT, M. BOZGA and J. SIFAKIS. « Translating AADL into BIP - Application to the Verification of Real-Time Systems ». pages 5–19, 2009.

[CTR] CTR TEAM. « Modeling and Analysis Suite for Real-Time Applications, http://mast.unican.es/ ».

[DEMS02] F. DUCLOS, J. ESTUBLIER, P. MORAT and Dassault SYSTÈMES. « Describing and Using Non Functional Aspects in Component Based Applications ». In *in Component Based Applications. 1 st International Conference on Aspect-Oriented Software Development*, pages 65–75, 2002.

[DLP] A. DURET-LUTZ and D. POITRENAUD. « SPOT, Spot Produces Our Traces, http://spot.lip6.fr/wiki/ ».

[dN07] D. de NIZ. « Diagrams and Languages for Model-Based Software Engineering of Embedded Systems : UML and AADL ». Technical Report, SEI, 2007.

[DOTY02] C. DAWS, A. OLIVERO, S. TRIPAKIS and S. YOVINE. « The tool Kronos, http://www-verimag.imag.fr/TEMPORISE/kronos/ », 2002.

[DPF09] J. DELANGE, L. PAUTET and P. H. FEILER. « Validating Safety and Security Requirements for Partitioned Architectures. ». In *Ada-Europe*, volume 5570 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2009.

[DTA+08] S. DEMATHIEU, F. THOMAS, C. ANDRÉ, S. GÉRARD and F. TERRIER. « First Experiments Using the UML Profile for MARTE ». In *ISORC '08 : Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 50–57. IEEE Computer Society, 2008.

[Eif08] EIFFEL SOFTWARE. « EiffelStudio - A Complete Integrated Development Environment, http://www.eiffel.com », 2008.

[EKP+05] S. EVANGELISTA, C. KAISER, C. PAJAULT, J-F. PRADAT-PEYRE and P. ROUSSEAU. « Dynamic Tasks Verification with Quasar ». In *Reliable Software Technology - Ada-Europe 2005, 10th Ada-Europe International Conference on Reliable Software Technologies*, volume 3555 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 2005.

[Est08] ESTEREL-TECHNOLOGIES. « SCADE Suite, http://www.esterel-technologies.com/products/scade-suite/ », 2008.

[Eva94] A. S. EVANS. « Specifying Verifying Concurrent Systems Using Z ». In *Proc. FME'94 Industrial Benefits of Formal Methods, volume 873 of Lecture Notes in Computer Science*, pages 366–380. Springer-Verlag, 1994.

[FL00] G. FREY and L. LITZ. « Formal methods in PLC programming ». In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, pages 2431–2436 vol.4, 2000.

[fle09] « Flex-eWare http://www.flex-eware.org/ », 2009.

[FM89] P.K.D. FROOME and B.Q. MONAHAN. « Specbox : a toolkit for BSI-VDM ». In *Software Engineering for Real Time Systems, 1989., Second International Conference on*, pages 50–54, Sep 1989.

[Fre04] L. FREITAS. « Proving Theorem with Z/Eves ». Technical Report, University of Kent, 2004.

[Gos07] S. GOSH. *Distributed Systems - An Algorithmic Approach*. Chapman and Hall CRC, 2007.

[Gre] GREATSPN. « Petri Nets suite : `http://www.di.unito.it/~greatspn` ».

[Had88] S. HADDAD. « A reduction theory for coloured nets ». In *European Workshop on Applications and Theory in Petri Nets*, pages 209–235, 1988.

[Hay92] I. J. HAYES. « VDM and Z : A Comparative Case Study ». *Formal Asp. Comput.*, 4(1) :76–99, 1992.

[HDD+03a] J. HATCLIFF, W. DENG, M. B. DWYER, G. JUNG and V. RANGANATH. « Cadena : An Integrated Development, Analysis, and Verification Environment for Component-based Systems ». In *in Proceedings of the 25th International Conference on Software Engineering*, 2003.

[HDD+03b] J. HATCLIFF, W. DENG, M. B. DWYER, G. JUNG, V. RANGANATH and R. ROBBY. « Slicing and partial evaluation of CORBA component model designs for avionics system ». *SIGPLAN Not.*, 38(10) :1–2, 2003.

[HHJ+93] I. J. HAYES, I. J. HAYES, C. B. JONES, C. B. JONES, J. E. NICHOLLS and J. E. NICHOLLS. « Understanding the differences between VDM and Z ». Technical Report, 1993.

[HHK+06] A. HAMEZ, L. HILLAH, F. KORDON, A. LINARD, E. PAVIOT-ADET, X. RENAULT and Y. THIERRY-MIEG. « New features in CPN-AMI 3 : focusing on the analysis of complex distributed systems. ». In *ACSD*, pages 273–275. IEEE Computer Society, 2006.

[Hil] L. HILLAH. « PNMLFramework `http://pnml.lip6.fr` ».

[HJ04] G. HOLZMANN and R. JOSHI. « Model-Driven Software Verification ». In S. GRAF and L. MOUNIER, editors, *Model Checking Software, 11th International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2004.

[HN07] I. HAMID and E. NAJM. « Real-Time Connectors for Deterministic Data-flow ». In *Proceedings of RTCSA'07*, pages 173–182, 2007.

[Hoa78] C. A. R. HOARE. « Communicating sequential processes ». *Commun. ACM*, 21(8) :666–677, 1978.

[Hoa85] C. A. R. HOARE. *Communicating Sequential Processes (Prentice Hall International Series in Computing Science)*. Prentice Hall, April 1985.

[Hol96] G. HOLZMANN. « On-the-fly model checking ». *ACM Comput. Surv.*, page 120, 1996.

[Hol00] G. HOLZMANN. « Logic Verification of ANSI-C Code with SPIN ». In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2000.

[Hol07] G. HOLZMANN. « On-the-fly, LTL Model Checking with SPIN, `http://spinroot.com/spin` », 2007.

[HS02] G. HOLZMANN and M. SMITH. « An Automated Verification Method for Distributed Systems Software Based on Model Extraction ». *IEEE Trans. Software Eng.*, 28(4) :364–377, 2002.

[HTMK+04] J. HUGUES, Y. THIERRY-MIEG, F. KORDON, L. PAUTET, S. BAARIR and T. VERGNAUD. « On the Formal Verification of Middleware Behavioral Properties ». In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, Linz, Austria, September 2004.

[Hug05] J. HUGUES. « *Architecture et Services des Intergiciels Temps Réel* ». PhD thesis, TELECOM-ParisTech, September 2005.

[HZPK08a] J. HUGUES, B. ZALILA, L. PAUTET and F. KORDON. « From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite ». *ACM Transactions in Embedded Computing Systems (TECS)*, oct 2008.

[HZPK08b] J. HUGUES, B. ZALILA, L. PAUTET and F. KORDON. « From the prototype to the final embedded system using the Ocarina AADL tool suite ». *ACM Trans. Embed. Comput. Syst.*, 7(4) :1–25, 2008.

[JHR+07] E. JAHIER, N. HALBWACHS, P. RAYMOND, X. NICOLLIN and D. LESENS. « Virtual execution of AADL models via a translation into synchronous programs ». In *EMSOFT*, pages 134–143, 2007.

[KCD+94] P. KELEHER, A.L. COX, S. DWARKADAS, H. DWARKADAS and W. ZWAENEPOEL. « TreadMarks : Distributed Shared Memory on Standard Workstations and Operating Systems ». In *In Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, 1994.

[KHR08] F. KORDON, J. HUGUES and X. RENAULT. « From Model Driven Engineering to Verification Driven Engineering ». In *6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008)*, Lecture Notes in Computer Science (LNCS), pages 381–393. Springer-Verlag, oct 2008. INT LIP6 MoVe.

[Kru98] P. KRUCHTEN. *The Rational Unified Process, An Introduction*. Addison Wesley, 1998.

[LAA] LAAS. « TIme petri Net Analyzer (TINA) toolbox ».

[Lab06] LABRI. « FAST - Fast Acceleration of Symbolic Transition systems, `http://www.lsv.ens-cachan.fr/fast` », 2006.

[LCV00] B. LEWIS, E. COLBERT and S. VESTAL. « Developing Evolvable, Embedded, Time-Critical Systems with MetaH ». In *TOOLS '00 : Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, page 447, Washington, DC, USA, 2000. IEEE Computer Society.

[LEK+98] C. LÜTH, E.W.KARLSEN, KOLYANG, S.WESTMEIER and B.WOLFF. « HOL-Z in the UniForM-Workbench – a Case Study in Tool Integration for Z ». In J. BOWEN, editor, *11. International Conference of Z Users ZUM'98*, LNCS 1493, pages 116–134. Springer Verlag, 1998.

[LIP] LIP6/MOVE. « The CPN-AMI home page, `http://www.lip6.fr/cpn-ami/` ».

[Mal08] F. MALLET. « Clock Constraint Specification Language ». Technical Report, INRIA, 2008.

[MC] Omg Object MANAGEMENT and Orb CORE. « . CORBA Standards ».

[McM] K. L. MCMILLAN. « The SMV System, `http://www.cs.cmu.edu/~modelcheck/smv.html` ».

[Med00] N. MEDVIDOVIC. « A Classification and Comparison Framework for Software Architecture Description Languages ». *IEEE Transactions on Software Engineering*, 26 :70–93, 2000.

[MH09] L. MESSAN-HILLAH. « *Intégration des méthodes formelles au développement dirigé par les modèles, pour la conception et la vérification des systèmes et applications répartis* ». PhD thesis, Université Pierre et Marie Curie, 2009.

[MNL08] E. MUMOLO, M. NOLICH and K. LENAC. « A real-time embedded kernel for nonvisual robotic sensors ». *EURASIP J. Embedded Syst.*, 2008 :1–13, 2008.

[MS97] I. MEISELS and M. SAALTINK. « The Z/EVES Reference Manual (for Version 1.5) », Sep 1997.

[NTI+04] T. NAKAJIMA, E. TOKUNAGA, H. ISHIKAWA, K. FUJINAMI and S. OIKAWA. « Human Factor Issues in Building Middleware for Pervasive Computing ». *Software Technologies for Future Embedded and Ubiquitous Systems, IEEE Workshop on*, page 3, 2004.

[OCW06] M. OLIVEIRA, A. CAVALCANTI and J. WOODCOCK. « Unifying Theories in ProofPower-Z ». In *UTP*, pages 123–140, 2006.

[OMG97] OMG. *OMG IDL Syntax and Semantics*. OMG, May 1997.

[OMG03] OMG. *MDA Guide v1.01*. OMG, 2003.

[OMG06a] OMG. « CORBA Component Model Specification, v4.0 ». Technical Report, Object Management Group, 2006.

[OMG06b] OMG. *CORBA Component Model Specification, version 4*. OMG, 2006.

[OMG06c] OMG. *Deployment and Configuration of Component-based Distributed Applications Specification*. Number formal/06-04-02. OMG, 2006.

[OMG07] OMG. « Data Distribution Service for Real-time Systems, v1.2 ». Technical Report, Object Management Group, 2007.

[OPE07] OPERA GROUP, UNIV. CAMBRIDGE. « QPME Homepage, `http://www.dvs.tu-darmstadt.de/staff/skounev/QPME/` », 2007.

[PAG03] A. POPOVICI, G. ALONSO and T. GROSS. « Spontaneous Container Services ». In *17th ECOOP*, 2003.

[Pau94] L. C. PAULSON. *Isabelle : a Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.

[Pau01] L. PAUTET. « *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition* ». Habilitation à diriger des recherches, Université Pierre et Marie Curie – Paris VI, December 2001.

[PC07] N. PONTISSO and D. CHEMOUIL. « Vérification formelle d'un modèle AADL à l'aide de l'outil UPPAAL ». *Génie Logiciel*, 80 :36–40, mars 2007.

[PDCD+00] C. L. PEREIRA, . Jr. da Silva D. C., R. G. DUARTE, A. O. FERNANDES, L. H. CANAAN, . Jr. Coelho C. J. N. and L. L. AMBROSIO. « JADE : An Embedded Systems Specification, Code Generation and Optimization Tool ». In *SBCCI '00 : Proceedings of the 13th symposium on Integrated circuits and systems design*, page 263, Washington, DC, USA, 2000. IEEE Computer Society.

[Pet62] C. A. PETRI. « Fundamentals of a Theory of Asynchronous Information Flow ». In *IFIP Congress*, pages 386–390, 1962.

[PK04] L. PAUTET and F. KORDON. « Des vertus de la schizophrénie pour le prototypage d'applications à composants interopérables ». *Technique et Science Informatiques*, 23(10) :1301–1328, 2004.

[PL92] N. PLAT and P.G. LARSEN. « An overview of the ISO/VDM-SL standard ». *SIGPLAN Not.*, 27(8) :76–82, 1992.

[POM03] R. PICHLER, K. OSTERMANN and M. MEZINI. « On Aspectualizing Component Models ». *Software – Practice and Experience*, 2003.

[pos04] « Standard for information technology - portable operating system interface (POSIX). Shell and utilities ». *IEEE Std 1003.1, 2004 Edition The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell*, pages –, 2004.

[PPSS00] A. PRETSCHNER, E. PRETSCHNER, O. SLOTOSCH and T. STAUNER. « Developing Correct Safety Critical, Hybrid, Embedded Systems ». In *In Proc. New Information Processing Techniques for Military Systems, Istanbul, October 2000. NATO Research and Technology Organization*, 2000.

[Pra08a] PRAGMADEV. « Real Time Developer Studio », 2008.

[Pra08b] PRAXIS HIGHT INTEGRITY SYSTEMS. « SPARKAda, http://www.praxis-his.com/sparkada/ », 2008.

[PRI08] PRISM TEAM. « PRISM - Probabilistic Symbolic Model Checker, http://www.prismmodelchecker.org/ », 2008.

[Qui03] T. QUINOT. « *Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables* ». PhD thesis, TELECOM-ParisTech, March 2003.

[RHK08] X. RENAULT, J. HUGUES and F. KORDON. « Formal Modeling of a Generic Middleware to Ensure Invariant Properties ». In *10th Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, Lecture Notes in Computer Science (LNCS), pages 185–200. Springer-Verlag, jun 2008.

[RKH09a] X. RENAULT, Fabrice K. and J. HUGUES. « From AADL architectural models to Petri Nets : Checking model viability ». In *12th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'09)*, pages 313–320, Tokyo, Japan, March 2009. IEEE Computer Society.

[RKH09b] X. RENAULT, F KORDON and J. HUGUES. « Adapting models to model checkers, a case study : Analysing AADL using Time or Colored Petri Nets ». In *Proceedings of the 20th International Workshop on Rapid System Prototyping*, pages 26–33, Paris, June 2009. IEEE Computer Society.

[RKMS07] X. RENAULT, F. KORDON, J. MALENFANT and L. SEINTURIER. « Components model for Flex-eWare ». Livrable de projet, 2007.

[Rol09] J-F. ROLLAND. « *Développement et validation d'architectures dynamiques* ». PhD thesis, Université de Toulouse, 2009.

[Rus94] John RUSHBY. « Critical System Properties : Survey and Taxonomy ». Technical Report SRI-CSL-93-1, Computer Science Laboratory, SRI International, 1994.

[Saa97] M. SAALTINK. « The Z/EVES Mathematical Toolkit Version 2.2 for Z/EVES Version 1.5 ». Technical Report, ORA, 1997.

[SAE08] SAE. « AADL Standard, V2 ». Technical Report, Society of Automotive Engineers, approved in Nov. 2008.

[SAEF68] Hayden S., Fraenkel A. A., Zermelo E. and Kennison J. F.. *Zermelo-Fraenkel set theory by S. Hayden and J. F. Kennison*. C. E. Merrill Columbus, Ohio,, 1968.

[SGHP97] D. C. SCHMIDT, A. GOKHALE, T. H. HARRISON and G. PARULKAR. « A High-performance Endsystem Architecture for Real-time CORBA ». *IEEE Communications Magazine*, 14 :72–77, 1997.

[Sin96] A. SINGHAL. « Real Time Systems : A Survey », 1996.

[Sin07] F. SINGHOFF. « The Cheddar project : a free real time scheduling analyzer, http://beru.univ-brest.fr/~singhoff/cheddar/ », 2007.

[Sip96] M. SIPSER. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[SP07] F. SINGHOFF and A. PLANTEC. « AADL modeling and analysis of hierarchical schedulers ». *Ada Lett.*, XXVII(3) :41–50, 2007.

[Spi] J.M. SPIVEY. « The fuzz type-checker for Z, http://spivey.oriel.ox.ac.uk/mike/fuzz/ ».

[Spi89a] J. M. SPIVEY. *The Z notation : a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Spi89b] J. M. SPIVEY. *The Z notation : a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[SRI08] SRI/CSL. « PVS Specification and Verification System, `http://pvs.csl.sri.com/index.shtml` », 2008.

[SS89] K. SHANNON and R. T. SNODGRASS. « Mapping the Interface Description Language Type Model into C ». *IEEE Trans. Softw. Eng.*, 15(11) :1333–1346, 1989.

[Sta93] United STATES.. *RTCA, Inc., Document RTCA/DO-178B [electronic resource]*. U.S. Dept. of Transportation, Federal Aviation Administration, [Washington, D.C.] :, 1993.

[Sub06] SAE AS-2C Architecture Description Language SUBCOMMITTEE. « ARCHITECTURE ANALYSIS and DESIGN LANGUAGE (AADL) », 2006.

[Sub09] SAE AS-2C Architecture Description Language SUBCOMMITTEE. « ARCHITECTURE ANALYSIS and DESIGN LANGUAGE (AADL) », 2009.

[THPT06] A. TOSKALA, H. HOLMA, K. PAJUKOSKI and E. TIIROLA. « Utran Long Term Evolution in 3GPP ». In *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, pages 1–5, 2006.

[TMH08] Y. THIERRY-MIEG and L. HILLAH. « UML behavioral consistency checking using instantiable Petri nets ». *ISSE*, 4(3) :293–300, 2008.

[TMPHK09] Y. THIERRY-MIEG, D. POITRENAUD, A. HAMEZ and F. KORDON. « Hierarchical Set Decision Diagrams and Regular Models ». In *TACAS '09 : Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–15, Berlin, Heidelberg, 2009. Springer-Verlag.

[TYZP05] W-T. TSAI, L. YU, F. ZHU and R. PAUL. « Rapid Embedded System Testing Using Verification Patterns ». *IEEE Software*, 22(4) :68–75, 2005.

[Upe] UPENN, DEPT OF COMPUTER SCIENCE. « CHARON, `http://rtg.cis.upenn.edu/mobies/charon/` ».

[UPP] UPPAAL GROUP. « UPPAAL, `http://www.uppaal.com/` ».

[Val03] R. VALK. « *Basic definitions* », Chapitre 4, pages 41–51. Springer Verlag, Petri nets and system engineering (Claude Girault and Rudiger Valk Eds), first edition, 2003.

[VAS05] VASY-TEAM. « Construction and Analysis of Distributed Processes, `http://www.inrialpes.fr/vasy/cadp.html` », 2005.

[VAS08] VASY PROJECT - INRIA. « TRAIAN : A Compiler for E-LOTOS/LOTOS NT Specifications, `http://www.inrialpes.fr/vasy/pub/traian` », 2008.

[Ver06] T. VERGNAUD. « *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées* ». PhD thesis, TELECOM-ParisTech, 2006.

[VHPK04] T. VERGNAUD, J. HUGUES, L. PAUTET and F. KORDON. « PolyORB : a schizophrenic middleware to build versatile reliable distributed applications ». In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063, pages 106 – 119, Palma de Mallorca, Spain, Jun 2004. Springer Verlag.

[Vor] J-B. VORON. « Coloane `http://coloane.lip6.fr` ».

[VZ06] T. VERGNAUD and B. ZALILA. « Ocarina : a Compiler for the AADL ». Technical Report, Télécom Paris, 2006. available at `http://ocarina.enst.fr`.

[WKL04] V. WINTER, F. KORDON and M. LEMOINE. « *Formal Methods for Embedded Distributed Systems : How to master the complexity* », Chapitre The BART Case Study, pages 3–22. Kluwer Academic Publishers, 2004.

[Zal08] B. ZALILA. « *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture* ». PhD thesis, TELECOM-ParisTech, 2008.

[Zub91] W. M. ZUBEREK. « Timed Petri nets definitions, properties, and applications ». *Microelectronics and Reliability*, 31(4) :627–644, 1991.

Annexe A

Règles de transformation AADL - Réseaux de Petri

Cette annexe a pour but de présenter les algorithmes sous-jacents à la création et à l'assemblage de réseaux de Petri depuis une spécification AADL. Nous y explicitons les structures de données et les procédures mises en jeu lors de la génération automatique de réseaux de Petri.

A.1 Algorithme principal de génération

Considérons un système AADL *System*, que nous transformons en réseaux de Petri *PN_System*.

Comme nous l'avons montré dans les chapitres précédents, le principal composant à prendre en compte lors de la génération est le `thread`. Les `process` ou les `system` contenant un thread sont donc assimilés aux threads qu'ils contiennent. Ils ne subsistent qu'au travers de l'espace de nommage des places et des transitions constituant un thread.

Par soucis de lisibilité, nous ne le montrerons pas dans les algorithmes suivants, mais les nom des places et les transitions sont tous préfixés par les noms des processus et des systèmes les englobant.

L'algorithme 1 présente la boucle principale de génération.

Pour chaque élément thread de la spécification AADL, sont construits successivement :

1. la structure du cycle de vie (depuis l'état Halted jusqu'à Wait_For_Dispatch)
2. sa partie traitement de donnée, au travers d'appels de sous-programmes, et la réception ou l'envoi de messages
3. la gestion du déclenchement du thread
4. les ports du threads (port de paramètres pour les sous-programmes, ports de déclenchement pour les threads apériodiques ou sporadiques)
5. l'assemblage final du système : assemblage de chaque thread, puis des threads entre eux

A.2 Algorithme de création de la structure d'un thread

`processThread` crée la structure du réseau de Petri de la figure 1.

Algorithme 1 Algorithme principal de génération

```
1: PN_System ← ∅
2: foreach T ∈ getThreads(System) do
3:     PN_T ← processThread (T)
4:     processComputation (T, PN_T)
5:     processThreadDispatch (T, PN_T)
6:     processThreadPorts (T, PN_T)
7:     appendThread (PN_T) to PN_System
8: end loop
9: processPN_Assembling (PN_System)
```

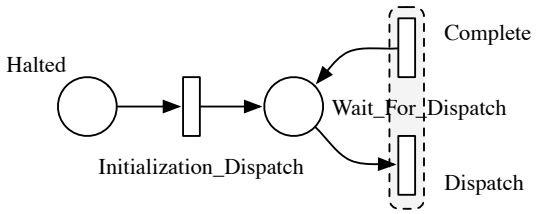


FIGURE 1 – Structure du squelette d'un thread en réseau de Petri

Algorithme 2 Création de la structure RdP d'un thread

```
1: procedure PROCESSTHREAD(T)
2:
3:     append_places (Halted, Wait_For_Dispatch) to T
4:     append_transitions (Initialization_Dispatch, Compute_Entrypoint, Complete) to T
5:
6:     connect (Halted, Initialization_Dispatch)
7:     connect (Initialization_Dispatch, Wait_For_Dispatch)
8:     connect (Wait_For_Dispatch, Compute_Entrypoint)
9:     connect (Complete, Wait_For_Dispatch)
10:
11:    case Formalism
12:        when Colored ⇒
13:            InitCPNThread (T);
14:            foreach a ∈ getArcs (T) do
15:                setValuation (a, ⟨x⟩)
16:            end loop
17:        ;;
18:        when Timed ⇒
19:            InitTPNThread (T)
20:        ;;
21:    esac
22: end procedure
```

Analyse des étapes de l'algorithme 2

$2-3, 6-9$) création du patron : ajout des places et des transitions nécessaires, connexion avec des arcs classiques. Cette étape est indépendante de tout formalisme

$11-21$) spécialisation du patron selon le formalisme choisi. Pour les réseaux de Petri colorés, les arcs doivent être valués de façon à transmettre l'identifiant des threads (lignes $14-16$)

Les procédures `InitTPNThread` et `InitCPNThread` initialisent les places et les transitions, en leur ajoutant les annotations nécessaires au formalisme dédié :

Algorithme 3 Initialisation colorée du squelette de thread

```
1: procedure INITCPNTHREAD(T)
2:     foreach p ∈ getPlaces(T) do
3:         setDomain(p, Thread_Ids)
4:     end loop
5:     setMarking(Halted, declareNEwThread(T))
6: end procedure
```

Analyse des étapes de l'algorithme 3

$2-4$) les places du patrons doivent recevoir et émettre l'identifiant du thread auquel elles appartiennent

5) marquage initial d'un thread : déclaration d'un nouvel identifiant par la primitive `declareNEwThread`. Chaque nouvelle déclaration ajoute un élément dans la déclaration de la classe de couleur

Algorithme 4 Initialisation temporelle du squelette de thread

```
1: procedure INITTPNTHREAD(T)
2:     if ∃ Initialization_Execution_Time then
3:         IET ← Initialization_Execution_Time
4:         setGuard(Initialization_Dispatch, [IET, IET])
5:     end if
6:     setMarking(Halted, 1)
7: end procedure
```

Analyse des étapes de l'algorithme 7

$2-5$) Exploitation de la propriété AADL Initialization_Execution_Time répercuté sur la transition d'initialisation.

6) marquage initial du thread

A.3 Algorithme de création de la partie traitement d'un thread

Les éléments AADL à prendre en compte pour cet aspect des thread sont les séquences d'appels de sous-programme. Les patrons de réseaux de Petri (partie structurelle) sont présentés à la figure 2 et mis en œuvre par l'algorithme 5. La figure a été présentée à la section 4.3.3 de ce manuscrit.

Analyse des étapes de l'algorithme 5

$2-8$) création du patron (première partie) : ajout des transitions d'interface (Dispatch, Complete, Send et Receive) et des places intermédiaires. Cette partie est indépendante de tout formalisme

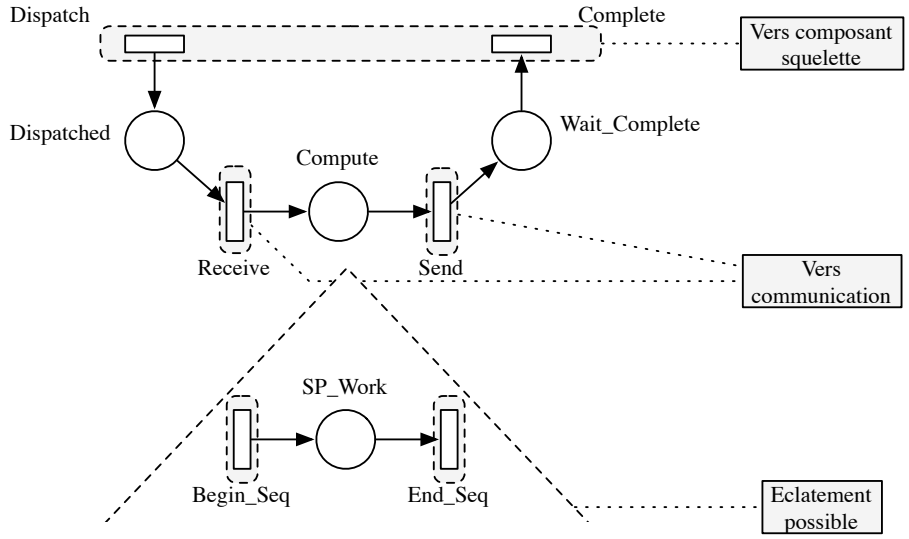


FIGURE 2 – Structure de la partie traitement d'un thread en réseaux de Petri

Algorithme 5 Structure de la partie traitement d'un thread

```
1:  procedure PROCESSCOMPUTATION(T, PN_T)
2:      append_transitions (Dispatch, Complete, Receive, Send) to PN_T
3:      append_places (Dispatched, Wait_Complete) to PN_T
4:
5:      connect (Dispatch, Dispatched)
6:      connect (Dispatched, Receive)
7:      connect (Send, Wait_Complete)
8:      connect (Wait_Complete, Complete)
9:      RCE ← ∅
10:     foreach cs ∈ getCallSequences (T) do
11:         CN_PN ← ∅
12:         foreach sp ∈ cs do
13:             CPN_PN ← buildSPPattern (sp)
14:         end loop
15:         InitRCEPattern (CN_PN)
16:         RCE ← CN_PN
17:     end loop
18:     appendRCE (RCE) to PN_T
19: end procedure
```

$10-17$) parcours des séquences d'appel du thread, création des sous-programmes des composants (l.$12-14$). Ces séquences sont ensuite initialisées selon le formalisme (l.15, algorithme 6). Regroupement des séquences (l.16)

18) ajout des nouveaux éléments dans le patron de thread en cours de construction

Algorithme 6 Initialisation de la partie traitement d'un thread

```
1:  procedure INITRCEPATTERN(RCE)
2:      case Formalism
3:          when Colored ⇒
4:              foreach a ∈ getArcs(T) do
5:                  setValuation(a, ⟨x⟩)
6:              end loop
7:          ;;
8:          when Timed ⇒
9:              foreach t ∈ getTransitions(T) do
10:                 setGuard(t, [0, 0])
11:             end loop
12:         ;;
13:     esac
14: end procedure
```

Analyse des étapes de l'algorithme 6

$4-6$) les valuations des arcs doivent être en mesure de transmettre les identifiants des threads

$9-11$) par défaut, les transitions sont initialisées comme des transitions immédiates

A.4 Algorithme de création des mécanismes de déclenchement d'un thread

Les mécanismes de déclenchement varient selon la variante de réseaux de Petri ciblée. Les déclenchements faits par l'intermédiaire des ports (apériodique, sporadique), sont traités par la procédure `processThreadPorts` (section A.5). Ici, nous nous focalisons sur les déclenchements périodiques.

Analyse des étapes de l'algorithme 7

$2-3$) ce patron n'a de sens pour que pour les réseaux de Petri temporels. La périodicité des threads est exprimée implicitement avec le cycle de vie des threads en réseaux de Petri colorés

$4-8$) création des places et des transitions nécessaires au patron de déclenchement de la figure 3. Cette figure a été présentée à la section 4.1.2 de ce mémoire

10) initialisation de ce patron. Nécessite le calcul de l'hyperpériode des threads présents dans le système

$11-14$) exploitation de l'attribut Period de la spécification AADL

A.5 Algorithme de création des ports d'un thread

Les ports pris en compte par cet algorithme sont les ports de données et les ports de donnée et d'événement. L'algorithme 8 permet de choisir quelle suite d'action choisir, en fonction du formalisme et du type de port.

Algorithme 7 Traitement des threads à déclenchement périodique

```
1:  procedure PROCESSTHREADDISPATCH(T, PN_T)
2:      case Formalism
3:          when Timed ⇒
4:              append_places (Hyperperiod_Bound, Clock) to PN_T
5:              append_transitions (Period_Event) to PN_T
6:              connect (Hyperperiod_Bound, Period_Event)
7:              connect (Period_Event, Hyperperiod_Bound)
8:              connect (Clock, Dispatch)
9:
10:             setMarking (Hyperpriod_Bound, computeHyperperiod (System))
11:             if ∃ Period then
12:                 P ← Period
13:                 setGuard (Period_Event, [P, P])
14:             end if
15:         ;;
16:     esac
17: end procedure
```

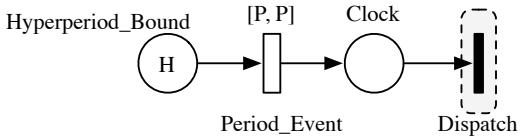


FIGURE 3 – Patron de déclenchement de thread périodiques en réseaux de Petri temporels.

Algorithme 8 Traitement des ports d'un thread

```
1:  procedure PROCESSTHREADPORTS(T, PN_T)
2:      PN_P ← ∅
3:      foreach p ∈ getPorts (T) do
4:          if ¬ Handled (p) then
5:              append_transitions (Push, Pop) to PN_P
6:              case Formalism
7:                  when Colored ⇒
8:                      buildCPNPort (p, PN_P)
9:                  ;;
10:                 when Timed ⇒
11:                     buildTPNPort (p, PN_P)
12:                 ;;
13:             esac
14:             setHandled (p, PN_P)
15:             appendPort (PN_P) to PN_T
16:         else
17:             appendPort (getHandled(p)) to PN_T
18:         end if
19:     end loop
20: end procedure
```

Analyse des étapes de l'algorithme 8

3) parcours de tous les ports du thread considéré

4) création du patron de port si et seulement si il n'a pas été déjà créé : un port de communication est référencé par deux threads (la source et la cible). Le port est créé par un premier thread. Le second ne récupère qu'une référence sur le port déjà créé (l.17)

5) ajout des transitions d'interface communes à tous les ports, et à tous les formalismes

8, 11) construction du port selon le formalisme (respectivement algorithmes 9 et 10)

14) notification que le port a été pris en compte

15, 17) ajout du patron de port dans le patron de thread

Algorithme 9 Traitement des ports d'un thread pour les réseaux de Petri colorés

```
 1: procedure BUILDCPNPORT(p, PN_P)
 2:     PN_P ← ∅
 3:     case Kind (p)
 4:         when Data ⇒
 5:             append_places (Storage) to PN_P
 6:             connect (Push, Storage) with (⟨x, data⟩)
 7:             connect (Storage, Push) with (⟨x, d⟩)
 8:             connect (Pop, Storage) with (⟨x, d⟩)
 9:             connect (Storage, Pop) with (⟨x, d⟩)
10:             setDomain (Storage, message)
11:             setMarking (Storage, ⟨0, data⟩)
12:         ;;
13:         when DataEvent ⇒
14:             append_transitions (Overflow) to PN_P
15:             buildFIFOStep (Queue_Size, PN_P)
16:             case Overflow_Policy
17:                 when Error ⇒
18:                     ∀ Slot • consume_tokens (Slot)
19:                 ;;
20:                 when DropOldest ⇒
21:                     shift token from first slots to last
22:                 ;;
23:                 when DropNewest ⇒
24:                     ignore new incoming value
25:                 ;;
26:             esac
27:         ;;
28:     esac
29: end procedure
```

Analyse des étapes de l'algorithme 9

$5-9$) gestion des ports de donnée : création et connexion des places et des transtitions composant le patron présenté figure 4 (section 4.3.4 du manuscrit). La structure est indépendante du formalisme

$10-11$) le marquage initial (l.11) est un marquage induit par la modélisation, pour éviter un blocage au démarrage du système : il est remplacé par le premier message arrivant dans le port

14) ajout d'une interface supplémentaire pour le port de donnée et d'événement

15) exploitation de l'attribut Queue_Size : donne le nombre d'étages de la FIFO. Un étage de FIFO correspond au patron décrit figure 5. La place Empty est le dual de la place Slot, et indique si une case est libre ou non dans la FIFO. Le domaine de la place Slot est en accord avec la définition du domaine des messages échangés

$16 - 24$) Variations autour de l'attribut Overflow_Policy de la spécification AADL. Les différentes variantes sont présentées à la figure 31 de la section 4.3.4 de ce mémoire

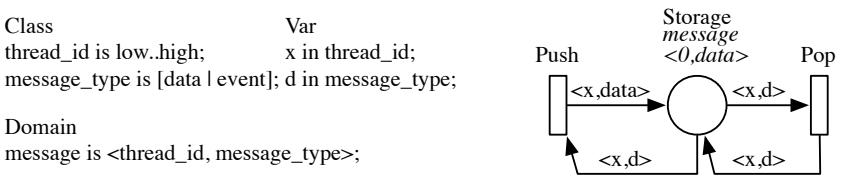


FIGURE 4 – Port de donnée AADL traduit en réseaux de Petri colorés

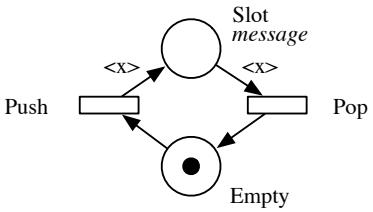


FIGURE 5 – Etage d'une FIFO en réseau de Petri colorés

Analyse des étapes de l'algorithme 10

$5 - 10$) mêmes opérations que l'algorithme 10 puisque la structure du patron est identique

$13 - 15$) construction du patron de ports données et événements

A.6 Algorithme d'assemblage du système

Les algorithmes précédents ont permis de construire pour chaque thread différents blocs, en fonction des valeurs des attributs disponibles. Les algorithmes présentés maintenant indiquent comment combiner ces blocs pour tout d'abord construire les threads entièrement, et ensuite comment les composer pour former le système final.

Analyse des étapes de l'algorithme 11

$2 - 4$) construction complète de chaque thread (algorithme 12)

$5 - 8$) ajout du patron relatif aux processeurs. Tout d'abord, création de la place représentant le processeur. Ensuite, connexion à celui-ci de toutes les parties de traitement des threads liés à lui

9) connexion des threads au travers des ports déjà créés. Résolution des ports référencés (cf. algorithme 8)

10) fusion de toutes les transitions d'initialisation des threads, conformément au standard AADL qui indique que les initialisations sont synchronisées

$11 - 13$) gestion des observateurs et des raffinements apparaissant au niveau système si nécessaire

Algorithme 10 Traitement des ports d'un thread pour les réseaux de Petri temporels

```
1:  procedure BUILDTPNPORT(p, PN_P)
2:      PN_P ← ∅
3:      case Kind (p)
4:          when Data ⇒
5:              append_places (Storage) to PN_P
6:              connect (Push, Storage)
7:              connect (Storage, Push)
8:              connect (Pop, Storage)
9:              connect (Storage, Pop)
10:             setMarking (Storage, 1)
11:         ;;
12:         when DataEvent ⇒
13:             append_places (Bus) to PN_P
14:             connect (Push, Bus)
15:             connect (Bus, Pop)
16:         ;;
17:     esac
18: end procedure
```

Algorithme 11 Assemblage du système

```
1:  procedure PROCESSPN_ASSEMBLING(PN_System)
2:      foreach t ∈ getThreads (PN_System) do
3:          thread_Assembling (t)
4:      end loop
5:      foreach pp getProcessors (PN_System) do
6:          append_places (Proc_pp) to PN_System
7:          ∀ t ∈ getThreads (PN_System) | t related to pp • connect Proc_pp with RCEComponent
8:      end loop
9:      merge inputPorts and outputPorts
10:     merge initialisation_transitions
11:     if Observers then
12:         addGlobalObservers (PN_System)
13:     end if
14: end procedure
```

Algorithme 12 Assemblage d'un thread

```
1:  procedure THREAD_ASSEMBLING(PN_T)
2:      foreach p ∈ getPorts (PN_T) do
3:          case p.Direction
4:              when input ⇒
5:                  if isDispatchPort (p) then
6:                      merge (Pop, Dispatch)
7:                  else
8:                      merge (Pop, Receive)
9:                  end if
10:             ;;
11:             when output ⇒
12:                 merge (Pop, Send)
13:             ;;
14:         esac
15:     end loop
16:     ∀ cs ∈ getCallSequences (PN_T) •
17:         merge subprogram calls into one sequence
18:     if Observers then
19:         addLocalObservers (PN_T)
20:     end if
21: end procedure
```

Analyse des étapes de l'algorithme 12

$2-15$) pour tous les ports du thread :

6) s'ils sont en entrée et provoquent un déclenchement, leur interface d'extraction (Pop) est fusionnée avec la transition de déclenchement (Dispatch)

8) s'ils sont en entrée sans autre effet, leur interface d'extraction est fusionnée avec les transitions de réception adéquates du thread (paramètres de sous-programme)

12) s'ils sont en sortie, leur interface d'émission (Send) est fusionnée avec la transition d'envoi de donnée adéquate du thread (paramètre en sortie)

$16-17$) fusionne chaque séquence d'appels (morcelée en appels de sous-programmes) en un seul bloc séquentiel

19) gestion des observateurs au niveau local

Nous présentons ici la gestion de propriétés au niveau système :

Algorithme 13 Insertion d'observateurs au niveau système

```
1:  procedure ADDGLOBALOBSERVERS(PN_System)
2:      case Observer
3:          when Priority ⇒
4:              ∀ t ∈ getThreads (PN_System) •
5:                  ∀ u ∈ getThreads(PN_System) \ {t} | getPriority (t) ≥ getPriority (u) •
6:                      connect (t.Dispatch, u.Dispatch)
7:          ;;
8:      esac
9:  end procedure
```

Analyse des étapes de l'algorithme 13

$3-6$) exploitation de l'attribut Priority de la spécification AADL. Des relations d'ordre entre les transitions de déclenchement de chaque thread du système sont créées par l'intermédiaire d'arc de priorité

Algorithme 14 Insertion d'observateurs au niveau système

```
1:  procedure ADDLOCALOBSERVERS(PN_T)
2:      case Observer
3:          when Stamp ⇒
4:              ∀ p ∈ getPorts (PN_T) | p.Direction = output •
5:                  append_places (Stamp) to PN_T
6:                  setDomain (Stamp, Msg_Ids)
7:                  setMarking (Stamp, ⟨Msg_Ids.all⟩)
8:                  connect (Stamp, Push)
9:          ;;
10:         when Scheduling ⇒
11:             append_places (Working) to PN_T
12:             connect (Dispatch, Working)
13:             connect (Working, Complete)
14:             caseScheduling Property
15:                 when χ ⇒
16:                     append_transitions (χ_OBS) to PN_T
17:                     append_places (χ_Property) to PN_T
18:                     connect (χ_OBS, χ_Property)
19:                     connect (usefull places, χ_OBS)
20:                 ;;
21:             esac
22:         ;;
23:     esac
24: end procedure
```

Analyse des étapes de l'algorithme 14

$4-9$) gestion de l'estampillage des messages pour les réseaux de Petri colorés. Ajout de la place correspondant au patron (l.5 à 8, figure 42, section 4.3.6 du manuscrit). Cela est appliqué pour chaque port en sortie du thread (l.4)

$10-22$) gestion des observateurs d'ordonnancement d'un thread. Ils nécessitent tous la création de la place Working indiquant si le thread est en train d'effectuer une tâche ou non (l.$11-13$).
Selon la propriété à vérifier, les mécanismes à mettre en place sont les mêmes : ajout d'une transition d'observation (l.16), qui monitore les places adéquates (l.19). Si un événement est détecté, une place prévue à cet effet (l.17) est marquée. La transition de surveillance peut être annotée d'une garde spécifiant un délai significatif (deadline à ne pas dépasser)
Le tableau suivant précise ces éléments pour différentes propriétés.

Propriété	AADL	Places à surveiller	Garde additionnelle
Activation	Activation_Deadline	Working, Clock	-
WCET	Compute_Execution_Time	Working	$[WCET, WCET]$

Annexe B

Réseaux de Petri générés pour le cas d'étude

B.1 Réseau de Petri Coloré

Nous présentons le réseau de Petri coloré généré à l'aide d'Ocarina pour le cas d'étude.

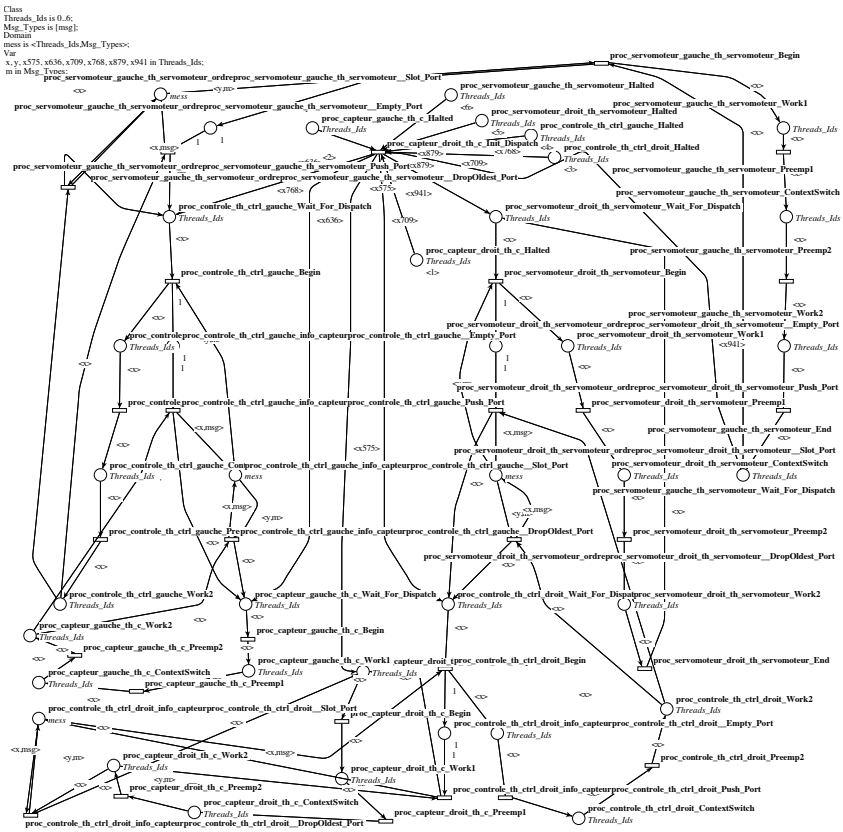


FIGURE 1 – Réseau de Petri coloré généré pour le cas d'étude

B.2 Réseau de Petri temporel à priorité

Nous présentons le réseau de Petri temporel à priorité généré à l'aide d'Ocarina pour le cas d'étude [7].

FIGURE 2 – Réseau de Petri temporel généré pour le cas d'étude

7. Nous déplorons la qualité de l'image due à des problèmes d'E/S du logiciel sur notre système d'exploitation

Annexe C

Préconditions des opérations de la spécification Z de PolyORB

Les préconditions des opérations de la spécification sont automatiquement calculées par Z/Eves. Il reste alors à la charge de l'utilisateur de prouver que ces préconditions peuvent être satisfaites, permettant l'application des opérations concernées.

La définition des préconditions d'une opération a été donnée dans la section 5.2.2.

Nous présentons ci-après les préconditions des opérations que nous avons prouvé. A chaque fois, nous prouvons que quelles que soient les variables en entrée, et l'état «avant», alors il est possible d'appliquer l'opération ciblée. Pour prouver les préconditions des opérations robustes, nous prouvons les préconditions des parties «OK» et «ERR» de chacune d'elles, avant de prouver celles de l'opération en elle même :

theorem topExportServantOKPRE
$\forall\, TOrb, inObject? : TObject$
$\mid \#managedServants < MaxServant$
• pre *opExportServantOK*

theorem topExportServantERRPRE
$\forall\, TOrb, inObject? : TObject$
$\mid \#managedServants \geq MaxServant$
• pre *opExportServantERR*

theorem topExportServantPRE
$\forall\, TOrb, inObject? : TObject$
• pre *opExportServant*

theorem topCreateReferenceOKPRE
$\forall\, TOrb, inServant? : TServant$
$\mid tapList \neq \emptyset$
• pre *opCreateReferenceOK*

theorem topCreateReferenceERRPRE
$\forall\, TOrb, inServant? : TServant$
$\mid tapList = \emptyset$
• pre *opCreateReferenceERR*

theorem topCreateReferencePRE
$\forall\, TOrb, inServant? : TServant$
$\bullet$ pre *opCreateReference*

theorem topExportReferencePRE
$\forall\, System, inReference? : TReference$
$\bullet$ pre *opExportReference*

theorem topCreateRequestPRE
$\forall\, inReference? : TReference, inMethodId? : TRequest$
$\bullet$ pre *opCreateRequest*

theorem topSendMessageOKPRE
$\forall\, System, inRequest? : TRequest$
$\mid \exists\, i : \mathrm{dom}\, orbs$
$\bullet \exists\, p : first\ inRequest?$
$\bullet\ (p.orbId = (orbs(i)).tsapName)$
$\wedge\ (p.protocolStack \in (orbs(i)).tapList)$
$\bullet$ pre *opSendMessageOK*

theorem topSendMessageERRPRE
$\forall\, System, inRequest? : TRequest$
$\mid \forall\, i : \mathrm{dom}\, orbs$
$\bullet \neg\, (\exists\, i : \mathrm{dom}\, orbs$
$\bullet \exists\, p : first\ inRequest?$
$\bullet\ (p.orbId = (orbs(i)).tsapName)$
$\wedge\ (p.protocolStack \in (orbs(i)).tapList))$
$\bullet$ pre *opSendMessageERR*

theorem topSendMessagePRE
$\forall\, System, inRequest? : TRequest$
$\bullet$ pre *opSendMessage*

theorem topReceiveMessageOKPRE
$\forall\, TOrb, inMessage? : TMessage$
$\mid \#requestQueue + 1 \leq MaxRequestQueue$
$\bullet$ pre *opReceiveMessageOK*

theorem topReceiveMessageERRPRE
$\forall\, TOrb, inMessage? : TMessage$
$\mid \#requestQueue + 1 > MaxRequestQueue$
$\bullet$ pre *opReceiveMessageERR*

theorem topReceiveMessagePRE
$\forall\, TOrb, inMessage? : TMessage$
$\bullet$ pre *opReceiveMessage*

theorem topActivationOKPRE

$\forall\, TOrb, inRequest? : TRequest$
$\mid (\mathrm{dom}\, managedServants \neq \emptyset$
$\land\ first\ inRequest? \neq \emptyset$
$\land\ (\exists p : first\ inRequest? \bullet (p.servantId \in \mathrm{dom}\, managedServants))$
$\land\ (\exists p : first\ inRequest?$
$\quad \mid p.servantId \in \mathrm{dom}\, managedServants$
$\quad \bullet\ first\ (managedServants(p.servantId)) \neq \emptyset)$
$\land\ (\exists p : first\ inRequest?$
$\quad \mid p.servantId \in \mathrm{dom}\, managedServants$
$\quad \land\ first\ (managedServants(p.servantId)) \neq \emptyset$
$\quad \bullet\ second\ inRequest? \in (first\ (managedServants(p.servantId))))$
$\bullet$ pre *opActivationOK*

theorem topActivationERRPRE

$\forall\, TOrb, inRequest? : TRequest$
$\mid \neg\, (\mathrm{dom}\, managedServants \neq \emptyset$
$\land\ first\ inRequest? \neq \emptyset$
$\land\ (\exists p : first\ inRequest? \bullet (p.servantId \in \mathrm{dom}\, managedServants))$
$\land\ (\exists p : first\ inRequest?$
$\quad \mid p.servantId \in \mathrm{dom}\, managedServants$
$\quad \bullet\ first\ (managedServants(p.servantId)) \neq \emptyset)$
$\land\ (\exists p : first\ inRequest?$
$\quad \mid p.servantId \in \mathrm{dom}\, managedServants$
$\quad \land\ first\ (managedServants(p.servantId)) \neq \emptyset$
$\quad \bullet\ second\ inRequest? \in (first\ (managedServants(p.servantId))))$
$\bullet$ pre *opActivationERR*

theorem topActivationPRE

$\forall\, TOrb, inRequest? : TRequest$
$\bullet$ pre *opActivation*

theorem topBeginExecutionOKPRE

$\forall\, TOrb, inObject? : TObject$
$\mid \#memory < MaxMemory$
$\bullet$ pre *opBeginExecutionOK*

theorem topBeginExecutionERRPRE

$\forall\, TOrb, inObject? : TObject$
$\mid \#memory \geq MaxMemory$
$\bullet$ pre *opBeginExecutionERR*

theorem topBeginExecutionPRE

$\forall\, TOrb, inObject? : TObject$
$\bullet$ pre *opBeginExecution*

theorem topEndExecutionOKPRE
$\forall\, TOrb, inObject? : TObject$
$\mid inObject? \in memory$
$\bullet$ pre *opEndExecutionOK*

theorem topEndExecutionERRPRE
$\forall\, TOrb, inObject? : TObject$
$\mid inObject? \notin memory$
$\bullet$ pre *opEndExecutionERR*

theorem topEndExecutionPRE
$\forall\, TOrb, inObject? : TObject$
$\bullet$ pre *opEndExecution*

theorem topExecutionPRE
$\forall\, TOrb, inObject? : TObject$
$\bullet$ pre *opExecution*