HAL open science

Intégration de la sécurité et de la sûreté de fonctionnement dans la construction d'intergiciels critiques

Julien Delange

▶ To cite this version:

Julien Delange. Intégration de la sécurité et de la sûreté de fonctionnement dans la construction d'intergiciels critiques. domain_other. Télécom ParisTech, 2010. Français. NNT : . pastel-00006301

HAL Id: pastel-00006301
https://pastel.hal.science/pastel-00006301v1

Submitted on 10 Sep 2010

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

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

TELECOM ParisTech **EDITE de Paris**
Département informatique et réseau

Intégration de la sécurité et de la sûreté de fonctionnement dans la construction d'intergiciels critiques

THÈSE

présentée et soutenue publiquement le 5 juillet 2010

pour l'obtention du

Doctorat de l'École Nationale Supérieure des Télécommunications
spécialité : Informatique et Réseaux

par

Julien Delange

Composition du jury

Rapporteurs :	Yvon Kermarrec	Professeur, TELECOM Bretagne
	Laure Petrucci	Professeur, LIPN - Université Paris 13 Nord
Examinateurs :	Oleg Sokolsky	Professeur, UPENN - Université de Pennsylvanie
	Jean-Charles Fabre	Professeur, LAAS - Institut National Polytechnique de Toulouse
Invité :	Jean-Paul Blanquart	EADS/Astrium Satellites
Directeurs :	Laurent Pautet	Professeur, TELECOM ParisTech
	Fabrice Kordon	Professeur, LIP6 - Université Pierre & Marie Curie

Laboratoire Traitement et Communication de l'Information — UMR 5141

Remerciements

Trop de personnes à remercier. Nous allons donc faire simple. Je remercie avant tout mes directeurs de thèse, Laurent Pautet et Fabrice Kordon. Ces remerciements se justifient par leur simple acceptation de supporter le mauvais élève que je suis. Merci à leur attention, mais surtout à leur patience et à leur contribution dans le travail que j'ai mené.

Je remercie également Peter Feiler, Jörgen Hanssen et bien entendu, Bruce Lewis pour leur accueil chaleureux lors de mon voyage au *Software Engineer Institute*, à Pittsburgh. Ce voyage m'a donné l'opportunité de découvrir un autre pays fin 2008, en plein changement. Je n'aurais jamais pensé venir aux Etats-Unis pendant ma thèse et encore moins y vivre une telle aventure. Je les remercie encore une fois, ces échanges m'ont permis de mettre en valeur mon travail de thèse à plusieurs reprises.

Je remercie également les membres du comité AADL pour leur contribution à la rédaction de l'annexe ARINC653 pour ce standard de modélisation. Je pense particulièrement à Thierry Cornillaud ou Oleg Sokolsky qui ont été de précieux relecteurs, traquant les erreurs et autres coquilles que j'avais pu laisser.

Enfin, j'aimerais encore remercier des dizaines de personnes qui m'ont aidé de près ou de loin dans mon travail. Je pense à Jérôme Hugues qui a été l'instigateur de discussions intéressantes sur la génération de code, à François Goudal, Julian Pidancet ou Laurent Lec qui ont été de très bons élèves et d'excellents contributeurs au projet que j'ai mené. Merci aussi à Bechir Zalila ou Irfan Hamid pour leur accueil à mon arrivée au sein du département Informatique et Réseaux de TELECOM ParisTech.

Enfin, merci à tous ceux qui ont été oubliés : amis ou ennemis d'un jour, personnes rencontrées au détour de conversations enflamées. Rien n'aurait été pareil sans vous et c'est ce mélange d'oppositions, de contradictions ou de complicité qui ont rendu ces trois années épuisantes mais passionantes.

« Les affaires de famille, ça force le respect »
Les Tontons Flingueurs

Table des matières

1
Introduction générale

1.1 Contexte général 12
1.2 Garantie de sécurité/sûreté : la métaphore de l'immeuble . . . 12
1.3 Difficultés de conception : causes et conséquences 18
1.4 Positionnement, définition des objectifs 24
1.5 Plan du mémoire 25

2
Problématique **27**

2.1 Approches de sécurité 29
2.2 Approches de sûreté 42
2.3 Processus de conception, contraintes et standards 50
2.4 Limites des solutions existantes 60

3
Approche proposée **67**

3.1 Identification des solutions pertinentes pour la constitution de notre approche 69
3.2 Présentation de l'approche proposée 72
3.3 Langages de modélisation 79

4
Spécification et validation de l'architecture **91**

4.1 Avant-propos : standardisation des patrons de modélisation . . 93
4.2 Modélisation de l'architecture partitionnée 94
4.3 Validation de l'architecture partitionnée 109
4.4 Modélisation des aspects sûreté 113
4.5 Validation des aspects sûreté 118
4.6 Modélisation des aspects sécurité 124
4.7 Validation des aspects sécurité 129

5
Implantation et certification automatiques **135**

5.1 Aperçu du processus de génération 138
5.2 Chaîne de génération 141
5.3 Patrons de génération de code 143
5.4 Plate-forme pour systèmes sûrs et sécurisés 158
5.5 Certification des exigences de la spécification 167
5.6 Assurance des exigences des standards de certification 172

6
Cas d'études **179**

6.1 Mise en avant de l'approche, éléments étudiés 181
6.2 Cas d'étude *« integrated avionics »* : gestionnaire de pilotage . 182
6.3 Cas d'étude *« MILS »* 191
6.4 Analyse de la couverture de code 201
6.5 Étude de l'empreinte mémoire 203

7
Conclusions et Perspectives **209**

7.1 Conclusions 209
7.2 Perspectives 212

Bibliographie **217**

A
Publications & glossaire **223**

A.1 Publications 224
A.2 Glossaire 226

B
Code source **229**

B.1 Code de POK 230
B.2 Théorèmes de validation implémentés avec REAL 233
B.3 Librairie de componsants 240
B.4 Représentation textuelle des cas d'études 247
B.5 Code source de test des cas d'études 275

Abstract

Safety-critical software (used in avionics, military or aerospace domains) must preserve their integrity, ensure a continuous operational state and enforce security of their data. There requirements are met through a dedicated development process that analyses and detects errors before system release.
However, these methods are not sufficient and safety or security still occurs in such systems (e.g. explosion of Ariane 5, mission failure of Mars Climate Orbiter, etc). In addition, meeting safety and security becomes more and more difficult due to an increasing number of functionnalities.
This thesis introduces a new method to build safety-critical systems and ensure their safety and security requirements. The approach proposes patterns for the specification of safe and secure systems. Then, a dedicated development process relies on them to (i) validate, (ii) automatically implement and (iii) certify the system, enforcing its requirements from the specifications to the code.
System validation (i) detects specification errors, ensuring its correctness and feasibility prior any development effort. The automatic implementation process (ii) translates system specification into code and ensures their requirements enforcement. The certification (iii) aspect verifies that specification requirements are met in the implementation by analyzing the system during its execution. It also evaluates its compliance against certification standards (such as DO178B).

Keywords : safety, security, ARINC653, MILS, code generation, partitioned system, POK, AADL, Ocarina.

Résumé

Les systèmes embarqués critiques (utilisés dans le domaine avionique, militaire ou médical) doivent assurer une continuité de service et la sécurité des données qu'ils contiennent ou échangent. La garantie de ces exigences s'effectue au travers d'un processus de développement rigoureux qui s'attache à détecter et corriger toute erreur avant la mise en production du système.
Toutefois, plusieurs exemples (explosion de la fusée *Ariane 5*, échec de la mission *Mars Climate Orbiter*) ont montré les limites de ces méthodes. De plus, l'augmentation des fonctionnalités fournies par ces systèmes complique la garantie de règles de sécurité et de sûreté.
Ce travail de thèse propose une méthode de conception de systèmes critiques visant à faciliter le respect des politiques de sécurité et de sûreté dans la production de systèmes critiques. L'approche décrite au sein de ce manuscrit définit des règles de spécifications des systèmes sûrs et sécurisés qui sont utilisés au cours d'un cycle de développement qui (i) valide, (ii) implante et (iii) certifie automatiquement le système.
La validation de l'architecture (i) assure la bonne constitution des spécifications et garantit leur faisabilité. L'implantation automatique (ii) génère le système à partir des spécifications validées, garantissant la bonne traduction des spécifications en code exécutable. L'aspect certification (iii) compare l'exécution du système avec ses spécifications et vérifie sa conformité avec les standards de certification inhérents aux systèmes critiques. Cette partie de notre approche assure que les systèmes implantés respectent les contraintes de sécurité et de sûreté décrites par l'utilisateur.

Mots-clés : sûreté, sécurité, ARINC653, DO178B, MILS, génération de code, système partitionné, POK, AADL, Ocarina.

Chapitre 1

Introduction générale

Sommaire du chapitre

1.1 Contexte général 12
1.2 Garantie de sécurité/sûreté : la métaphore de l'immeuble 12
1.2.1 Contexte 13
1.2.2 Assurance de la sécurité 14
1.2.3 Assurance de sûreté 15
1.2.4 Assurance de sécurité et la sûreté : difficultés 17
1.3 Difficultés de conception : causes et conséquences . 18
1.3.1 Causes de la difficulté de garantie de sécurité/sûreté 18
1.3.2 Conséquences sur la sécurité/sûreté 20
1.4 Positionnement, définition des objectifs 24
1.4.1 Définition des objectifs 24
1.5 Plan du mémoire 25

1.1 Contexte général

Jour après jour, nous utilisons toujours plus de systèmes embarqués. Ces derniers prennent place dans différents domaines de notre vie quotidienne : loisirs (consoles de jeux portables, lecteurs MP3), transport (pilotage automatique du métro, contrôle de vitesse etc.) Si des défaillances n'ont souvent que peu d'importance pour la majorité des utilisateurs, certains systèmes, dits *« critiques »* (par ex : domaine avionique ou médical), doivent en revanche garantir leur bon fonctionnement ou la protection des données qu'ils manipulent (par ex : domaine militaire). Une erreur de conception peut être catastrophique et causer l'abandon d'une mission ou des pertes humaines. Si le grand public n'est pas toujours acteur dans l'utilisation de ces systèmes, il n'en est pas moins utilisateur (passager d'un vol, conducteur de voiture, etc.) et subit les conséquences de ces défaillances.

Cependant, la conception de ces systèmes est devenue de plus en plus compliquée de par l'ajout constant de fonctionnalités. Ces nouvelles capacités compliquent l'architecture des systèmes, rendant la garantie de leur fonctionnement difficile. Cette complexité est d'autant plus préoccupante car certains aspects apparaissant comme triviaux peuvent avoir de lourdes conséquences sur le fonctionnement du système. Le cas de l'explosion de la fusée Ariane (erreur de conception du logiciel de pilotage [43]) ou du drône explorateur de Mars (erreur dans les spécifications des unités dans le logiciel de contrôle du drône [75]) montrent que certaines erreurs qui peuvent sembler triviales (par exemple, erreur dans l'intégration du système métrique) peuvent avoir de lourdes conséquences.

Les sections suivantes illustrent cette complexité. D'abord en partant d'un exemple simple de notre vie quotidienne (la construction d'un immeuble). Puis, en établissant une correspondance entre les problèmes identifiés avec ceux que rencontrés dans la production de systèmes critiques. Enfin, la fin de ce chapitre introduit l'approche développée tout au long de ce manuscrit.

1.2 Garantie de sécurité/sûreté : la métaphore de l'immeuble

Pour illustrer les principes de sécurité et de sûreté et la difficulté qu'ils impliquent, nous transposons ces problématiques dans le cadre de la construction d'un immeuble contenant des locaux hébergeant deux entreprises concurrentes : A et B. Au vu du conflit d'intérêt entre ces deux entreprises, il est nécessaire de les isoler l'une de l'autre afin d'éviter la fuite d'information. Cependant, les aspects économiques (coûts des locaux, de gestion, etc.) contraignent les entreprises à partager certaines ressources (gestion du courrier). Un aperçu global de l'architecture est proposé dans la figure 1.1.

Les accès autorisés sont illustrés dans la figure 1.2 : les employés de chaque entreprise pénètrent dans les locaux via une entrée dédiée à l'entreprise, empêchant ainsi la rencontre d'employés d'entreprises différentes et l'échange potentiel d'information entre eux. Ils peuvent circuler librement entre les bureaux et la salle d'archive par un escalier et couloir dédiés (les déplacements entre

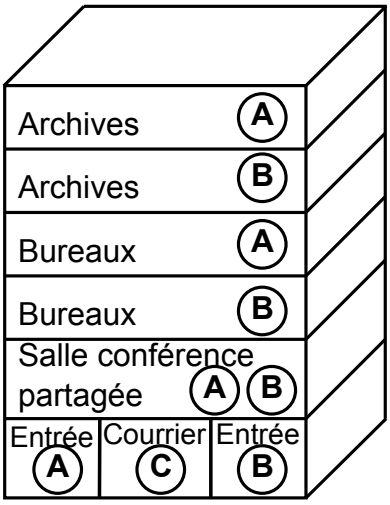


FIGURE 1.1 – Aperçu de l'architecture de l'immeuble

employés de chaque entreprise sont donc confinés dans un canal de communication dédié). La salle de réunion est accessible aux deux entreprises. Cependant, afin d'éviter la rencontre de personnes issues de sociétés différentes, l'accès à cette salle est restreint (ces mécanismes sont explicités dans la sous-section 1.2.2). Enfin, une dernière entité indépendante, C, transporte le courrier destiné à chaque entreprise. Afin d'éviter le vol de données, deux couloirs sont mis en place pour acheminer le courrier : un de C à A et un de C à B. Seuls les employés de l'entité C sont habilités à utiliser ces couloirs, empêchant que les employés de A ou B ne volent les plis qui ne leur sont pas destinés. Ainsi, les informations de chaque entreprise sont isolées, C garantissant leur bon acheminement.

Les paragraphes suivants expriment les mécanismes de sécurité/sûreté de cette métaphore pour contrôler la dissémination de l'information et assurer la garantie de bon fonctionnement de chaque équipe.

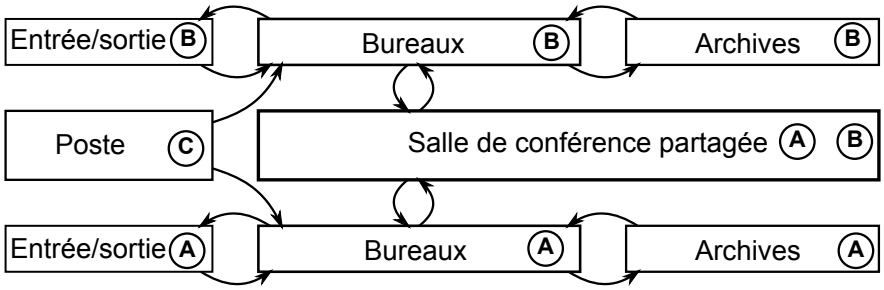


FIGURE 1.2 – Plan d'accès de l'immeuble

1.2.1 Contexte

Ces deux entreprises ne doivent pas s'échanger d'information. Il est donc nécessaire que leurs locaux soient isolés, de telle manière à ce que l'une ne

puisse voler les informations de l'autre. Cette fuite d'information peut avoir des sources diverses (employé mal intentionné, intrusion et vol d'information) qu'il convient alors d'identifier, d'analyser et supprimer.

Pour des raisons évidentes de coût, certaines infrastructures sont mises en commun :

- une salle de conférence. Celle-ci peut-être utilisée par chaque entreprise à des plages d'heures déterminées. Elle dispose de deux portes d'accès (figure 1.1), une pour chaque entreprise. Chaque entreprise a accès à cette salle à des heures déterminées.
- la gestion du courrier. Ce service est réalisé par une entité indépendante, C, financée à part égale par les deux entreprises A et B.

1.2.2 Assurance de la sécurité

Définition : Sécurité

Ensemble de fonctions et mécanismes destinés à protéger l'information des entités n'ayant pas l'autorisation de les manipuler.

Afin de garantir l'isolation de l'information entre les deux sociétés, plusieurs mécanismes sont mis en jeu dans la construction et l'exploitation du bâtiment.

Protection des sociétés par isolation des locaux

Pour assurer la protection de l'information de chaque société, bureaux et archives sont isolés dans un étage séparé, accessible aux seuls employés des entreprises concernées. Cette spécificité est visible sur l'architecture générale du bâtiment (figure 1.1), où un étage est réservé pour les bureaux et salles d'archives de chaque entreprise.

Contrôle des accès

Afin que les employés de chaque société ne puissent communiquer par l'utilisation d'un canal de communication commun (couloir ou escalier), des accès dédiés à chaque entreprise sont mis en place :

1. de l'extérieur vers les bureaux. Ainsi, une société ne peut accéder aux bureaux de sa concurrente de l'extérieur.
2. des bureaux vers les archives. Un escalier relie les bureaux de chaque société à sa salle d'archive.

Pour assurer que seuls les employés habilités franchissent la porte de leur société, un contrôle d'accès est mis en place. Il consiste à vérifier l'identité de l'employé à l'aide d'une carte, assurant que la personne qui se présente appartient bien à la société propriétaire des locaux accédés.

Contrôle du temps d'accès

La salle de conférence est cependant accessible aux deux entreprises. Ce double accès pourrait être exploité comme un canal caché pour accéder à l'entreprise concurrente, voire à une fuite d'information (un employé mal intentionné laisse des documents à destination de la société concurrente).

Pour assurer la protection de l'information, des restrictions temporelles sont mises en place. Les employés de la première société ont un accès à la salle le matin alors que les employés de la seconde société disposent d'un accès l'après-midi. Aucun accès n'est possible en soirée ou pendant la nuit.

Lorsqu'une société accède à la salle, l'accès de son concurrent est fermé. Lors du changement d'accès (à midi et le matin), la salle est inspectée et nettoyée afin de garantir qu'aucun employé n'est resté dans la salle et qu'aucun document n'y a été laissé.

Contrôle du routage du courrier

Le courrier des deux entreprises est reçu par un service dédié qui le redirige aux sociétés concernées. Lorsqu'il est reçu, la boîte aux lettres contient alors des plis destinés aux deux sociétés, mélangeant ainsi leurs informations.

Afin de garantir la bonne séparation du courrier, un service spécial de routage, financé à parts égales entre les entreprises, se charge de rediriger le courrier aux entreprises. Des escaliers et couloirs sont construits spécifiquement pour la personne assurant ce service avec des portes contrôlées par carte d'accès. Cela assure donc que les informations d'une société ne seront pas reçues par sa concurrente.

1.2.3 Assurance de sûreté

Définition : Sûreté

Ensemble de mécanismes mis en place pour assurer la continuité de fonctionnement du système.

Tout comme pour la sécurité, la politique de sûreté doit être spécifiée et pensée dès la conception des plans du bâtiment. Les mécanismes doivent être intégrés pour (i) détecter les fautes, (ii) les recouvrir et (iii) assurer leur confinement.

Détection des erreurs et politique de recouvrement

Chaque société doit spécifier sa politique de gestion d'erreurs. Il faut donc identifier les erreurs pouvant survenir dans les locaux, mettre en place des mécanismes pour les détecter, les éviter ou les réparer.

Par exemple, la gestion d'un incendie se fera par l'ajout dans les plans (spécifications) de mécanismes de détection de fumée. L'architecte adjoindra donc des capteurs de détecteurs de fumée à divers endroits de chaque pièce.

Il faut analyser la politique de gestion des erreurs. Dans le cas de l'incendie, cela peut revêtir plusieurs formes : appel automatique des secours, ouverture des portes, déclenchement de mécanismes d'extinction, etc.

Il est ensuite important de spécifier et d'analyser les impacts de la politique de recouvrement. Dans le cas de l'incendie, une entreprise peut demander à ce que toutes ses portes s'ouvrent et que les contrôles d'accès soient désactivés. Cependant, un employé malicieux pourrait provoquer un incendie volontaire pour ouvrir les portes et laisser entrer les employés de la société concurrente.

Il est donc nécessaire :

1. d'identifier les erreurs pouvant survenir au sein du système.
2. de définir la politique de recouvrement associée à chaque erreur.
3. d'analyser les spécifications pour détecter les impacts potentiels sur la sûreté et la sécurité.

Confinement de l'incendie

Au delà de la détection et de la gestion de l'incendie, il est important de confiner sa gestion au sein d'une entreprise (autrement dit, empêcher qu'un incendie détecté impacte l'entreprise concurrente).

Lorsque la fumée est détectée dans les locaux d'une entreprise, le système de détection d'incendie veillera à ce que sa gestion ne soit effective que dans les locaux impactés. En d'autres termes, l'alarme, l'ouverture potentielle des portes ou l'ouverture des vannes d'eau n'auront lieu que dans la pièce (ou étage) où la fumée a été détectée.

Cela évite à la compagnie concurrente de déclencher sa politique de recouvrement et de subir de potentiels désagréments. Cette *isolation spatiale* assure que les fautes/erreurs (ici, l'incendie) survenant au sein d'une société n'impacte pas sa concurrente. Si ce confinement des fautes n'était pas mis en place, une société mal intentionnée pourrait alors déclencher plusieurs incendies factices (par exemple, déclenchement des détecteurs de fumée à l'aide de cigarettes) et à la productivité de l'autre société (employés devant évacuer, etc.).

Assurance de délivrance du courrier

Le courrier entrant est déposé chaque matin au service courrier, à 8H. Afin d'assurer la délivrance du courrier, celui de la première société est trié puis remis à la fin de la matinée. Celui de la seconde est trié puis remis à la fin de l'après-midi. Le service chargé du tri et de la délivrance du courrier sépare ainsi son temps de travail afin de garantir que chaque entreprise reçoit le courrier qui lui est adressé.

Si l'employé n'a pas terminé de trier le courrier de la première société à la fin de sa matinée, il délivre celui déjà trié, les plis restants étant mis de côté et triés le lendemain matin. De même, les lettres à destination de la seconde entreprise sont triés l'après-midi puis remis en fin d'après midi. Si l'intégralité du courrier n'a pu être trié, le service traitera ces plis le lendemain.

Cette isolation du temps de traitement du service courrier permet de borner le nombre de plis et garantit que la quantité de courrier reçue par une entreprise n'impacte pas la bonne réception du courrier de sa concurrente.

Si ce mécanisme de séparation du temps de traitement n'était pas mis en place, une entreprise mal intentionnée pourrait alors s'envoyer des centaines lettres dans le but de saturer le service de tri et empêcher sa concurrente de recevoir les lettres lui étant destinés. Grâce à une telle isolation, une entreprise recevra *au moins* une quantité de courrier déterminée à l'avance.

1.2.4 Assurance de sécurité et la sûreté : difficultés

Les mécanismes de sécurité et de sûreté reposent sur des choix de conception du bâtiment mais aussi une bonne utilisation des services de sécurité/sûreté au quotidien. En outre, sécurité et sûreté ne peuvent être garanties que par :

1. l'adoption de choix de conception judicieux
2. des services d'exploitation sûrs

Les paragraphes ci-dessous donnent ici un aperçu de ces deux composantes, mais aussi de l'importance de leur intégration dès la fabrication du bâtiment.

De la construction ...

Les mécanismes que avons décrits (isolation des salles, construction d'accès dédiés, installation d'équipements pour la détection d'erreurs, etc.) doivent être pensés dès la construction du bâtiment.

Rajouter un escalier, un couloir ou tout autre élément dans le bâtiment sans pour autant changer son architecture serait difficile (modification de l'existant) et pourrait introduire des failles de sécurité (par exemple, les travaux obligeraient à détruire des murs, introduisant à cette occasion de vecteurs potentiels de communication entre les entreprises et donc, des failles dans leur politique de sécurité). Il est donc primordial que cette politique de sécurité soit définie par l'architecte, dès la conception des plans.

De même, d'un point de vue sûreté, il faut garantir l'isolation des fautes, en séparant les capteurs de fumées entre les sociétés. Les détecteurs d'une société sont alors physiquement isolés les uns des autres, assurant que le déclenchement d'un détecteur n'activera pas la politique de gestion d'incendie de la société concurrente.

Au delà de cet aspect architectural, il est important de s'assurer du bon fonctionnement des mécanismes de sécurité et de sûreté. Basiquement, le choix des matériaux de construction doit être judicieux (éviter les cloisons pouvant être facilement percées) et le matériel de contrôle d'accès sûr et non défaillant.

... à l'exploitation quotidienne

En plus de la validation de la construction du bâtiment, il est important de s'assurer du fonctionnement des mécanismes de sécurité et de sûreté au quotidien.

Par exemple, le service de routage du courrier doit assurer la séparation de son temps de tri et n'effectue aucune erreur (distribution d'un pli à la société concurrente). De plus, le mécanisme de nettoyage de la salle de conférence doit assurer la bonne détection de personnes et la bonne destruction des documents laissés dans la salle.

Enfin, d'un point de vue sûreté, il faut également s'assurer que les détecteurs de fumée sont toujours en état de fonctionnement et connectés aux éléments gérant l'incendie (vanne d'eau, contrôle des portes, etc.).

1.3 Difficultés de conception : causes et conséquences

Maintenant que la description des mécanismes de sécurité/sûreté est réalisée, il est important d'expliquer en quoi leur garantie est devenue difficile, voire impossible.

Pour cela, cette section se replace dans le contexte des systèmes critiques tout en établissant un parallèle avec la métaphore de l'immeuble.

1.3.1 Causes de la difficulté de garantie de sécurité/sûreté

Augmentation des fonctionnalités

Les systèmes embarqués actuels comportent toujours plus de fonctions : un téléphone, composé il y a encore quelques années d'un simple écouteur et d'un microphone comprend désormais plusieurs puces exécutant plusieurs applications en parallèle (navigateur web, messagerie, etc.). Cette remarque qui concerne l'informatique embarquée grand public, se vérifie également dans le domaine des systèmes embarqués et critiques.

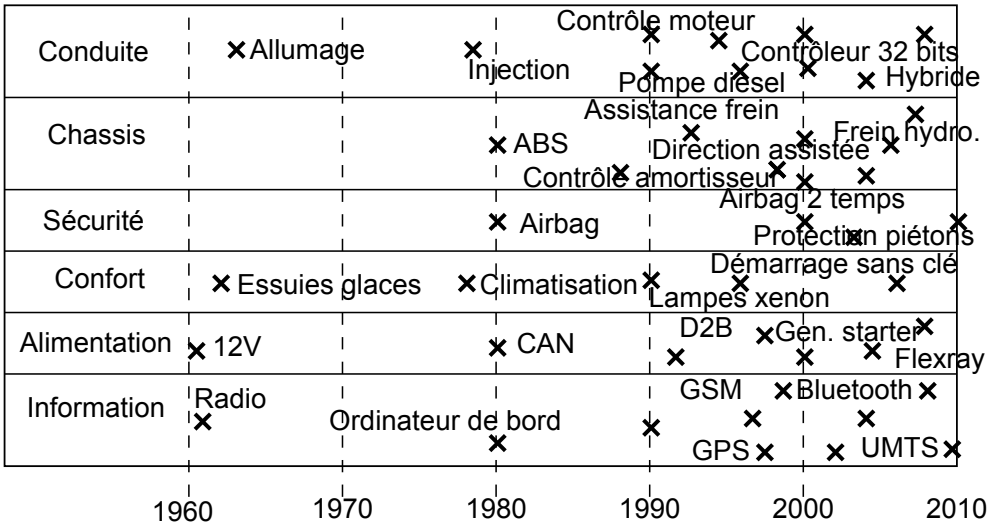


FIGURE 1.3 – Évolution du nombre de fonctionnalités dans les systèmes automobiles (source *« Hierarchical Scheduling of Complex Embedded Real-Time Systems »* - [78])

La figure 1.3 montre l'évolution des fonctionnalités dans les systèmes embarqués automobiles. Elle montre que 20 années auparavant, seules quelques fonctionnalités étaient sous le contrôle de systèmes électroniques (contrôle de l'ABS, injection du carburant, appareil de divertissement, etc.). Au fur et à mesure des années, le nombre de fonctionnalités mises en œuvre par l'électronique a augmenté de manière significative (assistance au freinage, contrôle des phares, des essuie-glaces, assistance au parking, régulateur de vitesse, etc.).

Cette croissance des fonctionnalités a compliqué la conception de ces systèmes et leur garantie de fonctionnement. Par exemple, il faut assurer qu'une fonction non-critique (système de divertissement) ou peu critique (contrôle des essuie-glaces) n'entravera pas le fonctionnement de celles particulièrement importantes (freinage).

Ces mêmes problèmes se posent dans le cadre de la conception de systèmes critiques. Par exemple, un drone fournit aujourd'hui plusieurs centaines de fonctions (auto-pilotage, géolocalisation, reconnaissance de terrain, etc.) ayant différents niveaux de criticité. Comme pour la conception de voiture, il faut garantir que les fonctionnalités peu critiques (acquisition d'image ou de son) n'entraveront pas celles plus importantes (auto-pilotage).

Dans le cas de notre métaphore de l'immeuble, cela reviendrait à rajouter toujours plus de fonctionnalités au sein du bâtiment. L'ajout d'un ascenseur ou d'une infrastructure de réseau informatique commune aux deux entreprises introduirait de nouvelles exigences pouvant impacter la politique de la sécurité ou de sûreté existante.

Une part grandissante du logiciel

Tout comme le nombre de fonctions, l'importance de l'aspect logiciel a considérablement augmenté au fil des ans. Si les fonctions d'un appareil étaient traditionnellement réalisées par des composants matériels, ces évolutions se font désormais au travers de composants logiciels, principalement pour des raisons de coût et d'adaptabilité (la mise à jour d'un composant logiciel est plus simple et moins coûteuse que la modification du matériel).

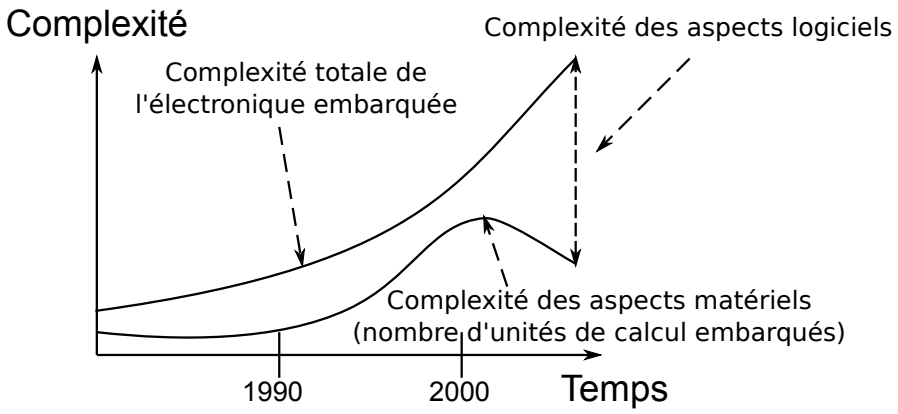


FIGURE 1.4 – Évolution de la complexité du logiciel et du matériel dans le domaine automobile (source *« Hierarchical Scheduling of Complex Embedded Real-Time Systems »* - [78])

Cependant, cette augmentation de l'importance du logiciel a compliqué le développement de ces systèmes, augmentant constamment le nombre de lignes de code (**SLOC** pour **S**ource **L**ine **O**f **C**ode). La figure 1.4 illustre cette évolution pour le secteur de l'automobile : on remarque que la complexité du logiciel a connu une croissance importante alors que celle des aspects matériels n'a que peu évolué.

La figure 1.5 montre quant à elle l'évolution du nombre de lignes de code

(**SLOC**) pour chaque série d'avions des compagnies BOEING et AIRBUS. On constate que le nombre de lignes de code évolue quasi linéairement, mais que ces entreprises ne sont plus capables de gérer des systèmes d'une telle taille. Une telle importance introduit en effet trop de difficultés d'un point de vue intégration, développement et vérification. De plus, les modes de production actuels (recours à la sous-traitance) demandent à ce que chaque partie puisse être développée indépendemment pour être intégrée plus tard. Cette méthode demande alors un effort de coordination difficile à obtenir lorsque le nombre de composants logiciels est si important.

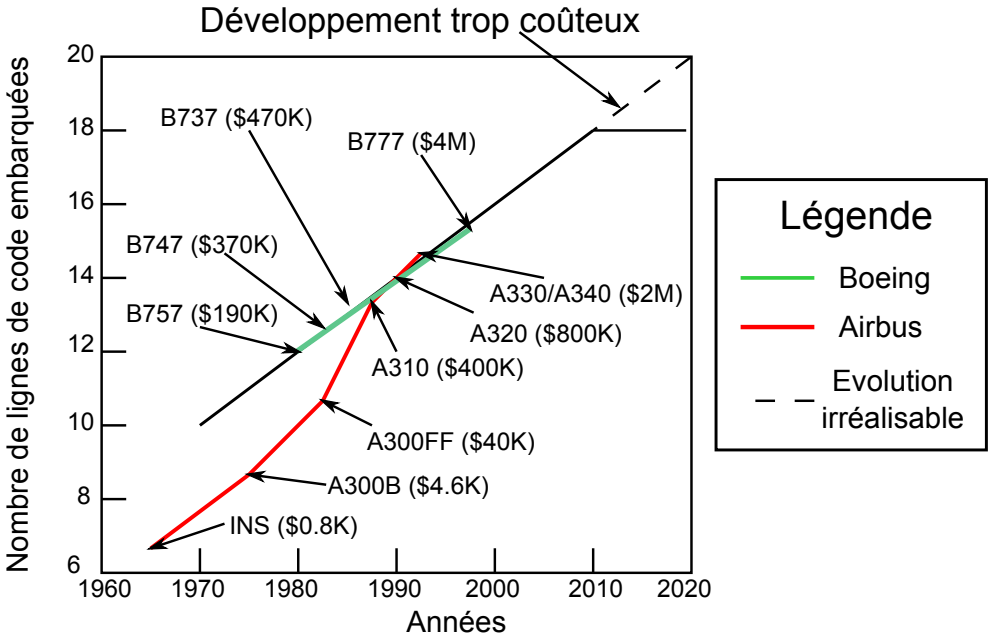


FIGURE 1.5 – Évolution du nombre de lignes de code dans les avions BOEING et AIRBUS (Source : *« System Architecture Virtual Integration : An Industrial Case Study »* - [47])

Les deux aspects décrits (augmentation des fonctionnalités et du nombre de lignes de code) ont des conséquences sur l'assurance de sécurité/sûreté. Ces éléments sont discutés dans les paragraphes suivants.

1.3.2 Conséquences sur la sécurité/sûreté

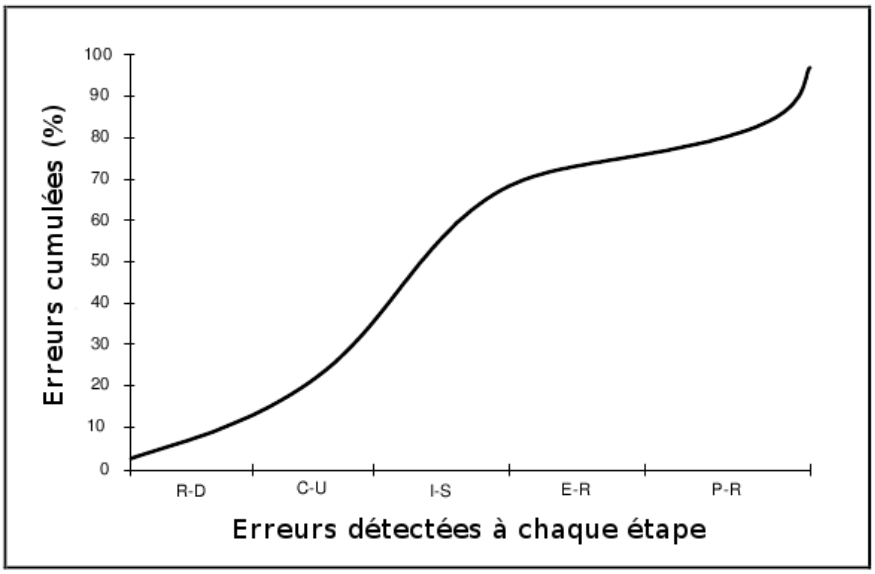
Une difficulté d'analyse du système

La croissance des fonctionnalités et donc, du nombre de lignes de code (SLOC) a pour conséquence de :

1. disséminer les exigences et les services, rendant difficile leur identification dans une quantité de code importante ;
2. introduire des interactions non souhaitées entre les différentes parties du système ;
3. rendre difficile les revues de code et l'utilisation d'outils d'analyse.

Or, la quantité de code étant toujours plus importante, l'analyse de l'impact d'une fonction sur une autre est devenue trop coûteuse. Actuellement,

comme l'illustre la figure 1.6, plus de 80% des erreurs du système sont détectées une fois le code du système produit alors que la majorité d'entre elles sont introduites lors de la spécification, avant même de commencer la phase d'implantation [77].



Légende
- ***R-D : description et analyse des exigences de l'architecture***
- ***C-U : implantation et tests unitaires***
- ***I-S : Intégration***
- ***E-R : Retour client, campagnes de tests***
- ***P-R : Après mise en production***

FIGURE 1.6 – Distribution de la localisation des erreurs au cours du développement des systèmes (Source : *« The Economic Impacts of Inadequate Infrastructure for Software Testing »* - [77])

En transposant ces problèmes sur la métaphore de l'immeuble, cette complexité serait mise en avant par l'introduction d'autres entreprises et d'autres mécanismes de sécurité et de sûreté. Dans le cas de la construction d'un immeuble hébergeant 100 entreprises, il deviendrait difficile de construire un bâtiment assurant l'isolation entre chacune d'entre elle. Dans une telle situation, il serait nécessaire de :

- veiller à ce que les dispositifs de détection d'incendie soient tous isolés ;
- assurer que le temps de traitement du service courrier soit divisé en 100 tranches de temps égales ;
- isoler les systèmes de contrôles d'identité pour l'ouverture de chaque porte.

De plus, il serait nécessaire, voire indispensable au vu du nombre d'étages de ce nouveau bâtiment, d'introduire un ascenseur partagé entre les entreprises ou d'ajouter un ascenseur pour chaque entreprise. Une telle infrastructure demanderait alors un contrôle d'accès spécial assurant l'isolation des employés de chaque entreprise. En d'autres termes, l'ajout de nouvelles entreprises dans un tel bâtiment ayant des contraintes de sécurité/sûreté fortes serait difficile, long et très coûteux.

Des outils peu adaptés

Toutefois, le besoin d'analyse, de validation et de vérification du système est toujours présent et il est impensable de mettre en production une application critique sans vérifier son bon fonctionnement au vu du impact potentiel des défaillances.

Pour ces raisons, des techniques ont été proposées pour vérifier la bonne conformité du système avec ses spécifications tout au long de son développement (analyse des spécifications et de la sûreté de fonctionnement [14, 17, 44, 16], vérification formelle [51, 14], analyse statique de code [3, 53], etc.). Malgré tout, ces techniques ont montré leurs limites, rendant difficile voire impossible la vérification des systèmes actuels :

- l'hétérogénéité des méthodes de vérification qui utilisent plusieurs formats de représentation qu'il faut alors intégrer. Cependant, ce processus d'intégration de ces méthodes est parfois impossible quand le système est trop important. Par exemple, l'expression des contraintes de sécurité et de sûreté se feront par deux formalismes différents. Il sera alors nécessaire de traduire les exigences et contraintes du système vers chaque représentation afin de vérifier chaque aspect (sécurité, sûreté). Cette étape peut être problématique au regard des différences sémantiques et conceptuelles des différentes approches (différentes manière de représenter les mêmes éléments du système, etc.).
- certaines techniques ne peuvent vérifier des systèmes trop importants de par leur taille.
- les outils actuels sont peu adaptés et ne détectent que peu d'erreurs. La figure 1.6 illustre ce phénomène et montre que plus de 70% des erreurs ne sont pas détectées avant l'intégration finale et le test du système. A l'inverse, on constate d'après la figure 1.1 que ces erreurs proviennent principalement des spécifications et pourraient être détectés plus tôt dans le processus de production.
- le coût de vérification croit plus vite que la taille du système à cause de l'augmentation des interactions potentielles entre composants logiciels. Cette caractéristique rend cette étape de la conception particulièrement critique en terme de coût.

Par conséquent, il y a un besoin important de créer de nouvelles approches de développement validant et garantissant les exigences les plus critiques, telles la sûreté et la sécurité.

Une conception coûteuse

L'augmentation de la taille du système et de ses fonctionnalités, allié à un processus de conception peu adapté rend le développement de systèmes critiques particulièrement coûteux. Deux causes sont à l'origine de ce coût prohibitif :

1. **l'absence d'analyse et de détection des erreurs** : les outils actuels ne sont pas appropriés à l'analyse des systèmes critiques (cet aspect est discuté dans la section précédente). Les erreurs introduites dans le développement sont découvertes trop tardivement (figure 1.1, demandant alors de réviser le développement du système à plusieurs reprises, aug-

Où les erreurs sont introduites	Où les erreurs sont trouvées					Total
	Analyse des exigences, conception de l'architecture	**Conception & tests unitaires**	**Intégration**	**Beta test**	**Production**	
Analyse des exigences, conception de l'architecture	3.5 %	10.5 %	35 %	6 %	15 %	70 %
Conception & tests unitaires		6 %	9 %	2 %	3 %	20 %
Intégration			6.5 %	1 %	2.5 %	10 %
Total	3.5 %	16.5 %	50.5 %	9 %	20.5 %	100 %

Table 1.1 – Localisation des erreurs et de leur découverte (Source : *« The Economic Impacts of Inadequate Infrastructure for Software Testing »* - [77])

mentent le temps (et les coûts) de développement.

2. **le manque d'automatisation du développement** : chaque étape est réalisée manuellement et séparément par des ingénieurs développant, analysant ou certifiant le système. La charge de travail relative à chaque rôle augmente en fonction du nombre de fonctions requises par le système. Cependant, les erreurs de conception n'étant détectées que tardivement, chaque étape doit être révisée à plusieurs reprises, augmentant charge de travail et coûts de production.

Cette idée est appuyée par l'étude [77] qui montre (tableau 1.2) que les coûts de développement pourraient être significativement réduits par une amélioration du processus de production. Les réductions potentielles sont induites par une diminution du nombre de développeurs et d'ingénieurs mais surtout, un nombre réduit d'erreurs à corriger lors des phases les plus avancées du développement (intégration et certification).

Pour terminer sur un parallèle avec la métaphore introduite au début de ce chapitre (section 1.2), le manque d'analyse reviendrait à ce que les propriétés de chaque matériau soit revues lors de la construction, que l'isolation soit vérifiée pour chaque étage. Si cela est coûteux mais réalisable pour un immeuble, cela est impossible lorsque plusieurs bâtiments doivent être créés quotidiennement. Dans ce contexte, le manque d'automatisation signifie que chaque élément de la construction est réalisé manuellement à partir des matières premières et qu'aucune partie du bâtiment n'est pré-construit. Si cela n'est pas envisageable dans le contexte de la construction, ce manque d'automatisation est bien présent dans la conception de systèmes critiques.

	Coûts relatifs à l'efficacité des tests (milliards de $)	**Potentielles réductions des coûts (milliards de $)**
Développeurs		
CAD/CAM/CAE et PDM	373.1	157.7
Utilisateurs		
Automobile	1229.7	377.0
Aérospatial	237.4	54.5
Total	1840.2	589.2

TABLE 1.2 – Impact financier des erreurs logicielles (Source : *« The Economic Impacts of Inadequate Infrastructure for Software Testing »* - [77])

1.4 Positionnement, définition des objectifs

Le but de cette thèse est d'accroître la garantie de sûreté et de sécurité des systèmes critiques. Ces exigences doivent être validées et correctement implantées au cours du cycle de développement (de la conception à l'exécution) :

- la spécification apporte des garanties quant aux exigences de sécurité et de sûreté (les plans de l'immeuble assurent l'isolation des entreprises).
- l'implantation est réalisée en suivant les spécifications (les ouvriers construisent effectivement le bâtiment spécifié par l'architecte), garantissant la présence des mécanismes de sécurité/sûreté.
- l'implantation est conforme à la spécification et que le système final fournit tous les services nécessaires (le maître d'ouvrage vérifie que le bâtiment construit par le maître d'œuvre correspond aux plans de l'architecte).

Dans le domaine du logiciel et du contexte de ces travaux, ce processus se traduit par trois étapes :

1. la capacité de spécifier, représenter et valider le système avec ses exigences et ses contraintes de sécurité et de sûreté (création des plans).
2. la définition de services de sécurité et la traduction des spécifications vers la réalisation (mise en place des matériaux, construction du bâtiment).
3. la vérification de la conformité du système créé avec les spécifications (vérification maître d'œuvre/maître d'ouvrage).

1.4.1 Définition des objectifs

Les éléments cités précédemment amènent à définir trois grands axes de développement pour nos travaux, chacun constituant un objectif :

1. **objectif 1** : une méthode de spécification des systèmes critiques avec leur contraintes de sécurité et de sûreté ainsi que la validation de ces mécanismes.
2. **objectif 2** : la définition d'une plate-forme garantissant la sécurité et la sûreté de fonctionnement associé à la génération automatique du système à partir des spécifications validées précédemment.

3. **objectif 3** : une certification automatisée du système qui vérifie le bon respect des contraintes de sécurité et de sûreté exprimées par les spécifications.

1.5 Plan du mémoire

Le chapitre 2 définit plus finement le contexte, en présentant l'état de l'art des solutions actuelles pour le développement de systèmes critiques sécurisés et sûrs et les contraintes de développement associées. Il évalue les critères pertinents dans la constitution de notre approche et évalue les limites de ces solutions.

Le chapitre 3 reprend les éléments pertinents des solutions existantes pour construire notre approche. Il présente la solution développée au cours de ce mémoire en mettant en avant l'architecture partitionnée, modèle retenu pour l'implantation de systèmes sécurisés et sûrs. Il donne également l'opportunité de dresser une liste des langages de spécification existants et de motiver notre choix pour AADL.

Le chapitre 4 détaille la méthode de spécification et de validation des systèmes partitionnés développée au cours de ces travaux, répondant au premier objectif défini dans 1.4. Il détaille l'extension et la spécialisation du langage AADL pour la spécification des contraintes de sécurité et de sûreté des systèmes critiques. Il explicite également les outils et mécanismes mis en œuvre pour valider les exigences à partir de cette représentation du système.

Le chapitre 5 explique les mécanismes de génération et de certification automatiques de systèmes partitionnés, répondant ainsi aux deux derniers objectifs définis (section 1.4). Il détaille les mécanismes d'implantation automatique à partir de spécifications, supprimant l'écriture manuelle de code, réduisant ainsi le nombre d'erreurs et augmentant la fiabilité globale du système. Il présente également le processus et les outils de certification automatiques développés afin de garantir la conformité des systèmes produits avec les standards de certification en vigueur.

Le chapitre 6 met à l'épreuve la solution présentée au cours des chapitres précédents en expliquant sa mise en œuvre au travers de deux cas pratiques. En particulier, il détaille la validation de la bonne implantation des mécanismes de sûreté ainsi que la vérification de la mise en œuvre de la politique de sécurité (chiffrement des communications, etc.).

Le chapitre 7 rappelle les objectifs, les résultats obtenus et présente nos conclusions. Il constitue également l'opportunité d'expliquer les limites de nos travaux et les perspectives de recherche pour leur amélioration.

Chapitre 2

Problématique

Sommaire du chapitre

2.1 Approches de sécurité 29

2.1.1 *Multiple Independent Level of Security* 29

2.1.2 Modèles de sécurité 34

2.1.3 Algorithmes de chiffrement 38

2.1.4 Synthèse des problèmes soulevés 41

2.2 Approches de sûreté 42

2.2.1 Identification et analyse des risques (approche FPTC) 42

2.2.2 ARINC653 43

2.2.3 Synthèse des problèmes soulevés 48

2.3 Processus de conception, contraintes et standards . 50

2.3.1 DO178B, un standard de certification 50

2.3.2 Les Critères Communs (Orange Book) 51

2.3.3 Assurance sur le code de l'application 55

2.3.4 Synthèse des problèmes soulevés 59

2.4 Limites des solutions existantes 60

2.4.1 Sécurité 60

2.4.2 Sûreté de fonctionnement 62

2.4.3 Méthodes de conception 63

2.4.4 Synthèse 64

Rappels

L'objectif des travaux présentés consiste à proposer une approche garantissant la sécurité et la sûreté tout au long du cycle de vie d'un système critique. L'introduction générale du précédent chapitre décrit le problème sous une forme simple, abordable.
Ce premier contact a introduit la difficulté d'assurance de sécurité et de sûreté dans le contexte de la construction de systèmes dits *« critiques »* . Il est très difficile d'assurer ces exigences tout au long du cycle de développement (des spécifications à la certification, en passant par l'implantation).
A cette complexité s'ajoutent des contraintes inhérentes à la criticité du domaine d'étude. Ces dernières imposent un cycle de conception spécifique, introduisant des méthodes dédiées et une certification obligatoire du système. Ces contraintes (issues des standards de certification comme la couverture de code), fortes, assurent le respect des exigences du concepteur (mécanismes de protection de données) et de plusieurs contraintes de conception (l'ensemble du code écrit est exécuté).

Objectifs du chapitre

Ce chapitre présente le contexte technique de cette thèse. Il décrit les approches existantes d'un point de vue sécurité, sûreté et méthodes de conception.
L'aspect sécurité présentera l'approche MILS, les politiques de sécurité existantes et les algorithmes de chiffrement existants.
La section traitant la sûreté décrit les méthodes d'analyse de la sûreté des spécifications d'un système et décrit le standard ARINC653 dédié à la sûreté de fonctionnement.
Enfin, la dernière section traite des méthodes de conception. Elle donne un aperçu du standard DO178B, référence du domaine avionique, et introduit les concepts de méthodes d'analyse de code.

2.1 Approches de sécurité

2.1.1 *Multiple Independent Level of Security*

L'objectif de MILS est d'assurer une isolation entre les différents niveaux de sécurité. Si nous reprenons la métaphore introduite dans le chapitre 1, MILS définit une méthode pour garantir l'isolation entre les deux entreprises concurrentes.

Pour ce faire, MILS (signifiant initialement *« Multiple Independent Levels of Security »*), est une approche qui définit une méthode de conception alliée à une plate-forme d'exécution dédiée dédiées à l'isolation des niveaux de sécurité. Les récents travaux (et principalement la constitution MILS [97]) spécifient cependant que le but de MILS n'est pas uniquement d'isoler les niveaux de sécurité, mais davantage de décrire comment garantir l'isolation entre plusieurs entités devant communiquer à différents niveaux de sécurité.

Bien que très cité dans la littérature et apparaissant dans les descriptions commerciales de certains produits, MILS n'est pas un standard. Cette approche initiée par John Rushby au début des années 1980, a pour seul but de définir des mécanismes pour l'analyse et l'implantation de systèmes critiques devant garantir la protection de données classifiées.

L'approche MILS vise à réduire le nombre de niveaux de sécurité simultanément présents dans chaque composant du système et à les isoler les uns des autres. Idéalement, chaque composant ne devrait manipuler des données qu'à un unique niveau de sécurité.

L'isolation des niveaux de sécurité évite qu'un même code manipulant plusieurs niveaux n'introduise une faille (par exemple, en écrivant une donnée confidentielle dans un canal de communication *non classifié*). Cette approche de division des niveaux de sécurité réduit alors les fuites potentielles d'information.

Pour illustrer l'utilité de MILS, prenons l'exemple d'un programme contrôlant deux écrans, le premier affichant des informations secrètes, l'autre affichant des informations non classifiées (cet exemple sera détaillé dans la suite, une représentation est donnée dans la figure 2.1). D'après cette spécification, l'approche MILS suggère de séparer ce programme en deux programmes distincts : l'un gérant le premier écran et ne manipulant que des données secrètes et l'autre gérant le second écran et ne contenant que des données non classifiées. Ainsi, chaque pilote gère indépendemment son niveau de sécurité.

Pour analyser les niveaux de sécurité lors de la conception du système et assurer l'isolation des niveaux de sécurité lors de son exécution, l'approche MILS repose sur deux éléments :

1. une représentation des composants du système et de leurs niveaux de sécurité associés.

2. une plate-forme d'exécution garantissant une isolation des composants.

Nous détaillons ci-après ces deux éléments.

Représentation des composants

MILS définit son propre formalisme de représentation des composants et des niveaux de sécurité. Celui-ci dissocie composants matériels et logiciels offrant la possibilité de spécifier les propriétés de sécurité. Cependant, peu d'efforts ont été réalisés pour intégrer cette méthode de représentation au sein de langages de modélisation existants tels UML, AADL ou SysML. Ce formalisme est uniquement graphique et définit une représentation ainsi que les annotations nécessaires à la spécification des contraintes de sécurité. [5, 20, 6] montrent plusieurs exemple de son utilisation.

Chaque composant spécifie :

- les niveaux de sécurité qu'il utilise (niveau de classification des données utilisées - par exemple : top-secret, secret, non classifié)
- les niveaux d'isolation maintenus entre les différents niveaux de sécurité au sein du composant. En effet, si ce dernier manipule plusieurs niveaux de sécurité, il doit alors spécifier les mécanismes d'isolation entre ces niveaux (prévention d'une fuite d'information) ou l'absence d'isolation (introduction d'un vecteur de faille de sécurité).

De cette spécification, trois catégories de composants sont définies [119] :

- Single Level of Security (SLS) : le composant ne manipule qu'un seul niveau de sécurité.
- Multiple Single Level of Security (MSLS) : le composant utilise plusieurs niveaux de sécurité et garantit leur isolation. Celle-ci doit cependant être prouvée (par tests, vérification formelle), bien que MILS n'indique pas quelles méthodes de validation doivent être utilisées, celles-ci étant laissées à la discréation du concepteur du système.
- Multiple Levels of Security (MLS) : le composant utilise plusieurs niveaux de sécurité et ne maintient aucune isolation entre eux.

Le but premier de MILS étant la séparation des niveaux de sécurité, il est préférable d'éviter d'utiliser des composants MLS, ces derniers pouvant être des vecteurs de failles ou de création de canaux cachés [119]. A l'inverse, ces composants peuvent être divisés suivant leurs niveaux de sécurité (un composant MLS contenant deux niveaux de sécurité peut être divisé en deux composants SLS manipulant chacun un unique niveau de sécurité). De tels travaux ont été menés dans [5].

Plus généralement, il est essentiel de réduire voire supprimer les composants manipulant plusieurs niveaux de sécurité. Les composants MSLS, de par la garantie d'isolation qu'ils doivent apporter, sont coûteux à développer et maintenir (utilisation de tests, méthodes formelles, etc.). La *« bonne pratique »* étant de maximiser l'utilisation des composants SLS, facilement analysables puisque ne pouvant pas créer de faille.

Exemple

L'exemple suivant illustre la représentation d'un pilote d'affichage gérant deux écrans et deux niveaux de sécurité, les informations de chaque niveau de sécurité étant affichées séparément sur un écran. La figure 2.1 présente le concept de l'approche MILS en terme de modélisation.

La partie supérieure de la figure montre l'implantation *« traditionnelle »*

du programme : un unique composant manipule tous les niveaux de sécurité simultanément. Aucune isolation n'est fournie entre ces différents niveaux de sécurité. Il peut donc écrire les données confidentielles sur l'écran affichant normalement les données non classifiées.

Pour éviter ce type de problème, MILS sépare le programme en fonction des niveaux de sécurité qu'il contient. Dans le cas du pilote d'écran, le composant est découpé en deux : l'un manipulant le niveau confidentiel, l'autre manipulant le niveau non classifié.

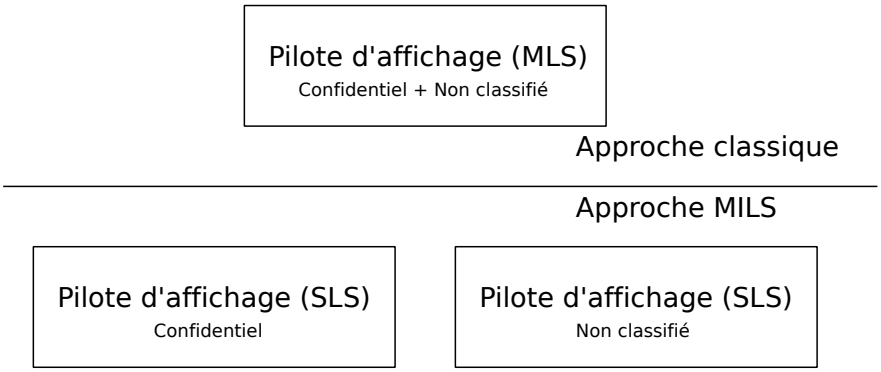


FIGURE 2.1 – Exemple de pilote implanté par une approche traditionnelle et une approche MILS

La représentation proposée par MILS ne comprend que les informations relatives à la sécurité et ne spécifie pas le déploiement ou la plate-forme d'exécution de l'application. MILS se base sur l'hypothèse de garantie d'isolation entre les composants et par conséquent, ne nécessite pas l'introduction d'information supplémentaire.

A titre d'exemple, les cas d'études de déploiement présents dans la littérature considèrent qu'une interface réseau garantit l'isolation des niveaux de sécurité (une telle interface est nommée *Trusted Network Interface Unit - TNIU - dans la littérature*). De la même manière, ces exemples s'appuient sur l'hypothèse de séparation des composants par le noyau et que chacun d'entre eux sont isolés spatialement (un composant ne peut accéder aux données d'un autre, sauf connection explicite) et temporellement (chaque composant dispose d'un quantum de temps pour s'exécuter).

Cependant, ce contexte restreint l'utilisation de ce format de représentation à la seule analyse et vérification des propriétés de sécurité du système. En outre, cet unique formalisme ne peut pas être utilisé comme tel pour diriger tout l'intégralité du développement d'un système critique. D'autres contraintes (temps d'exécution, erreurs potentielles) doivent être spécifiées afin de valider et implanter les autres aspects du système (sûreté, déploiement, etc.).

Une solution à ce problème consisterait à intégrer les principes de MILS dans un langage existant, plus complet et formel, offrant alors ses possibilités d'analyse, de validation et d'implantation.

Extension à d'autres formalismes de modélisation.

Le format de représentation proposé par MILS, bien que pertinent pour

l'analyse et la validation de la sécurité, présente quelques limites d'expressivité : les composants ne spécifient pas formellement quels éléments du système ils représentent (composants non types, absence de spécification des aspects logiciels et/ou matériels, etc.). Pour ces raisons, il apparait pertinent d'intégrer les concepts de MILS à d'autres formalismes de représentation ayant une sémantique plus précise.

Plusieurs initiatives ont été engagées pour décrire les exigences de sécurité de MILS dans d'autres langages de spécification, tel AADL. Par exemple, la société WW Technology a réalisé des efforts significatifs pour créer des outils d'analyse de modèles AADL validant des contraintes de MILS. Cependant, au vu du caractère commercial du produit, aucune information scientifique pertinente n'est disponible quant au contenu de ces produits et des fonctions de validation qu'ils proposent (niveau de vérification, rapport avec le support d'exécution sous-jacent, etc.). Cela limite l'analyse que l'on est amené à porter sur cette initiative, mais montre toutefois que la transposition des principes de MILS vers un langage de spécification général est une proposition pertinente.

Maintenant que l'approche de représentation des mécanismes de sécurité a été réalisée, nous abordons la plate-forme d'exécution de MILS, destinée à garantir l'isolation entre composants.

Plate-forme d'exécution MILS

La plate-forme d'exécution MILS a pour objectif de séparer les niveaux de sécurité entre les composants lors de leur exécution. Elle est composée d'un noyau (*separation kernel*) et (parfois, suivant les articles et la partie de la communauté présentant les travaux) d'un intergiciel.

Le développement de ces deux composants est toujours en cours, l'étape actuelle consistant à rédiger un *protection profile* (section 2.4.3) en vue de la validation des règles de MILS par la communauté des Critères Communs (présentée en section 2.3.2).

Le noyau définit des services d'isolation fortement similaires à ceux proposés par ARINC653. Cette similitude est d'ailleurs assumée par la communauté MILS [97]. Chaque application est exécutée dans une partition qui est soumise à une isolation :

1. **temporelle** : la partition est exécutée pendant une période de temps fixe et prédéterminée à la configuration. La durée de cette tranche de temps est invariable, une violation de cette contrainte constitue un canal caché pour obtenir des informations sur le système (par exemple, le temps d'exécution d'une tâche peut donner des informations sur son comportement).
2. **spatiale** : l'espace mémoire alloué à la partition est unique et non accessible aux autres partitions. Ainsi, chaque partition (et par conséquent, chaque niveau de sécurité) est isolé et ne peut lire ou écrire les données des autres partitions exécutées sur le même processeur. Ce mécanisme est fourni par le matériel exécutant le noyau (bien souvent, il s'agit de l'unité de gestion de mémoire - *Memory Management Unit* ou MMU - du processeur).

Les partitions peuvent communiquer par des liens prédéfinis lors du développement du système et ne violant pas les règles d'isolation de la sécurité. Ces canaux sont supervisés par le noyau qui contrôle les entités qui communiquent et assure l'absence de canal caché.

Ainsi, les partitions n'ont pas *« conscience »* de la présence des autres entités et s'exécutent comme si elles étaient sur un unique processeur.

MILS introduit des contraintes sur ses services de sécurité, ceux-ci devant respecter les règles **NEAT** :

- **N**on-Bypassable : on ne peut détourner les mécanismes de sécurité mis en place dans la plate-forme.
- **E**valuable : ces mécanismes doivent être légers et formellement prouvés.
- **A**lways-Invoked : les mécanismes de sécurité sont **toujours invoqués**.
- **T**amperproof : les données ne peuvent être modifiées qu'en passant par les mécanismes de sécurité, aucune manipulation non autorisée ne peut être effectuée.

Cependant, si le formalisme de modélisation et les articles requièrent que des mécanismes doivent être mis en place pour isoler les communications entre les différents nœuds d'un système réparti, ces derniers ne sont pas décrits d'un point de vue implantation. De plus, s'il est facile de garantir l'isolation de la sécurité sur un nœud local, cela est problématique dans le cas d'un système réparti utilisant des médiums de communication non sécurisés (comme un réseau Ethernet). Les paragraphes suivant détaillent ces deux cas.

Viabilité de l'architecture dans le cas local

Dans le cas d'une architecture ne communiquant pas vers l'extérieur, le respect des contraintes de sécurité est assez trivial : chaque niveau de sécurité est confiné dans un espace mémoire et sa partie applicative dispose d'une ou plusieurs tranches de temps pour exécuter ses tâches.

Les applications peuvent communiquer via des canaux supervisés par le noyau qui garantit l'isolement des données transportées. Enfin, les pilotes de périphériques partagent un seul niveau de sécurité et sont confinés dans une partition, assurant ainsi l'isolation du niveau de sécurité qu'ils manipulent.

Autrement dit, l'assurance de sécurité dans le cas d'un nœud local repose principalement sur la bonne implantation des mécanismes de sécurité au sein du noyau et une bonne configuration des ressources du système.

Viabilité de l'architecture dans le cas réparti

Dans le cas réparti, les différents nœuds de l'application communiquent à travers le réseau. Cela met en jeu plusieurs mécanismes :

- une pile protocolaire ;
- un accès au périphérique de communication (la carte réseau).

Pour chacun de ces mécanismes, des attaques potentielles sont possibles :

- écoute sur le médium de communication (par exemple : l'attaquant écoute les trames transmises sur un câble Ethernet) ;
- compromission des données dans les couches protocolaires (par exemple, séquencement et acquittement de segments TCP) ;

– modification des données par le pilote (données applicatives contenues par chaque couche).

De plus, dans le cas d'un pilote connecté à plusieurs partitions manipulant plusieurs niveaux de sécurité, ce pilote contiendra simultanément des données à plusieurs niveaux de sécurité. Ce cas est malheureusement un cas pratique, plusieurs drivers ne pouvant pas contrôler le même matériel à cause des mécanismes d'implantation (redirection d'interruption, etc.).

Il est donc nécessaire d'introduire un mécanisme d'isolation spatiale dans le traitement des connections réseaux. Ces mécanismes d'isolation, introduits sous le terme de *« middleware MILS »* sont très mal définis dans la littérature de MILS [5]. Ces propriétés de sécurité sont assurées par l'interface réseau dite de *« confiance »* (*« Trusted Network Interface Unit »*) et assure alors que pilote et pile protocolaire sont sûrs et garantissent l'isolation des niveaux de sécurité.

Cependant, considérer une interface réseau comme *« sûre »* implique :

– soit qu'elle fournit une isolation des niveaux de sécurité par une analyse, des tests ou une vérification formelle de l'implantation. Dans ce cas, ce composant sera considéré comme *MSLS* car garantissant une isolation entre les niveaux de sécurité qu'il manipule. Cependant, le développement de tels composants reste coûteux.
– soit qu'elle ne manipule qu'un seul niveau de sécurité, ce qu'il faut vérifier lors de la conception du système. Dans ce cas, ce composant est considéré comme *SLS*.

Cependant, ces contraintes ne sont pas clairement explicitées. Dans les articles récents comme [97], il est dit que les mécanismes de communication doivent apporter des garanties d'isolation [111], sans en donner une description précise. Ces mécanismes doivent toutefois offrir une protection des niveaux de sécurité et des attaques spécifiques au médium de communication (déni de service, écoute du médium, etc.).

A cause du manque de précision, les applications réparties requièrent des mécanismes spécifiques d'isolation pour garantir leur sécurité.

Éléments remarquables de la plate-forme MILS

A partir de cet aperçu de la plate-forme MILS, il est déjà possible de dégager quelques éléments pertinents dans le contexte de sécurité :

– le noyau se doit d'être réduit au strict minimum
– les pilotes ne doivent pas résider dans le noyau mais en espace utilisateur
– l'isolation temporelle et spatiale sont ici utilisées à des fins de sécurité
    1. **l'isolation temporelle** supprime un canal caché d'information (le temps d'exécution d'une application).
    2. **l'isolation spatiale** garantie l'intégrité et la non-modification des données entre les différents niveaux de sécurité.

2.1.2 Modèles de sécurité

Un modèle de sécurité est un ensemble de règles définissant les interactions possibles (envoi/réception de données, etc.) entre les composants d'un

système en fonction de leur niveau de sécurité. L'approche MILS, présentée dans la section précédente, peut être vue comme une politique de sécurité visant à garantir une isolation stricte entre les niveaux de sécurité (absence de communication entre composants ayant des niveaux de sécurité différents).

Plusieurs modèles de sécurité ont été développés au cours des années. Ils énoncent les opérations autorisées en fonction des niveaux de sécurité manipulés. La littérature propose de nombreux modèles de sécurité correspondant à des domaines d'applications différents, les trois plus populaires du milieu militaire (où la recherche est la plus active dans la sécurité) sont présentés dans ce mémoire : Bell-Lapadula [12], Biba [18] et Chinese Wall [23].

Sujets, objets et niveaux de sécurité

Les modèles de sécurité introduisent deux concepts importants :

1. **l'objet** correspond à « ce qui est manipulé » (le plus souvent : une donnée). Un objet a un niveau de sécurité (classifié, secret, etc.).
2. **le sujet** qui manipule un ou plusieurs objets. Il s'agit d'une entité active (tâche, partition, etc.) exécutant des actions. Son niveau de classification (niveau(x) de sécurité qui lui sont associé(s)) détermine les types de données qu'il peut manipuler.

Les modèles de sécurité définissent alors les opérations autorisées entre sujets en fonction :

1. des objets qu'ils manipulent ;
2. des connections avec les autres sujets. Par exemple, un modèle de sécurité définira si une connection entre un sujet *secret* et un sujet *non-classifié* est autorisée ou non.

Le modèle Bell-Lapadula

Bell-Lapadula [96] est probablement le modèle le plus cité dans la littérature de la sécurité. Le but consiste à empêcher les sujets classifiés à un certain niveau de sécurité (par exemple, secret) d'accéder aux données classifiées à un niveau supérieur. Ainsi, ce modèle se concentre sur la protection de la **confidentialité** des données au travers de **trois règles** :

1. *« no read-up »* : un sujet ne peut lire des objets classifiés à un niveau supérieur au sien. Autrement dit, un sujet à un niveau de sécurité N ne peut lire des objets classés à un niveau $N + x$ (avec $x > 0$).
2. *« no write down »* : un sujet ne peut pas écrire des objets à un niveau de sécurité inférieur au sien. Autrement dit, un sujet à un niveau de sécurité N ne peut transmettre (écrire) ses données à un sujet de niveau $N - x$ (avec $x > 0$).
3. un système ayant la possibilité de lire et d'écrire des données doit le faire au même niveau de sécurité. Cela empêche donc un sujet d'abaisser le niveau de classification d'un objet. Cette troisième règle est bien souvent peu explicitée. Toutefois, elle reste essentielle car elle permet un contrôle du flux de données d'un niveau de sécurité. On trouvera dans [58] une explication détaillée de cette règle.

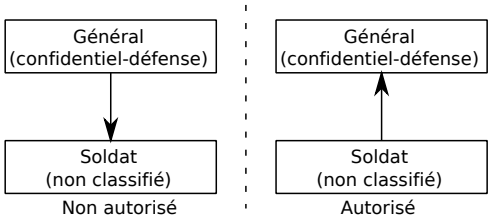


FIGURE 2.2 – Exemple de l'application du modèle de sécurité Bell-Lapadula

Le modèle de sécurité Bell-Lapadula est illustré dans la figure 2.2. Cet exemple illustre la règle *no write-down*, où un sujet évalué à un niveau élevé (un général) ne peut envoyer de données à un sujet de faible niveau de sécurité (soldat) afin de préserver la sécurité des données (les données classifiées à un niveau ne peuvent être communiquée à une entité évaluée à un niveau inférieur). A l'inverse, le soldat a l'autorisation d'envoyer des données au général, associé à un niveau de sécurité supérieur.

Ce modèle de sécurité répond aux contraintes de protection de l'information du domaine militaire où les données classifiées à un niveau de sécurité ne doivent pas être communiquées aux sujets de niveau *« inférieur »* . De par sa nature de précurseur, il a influencé l'écriture de l'Orange Book [93, 39], livre fondateur dans la sécurité ayant été à la base des Critères Communs (présentés en section 2.3.2).

Le modèle Biba

Le modèle de sécurité Biba est l'inverse de Bell-Lapadula. Alors que Bell-Lapadula se concentre sur la protection de la confidentialité des données, Biba a pour but la préservation de **l'intégrité des données**.

Pour cela, il s'assure que les données sécurisées peuvent être transmises et qu'une donnée classifiée à un haut-niveau de sécurité ne peut être altérée par un sujet classifié à un niveau inférieur.

Les règles régissant ce modèle sont l'inverse de celles introduites par Bell-Lapadula :

1. *« no read-down »* : un sujet ne peut lire un objet d'un niveau de sécurité inférieur. Autrement dit, un sujet à un niveau de sécurité N ne peut lire des objets classés à un niveau $N - x$ (avec $x > 0$)
2. *« no write up »* : un sujet ne peut écrire un objet classifié à un niveau de sécurité supérieur. Autrement dit, un sujet à un niveau de sécurité N ne peut transmettre (écrire) ses données à un sujet de niveau $N + x$ (avec $x > 0$).

La figure 2.3 illustre l'utilisation du modèle de sécurité Biba. L'exemple choisi est l'inverse de celui retenu pour la présentation du modèle Bell-Lapadula (figure 2.2) et marque l'opposition des deux modèles de sécurité.

Dans cet exemple, le général (classifié au plus haut niveau de sécurité) peut envoyer des données au soldat (niveau non-classifié). Cependant, l'inverse est impossible : l'envoi d'une donnée du soldat (non-classifiée) vers le général

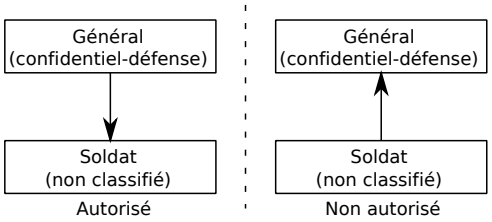


FIGURE 2.3 – Exemple de l'application du modèle de sécurité Biba

(confidentiel-défense) ne respecterait pas la règle du *« no write-up »* . La politique de sécurité est axée ici sur l'intégrité : en empêchant l'envoi des niveaux de sécurité inférieurs vers ceux supérieurs, on assure qu'aucune information potentiellement erronée ne parvient aux entités hautement classifiées.

Le modèle Biba considère que les données *« fiables »* et *« sûres »* sont celles qui ont le plus fort niveau de sécurité. Par conséquent, ce sont les données qui sont issues des sujets ayant le plus fort niveau de classification. A l'inverse, ces données peuvent être exploitées par les sujets de niveaux inférieurs mais ne peuvent pas être renvoyés aux autorités supérieures, préservant ainsi l'intégrité des données et l'absence de modification par un sujet de niveau *« inférieur »* à la donnée.

Le modèle Chinese Wall

Le modèle de sécurité *Chinese Wall* [23] (aussi connu sous le nom de *Brewer and Nash*, les inventeurs du modèle) consiste à éviter les conflits d'intérêts entre les sujets. Cette politique de sécurité est la transposition d'un modèle de sécurité du monde industriel, où, pour préserver le secret de l'information (données financières, futurs projets), les différents acteurs d'un même marché sont isolés, évitant les fuites d'information.

Il est important de constater ici que le but recherché n'est ni la confidentialité des données, ni leur intégrité, mais simplement leur non-divulgation d'un groupe de sujets à un autre. A titre d'exemple, la politique de sécurité *Chinese Wall* est celle qu'appliquent des entreprises concurrentes lorsqu'elles doivent se rencontrer : chacune isole ses ressources afin d'éviter de toute fuite d'information.

Ce modèle de sécurité est illustré par l'exemple d'un système regroupant trois $s$ujets : deux bureaux de conception, (chacun travaillant pour une marque différente) et une usine. Deux domaines de données sont également définis : A et B.

La figure 2.4 décrit un système ne respectant pas les règles du modèle de sécurité. Il est aisé de constater que l'usine a accès à des données des deux marques (deux domaines des objets). Il y a donc conflit d'intérêt car l'usine a accès aux deux domaines et pourrait communiquer des données à l'une ou l'autre des marques, créant ainsi un canal de communication entre deux domaines de données.

Pour éviter ce problème, il est nécessaire d'introduire une nouvelle usine, différenciant ainsi les usines pour chaque marque (ou domaine de données). Les

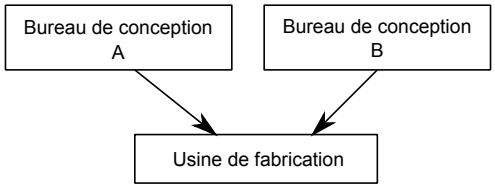


FIGURE 2.4 – Exemple de système ne respectant pas la politique de sécurité *Chinese Wall*, l'usine ayant accès aux deux domaines de données

usines ont accès aux données de leur bureau de conception respectif, empêchant une potentielle fuite d'information dans l'usine. Ce schéma est illustré dans la figure 2.5.

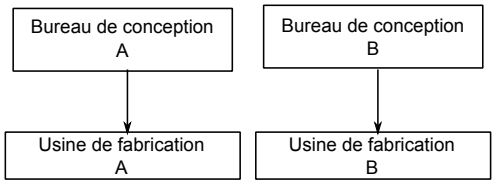


FIGURE 2.5 – Exemple de système respectant la politique de sécurité Chinese Wall, chaque domaine de donnée étant séparé

2.1.3 Algorithmes de chiffrement

Les algorithmes de chiffrement effectuent des opérations pour *c*rypter une donnée et éviter qu'elle puisse être lue par une entité non autorisée. Elles ont donc un but de préservation de la confidentialité, voire, dans certains cas de l'intégrité ou la non-répudiation. Dans notre contexte, ces techniques peuvent être particulièrement intéressantes lorsqu'une donnée classifiée à un haut niveau de sécurité doit transiter par un médium non sécurisé (par exemple : envoi de données *top-secret* sur un réseau Ethernet).

Sans toutefois rentrer dans les détails d'implantation, cette section a pour objectif de rappeler les principes fondateurs des techniques de chiffrement et de présenter les principales approches existantes. Nous mettons toutefois l'accent sur l'utilité de ces approches en indiquant leur adéquation avec les contraintes des systèmes critiques (déterminisme temporel, prédictabilité, consommation de ressources, etc.).

Algorithmes symétriques

Le chiffrement symétrique se base sur l'utilisation d'une clé unique pour chiffrer ou déchiffrer les messages. Comme illustré dans la figure 2.6, l'émetteur et le récepteur utilise la même clé pour chiffrer et déchiffrer. Elle doit donc être connue des deux parties. D'un point de vue développement, la clé doit donc être déployée statiquement ou échangée par des canaux de diffusion considérés comme sécurisés et fiables.

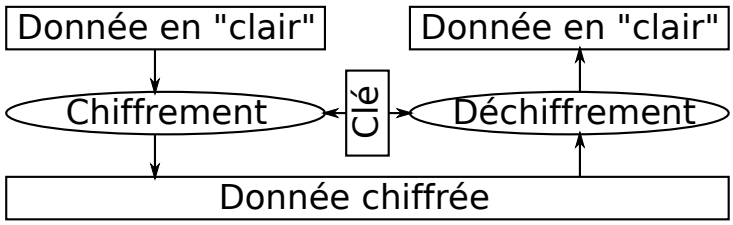


FIGURE 2.6 – Fonctionnement du chiffrement symétrique

Deux algorithmes de chiffrement symétriques très utilisés sont présentés : Data Encryption Standard et Blowfish.

Data Encryption Standard (DES) Data Encryption Standard [76] a été créé en 1976 à la suite d'une demande faite par le *National Bureau of Standards* (NBS) (connu depuis 1988 sous le nom du *National Institute of Standards and Technology* NIST). Le but était la conception d'un algorithme utilisable par les institutions ne disposant à l'époque que de ressources de calcul limitées.

DES chiffre un message par bloc de 64 bits à l'aide d'une clé de 56 bits. Les opérations mathématiques utilisées par l'algorithme sont principalement de la permutation, des opérations arithmétiques non-complexes (telles que `XOR`) et des substitutions de séquences binaires.

Ces propriétés (blocs de taille réduite, opérations arithmétiques simples) font de DES un algorithme adapté aux systèmes embarqués de par leur aspect déterministe : le temps de chiffrement/déchiffrement peut être facilement déduit de la taille de la donnée (nombre de blocs et d'itérations requis pour chiffrer une donnée). Par conséquent, l'ingénieur peut calculer le « coût » d'utilisation du protocole (quel est le nombre de calculs nécessaires pour chiffrer une donnée).

Il est à noter que cet algorithme est vulnérable à diverses attaques ou techniques de cryptanalyse [52]. Il remet donc en cause la confidentialité des données chiffrées. Des alternatives (comme le 3-DES [28]) ont été proposées afin de pallier ce problème.

Toutefois, DES reste utilisé de nos jours et constitue un bon exemple d'algorithme de chiffrement simple et déterministe.

Blowfish

Blowfish [102] est un algorithme de chiffrement conçu en 1993 par BRUCE SCHNEIER. L'objectif premier était de proposer une alternative viable à DES, alors critiqué pour ses vulnérabilités. Il présente l'avantage d'être libre, publiquement disponible et surtout, de ne jamais avoir été cassé par des techniques de cryptanalyse.

Tout comme DES, cet algorithme chiffre les données par blocs de 64 bits. Les clés de chiffrement/déchiffrement ont une taille variable, de 32 à 448 bits. Il repose sur des substitutions et l'usage de fonctions mathématiques simples (tels que `XOR` ou `modulo`). Cet algorithme est, tout comme DES, déterministe dans le temps de calcul nécessaire au chiffrement des données. Enfin, il est peu

consommateur de ressources, propriété particulièrement intéressante dans le contexte des systèmes embarqués.

Tout comme DES, Blowfish est un protocole très utilisé. A titre indicatif, son auteur donne une liste [101] des produits (commerciaux ou non) l'utilisant.

Algorithmes asymétriques

A l'inverse du chiffrement symétrique, les techniques de chiffrement asymétrique dissocient la clé de chiffrement de la clé de déchiffrement. La clé utilisée pour chiffrer les données est appelée *« clé publique »* . Elle est disponible à toute entité souhaitant envoyer un message à celui détenant la *« clé privée »* , dont l'utilité est de décrypter le message.

La figure 2.7 illustre l'utilisation d'un protocole de chiffrement asymétrique : les données sont chiffrées à l'aide de la clé publique, puis déchiffrées par la clé privée.

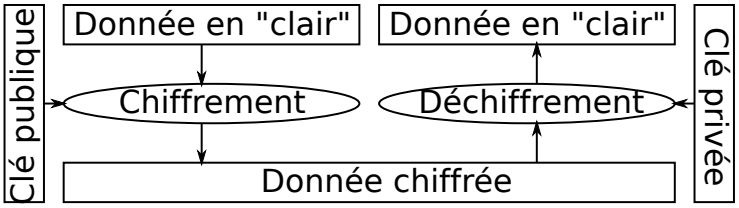


FIGURE 2.7 – Fonctionnement du chiffrement asymétrique

Le chiffrement asymétrique repose intégralement sur l'existence d'une fonction à sens unique dont l'inverse est impossible à trouver (ou dans un temps difficilement borné). Cette fonction servant à chiffrer les données est alors appelée *« clé publique »* . Elle est diffusée à toute entité souhaitant échanger des messages. Les correspondants l'utilisent pour chiffrer leurs messages.

L'entité émettrice de la clé publique est alors la seule à pouvoir déchiffrer le message chiffré, grâce à une fonction inverse qu'il aura pris soin de générer. Celle-ci n'est pas calculable (calculs demandant trop de temps ou de ressources) pour toute machine actuelle. Toute la complexité du chiffrement à clé publique repose sur le principe des *« brèches secrètes »* , où l'inverse d'une fonction est difficile à deviner et calculer par l'émetteur de la fonction. La fonction inverse, appelée *« clé privée »* sert alors à décrypter le message.

Dans un premier temps, l'algorithme de chiffrement asymétrique le plus connu (`RSA`) est présenté. Puis, il est expliqué en quoi les techniques de chiffrement asymétrique ne sont pas adaptées aux systèmes embarqués critiques.

Rivest Shamir Adleman (RSA) RSA a été décrit en 1977 par RON RIVEST, ADI SHAMIR et LEN ADLEMAN. C'est l'algorithme à clé publique le plus utilisé, implanté dans de nombreux objets numériques de notre vie courante (télévision, logiciels, consoles de jeux, etc.).

L'algorithme se base sur la génération de nombres premiers distincts, potentiellement importants et sur la capacité à générer un nombre aléatoire. Cependant, les méthodes de recherche de nombres premiers sont très coû-

teuses, aussi bien en calcul qu'en mémoire. De plus, cet algorithme n'est efficace qu'avec des nombres premiers de taille importante.

Le recours à des nombres premiers et la lourdeur de calcul inhérentes aux nombres de capacité importante fait de RSA un algorithme particulièrement coûteux en termes de ressources de calcul et de mémoire. Pour ces raisons, il est inadapté aux systèmes embarqués qui ne disposent que de ressources de calcul et de mémoire limitées.

Contraintes des systèmes critiques

Les systèmes embarqués critiques, de par leur faible ressources, ont d'importantes contraintes qui peuvent limiter l'utilisation des algorithmes de chiffrement :

- prédiction du temps d'exécution (afin qu'une tâche exécutant l'algorithme puisse respecter ses échéances)
- empreinte mémoire faible
- capacité de calcul restreinte

Au vu de ces contraintes, certains algorithmes ne peuvent pas être utilisables. En particulier, les méthodes de chiffrement symétrique sont légères et présentent des propriétés intéressantes dans le contexte des systèmes embarqués critiques : déterminisme du temps d'exécution, consommation mémoire fixe, etc.

A l'inverse, les méthodes à clé privées utilisent des nombres premiers de grande taille impliquant des calculs lourds et une importante consommation de mémoire. Ces propriétés en font de mauvais candidats pour les systèmes embarqués critiques.

2.1.4 Synthèse des problèmes soulevés

Cette section a présenté plusieurs approches concernant la sécurité. Les travaux présentés agissent à différents niveaux :

- l'approche MILS analyse l'architecture à partir de spécifications des contraintes de sécurité et s'appuie sur une plate-forme spécifique pour isoler les niveaux de sécurité à l'exécution du système. Cependant, certains mécanismes d'assurance de sécurité ne sont pas explicités, telle l'isolation des données dans le cas de systèmes répartis.
- les modèles de sécurité définissent les opérations autorisées entre composants (lecture/écriture de données) en fonction de leur niveau de sécurité. Cependant, ces règles sont textuelles et il est nécessaire de définir une méthode pour qu'elles soient vérifiées à partir d'une description standardisée du système.
- les protocoles de chiffrement permettent de protéger une information avant que celle-ci ne transite par des entités non classifiées. Ces techniques de protection doivent être liées avec des approches comme MILS, qui définissent un besoin de garantie de sécurité sans en expliciter les mécanismes (par exemple : chiffrer les données classifiées avant de les envoyer sur un médium non sécurisé).

Cependant, de par leurs caractéristiques, certains protocoles peuvent ne pas convenir dans le contexte des systèmes critiques au vu de leur manque de déterminisme ou de consommation de ressources trop importante.

Afin d'apporter une plus grande cohérence dans l'apport de la sécurité du système, ces approches d'être connectées entre elles afin d'assurer la sécurité au sein des spécifications (vérification d'une politique de sécurité, garantie de l'isolation, etc.), et à l'exécution du système (chiffrement de données, plate-forme assurant l'isolation des niveaux de sécurité).

2.2 Approches de sûreté

Cette section présente deux principales approches pour l'assurance de sûreté : une pour l'identification analyse des risques et le standard ARINC653. Dans le cas de l'identification et de l'analyse des risques, les méthodes spécifient quelles fautes peuvent être identifiées et analysées à partir de la spécification d'un système. Parallèlement, le standard ARINC653 définit une plate-forme isolant des composants logiciels dans un but de sûreté, garantissant leur fonctionnement comme s'ils étaient exécutés sur un processeur indépendant.

2.2.1 Identification et analyse des risques (approche FPTC)

L'équipe de recherche *High Integrity Systems Engineering Group* de l'université de York (Angleterre) a effectué de nombreux travaux autour de l'identification des fautes et de leur propagation. Pour cela, ils utilisent une représentation du système dédiée à la spécification des fautes et leur propagation [113, 72, 87].

Cette présentation introduit le formalisme FTPC (Failure Propagation and Transformation Calculus [113]) qui s'appuie sur le format *Real-Time Networks* [88] pour décrire les différents aspects du système. Les composants représentent des éléments matériels (capteurs, périphériques) et les connections modélisent les communications entre ces éléments.

Plusieurs types de connections sont possibles : canal d'événement, regroupement de communication, etc. Une connection peut être interprétée comme un composant indépendant, définissant alors ses propres erreurs[113].

Le concepteur décrit ensuite les erreurs pouvant survenir dans le système. Pour cela, il annote les composants en décrivant chaque erreur qu'il peut contenir. Enfin, il spécifie les interactions potentielles et propagations de ces fautes. Une liste d'erreurs prédéfinie est par ailleurs disponible [72], comme :

- **value** : erreur de valeur
- **late/early** : erreur de timing, évènement ou valeur reçus ou trop tard ou trop tôt.
- **omission/comission** : le service n'est pas rendu ou est rendu alors qu'il n'est pas demandé.

La figure 2.8 illustre cette approche. Cet exemple définit trois tâches (`T1`, `T2` et `T3`) communiquant entre elles. Le canal entre `T1` et `T2` utilise une file de messages tandis que la communication entre `T2` et `T3` utilise de la mémoire

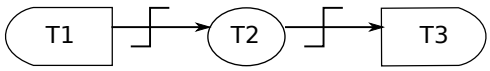


FIGURE 2.8 – Exemple de système représenté avec le formalisme de l'approche FPTC

partagée. La tâche `T2` attend la valeur en provenance de `T1` et la communique à `T3` via la mémoire partagée.

La spécification des erreurs et propagation de la tâche T2 sont indiquées ci-dessous :

```
late -> value
v    -> v
```

Cette spécification indique deux cas d'erreur :

- si la valeur sur le port d'entrée arrive trop tard (*late*), la valeur communiquée sur le port de sortie sera mauvaise (*value*).
- pour tout autre type d'erreur (e), cette erreur est propagée sur le port de sortir (e).

Un outil (CADiZ) [87] a été conçu pour l'analyse de la propagation des fautes entre les composants. Ce dernier offre la possibilité de détecter la propagation des fautes entre plusieurs composants chaînés : une erreur au sein d'un composant peut impacter la hiérarchie des composants connectés. Dans l'exemple présenté (figure 2.8), une faute dans la tâche `T1` peut avoir un impact sur `T3`, bien que cette dernière ne soit pas directement liée à `T1`).

Cette approche offre une analyse à travers de la sûreté et de la propagation des fautes à travers le maillage de composants. Cette analyse se focalise sur les spécifications et n'apporte aucune assurance quant à la sûreté de l'implémentation. Pour cette raison, elle doit être couplée aux outils d'analyse de code qui vérifient la bonne implantation des mécanismes de détection et de recouvrement d'erreurs.

Une telle analyse de l'architecture améliore la détection des erreurs et évite l'approche traditionnelle *fail-first-patch-later*, réduisant par la même occasion les coûts de production ainsi que le délai de mise en production.

2.2.2 ARINC653

ARINC653 [4] est un standard dédié à la sûreté de fonctionnement (première publication en 1996). Ce standard définit un système d'exploitation (O/S) pour l'exécution d'applications dans le domaine avionique. Il constitue un élément important [7] de l'architecture modulaire intégrée (IMA [73]) : la standardisation des composants et de leurs interfaces de communication facilite leur intégration.

Un système d'exploitation ARINC653 isole les composants logiciels dans des partitions en fonction de leur niveau de criticité. Il fournit à chaque partition un environnement et les isole de telle manière à ce qu'elles s'exécutent comme sur un processeur indépendant.

Le concept de partitionnement d'ARINC653 est fortement connexe à celui de MILS. Cependant, contrairement à MILS, cette politique de partitionnement est introduite dans un but de sûreté de fonctionnement [11]. Pour cela, l'O/S ARINC653 (i) isole les fautes, (ii) les répare en exécutant une action de recouvrement associée (redémarrage de l'entité fautive, arrêt de la partition, etc.) et (iii) limite leur propagation.

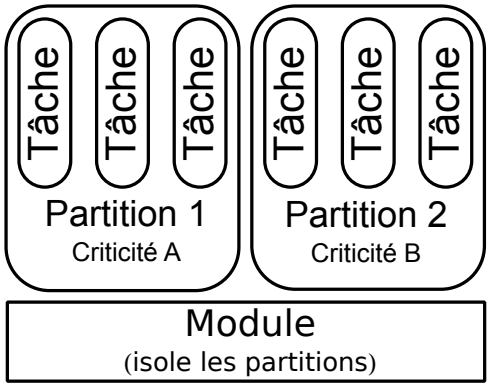


FIGURE 2.9 – Aperçu de l'architecture d'un système ARINC653

Trois niveaux d'exécution sont distingués, comme illustré dans la figure 2.9 :

1. le *module* assure l'isolation des partitions. Il s'agit du noyau du système qui contrôle l'exécution des partitions (allocations des ressources, ordonnancement, etc.).
2. les partitions fournissent un environnement (ressources, fonctions) dédié pour l'exécution de leurs tâches et les ordonnancent suivant une politique spécifique (un paragraphe de cette section est dédiée à la politique spécifique d'ordonnancement des systèmes ARINC653). Une partition est indépendante et ne dépend pas des autres entités du système.
3. une tâche est un fil d'exécution contenu dans une partition, contraint par des exigences temporelles (échéance, temps d'exécution, etc.). L'exécution de cette tâche (démarrage, préemption, etc.) est soumise à l'ordonnanceur de sa partition.

Un système d'exploitation ARINC653 soumet les partitions à une isolation :

- temporelle : chaque partition dispose d'une tranche de temps prédéterminée, fixe pour exécuter ses tâches. Une partition essayant de consommer davantage de temps qu'alloué sera donc suspendue.
- spatiale : chaque partition dispose d'une zone mémoire pour stocker son code et ses données. Une partition ne peut donc ni lire, ni écrire les données des autres partitions.

En plus de cette isolation, ARINC653 identifie et répare les fautes pouvant survenir au cours de l'exécution du système par l'exécution d'une action de recouvrement associée. Dans un O/S ARINC653, les fautes peuvent survenir à trois niveaux : noyau, partition ou tâche. Le concepteur spécifie la politique

de sûreté en décrivant la procédure de recouvrement associée à chaque faute des couches du système.

Ces mécanismes de partitionnement et d'isolation des fautes sont fournis par un noyau spécifique (appelé *module* dans la terminologie ARINC653). Il propose un jeu de services réduits : isolation des partitions, confinement des communications inter-partitions et détection/recouvrement d'erreurs. Il a pour but d'être petit et potentiellement certifiable et/ou vérifiable formellement.

Ordonnancement hiérarchique

Un système d'exploitation ARINC653 utilise une politique d'ordonnancement hiérarchique [74] (illustrée dans la figure 2.10) à deux niveaux :

1. ordonnancement des partitions par le noyau. Le noyau exécute cycliquement chaque partition pendant un temps fixe, prédéfini à la configuration. C'est ce premier niveau d'ordonnancement qui garantit l'isolation temporelle entre partitions. Il est représenté par le niveau 0 dans la figure 2.10.
2. ordonnancement des tâches dans les partitions. Ce second niveau d'ordonnancement est spécifique à chaque partition. L'algorithme peut donc être différent pour chacune d'elle : une partition peut ordonnancer ses tâches en utilisant l'algorithme *Rate Monotonic Scheduling* [66] alors qu'une autre utilisera un algorithme *Round Robin* comme le montre l'exemple de la figure 2.10.

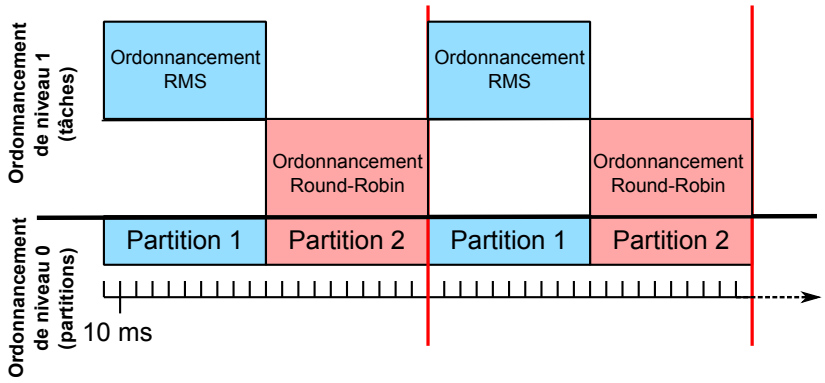


FIGURE 2.10 – Schéma d'ordonnancement des partitions dans ARINC653

Services de communication

Le standard ARINC653 définit un service de communication pour l'échange de données à l'intérieur (*intra-partition*) et à l'extérieur (*extra-partition*) des partitions.

Les communications intra-partition ne sont soumises à aucun contrôle du module ARINC653 et sont sous l'autorité de l'environnement d'exécution de la partition. Celui-ci est donc chargé d'assurer le bon établissement de la communication et du contrôle de ses ressources associées (tampons d'emission/réception, verrous, etc.).

A l'inverse, les communications inter-partitions sont prédéfinies lors de la configuration du *module*. Celui-ci supervise leur établissement et vérifie que seules les partitions autorisées peuvent établir des canaux de communication.

Ce contrôle évite ainsi tout dysfonctionnement dû à une erreur de communication inter-partitions, celle-ci pouvant avoir un impact sur des partitions évaluées à différents niveaux de criticité. Par exemple, une partition peu critique envoyant une quantité importante de messages à une partition critique peut la saturer et la rendre indisponible. Si ce service (communications inter-partitions contrôlées par le noyau) a un objectif de sûreté dans le cadre d'ARINC653, il constitue un élément clé de la sécurité pour MILS. Ainsi, ce même service serait donc pertinent au sein d'une plate-forme commune assurant sûreté et sécurité.

Exemple d'architecture ARINC653

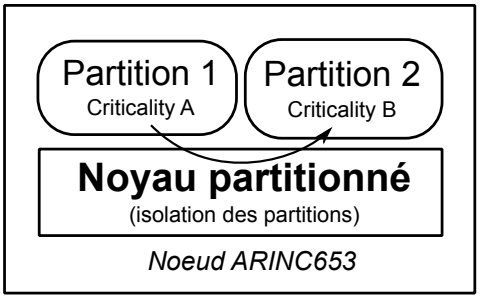


FIGURE 2.11 – Architecture globale d'un système ARINC653

Le concept de partitionnement d'ARINC653 est présenté au travers des figures 2.11 et 2.10.

La figure 2.11 montre l'architecture globale d'un système ARINC653 : un noyau (*module*) exécute des partitions tout en les isolant temporellement et spatialement. Dans le cas de l'architecture présentée dans la figure 2.11, les deux partitions utilisent le service de communication inter-partitions pour communiquer.

Dans cet exemple, chaque partition est exécutée pendant 100ms. La première utilise l'algorithme d'ordonnancement *Rate Monotonic Scheduling* tandis que la seconde utilise l'algorithme *Round Robin*. La figure 2.10 illustre l'exécution des partitions et montre que chaque partition dispose de son propre algorithme d'ordonnancement. Dans le cas d'un dysfonctionnement de la seconde partition, elle n'impactera pas la première qui continuera à être exécutée par le *module*.

Problèmes non adressés dans le standard

Si le standard définit un ensemble de services complet pour la construction d'un système partitionné, le standard ne précise pas les mécanismes d'implantation des pilotes de périphériques. Ces derniers peuvent être localisés dans le noyau (*module*) ou confinés dans une partition.

Cependant, le choix d'implantation est déterminant sur la sûreté de fonctionnement et une faute au sein d'un pilote peut avoir des répercutions importantes sur les propriétés critiques du système (comme l'isolation temporelle ou spatiale) : implanté dans le noyau, le pilote peut alors contenir une erreur et modifier le contenu ou consommer le temps d'exécution de partitions de haute criticité. Par exemple, il peut exécuter une boucle infinie qui impliquera la non-exécution d'une partition. En revanche, si le pilote est implanté comme une partition applicative, le partage du périphérique est alors plus difficile et il est nécessaire de créer des interfaces de communication avec le noyau pour l'exécution de quelques fonctions privilégiées.

Le tableau 2.1 synthétise les différents avantages et inconvénients de chaque choix d'implantation. L'implantation des pilotes au sein des partitions permet de contraindre le pilote à l'isolation spatiale et temporelle. De plus, une erreur dans l'implantation du pilote n'aura pas d'impact direct sur les autres partitions. Toutefois, cette méthode d'implantation demande à ce que des services dédiés soient introduits afin que la partition communique avec le matériel (opérations d'*input/output* exécutées uniquement dans le mode privilégié du processeur). A l'inverse, l'implantation dans le noyau est plus simple, puisque le développeur peut directement communiquer avec les couches matérielles. Toutefois, le pilote aura accès à toute la mémoire du système et pourra perturber son fonctionnement si bien qu'une erreur impactera les services les plus critiques du système.

Même la méthode d'implantation des pilotes de périphériques est laissée à la discrétion du concepteur de la plate-forme, il semble approprié que ces derniers soient confinés dans des partitions [69], afin de garantir au maximum les services de sûreté fournis par le *module*.

	Avantages	**Inconvénients**
Pilote dans une partition	• Isolation des fautes du pilote • Accès plus rapide si utilisé exclusivement par une partition • Réduction des dégâts possible causés par une partition	• Interface spécifique avec le noyau • Implantation plus difficile
Pilote dans le noyau	• Facilité d'implantation • Facilité de partage du périphérique entre partitions	• Impact sur toutes les partitions d'une défaillance du pilote.

TABLE 2.1 – Avantages et inconvénients de l'implantation d'un pilote de périphérique dans une partition ou dans le noyau

Déploiement de systèmes ARINC653

Le standard ARINC653 propose une méthode dédiée pour la configuration et le déploiement de systèmes partitionnés. Elle consiste à allouer les ressources nécessaires pour chaque entité du système et à apporter une cohérence dans le nommage du système réparti (identification des nœuds, nommage des ports de communication, etc.).

Il est donc nécessaire que chaque nœud ait connaissance des besoins de ses partitions (consommation mémoire, ordonnancement, communication intra et inter-partitions) et des données d'identification des autres nœuds/partitions (liens potentiels entre partitions de nœuds distincts, etc.).

Dans ce but, le standard ARINC653 introduit un fichier de description de l'architecture pour la description des ressources et des communications du système. Ce fichier, écrit en format XML, spécifie :

- le nombre de nœuds de l'application.
- les partitions de chaque module avec leurs exigences (taille mémoire requise, etc.).
- les connections entre partitions.

En utilisant ce fichier XML, l'intégrateur génère le code de configuration de chaque nœud de l'application répartie. Cependant, cette étape de génération de code se limite à la création de code de configuration et ne permet pas de générer le code correspondant aux aspects applicatifs du système.

Le cycle de développement complet qu'implique cette méthode de configuration et de déploiement est illustré dans la figure 2.12. Au cours de ce cycle, l'utilisateur décrit l'architecture de son système (fichier XML) et fournit le code applicatif des partitions (première étape de la figure 2.12). Le code de configuration du module est alors automatiquement créé à partir de la description XML fournie (étape 2). Ce code de configuration est ensuite intégré avec les partitions et les services du noyau pour créer l'implantation du système (étape 3). Cette opération est répétée pour chaque nœud de l'application répartie.

Vérification d'architectures ARINC653

Le standard ARINC653 ne propose pas d'outil d'analyse et de validation de l'architecture. Cela limite les possibilités de validation avant implantation alors que, comme discuté en introduction (section 1.3), c'est précisément lors de la spécification que les erreurs sont introduites.

Cependant, il est toutefois possible de s'appuyer sur le fichier de déploiement XML pour réaliser quelques validations. Même si ce fichier ne constitue pas une description complète de l'architecture, il permet d'effectuer plusieurs validations : des outils comme CARTS [40] utilisent un formalisme proche de ce format pour vérifier l'ordonnancement des tâches au sein des partitions.

2.2.3 Synthèse des problèmes soulevés

Nous avons présenté deux approches centrées sur la sûreté de fonctionnement : FPTC et ARINC653. Ces deux approches offrent des garanties quant

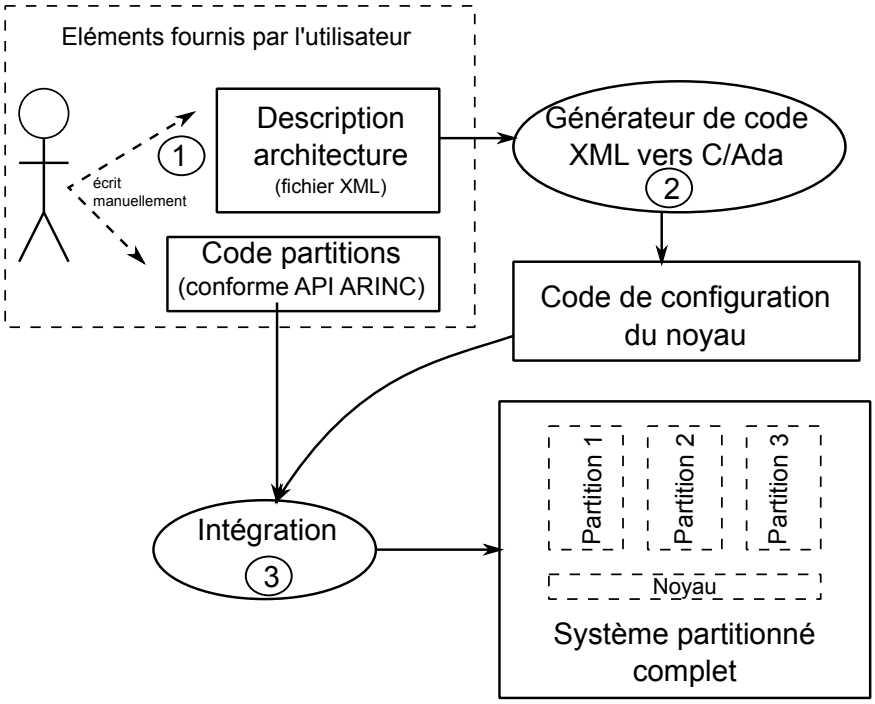


FIGURE 2.12 – Processus de développement d'une application ARINC653

à la sûreté de fonctionnement du système et agissent à différents niveaux du cycle de conception d'un système.

L'approche de détection et d'analyse des fautes son propre formalisme et nécessite donc de traduire les spécifications du système avant de procéder à son analyse. Il ne propose qu'une méthode de détection d'erreurs suite à des échanges de données mais ne définit aucun mécanisme quant à l'exécution du système ni les politiques associées au recouvrement des erreurs.

A l'inverse, ARINC653 définit une plate-forme d'exécution garantissant la sûreté de fonctionnement entre les composants contenus dans le système. Cependant, ce dernier ne s'appuie sur aucune analyse de l'architecture et ne permet pas de détecter les erreurs au cours de la spécification du système (le formalisme XML est davantage destiné à des fins de génération de configuration qu'à un but d'analyse des aspects sûreté).

Afin de pouvoir représenter et valider les aspects sûreté d'un système (premier objectif de nos travaux, section 1.4), il est donc nécessaire d'unifier ces approches par :

- une méthode de représentation commune des systèmes sûrs (avec leurs services d'isolation tels ceux définis dans ARINC653).
- une utilisation des spécifications pour la validation de la sûreté et la détection de fautes potentielles.

2.3 Processus de conception, contraintes et standards

Cette section présente plusieurs standards et méthodes de conception utilisées dans le contexte des systèmes critiques : DO178B, les Critères Communs et les techniques d'analyse de code. Nous donnons un aperçu des niveaux d'assurance à fournir pour la construction et la certification de systèmes critiques.

Nos travaux étant orientés vers les systèmes avioniques, nous présenterons le standard DO178B dans ce domaine. La description des Critères Communs donne un aperçu des contraintes liées à la sécurité tandis que la présentation des techniques d'analyse de code explicitent les technologies actuelles permettant de vérifier plusieurs caractéristiques de l'implantation.

2.3.1 DO178B, un standard de certification

La notion de niveau de sûreté

Au sein des standards de certification, la notion de niveau de sûreté (*Safety Integrity Level* - SIL) est centrale. Cette notion spécifie le niveau de criticité d'un composant ou d'une fonction. De la valeur de cette propriété (critique, très critique, etc.) dépend les contraintes qui doivent être satisfaites. Les méthodes et contraintes imposées au développement du système dépendront donc de son SIL.

Dans la terminologie du standard DO178B, cette notion est appelée DAL *Development Assurance Level* ou *Design Assurance Level*. Les composants logiciels ou matériels sont qualifiés par rapport à un niveau de DAL, et garantissent plusieurs contraintes (analyse des risques, validation, vérification formelle, etc.) en fonction du niveau qu'il utilisent.

Plusieurs approches/méthodes doivent être utilisées conjointement pour qualifier un produit à un certain niveau d'assurance. La liste suivante (donnée à titre indicatif et incomplète) donne un aperçu des contraintes pouvant être requises :

- vérification formelle
- validation de modèles
- analyse des risques et de leur propagation
- relecture manuelle du code (audit)
- analyse de couverture du code
- analyse statique du code

Ces techniques ont pour objectif d'accroître la confiance dans le logiciel en détectant les potentiels vecteurs d'erreurs lors du développement de système. Une métrique dédiée permet de caractériser la fréquence de survenue des fautes : le MTBF *Mean Time Between Failures*. Plus ce dernier est important, plus faibles sont les chances qu'une faute perturbe le fonctionnement du système. Il faut donc maximiser le MTBF afin de s'assurer que la probabilité de survenue d'une erreur soit la plus faible possible. A titre d'exemple, les méthodes actuelles de mesure parviennent à évaluer des taux entre 10^{-3} et 10^{-4} erreur par heure de fonctionnement [71].

Différentes méthodes [107, 106, 70, 8] évaluent le MTBF. Cependant, cette

évaluation est très difficile au vu de (i) la quantité de tests nécessaires et (ii) le temps nécessaire à leur réalisation. Aujourd'hui, la méthode d'évaluation la plus utilisée reste une procédure de test intensive. Or, de tels tests sont très coûteux en terme d'argent, de temps et allongent considérablement le temps requis pour développer une application.

A titre d'exemple, on estime le taux d'erreur du *« plus sûr »* système avionique à 4.10^{-8} erreur par heure. Ce système a été opérationnel pendant plus de 25 millions d'heures sans constater de faute impactant sa sûreté [71]. Si cet exemple peut sembler rassurant, il illustre toutefois les limites des méthodes actuelles : l'étude d'un système sur une telle période est trop longue et le produit est déjà considéré comme *« en fin de vie »* lorsqu'elle se termine. Ceci illustre le besoin de développer des méthodes appropriées pour l'analyse et l'évaluation des risques.

Détail de la norme DO178B

DO178B [94], aussi connu sous le nom de *« Software considerations in airborne systems and equipment certification »* est un standard publié en 1992 par RTCA, une organisation non-lucrative publiant des spécifications dans le domaine de l'avionique. La norme DO178B décrit les contraintes applicables au développement de systèmes critiques dans le domaine de l'avionique.

Elle définit 5 niveaux de DAL [94, 71, 92] (allant de E à A, E étant le moins critique). Chaque niveau de DAL définit les potentielles conséquences d'erreurs et les vérifications, certifications et efforts de validations annexés.

La norme requiert la production d'une dizaine de documents au cours du développement dans lesquels le concepteur décrit les moyens mis en œuvre pour respecter les exigences du système en accord avec le niveau de DAL attendu.

Un point important dans la norme DO178B est que les méthodes utilisées pour respecter les contraintes du standard ne sont pas imposées. A l'inverse, la norme précise quelles méthodes sont à éviter et ne sont pas acceptées par le standard [65], le concepteur restant alors libre des outils à utiliser.

DO178C en gestation

La prochaine version de la norme DO178B est en cours d'élaboration. Cette prochaine version, appelée logiquement DO178C, définie quelles sont les nouvelles méthodes autorisées pour certifier un système.

Parmi les nouveautés, cette révision de la norme autorise désormais la certification du système via la validation de modèles [65]. Cette nouvelle caractéristique facilite ainsi la certification du système en utilisant une spécification de haut-niveau *« générique »* (modèles SysML, UML ou AADL) et simplifierait le processus de certification.

2.3.2 Les Critères Communs (Orange Book)

Le standard Critères Communs [1] (*Common Criteria*) est un standard international (standard *ISO/IEC 15408*) réalisé pour l'évaluation et la certification de la sécurité des systèmes informatiques. C'est l'évolution de l'Orange

DAL	Conséquences d'une faute	Efforts de certification
E	Aucune conséquence	Aucun effort de certification
D	Mineure	• Assurance qualité (suivi des modifications du logiciel) • Documentation • Traçabilité des spécifications de haut niveau du logiciel, des spécifications du système et des vérifications • Vérification formelle des spécifications (optionnelle)
C	Majeure	• Conformités aux exigence du DAL de niveau D • Traçabilité des spécifications jusqu'au code source • Couverture structurelle du code (toutes les instructions sont exécutées - voir section 2.3.3). • Conformité aux standards industriels (e.g. ARINC653)
B	Sérieuse (morts potentiels)	• Conformités aux exigence du DAL de niveau C • Couverture de code au niveau décision (section 2.3.3). • Le développement et la vérification sont effectuées par des entités indépendantes
A	Catastrophique (crash de l'avion)	• Conformités aux exigence du DAL de niveau B • Couverture de code MC/DC (voir section 2.3.3).

Table 2.2 – Définition des DAL de la norme DO178B

Book [39] (aussi appelé TCSEC), lui même dérivé des travaux de Bell et Lapadula [12].

Objectifs

Le but consiste à évaluer la robustesse des mécanismes de sécurité d'un système, nommé le *« Target Of Evaluation »* (*TOE*). L'évaluation s'effectue suivant le type de TOE et le niveau de sécurité recherché. Pour chaque TOE, les exigences de sécurité (*Security Functional Requirements*) doivent être spécifiées (par exemple : le système doit contrôler les communications en fonction d'un niveau de sécurité donné).

En plus de ces exigences, il est impératif de spécifier les propriétés et mécanismes de sécurité du *TOE* (appelé *Security Target*). Ces propriétés indiquent la manière dont le système apporte la sécurité et les mécanismes pour assurer leur garantie.

Pour chaque type de *TOE*, les critères communs définissent un profil de

protection (*Protection Profile*). Il définit les risques potentiels et fonctions de sécurité devant être mis en œuvre (par exemple, la protection des données d'une carte bancaire en cas de vol). Un *TOE* peut être soumis à plusieurs profils de protection, suivant les différents domaines qu'il peut recouper.

Classification des produits

La sécurité d'un produit est évaluée par deux métriques :

1. **l'assurance des exigences de sécurité** (*Security Assurance Requirement* - SAR) : définit les méthodes employées pour garantir la sécurité du système (processus de développement adapté, versionnement du code source, tests, vérification formelle, etc.).
2. **l'évaluation du niveau d'assurance** (**Evaluation Assurance Level** - EAL) : indique le niveau de classification auquel le produit est évalué en fonction de la (SAR).

Les Critères Communs définissent sept niveaux d'assurance (du plus faible au plus exigeant), détaillés dans le tableau 2.3 et montre que plus le niveau d'évaluation demandé est important, plus les contraintes sont fortes.

Niveau	Exigences
EAL1	test structurel du système : l'ingénieur le considère comme une *« boîte noire »* (il n'a pas connaissance des mécanismes internes) et le soumet à un ensemble de valeurs d'entrées pour tester son bon fonctionnement.
EAL2	test fonctionnel du système : l'ingénieur le considère comme une *« boîte blanche »* (il a connaissance du fonctionnement interne) et réalise plusieurs tests pour vérifier son bon fonctionnement (test de défaillance, etc.).
EAL3	testé et vérifié méthodiquement : utilisation d'une suite de tests pour valider les mécanismes de sécurité, réalisation d'une documentation de tests.
EAL4	conçu, testé et vérifié méthodiquement, même exigences que le niveau EAL3, le développement est également documenté.
EAL5	validé et testé. Pour cela, le concepteur et l'ingénieur certifiant le système utilisent une représentation du système pour l'analyser à l'aide d'outils dédiés.
EAL6	conception vérifiée de façon semi-formelle et système testé. Même niveau d'exigence que le niveau précédent. Ici, le niveau EAL5 requiert que le test et/ou la certification soient également réalisés à partir d'une représentation/modélisation du système.
EAL7	conception vérifiée de façon formelle et système testé. Même niveau que précédemment sauf que la représentation utilisée est formelle.

Table 2.3 – Définition des contraintes pour les niveaux d'évaluation des Critères Communs

Niveau *EALn+*

A noter toutefois que certaines évaluations peuvent spécifier que le système remplit davantage de contraintes que son niveau d'évaluation, mais pas assez pour le niveau supérieur. Par exemple, un produit qui satisfait davantage de contraintes que le niveau *EAL5* mais pas toutes celles d'*EAL6* recevra alors l'évaluation *EAL5+*.

Critiques

Cependant, le processus de certification proposé par les Critères Communs est vivement critiqué [62]. Il lui est reproché de se focaliser sur les aspects documentation aux dépends de la vérification des mécanismes implantés. En particulier, seul le niveau EAL7 nécessite une revue du code de l'application. Et même si les niveaux inférieurs requièrent une assurance des mécanismes de sécurité, peu de contraintes sont imposées sur l'implantation du système.

[62] met en avant le fait que seules les évaluations menées au niveau EAL5 valident une qualité des mécanismes de sécurité. Cette critique remet en cause le véritable but des Critères Communs et les garanties que ces contraintes apportent quant à l'assurance de sécurité.

Enfin, le coût du processus de certification est mis en avant. Celui-ci est extrêmement coûteux d'un point de vue pécuniaire et temporel. Certains systèmes sont déjà considérés comme obsolètes avant même la fin du processus de certification, remettant en cause son utilité.

Profils de protection et systèmes partitionnés

Les Critères Communs proposent un profil de protection pour la validation d'architectures partitionnées [67, 49]. Il décrit les exigences de sécurité devant être respectées par un système partitionné en vue de sa certification. Son existence montre l'intérêt des industriels pour ce type d'architecture : ce profil est le fruit d'un travail de la *National Security Agency* (NSA).

De son côté, la communauté MILS rédige également d'un profil de protection dédié à la validation et l'implantation de système respectant les contraintes de cette approche. Ce profil rassemble également les personnes impliquées dans le profil précédemment cité [67, 49] et décrit les aspects spécifiques à l'approche MILS. Il devrait être publié prochainement [97].

Transparence de l'évaluation

Le processus d'évaluation et de certification se veut transparent : les profils de protection et les rapports de certification sont disponibles en ligne. Il est donc possible de savoir comment atteindre un certain niveau d'assurance de sécurité (technologies et approches utilisées, etc.) et de connaître les mécanismes de sécurité utilisés par les systèmes soumis au processus de certification.

Exemples de certifications EAL

De nombreux produits de notre quotidien ont été soumis à l'évaluation des Critères Communs : *Microsoft Windows Vista* est évalué au niveau **EAL1**.

Solaris et *Red-Hat Enterprise* au niveau **EAL4+**.

Dans le domaine critique, plusieurs systèmes d'exploitation sont évalués à un haut niveau (supérieur à EAL5) :

- *XTS-400* est un système d'exploitation de type UNIX avec de nombreuses fonctions de sécurité, certifié au niveau **EAL5+** [27].
- *INTEGRITY 178-B* est un système d'exploitation partitionné, utilisé dans le domaine avionique. Il est certifié **EAL6** [26].
- *LynxSecure Embedded Hypervisor and Separation Kernel* est une machine virtuelle partitionnant des instances de systèmes, certifié au niveau **EAL7** [68].

2.3.3 Assurance sur le code de l'application

Nous avons présenté deux approches (DO178B ou Critères Communs) qui ont montré qu'il était nécessaire d'assurer la bonne implantation des mécanismes de sécurité et de sûreté d'un système. Afin de satisfaire ces exigences, une méthode consiste à analyser le code de l'application afin de détecter les erreurs potentielles qu'il contient.

Il est généralement admis qu'une erreur est présente toutes les 1000 lignes de code[71]. Cela est principalement dû au facteur humain, les développeurs introduisant des erreurs au fur et à mesure du développement (inattention, complexité du code, écueils syntaxiques/sémantiques du langage d'implantation, etc.).

Toutefois, les techniques de développement dédiées, la relecture et l'analyse de code permettent de limiter les nombre d'erreurs. Une analyse sur le code présent dans la navette spatiale *Space Shuttle* a montré qu'il contient moins de 0.1 faute pour 1000 lignes de code [10], soit 10 fois moins que la moyenne. Ce résultat a été obtenu par plusieurs relectures et révisions du code.

Un autre exemple en provenant du ministère de la défense anglais (*Ministry of Defense*, *MoD*) a également procédé à une analyse du code produit pour le contrôle de l'avion *« Super » Hercules C-130J* produit par Lockheed Martin [53]. Avant l'analyse du code, le taux d'erreur était de 23 erreurs pour 1000 lignes de code. Après utilisation d'outils d'analyse statique de code, ce taux était de 1.3.

D'un point de vue industriel, des études [48, 91] ont montré que le nombre de fautes dans les produits commerciaux était généralement de l'ordre de 30 erreurs pour 1000 lignes de code avant la mise sur le marché et qu'après une étape de retour des utilisateurs (et corrections des erreurs), ce chiffre tombait à 10 erreurs pour 1000 lignes de code. Ces chiffres sont contredits par d'autres études comme [59] qui avance que le taux d'erreur dans les produits commerciaux est de l'ordre de 4 erreurs pour 1000 lignes de code.

Si les études divergent sur le nombre précis d'erreurs présent dans les produits commerciaux, on peut se satisfaire d'une estimation large et considérer que ces produits contiennent entre 1 et 10 fautes pour 1000 lignes de code. Même à son plus bas niveau, ce taux est bien trop important pour les systèmes critiques et oblige donc à recourir à des outils d'analyse beaucoup plus fine.

La présentation du standard DO178B (section 2.3.1) a indiqué que des

domaines critiques tels l'avionique nécessitaient que le code des applications soit évalué en terme de couverture. Cependant, les techniques traditionnelles de production de code ne sont pas appropriées pour créer du code fiable et exempt d'erreur. A l'inverse, il a été soulevé (à travers l'exemple du *Space Shuttle*) qu'une analyse et vérification indépendante du code permettait de réduire le nombre d'erreurs. Il est donc nécessaire de réaliser plusieurs analyses afin de détecter les erreurs au sein du code et assurer sa qualité.

Nous présentons maintenant les différentes approches d'analyse de code.

L'analyse statique de code

L'analyse statique de code est un ensemble de techniques qui s'attache à détecter les erreurs potentielles dans un programme sans l'exécuter. Le but de ces techniques est d'analyser les instructions et le flux des données pour essayer de trouver de fautes potentielles (par exemple, le débordement d'un type). L'avantage réside dans la capacité à déceler des erreurs sans à avoir exécuter le programme.

Plusieurs outils aussi bien libres que commerciaux, sont disponibles. Une liste des outils utilisés est disponible sur [3] et dans [121].

Dans la communauté Française, le projet Frama-C [24], initié par le *Commissariat à l'Energie Atomique* (*CEA*) et l'*Institut National de Recherche en Informatique et en Automatique* (*INRIA*), vérifie plusieurs propriétés du code (valeur d'une variable, etc.). Il divise un programme en plusieurs sous-programmes indépendants, facilitant ainsi l'analyse de l'application globale.

De nombreux autres outils en provenance du monde commercial (PolySpace) ou académique (E-SETHEO [114]) sont disponibles [121]. En fonction des approches, plusieurs langages (C, Ada, etc.) sont analysés afin de détecter les erreurs de sécurité (dépassement de tampon) que de sûreté (non déterminisme du code).

La génération de code

La génération de code consiste à produire automatiquement l'implantation d'un système à partir de ses spécifications. Ce concept n'est pas nouveau et est déjà utilisé dans de nombreux produits commerciaux comme Matlab/Simulink [90] ou encore SCADE (Esterel ou Lustre [56]).

Ces approches suppriment le facteur principal de création d'erreur : le programmeur. Le formalisme source (Matlab/Simulink [90], Lustre [56]) fait abstraction des notions complexes des langages de programmation et cache les difficultés d'implantation. Le processus de génération transpose alors ces constructions abstraites en une implantation dans le langage cible (C, Ada, etc.). Suivant l'expressivité du formalisme source, le générateur produit un exécutable complet ou un squelette : des langages avec une sémantique forte (tel Lustre [56]) permettent de générer l'implantation intégrale d'un programme alors que des approches utilisant UML [65] se limiteront à la production de squelettes (sémantique faible ne comprenant pas toutes les informations nécessaires à la génération complète du système).

Le programmeur spécifie son système via ce formalisme dédié et ne se préoccupe aucunement des détails d'implantation. Tous les aspects complexes de la programmation sont donc cachés et gérés par le générateur de code (manipulation de pointeurs, gestion des types de données, etc.).

Cependant, plusieurs problèmes se posent aujourd'hui aux outils utilisant la génération de code :

1. **la sémantique du formalisme d'entrée** doit être appropriée et décrire parfaitement toutes les contraintes et exigences du système. Si tel n'est pas le cas, le développeur devra alors compléter manuellement le code, supprimant potentiellement les caractéristiques de sûreté apportées par l'étape de génération.
2. **le niveau d'assurance du code généré** : même si le code peut être considéré comme plus *fiable* et que l'on lui accordera davantage de confiance, il faut toutefois que ce code puisse être qualifié au regard d'un SIL ou DAL. Pour cela, il faut que le code généré soit certifié ou que le générateur de code lui-même soit certifié.
3. **le surplus de code** (appelé couramment *« overhead »*). Les approches de génération de code introduisent souvent du code inutilisé par le système à cause de stratégies de génération non adaptés aux besoins du système (ajout d'accesseurs, introduction de bibliothèques dont le code généré est dépendant). C'est particulièrement problématique quand certaines parties du code généré ne sont pas couvertes et remet en cause la certification (niveau A, B ou C de la norme DO178B par exemple)
4. **l'intégration avec du code tiers**. Le code généré doit bien souvent s'interfacer avec du code tiers, la génération de l'ensemble du système étant impossible et le code généré doit s'interfacer avec d'autres programmes. Cette intégration est bien souvent réalisée manuellement, remettant alors en cause les propriétés de sûreté apportées.
5. **la transposition des problèmes du code au modèle**. Le processus de génération assure que le code produit est conforme au modèle fourni en entrée, rien ne garantit que le modèle source est exempt d'erreurs. Autrement dit, les potentielles erreurs sont déportées du code vers le modèle et il est nécessaire de valider le modèle avant de procéder à la génération.

La génération de code présente de nombreuses caractéristiques renforçant la fiabilité de l'implantation d'un système (production de code déterministe à partir de patrons de génération, assistance de la procédure de tests, etc.). Cependant, elle montre certaines limites : le formalisme source doit avoir une sémantique riche pour offrir suffisament d'informations pour produire l'intégralité de l'application. De plus, ce langage source doit être analysable afin de vérifier les caractéristiques du système avant de procéder à la génération de l'application.

La couverture de code

Les techniques de couverture de code consistent à vérifier que la totalité du programme est exécutée sans qu'aucune faute ne survienne. On s'assure ainsi

que le système ne contient pas de code non testé et potentiellement dangereux.

La norme DO178B impose qu'une analyse de couverture de code soit menée afin de certifier un système. Les contraintes de cette analyse dépendent du niveau de certification (DAL) recherché.

Il existe trois différents niveaux de couverture du code :

1. **statement Coverage** (SC) ou couverture des instructions : assure que toutes les instructions sont exécutées au moins une fois. Ce niveau de couverture est requis pour la qualification d'un composant au DAL C de la norme DO178B.
2. **decision Coverage** (DC) ou couverture des décisions : assure que toutes les instructions sont exécutées et que toutes les conditions sont évaluées à vrai et faux. Ce niveau de couverture est requis pour la qualification d'un composant au DAL B de la norme DO178B.
3. **Modified Condition/Decision Coverage** (MC/DC) : assure que chaque instruction est exécutée et que chaque branche de chaque condition est évaluée au moins une fois avec une valeur différente. Ce type de couverture est explicité dans [63, 60]. Ce niveau de couverture est requis pour la qualification d'un composant au DAL A de la norme DO178B.

```
1 procedure P (A, B, C : Boolean) is
2 begin
3    if (A and then B) or else C then
4       Do_Something;
5    end if;
6 end P;
```

Listing 2.1 – Code Ada illustrant les méthodes de couverture

Les différentes méthodes de couverture de code sont illustrées par le source Ada du listing 2.1. La couverture des instructions (SC) est obtenue en vérifiant que les lignes 3 et 4 sont exécutées. Il faut donc que la condition de la ligne 3 soit évaluée à *vrai* et que l'instruction `Do_Something` soit exécutée. La couverture des décisions est réalisée en évaluant la condition de la ligne 3 à *faux* **et** à *vrai*. L'analyse assure donc que les deux branches de la condition soient testées et que les instructions découlant de ces évaluations soient exécutées.

Pour parvenir à une couverture de type MC/DC, il faut que chaque point d'entrée et de sortie du programme soit exécuté une fois et que chaque décision ait pris toutes les valeurs possibles au moins une fois. En pratique, il faut s'assurer que toutes les conditions sont évaluées à *vrai* et *faux*, que pour toute condition C d'une décision D, l'ensemble des vecteurs de D contient une valeur identique pour C qui implique que D soit évalué à *vrai* et *faux*. Sur l'exemple du listing 2.1, il est possible d'utiliser les vecteurs (V,V,F), (V,F,V), (V,F,F), (F,V,F). Avec ces vecteurs, toutes les conditions sont évaluées à *vrai* et *faux*. L'indépendance de l'impact de `A` est montrée par les vecteurs (V,V,F) et (F,V,F), l'indépendance de `B` est montrée par les vecteurs (V,V,F) et (V,F,F) et l'indépendance de `C` est obtenue par les vecteurs (V,F,V) et (V,F,F). La couverture MC/DC demande à ce que $n + 1$ tests soient effectués pour couvrir une condition composée de n décisions [63]. Ce qui se vérifie ici : la décision (ligne 3 du listing 2.1) est composée de 3 décisions et 4 vecteurs de valeurs sont utilisés pour la couvrir.

Toutefois, l'efficacité de cette approche est discutée dans la communauté scientifique [64, 115]. Le premier sentiment qu'ont les utilisateurs est que la fiabilité d'un système évolue linéairement avec son niveau de couverture. Cependant, [115] montre l'inverse et que seule la couverture intégrale du code est valable, les quelques morceaux non couverts pouvant impacter le code validé. La fonction définissant la sûreté d'un système en fonction de sa couverture du code est exponentielle. Ainsi, il est préférable d'avoir une analyse de couverture intégrale au niveau MC/DC, comme le requiert le DAL de niveau A de la norme DO178B afin de garantir la meilleure sûreté possible au niveau de l'implantation du système. Et paradoxalement, une faible couverture du code ne donnera quasiment aucun supplément d'assurance dans la fiabilité du système par rapport à un système non soumis à cette analyse.

En résumé, dans un contexte de systèmes critiques, seule la couverture intégrale du code fait sens, un taux de couverture trop faible n'apporte que peu d'assurance sur la fiabilité de l'application.

2.3.4 Synthèse des problèmes soulevés

Cette section a mis en lumière la problématique de conception des systèmes critiques : ces derniers doivent être construits suivant des méthodes rigoureuses, précises et vérifier plusieurs contraintes importantes (couverture de code, vérification de l'implantation par rapport aux spécifications, etc.).

Cependant, le respect des contraintes associées à ces méthodes de développement sont extrêmement coûteuses en termes de ressources (financières, temporelles, etc.) et difficiles à atteindre (manque d'outillage, faible intégration au cycle de développement, etc.). Le recours à différents formalismes de spécifications pour valider le système (associé à la vérification de la conformité de ces représentations), la vérification du code produit manuellement avec les spécifications ou l'analyse de couverture de code sont autant d'éléments qui s'ajoutent et rendent ce processus coûteux et lourd à supporter.

L'approche présentée au cours de ce manuscrit s'attachant à proposer une méthode de conception pour les systèmes critiques, elle doit intégrer des techniques réduisant les erreurs et assurant la conformité avec les standards de certification en :

1. validant le formalisme utilisé comme langage source par le générateur de code (objectif 1, section 1.4). Ceci assure alors que le générateur lui-même n'introduira pas d'erreur.

2. générant automatiquement l'application afin de supprimer toute erreur introduite par la production manuelle de code (objectif 2, cf 1.4).

3. analysant automatiquement l'application (test de couverture, etc.), assurant sa conformité aux standards de certification (objectif 3, section 1.4) et apportant des garanties quant à la qualité de son code.

Un tel processus permet d'améliorer la fiabilité du développement et limite l'introduction d'erreurs à chacune de ces étapes.

2.4 Limites des solutions existantes

Cette section détaille les limites des solutions présentées précédemment.

2.4.1 Sécurité

MILS

Formalisme de représentation

L'approche MILS représente l'architecture et introduit une abstraction de ces concepts afin d'assurer l'isolation des niveaux de sécurité.

Cependant, le formalisme de représentation est sémantiquement pauvre, l'expressivité est insuffisante pour le développement d'applications critiques. En particulier, il serait approprié de :

1. spécifier le lien entre les composants et les éléments (logiciels et matériels) de l'architecture qui les exécutent. Cette faculté permettrait à une meilleure analyse puisque l'association du logiciel et du matériel peut avoir un impact sur la sécurité du système.
2. pouvoir annoter les composants dans le but de spécifier les autres exigences que pourrait avoir le concepteur d'applications (ajout des aspects de performance ou de sûreté).

Dans un cycle traditionnel de développement, l'ajout de ces informations est réalisé par l'utilisation d'annotations pour analyser les différents aspects du système. Cependant, il est important d'éviter la multiplication des formats de représentations ; la transposition d'une représentation à une autre est coûteuse et source d'erreurs (précision de la sémantique de chaque langage, erreur de transformation des spécifications, etc.).

Plate-forme d'exécution

Afin de garantir leur isolation, MILS confine les niveaux de sécurité dans des partitions. Lors de l'exécution du système, MILS définit une plate-forme d'exécution sécurisée qui isole les partitions temporellement et spatialement. Les communications entre partitions sont analysées à l'exécution afin d'assurer que seules les partitions autorisées communiquent.

Toutefois, le manque de précision dans la description de la plate-forme MILS limite son utilisation. La relation entre les spécifications de haut-niveau et les services de la plate-forme d'exécution ne sont pas explicités, laissant alors le développeur seul responsable des choix quant à la transformation des spécifications vers du code exécutable. De plus, les services devant être fournis par la plate-forme d'exécution ne sont pas clairement définis (publication d'une API, guide de l'utilisateur/développeur, etc.) et toujours en cours de standardisation au sein d'un profil de protection [97].

Cependant, à partir de la description globale de l'architecture, il est évident que les concepts d'isolation temporelle et spatiale sont semblables à ceux introduits par le standard ARINC653 (présenté dans la section 2.2.2). Pourtant, l'absence de définition formelle des services, de relation entre l'abstraction du système et sa plate-forme d'exécution font qu'il est :

1. difficile de certifier la conformité d'une application avec cette approche.
2. complexe et coûteux de produire un système à partir de sa seule spécification MILS

Politiques de sécurité

Les politiques de sécurité (telles Bell-Lapadula, Biba, Chineese Wall) posent plusieurs problèmes quant à leur utilisation.

En premier lieu, chaque modèle apporte des solutions spécifiques, très contraignantes et il est impossible de les intégrer ensemble (par exemple, Biba et Bell-Lapadula ne sont pas intégrables, leurs règles étant totalement opposées comme le montrent les figures 2.2 et 2.3).

De plus, ces modèles agissent à un niveau abstrait et ne proposent pas de représentation commune des composants de sécurité. Ces approches mènent à une première description de la politique de sécurité d'un système et peuvent constituer une première phase de spécification de la sécurité. Mais cela ne donne aucunement le découpage logique de l'application avec ses différents composants.

Enfin, le déploiement de ces modèles n'est pas défini et il n'est pas dit quels mécanismes de sécurité doivent être mis en œuvre pour respecter ces politiques de sécurité. Toutes ces approches ne donnent aucune indication quant aux services de sécurité qui doivent être implantés au sein d'une plate-forme conforme aux règles de MILS.

Algorithmes de chiffrement

Les algorithmes de chiffrement constituent des mécanismes de protection de l'information puissants et doivent être davantage liés à la spécification du système. En particulier, un algorithme ou une clé de chiffrement différents seront utilisés pour protéger les données d'un niveau de sécurité spécifique. Dès lors, il devient intéressant de spécifier ses algorithmes et d'analyser les interactions possibles entre niveaux de sécurité.

Il serait donc nécessaire, au niveau de la spécification, de connecter ces algorithmes et leur spécificités de configuration (fonctions et clés de chiffrement utilisées, etc.) avec les contraintes de la spécification (niveaux de sécurité). Ceci permettrait de les analyser plus rapidement et de détecter les erreurs potentielles quant à leur utilisation (erreur de configuration, absence de chiffrement pour les niveaux de sécurité importants, etc.).

Dans le contexte des systèmes critiques, il serait pertinent d'associer l'utilisation de ces mécanismes de sécurité aux niveaux de sécurité du système (*confidentiel*, *secret*, etc.). Cette nouvelle information spécifierait alors les niveaux de sécurité utilisés par chaque composant d'une architecture MILS, et permettrait donc de valider la bonne isolation des niveaux de sécurité.

2.4.2 Sûreté de fonctionnement

Analyse des risques

Bien que les approches décrites offrent une capacité d'analyse des fautes et de leur propagation, elles présentent plusieurs limites :

1. spécification au moyen d'un formalisme de représentation dédié, limitant son utilisation avec d'autres approches, limitant ainsi son intégration dans les approches de développement actuelles
2. aucun suivi des exigences de sûreté au sein du code. Ici, il peut être pertinent de rajouter les exigences décrites au sein du code correspondant à l'implantation du système.
3. spécification explicite des fautes. Ici, le développeur doit spécifier manuellement les fautes. L'utilisation d'une sémantique plus précise dans la représentation du système et la spécialisation des composants, pourrait automatiser en partie la détection des fautes en utilisant des inférences sur les types de connections, leurs propriétés et leurs comportements.

ARINC653

Manque de description, d'analyse

ARINC653, de part l'isolation qu'il assure entre les composants, détecte et recouvre les fautes logicielles au sein des partitions lors de l'exécution du système. Le standard spécifie clairement les interactions possibles au sein et entre les partitions.

Cependant, ARINC653 ne propose pas d'abstraction du système, ni une spécification haut-niveau de ses mécanismes. Ce manque limite les possibilités d'analyse du système et la détection de la propagation des erreurs entre partitions.

Le standard propose une description globale de l'architecture par la définition d'un fichier de déploiement (fichier XML). Cependant, les informations qu'il contient sont insuffisantes pour procéder à une analyse des fautes et de leur impact. Il est donc nécessaire de décrire ce type d'architecture logicielle au moyen d'un autre format de spécification, afin de pouvoir procéder à une meilleure analyse de ces systèmes.

Localisation des pilotes de périphériques

Bien que le standard ARINC653 explicite clairement la spécification de la plupart des services, la méthode d'implantation des pilotes de périphériques est laissée à la discrétion du concepteur de la plate-forme.

Toutefois, ce détail d'implantation est particulièrement important : implantés dans le noyau, une erreur dans un pilote peut alors porter atteinte au code exécuté dans le module, telle la gestion de l'isolation temporelle ou spatiale.

Cependant, la gestion des périphériques peuvent être confinée dans des partitions. Dans ce cas, ces dernières nécessitent alors des services noyaux dédiés pour communiquer avec le matériel. Or, le standard ne précise aucunement les mécanismes d'interface partitions/noyau alors que cette fonctionnalité est

essentielle et peut avoir un impact sur les autres services (isolation spatiale/-temporelle).

2.4.3 Méthodes de conception

Standards de certification : DO178B et Critères Communs

Les standards DO178B et Critères Communs décrivent les exigences pour qu'un système soit certifié. Ces standards définissent plusieurs niveaux, les plus exigeants imposants des contraintes de plus en plus importantes.

DO178B

DO178B ne décrit pas de manière précise toutes les exigences de certification et le processus de vérification associé. En particulier, si certaines contraintes sont explicites et claires (couverture de code de l'application en fonction du DAL), d'autres sont plus floues (méthodes et moyens utilisés pour la traçabilité des exigences au cours du développement).

Critères Communs

Les Critères Communs sont particulièrement opaques quant à la définition des documents et des preuves à fournir concernant les mécanismes de sécurité. En particulier, le standard utilise une suite de tests afin de vérifier le bon comportement du logiciel. Cependant, celle-ci peut se révéler incomplète, certaines fonctions critiques n'étant soumise à aucun test.

De même, le recours à la vérification formelle pour les plus haut-niveaux de sécurité implique souvent l'utilisation d'une abstraction du système et n'est que peu lié à l'implantation. Toutefois, ce standard fait autorité dans le domaine industriel et se doit d'être considéré dans le contexte de la construction d'une méthode de conception de systèmes sécurisés.

Assurance sur le code source

Les approches présentées ont pour but de créer du code plus fiable, en supprimant le caractère *« manuel »* de production (génération de code) ou en le vérifiant (analyse statique de code, analyse de couverture).

Analyse statique de code

Dans le cas de l'analyse statique de code, bien que ces outils vérifient les détails de l'implantation, ils sont limités à la sémantique du langage qu'ils analysent. Autrement dit, ils ne peuvent qu'être utilisés pour vérifier des mécanismes d'implantation et non une architecture globale. Par exemple, un outil d'analyse statique de code ne sera pas utilisé pour assurer qu'une entité d'un faible niveau de sécurité enverra une donnée à une entité d'un haut niveau de sécurité.

Il faut donc utiliser cet outil comme un support vérifiant l'absence d'erreur syntaxique et sémantique dans le code de l'implantation. Cependant, il ne permet aucunement de vérifier les exigences des spécifications. En ce sens, il est nécessaire de coupler cette approche au sein d'un processus global, apportant des garanties au niveau des spécifications.

Génération de code

La génération de code apporte une avancée significative dans la qualité du système créé. Cependant, le code généré doit toujours être soumis à un effort de certification, à moins que le générateur lui-même le soit.

Sauf cas particuliers de générateurs très optimisés, ces outils génèrent la plupart du temps de nombreuses fonctionnalités inutiles, alourdissant alors l'empreinte mémoire du système généré et minimisant le taux d'instructions effectivement exécutées.

Couverture de code

La couverture de code consiste à analyser l'exécution du système et à assurer l'exécution de chaque instruction lors de tests. Cette approche est toutefois très contraignante et n'est utilisable qu'avec un sous-ensemble des langages de programmation (ex : Ada ou sous-ensemble de C). Cependant, dans le cadre des systèmes critiques, cette contrainte n'a que peu d'impact car ces deux langages sont majoritairement utilisés.

De plus, même si elle se révèle optimale, la couverture de code ne peut assurer seule le bon fonctionnement d'un système. En particulier, d'autres fautes peuvent survenir pendant l'exécution (par exemple : réception d'une valeur erronée, dépassement de tampon). Ces erreurs ne sont pas analysées par ces techniques.

Enfin, il est difficile, voire impossible, d'assurer la couverture de l'intégralité d'un système. En particulier, certains services peuvent n'être jamais utilisés alors qu'ils ont une importance (par exemple, les exceptions processeur). Certaines parties du code devront donc être analysées séparément par l'ingénieur souhaitant certifier le système.

Ces arguments montrent que, dans notre contexte de sécurité/sûreté, la couverture de code est un outil nécessaire mais pas suffisant. Il apporte une garantie sur la fiabilité du code, mais doit être couplé à d'autres approches d'analyse. En particulier, la validation de l'architecture du système (afin de détecter les erreurs pouvant survenir à l'exécution), ainsi que la génération de code (assurant la conformité de l'implantation avec la spécification).

2.4.4 Synthèse

Le tableau 2.4 constitue une grille de lecture des différentes approches étudiée et synthétise leur pertinence dans la constitution de notre approche par rapport aux objectifs de nos travaux (section 1.4).

En outre, il met en lumière le fait que les travaux étudiés apportent une plus-value à chaque niveau du cycle de développement. Cependant, ces approches sont déconnectées entre elles et il apparait nécessaire de pouvoir les connecter entre elles, afin que les validations de sûreté/sécurité réalisées sur les spécifications se répercutent sur les phases suivantes du développement (implantation, certification, exécution).

	Représentation et validation (objectif 1)	**Génération automatique et plate-forme sécurisée (objectif 2)**	**Certification de l'implantation (objectif 3)**
Analyse des fautes	++	–	–
ARINC653	+	++	–
MILS	+	++	–
Politique de sécurité	++	–	–
Algorithmes de chiffrement	+	++	–
DO178B	+	+	++
Critères Communs	–	+	++
Assurance sur le code de l'application	–	++	++

Table 2.4 – Pertinence de chaque approche avec nos objectifs

Synthèse de ce chapitre

Ce chapitre a dressé un aperçu des méthodes destinées à assurer sécurité et sûreté de fonctionnement. L'objectif de cette thèse étant d'assurer ces propriétés de bout en bout, ce chapitre a présenté des approches qui montrent leur garantie au cours du cycle de vie du système critique : spécification (représentation des exigences et mécanismes de sécurité/sûreté) et l'implantation (services d'isolation des niveaux de sécurité). Il a également introduit les contraintes associées à la construction de systèmes critiques (génération/couverture de code, etc.).

Il a été souligné qu'il est difficile d'assurer les mécanismes de sécurité et de sûreté . Ces aspects doivent être spécifiés et validés (objectif 1, section 1.4) mais aussi correctement implantés (objectif 2, section 1.4). Cependant, la vérification de ces deux aspects au sein du même système reste problématique, les techniques actuelles utilisant des formalismes spécifiques et coûteux à intégrer. Il est donc nécessaire de définir une méthode de représentation unique afin de s'affranchir du coût de transformation des spécifications.

Ces exigences doivent également satisfaire les contraintes inhérentes au développement de systèmes critiques. Ceux-ci imposent de nombreuses règles contraignantes en termes de validation et de certification (assurance des mécanismes d'implantation, analyse de couverture de code) se révélant coûteuses et difficiles à satisfaire (manque d'outils, absence d'automatisation, etc.). Il est donc nécessaire de les intégrer dans un processus de développement global et proposer des méthodes et outils pertinents automatisant ces tâches de validation/certification (objectif 3, section 1.4).

Chapitre 3

Approche proposée

Sommaire du chapitre

3.1 Identification des solutions pertinentes pour la constitution de notre approche 69

3.1.1 Architecture partitionnée pour la sécurité ou la sûreté 69

3.1.2 Représentation et analyse 70

3.1.3 Standards et processus de développement 71

3.2 Présentation de l'approche proposée 72

3.2.1 Spécification des architectures partitionnées et des contraintes de sécurité/sûreté associées . . . 73

3.2.2 Validation de l'architecture et de ses exigences de sécurité/sûreté 75

3.2.3 Implantation automatique à partir des spécifications validées 76

3.2.4 Certification automatique du système généré 77

3.3 Langages de modélisation 79

3.3.1 Motivations d'une représentation unifiée de l'architecture 79

3.3.2 Langages existants 79

3.3.3 Motivation du choix d'AADL 82

3.3.4 Présentation du langage AADL 83

3.3.5 AADL et outils de validation 87

Rappels

Le précédent chapitre a présenté la problématique en listant les solutions existantes suivant trois aspects :

1. **sécurité** : approches visant à assurer la mise en place d'une politique de sécurité dans les spécifications et à définir les mécanismes nécessaires à la protection des données dans l'implantation.
2. **sûreté** : services et approches pour assurer son respect lors de la conception du système (détection des erreurs, procédures de recouvrement).
3. **standards de développement** : contraintes imposées pour le développement de systèmes critiques.

Le précédent chapitre a également énuméré les limites des travaux existants, mettant en lumière les aspects pouvant être améliorés par notre approche.

Objectifs du chapitre

Ce chapitre dégage les éléments pertinents des solutions étudiées et présente ceux qui sont repris et/ou améliorés pour constituer notre approche. Trois principaux manques sont identifiés et constituent une base de travail pour améliorer l'assurance de sécurité/sûreté dans la construction de systèmes critiques.
L'approche est ensuite détaillée en reprenant les objectifs identifiés en introduction (section 1.4) : spécification et validation de l'architecture, implantation et certification. Pour chaque objectif, les solutions pertinentes sont référencées.
La définition de notre approche met en lumière le besoin de représentation unique de l'architecture avec ses contraintes de sécurité et de sûreté pour diriger l'ensemble du processus de développement. A ce titre, ce chapitre se termine par une section dédiée à la présentation des langages de spécification existants et détaille les motivations du choix du langage AADL.

3.1 Identification des solutions pertinentes pour la constitution de notre approche

Cette section présente les solutions que nous avons identifiées comme pertinentes. Ces dernières sont donc intégrées dans le cadre de nos travaux. Nous rappelons leurs limites et les possibles améliorations qui seront réalisées.

3.1.1 Architecture partitionnée pour la sécurité ou la sûreté

Les paragraphes suivants résument l'utilité de l'isolation fournie par l'architecture partitionnée et décrivent les manques que nous avons identifié pour une assurance commune de sécurité et de sûreté.

Pertinence et adéquation de l'isolation pour la sécurité/sûreté

L'architecture partitionnée est utilisée soit dans un contexte de sécurité (MILS, section 2.1.1) soit dans un contexte de sûreté (ARINC653, section 2.2.2).

Dans ces deux cas, les composants du système sont confinés dans des partitions soumises à l'isolation temporelle et spatiale. Les raisons de cette isolation dépendent de la contrainte (sûreté, sécurité) assurée par le système comme le résume le tableau 3.1.

Type d'isolation	**Sécurité**	**Sûreté**
Spatiale	Isolation et contrôle des échange entre les niveaux de sécurité.	Assurance de l'absence d'impact entre entités de criticité différente.
Temporelle	Absence d'attaque basée sur l'analyse du temps d'exécution.	Garantie d'un temps d'exécution fixe pour chaque partition.

Table 3.1 – Utilisation du partitionnement pour assurer la sécurité ou la sûreté

Ces motivations montrent l'utilité de la notion de partitionnement pour la sécurité et la sûreté. Alors que les approches existantes (ARINC653 ou MILS) l'utilisent pour fournir indépendemment chaque exigence, il semble pertinent de définir une plate-forme les garantissant simultanément.

Au delà de la seule isolation, d'autres services sont également requis. La sûreté de fonctionnement nécessite de détecter et recouvrir les erreurs survenant à l'exécution (service de *Health Monitoring* d'ARINC653 décrit en section 2.2.2). La sécurité demande le chiffrement des données échangées entre les partitions lorsque celles-ci transitent par des moyens de transmissions non sécurisés (par exemple bus de type *Ethernet*).

Besoin de sûreté et de sécurité

A l'heure actuelle, les systèmes actuels proposent des services afin d'assurer la sécurité ou la sûreté mais n'intègrent pas simultanément ces deux

contraintes. Les approches étudiées (ARINC653 ou MILS) apporte des services d'isolation afin de garantir séparément une de ces exigences.

De plus, ces systèmes introduisent des contraintes qu'il est difficile de vérifier ou certifier, aussi bien d'un point de vue sûreté que sécurité :

- **Sûreté** : la politique de recouvrement d'erreurs (actions déclenchées lors de la détection d'une erreur) peut avoir un impact entre partitions évaluées à des niveaux de sûreté/sécurité hétérogènes. Par exemple, une erreur survenant dans une partition classifiée à un faible niveau de sûreté/sécurité peut impacter une partition de plus haut niveau (cas d'une communication entre ces deux partitions). Pour cette raison, il est nécessaire d'analyser son impact sur l'architecture.
- **Sécurité** : les communications entre les partitions sont validées et vérifiées pour assurer l'isolation des niveaux de sécurité.

D'autres exigences communes à la sécurité et à la sécurité compliquent la validation du système. Par exemple, l'ordonnancement hiérarchique utilise plusieurs algorithmes simultanément et rend difficile l'analyse de ses contraintes temporelles. Il est donc nécessaire de définir de nouvelles méthodes répondant à ces problèmes.

3.1.2 Représentation et analyse

Les paragraphes suivants rappellent le besoin de représentation et d'analyse de l'architecture dans le contexte du développement de systèmes critiques, justifiant l'intégration de ces aspects dans notre approche. Ils rappellent leurs limites qui devront alors être contournées par nos solutions.

Pertinence et adéquation

Notre études préliminaires des solutions existantes a montré l'importance de la représentation et de l'analyse du système. Sa validation à l'aide d'un formalisme de haut-niveau assure l'absence d'erreur en amont des autres étapes du développement. Cette phase de validation est importante et doit être conduite au plus tôt, 70% des erreurs étant introduites lors de la phase de spécification [77].

Deux méthodes de représentation et d'analyse du système ont été présentées pour la :

1. **sûreté** : l'approche HAZOP (section 2.2.1) analyse le système, ses potentielles fautes et leurs propagation. Elle permet de constater l'impact d'une faute et de détecter celles étant critiques (par exemple, une faute d'un composant non-critique impactant un composant critique).
2. **sécurité** : l'approche MILS (section 2.1.1) et les politiques de sécurité permettent une analyse de l'architecture afin de veiller à l'isolation des niveaux (top-secret, secret, etc).

Ces méthodes de validation représentent le systèmes et ses contraintes au moyen d'un langage spécifique, non standardisé. L'utilisateur doit donc transposer, pour chaque exigence (sécurité, sûreté) les spécifications de son système vers cette notation. Cette étape de traduction peut introduire des erreurs (mauvaise interprétation des spécifications, erreur de traduction, etc.) et remet

donc en cause l'utilité de leur validation. De plus, ces formalismes ne sont pas réutilisés dans les autres étapes du développement, rien ne garantit donc que les mécanismes implantés correspondent à ceux validés.

Multiplicité des formalismes de spécification

Les techniques de validation présentées (HAZOP, section 2.2.1, MILS, section 2.1.1 ou les politiques de sécurité, section 2.1.2) reposent sur des formalismes spécifiques et décorellés les uns des autres. Cela pose un problème en termes de coût (temps de traduction d'un format de représentation vers un autre) et d'erreurs (l'utilisateur peut introduire des erreurs dans la traduction des spécifications textuelles vers chaque format de représentation).

De par l'utilisation de multiples représentations et outils, la validation des caractéristiques des systèmes partitionnés demande un travail important, proportionnel au nombre d'exigences à valider : pour chacune d'elles, l'utilisateur doit transposer les spécifications dans un formalisme dédié et utiliser un outil spécifique.

Cette étape de validation pourrait être allégée en unifiant les formalismes de représentation et les outils de validation. L'utilisateur exprimerait l'architecture de son système et l'ensemble de ses caractéristiques au sein d'une unique représentation. Celle-ci serait exploitée par des outils de validation qui procéderait alors à la validation de chaque exigence (isolation, ordonnancement, etc).

3.1.3 Standards et processus de développement

Les paragraphes suivants rappellent les exigences et l'importance des standards de développement. Ils énumèrent par ailleurs leurs limites que notre approche doit prendre en compte en vue de leur intégration.

Pertinence et adéquation

Notre étude préliminaire a mis en avant plusieurs standards référents (DO178B et Critères Communs) dans la certification des systèmes critiques. Ces derniers requièrent la vérification de la bonne implantation des exigences du système et le respect de plusieurs contraintes fortes (telle la couverture de code).

Ces standards imposent un processus de développement contraignant et rigoureux consistant à vérifier le système de bout en bout :

- spécifications : validation des caractéristiques pour prévenir toute erreur dans les étapes suivantes du développement. Comme détaillé dans les paragraphes précédents (section 3.1.2), cette étape est coûteuse et source d'erreurs.
- implantation : traduction des spécifications en code exécutable. Cette étape est source d'erreur de par le caractère *humain* du processus.
- certification : analyse du système et de ses caractéristiques (ordonnançabilité, couverture de code, etc.). Cette étape étant réalisée manuellement

par des ingénieurs qui analysent le système, celle-ci peut être source d'erreur : les spécifications pouvant être mal interprétées et mener à certifier un système ne correspondant pas à la description initiale.

Ces différentes étapes sont isolées les unes des autres et les composants (modèles, représentation, code, etc.) ne sont pas réutilisés. Par exemple, les représentations du systèmes produites à des fins de validation ne sont pas réutilisées pour l'implantation ou la certification. Ce problème est discuté dans les paragraphes suivants.

Problème d'unification et d'automatisation

Le processus de développement doit être amélioré suivant deux axes :

1. une unification par une réutilisation des éléments. Par exemple, les représentations du système peuvent être réutilisées pour l'implantation ou la certification.
2. une automatisation de chaque étape du développement.

L'unification du processus introduit un lien entre chaque étape. Les spécifications doivent être décrites à l'aide d'un langage formalisant les contraintes du système afin d'éviter tout ambiguité. La syntaxe et la sémantique choisie doit être suffisament précise pour être exploitable par des programmes.

L'utilisation d'une telle représentation permet sa réutilisation à chaque étape du développement et supprime l'utilisation de multiples formalismes, problème identifié dans les sections précédentes (section 3.1.2).

L'automatisation du processus supprime toute étape manuelle source d'erreur. Les techniques de développement actuelles ont montré leurs limites en termes :

1. qualitatif : la validation, certification ou le développement manuels constituent une source importante d'erreur (section 1.3) ;
2. coût : l'emploi de plusieurs personnes, la vérification et le test de leur travail augmentent les coûts temporels et financiers [77].

L'automatisation de chaque aspect du développement (validation, implantation, certification) assure alors leur bonne réalisation et supprime toute erreur relative au caractère « humain ». De plus, elle apporte un gain de temps significatif, tout le processus de développement étant supporté par des outils.

Dans notre contexte, elle assure la bonne implantation des mécanismes de sécurité et de sûreté mais également leur certification. Ainsi, en guise de résultat, l'utilisateur obtiendrait automatiquement une application conforme à ses exigences.

3.2 Présentation de l'approche proposée

L'approche proposée, illustrée dans la figure 3.1, se divise en quatre étapes :

1. spécification : l'utilisateur décrit l'architecture et les caractéristiques de son système ;
2. validation des spécifications : détection des erreurs potentielles ;
3. implantation automatique : génération de code à partir des spécifications ;
4. certification : analyse de l'application créée, validation de son exécution par rapport aux spécifications.

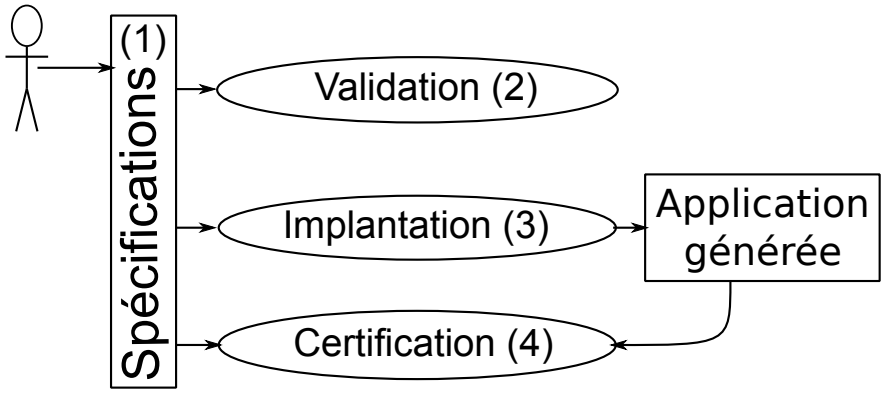


FIGURE 3.1 – Processus de développement de l'approche proposée

Les sections suivantes détaillent chaque partie de l'approche. La première motive le besoin de représentation unique de l'architecture partitionnée et de ses caractéristiques. La seconde décrit l'étape de validation du système. La troisième présente le processus de génération automatique de code et une dernière explique les différents mécanismes de certification.

Pour chaque thème, les éléments pertinents des solutions existantes sont mis en relief, positionnant ainsi notre contribution par rapport aux travaux existants.

Les étapes de ce processus s'appuient sur un langage de spécification unique. Une dernière section présente les langages existants et motive notre choix pour le langage AADL.

3.2.1 Spécification des architectures partitionnées et des contraintes de sécurité/sûreté associées

Motivations

La critique de l'existant (section 3.1.2) a mis en lumière les problèmes posés par l'usage de multiples représentations du système. Il est donc nécessaire de le spécifier les exigences du système au moyen d'un formalisme de représentation commun. Celui-ci doit être extensible afin de donner l'opportunité à l'utilisateur de spécifier d'autres contraintes que celles relatives à la sécurité et la sûreté (ex : ordonnancement).

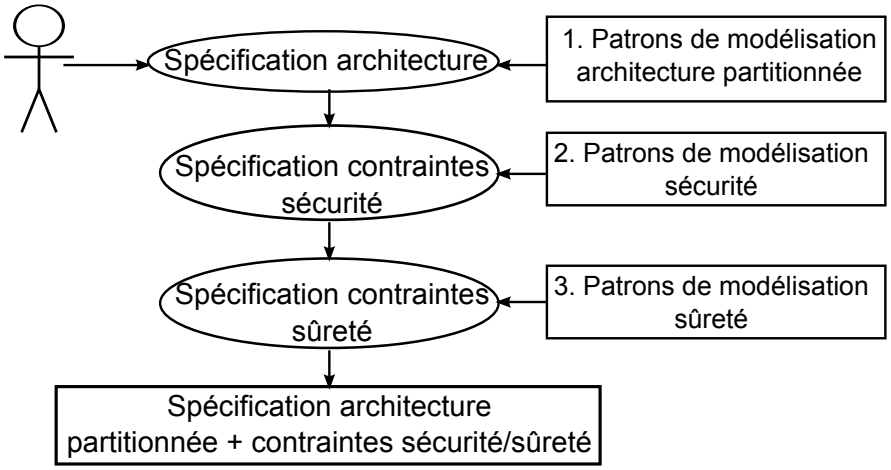


FIGURE 3.2 – Aspect modélisation de l'approche proposée

Conception de patrons de modélisation pour les systèmes partitionnés

Afin de guider l'utilisateur dans la description de l'architecture, il faut proposer une méthode de représentation des systèmes partitionnés, indiquant quels éléments du langage doivent être utilisés.

Elle doit spécifier les caractéristiques nécessaires à chaque étape du développement (validation, implantation, certification) afin d'être utilisée par des outils les automatisant.

Cette méthode de représentation introduit plusieurs patrons de modélisation (figure 3.2) qui définissent des règles de spécifications d'une architecture partitionnée (élément 1 de la figure 3.2) avec ses exigences de sécurité et sûreté (éléments 2 et 3 de la figure 3.2). De manière plus pragmatique, ces patrons établissent la correspondance entre les exigences du concepteur d'application (ex : isolation spatiale d'une partition) et le langage de spécification (ex : composant mémoire du langage de spécification). Cette première étape guide l'utilisateur dans la conception de son système en imposant une unique méthode de représentation de ses contraintes.

Reprise de l'existant

Les patrons de modélisation proposés représentent l'architecture partitionnée et ses mécanismes de sûreté/sécurité. Leur conception s'est faite à partir d'une analyse des architectures partitionnées existantes (ARINC653 et MILS, voir sections 2.2.2 et 2.1.1). De la même manière, les exigences de validation découlant de ces patrons ont été conçus par une revue des solutions existantes.

Les patrons de modélisation pour la sûreté utilisent les erreurs (ordonnancement, division par zéro, violation mémoire, etc.) et politiques de recouvrement (redémarrage d'une tâche, d'une partition, etc.) identifiées dans le standard ARINC653 (section 2.2.2) ainsi que dans les approches d'analyse de fautes (HAZOP, section 2.2.1).

Les patrons proposés pour modéliser les aspects sécurité transposent les

techniques de représentation de MILS (description des niveaux de classification, association aux éléments de l'architecture, etc). Ils spécifient également les mécanismes de chiffrement qui leurs sont associés (voir section 2.1.3), caractéristique non décrite par le formalisme défini par MILS.

3.2.2 Validation de l'architecture et de ses exigences de sécurité/sûreté

Motivations

Dans notre contexte, la validation de l'architecture apparaît comme un pré-requis : elle permet de détecter les fautes au plus tôt et est requise par les standards de certification (tels DO178B ou Critères Communs). Elle doit donc être la plus complète possible et analyser autant de caractéristiques du système que possible.

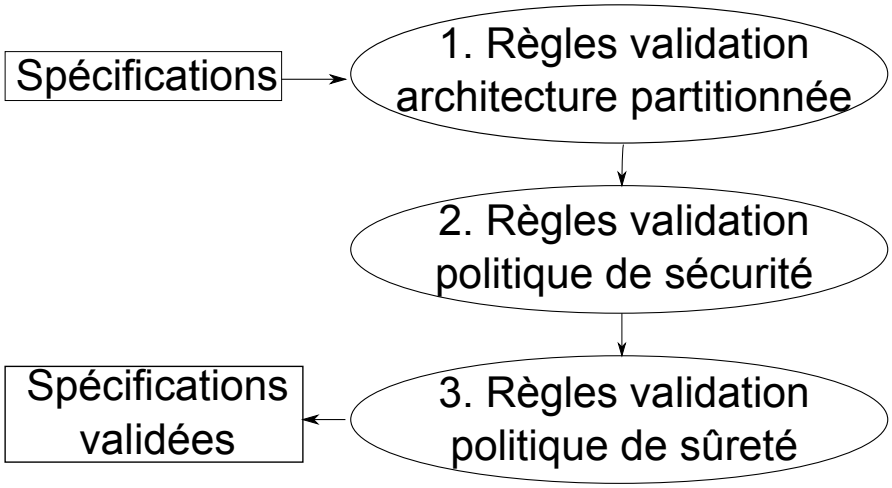


FIGURE 3.3 – Aspect validation de l'approche proposée

Règles de validation proposées

Afin de valider le système et assurer le respect des exigences de sécurité et de sûreté, des outils analysent les spécifications et détectent les potentielles erreurs de conception (figure 3.3).

Une première étape consiste à assurer que la spécification proposée par l'utilisateur respecte les patrons de modélisation proposés (première étape de la figure 3.3). Les règles de validation vérifient la bonne constitution du modèle et la cohérence des propriétés d'isolation spatiale et temporelle.

D'autres règles, spécifiques à la sécurité et à la sûreté (étapes 2 et 3 de la figure 3.3) sont appliquées. Celles dédiées à la sécurité (étape 2 de la figure 3.3) analysent les spécifications et vérifient le respect de politiques de sécurité comme *Bell-Lapadula*, *Biba* ou *MILS* (section 2.1.2). De plus, elles assurent la cohérence de la spécification des mécanismes de protection de données (utilisation d'algorithmes de chiffrement, cohérence des paramètres de configuration comme les clés de chiffrement, etc.).

Les règles spécifiques à la sûreté (étape 3 de la figure 3.3) assurent la bonne spécification de la politique de gestion des fautes en vérifiant que chacune d'elle est reprise lors de l'exécution. En outre, elles garantissent que toute faute pouvant survenir est reprise et associée à une procédure de recouvrement.

L'utilisation de cet ensemble de règle donne ainsi l'opportunité au concepteur d'application de détecter les erreurs de spécification de son système avant de procéder à son implantation.

Reprise de l'existant

La conception de ces règles de validation s'est appuyée sur plusieurs travaux :

- ARINC653 et MILS pour la validation de l'architecture partitionnée
- MILS et les politiques de sécurité (*Bell-Lapadula*, *Biba*, etc.) pour les aspects sécurité.
- ARINC653 et HAZOP/SHARD [72] pour les aspects sûreté.

3.2.3 Implantation automatique à partir des spécifications validées

Motivations

Les approches présentées précédemment (DO178B et Critères Communs) ainsi que les sections 3.1.2 et 3.1.3 ont montré le manque de lien entre les différentes étapes du développement. Elles ont montré que le développement de systèmes critiques devait être réalisé par un processus rigoureux assurant :

1. la bonne implantation des spécifications,
2. l'absence de code mort,
3. la prédictabilité du code,
4. l'absence d'erreur.

La composante humaine d'un développement introduisant des erreurs, de telles contraintes ne peuvent être satisfaites par des méthodes de production traditionnelles (sauf recours à de coûteux tests et relectures).

Pour ces raisons, l'implantation du système est automatiquement réalisée par un outil dédié. Ce dernier traduit les spécifications du concepteur en code, assurant ainsi la bonne traduction de ses exigences par le recours à des patrons de génération : ces derniers créent automatiquement le code de l'application en fonction des spécifications du concepteur d'application.

Définition d'un générateur automatique de systèmes partitionnés

Le processus d'implantation automatique est divisé en deux parties (figure 3.4) :

1. un outil de génération de code qui analyse et transforme les spécifications en code au moyen de patrons de génération.
2. une plate-forme d'exécution qui s'interface avec le code généré et fournit les services nécessaires (isolation/sécurité/sûreté).

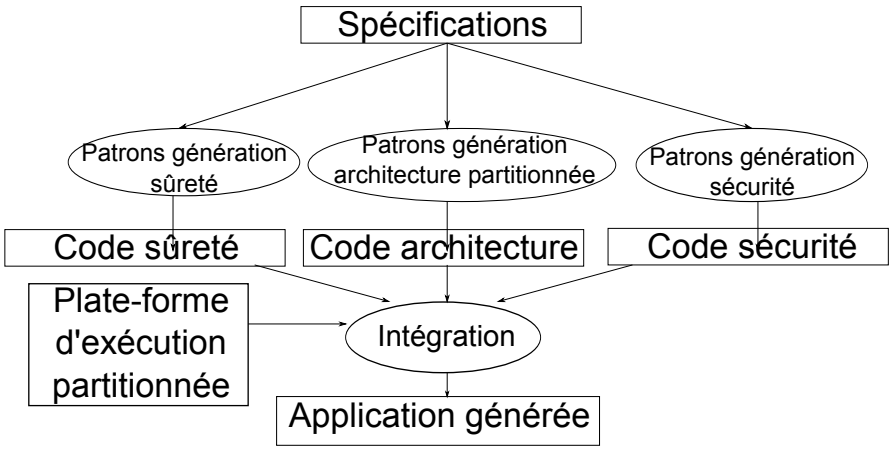


FIGURE 3.4 – Aspect génération de code de notre approche

Le processus de génération utilise les spécifications comme langage source et les traduit en code à l'aide de patrons de génération (première étape de la figure 3.4). Ces derniers sont prédéterminés et garantissent l'absence d'erreur, la bonne implantation des caractéristiques spécifiées et la prédictabilité du système.

La plate-forme d'exécution fournit les services requis par le code généré (seconde partie de la figure 3.4) : isolation entre partitions, chiffrement des communications, etc. Elle doit être hautement configurable afin de s'adapter aux besoins du système et n'inclure que les services nécessaires (suppression du code mort).

Reprise de l'existant

Les contraintes s'imposant à la génération de code ont été déduites à partir des standards de certification (section 2.3.1). Ils mettent un accent particulier sur la traçabilité des exigences et la conformité de l'implantation avec les spécifications. Le code produit doit être optimisé et réduire le code mort introduit, ce dernier impactant l'empreinte mémoire et le taux de couverture.

La conception de la plate-forme sécurisée s'inspire des systèmes d'exploitation partitionnés décrits par les approches ARINC653 et MILS. De nouveaux services (protocoles de chiffrement des communications) ont été ajoutés afin de supporter simultanément les contraintes de sécurité et de sûreté. Son architecture a été conçue pour être hautement configurable, et peut ainsi s'adapter au code généré.

3.2.4 Certification automatique du système généré

Définition de la certification dans notre contexte

Le mot certification a différentes significations suivant le contexte et le domaine dans lequel il est employé. En avionique (standard DO178B), le processus de certification implique par exemple la définition d'un jeu de test et de couverture de code pour chaque exigence. A l'inverse, dans le contexte spatial

(standards de la série ECSS-E-40 [41]), ces étapes de vérification sont nommées *« qualification »* .

Dans le contexte de cette thèse, nous regroupons derrière le terme de certification l'ensemble de solutions (analyse du système, politique de tests, etc.) offrant davantage d'assurance et de confiance dans l'implantation du système. Elles peuvent être réalisées par une analyse de l'application finale, des spécifications, une suite de tests, etc.

Motivations

Les standards de conceptions (DO178B, Critères Communs) requièrent que l'utilisateur apporte la preuve de la bonne implantation des spécifications. Notre approche doit donc intégrer ces contraintes et assurer :

1. le respect de la bonne implantation des spécifications du système et de ses caractéristiques.
2. la garantie des exigences des standards de certification, indépendemment de ses spécifications (par exemple, couverture de code).

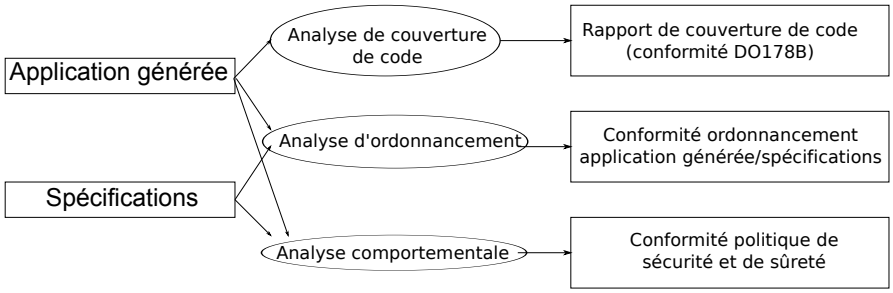


FIGURE 3.5 – Aspect certification de notre approche

Vers une certification automatique

Pour assurer la cohérence des applications produites, nous introduisons des outils (figure 3.5 qui analysent l'implantation générée, vérifient le respect des spécifications et le suivi des règles de développement des standards inhérents aux systèmes critiques. Cette étape est réalisée par la création d'outils dédiés qui supervisent l'exécution du système généré et analysent son comportement (code exécuté, évênements d'ordonnancement, context-switch, etc.). Ils comparent les résultats de cette analyse avec les spécifications du concepteur (modèle AADL) ou les contraintes inhérentes au standard de développement (ex : taux de couverture requis par la norme DO178B).

Ces outils s'appuient sur nos patrons de modélisation et extraient les informations pertinentes à analyser (par exemple : caractéristiques relatives à l'ordonnancement comme la période ou l'échéance d'une tâche). Ils récupèrent ensuite les informations correspondantes lors de l'exécution du système (par exemple, instant d'activation et temps d'exécution d'une tâche). Enfin, ils comparent ces deux données et vérifient leur cohérence (par exemple : vérification que la tâche respecte l'échéance spécifiée lors de son exécution).

Afin de satisfaire les contraintes de certification des standards comme la couverture de code, d'autres outils analysent le système lors de son exécution. Pour ce faire, un programme dédié supervise l'exécution de l'application et analyse ses caractéristiques ou son comportement. Par exemple, dans le cas de l'analyse de la couverture de code, un ensemble de programmes enregistre chaque instruction exécutée et produit un rapport de couverture.

L'aspect automatique de cette approche et la réutilisation des spécifications pour la certification évite des erreurs potentielles pouvant être réalisées au cours de cette étape. Enfin, l'utilisation d'un processus de développement cohérent, réutilisant la même spécification à chaque étape facilite la certification. A ce titre, la section 5.6 5) discute de l'intérêt de l'approche de génération de code pour la certification.

3.3 Langages de modélisation

Cette section présente les raisons qui nous ont amenées à représenter le système et ses contraintes au moyen d'un unique langage. Il présente les différents formalismes existants avec leurs avantages et leur limite. Puis, il motive le choix d'AADL et introduit ce langage de modélisation.

3.3.1 Motivations d'une représentation unifiée de l'architecture

Notre critique des solutions existantes (section 2.4) et l'identification de celles pertinentes dans la constitution de notre approche (section 3.1) a montré le besoin d'une représentation commune de l'architecture et de ses caractéristiques.

Il est nécessaire de s'appuyer sur un langage pour la définition des aspects logiciels et matériels. Mais surtout, ce langage doit être extensible afin de définir toutes les contraintes requises (dans notre cas : architecture partitionnée, niveaux de sécurité et de sûreté). De plus, afin d'être exploitable par des programmes et un ingénieur, il est préférable que le formalisme retenu propose une notation graphique (facilité d'utilisation pour l'ingénieur) et textuelle (exploitation par des programmes). Enfin, sa syntaxe doit être simple afin que l'utilisateur puisse facilement s'approprier les concepts de modélisation.

Les sections qui suivent présentent les langages de modélisation existants et motive notre choix pour celui utilisé au cours de nos travaux : AADL.

3.3.2 Langages existants

On dénombre trois principaux formalismes pour spécifier des systèmes critiques :

1. **MARTE** [84], extension au standard **UML** pour la description des exigences non-fonctionnelles. Ce langage est très connu de par l'héritage d'UML.
2. **SysML** [83], extension à UML pour la définition des aspects systèmes et de leurs contraintes.

3. **AADL** [98], langage dédié à la modélisation d'architecture. Il hérite du langage Meta-H créé par Honeywell il y a plusieurs années.

UML/Marte

MARTE [84] est un profil UML pour la définition des contraintes de temps et de performances d'un système. Il étend la sémantique d'UML et se place comme le successeur du profil *Schedulability, Performance and Time Specification* (SPT [80]). Il définit un ensemble de classes pour modéliser les ressources logicielles/matérielles d'un système, leurs associations et caractéristiques non-fonctionnelles.

L'utilisateur spécifie son système par l'intermédiaire de plusieurs diagrammes (cas d'utilisation, etc.), chacun d'eux exprimant les caractéristiques du système.

MARTE introduit également trois types d'analyse : l'analyse quantitative générique (pour valider la faisabilité des propriétés non-fonctionnelles), l'analyse de performance (vérification du temps de réponse, du comportement des composants, etc.) et l'analyse d'ordonnançabilité (assurant alors le respect des échéances dans le pire-cas d'exécution).

Il est actuellement utilisé en coordination avec plusieurs partenaires pour valider son utilisation dans un contexte industriel [38] . Des travaux ont été initiés afin de valider plusieurs caractéristiques comme l'ordonnancement en interfaçant le profil avec des outils d'analyse tel Cheddar [9].

Trois problèmes se posent dans l'utilisation de MARTE :

1. les multiples diagrammes, limitent leur utilisation par des outils de génération de code. Le manque de précision dans la sémantique (dû principalement à l'héritage d'UML) du formalisme (un même élément peut avoir plusieurs représentations) rend difficile l'analyse et l'exploitation du modèle.

2. l'outillage est encore peu fourni. Plusieurs outils supportent UML et potentiellement MARTE, mais peu d'entre eux fournissent des fonctionnalités d'analyse et/ou de génération de code.

   Certains outils provenant du monde UML (tels ceux supportant la validation de modèles à l'aide d'*Object Contraint Language* (OCL [81]) peuvent être utilisés. Toutefois, la complexité de l'interaction entre OCL et UML rend ces approches peu pratiques.

   De plus, pour être utilisés pour la production de systèmes critiques, des outils dédiés devront être développés. Par exemple, en terme de génération de code, les solutions actuelles ne sont pas adaptées aux applications temps-réel (introduction de code mort, génération de code limitée à des squelettes d'implantation, etc.).

3. la syntaxe du langage, la surcharge du meta-modèle et l'héritage d'UML font de MARTE une solution à la sémantique complexe et difficile d'usage.

Ces éléments constituent un obstacle à son utilisation.

SysML

SysML [83] est un langage dédié à la spécification, l'analyse et la validation d'architectures. Le besoin d'un nouveau langage est né du constat qu'UML était trop orienté vers les aspects logiciels. SysML propose de représenter également l'architecture matérielle avec ses caractéristiques.

SysML étend la sémantique d'UML en redéfinissant les types de diagrammes existants ou en introduisant de nouveaux (*requirements* pour spécifier les exigences et contraintes du système et *parametrics* pour exprimer les contraintes associées au paramètres et interfaces du système). L'utilisateur peut spécifier les composants logiciels et matériels du système, leurs interactions et leurs paramètres (par exemple, le système de *parametrics* définit les contraintes associées aux interfaces de communications). Enfin, le comportement des composants est représenté au moyen d'automates finis, décrivant leur aspect comportemental.

Tout comme UML, le langage est exclusivement graphique et repose sur plusieurs diagrammes pour représenter les différents aspects du système (cas d'utilisation, organisation des composants, comportement, etc.).

SysML se pose comme un complément à UML, l'un dédié les aspects architecturaux, l'autre à ceux logiciels. Utilisés conjointement, ils peuvent ainsi couvrir ces deux aspects de la construction d'un système.

Cependant, à l'inverse de MARTE, l'outillage supportant ce langage est fourni. De nombreux outils aussi bien commerciaux (ObjectGeode, Tau [15]) que libres (Papyrus) supportent SysML et proposent des fonctions de modélisation, validation ou génération de code. Une liste complète des outils est disponible en [108].

Toutefois, plusieurs inconvénients ont été identifiés dans l'utilisation de l'approche SysML :

- le manque de précision sémantique (un même élément peut être représenté sous différents noms, certains aspects ne sont pas standardisés, etc.) sont un frein à son utilisation.
- le recours à plusieurs diagrammes pour la spécification d'un même aspect du système complexifie l'exploitation des modèles par des outils.

Ces problèmes sont similaires à ceux qui se posent à l'utilisation de MARTE. Cette similitude provient de l'héritage d'UML et du manque de précision dans la sémantique de ce langage (premier problème soulevé dans l'analyse de MARTE et SysML).

AADL

Le langage *Architecture Analysis and Design Language* (AADL)[98] est un langage de modélisation d'architectures. Il dispose d'une syntaxe textuelle et graphique. Le langage permet à l'utilisateur de définir des composants logiciels et matériels qui s'agrègent suivant des règles prédéterminées (par exemple : un noeud d'un système réparti peut utiliser un bus).

Tout comme avec MARTE ou SysML, il est possible d'annoter les composants du modèle pour décrire leurs caractéristiques (par exemple : contraintes temporelles ou d'espace mémoire).

Le langage a été utilisé dans plusieurs projets académiques et industriels. En Europe, le projet ASSERT [61] l'a utilisé comme support pour la conception d'applications spatiales. Plus spécifique à la France, le projet Flex-eWare [36] emploie AADL pour la spécification d'applications réparties critiques. L'entreprise Thalès l'intègre dans la suite d'outils MyCCM-HI produite à l'issue de ce projet. Outre-Atlantique, aux États-Unis, le projet AVSI [25] utilise ce langage pour la spécification de systèmes avioniques.

AADL dispose de plusieurs suites d'outils pour modéliser (Eclipse/OSATE [105]), valider (Eclipse/OSATE, Ocarina/REAL [46, 54]) ou générer du code (Ocarina [116, 117]).

3.3.3 Motivation du choix d'AADL

Au cours de l'étude des langages de modélisation, il est apparu qu'AADL était celui répondant le mieux à nos contraintes (langage définissant architecture logicielle et matérielle, extensible, typage fort des composants, etc.).

D'abord, ce langage définit une sémantique précise avec des composants logiciels et matériels. Il définit clairement les interactions entre les composants par des mécanismes d'association ou de connection.

AADL évite les écueils sémantiques identifiés dans les autres langages empêchant les imprécisions dans la représentation des composants (une même caractéristique ne peut être définie de plusieurs manières). En particulier, il n'offre qu'une unique représentation rassemblant toutes les caractéristiques et attributs du système. En ce sens, il constitue une connexion à chaque aspect du système alors que les autres approches (MARTE ou SysML) utilisent des vues séparées ayant une sémantique parfois différente pour la représentation des mêmes concepts.

Les mécanismes d'extension du langage définissent les spécificités de chaque composant (comportement, erreurs, etc.). Ces informations additionnelles peuvent être réutilisées au cours de l'exploitation du modèle (validation, implantation ou certification).

Enfin, le double format de représentation (graphique, textuel) facilite l'utilisation d'AADL. La forme graphique est adaptée pour l'utilisateur final, la textuelle est plus appropriée pour une exploitation par des outils d'analyse.

Importance de la spécification système

Il est important de ne pas négliger les autres approches, celles-ci pouvant avoir des bénéfices pour d'autres aspects du développement. Si AADL apparaît ici comme le langage de modélisation adéquat, les autres solutions présentent aussi des caractéristiques intéressantes.

En particulier, SysML spécifie plus finement le comportement du système par l'utilisation d'automates. Bien que cela ne soit pas utile dans l'approche proposée, cet aspect du langage peut à l'inverse être particulièrement intéressant dans les phases préliminaires du développement (communication entre équipes de conception, etc.). Des travaux sont en cours pour intégrer ces aspects à AADL (via l'association d'un langage d'un langage tiers aux composants, l'annexe comportementale [51]).

3.3.4 Présentation du langage AADL

AADL est un langage de modélisation d'architecture définissant les composants matériels et logiciels du système, leurs caractéristiques (nommées propriétés) et leurs interactions (connections, associations). Nous présentons d'abord le coeur du langage et décrivons chaque type de composant.

Composants

AADL définit les composants logiciels suivants :

- le composant **data** représente un type de donnée, un emplacement en mémoire (mémoire partagée) ou une simple variable.
- le composant **process** modélise un espace d'adressage séparé dans lequel s'exécutent plusieurs tâches. Ce composant peut être interprété comme un *processus* UNIX.
- le composant **thread** modélise un fil d'exécution qui exécute du code applicatif. Sa sémantique est similaire à celles des threads *POSIX*.
- le composant **subprogram** représente une séquence de code exécutable par une tâche (composant **thread)**. Il peut être une référence à du code traditionnel (code *C* ou *Ada*) ou à un modèle applicatif (*Scade*, *Simulink*, etc.).

AADL définit les composants matériels suivants :

- le composant **device** modélise un périphérique (clavier, souris, etc.) ou du matériel spécifique (capteur, moteur, etc.). Il représente l'élément comme une boîte noire et ne spécifie pas le contenu de son implantation. Cette spécificité peut être ajoutée par la suite à l'aide d'une propriété.
- le composant **processor** représente un micro-contrôleur et un système d'exploitation minimal (ordonnanceur, pilotes de périphériques, etc.). Un processeur peut contenir des **virtual processor** et accéder à un **bus**.
- le composant **virtual processor** spécifie un processeur *« virtuel »* : un ordonnanceur ou une machine virtuelle. Ce composant est utilisé pour diviser un processeur physique (cas de la virtualisation ou du partitionnement) ou spécifier ces particularité d'implantation (processeur contenant plusieurs coeurs).
- le composant **bus** fournit l'accès entre les composants **processor**, **device** et **memory**. Les connections entre composants peuvent être associées à un **bus**, décrivant ainsi la stratégie de déploiement d'un système réparti.
- le composant **virtual bus** représente une partie d'un bus ou un protocole. Ce composant peut être utilisé pour segmenter un bus (spécification des propriétés de qualité de service d'un bus) ou décrire les fonctions d'un protocole.
- le composant **memory** modélise une mémoire physique (disque dur, RAM, ROM, etc.). Ces composants sont utilisés pour stocker des données et peuvent être utilisés par des **process** ou des **thread**.

Enfin, le langage définit également un dernier type de composant (appelé **system**), servant à agréger composants matériels et logiciels.

Règles de composition

Une spécification complète décrit l'ensemble de l'application, avec ses composants matériels et logiciels. Les composants peuvent contenir d'autres composants, appelés *sous-composants*. Au niveau le plus haut, un composant de type `system` constitue la racine du modèle. Cette dernière représente la base du système, composé la plupart du temps de composants `process`, `processor` et `memory`).

Le standard précise les règles sémantiques spécifiant les agrégations de composants autorisées. Ces règles découlent de la logique de composition de l'application : ainsi, il est interdit pour un composant `data` de contenir un `process` ou un `processor` car cela ne correspond à aucun assemblage possible dans des cas pratiques.

Les règles de composition sont résumées dans le tableau 3.2.

Composant	**Peut contenir**
system	system, processor, virtual processor, memory, bus, virtual bus, device, process, data, subprogram, subprogram group, abstract
abstract	data, subprogram, subprogram group, thread, thread group, process, virtual processor, memory, bus, virtual bus, device, system, abstract
data	data, subprogram
processor	virtual processor, memory, virtual bus, abstract
virtual processor	abstract, virtual processor, virtual bus
memory	memory, bus, abstract
bus	virtual bus, abstract
virtual bus	virtual bus, abstract
device	bus, virtual bus, abstract
subprogram	data, abstract
subprogram group	subprogram, abstract
thread	data, subprogram group, subprogram, abstract
thread group	data, subprogram group, subprogram, abstract, thread, thread group
process	data, subprogram, subprogram group, thread, thread group, abstract

TABLE 3.2 – Règles de composition des composants AADL

Interfaces et connections

Un composant peut définir des mécanismes pour s'interfacer et échanger des données avec les autres éléments de l'architecture (paramètre d'un sous-programme, port de communication, etc.). La communication entre composants est alors établie en connectant ces interfaces.

Une interface est une caractéristique visible entre les composants. Ce sont des entités nommées de l'architecture qui permettent d'échanger des données ou envoyer des signaux. L'échange de données est réalisée grâce aux ports de type données (`data`) et aux paramètres des composants `subprogram`. L'envoi et la réception de signaux se fait grâce aux ports de type évènement (`event`).

Le listing 3.1 illustre l'utilisation des interfaces et des connections en AADL. Un composant `process` (`Transmitter_Process.Impl`) contient un composant

`thread` (`Transmitter_Thread.Impl`) qui appelle un sous-programme `Transmit`. Le port d'entrée du composant `process` est connecté à celui du composant `thread` qui transmet la donnée au sous-programme. De la même manière, la valeur renvoyée par le sous-programme est transmise au composant `thread` qui la retransmet au `process`.

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

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

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

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

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

Listing 3.1 – Exemple d'utilisation des interfaces et des connections AADL

Extensions possibles au langage

Le langage définit deux mécanismes d'extension pour enrichir un modèle et ajouter des informations non prises en charge par le coeur du langage : les *annexes* et les *propriétés*.

Annexes Les annexes associent un langage tiers à la description d'un composant. Cette fonctionnalité permet d'interfacer un modèle AADL avec d'autres outils pouvant exploiter ces déclarations.

```
data Sample end Sample;

thread Collect_Samples
features
  Input_Sample   : in data port  Sample;
  Output_Average : out data port Sample;
annex OCL {
```

```
  pre    : 0 < Input_Sample < maxValue;
  post   : 0 < Output_Sample < maxValue;
};
end Collect_Samples ;
```

Listing 3.2 – Exemple d'utilisation d'un langage d'annexe dans un composant AADL

Le listing 3.2 donne un exemple de composant utilisant un langage d'annexe. Ici, OCL [81] est associé au `thread` et définit une pré-condition sur la valeur associée au port d'entrée et une post-condition sur celui de sortie.

Plusieurs langages d'annexes sont déjà disponibles et complètent la description d'une architecture. Les plus utilisés sont :

- **l'annexe comportementale** (ou *Behavioral annex* [51]) qui décrit le comportement des composants.
- **l'annexe de modélisation des erreurs** (ou *error model annex* [95]) qui spécifie les erreurs (et leur gestion) de chaque composant.

Propriétés Les propriétés AADL sont des annotations associées aux différentes entités du modèle. Ce sont des attributs typés qui leur ajoutent une information. Cette dernière peut être réutilisée par les outils d'analyse, de validation ou de génération de code.

La plupart du temps, l'information rajoutée spécifie une contrainte, une exigence ou une caractéristique du système (par exemple : la capacité de calcul, le niveau de sécurité, etc.). Le standard AADL définit un ensemble de propriétés *« standards »* , mais l'utilisateur peut également définir ses propres propriétés.

```
subprogram Transmit
features
  Input  : in  parameter Message ;
  Output : out parameter Message ;
properties
  Source_Language => Ada95 ;
  Source_Name     => "Transmitter.Do_Transmit";
end Transmit;

thread implementation Transmitter_Thread.Impl
calls
  { Do_Transmit : subprogram Transmit;};
connections
  parameter Input                -> Do_Transmit.Input;
  parameter Do_Transmit.Output -> Output;
properties
  Dispatch_Protocol => Sporadic;
  Period            => 20 ms;
end Transmitter_Thread.Impl;
```

Listing 3.3 – Exemple d'utilisation des propriétés dans un composant AADL

Le listing 3.3 montre l'exemple de deux composants AADL enrichis par

l'ajout de propriétés. Par l'intermédiaire de ces propriétés, le concepteur a spécifié le langage d'implantation (*Ada95*) et le nom de la procédure correspondant au composant `subprogram Transmit`. Il a également défini les propriétés d'ordonnancement du `thread Transmitter_Thread` via l'association des propriétés `Dispatch_Protocol` et `Period`.

3.3.5 AADL et outils de validation

Cette section propose un survol des outils supportant AADL en présentant les deux plus utilisés : OSATE et Ocarina.

OSATE

OSATE (Open Source AADL Toolkit Environment) est un environnement de modélisation intégré à l'Eclipse Modeling Framework (EMF). Cet outil se présente sous la forme d'un plug-in Java s'exécutant au sein d'Eclipse et se pose comme l'outil de référence. Il est par ailleurs intégré dans la suite d'outils de développement de systèmes critiques TOPCASED. La figure 3.6 présente une capture d'écran de l'outil.

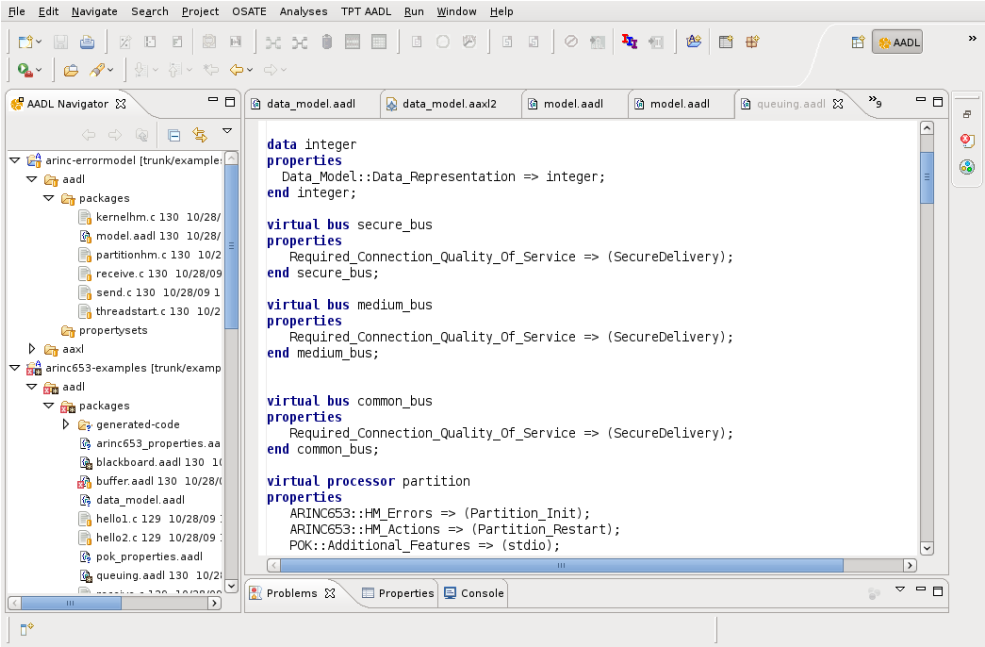


FIGURE 3.6 – Capture d'écran de l'outil OSATE

Au delà de la simple modélisation, OSATE exporte un ensemble de fonctionnalités pour la lecture et la modification de modèles AADL. Ces mécanismes, implantés en Java, intègrent une API permettent à l'utilisateur de concevoir ses propres applications utilisant les modèles AADL.

Nos travaux nous ont amené à produire plusieurs *plug-ins* à OSATE. Ces derniers se concentrent principalement sur la validation des aspects relatifs à la sûreté d'une architecture (travaux relatifs à l'annexe ARINC653 [34]).

Ocarina

Ocarina [118] est une suite d'outils pour l'analyse, la validation et la génération de systèmes à partir de modèles AADL. Il est disponible sous licence libre et fonctionne sur de nombreuses plates-formes (*Windows*, *Linux*, *Darwin*, *FreeBSD*, *Solaris*).

Les fonctions de validation d'Ocarina reposent sur *Requirements Enforcement Analysis Language* (REAL [54]), un langage de vérification de contraintes dédié à AADL. Développé à TELECOM PARISTECH, au sein de l'équipe de recherche dans laquelle les présents travaux ont pris naissance, il définit une syntaxe et une sémantique adaptée à AADL.

REAL se base sur la définition de théorèmes. Un théorème exprime les contraintes qu'un composant doit vérifier. Un exemple de théorème est présenté dans le listing 3.4.

Ce théorème spécifie que pour chaque bus du modèle (ligne 2, `foreach e in bus_set`), les connections qui lui sont associées et les données qu'il transporte sont analysées. Il vérifie ensuite que la taille de la somme des données qu'il transporte est inférieur à la bande passante du bus (ligne 9). Ainsi, il est possible d'assurer par analyse du modèle que l'envoi des données ne surcharge pas le bus.

Afin de vérifier automatiquement les spécifications d'une architecture, un ensemble de théorèmes a été intégré à la chaîne d'outils que nous avons produite au cours de nos travaux. Ocarina est ainsi invoqué systématiquement sur chaque modèle AADL pour valider la cohérence des spécifications (respect des patrons de modélisation proposés) et les exigences de sécurité/sûreté.

```
1  theorem Buses_Rate
2  foreach e in bus_set do
3    Cnx_Set(e) := {x in connection_set |
4                   is_bound_to (x, e)};
5    Connected_Data_Set := {x in data_set |
6                           is_accessed_by (x, Cnx_Set)};
7
8    check (get_property_value
9           (e, "ASSERT_Properties::Access_Bandwidth") >=
10          sum (set_property_value
11             (Connected_Data_Set, "source_data_size")));
12 end;
```

Listing 3.4 – Exemple de théorème REAL

Ocarina propose également des fonctionnalités de génération de code à partir de modèles AADL. Capable de générer l'implantation d'architectures réparties à partir des spécifications, il s'interface avec plusieurs langages utilisés pour la construction de systèmes critiques (tels *Ada* ou *C*).

Ces fonctionnalités ont été le fruit de précédents travaux de recherche [116, 112, 36] et ont été évalués lors de plusieurs projets aussi bien européens (ASSERT) que nationaux (Flex-eWare, MOSIC et prochainement, PARSEC). En outre, il a été montré [116] que le recours à une génération de code à partir des spécifications supprimait tout code superflu et limitait ainsi le surplus (*overhead*) de mémoire introduit traditionnellement par les approches de génération de code. Par exemple, au cours du projet national Flex-eWare, il a été montré que le recours à la génération de code à partir d'AADL produisait des applications dont la taille était 500 fois inférieure à celles générées par des approches à composants de type CCM [82, 116]. Ceci permet de limiter l'empreinte mémoire des systèmes, élément particulièrement critique dans le domaine de l'embarqué.

Synthèse de ce chapitre

Ce chapitre énumère les solutions reprises dans la constitution de notre approche et rappelle leurs limites :

1. l'absence de plate-forme d'exécution commune pour la sûreté et la sécurité ;
2. le manque de cohérence et l'éparpillement des spécifications dans plusieurs documents utilisant des formalismes différents ;
3. l'absence de relation entre les spécifications validation validées et l'implantation ;
4. le manque d'automatisation du cycle de développement, augmentant les coûts (temporels, financiers) de développement.

A partir de ces éléments, nous présentons notre approche. Celle-ci s'articule autour de quatre principaux points correspondant à nos objectifs initiaux (section 1.4) :

1. **la représentation de l'architecture** (première partie de l'objectif 1) avec ses contraintes de sécurité et de sûreté au moyen d'un unique langage de modélisation (AADL). L'utilisation d'un seul langage permet de s'affranchir de multiples transformations de modèles, coûteuses et sources d'erreur.
2. **la validation automatique des spécifications** (seconde partie de l'objectif 1) vérifie que la spécification exprimée par le concepteur du système est exempte de défaut tant dans sa structure que dans ses caractéristiques.
3. **la génération automatique de l'implantation** (objectif 2) garantit que les spécifications sont respectées dans le système généré. En outre, ce processus créé le code à l'aide de patrons de génération assurant la prédictabilité du code et supprimant le facteur d'erreur humain.
4. **la certification automatique** (objectif 3) assure que les contraintes exprimées dans la spécification sont bien respectées dans l'application générée. Il vérifie également que les exigences de développement des standards (tels DO178B ou les Critères Communs) sont respectées.

Les deux premiers points sont discutés dans le chapitre 4 tandis que les deux derniers font l'objet du chapitre 5.

Chapitre 4

Spécification et validation de l'architecture

Sommaire du chapitre

4.1 Avant-propos : standardisation des patrons de modélisation 93

4.2 Modélisation de l'architecture partitionnée 94

4.2.1 Noyau 95

4.2.2 Partitions 96

4.2.3 Isolation spatiale 98

4.2.4 Isolation temporelle 100

4.2.5 Tâches 101

4.2.6 Communications intra-partition 103

4.2.7 Communications inter-partitions 106

4.3 Validation de l'architecture partitionnée 109

4.3.1 Conformité de la structure du modèle 109

4.3.2 Isolation temporelle 110

4.3.3 Isolation spatiale 111

4.3.4 Analyse d'ordonnancement 112

4.4 Modélisation des aspects sûreté 113

4.4.1 Niveau de criticité d'une partition 113

4.4.2 Représentation des fautes 114

4.4.3 Politiques de recouvrement 116

4.5 Validation des aspects sûreté 118

4.5.1 Conformité de la politique de sûreté 118

4.5.2 Couverture des fautes 120

4.5.3 Impact de la politique de recouvrement 121

4.6 Modélisation des aspects sécurité 124

4.6.1 Niveaux de sécurité 124

4.6.2 Mécanismes de chiffrement 126

4.7 Validation des aspects sécurité 129

4.7.1 Implantation des mécanismes de sécurité 129

4.7.2 Politique de sécurité *Bell-Lapadula* 131

4.7.3 Politique de sécurité *Biba* 132

4.7.4 Conformité MILS 132

Rappels

Dans le chapitre précédent, nous avons identifié les solutions pertinentes pour l'assurance de sécurité et de sûreté et nous avons présenté notre approche. Celle-ci s'appuie sur deux parties : l'une de **spécification et validation** (premier objectif de nos travaux), l'autre, **d'implantation et de certification** (second et troisième objectifs). Ce chapitre du manuscrit se concentre sur le premier objectif : **la spécification et la validation de l'architecture**.

Objectifs du chapitre

Ce chapitre décrit la spécification et la validation des systèmes partitionnés. Dans un premier temps, il introduit les patrons de modélisation dédiés à la représentation d'architectures sûres et sécurisées. Dans un second temps, il décrit les règles de validation de la cohérence du système et de ses caractéristiques de sécurité et de sûreté. Cette vérification réalisée sur les spécifications assurent que l'utilisateur n'a pas introduit d'erreur lors de la conception, évitant ainsi leur propagation lors des phases suivantes du développement.

4.1 Avant-propos : standardisation des patrons de modélisation

Le présent chapitre introduit les règles de spécification et de validation pour les architectures partitionnées avec leurs caractéristiques de sécurité et sûreté, premier objectif de nos travaux (section 1.4).

Ces règles de représentation ont fait l'objet d'une collaboration avec les membres du comité de standardisation AADL. A son issue, un document annexe au standard (annexe de modélisation ARINC653 pour le langage AADL [34]) a été produit dans le but d'y intégrer nos règles de modélisation. Ces travaux proposent un ensemble de règles (utilisation des composants, interactions possibles, etc.) au concepteur d'application pour décrire et représenter les architectures partitionnées.

[34] est le fruit de dix-huit mois de collaboration avec les différents acteurs de la communauté AADL. Au moment de la rédaction de ce manuscrit, il est en cours de finalisation et devrait être publié par le *Society of Automotive Engineers* (SAE International, organisme chargé de la publication du standard AADL) fin 2010.

Exemple de patron de modélisation ($\mathrm{PM}^{nom-patron}$)	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Exigence 1 (E1)	Composant de type `type_composant` associé au composant de type `type_composant2`.
Exigence 2 (E2)	Propriété standard `Exemple_De_Propriété`
Exigence 3 (E3)	*Nouvelle propriété* *`Propriété_Proposée`* – `Nouvelle_Propriété : type_propriété applies to (type_composant`

Table 4.1 – Exemple de patron de modélisation

Convention de représentation des patrons

Au cours de ce chapitre, nous introduisons les patrons de modélisation conçus pour représenter chaque élément de l'architecture partitionnée. Ils reposent sur l'utilisation de composants et de propriétés du standard AADL. Lorsque sa sémantique ne permettait d'exprimer certaines caractéristiques du système (par exemple, isolation temporelle), nous l'avons étendue par l'introduction de nouvelles propriétés.

Chaque patron est décrit par un tableau explicitant les associations de composants et leurs propriétés en rapport avec les exigences des architectures partitionnées. Le tableau 4.1 donne un aperçu de cette représentation : la colonne de gauche énumère les exigences des architectures partitionnées, celle de droite décrit sa correspondance en AADL. Lorsque le patron requiert l'introduction d'une nouvelle propriété, celle-ci est écrite en *italique* avec sa définition. Dans tableau d'exemple 4.1, la première exigence (E1) est transcrite

en AADL par une association de composant, la seconde (E2) par l'association d'une propriété standard et la troisième (E3) par l'association d'une nouvelle propriété (texte en italique et ajout de la définition de la propriété).

Un identifiant est associé à chaque patron ($PM^{nom-patron}$ pour **P**atron de **M**odélisation), permettant de les référencer dans la suite du manuscrit.

Enfin, pour chaque patron, un exemple (textuel et graphique) illustre sa mise en œuvre. L'exemple graphique est omis lorsque le patron décrit uniquement l'ajout de propriété : le standard AADL ne spécifiant pas leur description sur la notation graphique, les représenter est donc impossible.

Validation

Ce chapitre présente également les règles de validation de modèles. Elles assurent la conformité de la spécification par l'analyse des mécanismes de sécurité/sûreté, veillant à ce qu'aucune erreur n'ait été introduite par le concepteur du système.

Au cours d'expérimentations menées, nous avons implanté les règles de validation au sein d'OSATE et Ocarina (section 3.3.4). L'implantation dans deux outils différents montrent l'indépendance des règles par rapport à la méthode de manipulation du modèle (API Java dans le cas d'OSATE, langage de contraintes pour Ocarina).

Pour chaque règle de validation, nous présentons le pseudo-code de leur implantation. Leur mise en œuvre au sein d'Ocarina avec le langage REAL est disponible en annexe de ce manuscrit. Celle d'OSATE est disponible via le *plug-in* Eclipse pour Ocarina téléchargeable sur la page officielle d'Ocarina [109].

4.2 Modélisation de l'architecture partitionnée

Cette section présente les patrons de modélisation pour la spécification d'architectures partitionnées. Ils ont été conçus dans une optique de généricité : il est possible de représenter plusieurs types d'architectures partitionnées. Ils peuvent être utilisés aussi bien dans un but de sûreté que de sécurité et sont également adaptables à d'autres exigences (analyse d'ordonnançabilité, de la latence, etc.).

Les règles proposées définissent la sémantique des composants AADL pour la représentation d'architectures partitionnées : elles explicitent quelle partie de l'architecture est modélisée par chaque composant. Si le type de composant n'est pas redéfini, la sémantique originale du standard s'applique. Le cœur d'AADL ne permettant pas d'exprimer certaines caractéristiques des architectures partitionnées, de nouvelles propriétés sont introduites dans ce but (description de l'isolation spatiale, temporelle, etc.).

La base de l'architecture partitionnée (noyau d'isolation et partitions) est représentée par les composants `processor` (noyau), `virtual processor` (plate-forme d'exécution d'une partition) et `process` (contenu d'une partition). Les sections suivantes détaillent les patrons de modélisation de chacun d'eux, leurs interactions avec les autres entités du standard AADL et les propriétés introduites.

4.2.1 Noyau

Le noyau est spécifié avec le composant AADL `processor` (figure 4.1 et listing 4.1), le patron de modélisation est décrit dans le tableau 4.2. Servant de support d'exécution aux partitions (exigence E1 du patron 4.2), le patron de modélisation requiert l'ajout de sous-composants `virtual processor` (plate-forme d'exécution d'une partition) au `processor`. Des propriétés dédiées spécifient la politique d'isolation temporelle (tranches de temps allouées à chaque partition, exigence E2 décrit dans 4.2).

Un exemple de ce patron est illustré dans le listing 4.1. Le noyau représenté ici contient deux partitions, alloue 100ms pour l'exécution de la première et 200ms pour la seconde. La *Major_Frame* est définie à 300ms.

Noyau de Partitionnement (PM^{noyau})	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Contrôle l'exécution des partitions (E1)	Composant `processor` contenant l'environnement des partitions (`virtual processor`) en tant que sous-composant.
Ordonnancement fixe et prédéterminé des partitions (E2)	• *Propriété `Slots` spécifiant une liste de tranches de temps* – `Slots : list of time applies to (processor)`. • *Propriété `Slots_Allocation` associant chaque tranche de temps aux partitions* – `Slots_Allocation : list of reference (virtual processor) applies to (processor)`.
Définition de la *Major Frame*, instant de réalisation des communication (E3)	• *Propriété `Major_Frame`* – `Major_Frame : time applies to (processor)`.

TABLE 4.2 – Patron de modélisation du noyau partitionné

```
processor noyau
end noyau;

processor implementation noyau.i
subcomponents
   p1 : virtual processor environnement_partition.i;
   p2 : virtual processor environnement_partition.i;
properties
   Slots            => (100ms, 200ms);
   Slots_Allocation => (reference (p1), reference (p2));
   Major_Frame      => 300ms;
end noyau.i;
```

Listing 4.1 – Exemple d'un composant processor modélisant un noyau de partitionnement

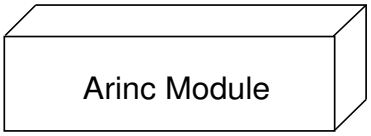


FIGURE 4.1 – Représentation graphique du noyau d'un système partitionné

4.2.2 Partitions

Le patron proposé (tableau 4.3) utilise deux composants pour représenter une partition, chacun représentant une partie de ses exigences :

1. un ensemble ressources (tâches, canaux de communication, etc. - exigence E1 de 4.3).
2. un environnement d'exécution (ordonnancement de ses tâches, fonctions accessibles au code applicatif, etc. - exigence E2 de 4.3)

Partition ($PM^{partition}$)	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Contient les ressources d'exécution de l'application dans un espace d'adressage dédié (E1)	Composant AADL `process` associé à des interfaces (`ports` AADL) et des sous-composants `thread` ou `data`.
Exécution dans un environnement d'exécution confiné (E2).	Composant AADL `virtual processor` associé au `process` via la propriété standard `Actual_Processor_Binding`
Ordonnancement des tâches par un algorithme d'ordonnancement spécifique à chaque partition(E3).	Propriété standard `Scheduling_Protocol` associée au composant `virtual processor`
Spécification des exigences mémoires (E4)	Description de la taille requise pour le code et les données via les propriétés standards `Source_Code_Size` et `Source_Data_Size` au composant `process`
Apport des fonctions nécessaires aux composants applicatifs (E5)	*Énumération des fonctionnalités de l'environnement via l'ajout de la propriété `Additional_Features` au composant `virtual processor`* – `Additional_Features : list of Supported_Additional_Features applies to (virtual processor)`.

TABLE 4.3 – Patron de modélisation d'une partition

Description du patron

L'environnement d'exécution est modélisé par un composant AADL `virtual processor`. Il est contenu au sein d'un composant `processor` (module/noyau exécutant la partition) et décrit ses caractéristiques par l'association de propriétés.

Les ressources de la partition sont contenues dans un composant AADL `process`. Il spécifie les tâches (composants `thread`), les communications (`ports` et `data`) utilisées par la partition avec leurs caractéristiques (taille de la pile des tâches, taille des canaux de communication, etc.).

Ce patron requiert (exigence E2) que les deux composants (`virtual-processor` et `process`) représentant une partition soient associés via la propriété `Actual_Processor_Binding`, réunissant ainsi les deux aspects de la partition (plate-forme d'exécution et contenu).

La politique d'ordonnancement interne à la partition (exigence E3 du tableau 4.3), les exigences mémoire (exigence E4) ou la liste des fonctions proposées à la couche applicative (exigence E5) sont spécifiées par l'association de propriétés aux composants modélisant la partition (`process` ou `virtual processor`).

Représentation graphique

La figure 4.2 donne un exemple de représentation graphique du patron : le composant `process`, qui modélise le contenu applicatif de la partition (tâches, données, etc.), est associé (flèche en pointillée) à un `virtual processor`, environnement assurant leur exécution.

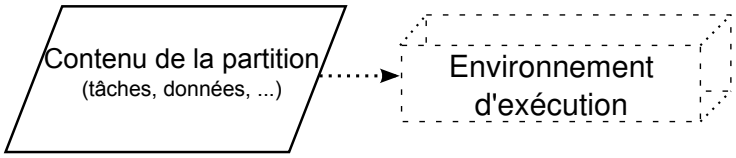


FIGURE 4.2 – Représentation graphique d'une partition

Exemple

Le listing 4.2 met en œuvre ce patron de modélisation. Le modèle décrit un composant représentant l'environnement d'exécution de la partition (`environnement_partition`). Celui-ci est contenu dans un composant de type `processor` comme l'exige le patron de modélisation précédemment présenté (section 4.2.2). Celui-ci spécifie les caractéristiques/contraintes d'exécution de la partition : politique d'ordonnancement (exigence E3, ici, l'algorithme *Round-Robin* est utilisé) via la propriété `Scheduling_Protocol`, fonctions applicatives disponibles (exigence E5, les environnements modélisés supportent les bibliothèques d'entrées/sorties standard (`libc_stdio`) et mathématique (`libmath`) comme le décrit la propriété `Additional_Features`).

Les ressources et interfaces de la partition (exigence E1) sont représentées par le composant `contenu_partition` (contenant ici une donnée, une tâche et deux interfaces). La description de leurs contraintes mémoires (exigence E4) est décrite par l'association des propriétés standards `Source_Code_Size` et `Source_Data_Size` au composant `contenu_partition`.

Enfin, cet ensemble de ressources (composant `contenu_partition`) est associé à son environnement d'exécution (exigence E2) par la définition de la propriété `Actual_Processor_Binding`.

```
virtual processor environnement_partition
end environnement_partition;

virtual processor environnement_partition.i
properties
   Scheduling_Protocol => Round_Robin;
   Additional_Features => (libc_stdio, libmath);
end environnement_partition.i;

process contenu_partition
features
   interface_entree : in data port montype;
   interface_sortie : out event data port montype;
end contenu_partition;

process implementation contenu_partition.i
subcomponents
   tache  : thread type_thread.i;
   donnee : data type_data.i;
properties
   Source_Code_Size => 40 Kbyte;
   Source_Data_Size => 20 Kbyte;
end contenu_partition.i;

processor noyau_partionnement
end noyau_partitionnement;

processor implementation noyau_partitionnement.i
subcomponents
   environnement1 : virtual processor environnement_partition.i;
   environnement2 : virtual processor environnement_partition.i;
end noyau_partitionnement.i;

system architecture_partitionnee
end architecture_partitionnee;

system implementation architecture_partitionnee.i
subcomponents
   partition1 : process contenu_partition.i;
   partition2 : process contenu_partition.i;
   noyau      : processor noyau_partitionnement.i;
properties
   Actual_Processor_Binding => (reference (noyau.environnement1))
            applies to partition1;
   Actual_Processor_Binding => (reference (noyau.environnement2))
            applies to partition2;
end architecture_partitionnee.i;
```

Listing 4.2 – Exemple de modèle utilisant le patron de modélisation des partitions

4.2.3 Isolation spatiale

L'isolation spatiale consiste à dédier une partie de la mémoire du système à une partition pour le stockage de son code et de ses données. Le composant `process` modélise un espace d'adressage contenant les ressources nécessaires à l'exécution d'un programme sans pour autant définir ses caractéristiques de déploiement (emplacement mémoire, taille, etc.). Il est donc nécessaire de spécifier l'association entre l'espace d'adressage (composant `process`) et sa localisation au sein de la mémoire du système (composant `memory`).

Isolation spatiale (PM$^{isolation-spatiale}$)	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Division de la mémoire en segments (E1)	Composant `memory` contenant des sous-composants `memory` représentant chaque segment.
Spécification de la taille des segments (E2)	Définition de la propriété `Byte_Count` sur chaque segment mémoire (composant `memory`).
Chaque partition est associée à un segment (E3)	Association du composant `process` de la partition avec un composant `memory` via la propriété `Actual_Memory_Binding`.

TABLE 4.4 – Patron de modélisation de l'isolation spatiale

Description

Le patron de modélisation de l'isolation spatiale (tableau 4.4) définit la méthode de représentation de la mémoire du système, sa division en segments et le déploiement des espaces d'adressages. Il requiert que la mémoire centrale de la plate-forme soit spécifiée par un composant `memory` racine qui représente la mémoire centrale du système (RAM). Ce composant est divisé en plusieurs sous-composants `memory` (exigence E1 du tableau 4.4), chacun représentant un segment mémoire. Les caractéristiques de chaque segment (taille, permission d'exécution, etc. exigence E2 du tableau 4.4) sont décrites à l'aide de propriétés AADL.

Le patron requiert que chaque segment soit associé (exigence E3 du tableau 4.4) à une partition (composant `process`). Cela est réalisé *via* via la propriété AADL standard `Actual_Memory_Binding`.

Ce patron de modélisation explicite ainsi la stratégie d'isolation spatiale définissant les caractéristiques de la mémoire et l'association partitions/segments. Cette définition sera par ailleurs utilisée pour valider (section 4.3.3) ou implanter (section 5.3.1) cette stratégie.

Représentation graphique

La représentation graphique du patron est illustrée en figure 4.3 : le composant mémoire est divisé en plusieurs sous-composants, chacun représentant un segment. Ces derniers sont associés aux partitions (flèche en pointillés).

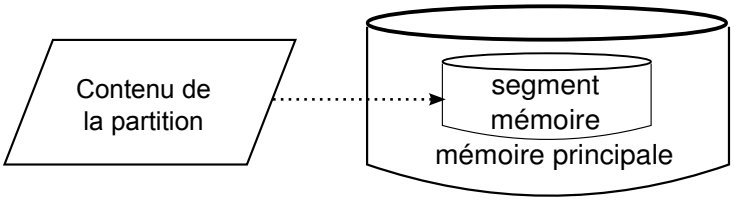


FIGURE 4.3 – Représentation graphique de l'isolation spatiale

Exemple

Le listing 4.3 illustre l'utilisation de ce patron. Le composant `mémoire_principale` décrit sa décomposition en segments (exigence E1) par la définition de sous-composants de type `memory`. Deux segments sont spécifiés : `segment.petit` et `segment.grand`. Chacun définit sa taille (exigence E2) via l'association de la propriété standard `Byte_Count`. Enfin, le déploiement partitions/segments est modélisé par l'ajout de la propriété `Actual_Memory_Binding` aux composants de type `process`.

```
process implementation partition.i
...
end partition.i;

memory segment
end segment;

memory implementation segment.petit
properties
   Byte_Count => 10;
end segment.petit;

memory implementation segment.grand
properties
   Byte_Count => 20000;
end segment.grand;

memory memoire_principale
end memoire principale;

memory memoire_principale.i
subcomponents
   segment1 : memory segment.petit;
   segment2 : memory segment.grand;
end memoire_principale.i;

system implementation systeme_partitionne.i
subcomponents
   partition1 : process partition.i;
   partition2 : process partition.i;
   memoire    : memory memoire_principale.i;
properties
   Actual_Memory_Binding => (reference (memoire.segment1))
         applies to partition1;
   Actual_Memory_Binding => (reference (memoire.segment2))
         applies to partition2;
end systeme_partitionne.i;
```

Listing 4.3 – Exemple de spécification de l'isolation spatiale

4.2.4 Isolation temporelle

L'isolation temporelle consiste à exécuter les partitions suivant des tranches de temps fixes et prédéfinies. C'est le noyau du système (représenté par le composant `processor`) qui assure le respect de cette caractéristique du système.

Les caractéristiques relatives à l'isolation temporelle sont donc définies par le patron de modélisation du noyau de partitionnement (section 4.2.1). En par-

ticulier, il spécifie cette politique par l'association de deux nouvelles propriétés au composants de type `processor` : `Slots` et `Slots_Allocation`.

Politique d'ordonnancement au sein des partitions

Chaque partition utilise sa propre politique d'ordonnancement de tâches. L'algorithme utilisé est spécifié par le patron de modélisation des partitions (section 4.2.2) en associant la propriété standard `Scheduling_Protocol` à l'environnement d'exécution de la partition (composant `virtual processor`).

4.2.5 Tâches

Les partitions contiennent des tâches exécutant le code applicatif du système. Un patron de modélisation (tableau 4.5) décrit comment les tâches d'un système partitionné sont représentées en AADL.

Tâche (PM^{tache})	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Exécution de composants applicatifs au sein d'une partition (E1)	Composant `thread` contenu dans un composant `process` et exécutant des `subprogram`.
Spécification des contraintes temporelles d'exécution (période, échéance, etc.) (E2)	Association des propriétés standards suivantes au composant `thread` : `Period`, `Deadline`, `Priority` et `Compute_Execution_Time`.
Déclaration des contraintes mémoire (taille du code, des données, de la pile) (E3)	Définition des propriétés standards `Source_Data_Size`, `Source_Code_Size` et `Source_Stack_Size`.

Table 4.5 – Patron de modélisation d'une tâche

Description du patron de modélisation

Le patron associé aux tâches (tableau 4.5) les représente à l'aide des composants de type `thread`. Ceux-ci exécutent le code applicatif (composants `subprogram`) et sont contenus dans un composant `process`, conformément à l'exigence E1 du tableau 4.5. Les contraintes temporelles et de mémoire (exigences E2 et E3) sont satisfaites par l'association de propriétés standards au composant `thread`.

Représentation graphique

La figure 4.4 illustre l'utilisation de ce patron de modélisation : un composant `thread` est contenu dans un sous-composant `process`. La notation graphique ne permettant pas de représenter les propriétés d'un composant AADL, les caractéristiques ne peuvent être spécifiées dans ce format.

Exemple

Le listing 4.4 illustre l'utilisation de ce patron. Un même type de tâche

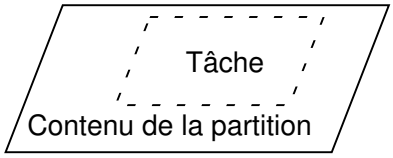


FIGURE 4.4 – Représentation graphique de la modélisation d'une tâche dans une partition

(`tache`) est dérivé à travers deux implantations différentes : une lente (`tache.lente`) et une rapide (`tache.rapide`). Toutes deux définissent et partagent les mêmes caractéristiques mémoire (exigence E3) à l'aide des propriétés `Source_Data_Size`, `Source_Stack_Size` et `Source_Code_Size`.

Ces tâches spécifient également leurs contraintes temporelles (exigence E2) en définissant les propriétés `Period`, `Deadline` et `Compute_Execution_Time`. Les valeurs diffèrent entre les implantations, l'une (`tache.rapide`) étant exécutée plus rapidement (propriété `Compute_Execution_Time`) et fréquemment (propriétés `Period` et `Deadline`) que l'autre (`tache.lente`).

Enfin, ces composants `thread` sont contenus dans un `process`, modélisant leur appartenance à une partition (exigence E1).

```
process partition
...
end partition;

thread tache
properties
   Source_Code_Size  => 3 KByte;
   Source_Data_Size  => 4 KByte;
   Source_Stack_Size => 10 KByte;
end tache;

thread implementation tache.rapide
properties
   Compute_Execution_Time => 1 .. 2ms;
   Period                 => 5 ms;
   Deadline               => 5 ms;
end tache.rapide;

thread implementation tache.lente
properties
   Compute_Execution_Time => 10 .. 20ms;
   -- Temps minimal d'execution : 10ms
   -- Temps maximal d'execution : 20ms
   Period                 => 50 ms;
   Deadline               => 50 ms;
end tache.lente;

process implementation partition.i
subcomponents
   tache1 : thread tache.rapide;
   tache2 : thread tache.lente;
end partition.i;
```

Listing 4.4 – Exemple de spécification d'une tâche

	Services intra-partition ($PM^{intra-partition}$)	
	Exigences de l'architecture partitionnée	**Modélisation AADL**
Buffer	Envoi/réception de données entre tâches avec mise en file des données (E1)	Connection de plusieurs `event data port` entre plusieurs composants de type `thread`.
	Taille des données échangées (E2)	Type de données dans la déclaration de `event data port`.
	Taille de la file de réception (E3)	Association de la propriété `Queue_Size` à l'`event data port`.
	Politique de mise en file du port (E4)	Association de la propriété `Queue_Processing_Protocol` à l'`event data port` (valeurs possibles : `Fifo` et `Priority_Order`)
Blackboard	Envoi/réception de données entre tâches sans mise en file des données (E5)	Connection de plusieurs `data port` entre plusieurs composants de type `thread`.
	Taille des données échangées (E2)	Type de données dans la déclaration de `data port`.
	Temps d'attente maximal d'une donnée (E6)	*Association de la propriété `Timeout` au `data port`.* – `Timeout : time applies to (port, data)`.
Events	Envoi/réception de signaux entre tâches d'une même partition (E7)	Connection de plusieurs `event port` entre plusieurs composants de type `thread`.
	Non conservation de l'historique des événements (E8)	Association à l'`event port` de la propriété `Queue_Size` avec une valeur définie à 1.
	Temps d'attente maximal d'une donnée (E6)	*Association de la propriété `Timeout` à `event port`.* – `Timeout : time applies to (port, data)`.
Semaphore	Mécanisme de protection associé à une donnée partagée entre plusieurs tâches d'une partition (E9)	Composant `data` partagé entre plusieurs `thread` d'un même `process`.
	Spécification de la politique d'attente (E10)	Association de la propriété `Concurrency_Control_Protocol` au composant `data`. (valeurs possibles : `Fifo` et `Priority_Order`)
	Temps d'attente maximal (E6)	*Association de la propriété `Timeout` au composant `data`* – `Timeout : time applies to (port, data)`.

TABLE 4.6 – Patrons de modélisation des services intra-partition

4.2.6 Communications intra-partition

Une partition supporte les mécanismes de communication suivants pour la communication entre tâches :

1. *buffer*
2. *blackboard*
3. *events*

4. *semaphores*

Pour chaque type de communication, un patron de modélisation est proposé dans le tableau 4.6.

Description des patrons de modélisation

Les services `buffer`, `blackboard` et `events` sont respectivement représentés par la connection d'`event data`, `data` ou `event` ports `aadl`. Le sémaphore est quant à lui spécifié par une donnée partagée. Les composants `thread` connectés ou partageant une donnée doivent appartenir à une même partition (exigences E1, E5, E7 et E9 du patron 4.6).

La propriété AADL `Queue_Processing_Protocol` décrit la politique de mise en file relative au service *buffer* (exigence E4).

Lorsque les événements ou les données échangées sont conservées dans une file, sa taille est spécifiée par la propriété `Queue_Size` (exigence E3). En revanche, le service d'événement ne conserve aucun historique des données reçues (exigence E8) alors que la sémantique des `events port` d'AADL est contraire à cette contrainte. Afin d'éviter toute confusion, et expliciter la sémantique de la modélisation, le patron requiert que cette propriété soit associée au port et que sa valeur soit égale à 1 (exigence E8 du service d'événement : la file de donnée correspondant au service d'événement est donc de taille 1).

La taille des données échangées (exigence E2) est déduite du type de données associé aux `ports`.

Lorsqu'un service introduit un temps d'attente maximal (*timeout*), il est spécifié par la définition de la propriété `Timeout` au port ou à la donnée AADL modélisant le service (exigence E6).

Enfin, l'association de la propriété standard AADL `Concurrency_Control_Protocol` modélise la protection de la donnée par un sémaphore mais également la politique d'attente utilisée. Les valeurs autorisées sont `Fifo` et `Priority_Order`, politiques supportées par les architectures partitionnées telles ARINC653 ou MILS.

Représentation graphique

La figure 4.5 illustre les patrons associés aux services intra-partition. Deux tâches contenues au sein d'une même partition échangent des données *via* les différents canaux de communication intra-partition. La représentation graphique ne permettant pas d'exprimer les propriétés associées aux ports et connections, celles-ci sont détaillées dans l'exemple.

Exemple

Le listing 4.5 illustre l'utilisation de ces patrons de modélisation. Il définit deux tâches (l'une émettrice, l'autre réceptrice) au sein d'une même partition qui échangent des données au moyen des quatre services de communication intra-partition.

Le *buffer* utilise une politique de mise en file `Fifo` : chaque donnée est mise en file à sa réception, indépendemment de sa priorité. La taille de la file associée à la tâche réceptrice est de 2 (propriété `Queue_Size`).

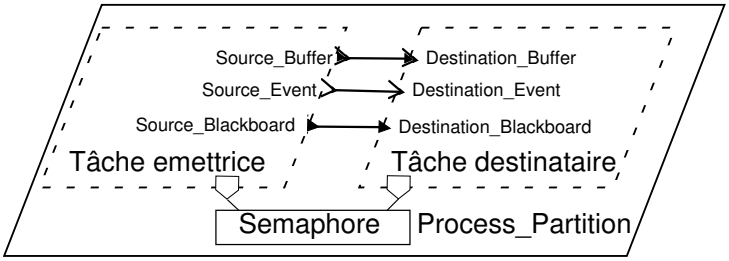


FIGURE 4.5 – Représentation graphique des services de communication intra-partition

Le *blackboard* définit un temps d'attente maximal de la tâche réceptrice de 100 ms (propriété `Timeout`) : passé ce temps, si aucune donnée n'est reçue, la tâche continue son exécution.

L'*events* utilise quant à lui un temps d'attente maximal de 100 ms (propriété `Timeout`) et associe également la propriété `Queue_Size` comme l'exige le patron de modélisation (exigence E8).

Enfin, le semaphore est représenté par une instance de composant `type_donnee`. La politique de partage est de type `Fifo` (la première tâche ayant demandé un accès au semaphore l'obtient propriété `Concurrency_Control_Protocol`) et le temps d'attente maximal à 100 ms (une tâche n'attend pas plus de 100ms la libération du verrou associé à la ressource).

```
data implementation type_donnee.i
...
end type_donnee.i;

thread emetteur
features
   source_buffer      : out event data port type_donnee
      {Queue_Processing_Protocol => Fifo;};
   source_blackboard  : out data port type_donnee;
   source_event       : out event port
      {Queue_Size => 1;};
   acces_semaphore    : requires data access type_donnee.i;
properties
   ...
end emetteur;

thread implementation emetteur.i
...
end emetteur.i;

thread recepteur
features
   destination_buffer       : in event data port type_donnee
      {Queue_Processing_Protocol => Fifo;
       Queue_Size                => 2};
   destination_blackboard   : in data port type_donnee
      {Timeout => 100 ms;};
   destination_event        : in event port
      {Timeout    => 100 ms;
       Queue_Size => 1;};
   acces_semaphore    : requires data access type_donnee.i;
end recepteur;
```

```
thread implementation recepteur.i
...
end recepteur.i;

process partition
...
end partition;

process implementation partition.i
subcomponents
   tache_emettrice    : thread emetteur.i;
   tache_receptrice   : thread recepteur.i;
   donnee_partagee    : data type_donnee
      {Concurrency_Control_Protocol => Fifo;
       Timeout                      => 200 ms;};
connections
   port tache_emettrice.source_buffer ->
        tache_receptrice.destination_buffer;
   port tache_emettrice.source_blackboard ->
        tache_receptrice.destination_blackboard;
   port tache_emettrice.source_event ->
        tache_receptrice.destination_event;
   data access donnee_partagee ->
         tache_emettrice.acces_semaphore;
   data access donnee_partagee ->
         tache_receptrice.acces_semaphore;
properties
   ...
end partition.i;
```

Listing 4.5 – Exemple de spécification de services intra-partition

4.2.7 Communications inter-partitions

Il existe plusieurs mécanismes de communication entre les partitions :

1. *queuing ports* : conservation de l'historique des données reçues dans une file de taille prédéterminée.
2. *sampling ports* : stockage de la dernière valeur reçue.

Description des patrons

Deux patrons (tableau 4.7) représentent ces services en AADL. Ces derniers sont modélisés par la connection de `ports` AADL entre plusieurs composants `process`, connectant ainsi plusieurs partitions entre elles (exigence E1 de chaque patron).

La taille des données échangées par ces deux services est spécifiée par le type de données AADL (composant `data`) associé au `port`.

La taille de la file de réception d'un queuing port (exigence E3) ainsi que sa politique de mise en file (exigence E4) sont décrites par l'association des propriétés `Queue_Size` et `Queue_Processing_Protocol` aux `ports`.

Le temps maximal d'attente d'une donnée sur un *queuing port* est représenté par l'association de la propriété *Timeout* au port récepteur de la donnée.

Le service de *sampling port* spécifie quant à lui un intervalle de temps entre deux occurrences de données (exigence E7), permettant ainsi de déterminer

	Services inter-partitions ($PM^{inter-partitions}$)	
	Exigences de l'architecture partitionnée	**Modélisation AADL**
Queuing Port	Envoi/réception de données entre partitions avec mise en file des donnees (E1)	Connection de plusieurs `event data port` entre plusieurs composants de type `process`.
	Taille des données échangées (E2)	Type de données associé à l'`event data port`.
	Taille de la file de réception (E3)	Association de la propriété `Queue_Size` à l'`event data port`.
	Politique de mise en file (E4)	Association de la propriété `Queue_Processing_Protocol` à l'`event data port` (valeurs possibles : `Fifo` et `Priority_Order`)
	Temps d'attente maximal d'une donnée (E5)	*Association de la propriété `Timeout` à l'`event data port`.* – `Timeout : time applies to (port, data)`.
Sampling Port	Envoi/réception de données entre tâches sans mise en file des données (E6)	Connection de plusieurs `data port` entre plusieurs composants de type `process`.
	Taille des données échangées (E2)	Type de données dans la déclaration de `data port`.
	Temps d'arrivée entre deux données (E7)	*Association de la propriété `Refresh_Period` au `data port`.* – `Refresh_Period : time applies to (port)`.

TABLE 4.7 – Patrons de modélisation des services inter-partitions

la validité d'une donnée. Cette caractéristique est représentée par la propriété `Refresh_Period` associée à un `data port`.

Représentation graphique

La représentation graphique des patrons est illustrée dans la figure 4.6 qui contient deux composants `process`. Ceux-ci échangent des données via le service de *queuing* et *sampling* ports. Ce format de représentation permet de représenter les connections mais les spécificités de chaque service ne sont pas décrit. L'exemple ci-après les définit.

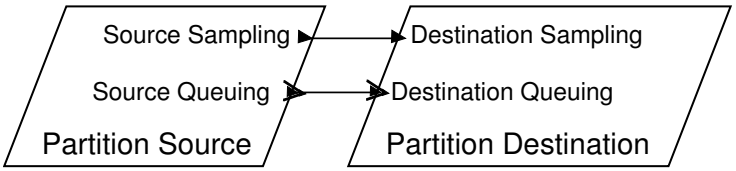


FIGURE 4.6 – Représentation graphique des services de communication inter-partitions

Exemple

Le listing 4.6 montre la mise en œuvre de ces deux patrons de modélisation. Il définit deux partitions (`partition_emettrice` et `partition_receptrice`) échangeant des données via les services de *sampling* et *queuing* ports.

Les deux services échangent des données de type `type_donnee`, permettant ainsi de caractériser leur taille (exigence E2).

Le service de *queuing* port reçoit les données dans une file pouvant contenir deux données (exigence E3 satisfaite par l'association de la propriété `Queue_Size`) et les ordonne suivant leur ordre d'arrivée (valeur de la propriété `Queue_Processing_Protocol` à `Fifo`).

Le service de *sampling* port décrit dans cet exemple envoie des données à un intervalle inférieur à 100 ms (exigence E7, propriété `Refresh_Period` associée au port `destination_sampling`). Si une donnée est lue plus de 100 ms après sa réception, elle sera alors considérée comme invalide.

```
data implementation type_donnee.i
...
end type_donnee.i;

process emetteur
features
   source_queuing      : out event data port type_donnee
      {Queue_Processing_Protocol => Fifo;};
   source_sampling     : out data port type_donnee;
properties
   ...
end emetteur;

process implementation emetteur.i
...
end emetteur.i;

process recepteur
features
   destination_queuing      : in event data port type_donnee
      {Queue_Processing_Protocol => Fifo;
       Queue_Size                => 2;
       Timeout                   => 100 ms};
   destination_sampling     : in data port type_donnee
      {Refresh_Period => 100 ms;};
end recepteur;

process implementation recepteur.i
...
end recepteur.i;

system systeme_partitionne
...
end systeme_partitionne;

system implementation systeme_partitionne.i
subcomponents
   partition_emettrice   : process emetteur.i;
   partition_receptrice  : process recepteur.i;
connections
   port partition_emettrice.source_queuing ->
        partition_receptrice.destination_queuing;
   port partition_emettrice.source_sampling ->
        partition_receptrice.destination_sampling;
properties
```

```
   ...
end partition.i;
```

Listing 4.6 – Exemple de spécification de services inter-partitions

4.3 Validation de l'architecture partitionnée

Les sections suivantes présentent les règles de validation dédiées aux architectures partitionnées. Dans un premier temps, la structure du modèle est vérifiée assurant que l'utilisateur respecte les patrons de modélisation proposés. Puis, les caractéristiques (isolation temporelle, spatiale, etc.) du système sont analysées afin de vérifier la faisabilité de l'architecture (les segments mémoires sont-ils suffisament dimenssionnés, évaluation de l'ordonnançabilité du système, etc.).

4.3.1 Conformité de la structure du modèle

Cette première étape de validation analyse le modèle AADL et vérifie que l'utilisateur a respecté les patrons de modélisation dédiés à la représentation de systèmes partitionnés. Cette phase de validation assure que l'utilisateur les a correctement utilisés, détectant ainsi toute erreur au plus tôt dans le cycle de développement.

La première règle (listing 4.7) vérifie l'association entre les composants `process`, `memory` et `virtual processor`. Elle assure que l'utilisateur a correctement spécifié la partition et ses exigences en modélisant le contenu de la partition (composant `process`), son environnement d'exécution (composant `virtual processor`) et son segment mémoire (composant `memory`). Elle vérifie également qu'à chaque partition n'est associé qu'un un seul segment mémoire et un unique environnement d'exécution.

```
Pour tout composant process prs faire
   liste segmentsMemoire = composants memory associes a prs;
   liste partitions      = composants virtual processor associes a prs;

   Verifier que
      ((taille (segmentsMemoire) == 1)
       et
       (taille (partitions) == 1)
      );
fin pour tout;
```

Listing 4.7 – Règle validant la conformité des composants process

La seconde règle (listing 4.8) valide la spécification de la mémoire du système. Elle analyse le composant `memory` principal du système racine (représentant la mémoire RAM) et vérifie qu'il spécifie plusieurs sous-composants `memory` (segments) avec leurs caractéristiques (propriétés AADL décrivant la

taille et l'emplacement du segment).

Cette règle assure que l'utilisateur a modélisé la mémoire interne du système en la segmentant en sous-composants destinés à isoler spatialement les partitions.

```
Pour tout composant memory memoirePrincipale du systeme racine
   liste segmentsMemoire = sous-composants memory de memoirePrincipale;

   verifier que
      ((memoirePrincipale existe)
      et
       (la propriete "Byte_Count" est definie sur
        chaque composant de la liste segmentsMemoire)
      );
fin pour tout;
```

Listing 4.8 – Règle validant la conformité des composants memory

4.3.2 Isolation temporelle

L'isolation temporelle consiste à ordonnancer périodiquement les partitions suivant des tranches de temps fixes et prédéterminées. A la fin de chaque période (*Major Frame*), les communications inter-partitions sont réalisées. La validation de cette exigence doit assurer que :

1. chaque partition est exécutée au moins une fois à chaque période.
2. la valeur du *Major Frame* est égale à la somme des fenêtres de temps d'exécution allouées aux partitions.

La règle validant l'exécution de chaque partition analyse la liste des tranches de temps que chaque noyau (composant `processor`) alloue à ses partitions (composant `virtual processor`). Elle vérifie que chaque partition (composant `virtual processor` du composant `processor`) est référencée dans cette liste.

Le listing 4.9 illustre cette règle : pour tout composant processor (noyau partitionné), elle vérifie que chaque `virtual processor` (partition) qu'il contient est référencé dans la liste des tranches de temps.

```
Pour tout composant processor cpu faire
   liste partitions      = ensemble des sous-composants
                           virtual processeur de cpu;
   liste tranchesDeTemps = propriete ARINC653::Slots_Allocation
                           du composant cpu;

   Pour tout partition p de la liste partitions
      Verifier que p est reference dans trancheDeTemps;
   fin pour tout;
fin pour tout;
```

Listing 4.9 – Règle validant l'exécution de chaque partition

La règle validant le major frame est illustrée dans le listing 4.10. Elle analyse chaque composant `processor` et assure que sa *major frame* (période à laquelle est répétée l'ordonnancement des partitions et spécifiée par la propriété `ARINC653::Module_Major_Frame`) est égale à la somme des temps d'exécution alloués aux partitions (propriété `ARINC653::Partitions_Slots`).

```
Pour tout composant processor cpu faire
   variable majorFrame      = propriete
                              "ARINC653::Module_Major_Frame" de cpu;
   liste    tranchesDeTemps = propriete "ARINC653::Partitions_Slots"
                              de cpu;

   Verifier que
      (somme des valeurs de tranchesDeTemps == majorFrame);
fin pour tout;
```

Listing 4.10 – Règle validant la conformité du major frame

4.3.3 Isolation spatiale

La règle associée à la validation de l'isolation spatiale assure que chaque partition est associée à un segment mémoire et que ce dernier est correctement dimensionné.

```
Pour tout composant process prs faire
   liste segmentsMemoires = composants memory associes a prs;
   liste memoiresPrincipales = composants parents de segmentsMemoires;

   Verifier que
      ((taille (liste segmentsMemoires) > 0)
       et
       (composants de la liste memoiresPrincipales sont de type memory));
fin pour tout;
```

Listing 4.11 – Règle validant l'assignation des partitions à un segment mémoire

La validation de l'assignation d'une partition à un segment mémoire est illustrée dans le listing 4.7 (section 4.3.1) et a pour but de vérifier la bonne constitution d'une partition (chaque partition est associée à un seul composant mémoire).

La règle de validation du listing 4.11 valide également cette exigence (chaque composant `process` est associé à un composant `memory`) mais vérifie également que le segment utilisé est inclus dans la mémoire principale (RAM).

La validation du dimensionnement de chaque segment mémoire est illustrée dans le listing 4.12. Elle analyse les ressources de la partition (composants AADL `thread`, `data`, etc.), leurs exigences mémoires (propriétés AADL associées à ces composants telles `Source_Stack_Size`, `Source_Code_Size`, etc.) et vérifie que la taille du segment associé à la partition est supérieure aux besoins

de ses ressources.

```
Pour tout composant process prs faire
   liste taches = sous-composants thread de prs;
   liste segmentsMemoire = composants memory associes a prs;

   liste tailleMemoire = propriete "Byte_Count" de segmentsMemoire;
   variable tailleDisponible = somme des valeurs de tailleMemoire;
   variable memoireRequise = propriete "Source_Stack_Size" de taches +
                             propriete "Source_Data_Size" de taches +
                             propriete "Source_Code_Size" de taches;

   Verifier que (tailleDisponible >= memoireRequise);
fin pour tout;
```

Listing 4.12 – Règle validant la taille des segments mémoires associés aux partitions

4.3.4 Analyse d'ordonnancement

Dans le contexte des systèmes critiques, la validation des contraintes temporelles est importante. Le non-respect d'une échéance signifie qu'une fonction n'a pu être exécutée, et peut avoir des conséquences graves. Plusieurs outils simulent et vérifient l'ordonnancement de systèmes temps-réel. Cependant, dans le cas des systèmes partitionnés, la validation de cette exigence est plus complexe car l'ordonnancement s'effectue à deux niveaux (ordonnancement dit *hiérarchique* opérant au niveau du noyau et des partitions).

Cette étape de validation doit donc considérer chaque niveau d'ordonnancement (noyau, partition) avec ses caractéristiques (tranches de temps allouées aux partitions, période et échéance de chaque tâche, etc.). Le nombre de paramètres étant important, cette validation est complexe et ne peut être réalisée manuellement.

Pour l'automatiser, nous utilisons des outils appropriés. Au cours des travaux présentés dans ce manuscrit, nous avons collaboré avec le laboratoire **LISyC**, initiateur du projet de simulateur d'ordonnancement **Cheddar** [104], dans le but de valider l'ordonnancement de systèmes partitionnés à partir de modèles AADL.

Ces travaux ont permis d'interfacer nos patrons de modélisation AADL avec **Cheddar**. Les modèles sont exportés vers Cheddar qui simule l'ordonnancement du système en incluant les contraintes temporelles de chaque niveau d'exécution du système (politique d'ordonnancement du noyau et de ses partitions). En particulier, il utilise notre patron de modélisation de l'isolation temporelle pour ordonnancer périodiquement chaque partition.

Cette simulation permet :

1. de visualiser l'ordonnancement du système (instants d'activation d'une tâche, temps de calcul, etc.).
2. de vérifier la faisabilité de la politique d'ordonnancement par une simulation du pire temps d'exécution de chaque tâche.

Ainsi, l'utilisateur obtient des garanties quant à l'ordonnancement de son système à partir de ses modèles AADL. Ces travaux ont été l'objet d'une publication dans la conférence *SigAda09* [37].

4.4 Modélisation des aspects sûreté

Cette section présente les patrons de modélisation dédiés à la modélisation des erreurs et de leur politique de recouvrement. Elle introduit les règles de validation assurant la reprise des erreurs à l'exécution et l'analyse de l'impact des politiques de recouvrement.

AADL propose un langage d'annexe dédié à la modélisation des erreurs [44]. Cependant, celui-ci s'est avéré inadapté à la modélisation des systèmes partitionnés pour les raisons suivantes :

1. la version actuelle de l'*Error-Model Annex* s'appuie sur la première version du standard AADL. Or, nos patrons se basent sur les composants introduits par la seconde version. Par conséquent, il n'est pas possible de l'utiliser avec nos patrons de modélisation.
2. cette première version de l'annexe de modélisation des erreurs a été conçue pour détecter les erreurs matérielles. Elle est donc davantage orientée vers l'ingénierie système que l'ingénierie logicielle. Pour cette raison, elle est peu adaptée à la description des mécanismes ayant lieu pendant l'exécution (division par zéro, erreur de segmentation, redémarrage d'une tâche ou d'une partition, etc.)

Pour ces raisons, nous avons introduit des règles de modélisation des erreurs en ajoutant de nouvelles propriétés pour les composants `processor` (noyau), `virtual processor` (partition) et `thread` (tâche). Celles-ci définissent les erreurs et procédures de recouvrement associées à chaque couche de l'architecture partitionnée.

Cependant, l'utilisation d'une annexe adaptée à la description des erreurs reste pertinente. Elle offrirait davantage de souplesse et de généricité dans la description de l'architecture. Cette perspective est discutée au cours de notre conclusion, au cours de la section 7.2.1.

4.4.1 Niveau de criticité d'une partition

Définition : Criticité

La criticité représente le degré d'importance d'un composant. Dans le cas des architectures partitionnées, la criticité d'une partition spécifie l'importance des fonctions remplies par cette partition au regard des autres [99, 94].

Le niveau de criticité définit l'*importance* de la partition et de ses contraintes (niveau de certification requis, etc.). Cette information peut être exploitée pour détecter les interactions possibles entre partitions de criticité hétérogène (par exemple, détecter qu'une partition peu critique peut perturber le fonctionnement d'une partition l'étant davantage).

Criticité d'une partition	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Définition de la criticité d'une partition (E1)	*Propriété* `Criticality` *associée à un composant* `virtual processor` *(environnement d'exécution d'une partition) –* `Criticality : Supported_Criticality_Values applies to (virtual processor)`.

TABLE 4.8 – Patron de modélisation de la criticité d'une partition

Description

Le patron de modélisation (tableau 4.8) introduit la propriété `Criticality` et l'associe au composant AADL `virtual processor`. La valeur de cette propriété est une lettre (allant de `A` à `E`) spécifiant le niveau de criticité de la partition comme le prévoit le standard DO178B.

Exemple

Le listing 4.13 illustre la définition de la criticité d'une partition. La propriété AADL `Criticality` est associée au composant `virtual processor` modélisant l'environnement d'exécution de chaque partition.

```
virtual processor environnement_execution
...
properties
   Criticality => B;
end environnement_execution;
```

Listing 4.13 – Exemple de spécification de la criticité d'une partition

4.4.2 Représentation des fautes

Chaque niveau de l'architecture partitionnée (noyau, partition, tâche) peut détecter et réparer les erreurs survenant lors de l'exécution du système. Un patron (tableau 4.9) permet de les décrire.

Fautes (PM^{fautes})	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Spécification des fautes des tâches (E1)	*Propriété* `HM_Errors` *associée à un composant* `thread` *(tâche)* – `HM_Errors : Supported_Error_Code applies to (thread, processor, virtual processor).`
Spécification des fautes des partitions (E2)	*Propriété* `HM_Errors` *associée à un composant* `virtual processor` *(environnement d'exécution d'une partition)* – `HM_Errors : Supported_Error_Code applies to (thread, processor, virtual processor).`
Spécification des fautes du noyau de partitionnement (E3)	*Propriété* `HM_Errors` *associée à un composant* `processor` *(noyau de partitionnement)* – `HM_Errors : Supported_Error_Code applies to (thread, processor, virtual processor).`

TABLE 4.9 – Patron de modélisation des erreurs

Description du patron

Le patron dédié à la représentation des fautes (tableau 4.9) introduit la propriété AADL `HM_Errors`. Elle spécifie la liste des erreurs gérées par chaque couche de l'architecture : noyau (exigence E1), partition (E2) et tâche (E3). Le tableau 4.10 indique la liste des valeurs qui peuvent être associées à cette propriété.

Exemple

Le listing 4.14 utilise la propriété proposée (`HM_Errors`) pour les trois couches de l'architecture partitionnée :

1. tâche : le composant `tache` gère les erreurs applicatives (`Application_Error`) ainsi que celles relatives à la gestion des nombres (`Numeric_Error`).
2. partition : l'environnement d'exécution (composant `environnement_partition`) se charge des erreurs de configuration (`Partition_Config`) et d'ordonnancement (`Partition_Scheduling`).
3. noyau : le composant `noyau` détecte les erreurs d'initialisation (`Kernel_Init`) et d'ordonnancement des partitions (`Kernel_Scheduling`).

```
thread tache
...
properties
   HM_Errors    => (Numeric_Error, Application_Error);
   HM_Actions   => (Thread_Restart, Partition_Restart);
end tache;

virtual processor environnement_execution
```

```
...
properties
   HM_Errors    => (Partition_Config, Partition_Scheduling);
   HM_Actions   => (Partition_Stop, Partition_Restart);
end environnement_execution;

processor noyau
...
properties
   HM_Errors    => (Kernel_Init, Kernel_Scheduling);
   HM_Actions   => (Kernel_Stop, Kernel_Restart);
end noyau;
```

Listing 4.14 – Exemple de spécification des erreurs à chaque niveau de l'architecture

Valeur	**Signification**
`Module_Config`	configuration du module invalide (ex : le *« Major Frame »* est incorrect).
`Module_Init`	erreur lors de l'initialisation du module (ex : impossible de charger une partition).
`Module_Scheduling`	erreur lors de l'exécution du premier niveau d'ordonnancement (partitions).
`Partition_Scheduling`	erreur lors de l'exécution du second niveau d'ordonnancement (tâches de la partition élue).
`Partition_Config`	erreur dans la configuration de la partition (ex : nombre de tâche invalide).
`Partition_Handler`	erreur lors de l'exécution de la politique de recouvrement de la partition.
`Partition_Init`	erreur lors de l'initialisation de la partition.
`Deadline_Miss`	échéance ratée par une tâche au sein d'une partition.
`Application_Error`	erreur applicative levée par une tâche au sein d'une partition.
`Numeric_Error`	erreur relative à une opération mathématique (ex : division par zéro).
`Illegal_Request`	requête invalide faîte au noyau par une tâche (ex : appel système invalide).
`Stack_Overflow`	débordement dans la pile d'exécution d'une tâche.
`Memory_Violation`	violation de l'isolation spatiale (la tâche essaye d'accéder à une zone mémoire qui n'appartient pas à sa partition).
`Hardware_Fault`	erreur matérielle (ex : périphérique ayant levé une erreur).
`Power_Fail`	erreur d'alimentation (ex : le système va s'arrêter dans 5 minutes).

Table 4.10 – Valeurs possibles de la propriété `HM_Errors` (`Supported_Error_Code`)

4.4.3 Politiques de recouvrement

Il est nécessaire que chaque composant associe une action à chaque erreur détectée. Un patron de modélisation (tableau 4.11) spécifie ces différentes procédures.

Action de recouvrement (PM^{actions})	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Spécification des actions de recouvrement des tâches (E1)	*Propriété* `HM_Actions` *associée à un composant* `thread` *(tâche)* – `HM_Actions : Supported_Recovery_Action applies to (thread, processor, virtual processor)`.
Spécification des actions de recouvrement des partitions (E2)	*Propriété* `HM_Actions` *associée à un composant* `virtual processor` *(environnement d'exécution d'une partition)* – `HM_Errors : Supported_Recovery_Action applies to (thread, processor, virtual processor)`.
Spécification des actions de recouvrement du noyau de partitionnement (E3)	*Propriété* `HM_Actions` *associée à un composant* `processor` *(noyau de partitionnement)* – `HM_Actions : Supported_Recovery_Action applies to (thread, processor, virtual processor)`.

TABLE 4.11 – Patron de modélisation des actions de recouvrement

Description

Le patron (tableau 4.11) spécifie la politique de recouvrement associée à chaque faute. Il introduit la propriété (`HM_Actions` pour *Health Monitor Action*) qui décrit les actions de recouvrement associées aux erreurs détectées par chaque niveau de l'architecture partitionnée : tâche (exigence E1), partition (exigence E2) et noyau (exigence E3). Les valeurs qu'elle peut contenir (énumération AADL `Supported_Recovery_Actions`) sont énumérées dans le tableau 4.12. Leur ordre respecte celui des erreurs spécifiées par la propriété `HM_Errors` (section 4.4.2) : la n^{ime} valeur de la propriété `HM_Actions` représente la procédure de recouvrement pour la n^{ime} faute spécifiée par la propriété `HM_Errors`.

Exemple

Le listing 4.14, issu de la section précédente illustre l'utilisation de ce patron. Pour chaque erreur décrite par la propriété `HM_Errors`, une action de recouvrement est spécifiée par `HM_Actions`.

Les actions de recouvrement sont spécifiées pour chaque niveau de l'architecture :

1. tâche : une erreur numérique détectée entraîne son redémarrage (`Thread_Restart`) et une erreur applicative implique le redémarrage de la partition (`Partition_Restart`).
2. partition : une erreur de configuration l'arrête (`Partition_Stop`) et une erreur d'ordonnancement la redémarre (`Partition_Restart`).
3. noyau : une erreur d'initialisation entraîne son arrêt (`Kernel_Stop`) et une erreur d'ordonnancement implique son redémarrage (`Kernel_Restart`).

Valeur	Signification
`Ignore`	ignorer et l'enregistrer.
`Confirm`	attendre une confirmation de l'erreur.
`Partition_Stop`	arrêter la partition ayant levé l'erreur.
`Module_Stop`	arrêter le module.
`Process_Stop`	arrêter le processus fautif.
`Process_Restart`	redémarrer le processus/la tâche fautive.
`Partition_Restart`	redémarrer la partition contenant l'erreur.
`Module_Restart`	redémarrer le module et toutes les partitions qu'il exécute.
`Nothing`	ne rien faire.

TABLE 4.12 – Valeurs possibles de la propriété `HM_Actions`

4.5 Validation des aspects sûreté

Les paragraphes suivants présentent les règles de validation dédiées à la sûreté. Dans un premier temps, une analyse sémantique assure la faisabilité de la politique de sûreté décrite par l'utilisateur. Puis, d'autres règles assurent la couverture d'erreurs (toute erreur est associée à une politique de recouvrement) et l'impact des fautes entre partitions de criticité hétérogène.

4.5.1 Conformité de la politique de sûreté

Les règles illustrées dans les listings 4.15, 4.16 et 4.17 valident la conformité des erreurs et les politiques de recouvrement spécifiées par l'utilisateur. Elles valident que les erreurs et actions spécifiées sont sémantiquement correctes au regard du niveau qui est censé le détecter (par exemple, une tâche ne peut pas redémarrer le module contrôlant l'ensemble des partitions, cela impacterait alors des partitions qui ne l'incluent pas et violerait les propriétés d'isolation). Ces règles de validation correspondent à celles décrites dans le standard ARINC653 [4] et veillent à ce que la spécification du système respecte les caractéristiques de confinement des systèmes partitionnés.

```
Pour tout composant processor cpu faire
   liste erreursGerees  = propriete "HM_Errors" de cpu;
   liste erreursActions = propriete "HM_Actions" de cpu;


   Pour toute valeur erreur de la liste erreursGerees faire
      Verifier que erreur est inclue dans la liste
         ("Module_Config", "Module_Init", "Module_Scheduling");
   fin pour tout;

   Pour toute valeur action de la liste erreursActions faire
      Verifier que action est inclue dans la liste
         ("Nothing", "Module_Stop", "Module_Restart");
   fin pour tout;
fin pour tout;
```

Listing 4.15 – Validation de la spécification du *Health Monitoring* du Noyau

La règle du listing 4.15 assure que le noyau (composant `processor`) ne détecte que les erreurs d'initialisation et d'ordonnancement des partitions. De la

même manière, elle valide la politique de recouvrement en assurant que seules certaines actions sont déclenchées en cas de détection d'une erreur (aucune action, arrêt ou redémarrage du noyau).

```
Pour tout composant virtual processor partition faire
   liste erreursGerees = propriete "ARINC653::HM_Errors" de partition;
   liste actionsGerees = propriete "ARINC653::HM_Action" de partition;

   Pour toute valeur erreur de la liste erreursGerees faire
      Verifier que erreur est comprise dans la liste suivante
          ("Partition_Config", "Partition_Init",
           "Partition_Scheduling", "Partition_Handler");
   fin pour tout;

   Pour toute valeur action de la liste actionsGerees faire
      Verifier que action est comprise dans la liste suivante
         ("Nothing", "Partition_Stop", "Partition_Restart");

   fin pour tout;
fin pour tout;
```

Listing 4.16 – Validation de la spécification du *Health Monitoring* des Partitions

La règle du listing 4.16 garantit que les partitions (composants `virtual processor`) ne peuvent détecter que les erreurs d'initialisation et d'ordonnancement de leurs tâches. La politique de recouvrement associée à chacune des actions ne peut consister qu'à redémarrer la partition, l'arrêter ou ne rien faire. En aucun cas la partition ne peut avoir un impact sur le fonctionnement du noyau et donc, perturber le bon fonctionnement des autres partitions.

La règle du listing 4.17 garantit quant à elle que :

- seules les erreurs applicatives, relatives à l'exécution du code, peuvent être détectées par les tâches (division par zéro, erreur d'accès mémoire, etc.)
- les politiques de recouvrement exécutées par les tâches ne peuvent qu'impacter les tâches appartenant à la même partition (assurance du confinement des fautes).

Ainsi, cette règle assure la conformité de la politique de détection et de recouvrement d'erreur spécifiée par ARINC653 et garantit l'absence d'impact entre partitions ayant potentiellement des niveaux de criticité différents.

```
Pour tout composant thread tache faire
   liste erreursGerees = propriete "ARINC653::HM_Errors" de tache;
   liste actionsGerees = propriete "ARINC653::HM_Actions" de tache;

   Pour toute valeur erreur de erreursGerees faire
      Verifier que erreur est compris dans la liste
         ("Deadline_Miss", "Application_Error", "Numeric_Error",
         "Illegal_Request", "Stack_Overflow", "Memory_Violation",
         "Hardware_Fault", "Power_Fail");
   fin pour tout;

   Pour toute valeur action de actionsGerees faire
```

```
      Verifier que action est compris dans la liste
      ("Ignore", "Confirm", "Process_Stop",
       "Process_Stop_And_Start_Another",
       "Process_Restart", "Partition_Stop", "Partition_Restart");
   fin pour tout;
fin pour tout;
```

Listing 4.17 – Validation de la spécification du *Health Monitoring* des Tâches

Enfin, la règle présentée dans le listing 4.18 assure que l'utilisateur du modèle a spécifié le niveau de criticité de chaque partition. La validation de cette règle permet alors aux outils d'analyser le système en fonction de chaque niveau de criticité, cette notion étant utilisée pour l'analyse (section 4.5.3) et pour la certification du système.

```
Pour tout composant virtual processor partition
   verifier que
      (la propriete "ARINC653::Criticality"
       est definie sur partition);
fin pour tout;
```

Listing 4.18 – Validation de la criticité des partitions

4.5.2 Couverture des fautes

L'analyse de la couverture de fautes au sein de l'architecture partitionnée assure qu'à chaque instant, toutes les fautes potentielles que nous proposons de spécifier sont gérées et associées à une politique de recouvrement.

La règle de validation associée analyse chaque tâche de l'architecture. Pour chacune d'entre elles, elle calcule la liste des fautes gérées respectivement par la tâche, la partition et le noyau qui l'exécutent. Elle vérifie alors que cette liste de fautes correspond à l'ensemble des fautes que le système est capable d'intercepter. Si cette caractéristique se vérifie pour chaque tâche, on considère le système sera capable d'intercepter toutes les fautes à chaque instant de son exécution.

Cette règle est illustrée dans le listing 4.19 et analyse la couverture des fautes en utilisant nos patrons de modélisation. Pour chaque tâche, elle calcule l'ensemble des fautes gérées par la tâche, la partition et le noyau qui l'exécutent. Puis, elle compare cette liste à l'ensemble des fautes pouvant survenir au sein du système, assurant ainsi que toute faute pouvant survenir est spécifiée.

```
Pour tout composant thread tache
   variable prs       = composant parent de type process de tache;
   variable partition = composant virtual processor associe a prs;
   variable noyau     = composant parent de type processor de partition;

   liste erreursGerees = propriete "ARINC653::HM_Errors" de tache +
                         propriete "ARINC653::HM_Errors" de partition +
                         propriete "ARINC653::HM_Errors" de noyau

   Verifier que
```

```
    (erreursGerees =  "Module_Config", "Module_Init",
                      "Module_Scheduling"    ,
                      "Partition_Scheduling", "Partition_Config",
                      "Partition_Handler"    , "Partition_Init",
                      "Deadline_Miss"        , "Application_Error",
                      "Numeric_Error"        , "Illegal_Request",
                      "Stack_Overflow"       , "Memory_Violation",
                      "Hardware_Fault"       , "Power_Fail");
fin pour tout;
```

Listing 4.19 – Validation de la couverture de fautes

4.5.3 Impact de la politique de recouvrement

Lorsqu'une erreur est détectée, une tentative de réparation est exécutée par le déclenchement d'une action de recouvrement. Cependant, celle-ci peut avoir un impact sur l'exécution des autres entités du système par l'intermédiaire des communications inter-partitions. Lorsqu'une tâche connectée à une autre partition lève une erreur, l'exécution de l'action de recouvrement peut impacter la partition connectée. Dans ce cas, l'absence de données peut être **temporaire** (pendant le redémarrage de la tâche fautive) ou **permanente** (à cause de l'arrêt de la tâche).

Nous considérons alors qu'il y a un impact significatif lorsque la tâche fautive est contenue dans une partition moins critique que la tâche réceptrice : l'absence de données au sein d'une partition critique peut alors elle-même entraîner des défaillances applicatives importantes. L'erreur se propage donc entre partitions de criticités hétérogènes.

Afin de vérifier l'impact d'une erreur et de sa politique de recouvrement, deux règles de validation détectent les impacts des politiques de recouvrement entre partitions ayant une criticité différente. La première, décrite dans le listing 4.20, détecte les impacts temporaires alors que la seconde, illustrée dans le listing 4.21, détecte les impacts permanents.

Pour établir ces deux règles de validation, il faut dissocier les politiques de recouvrement ayant un impact de celles n'en ayant pas. Cette dichotomie est reprise des travaux d'analyse des erreurs [72] et sépare les politiques de recouvrement en trois groupes :

1. celles n'ayant aucun impact sur le système (faute ignorée ou devant être confirmée)
2. celles ayant un impact transitoire sur le système (redémarrage de l'entité fautive)
3. celles ayant un impact permanent sur le système (arrêt du composant fautif)

A partir de ces informations, chaque tâche est analysée :

- si la tâche est connectée à une tâche de plus forte criticité et qu'elle utilise un mécanisme de recouvrement pouvant arrêter le composant, la règle de validation spécifie qu'une erreur permanente peut impacter un composant plus critique.
- si la tâche est connectée à une tâche de plus forte criticité et que le mécanisme de recouvrement ne l'impacte que transitoirement, la règle

rapporte qu'une erreur transitoire peut impacter un composant plus critique.
- si la tâche utilise un mécanisme de recouvrement n'ayant aucun impact, aucune erreur n'est raportée.

Cette première étape d'analyse de la politique de recouvrement des erreurs assure à l'utilisateur l'absence d'impact entre composants de criticité différentes. Elle apporte une réelle plus-value car l'analyse des interactions entre composants de criticités différentes est difficile et n'est bien souvent détectée qu'à un stade trop avancé du processus de développement [77].

```
Pour tout composant process processSource faire
   variable partitionSource = composant virtual processor associe
                              a processSource;
   liste    tachesSource    = sous-composants thread de processSource;
   variable criticiteSource = propriete "ARINC653::Criticality"
                              de partitionSource;
   liste processesDestination = composants process connecte
                                a ProcessSource;

   Pour tout processDestination de la liste processesDestination faire
      variable partitionDestination = composant virtual processor
                                      associe a processDestination;
      variable criticiteDestination = propriete "ARINC653::Criticality"
                                      de partitionDestination;
      liste tachesDestination = sous-composants thread de
                                processDestination;

      liste actionsDestination = propriete "ARINC653::HM_Actions"
                                 de partitionDestination +
                                 propriete "ARINC653::HM_Actions"
                                 de tachesDestination;

      Verifier que
      ((criticiteDestination < criticiteSource)
       ou
       (valeurs de actionsDestination ne contient pas la valeur
        ("Partition_Restart", "Process_Restart", "Confirm")));
   fin pour tout;
fin pour tout;
```

Listing 4.20 – Règle de validation analysant l'impact transitoire d'une faute sur les autres partitions

```
Pour tout composant process processSource faire
   variable partitionSource = composant virtual processor
                              associe a processSource;
   liste    tachesSource   = sous-composants thread de processSource;
   variable criticiteSource = propriete "ARINC653::Criticality"
                              de partitionSource;
   liste processesDestination = composants process connecte a ProcessSource;

   Pour tout processDestination de la liste processesDestination faire
      variable partitionDestination = composant virtual processor
                                      associe a processDestination;
      variable criticiteDestination = propriete "ARINC653::Criticality"
                                      de partitionDestination;
      liste tachesDestination = sous-composants thread de
                                processDestination;

      liste actionsDestination = propriete "ARINC653::HM_Actions"
                                 de partitionDestination +
                                 propriete "ARINC653::HM_Actions"
                                 de tachesDestination;
      Verifier que
      ((criticiteDestination < criticiteSource) ou
       (valeurs de actionsDestination ne contient pas la valeur
        ("Partition_Stop", "Process_Stop",
         "Process_Stop_And_Start_Another")));
   fin pour tout;
fin pour tout;
```

Listing 4.21 – Règles de validation analysant l'impact permanent de la politique de reprise

4.6 Modélisation des aspects sécurité

Afin d'analyser et implanter les mécanismes de sécurité, il est nécessaire de proposer une méthode de représentation des différents niveaux et des ressources utilisées pour assurer la protection des données qui leur sont associées. Les prochaines sections présentent nos patrons de modélisation associés aux aspects sécurité des architectures partitionnées.

4.6.1 Niveaux de sécurité

Un niveau de sécurité (*top-secret*, *secret*, etc.) est modélisé en AADL par un composant `virtual bus`. Cette catégorie de composant est adaptée à la modélisation d'un concept abstrait comme un niveau de sécurité : il peut être étendu/hérité/redéfini et s'associe aux entités d'exécution utilisées pour les échanges de données : `virtual processor` (partitions), `port` (interfaces de communication), `bus`, etc.

Niveau de sécurité ($PM^{niveau-securite}$)	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Description d'un niveau de sécurité (E1)	Composant `virtual bus`.
Spécification de l'évaluation du niveau de sécurité (E2)	*Propriété `Security_Level` associée à un composant `virtual bus`* – `Security_Level : aadlinteger applies to (virtual bus)`.
Définition des niveaux de sécurité à une partition (E3)	Association de la propriété `Provided_Virtual_Bus_Class` (liste de composants `virtual bus`) à un composant `virtual processor`.
Définition du niveau de sécurité associé à un port inter-partitions (E4)	Association de la propriété `Allowed_Connection_Binding_Class` (liste de composants `virtual bus`) à un `port`.

TABLE 4.13 – Patron de modélisation des niveaux de sécurité

Description

Le patron de modélisation dédié aux niveaux de sécurité est présenté dans le tableau 4.13. Il représente un niveau de sécurité par un composant `virtual bus` (exigence E1 du tableau 4.13). Il introduit une propriété (`Security_Level`) pour spécifier l'évaluation du niveau de sécurité (exigence E2). Cette propriété a pour valeur un entier : plus l'entier est important, plus la contrainte de sécurité est forte.

Chaque partition décrit les niveaux de sécurité qu'elle supporte (exigence E3) par la définition de la propriété `Provided_Virtual_Bus_Class`. Elle associe à l'environnement d'exécution de la partition (composant `virtual processor`) la liste des niveaux de sécurité (`virtual bus`) qu'elle fournie.

Enfin, bien qu'une partition puisse supporter plusieurs niveaux de sécurité, chaque port de communication n'en utilise qu'un niveau pour échanger des données avec les autres entités. Il est donc nécessaire de représenter celui qui est utilisé par chaque interface de communication (exigence E4). Pour cela, le patron requiert que le niveau de sécurité soit défini par l'association de la propriété `Allowed_Connection_Binding_Class` à chaque port inter-partition et désigne le niveau de sécurité (`virtual bus`) supporté.

Représentation graphique

La figure 4.7 illustre l'utilisation du patron ainsi que l'exemple décrit dans le paragraphe suivant. Il est composé de deux niveaux de sécurité (composants `virtual bus`). La partition supporte ces deux niveaux et chacune de ses interfaces est associée à l'un d'entre eux (le port `queuing_port` est associé à `niveau_securite.secret`, `sampling_port` à `niveau_securite.topsecret`).

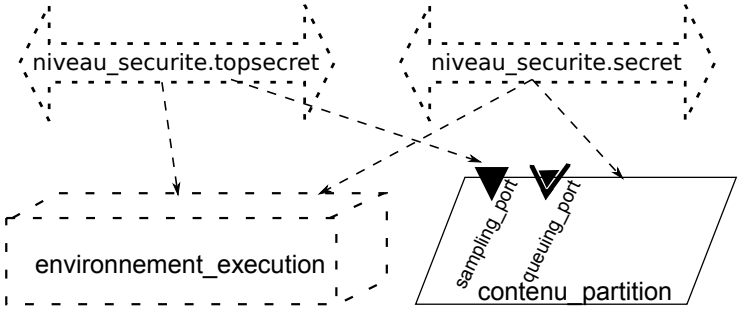


FIGURE 4.7 – Représentation graphique du patron de modélisation des niveaux de sécurité

Exemple

Le listing 4.22 illustre l'utilisation de ce patron de modélisation. Deux composants `virtual bus` (`niveau_securite.secret` et `niveau_securite.topsecret`) définissent deux niveaux de sécurité, le second étant classifié à un niveau plus élevé. Dans cet exemple, la partition est associée à ces deux niveaux de sécurité.

Ces deux niveaux sont associés à la partition, via l'association de la propriété `Provided_Virtual_Bus_Class` au composant `virtual processor` (modélisant son environnement). Enfin, chaque port de la partition utilise un niveau de sécurité, l'`eventdataport` est associé au niveau `niveau_securite.topsecret` et le `dataport` à `niveau_securite.secret`.

```
virtual bus implementation niveau_securite.secret
...
properties
   Security_Level => 2;
end niveau_securite.secret;

virtual bus implementation niveau_securite.topsecret
...
properties
```

```
   Security_Level => 5;
end niveau_securite.topsecret;

virtual processor implementation environnement_execution
...
properties
   Provided_Virtual_Bus_Class =>
         (classifier (niveau_securite.secret),
          classifier (niveau_securite.topsecret));
end environnement_execution;

process contenu_partition
features
   sampling_port : in data port type_donnee
      {Allowed_Connection_Binding_Class =>
         (classifier (niveau_securite.topsecret));};
   queuing_port : event in data port type_donnee
      {Allowed_Connection_Binding_Class =>
         (classifier (niveau_securite.secret));};
end contenu_partition;
```

Listing 4.22 – Exemple de spécification et du déploiement des niveaux de sécurité

4.6.2 Mécanismes de chiffrement

Le patron précédent a introduit les constructions AADL nécessaire pour définir un niveau de sécurité dans l'architecture. Cependant, les informations ajoutées ne décrivent pas les mécanismes de protection de données qu'ils utilisent (tels les algorithmes de chiffrement). Dans ce but, un patron de modélisation permet de les définir.

Mécanismes de chiffrement ($PM^{chiffrement}$)	
Exigences de l'architecture partitionnée	**Modélisation AADL**
Représentation des ressources nécessaires à la protection des données (E1)	Modélisation des sous-programmes et types de données requises par le mécanisme représenté dans un composant de type `abstract`.
Association du mécanisme de protection à un niveau de sécurité (E2)	Propriété standard `Implemented_As` associée à un composant `virtual bus` et pointant sur le composant `abstract` introduit.
Description des paramètres de configuration (E3)	*Propriétés AADL associées au niveau de sécurité (`virtual bus`) et définissant les données de configuration/déploiement de chaque mécanisme..*

Table 4.14 – Patron de modélisation des mécanismes de sécurité

Description

Les mécanismes de chiffrement associés aux niveaux de sécurité (`virtual bus`) spécifient les ressources (sous-programmes, types de données, etc.) et

informations (clé de chiffrement, données d'initialisation, etc.) nécessaires à leur analyse et à leur implantation.

Le patron associé (tableau 4.14) définit les composants nécessaires au chiffrement des données (exigence E1) dans un composant de type `abstract`. Ce dernier contient les sous-composants suivants :

1. `send` : sous-composant de type `subprogram` modélisant la routine de chiffrement des données.
2. `receive` : sous-composant de type `subprogram` spécifiant la routine de déchiffrement des données.
3. `marshalling_type` : sous-composant de type `data` représentant le format des données une fois chiffrées.

Le composant `abstract` englobant ces ressources est associé à un niveau de sécurité (`virtual bus`), comme le requiert l'exigence E2 du patron, *via* la propriété `Implemented_As`.

Enfin, les mécanismes de chiffrement nécessitent la définition de paramètres supplémentaires à des fins de configuration/déploiement (clé de chiffrement, données d'initialisation, etc. exigence E3 du tableau 4.14). Dans ce but, de nouvelles propriétés AADL sont introduites pour spécifier chaque paramètre de configuration. Par exemple, dans le cas de la clé de chiffrement du protocole *Blowfish*, la propriété `Blowfish_Key` spécifie la clé de chiffrement et s'associe aux composant `virtual bus`.

Représentation graphique

La figure 4.8 illustre l'utilisation de ce patron. Ici, un `virtual bus`, `secret.i` définit un niveau de sécurité. Il hérite en outre du composant `des.i`. Ce dernier spécifie les mécanismes de protection *via* la propriété `Implemented_As` qui désigne un composant `abstract` contenant toutes les ressources nécessaires à son implantation (sous-programme utilisés pour chiffrer/déchiffrer les données, etc.).

Exemple

Le listing 4.23 (représentation textuelle) et la figure 4.8 (représentation graphique) illustrent l'utilisation de ce patron de modélisation. Dans cet exemple un niveau de sécurité de valeur 2 est défini. Les mécanismes de protection sont spécifiés (au moyen de la propriété `Implemented_As`) par un composant de type `abstract` contenant les ressources et informations nécessaires à leur implantation :

- un sous-programme pour le chiffrement des données : (`des_send.i`)
- un sous-programme pour le déchiffrement des données (`des_receive.i`)
- un composant `data` (`des_data.i`) qui spécifie la représentation des données une fois chiffrées.

Ces composants peuvent ensuite être analysés afin d'évaluer l'impact de la politique de sécurité (en terme de coût d'exécution, mémoire, etc.) sur les performances du système : une analyse ou une simulation de l'ordonnancement fine pourra alors tenir compte du temps nécessaire au chiffrement des données envoyées à chaque activation d'une tâche.

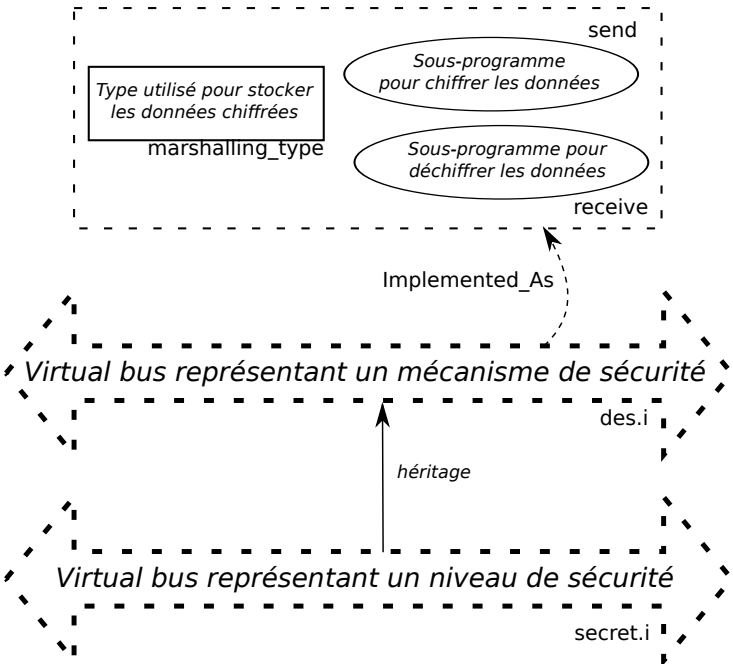


FIGURE 4.8 – Représentation graphique de la modélisation des mécanismes de sécurité

```
subprogram des_send ...
end des_send;

subprogram implementation des_send.i ...
end des_send.i;

subprogram des_receive ...
end des_receive;

subprogram implementation des_receive.i ...
end des_receive.i;

data des_data
end des_data;

data implementation des_data.i ...
end des_data.i;

abstract vbus_des_wrapper
end vbus_des_wrapper;

abstract implementation vbus_des_wrapper.i
subcomponents
   send              : subprogram des_send.i;
   receive           : subprogram des_receive.i;
   marshalling_type  : data des_data.i;
end vbus_des_wrapper.i;

virtual bus des
end des;

virtual bus implementation des.i
properties
   Implemented_As => classifier (virtualbusdes::vbus_des_wrapper.i);
end des.i;
```

```
virtual bus secret
end secret;

virtual bus implementation secret.i extends virtualbusdes::des.i
properties
   POK::Security_Level => 2;
   POK::Blowfish_Key =>
   "{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
     0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87}";
   POK::Blowfish_Init =>
   "{0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10}";
end secret.i;

end virtualbusdes;
```

Listing 4.23 – Niveau de sécurité utilisant le protocole de chiffrement DES

4.7 Validation des aspects sécurité

Les règles présentées dans les paragraphes suivants valident les aspects sécurité de l'architecture, assurant qu'un système respecte une politique de sécurité. Celles-ci considèrent que toutes les ressources d'une partition (tâches, données, etc.) sont évaluées au même niveau de sécurité. Celles-ci partageant le même espace d'adressage, une faille de sécurité de l'un d'eux expose *de facto* les autres composants de la partition.

Cela marque une différence avec certaines approches d'analyse de la sécurité qui se basent sur le fait qu'une partition peut contenir plusieurs tâches évaluées à des niveaux de sécurités hétérogènes. Ces approches ([111, 55]) vérifient l'isolation au sein des partitions alors que l'espace mémoire n'est pas protégé.

Notre approche est différente et considère que tous les composants d'une même partition sont évalués au même niveau de sécurité, tout comme des standards comme ARINC653 spécifie un unique niveau de criticité pour chaque partition (une même partition ne peut contenir des tâches ayant un niveau de criticité différent).

Il est également important de rappeler la potentielle incompatibilité des politiques de sécurité analysées. Par exemple, un modèle vérifiant la politique *Bell-Lapadula* ne vérifiera pas la politique *Biba*, ces deux politiques de sécurité étant opposées (leur description peut être trouvée dans la section 2.1.2). Le rôle de nos règles de validation est de montrer que nos patrons de modélisation peuvent être réutilisés pour assurer la conformité d'une architecture avec une politique. C'est ensuite à la charge de l'utilisateur de savoir quelle politique est la plus appropriée (en fonction des contraintes de son système, de son domaine d'application, etc.).

4.7.1 Implantation des mécanismes de sécurité

Cette règle de validation est réalisée en deux étapes :

1. l'assurance que les mécanismes d'implantation de chaque niveau de sécurité ont été définis (par exemple, l'utilisateur a-t-il spécifié les mécanismes de protection des donnés ?)
2. l'assurance que l'utilisateur a spécifié les caractéristiques nécessaires à l'analyse et l'implantation de ces mécanismes.

La première étape est réalisée par une analyse sémantique du modèle (listing 4.24) et assure que chaque niveau de sécurité (composant de type `virtual bus`) spécifie ses mécanismes de sécurité. La règle associée valide donc la bonne définition de la propriété `Implemented_As` sur chaque composant `virtual bus` et assure la bonne spécification du composant qui lui est ainsi associé.

```
Pour tout composant virtual bus niveauSecurite faire
   variable composant = composant associe a la propriete
                        "Implemented_As" de niveauSecurite;
   Verifier que
      ((la variable composant est definie)
       et que (composant contient un sous-composant send)
       et que (composant contient un sous-composant receive)
       et que (composant contient un sous-composant
               marshalling_type de type data)
      )
fin pour tout;
```

Listing 4.24 – Règle validant l'implantation d'un niveau de sécurité

La seconde étape analyse les composants `virtual bus` classifiés à des niveaux de sécurité différents (par exemple, deux composants `virtual bus`, l'un assurant le niveau de sécurité *top-secret*, l'autre le niveau *confidentiel*).

Deux niveaux de sécurité différents doivent offrir des mécanismes de protection distincts. Si tel n'est pas le cas, cela signifierait qu'une entité de faible niveau puisse lire ou écrire les données destinées à un autre niveau (par exemple, une entité secrète envoie des données à une entité top-secrète). Ce type de communication est interdit pour certains modèles de sécurité, tel MILS (qui doit assurer l'isolation de chaque niveau de sécurité).

Une règle de validation analyse chaque niveau de sécurité et vérifie l'existence de mécanismes de protection distincts (analyse des propriétés AADL des composants `virtual bus`). Cela assure alors qu'un niveau de sécurité donné ne peut lire ou écrire les données en provenance d'un autre niveau.

Cette règle est illustrée dans le listing 4.25 pour le protocole *Blowfish*. Elle analyse chaque composant `virtual bus` : lorsque ceux-ci sont classifiés à un niveau différent (valeur de la propriété `Security_Level`), elle vérifie que les mécanismes de protection spécifiés (algorithmes et clés de chiffrement) ne sont pas identiques.

Cette analyse du modèle assure la cohérence des mécanismes de protection

des données et assure qu'un certain niveau de sécurité n'aura pas accès aux données des autres à cause d'une erreur de spécification.

```
Pour tout composant virtual bus niveauSecurite faire
   liste autresNiveauxDeSecurite = composants
      virtual bus ayant une propriete "POK::Security_Level"
      differente de niveauSecurite;

   Pour tout composant autreNiveauDeSecurite faire
      Verifier que
      ((
         (propriete "POK::Blowfish_Key" de niveauSecurite) !=
         (propriete "POK::Blowfish_Key" de autreNiveauDeSecurite)
      )
      et
      (
         (propriete "POK::Blowfish_Init" de niveauSecurite) !=
         (propriete "POK::Blowfish_Init" de autreNiveauDeSecurite)
      ))
      ;
   fin pour tout;
fin pour tout;
```

Listing 4.25 – Validation de la cohérence des mécanismes de sécurité pour le protocole *Blowfish*

4.7.2 Politique de sécurité *Bell-Lapadula*

La validation de la politique de sécurité *Bell-Lapadula* doit vérifier qu'une partition :

- ne reçoit pas de données d'une partition classifiée à un un niveau de sécurité supérieur (règle *no read-up*).
- n'envoie pas de données à une partition d'un niveau de sécurité inférieur (règle *no write-down*).

Le listing 4.26 valide ces contraintes : pour chaque partition, la règle assure que le niveau de sécurité de la partition réceptrice est inférieur à celui de la partition émettrice. Elle inspecte chaque connection inter-partitions (connection entre composants `process`), analyse les niveaux de sécurité et compare leur classification (propriété `Security_Level` des composants `virtual bus` associés aux partitions, composant `virtual processor`).

```
Pour tout composant process processSource faire
   variable partitionSource      = composant virtual processor
                                   associe a prs;
   liste niveauxDeSecuriteSource = composants virtual bus
                                   associe a partition;

   liste processesDestination    = composants process connecte
                                   a ProcessSource;
   liste partitionsDestination   = composants virtual processor
                                   associes a processesDestination;
   liste niveauxDeSecuriteDestination = virtual bus associe
                                        a partitionsDestination;
```

```
   Verifier que
      (niveauxDeSecuriteSource <= niveauxDeSecuriteDestination);
fin pour tout;
```

Listing 4.26 – Validation de la politique de sécurite Bell/Lapadula

4.7.3 Politique de sécurité *Biba*

La validation de la politique de sécurité *Biba* requiert qu'une partition :
- ne reçoit pas de données d'une partition d'un niveau de sécurité inférieur (*no read-down*).
- n'envoie pas de données à une partition d'un niveau de sécurité supérieur (*no write-up*)

Tout comme pour la politique de sécurité *Bell-Lapadula*, une règle dédiée (listing 4.27) analyse chaque connection inter-partitions et assure que le niveau de sécurité de la partition réceptrice est supérieur à celui de la partition émettrice. Pour cela, elle examine les composants `virtual bus` et `virtual processor`.

```
Pour tout composant process processSource faire
   variable partitionSource      = composant virtual processor
                                   associe a prs;
   liste niveauxDeSecuriteSource = composants virtual bus associe
                                   a partition;

   liste processesDestination    = composants process connecte
                                   a ProcessSource;
   liste partitionsDestination   = composants virtual processor
                                   associes a processesDestination;

   liste niveauxDeSecuriteDestination = virtual bus associe
                                        a partitionsDestination;

   Verifier que
      (niveauxDeSecuriteSource >= niveauxDeSecuriteDestination);
fin pour tout;
```

Listing 4.27 – Validation de la politique de sécurité Biba

4.7.4 Conformité MILS

La règle associée à la validation des contraintes de MILS assure l'isolation de chaque niveau de sécurité et détecte donc les potentiels échanges entre niveaux de sécurité différents.

Cette règle, illustrée dans le listing 4.28 assure l'isolation des niveaux de sécurité. Elle analyse les connections entre partitions et assure que les partitions communiquant partagent le même niveau de sécurité.

```
Pour tout composant process processSource faire
   variable partitionSource      = composant virtual processor
                                   associe a prs;
   liste niveauxDeSecuriteSource = composants virtual bus associe
                                   a partition;
```

```
   liste processesDestination     = composants process connecte
                                    a ProcessSource;
   liste partitionsDestination    = composants virtual processor
                                    associes a processesDestination;

   liste niveauxDeSecuriteDestination = virtual bus associe
                                        a partitionsDestination;

   Verifier que
      (niveauxDeSecuriteSource == niveauxDeSecuriteDestination);
fin pour tout;
```

Listing 4.28 – Règle validant la politique de sécurite MILS

Cette règle de validation, alliée à celles définies précédemment (section 4.3.1), permet au concepteur d'un système de vérifier toutes les contraintes de MILS : isolation spatiale, isolation temporelle, confinement des niveaux de sécurité, etc.

Synthèse de ce chapitre

Ce chapitre a présenté nos patrons de modélisation pour la description d'architecture partitionnées via le langage AADL. Ils offrent en particulier la capacité d'exprimer les contraintes (isolation spatiale, temporelles, exigences de sécurité/sûreté) et les services/ressources (tâches, communications intra-partition et inter-partitions) de ce type d'architecture. Les patrons proposés dans ces travaux ont fait l'objet d'un travail au sein du comité AADL sous la forme d'une annexe au standard [34].
Dans un second temps, ce chapitre a introduit plusieurs règles de validation de modèles AADL. Ces dernières se divisent en trois groupes :

1. **validation du respect des patrons de modélisation**. Ces règles assurent que l'architecture spécifiée par l'utilisateur utilise correctement les patrons de modélisation décrits précédemment.
2. **validation de la politique de sûreté**. Ces règles vérifient que la politique de sécurité spécifiée par l'utilisateur permet de recouvrir toutes les fautes potentielles que nous proposons dans nos patrons de modélisation. Elles analysent les interactions possibles entre partitions classifiées à des niveaux de criticité différents et détectent les impacts potentiels entre partitions de faible criticité et partitions de haute criticité.
3. **validation de la politique de sécurité**. Ces règles assurent le respect de politiques de sécurité (Bell-Lapadula, Biba, MILS) et assure que l'utilisateur a correctement spécifié les mécanismes de protection de données (chiffrement, etc.).

Ces règles doivent être considérées comme un support pour le concepteur d'application, lui spécifiant comment doit être constituée la spécification de son système (composants et propriétés à utiliser). Cette description est réutilisée par des règles de validation, détectant les erreurs potentiellement introduites par l'utilisateur. Ces deux éléménts (patrons de modélisation, règles de validation) constituent une réponse au premier objectif fixé en introduction (section 1.4).
Une fois l'architecture validée, il est nécessaire d'implanter le système. Le chapitre suivant détaille comment il est possible de procéder à cette implantation automatiquement, à partir de ces modèles architecturaux.

Chapitre 5

Implantation et certification automatiques

Sommaire du chapitre

5.1 Aperçu du processus de génération 138
5.1.1 Objectifs du processus d'implantation 138
5.1.2 Objectifs du processus de certification 139
5.1.3 Bénéfices du processus d'implantation 139
5.1.4 Bénéfices du processus de certification 141
5.2 Chaîne de génération 141
5.2.1 Travaux existants 141
5.2.2 Détail du processus de génération 142
5.3 Patrons de génération de code 143
5.3.1 Composant `processor` 144
5.3.2 Composants `virtual processor` et `process` 148
5.3.3 Composant `device` 149
5.3.4 Composant `memory` 149
5.3.5 Composant `thread` 149
5.3.6 Composant `data` 151
5.3.7 Connections entre composants `thread` 152
5.3.8 Connections entre composants `process` 153
5.3.9 Intégration de code applicatif externe 155
5.3.10 Optimisation 156
5.3.11 Résumé des patrons de génération 158
5.4 Plate-forme pour systèmes sûrs et sécurisés 158
5.4.1 Description générale de POK 158
5.4.2 Architecture et services 158
5.4.2.1 Couche noyau 159
5.4.2.2 Couche partition (libpok) 160
5.4.2.3 Modularité et finesse de configuration . . . 162
5.4.3 Mécanismes de sûreté/sécurité 162
5.4.3.1 Isolation temporelle 162
5.4.3.2 Isolation spatiale 163
5.4.3.3 *Health Monitoring* 164
5.4.3.4 Algorithmes de chiffrement 165
5.4.4 Implantation des pilotes de périphérique 166
5.4.4.1 Modèles et choix d'implantation de POK . 166
5.4.4.2 Nouveaux services introduits 167
5.4.4.3 Implantation 167
5.5 Certification des exigences de la spécification . . . 167

5.5.1 Ordonnancement 168
5.5.1.1 Présentation de l'outil 169
5.5.1.2 Mise en pratique 169
5.5.1.3 Retour d'expérience 170
5.5.2 Analyse comportementale 171
5.6 Assurance des exigences des standards de certification 172
5.6.1 Contexte 173
5.6.2 Intégration avec le processus de développement . . . 173
5.6.3 Intérêt de la génération de code 174
5.6.4 Retours d'expériences 175

Rappels

Le chapitre précédent décrit des patrons de modélisation et des règles de validation dédiées aux systèmes partitionnés. Les patrons guident le concepteur du système en indiquant la bonne manière pour représenter le système et ses caractéristiques. Les règles de validation associées assurent que la spécification créée respecte bien les exigences d'une architecture partitionnée avec ses caractéristiques de sécurité/sûreté. Ces étapes garantissent l'absence d'erreur dans les spécifications du système, permettant de procéder à son implantation et sa certification.
Ce chapitre détaille le processus de génération et de certification à partir des spécifications validées. Il décrit les outils utilisés, et explicite les garanties apportées quant aux mécanismes de sécurité/sûreté implantés.

Objectifs du chapitre

Ce chapitre a un double objectif : (i) la description du processus de génération automatique de l'architecture de systèmes critiques et (ii) leur certification.
Dans un premier temps, il détaille le processus de génération de code à partir des spécifications, produisant automatiquement une implantation C de modèles AADL. Cette première partie du chapitre présente les patrons de génération de code qui transposent les constructions AADL vers C et leurs avantages (réduction de l'empreinte mémoire, prédictabilité du code, etc.).
Dans un second temps, ce chapitre présente la plate-forme AADL pour les applications sûres et sécurisées : POK [33, 32]. Celle-ci intègre le code généré et assure les propriétés de sécurité et de sûreté décrites par le concepteur du système (isolation spatiale/temporelle, confinement des communications, etc.).
Par ailleurs, nous proposons des outils d'analyse à l'exécution pour :

1. vérifier le comportement du système implanté par rapport aux spécifications, garantissant la conformité de l'implantation par rapport à la description de l'utilisateur.
2. assurer la garantie des exigences des standards de certifications (par exemple : couverture de code).

Enfin, ce chapitre met l'accent sur l'utilité de l'approche dirigée par les modèles : chaque étape du processus repose sur l'utilisation des patrons de modélisation proposés en section 4.

5.1 Aperçu du processus de génération

Les sections suivantes situent les deux étapes abordées dans ce chapitre par rapport à nos objectifs :

1. implantation automatique à partir des spécifications (second objectif de la thèse, section 1.4)
2. certification du système à l'exécution (troisième objectif de la thèse, section 1.4).

5.1.1 Objectifs du processus d'implantation

L'objectif de cette étape est de produire le système tout en garantissant le respect de ses spécifications. Pour parvenir à le satisfaire, il est nécessaire de lier spécifications et implantation.

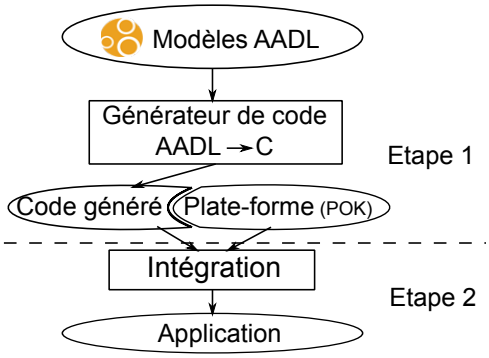


FIGURE 5.1 – Étapes d'implantation d'un système partitionné à partir de modèles AADL

Le processus proposé génère automatiquement l'implantation à partir de modèles AADL validés. Il repose sur deux parties, illustrées dans la figure 5.1 :

1. **une génération de code qui transpose les constructions AADL en code C (étape 1 de 5.1).** Cette étape génère toutes les constructions nécessaires à l'exécution du système spécifié et assure la liaison entre les composants (génération de code dit *« glue »*). Le code créé respecte ainsi les caractéristiques du modèle et les exigences du concepteur (en terme de sécurité, sûreté, performances, etc.).
2. **une intégration avec une une plate-forme d'exécution (étape 2 de 5.1.** Elle fournit les services nécessaires à l'exécution du système (isolation temporelle, spatiale, etc.). Elle doit être hautement configurable afin d'être configurée et déployée en fonction des directives de configuration dérivées des caractéristiques du modèle AADL (configuration des partitions, de la politique de sécurité, sûreté, etc.). Ainsi, le générateur de code pourra produire des directives de configuration afin d'adapter au mieux la plate-forme aux besoins décrits par le concepteur d'application.

Au début de ces travaux, aucune plate-forme de partitionnement libre n'existait. Le projet que nous avons initié, POK [33, 32] répond à cette demande. Il a pour objectif de faciliter la conception de systèmes sûrs et sécurisés en incluant :

1. un système d'exploitation partitionné comprenant les mécanismes de sécurité et de sûreté requis par les approches ARINC653 et MILS.
2. une suite d'outils facilitant le développement de systèmes sûrs et sécurisés à partir de modèles AADL.

5.1.2 Objectifs du processus de certification

La partie certification répond au troisième objectif de notre approche (section 1.4). Elle est mise en œuvre par un processus en deux étapes :

1. **CERTIF-SPEC** : certification de l'exécution du système par rapport aux spécifications.
2. **CERTIF-STD** : assurance du respect des contraintes des standards de certification (tel DO178B).

Ces contraintes de certification nécessitent la création d'outils qui mettent en relation l'exécution du système avec ses spécifications pour les comparer et assurer leur conformité.

Notre processus de certification est illustré en figure 5.2 : **CERTIF-SPEC** (partie gauche de la figure) s'appuie sur les modèles AADL utilisés par les étapes précédentes du processus de développement (validation, implantation). Ce type de certification compare exécution et spécifications au moyen de deux outils analysant les aspects suivants :

1. **ordonnançabilité** : vérification que les événements d'ordonnancement (activation des tâches, préemption, etc.) observés à l'exécution respectent les caractéristiques temporelles de la spécification. Pour cela, un outil enregistre les évênements d'ordonnancement (*context-switch*, changement de partition) lors de l'exécution du système et les compare avec les résultats provenant de la simulation de l'ordonnancement réalisée à partir des spécifications.
2. **comportementales** : analyse de la conformité des fonctions exécutées par le système.

CERTIF-STD (partie droite de la figure 5.2) apporte des garanties sur le système généré indépendamment de ses spécifications. Il a pour but de vérifier que les systèmes produits respectent les exigences des standards de certification (comme la couverture de code requise par DO178B).

5.1.3 Bénéfices du processus d'implantation

Le processus d'implantation traduit automatiquement les spécifications de l'architecture (modèles AADL) en code C intégré à une plate-forme d'exécution. Il apporte les bénéfices suivants :

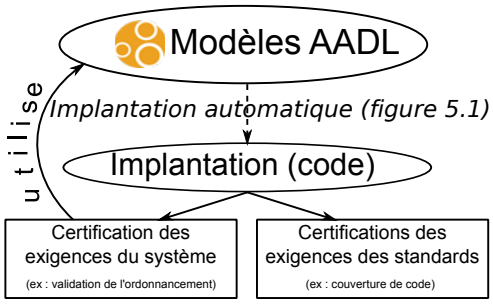


FIGURE 5.2 – Illustration de l'utilisation des différents outils de certification

1. assurance du respect des spécifications ;
2. déterminisme dans la production de code, facilitant ainsi l'analyse de l'application produite ;
3. absence d'erreur dans le code produit, par la suppression du caractère *« manuel »* de la production de code.

L'assurance du respect des spécifications garantit que les caractéristiques du système sont correctement implantées. Dans un processus de conception traditionnel, rien ne garantit leur respect : les développeurs transposent manuellement la description du concepteur en code et peuvent donc introduire des erreurs. Le recours à un processus de transformation automatique de code assure que les constructions du langage source (modèles AADL) seront correctement transposées dans l'implantation.

Le processus de génération facilite l'analyse du code par le recours à des patrons de modélisation prédéfinis. Ces derniers définissent les règles de transformation de code et ses caractéristiques (appels de fonctions, introduction de ressources supplémentaires tels que les variables, tâches, etc.). Ainsi, le structure du code est prédéterminée, facilitant son analyse et la garantie du respect d'exigences (par exemple : temps d'exécution nécessaire à l'envoi des messages, *overhead* mémoire introduit par le code généré, temps d'exécution des fonctions, etc.). Ces analyses s'appuient traditionnellement sur des outils indépendants et dédiés qui inspectent l'implantation (consommation mémoire, temps d'exécution, etc.) et reportent les métriques à l'utilisateur qui estime si les résultats sont conformes aux exigences de production. A l'inverse, notre processus à partir de patrons de génération prédéterminés, permet de déduire le coût (temporel, mémoire, etc.) du code produit et affranchit l'utilisateur de procéder à de coûteuses analyses de l'implantation.

La génération supprime le facteur d'erreur humain inhérent à la production de code traditionnelle. La traduction automatique de code à partir des spécifications assure leur bonne implantation et limite les erreurs traditionnellement introduites par la création manuelle de code. Cependant, cela suppose que le générateur ne comporte lui-même aucune erreur. L'utilisation d'une telle approche dans un contexte industriel nécessitera alors que

le générateur lui-même soit certifié.

5.1.4 Bénéfices du processus de certification

Le processus de certification proposé apporte les bénéfices suivants :

1. assurance de la conformité de l'exécution avec les spécifications
2. garantie automatique des exigences des standards de certifications
3. processus non intrusif, automatique

La vérification de la conformité de l'exécution avec les spécifications assure le respect des exigences décrites par l'utilisateur. De telles validations reposent traditionnellement sur une analyse ou une instrumentation du code source. Les ingénieurs modifient donc le système pour superviser son comportement et le comparer avec les spécifications. A l'inverse, le processus proposé introduit des outils ne requérant aucune modification de l'implantation. Par exemple, la vérification de l'ordonnancement vérifie que la plateforme d'exécution exécute les tâches conformément aux simulations réalisées à partir des spécifications sans pour autant modifier son ordonnanceur.

La garantie des exigences des standards permet de qualifier automatiquement les systèmes produits. Les processus de certification traditionnels requièrent une analyse et une relecture du code des applications. Celui-ci doit par ailleurs être modifié afin de respecter les contraintes des standards de certification. Le processus proposé permet de vérifier ces contraintes automatiquement à partir des applications générées. Par exemple, les outils d'analyse de couverture de code déduisent automatiquement le taux de couverture et le niveau de qualification des applications générées.

Le processus proposé est non-intrusif et ne requiert pas de modification de l'application produite. Il procède à son analyse dans des conditions similaires à son mode opérationnel. A l'inverse, les outils de certification traditionnels nécessitent une instrumentation manuelle du code n'assurant pas la stricte conformité du système testé et du système en production. Les outils que nous proposons, de par leur caractère non-intrusif (aucune modification du code n'est nécessaire), assure que le système testé correspondant à celui qui sera mis en exploitation.

5.2 Chaîne de génération

Les sections suivantes donnent un aperçu des travaux existants et présentent les outils utilisés et améliorés dans le cadre de nos travaux.

5.2.1 Travaux existants

L'approche de génération de code présentée dans ce manuscrit se base sur les travaux menés au sein de notre laboratoire. Ces derniers portent sur la traduction de modèles AADL en code C [36] ou Ada [116]. Les applications

ainsi produites sont exécutées sur des systèmes d'exploitation profilés pour l'embarqué tel RTEMS [79].

L'accent a été mis sur le déterminisme et la réduction de l'empreinte mémoire : les patrons de génération transposent chaque composant AADL en code C ou Ada en fonction de ses propriétés. Par exemple, le patron de génération associé aux composants de type `thread` produit le code nécessaire pour créer ses tâches en accord avec ses contraintes temporelles (par exemple, propriétés `Period` ou `Deadline` du composant).

Le code généré s'exécute sur une plate-forme dédiée. Dans le cadre de ces travaux, des bibliothèques d'abstraction des systèmes d'exploitation Linux et RTEMS (PolyORB-HI-C [31] et PolyORB-HI-Ada [116]) ont été définies. Elles interfacent et intègrent le code généré avec les fonctions proposés par l'environnement d'exécution.

Enfin, notons que ces travaux ont été intégrés dans les projets de recherche ASSERT [42, 61] et Flex-eWare [50, 21, 36]. Ils ont été utilisés comme support d'exécution pour l'intergiciel MyCCM-HI [21], Framework à composant pour systèmes critiques réalisé par la société THALES.

Cependant, si cette approche de génération présentait plusieurs des caractéristiques décrites dans la section 5.1.3 (patrons déterministes de génération, réduction des erreurs), elle se heurte toutefois à deux limites :

1. le code produit n'apporte aucune garantie de sûreté/sécurité et s'appuie sur des systèmes d'exploitation traditionnels, ne pouvant assurer ces propriétés. Cet aspect ne convient pas au second objectif de nos travaux (section 1.4). Il est donc nécessaire d'améliorer les travaux existants en vue d'apporter des garanties quant à la sécurité et la sûreté.
2. aucune certification du code généré n'est réalisée. Cependant, l'application produite doit être certifiée afin d'assurer que le processus de génération n'introduit pas d'erreur. Cette limite montre que cette approche doit être améliorée pour répondre au troisième objectif fixé (section 1.4).

Nous avons donc amélioré les travaux existants pour atteindre nos objectifs par :

1. la définition de patrons de génération de code utilisant les nouveaux composants de la version 2 d'AADL pour la sécurité et la sûreté (tels les `virtual processor` et les `virtual bus`).
2. la conception d'un environnement d'exécution supportant le code généré et comprenant des services de sécurité/sûreté.
3. réalisation d'outils de certification de l'implantation.

5.2.2 Détail du processus de génération

Le processus de génération de code se divise en plusieurs étapes :

1. analyse du modèle AADL
2. recherche du composant `system` racine
3. parcours des composants AADL et application des patrons de génération produisant du code C
4. écriture du code généré

Au cours de l'analyse du modèle AADL, le générateur vérifie sa conformité. Cette phase assure que le concepteur n'a pas introduit d'erreur syntaxique (erreur dans l'utilisation du langage) ou sémantique (par exemple, utilisation de composants non autorisés par le langage ou impliquant des erreurs dans les spécifications).

La recherche du système racine détecte le composant AADL `system` contenant l'architecture à générer. Un modèle AADL peut en effet contenir plusieurs composants `system` et il est nécessaire de choisir celui de plus haut niveau. Ocarina peut le détecter automatiquement ou laisser l'utilisateur le spécifier.

Le parcours du modèle s'effectue à partir du composant `system` racine : le générateur inspecte la hiérarchie de composants et applique ses patrons de génération de code. Ces derniers produisent alors du code C qu'ils stockent dans un AST [1]. Cette étape est similaire à une compilation, le langage AADL étant ici le langage source, le C étant le langage cible.

Une fois les patrons de génération de code appliqués, et l'intégralité du code C contenu dans son AST, le générateur écrit donc ce code sur le système de fichiers, créant ainsi les fichiers sources constituant l'application.

Pour parvenir à l'objectif de génération de systèmes sûrs et sécurisés (second objectif identifié dans la section 1.4), nous avons introduit de nouveaux patrons de génération de code dédiés aux systèmes partitionnés au sein d'Ocarina.

5.3 Patrons de génération de code

Les sections suivantes présentent nos règles de génération de code pour chaque composant AADL utilisé par nos patrons de modélisation (chapitre 4). Pour chacun d'eux, nous détaillons les propriétés AADL réutilisées pour implanter et configurer le système.

Définition : Patron de génération

Un patron de génération est un ensemble de règles qui transforment des constructions d'un langage source en des blocs de code équivalents dans un langage cible. Ce processus doit conserver la sémantique du langage. Dans le cadre de nos travaux, les patrons de génération transposent les composants de modèles AADL en code C.

Fonctionnement des patrons

La figure 5.3 illustre le fonctionnement d'un patron de génération pour le composant `thread`. Il définit une base de code fixe et une base de code variable

1. voir lexique

(écrite en majuscule). Lorsque le générateur de code analyse le composant, il reproduit le code et adapte ses parties variables en fonction des propriétés du composant AADL. Par exemple, la période et l'échéance (deadline) du `thread` sont transposées dans le code généré. Si des propriétés ne sont pas définies, le générateur utilise des valeurs par défaut (c'est le cas ici pour la taille de la pile).

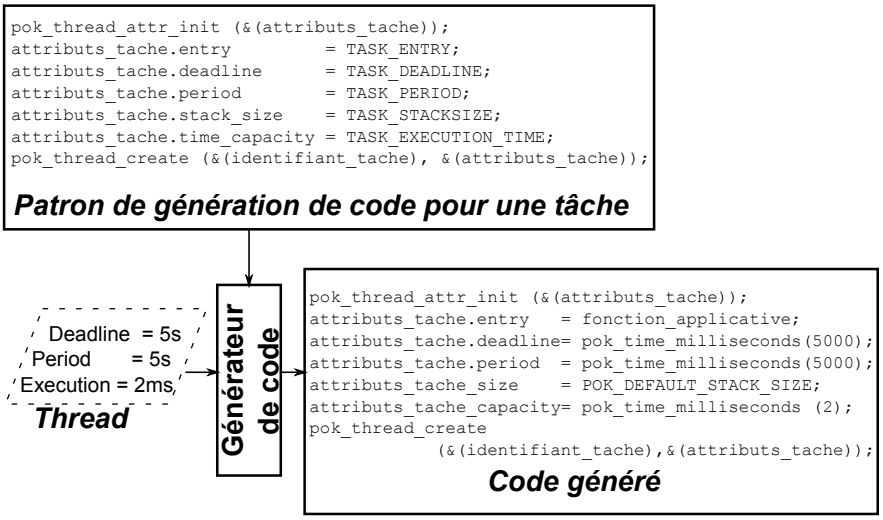


FIGURE 5.3 – Application d'un patron de génération de code sur un composant AADL `thread`

Description des patrons de génération Tout comme pour les patrons de modélisation du chapitre 4, ceux dédiés à la génération de code sont présentés au moyen de tableaux. Un exemple est illustré dans le tableau 5.1 : il décrit, pour chaque composant AADL les règles de transformation (R1 et R2) produisant du code C. Ces patrons s'appuient sur les règles de modélisation décrites précédemment (chapitre 4) et les référencent par leur identifiant et leur exigence ($\mathrm{PM}^{nom-patron}\mathrm{E}_n$) . Ainsi, l'exigence codifiée PM^{noyau}E2 désigne la seconde exigence du patron de modélisation associée au noyau.

Ces patrons de génération sont également identifiés par un symbole de la forme suivante : $\mathrm{PG}^{nom-patron}$ (**P**atron **G**énération) afin de faciliter leur référence.

5.3.1 Composant `processor`

Le patron de génération associé au composant `processor` est illustré dans le tableau 5.2. Il configure le noyau et les partitions qu'il exécute.

La première règle de génération (R1) décrit les partitions contrôlées par le noyau et alloue leurs ressources. En outre, il permet de générer les structures de données utilisées par les services du noyau (ordonnanceur, mécanismes de communication, etc.).

Exemple de patron de génération ($PG^{nom-patron}$)	
Entité AADL	**Code C**
R1 : Propriété $PM^{nom-patron}$ R2 : Sous-composant de type `thread` ($PM^{nom-patron}$)	Création d'une variable/tableau Appel de fonction

TABLE 5.1 – Exemple de patron de génération

La génération de la politique d'isolation temporelle est mise en œuvre par les règles R2, R3 et R4 : les valeurs des tranches de temps et leur allocation aux partitions sont déduites des propriétés AADL `Slots` et `Slots_Allocation`. En leur absence, le générateur rapporte une erreur à l'utilisateur (aucune valeur par défaut). Les listings 5.1 et 5.2 illustrent l'utilisation de ce patron. Dans ce modèle, le noyau de partitionnement exécute (propriétés `Partitions_Slots` et `Slots_Allocation`) la première partition pendant 150ms et la seconde pendant 50ms. La *Major_Frame* (propriété `Module_Major_Frame`) est définie à 200ms. Le code généré (listing 5.2) traduit ces exigences par des directives de configuration :

- la macro `POK_CONFIG_SCHEDULING_NB_SLOTS` définit le nombre de tranches de temps allouées au sein du noyau (correspondant au nombre de valeurs contenues dans la propriété `Partitions_Slots`).
- la macro `POK_CONFIG_SCHEDULING_SLOTS` définit la valeur des tranches de temps (valeurs de la propriété `Partitions_Slots`).
- la macro `POK_CONFIG_SCHEDULING_SLOTS_ALLOCATION` spécifie l'allocation des tranches de temps entre les partitions. Ici, la première tranche de temps (150ms) est allouée à la première partition et la seconde (50ms), à la seconde partition (propriété `Slots_Allocation`).
- la macro `POK_CONFIG_SCHEDULING_MAJOR_FRAME` définit la *Major Frame* (propriété `Module_Major_Frame`) associée au noyau (période à laquelle la politique d'ordonnancement est répétée).

```
processor implementation mycpu.impl
subcomponents
   Environnement1 : virtual processor partition.impl;
   Environnement2 : virtual processor partition.impl;
properties
   ARINC653::Module_Major_Frame => 200ms;
   ARINC653::Partition_Slots => (150ms, 50ms);
   ARINC653::Slots_Allocation =>
      (reference (Environnement1), reference (Environnement2));
end mycpu.impl;

memory implementation MemoirePrincipale.impl
subcomponents
   Segment1: memory partitionmemory {Byte_Count => 85000;};
   Segment2: memory partitionmemory {Byte_Count => 98387;};
end MemoirePrincipale.impl;

system implementation node.impl
```

Noyau de partitionnement : composant `processor` (PG^{noyau})	
Entité AADL	**Code C**
Contrôle des partitions : sous-composants `virtual processor` ($PM^{noyau}E1$)	R1 : Macro définissant le nombre de partitions exécutées par le noyau (`POK_CONFIG_NB_PARTITIONS`)
Définition de la *Major Frame* : propriété `Major_Frame` ($PM^{noyau}E2$)	R2 : Macro contenant la valeur de la Major Frame convertie en millisecondes (`POK_CONFIG_SCHEDULING_MAJOR_FRAME`)
Allocation de tranches de temps : propriété `Slots` ($PM^{noyau}E2$)	R3 : Macro définissant la liste des tranches de temps converties en millisecondes (`POK_CONFIG_SCHEDULING_SLOTS`)
Assignation des tranches de temps aux partitions : propriété `Slots_Allocation` ($PM^{noyau}E3$)	R4 : Macro définissant l'allocation des tranches de temps aux partitions (`POK_CONFIG_SCHEDULING_SLOTS_-ALLOCATION`)
Spécification des segments mémoire et association aux partitions ($PM^{isolation-spatiale}$E1, E2 et E3)	R5 : Macro décrivant la taille de chaque segment (convertie en octets) `POK_CONFIG_PARTITIONS_SIZE`
Déclaration des erreurs : propriété `HM_Errors` ($PM^{fautes}E3$)	R6 : Génération d'une fonction traitant chaque erreur déclarée (`pok_kernel_error`).
Déclaration des actions de recouvrement : propriété `HM_Actions` ($PM^{actions}E3$)	R7 : Génération d'un appel aux sous-programmes de recouvrement dans la fonction `pok_kernel_error`.

TABLE 5.2 – Patron de génération du noyau de partitionnement

```
subcomponents
   cpu    : processor mycpu.impl;
   part1  : process   partition1.impl;
   part2  : process   partition2.impl;
   mem    : memory    MemoirePrincipale.impl;
properties
   Actual_Processor_Binding =>
      (reference (cpu.Environnement1)) applies to part1;
   Actual_Processor_Binding =>
      (reference (cpu.Environnement2)) applies to part2;
   Actual_Memory_Binding    =>
      (reference (mem.Segment1)) applies to part1);
   Actual_Memory_Binding    =>
      (reference (mem.Segment2)) applies to part2);
end node.impl;
```

Listing 5.1 – Définition d'un système partitionné comprenant deux partitions

```
#define POK_CONFIG_PARTITIONS_SIZE                {85000,98387}
#define POK_CONFIG_SCHEDULING_SLOTS               {150,50}
#define POK_CONFIG_SCHEDULING_SLOTS_ALLOCATION    {0,1}
#define POK_CONFIG_SCHEDULING_NBSLOTS             2
#define POK_CONFIG_SCHEDULING_MAJOR_FRAME         200
```

Listing 5.2 – Politique d'isolation temporelle et spatiale générée

La configuration de l'isolation spatiale est mise en œuvre par la règle R5 qui configure les segments mémoires de chaque partition. Leur taille et leur déploiement sont déduits du composant `memory` associé à chaque partition (et de ses propriétés). Les listings 5.1 et 5.2 illustrent la transposition des exigences mémoire des partitions en code de configuration C. La première partition requiert un segment d'une taille de 85000 octets tandis que celui associé à la seconde partition a une taille de 98387 octets (sous-composants `Segment1` et `Segment2` du composant `MemoirePrincipale`). Le code généré dans le listing 5.2 contient alors une directive de configuration (macro `POK_CONFIG_PARTITIONS_SIZE`) qui définit la taille du segment alloué à chaque partition.

La politique de traitement des fautes (*Health-Monitoring*) est générée par les règles R6 et R7. Elles analysent les propriétés `HM_Errors` et `HM_Actions` et génèrent une fonction qui appelle les actions de recouvrement en fonction de l'erreur détectée par le noyau.

Les listings 5.3 et 5.4 illustrent la mise en œuvre des règles R6 et R7. Le noyau de partitionnement modélisé (composant `mycpu.i`, listing 5.3) s'arrête lors d'une erreur d'initialisation et il redémarre lors d'une faute d'ordonnancement des partitions. Le patron génère une fonction C (listing 5.4) qui exécute la procédure de recouvrement associée à chaque erreur spécifiée.

```
processor implementation mycpu.i
subcomponents
  -- Ajout des plates-formes des partitions (virtual processors)
properties
  -- Definition des autres proprietes (ordonancement ...)
   ARINC653::HM_Errors          => (Kernel_Init, Kernel_Scheduling);
   ARINC653::Recovery_Actions => (Kernel_Stop, Kernel_Restart);
end mycpu.i;
```

Listing 5.3 – Définition d'un composant `processor` avec sa politique de reprise d'erreurs

```
void pok_kernel_error
    (uint32_t error)
{

  switch (error)
  {
    case POK_ERROR_KIND_KERNEL_INIT:
    {
      pok_kernel_stop ();

      break;
    }
    case POK_ERROR_KIND_KERNEL_SCHEDULING:
    {
      pok_kernel_restart ();

      break;
    }
  }
```

}

Listing 5.4 – Code de reprise d'erreur généré

5.3.2 Composants `virtual processor` et `process`

Le patron associé au composant AADL `virtual processor` est défini dans le tableau 5.3.

Partition : composants `virtual processor` et `process` ($PG^{partition}$)	
Entité AADL	**Code C**
Génération des ressources de la partition : sous-composants du `process` ($PM^{partition}E1$)	R1 : Génération des tâches et données de la partition (voir patrons 5.3.5 et 5.3.6) et dimenssionnement des ressources par la définition de macros (`POK_CONFIG_NB_TASKS`, etc.)
Confinement de la partition : propriété `Actual_Processor_Binding` ($PM^{partition}E2$)	R2 : Association des ressources générées à un environnement d'exécution
Politique d'ordonnancement de la partition : propriété `Scheduling_Protocol` ($PM^{noyau}E3$)	R3 : Définition d'une macro spécifiant la politique d'ordonnancement de chaque partition (`POK_CONFIG_PARTITIONS_SCHEDULER`)
Description des fonctions requises pour le code applicatif : propriété `Additional_Features` ($PM^{noyau}E3$)	R4 : Définition de macros configurant l'environnement d'exécution pour l'inclusion de fonctions supplémentaires (macros `POK_NEEDS_...`). Par exemple, l'inclusion de la bibliothèque mathématique nécessaire pour l'inclusion de code Simulink définit la macro `POK_NEEDS_LIBMATH`.
Déclaration des erreurs : propriété `HM_Errors` ($PM^{fautes}E2$)	R5 : Génération d'une fonction de recouvrement gérant chaque faute décrite (`pok_partition_error`).
Déclaration des actions de recouvrement : propriété `HM_Actions` ($PM^{actions}E2$)	R6 : Génération d'appels aux fonctions de recouvrement dans la fonction `pok_partition_error`.

Table 5.3 – Patron de génération des partitions

Le processus de configuration alloue statiquement et dimensionne les ressources utilisées par la partition (règle R1). Leur nature et leur nombre est déduit de ses sous-composants (`thread` et `data`), de ses interfaces (`ports`) et de leurs propriétés. Le patron génère des directives de configuration spécifiant les ressources requises et leur nombre.

La règle de génération R2 associe les ressources à la partition. En outre, elle assure la bonne allocation des ressources à chaque partition en fonction des exigences du modèle. Ainsi, tous les composants contenus dans

le `process` (tâches, ports de communication, données partagées, etc.) de la partition sont associés à son environnement d'exécution.

La politique d'ordonnancement est générée à partir de la propriété AADL `Scheduling_Protocol` (règle R3). Sa valeur AADL est convertie en une directive de configuration (macro C) définie dans notre plate-forme d'exécution AADL. Celle-ci est ensuite réutilisée par l'environnement d'exécution pour choisir l'algorithme approprié pour exécution les tâches de la partition.

La règle de génération R4 configure les fonctions applicatives requises par la définition de macros incluant des fonctions au sein de la plate-forme d'exécution de chaque partition. Ces macros sont déclarées en fonction de la propriété `Additional_Features` (fonction explicitement demandée par le concepteur du modèle) ou par une analyse des sous-composants de la partition (par exemple, une partition exécutant un sous-programme généré à partir de Simulink aura besoin des bibliothèques mathématiques). Cette détection des dépendances applicatives assure que toute fonction requise sera inclue, évitant tout problème à la compilation ou à l'édition de lien.

La politique de détection et reprise des fautes (*Health Monitoring*) est générée (règles R5 et R6) à partir des valeurs des propriétés `HM_Actions` et `HM_Errors` associées au composant `virtual processor`. Le patron produit alors une fonction qui exécute les fonctions de recouvrement associées à chaque faute.

5.3.3 Composant `device`

Dans notre approche, les pilotes de périphériques sont exécutés dans une partition dédiée. Le composant AADL `device` est traduit comme une partition système : le logiciel qu'elle exécute gère le périphérique modélisé.

Par conséquent, nous appliquons à ce composant le même patron de modélisation qu'aux partitions afin de créer toutes les ressources nécessaires à l'exploitation du périphérique spécifié. Cependant, le générateur réalise une passe de configuration supplémentaire et adapte le noyau pour que la partition puisse accéder au matériel qu'elle contrôle (bus PCI, *entrées/sorties*, etc.).

5.3.4 Composant `memory`

Le composant AADL `memory` représente les segments associés aux partitions et spécifie la politique d'isolation spatiale (section 4.2.3). Ce composant est utilisé par la règle de génération R5 associée au noyau de partitionnement (section 5.3.1).

5.3.5 Composant `thread`

Le patron de génération associé aux composants de type `thread` (section 4.2.5) est illustré dans le tableau 5.4.

Tâche : composant `thread` (PG^{tache})	
Entité AADL	**Code C**
Exécution de code applicatif : appels de composants de type `subprogram` ($PM^{tache}E1$)	R1 : Génération des appels de fonctions correspondant aux composants AADL `subprogram`.
Description des contraintes temporelles : propriétés `Deadline`, `Period`, etc. ($PM^{tache}E2$)	R2 : Génération de la fonction de la création de la tâche avec les propriétés du modèle.
Description des contraintes mémoire : propriétés `Source_Stack_Size` ($PM^{tache}E3$)	R3 : Appel d'une fonction définissant la taille de la tâche à son initialisation.
Déclaration des erreurs : propriété `HM_Errors` ($PM^{fautes}E1$)	R4 : Génération du handler d'exception associé à la tâche.
Déclaration des actions de recouvrement : propriété `HM_Actions` ($PM^{actions}E1$)	R5 : Génération des appels de fonctions correspondant aux politiques de recouvrement.

TABLE 5.4 – Patron de génération des composants `thread`

La règle de génération R1 crée une fonction périodique exécutée par la tâche et appelant les sous-programmes qu'elle contient (section `calls` du composant AADL `thread`). Le code de cette fonction est rythmé par trois étapes :

1. réception des données nécessaires au sous-programme (déduit des interfaces du `thread`)
2. exécution du code applicatif (déduit des appels aux composants `subprogram` — section `calls` du `thread`)
3. émission des données produites par le sous-programme (déduit des interfaces du `thread`).

Ce patron est illustré dans la figure 5.4 qui définit trois tâches (`thr0`, `thr1` et `thr2`) communiquant entre elles. Le code généré pour la tâche `thr0` ne réceptionne pas de données car sa spécification n'a aucune connection entrante. La tache `thr1` reçoit et émet des données : le composant correspondant a des connections entrantes avec `thr0` et sortantes avec `thr2`. Enfin, le composant `thr2` n'est que récepteur.

Le générateur instancie la tâche à l'initialisation de la partition (règle R2). Pour cela, il utilise les propriétés d'ordonnancement de la tâche pour assurer une création conforme aux spécifications (période, échéance, etc.).

Par le même mécanisme, la règle de génération R3 assure que les exigences mémoire sont transposées dans le code. La valeur de la propriété `Source_Stack_Size` du composant AADL est réutilisée pour configurer la tâche et assurer le bon dimensionnement de sa pile.

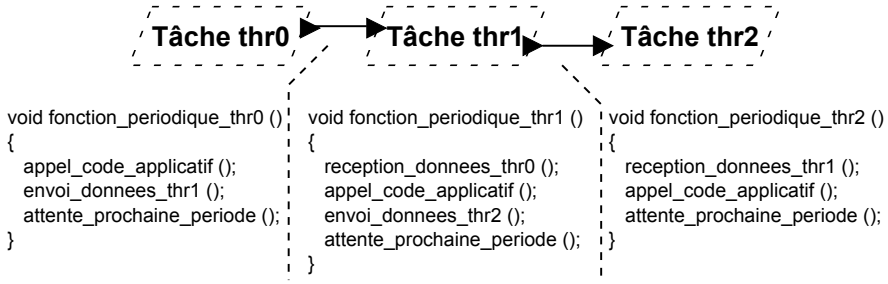


FIGURE 5.4 – Pseudo-code des fonctions périodiques pour trois tâches communiquant entre elles

La politique de gestion et de recouvrement des erreurs (*Health Monitoring*, section 4.4.3) est mise en œuvre par les règles de génération R4 et R5. La règle R4 crée une fonction qui énumère chaque erreur décrite par la propriété `HM_Errors` alors que la règle R5 génère les appels aux actions de recouvrement associées à chaque erreur.

5.3.6 Composant `data`

Le composant AADL `data` modélise l'accès à une donnée partagée entre les tâches d'une même partition. Le patron de génération associé à ce composant est illustré dans le tableau 5.5.

Communication intra-partition (*sémaphore*), composant `data` (PG^{data})	
Entité AADL	**Code C**
Donnée partagée entre plusieurs tâches ($PM^{intra-partition}E9$)	– R1 : Déclaration d'une variable globale et d'un sémaphore dans la partition ; – R2 : Instantiation d'un sémaphore pour son partage.
Politique d'attente : propriété `Concurrency_Control_Protocol` ($PM^{intra-partition}E10$)	R3 : Configuration des paramètres de la fonction de création du sémaphore : spécification du paramètre décrivant la politique d'attente.
Temps d'attente maximal sur la ressource partagée : propriété `Timeout` ($PM^{intra-partition}E6$)	R4 : Configuration des paramètres de la fonction utilisant le sémaphore, ajout du temps maximal d'attente.

TABLE 5.5 – Patron de génération des sémaphores

Ce patron génère une variable globale correspondant à la donnée (règle R1) et l'associe à un sémaphore (règle R2). L'allocation d'une telle ressource pour chaque donnée partagée garantit l'exclusion mutuelle des

tâches concurrentes pouvant manipuler la variable et assure donc la cohérence de la donnée.

La règle de génération R3 utilise la propriété `Concurrency_Control_Protocol` associée au composant `data`. Elle configure l'application pour que le sémaphore associé à la donnée partagée utilise le protocole de verrouillage spécifié par le concepteur d'application.

Enfin, la règle de génération R4 assure que le temps maximal d'attente de la donnée est respecté dans le code : à chaque tentative d'accès de la variable partagée, le temps d'attente (propriété `Timeout` du composant `data`) maximal sera réutilisé pour accéder au sémaphore.

5.3.7 Connections entre composants `thread`

La connection d'interfaces entre composants `thread` modélisent les communications intra-partition, chaque type de port (`data`, `event data` et `event`) représentant un service de spécifique (*buffer*, *event* ou *blackboard*, section 4.2.6).

Communications intra-partition (*buffer*, *blackboard*, *events*) : connections entre `threads` ($PG^{intra-partition}$)	
Entité AADL	**Code C**
Connection de ports entre composants de type `thread` ($PM^{intra-partition} E1, E5, E7$)	– R1 : Génération de tableaux définissant les connections entre les ports (politique de routage). – R2 : Instantiation des communications (génération d'appels de fonctions à l'initialisation des partitions).
Taille des données (composant data associé aux ports connectés) ($PM^{intra-partition} E2$)	R3 : Allocation statique de tampons de réception pour stocker les données à envoyer/recevoir.
Temps d'attente maximal d'attente : propriété `Timeout` ($PM^{intra-partition} E6$)	R4 : Ajout d'un paramètre aux fonctions d'envoi et réception de données spécifiant le temps maximal d'attente.
Taille de la file de réception : propriété `Queue_Size` ($PM^{intra-partition} E3 et E8$)	R5 : Configuration de la file de réception à la création des *buffer* : déclaration du nombre de données pouvant être stockées.
Politique de mise en file : propriété `Queue_Processing_Protocol` ($PM^{intra-partition} E4$)	R6 : Configuration de la politique d'attente (paramètre de la fonction de création) des canaux *events* et *queuing*.

TABLE 5.6 – Patron de génération des communication intra-partition

La règle R1 génère des tableaux décrivant les connections entre les ports de communications (canaux de communication). Ils sont

réutilisés par la plate-forme pour copier les données d'un port émetteur à un port récepteur.

La règle R2 crée les appels de fonctions instantiant les ports de communication à l'initialisation de la partition. Ces derniers utilisent des paramètres dérivés des propriétés associées au port de communication : taille de la file de réception (règles E3 et E8), temps maximal d'attente d'une donnée (règle R4) ou politique d'attente (règle R6). Enfin, la règle R2 insère ce code dans la procédure d'initialisation de la partition, assurant que tout canal est alloué avant le démarrage des tâches.

L'allocation statique de ressources (règle R3) associées aux communications. Le recours à une allocation dynamique induirait du temps d'exécution supplémentaire difficilement prédictible. L'utilisation de ce patron facilite donc l'analyse de l'application, supprimant un facteur d'indéterminisme.

5.3.8 Connections entre composants `process`

La connection de `port` entre composants `process` modélise une communication inter-partitions (section 4.2.7). Le patron de génération associé à ces entités AADL est illustré dans le tableau 5.7.

La règle R1 génère des tableaux décrivant l'inter-connection entre les partitions. Ces données sont réutilisées par la plate-forme afin d'établir les communications entre partitions. Ces dernières sont responsables du bon routage des données conformément au modèle.

Les communications sont initialisées par des appels de fonctions insérées à l'initialisation de la partition (règle R2). Leurs paramètres réutilisent les propriétés du port de communication : taille de la file de données (`Queue_Size`, R4), politique de réception (`Queue_Processing_Protocol`, R5) de la fréquence d'arrivée des données (`Refresh_Period`, R7) ou du temps maximal d'attente d'une donnée (`Timeout`, R6). La génération de ces appels à l'initialisation de la partition garantit que les canaux sont créés avant le démarrage des tâches et assure leur disponibilité. De plus, cette caractéristique supprime un facteur d'indéterminisme temporel : une création d'un canal de communication à l'exécution consomme du temps processeur difficilement prédictible et pouvant perturber l'exécution des tâches.

L'allocation statique des tampons requis pour l'émission/réception des données (règle R3) garantit la réservation de la mémoire nécessaire à toutes les communications. De plus, elle facilite l'analyse de l'application générée : le recours à une allocation dynamique entraînerait un facteur d'indéterminisme (temps de réservation de la mémoire, etc.).

Le patron de génération produit le code nécessaire au chiffrement des données (règles R8, R9 et R10). Si le port est associé à un niveau de sécurité (composant `virtual bus`, section 4.6.1), ses mécanismes de protection sont automatiquement appelés par la partition. La règle R8 ajoute des appels de fonctions avant tout envoi sur un port utilisant des mécanismes de sécurité. Les sous-programmes de chiffrement sont appelés, transformant ainsi la donnée avant son envoi. La règle R9 quant à elle décrypte les données reçues sur un port utilisant de tels mécanismes et appelle les sous-programmes pour déchiffrer les données reçues.

Enfin, la règle de génération R10 configure les protocoles de chiffrement en fonction des propriétés AADL associées aux mécanismes de sécurité (clés, données d'initialisation, etc.) par la définition de macros.

Le fonctionnement de ce patron de génération est illustré dans les listings 5.5 et 5.6. Le modèle (listing 5.5) définit une partition contenant un `port` associé au niveau de sécurité *topsecret* (composant `virtual bus topsecret.i`). Ce dernier spécifie ses mécanismes de chiffrement conformément au patron de modélisation (section 4.6.2).

Le code généré (listing 5.6) configure les protocoles de chiffrement via des directives de configuration spécifiques (macros `POK_PROTOCOLS_BLOWFISH_INIT` et `POK_PROTOCOLS_BLOWFISH_KEY`). Il ajoute un appel de fonction (`pok_protocols_blowfish_marshall`) qui transforme la donnée à envoyer en clair (`valeur_clair`) en une donnée chiffrée (`valeur_chiffree`). Ainsi, la valeur *en clair* est confiné dans l'espace mémoire de la partition, seule la valeur chiffrée est envoyée à l'extérieur.

Cette caractéristique du patron assure l'isolation des données en garantissant que les mécanismes de chiffrement sont toujours invoqués et qu'aucune donnée en clair ne circule sur des médiums de communication partagés et potentiellement non fiables.

```
abstract implementation vbus_blowfish_wrapper.i
subcomponents
   send              : subprogram blowfish_send.i;
   receive           : subprogram blowfish_receive.i;
   marshalling_type  : data blowfish_data.i;
end vbus_blowfish_wrapper.i;

virtual bus implementation topsecret.i
properties
   POK::Security_Level => 3;
   Implemented_As => classifier (vbus_blowfish_wrapper.i);
   POK::Blowfish_Key =>
      "{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,0xf0,0xe1,
        0xd2,0xc3,0xb4,0xa5,0x96,0x87}";
   POK::Blowfish_Init =>
      "{0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10}";
end topsecret.i;

process partition_topsecret_send
features
   valueout : out data port types::integer
      {Allowed_Connection_Binding_Class =>
         (classifier (layers::topsecret.i));};
end partition_topsecret_send;

process implementation partition_topsecret_send.i
```

```
subcomponents
   thr : thread threads::thr_send_int;
connections
   port thr.outvalue -> valueout;
end partition_topsecret_send.i;

thread thr_send_int
features
   outvalue : out data port types::integer;
properties
   Dispatch_Protocol => Periodic;
end thr_send_int;
```

Listing 5.5 – Spécification d'une partition chiffrant les données envoyées

```
#define POK_PROTOCOLS_BLOWFISH_INIT \
   {0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10}
#define POK_PROTOCOLS_BLOWFISH_KEY \
   {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
    0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87}

void* thr_job ()
{
  types__integer sender_partition_topsecret_valueout_dvalue;

  while (1)
  {
    /* XXX code applicatif */
    pok_protocols_blowfish_marshall (&(valeur_claire),
              sizeof (int), &(valeur_chiffree), &(taille));
    pok_port_sampling_write (identifiant_canal,
                             &(valeur_chiffree), taille);
    pok_thread_period ();
  }
}
```

Listing 5.6 – Génération automatique du chiffrement des données

5.3.9 Intégration de code applicatif externe

Les patrons présentés produisent le code relatif à l'architecture du système, chargé d'exécuter sa partie applicative. Celle-ci est exécutée par des tâches qui appellent des sous-programmes implantés dans des langages de programmation tels que le C ou l'Ada.

Cependant, l'aspect applicatif des systèmes critiques est rarement écrit manuellement avec de tels langages et utilisent des modèles applicatifs tels SCADE ou Simulink. Ces approches permettent, à l'image de nos travaux portant sur l'architecture, de vérifier certaines caractéristiques de l'application (par exemple, vérifier qu'une valeur respecte certaines bornes). Il est donc nécessaire d'intégrer ces méthodes de conception applicatives à notre processus de développement afin d'embarquer leur code au sein des partitions.

Nous distinguons ainsi deux cas d'intégration de code applicatif :

1. intégration de code *legacy* (traditionnellement écrit manuellement, tels Ada, C) ;
2. intégration de modèles applicatifs (Simulink, Scade).

Dans les deux cas, l'aspect applicatif est modélisé par un composant AADL `subprogram` appelé par une tâche (composant `thread`). Le concepteur spécifie les contraintes d'implantation du sous-programme (nombre et type des arguments, nom du fichier contenant le code, etc.) au moyen de propriétés standards AADL (`Source_Text`, `Source_Name`, `Source_Language`, etc.). Le type de code est spécifié à l'aide de la propriété `Source_Language` : un sous-programme *traditionnel* définira la valeur `C` ou `Ada` alors que les modèles applicatifs utiliseront la valeur `Simulink` ou `Scade`.

L'intégration de code *legacy* s'effectue à l'édition de lien : les fichiers objets produits lors de la compilation du sous-programme sont intégrés à la partition qui l'utilise.

L'intégration de modèles applicatifs est plus complexe et nécessite une correspondance entre le modèle AADL et le modèle applicatif [35] (association des interfaces de communication). En outre, il est nécessaire que la sémantique du modèle applicatif (Simulink, Scade) soit compatible avec l'architecture (respect du flot de données, des contraintes temporelles, etc.).

Un patron de génération intègre les modèles applicatifs comme Simulink ou Scade. Il analyse le composant `subprogram` qui les représente et configure la partition afin qu'elle supporte les fonctions nécessaires à l'exécution du modèle applicatif (inclusion des bibliothèques mathématiques, etc.). Enfin, il connecte les deux modèles afin qu'ils puissent communiquer : les données de l'architecture sont injectées dans le modèle applicatif et inversement.

Au cours de nos travaux, nous avons modifié le processus de génération et d'intégration afin de pouvoir inclure dans les partitions du code *legacy* (Ada, C), et du code provenant de modèles Simulink ou SCADE. Ces aspects ont été discutés au sein d'une publication dans le workshop UML&AADL 2010 [35].

5.3.10 Optimisation

Lors du parcours de la hiérarchie de composants AADL, le générateur analyse et déduit les services requis par chaque couche du système (noyau, partition) et supprime toute fonction et ressource inutile ou inutilisée. Cela :

1. minimise l'empreinte mémoire : les fonctions inutiles n'étant pas intégrées au noyau ou aux partitions, leur code sera donc supprimé, réduisant ainsi la taille de l'application.
2. maximise le taux de couverture : la suppression de toute fonction inutilisée évite l'inclusion de code mort et ne conserve que le code réellement nécessaire à l'exécution des tâches. Les résultats obtenus par le biais de cette optimisation sont discutés dans la section 5.6.

Cette caractéristique d'optimisation de notre générateur et de notre plateforme est illustrée dans les figures 5.5 et 5.6 qui définissent deux systèmes ayant le même nombre de tâches. Le premier utilise des canaux intra et inter-partitions alors que le second n'utilise aucun mécanisme de communication (une telle situation est bien entendu caricaturale, les systèmes communiquant entre eux la plupart du temps). La figure 5.5 montre l'optimisation réalisée par la génération de code en explicitant les services inclus lors de la création d'un système contenant deux partitions connectées. Les patrons configurent

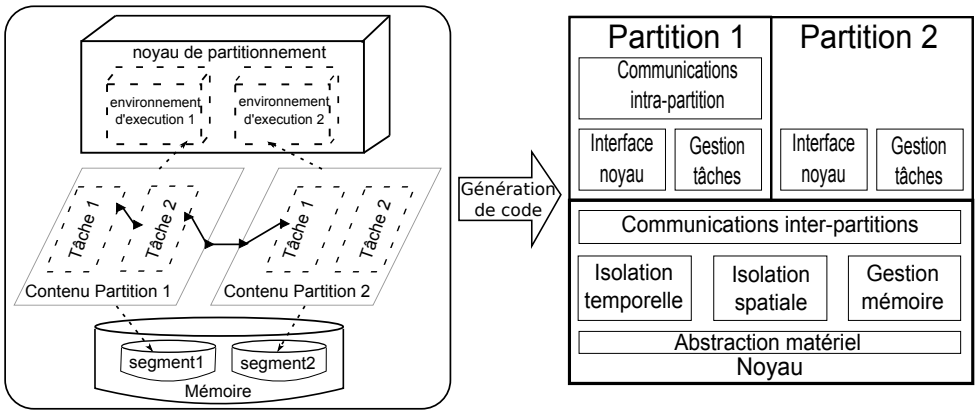


FIGURE 5.5 – Génération d'un système contenant deux partitions avec communications intra et inter-partitions

le système en incluant les services intra-partition dans la première partition et les fonctions de communication inter-partitions au noyau. Les tâches de la seconde partition n'étant pas connectées, les patrons n'incluent pas de service d'échange de données.

La figure 5.6 illustre la production du système n'utilisant aucun service de communication : le générateur configure la plate-forme afin de supprimer tout service relatif à l'échange de données dans chaque couche de l'architecture (noyau, partitions).

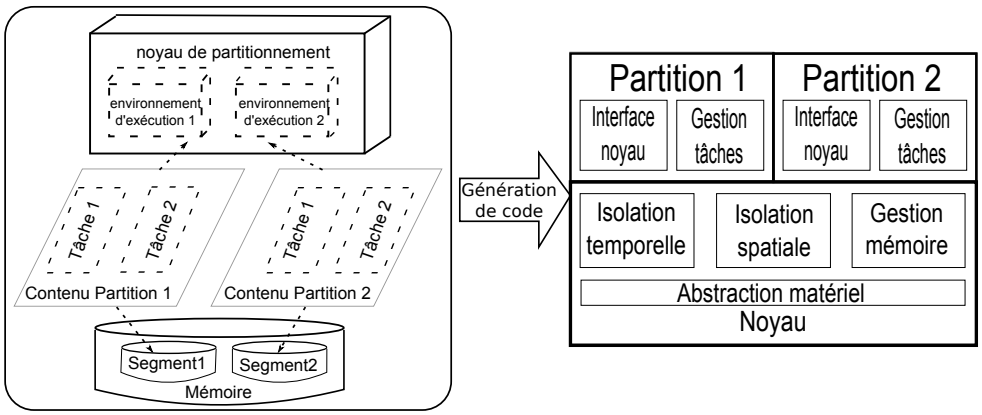


FIGURE 5.6 – Génération d'un système comprenant deux partitions sans connexion

Cette phase d'optimisation n'est réalisable que si l'environnement d'exécution propose des mécanismes de configuration avancés pour activer/désactiver les services. Les patrons de génération produisent des artefacts de configuration spécifiant les services requis par l'application, il est nécessaire que la plate-forme sous-jacente propose des mécanismes de configuration pour activer ou désactiver ses services. Notre plate-forme supportant le code généré, POK, permet d'adapter ces services et de les désactiver en

fonction du code généré. Cette particularité est détaillée au cours de la section 5.4.2.3.

5.3.11 Résumé des patrons de génération

Le tableau 5.8 synthétise les patrons de génération que nous proposons. Il donne un aperçu de l'utilisation de chaque composant dans le processus de création de code.

5.4 Plate-forme pour systèmes sûrs et sécurisés

La section précédente a présenté les patrons de génération de code spécifiques aux systèmes partitionnés. Le processus de production automatique de code traduit les modèles AADL en C, remplissant ainsi une partie du second objectif (section 1.4). Cependant, ce code nécessite une plate-forme d'exécution afin d'être compilé, intégré et produire l'application finale.

La présente section décrit la plate-forme logicielle que nous avons définie pour supporter l'exécution du code généré (remplissant ainsi la partie manquante de notre second objectif — section 1.4).

5.4.1 Description générale de POK

POK est un environnement d'exécution dédié à la sécurité et la sûreté de fonctionnement. Il a pour objectif d'être flexible, hautement configurable et conforme à plusieurs standards et approches, qu'ils soient spécifiques à un domaine (tels ARINC653 ou MILS) ou non (POSIX). A ce jour, POK est compatible avec plusieurs langages et standards (Ada, POSIX, ARINC653, etc.).

Ses services (dérivés de la sémantique des composants AADL) ont été profilés pour les systèmes critiques et respectent les fortes contraintes de ces systèmes : faible complexité des algorithmes, empreinte mémoire réduite, allocation statique des ressources, etc. Ils sont hautement configurables et peuvent être ajoutés/supprimés grâce à des directives de configuration. Cette caractéristique permet d'adapter POK le plus finement possible aux besoins du système.

Par ailleurs, des règles de développement strictes, adoptées au sein du développement de POK [32], réduisent la taille des services les plus critiques. Cette particularité a pour conséquence de garder un noyau minimal : ce dernier est actuellement composé de moins de 5000 lignes de code C.

POK est actuellement disponible sur trois architectures (x86, PowerPC et Sparc) et s'interface avec plusieurs langages (C ou Ada). Le projet est publié sous la licence libre BSD [85].

5.4.2 Architecture et services

L'architecture de POK est divisée en deux principales couches dont un aperçu est donné dans la figure 5.7 et le détail est illustré par les figures 5.8 et 5.9 :

1. **la couche noyau (POK kernel)** contient toutes les fonctionnalités nécessaires au confinement des partitions ;
2. **la couche partition (libpok)** définit les services nécessaires à l'exécution des tâches et du code applicatif. Cette couche constitue un support d'exécution pour le code applicatif fourni par l'utilisateur.

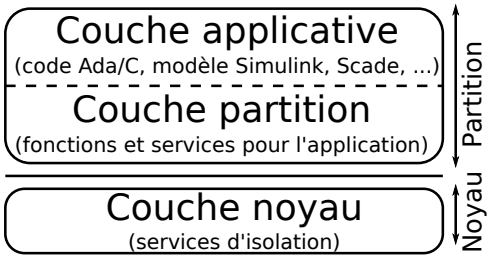


FIGURE 5.7 – Architecture générale de POK

5.4.2.1 Couche noyau

La couche noyau (détaillée par la figure 5.8) isole les partitions (isolation temporelle/spatiale) et contrôle les communications inter-partitions, assurant que seules celles autorisées peuvent être établies. Sa complexité a été volontairement réduite afin de faciliter de potentiels efforts de certification.

Elle se décompose en plusieurs services répartis en plusieurs couches. La première est une abstraction du matériel qui propose des fonctions génériques s'interfaçant avec les couches bas niveau du système. La définition de telles fonctions favorise le portage du noyau sur d'autres architectures et facilite l'écriture des services de haut niveau (ordonnancement, confinement, etc.).

Les services de gestion du temps et de la mémoire offrent une représentation commune du temps aux services d'isolation spatiale et temporelle. Ils comprennent des fonctions pour l'allocation des segments mémoire et l'ordonnancement de chaque partition.

Le service de gestion des fautes gère les exceptions et interruptions relatives aux erreurs survenant lors de l'exécution du système (erreur d'accès mémoire, division par zéro, etc.). Ce service propage les erreurs dans les partitions les ayant initiées.

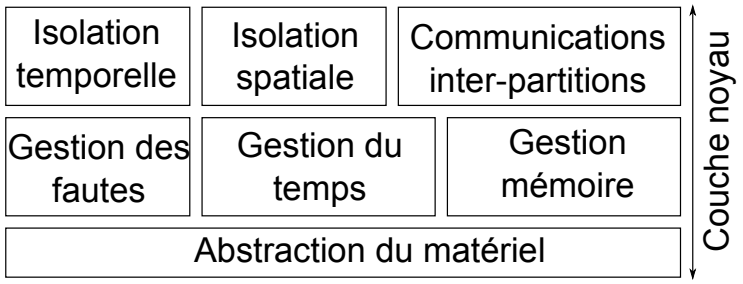


FIGURE 5.8 – Architecture détaillée de la couche noyau

5.4.2.2 Couche partition (libpok)

La couche partition (détaillée au sein de la figure 5.9) contient les services requis par le système et non supportés par le noyau. Au sein de cette section, nous présentons d'abord les services *« cœur »* de cette couche (interface noyau, gestion des tâches, communications) et décrivons les services étendus pour l'adaptation avec les standards actuels (POSIX, ARINC653).

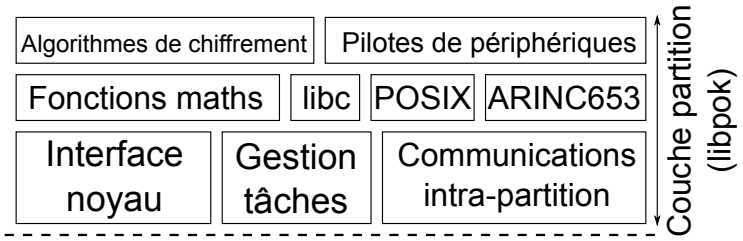


FIGURE 5.9 – Architecture détaillée de la couche partition

Services *« cœur »*

La couche partition s'interface avec le noyau pour accéder aux services qu'il fournit et requiert l'utilisation d'instructions privilégiées. Par exemple, un tel service est requis pour les communications inter-partitions : le noyau les contrôlant, il faut que la partition émette des requêtes au noyau pour les utiliser.

Le gestionnaire de tâches propose des fonctions pour leur création et leur gestion (démarrage, arrêt, attente, etc.). Il inclut également les primitives de gestion de la concurrence (mécanismes de verrouillage, sémaphores, etc.).

Les communications intra-partition gèrent trois types d'interactions au sein de la partition, repris de la sémantique d'AADL : *buffer*, *blackboard* et *event*. Elles proposent une sémantique identique aux mécanismes intra-partition d'ARINC653 (section 2.2.2). Chaque canal intra-partition doit être spécifié par le développeur à la compilation afin d'allouer statiquement les ressources qu'il requiert à la compilation.

Adaptation aux standards

La couche partition de POK comprend une interface d'adaptation avec les standards les plus utilisés en temps-réel. Cette conformité augmente le potentiel de portabilité des applications vers POK et permet de réutiliser des applications sur ce noyau par recompilation.

Plusieurs standards sont actuellement supportés :

1. ARINC653. Cette couche inclut l'API du standard ARINC653 et l'interface aux services *« cœur »* de POK. Bien que les sémantiques d'ARINC653 et de POK soient très proches, il est nécessaire de réaliser une adaptation syntaxique (introduction des fonctions et paramètres de l'API ARINC653, etc.). Cette couche d'adaptation permet au développeur de réutiliser ses applications existantes écrites pour le standard ARINC653 (objectif de portabilité).

2. POSIX [89], et la bibliothèque standard du C. Leur support n'est que partiel, car leur sémantique n'est pas toujours transposable aux systèmes partitionnés (ouverture de *« socket »* ou création de mémoire partagée par exemple, celles-ci violant les contraintes d'isolation spatiale des systèmes partitionnés). Cependant, un maximum de fonctions ont été portées afin de développer la couche d'adaptation la plus complète possible.
3. Fonctions mathématiques (libmath). Leur support a été requis pour l'intégration de modèles applicatifs (tels SCADE ou Simulink). L'implantation est réalisée par un port des fonctions disponibles dans le système d'exploitation NetBSD [57].

Service de chiffrement

Le service de chiffrement définit un ensemble de fonctions pour chiffrer et déchiffrer les données au sein des partitions. Il est utilisé par les partitions pour garantir la protection des données qu'elles envoient ou reçoivent lors de l'utilisation de connections inter-partitions (par exemple, lors de l'envoi de données classifiée sur un medium de communication non sécurisé tel un réseau *Ethernet*).

Le service de chiffrement utilise les fonctions mathématiques de la couche partition ; son utilisation requiert l'inclusion automatique des fonctions nécessaires à la réalisation des opérations arithmétiques. Enfin, il est conforme aux règles de développement du projet [32] : absence d'allocation dynamique, temps d'exécution borné, etc. Les paramètres d'initialisation et les clés associés aux protocoles de chiffrement sont spécifiés par des directives de compilation (macros C). Cette flexibilité de configuration permet d'interfacer facilement le code généré avec ce service.

Trois protocoles ont été implantés : *Ceasar cipher* [100], *Blowfish* [103] et *Data Encryption Standard* (DES) [19]. Tous trois sont des protocoles de chiffrement symétriques dont le temps d'exécution peut être calculé et déterminé en fonction de la taille des données à chiffrer.

Gestion des périphériques

Le service dédié aux périphériques gère le matériel au sein des partitions. Il remplit un objectif de sûreté : une défaillance du matériel ou des tâches qui le contrôle n'impactera pas les autres partitions. A l'inverse, comme discuté dans notre étude des solutions existantes (section 2.2.2), leur implantation au sein du noyau peut perturber le fonctionnement du système dans son ensemble.

Le contrôle des données du périphérique s'effectue via une interface dédiée avec le noyau (lecture/écriture des entrées/sorties du périphérique, accès au bus PCI, etc.). La gestion du périphérique est soumise à l'isolation spatiale, imposant des contraintes d'implantation : les fonctions de contrôle d'un périphérique doivent être déterministes et s'exécuter pendant le temps alloué à la partition.

Pour ces raisons, le pilote implanté dans POK utilise le modèle du *« polling »* [2] : lors de son exécution, la partition effectue des requêtes au périphérique

2. voir glossaire

pour obtenir les données qu'il contient (récupération de nouveaux paquets en provenance du réseau), réalise les traitements nécessaires (déterminer les destinataires des paquets) et lui envoie de nouvelles données (paquets à envoyer sur le réseau). Ce mode présente l'avantage d'être déterministe et prédictible (le temps d'exécution peut être borné par une analyse de la charge réseau).

Afin de montrer la bonne implantation de cette couche, nous avons implanté le pilote de la carte réseau *Realtek 8029*[3]. Cette première implantation a permis de créer une version répartie de POK tout en conservant l'isolation spatiale et temporelle entre les partitions. Une section dédiée (section 5.4.4 détaille les mécanismes spécifiques à l'implantation des pilotes de périphériques.

5.4.2.3 Modularité et finesse de configuration

Nous avons conçu les services de POK dans une optique de modularité :

1. la configuration du comportement des services : chacun d'eux doit pouvoir adapter son comportement par rapport aux directives de configuration. Par exemple, pour chaque partition, la politique d'ordonnancement peut être modifiée par une directive de configuration (macro C).
2. chaque service est activé par la définition d'une directive de configuration (macro C). Ainsi, l'utilisateur peut définir les services dont il souhaite disposer aussi bien dans le noyau que dans les partitions. Par défaut, tous les services sont désactivés, et chaque couche du système (noyau, partitions) doit être configurée afin de définir quels services elle contient. D'un point de vue implantation, une macro est dédiée à l'activation de chaque service. Par exemple, la définition de la macro `POK_NEEDS_PORTS_QUEUEING` active le service des *queuing ports* (communications inter-partitions).

Ces deux caractéristiques réduisent le code mort et l'empreinte mémoire. Par ailleurs, elles améliorent le taux de couverture du système : la configuration de chaque service spécifie quelles sont leurs caractéristiques utilisées, et désactive automatiquement celles qui ne seront pas exécutées. Leur activation par directive de configuration (et surtout, la désactivation par défaut) permet de ne conserver que ceux étant réellement nécessaires, supprimant ainsi tout code inutile et non exécuté par l'application.

Une telle optimisation est traditionnellement réalisée par des outils d'analyse qui détectent le code inutilisé et le suppriment une fois le système implanté. En l'intégrant à POK, nous affranchissons le développeurs d'avoir recours à de tels outils et apportons un gain de temps significatif, l'empreinte mémoire et la couverture de code étant automatiquement optimales à la conception.

5.4.3 Mécanismes de sûreté/sécurité

5.4.3.1 Isolation temporelle

L'isolation temporelle est fournie par la couche noyau et assure que chaque partition est exécutée pendant une tranche fixe, suivant un protocole d'ordonnancement cyclique, périodique.

3. cette carte est émulée par la machine virtuelle exécutant POK, QEMU [13].

Au sein de POK, l'ordonnanceur du noyau suit strictement cette politique d'ordonnancement périodique des partitions. Pour configurer l'isolation temporelle, le développeur définit :

- le nombre de tranches de temps réservées ;
- les valeurs des tranches de temps ;
- l'allocation des tranches de temps pour chaque partition ;
- la *major frame* (période à laquelle l'ordonnancement des partitions est répétée).

L'ordonnanceur des partitions est exécuté chaque milliseconde et détermine à partir de ces informations si un changement de partition est nécessaire.

Un changement de partition modifie le contexte d'exécution du processeur : il sauvegarde le contexte d'exécution de la partition courante et restaure celui de la partition élue. Cette opération modifie les descripteurs de segments de mémoire et assure que la partition élue n'a pas accès aux données de la partition précédemment exécutée.

5.4.3.2 Isolation spatiale

L'isolation spatiale de POK repose sur deux principales fonctions :

1. l'allocation d'un segment mémoire pour chaque partition ;
2. le bon fonctionnement du changement de contexte lors du changement de partition.

L'allocation des segments est réalisée à l'initialisation (appel à la fonction `pok_bsp_mem_alloc()`, listing B.1) du noyau et ne peut plus être modifiée au cours de l'exécution (essayer de réserver de nouveaux espaces mémoire entraîne un potentiel indéterminisme). La taille des segments est spécifiée par une directive de configuration (macro C), illustrée dans le listing 5.7 : ici, la première partition dispose de 110000 octets et la seconde de 92000.

Le service d'isolation spatiale doit par ailleurs être déterministe : la réservation de segments et les changements de partition doivent être réalisés en un temps borné, fixe. Les algorithmes traditionnellement utilisés dans les systèmes d'exploitation (la pagination [4]) empêchent la garantie de telles contraintes (l'allocation de pages mémoire et la restauration de la table des pages est difficilement évaluable). A l'inverse, le temps d'allocation mémoire et celui nécessaire au changement de contexte peut être facilement évalué par l'utilisation de segmentation. Ce dernier définit plusieurs zones mémoires fixes et ne requiert pas le maintient d'une table des pages allouées.

Pour pallier ce problème et garantir le déterminisme lorsque seule la pagination est disponible, la segmentation est émulée. Une table des pages statique est construite à l'initialisation du noyau à l'aide des informations de configuration des segments (listing 5.7). Le nombre de pages alloué est donc fixe et son utilisation (lors d'un changement de contexte) peut donc être borné.

```
#define POK_CONFIG_PARTITIONS_SIZE {110000,92000}
```

Listing 5.7 – Configuration de la taille des segments mémoire des partitions

4. voir lexique

Dans le cadre des systèmes partitionnés, un changement de contexte est une opération qui vise à changer la partition couramment exécutée par le noyau. Celle-ci modifie les zones mémoires mises à disposition de l'application. Une erreur dans sa mise en œuvre perturbe les propriétés de l'isolation spatiale de notre environnement d'exécution.

Au sein de la couche noyau, l'ordonnanceur maintient une variable contenant la partition en cours d'exécution (`pok_current_partition`). A chaque changement de partition, cette variable est modifiée et prend pour valeur la partition élue. Cette caractéristique garantit que l'isolation spatiale est assurée parallèlement à l'isolation temporelle et que les segments mémoires sont bien modifiés à chaque changement de partition.

5.4.3.3 *Health Monitoring*

Le service de *Health Monitoring* détecte les fautes et déclenche les actions appropriées pour les réparer. La principale difficulté de l'implantation d'un tel service réside dans sa flexibilité : il doit être paramétrable pour chaque couche en fonction du type de faute détectée et des actions de recouvrement associées. De plus, son implantation est différente des gestionnaires de fautes usuellement implémentés : les noyaux *« traditionnels »* gèrent toutes les fautes uniformément (une politique unique pour tous les composants) alors que l'architecture partitionnée introduit plusieurs niveaux avec une politique de confinement et de propagation des erreurs.

Les paragraphes suivants détaillent les mécanismes que nous avons implanté pour chaque couche de l'architecture de POK.

Couche noyau

La couche noyau dispose d'un accès privilégié au matériel qui lui permet de détecter les erreurs reportées par le processeur (division par zéro, erreur d'accès mémoire, etc.). Ces erreurs sont soit traitées localement par le noyau (cas de fautes causées par le code du noyau) soit propagées à la partition/tâche les ayant déclenchées.

L'implantation du noyau contient deux principaux mécanismes :

1. la détection et propagation des fautes matérielles aux partitions/tâches. Elles sont mises en place par le gestionnaire d'exceptions [5] du processeur, localisé dans le noyau. Lorsqu'une erreur lui est rapportée, il exécute une fonction de recouvrement. Si la nature de l'erreur est propre à la partition (division par zéro, dépassement de pile, etc.), le noyau la propage à la couche partition.
2. le traitement des erreurs propres à la couche noyau se fait au sein de ses fonctions : chaque service peut lever une erreur en appelant la fonction de recouvrement qui lui est associée. Celle-ci exécute alors l'action appropriée en fonction du type de faute détectée.

Dans notre contexte, cette fonction de recouvrement est générée automatiquement à partir du modèle AADL et des propriétés associées au composant

5. voir glossaire

`processor` (section 5.3.1). A titre d'exemple, le listing B.2 des annexes (section B.1) donne un exemple de procédure de recouvrement qui arrête le noyau à la découverte d'une erreur d'initialisation et le redémarre lorsqu'une erreur d'ordonnancement survient.

Couche partition

Le service de *Health Monitoring* de la couche partition gère deux types d'erreurs :

1. les erreurs internes à la partition, spécifiques à l'environnement d'exécution (ordonnancement des tâches, erreur d'initialisation de la partition, etc.) ;
2. les erreurs de ses tâches (division par zéro, erreur applicative, etc.)

La gestion des erreurs propres à la partition (ordonnancement, configuration, initialisation) utilise les mêmes mécanismes que le noyau. Le développeur définit une fonction de recouvrement (potentiellement générée) qui est appelée lorsqu'une erreur est levée. Un exemple de code est donné dans le listing B.3 : les partitions sont arrêtées à la levée d'une erreur.

La gestion des erreurs applicatives est réalisée au moyen d'une tâche dédiée à la gestion des fautes. Celle-ci est créée à l'initialisation de la partition puis est automatiquement mise en sommeil (état *idle*). Elle possède la plus forte priorité au sein de la partition et est activée en cas de détection d'erreur. Cette particularité doit être prise en compte lors de l'analyse d'ordonnançabilité : l'activation et le temps d'exécution d'une tâche de priorité maximale impacte le fonctionnement des autres ressources de la partition.

Cette tâche exécute indéfiniment une procédure de traitement d'erreurs. Á chaque réveil, elle exécute une procédure de recouvrement en fonction du type d'erreur (division par zéro, dépassement de pile, etc.) reporté et de l'entité fautive (la tâche de la partition ayant généré la faute). Le listing B.4 montre la constitution d'une telle procédure : pour chaque erreur détectée (erreur applicative, échéance non atteinte), la tâche fautive est redémarrée.

5.4.3.4 Algorithmes de chiffrement

Trois algorithmes de chiffrement ont été implantés : *Ceasar Cipher* [100], *Blowfish* [103] et *Data Encryption Standard* [19]. Ce sont tous trois des algorithmes de chiffrement symétrique déterministes (leur temps de calcul peut être déduit de la taille des données chiffrées) et légers (encombrement mémoire faible). Les mécanismes d'implantation ne sont pas détaillés (portage des fonctions d'*OpenSSL* [110]), l'accent des paragraphes suivants est mis sur sa flexibilité de configuration et d'utilisation, facilitant leur intégration avec le code généré.

Comme pour chaque service de POK, l'utilisation de ces algorithmes doit être explicitement déclaré au moyen de directives de configuration (macro C) afin de les inclure dans les partitions. Pour chaque algorithme de chiffrement,

nous avons défini un ensemble de directives constituant les caractéristiques de chaque protocole de chiffrement tels que :

- la clé de chiffrement
- les données d'initialisation

Cette facilité de configuration adapte chaque algorithme de chiffrement aux besoins de l'utilisateur, évitant de modifier ces paramètres au sein du code. Dans notre contexte, ces directives de configuration sont automatiquement générées à partir des informations associées aux niveaux de sécurité (section 5.3.8).

La mise en œuvre des protocoles de chiffrement est discuté au cours de la section dédiée au cas d'étude MILS (section 6.3).

5.4.4 Implantation des pilotes de périphérique

Les sections suivantes rappellent les modèles d'implantation des pilotes de périphérique au sein des architectures partitionnées et présentent les choix qui ont été décidés pour POK.

5.4.4.1 Modèles et choix d'implantation de POK

Comme discuté au cours des sections 2.1.1 et 2.2.2, les pilotes de gestion de périphériques peuvent être implantés soit dans le noyau, soit dans les partitions. Les avantages et inconvénients de ces deux méthodes ont été discutés au cours de la section 2.2.2, le tableau récapitulatif 2.1 synthétisant notre analyse de ces problèmes.

L'implantation des pilotes dans le noyau est intrusive : elle introduit du code spécifique à un cas d'utilisation (gestion d'un périphérique) et implique de revalider le système. Les nouvelles fonctions ont un impact sur les performances du noyau (temps de calcul des fonctions de gestion, ressources nécessaires, etc.) qu'il est alors nécessaire de comptabiliser lors de son analyse. Enfin, elles présentent un vecteur de faille important : une erreur dans ces fonctions peut générer une violation de l'isolation temporelle (boucle infinie) ou spatiale (écriture dans les segments d'une partition autorisé, le noyau ayant accès à tous les segments).

L'implantation dans les partitions est contraignante mais lève les problèmes soulevés précédemment. Ce méthode soumet les fonctions de gestion du périphériques aux mécanismes d'isolation spatiale et temporelle. Elles n'ont donc aucun impact sur le noyau, supprimant ainsi tout besoin de nouvelle analyse ou certification de ce dernier. Cependant, ce confinement dans les partitions empêche les fonctions de gestion d'avoir un accès direct au matériel. Par conséquent, pour accéder à ses données, il est nécessaire de développer des services dédiés au sein du noyau.

Les pilotes de périphériques de POK ont été implantés au sein des partitions, afin de garantir la sûreté et la sécurité du système. En particulier, nous avons souhaité :

- supprimer tout ajout de code intrusif dans le noyau requérant potentiellement de nouvelles analyses/certification.
- assurer l'isolation spatiale et temporelle sur tous les éléments du système, y compris les pilotes de périphériques.

5.4.4.2 Nouveaux services introduits

Dans POK, un gestionnaire de périphérique doit donc accéder au matériel qu'il contrôle pour communiquer avec ce dernier (envoi d'informations de contrôle, de paquets, etc.). Cependant, cet accès s'effectue dans un mode d'exécution privilégié (dédié au noyau) alors que les partitions s'exécutent dans un mode normal (dit mode *« utilisateur »* , n'étant pas autorisé à communiquer avec le matériel.

Il est nécessaire d'ajouter un service de contrôle du matériel avec le noyau et de l'interfacer avec les partitions. Ainsi, ces dernières peuvent effectuer des requêtes vers le noyau pour que celui-ci réalise les opérations nécessaires pour envoyer ou recevoir des données avec le périphérique contrôlé.

Toutefois, ce nouveau service ne doit pas se faire au détriment de la sécurité ou de la sûreté. En particulier, le noyau contrôle les requêtes effectuées par les partitions, assurant qu'elles ne concernent que les périphériques qu'elles contrôlent. Pour cette raison, un périphérique n'est géré que par une seule partition, facilitant ainsi l'analyse du système et empêchant de potentiels accès partagés (partitions communiquant via la mémoire du périphérique).

5.4.4.3 Implantation

Le gestionnaire de périphérique de la carte Realtek 8029 (modèle émulé par QEMU [13], simulateur utilisé par nos outils de certification, section 5.5) a été implanté pour nos travaux sur la version répartie de POK. Il gère la couche *Ethernet* du modèle OSI [120] et a la capacité de recevoir ou d'envoyer des paquets sur le réseau.

L'implantation de notre pilote utilise le modèle du *polling* : le gestionnaire interroge le périphérique périodiquement pour obtenir les paquets reçus par l'interface. Ce dialogue pilote/périphérique est réalisé par l'interface dédiée entre le noyau et la partition. Une fois les paquets reçus, ils sont traités par la partition. Lorsque le gestionnaire envoie des paquets, il effectue des entrées et sorties via le service d'interface du noyau. Ce dernier transmet alors la requête au périphérique qui envoie les données sur le médium physique.

5.5 Certification des exigences de la spécification

Les sections précédentes ont présenté le processus de génération d'applications partitionnées à partir de modèles AADL, remplissant ainsi le second objectif de notre approche (section 1.4). Cependant, bien que les patrons de génération apportent des garanties sur le code produit, rien n'assure que les applications automatiquement produites implantent les exigences spécifiées. Une telle assurance pourrait être apportée par une vérification et certification du générateur de code et de la plate-forme associée.

Afin de vérifier la conformité du système avec les spécifications, nous avons introduit une phase de certification. Elle est constituée d'un ensemble d'outils qui analysent les applications produites et vérifient plusieurs caractéristiques (implantation des mécanismes de sécurité/sûreté, couverture de code, etc.) lors de leur exécution. Nos outils offrent un support à l'utilisateur pour remplir les objectifs de certification applicables aux systèmes critiques (tels DO178B).

Les outils que nous proposons reposent principalement sur une utilisation du simulateur QEMU [13]. Ce dernier supervise le système lors de son exécution dans des conditions similaires à sa mise en production. Cette méthode présente l'avantage de tester le système sans avoir à l'instrumenter. En ce sens, elle n'est pas intrusive et ne nécessite pas de modifier le code de l'application.

Ces méthodes s'appuient sur les modèles AADL fournis par l'utilisateur. Les caractéristiques à analyser par chaque outil sont extraites de la spécification afin de les comparer avec l'exécution du système généré et vérifier leur respect. Deux approches de certification des exigences sont abordées : l'ordonnancement et le comportement du système.

5.5.1 Ordonnancement

Les systèmes partitionnés reposent sur une politique d'ordonnancement hiérarchique à deux niveaux :

1. le premier niveau élit les partitions au moyen d'un algorithme périodique contrôlé par le noyau. Ce dernier ordonnance chaque partition pendant au moins une tranche de temps fixe et prédéterminée à la configuration. La liste des tranches de temps de chaque partition est décrite dans les spécifications (section 4.2.1).
2. le second niveau est spécifique à chaque partition : chacune peut utiliser sa propre politique (RMS, EDF, *Round-Robin*, etc.).

Le caractère hiérarchique (deux niveaux d'ordonnancement) et hétérogène (politique d'ordonnancement spécifique à chaque partition) de l'ordonnancement des systèmes partitionnés complique sa vérification.

Nos travaux ont permis de valider l'ordonnancement du système à partir des spécifications (section 4.3.4). Cheddar [104] assure la faisabilité de l'ordonnancement par simulation du pire temps d'exécution des tâches des partitions. Cette première étape assure la conformité des spécifications mais ne réalise qu'une simulation pessimiste (seul le pire temps d'exécution est considéré).

Cependant, rien ne garantit que cette simulation est reproductible et que le schéma d'ordonnancement simulé sera similaire à l'exécution. Plusieurs problèmes peuvent se poser et générer des incohérences d'ordonnancement :

- la simulation ne considère que le pire temps de calcul alors que l'exécution rapporte la consommation réelle de temps processeur.
- le processus de génération de code peut produire une politique d'ordonnancement erronée.

– les algorithmes d'ordonnancement de la plate-forme d'exécution peuvent contenir des erreurs.
– la plate-forme consomme des ressources processeur pour contrôler le système (temps nécessaire pour envoyer une donnée, réaliser un *context switch*, etc.). En ce sens, l'utilisation de ses services a un impact sur le temps alloué aux partitions et peut impacter l'ordonnançabilité du système.

Il est donc nécessaire d'analyser l'ordonnancement à l'exécution et de le comparer avec la simulation réalisée au moyen des spécifications (analyse réalisée avec Cheddar et les modèles AADL- section 4.3.4). Les prochaines sections présentent l'outil dédié à cette validation.

5.5.1.1 Présentation de l'outil

La validation de l'ordonnancement à l'exécution est réalisée par une chaîne d'outils spécifiques [37] basée sur Cheddar [104], Ocarina [118] et POK [33]. Son fonctionnement, détaillé dans [37] et présenté en figure 5.10, s'articule autour de 5 étapes :

1. Production du schéma d'ordonnancement à partir du modèle AADL.
2. Génération de l'implantation du système à partir du modèle AADL.
3. Instrumentation de l'implantation, ajout d'instructions de supervision de l'ordonnancement.
4. Exécution du système instrumenté, production du schéma d'ordonnancement à l'exécution.
5. Comparaison des schémas d'ordonnancement issus des spécifications et de l'exécution.

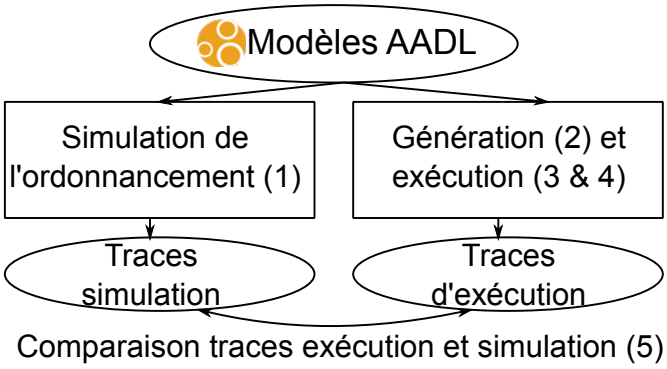


FIGURE 5.10 – Etapes de certification de l'ordonnancement

5.5.1.2 Mise en pratique

Le processus, illustré dans la figure, 5.10 a été mis en œuvre dans une chaîne de production dédiée qui :

1. utilise l'outil Cheddar pour produire le schéma d'ordonnancement à partir de modèles AADL (étape 1 de la figure 5.10).

2. génère l'implantation du système à partir de ces spécifications AADL en utilisant Ocarina et POK (section 5.2) (étape 2 de la figure 5.10).
3. instrumente l'implantation générée en vue de produire le schéma d'ordonnancement à l'exécution (étape 3 de la figure 5.10).
4. exécute du système et reproduit son schéma d'ordonnancement en utilisant QEMU [13] (étape 4 de la figure 5.10).
5. compare le schéma d'ordonnancement produit par Cheddar (étape 1) avec celui obtenu à l'exécution (étape 5 de la figure 5.10).

L'appel à l'outil Cheddar (étape 1) simule l'ordonnancement du système. A l'issue de cette étape, il crée le schéma d'ordonnancement et l'enregistre dans un fichier au format XML.

L'instrumentation des événements d'ordonnancement (étape 3) est réalisée par une modification automatique de l'ordonnanceur de POK. A chaque changement de partition ou de tâche, celui-ci affiche les informations nécessaires à la constitution du schéma d'ordonnancement.

Ces messages sont ensuite réutilisés (étape 4) pour produire le schéma d'ordonnancement à l'exécution. Ce dernier est stocké dans un fichier au format identique à celui de Cheddar.

Enfin, un outil dédié compare (étape 5) les schémas issus de la simulation et de l'exécution. Ce dernier vérifie ainsi si les événements d'ordonnancement obtenus à l'exécution sont cohérents avec ceux calculés sur la spécification.

Par ailleurs, il tient compte du caractère pessimiste de la simulation qui ne considère que le pire temps d'exécution. Ainsi, la vérification consiste à assurer que l'ordre d'exécution des tâches est identique entre l'exécution et la simulation et que l'isolation temporelle entre partitions est bien respectée.

Cependant, il est à noter cette méthode ne prend pas en compte tous les cas d'exécution possibles. En particulier, elle analyse le système dans son mode opératoire mais ne traite pas les situations introduisant des erreurs logicielles ou matérielles. Pour combler ces manques, il serait pertinent de relier cette approche à un ensemble de programmes testant exhaustivement tous les cas d'exécution possibles. Ce point est discuté en section 7.2.2.

5.5.1.3 Retour d'expérience

Lors de l'utilisation de notre outil, nous avons constaté que le schéma d'ordonnancement obtenu à l'issue de l'exécution était différent de celui calculé par simulation des spécifications. Ces différences s'expliquent par :

1. le coût de l'exécution (temps requis par la plate-forme pour ordonnancer les tâches, initialiser le système, etc.) non pris en compte lors de la simulation.
2. le caractère pessimiste de la simulation : seul le pire temps d'exécution est considéré.

La simulation du système à partir des modèles ne tient pas compte de certains coûts liés à l'exécution : temps d'initialisation, changement de contexte, etc. Par conséquent, lors de la comparaison des événements d'ordonnancement, on observe une différence de temps d'exécution entre la simulation et

l'exécution. Par exemple, la simulation considère que les tâches démarrent à l'initialisation des partitions alors qu'à l'exécution, on observe un délai entre l'initialisation de la partition et le démarrage de ses tâches (temps de création des ressources de la partition, etc.).

De plus, le temps d'exécution d'une tâche est souvent inférieur à son pire temps d'exécution qui est la seule métrique utilisée par la simulation. Cette caractéristique doit être prise en compte lors de la comparaison des événements d'exécution.

Malgré ces différences, ces travaux nous ont permis de vérifier plusieurs aspects de l'ordonnancement, à savoir :

- l'isolation temporelle entre partitions ;
- la bonne exécution des tâches (ordre d'exécution) et le respect de leurs contraintes temporelles.

Ce point a fait l'objet d'une présentation dans la conférence SIGAda09 [37].

5.5.2 Analyse comportementale

Afin de valider le comportement de l'application, nous avons réalisé un outil qui supervise l'exécution du système afin de détecter tout comportement anormal du système (exécution d'une instruction non spécifiée). Cette étape assiste l'utilisateur dans la validation du respect des spécifications : elle assure que chaque opération réalisée est spécifiée et correspond aux services décrits par l'utilisateur (par exemple, l'utilisation de `data port` dans le modèle AADL entraîne l'utilisation des fonctions de `sampling port` de la plate-forme).

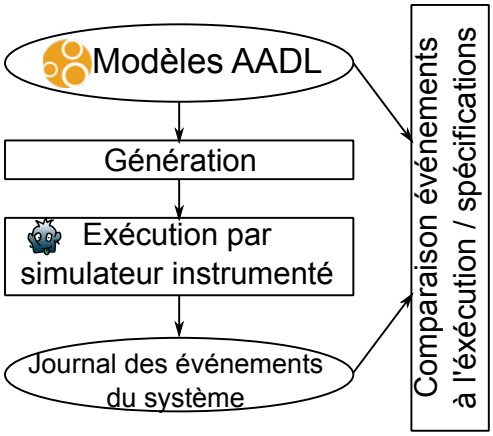


FIGURE 5.11 – Processus de supervision de l'activité des systèmes générés

L'outil *System Profiler Over QEMU* (SPOQ, version modifiée de QEMU) et son interfaçage dans notre chaîne de production est illustré dans la figure 5.11. Il exécute les systèmes générés et produit un rapport de l'activité du système. L'utilisateur les compare avec les spécifications et est ainsi capable d'analyser tout comportement déviant : partition utilisant une fonction non requise, erreur dans le changement de contexte d'une partition, etc.

SPOQ reporte les mesures suivantes à l'exécution du système :

- les segments mémoires utilisés (affichage des registres segments du processeur) ;
- les occurrences d'exceptions matérielles [6] ;
- les appels système et leurs paramètres associés.

La supervision des segments mémoires trace leur utilisation au cours de l'exécution. L'analyse de ces valeurs permet à l'utilisateur de valider la bonne implantation de l'isolation spatiale, chaque partition devant utiliser des segments mémoires différents.

La détection des occurrences d'exceptions supervise les fautes matérielles détectées à l'exécution du système. Leur analyse donne l'opportunité de valider la détection d'une erreur et la bonne implantation des mécanismes de recouvrement (service de *Health Monitoring*).

La supervision des appels système reporte toute requête d'une partition vers le noyau. L'utilisation de cette mesure permet de :

1. vérifier la conformité des paramètres des appels système par rapport aux spécifications. Par exemple, si une tâche envoie des données par un canal inter-partitions, l'utilisateur peut vérifier que la taille des données envoyées correspond au type de données spécifié.
2. détecter toute demande de service illégitime (demande de service non requis dans les spécifications). Par exemple, une telle analyse peut détecter les tentatives de contrôle d'un périphérique par une partition alors que ses spécifications n'indiquent pas qu'elle gère du matériel.

Nous avons pu surveiller l'activité des systèmes produits, en particulier l'isolation spatiale entre partitions : nous observons qu'en parallèle des évènements d'ordonnancement reportés par POK, les segments mémoire sont modifiés à chaque changement de partition.

Enfin, nous avons pu vérifier par analyse des traces d'exécution que seuls les appels système légitimes sont exécutés, montrant l'absence de comportement déviant du système.

Cependant, cette approche nécessiterait que toutes les possibilités d'exécution soient évaluées lors de l'exécution afin de tester le système dans des conditions similaires à la mise en production. A ce titre, elle constitue davantage un outil à coupler avec des bancs de tests qui réaliseront une exécution exhaustive des fonctionnalités du système (section 7.2.2).

La section 6.2.4 propose une mise en œuvre de cet outil sur un cas d'étude issu du domaine avionique.

5.6 Assurance des exigences des standards de certification

Les étapes de certification précédentes vérifient le comportement du système par rapport à ses spécifications. Cependant, certains standards (tel DO178B pour le domaine avioniques ou ECSS40 [41] pour le domaine spatial) requièrent que les systèmes générés soient conformes à des exigences décorellées des spécifications, telle la couverture de code (section 2.3.3).

6. voir glossaire

5.6.1 Contexte

Ces travaux ont pris place au sein du projet COUVERTURE ayant pour but la création d'outils d'analyse de couverture de code pour la certification DO178B de systèmes critiques. Le laboratoire dans lequel les présents travaux ont été effectués étant partenaire de ce projet, cela a été une opportunité de fournir un retour d'expérience et d'améliorer les outils du projet (xcov et version dédiée de QEMU).

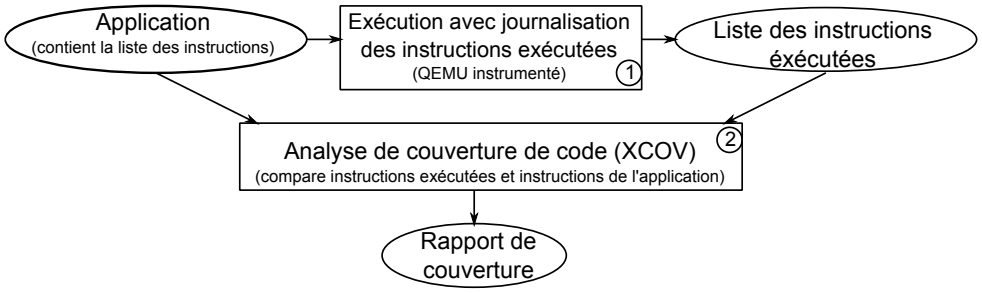


FIGURE 5.12 – Processus d'analyse de couverture avec les outils du projet COUVERTURE

Au cours du projet COUVERTURE, deux outils ont été développés afin de procéder à l'analyse de la couverture de code des applications :

1. une machine virtuelle instrumentée qui enregistre les instructions exécutées. Dans le contexte de COUVERTURE, ce programme a été réalisé par une modification de l'émulateur QEMU [13].
2. xcov, un outil d'analyse de couverture de code qui compare la liste des instructions exécutées avec la liste complète des instructions contenues dans un programme. Cet outil a été réalisé par ADACORE, partenaire du projet.

Le fonctionnement général de l'outil est illustré en figure 5.12 et repose sur deux étapes :

1. une exécution instrumentée de l'application (étape 1) qui enregistre la liste des instructions exécutées.
2. une comparaison automatique (étape 2) de la liste des instructions exécutées avec celles contenues dans l'application générée. Cette étape est mise en œuvre par un programme dédié, xcov, qui, à partir de ces éléments, produit des rapports de couverture de code (rapport du code exécuté/code non exécuté, fonctions appelées, etc.).

5.6.2 Intégration avec le processus de développement

L'analyse de couverture de code a été intégrée à notre chaîne d'outils de manière à produire automatiquement les rapports de couverture de code de chaque composant (noyau, partitions). Cela apporte un gain de temps conséquent puisque ces analyses sont traditionnellement réalisées manuellement par

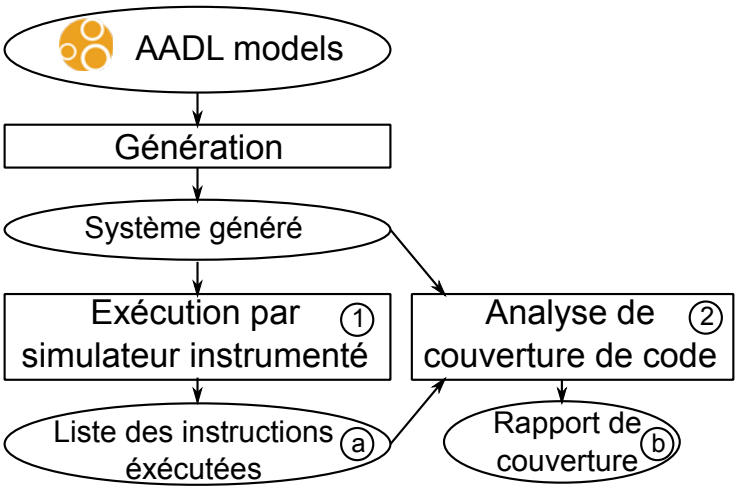


FIGURE 5.13 – Processus d'analyse de la couverture de code

les ingénieurs. Par ailleurs, elle ne demande aucune modification des fonctions du système, assurant que la certification est réalisée dans des conditions similaires à son exploitation.

Le fonctionnement de l'outil et son intégration dans notre chaîne sont illustrés dans la figure 5.13. L'application produite par le processus de génération est exécutée à l'aide de la machine virtuelle (étape `1` de la figure) qui produit la liste des instructions utilisées (élément `a` de la figure).

L'outil d'analyse xcov compare (étape `2`) la liste des instructions de l'application avec celles exécutées (élément `a` de la figure). Il peut ainsi différencier les instructions du programme et produire un rapport de couverture de code (élément `b` de la figure) qui spécifie son niveau de couverture.

Quelques limites sont toutefois imposées par l'outil utilisé. A ce jour, seul le code Ada peut bénéficier d'une analyse de couverture de code au niveau MC/DC. Nous devons donc nous restreindre à une analyse de couverture de code au niveau instruction et condition (section 2.3.3). Une couverture à ce niveau permet de certifier avec le standard DO178B des applications de niveau B (section 2.3.1).

5.6.3 Intérêt de la génération de code

Ces travaux sur la couverture nous donnent l'opportunité de montrer l'intérêt de la génération de code à partir des spécifications du système. Lors du processus de génération, l'outil (Ocarina) analyse le système et produit du code qui sélectionne les services requis par chaque couche de l'architecture partitionnée (section5.3.10) : seules les fonctions requises sont incluses dans l'application finale, réduisant ainsi significativement la quantité de code non couvert.

Toutefois, cette caractéristique que nous avons mise en œuvre dans notre processus n'est réalisable qu'à deux conditions :

1. le langage de modélisation offre une sémantique appropriée à la spécification des besoins du système et l'utilisateur doit être guidé dans son utilisation. C'est ce que nous proposons par la définition de patrons de conception avec le langage AADL (du chapitre 4). Les patrons de génération peuvent ensuite exploiter ces spécifications pour déduire les services requis par l'architecture et configurer finement la plate-forme.
2. la plate-forme d'exécution est modulaire et propose des mécanismes de configuration pour activer les services. Dans le cas de POK, cette contrainte est considérée dès la conception du système (section 5.3.10), permettant ainsi de désactiver chaque service à l'aide directives de configuration (macros C).

5.6.4 Retours d'expériences

Nous avons réalisé plusieurs tests afin d'obtenir le taux de couverture moyen des applications produites par notre approche.

Nos expériences ont mis en lumière que le taux de couverture du noyau était compris entre 80% et 90%, en fonction des fonctionnalités requises par le système. Bien que ces résultats constituent une première base intéressante, ils montrent que notre approche ne permet pas d'atteindre un taux de couverture intégral, requis pour les applications les plus critiques.

Nos expérimentations démontrent que le taux de couverture était fortement dépendant des fonctionnalités requises. A titre d'exemple, lorsqu'un composant applicatif nécessite la bibliothèque mathématique, de nombreuses fonctions sont ajoutées dans l'application alors que peu seront réellement exécutées, diminuant ainsi le niveau de couverture de manière conséquente. Cette inclusion du code *superflu* est responsable de l'inclusion de code non exécuté.

Ce problème peut être corrigé en raffinant les spécifications et en découpant davantage les services de la plate-forme d'exécution. Une description plus précise des exigences du système permettrait au générateur de code de configurer plus finement le système, supprimant certaines fonctionnalités toujours incluses par notre approche. Cette nouvelle configuration serait alors plus proche des exigences spécifiées par l'utilisateur. En parallèle, un découpage plus précis des services de la plate-forme adapterait les applications produites à cette nouvelle configuration et assurerait que seuls les services nécessaires seraient inclus dans les applications produites. Cette perspective est discutée plus longuement en section 7.2.2. Enfin, la génération automatique d'un jeu de tests complet essayant d'exploiter au maximum les services de la plate-forme (et donc, d'exécuter chaque instruction qui les composent) permettrait de procéder à une politique d'analyse de couverture plus intensive.

Les résultats de l'analyse de couverture de code ont constitué une base de discussion pour un article [22] publié avec les partenaires du projet COUVERTURE. Ils sont également abordés plus en détail au sein de l'étude de nos deux exemples (sections 6.2 et 6.3).

Synthèse de ce chapitre

Ce chapitre a abordé les éléments correspondant au second et troisième objectifs (génération et certification de systèmes sécurisés et sûrs, voir section 1.4) de notre approche :

- génération automatique de code pour systèmes partitionnés à partir de modèles AADL. En particulier, les patrons de production de code assurent le suivi des spécifications tout en limitant les inconvénients traditionnels des approches de génération de code (*overhead* mémoire, couverture de code faible, etc.).
- environnement d'exécution sûr et sécurisé pour le code généré (POK). Son caractère modulaire et hautement configurable permet d'intégrer le code généré facilement. Par ailleurs, cela permet d'assurer automatiquement plusieurs contraintes spécifiques aux systèmes critiques (réduction de l'empreinte mémoire, augmentation de la couverture de code).
- les outils de certification qui assistent l'utilisateur dans la validation de la conformité des applications produites pour :
  1. la vérification de l'ordonnancement par la comparaison des événements d'ordonnancement de la simulation avec ceux issus de l'exécution.
  2. la validation du comportement du système à l'exécution et la détection de comportements non spécifiés.
  3. l'analyse du taux de couverture de code de l'application générée en vue d'une certification avec les standards (DO178B et ECSS [41]).

Communications inter-partitions (*queuing* et *samplings ports*) : connections entre `process` ($PG^{inter-partitions}$)	
Entité AADL	**Code C**
Connection de ports entre composants de type `process` ($PM^{inter-partitions}E1etE6$)	– R1 : Génération de tableaux spécifiant les connections entre ports. – R2 : Instantiation des connections (génération d'appels de fonction à l'initialisation de la partition).
Taille des données échangées : composant `data` associé aux ports ($PM^{inter-partitions}E2$)	R3 : Allocation statique de tampons d'envoi/réception des données.
Taille de la file de réception : propriété `Queue_Size` ($PM^{inter-partitions}E3$)	R4 : Configuration de l'appel de fonction créant le port de communication, déclaration de la taille des données véhiculées.
Politique de mise en file : propriété `Queue_Processing_Protocol` ($PM^{inter-partitions}E4$)	R5 : Configuration de l'appel de fonction créant le port, et adaptation du paramètre spécifiant la politique de mise en file (*queuing ports* uniquement).
Temps maximal d'attente : propriété `Timeout` ($PM^{inter-partitions}E5$)	R6 : Ajout du temps maximal d'attente lors de la réception ou de l'envoi d'une donnée (illimité par défaut).
Période d'arrivée des données : propriété `Refresh_Period` ($PM^{inter-partitions}E7$)	R7 : Spécification de la période d'arrivée des données lors de l'appel de fonction initiant un *sampling port*.
Génération des mécanismes de chiffrement ($PM^{chiffrement}E1$ et $PM^{niveau-securite}$E4)	– R8 : Appels de fonctions chiffrant les données avant envoi. – R9 : Appels de fonctions déchiffrant les données à la réception.
Configuration des mécanismes de sécurité ($PM^{chiffrement}E3$)	R10 : Génération de macros définissant la configuration des mécanismes de sécurité (`POK_CONFIG_PROTOCOLS_...`).

TABLE 5.7 – Patron de génération des communication inter-partitions

Composant AADL	**Patron de génération**
`processor`	Configuration du noyau et de sa politique de partitionnement spatial et temporel. Configuration de la politique de *Health Monitoring* au niveau noyau.
`virtual processor`	Configuration de la politique d'ordonnancement de la partition, des fonctions requises et de la politique de *Health Monitoring*.
`process`	Instanciation des ressources de la partition (données, tâches, etc.).
`thread`	Création de la tâche conformément à ses contraintes et génération d'une fonction périodique d'exécution. Configuration du health-monitoring pour la tâche concernée.
Connection entre `thread`	Initialisation d'un canal de communication intra-partition (*buffer*, *event* ou *blackboard*) et génération de l'envoi/réception des données à travers ce canal.
Connection entre `process`	Initialisation d'un canal de communication inter-partitions (*queuing* ou *sampling port*), génération des envois/réceptions de données, déclaration dans le noyau (afin d'éviter la création de canaux cachés/non autorisés) et chiffrement automatique des données envoyées.
`Data`	Déclaration d'une variable partagée entre les tâches, initialisation et utilisation d'un *sémaphore* pour la protéger des accès concurrents.

TABLE 5.8 – Résumé des patrons de génération de code

Chapitre 6

Cas d'études

Sommaire du chapitre

6.1 Mise en avant de l'approche, éléments étudiés . . . 181

6.1.1 Aspect sûreté 181

6.1.2 Aspect sécurité 181

6.1.3 Plan de l'étude 181

6.2 Cas d'étude *« integrated avionics »* : gestionnaire de pilotage 182

6.2.1 Introduction 182

6.2.2 Spécifications AADL 184

6.2.3 Validation de la spécification 188

6.2.4 Validation des mécanismes de *Health Monitoring* . . 189

6.3 Cas d'étude *« MILS »* 191

6.3.1 Introduction, spécification 191

6.3.2 Spécifications AADL 194

6.3.3 Validation des spécifications 198

6.3.4 Validation de l'implantation 200

6.4 Analyse de la couverture de code 201

6.4.1 Cas d'étude *integrated* 201

6.4.2 Cas d'étude *MILS* 202

6.4.3 Analyse des résultats 203

6.5 Étude de l'empreinte mémoire 203

6.5.1 Protocole de test 204

6.5.2 Résultats du cas d'étude *integrated* 205

6.5.3 Résultats du cas d'étude *MILS* 205

6.5.4 Analyse des résultats 206

6.5.5 Comparaison avec les travaux précédents 207

Rappels

Les chapitres précédents ont détaillé notre processus de développement de systèmes sûrs et sécurisés au travers de :

1. **Règles de modélisation et de validation (chapitre 4)**. Celles-ci guident le concepteur du système dans la spécification de son architecture et de ses caractéristiques. L'utilisation d'un langage de modélisation standardisé, associé à ces règles permet d'utiliser des outils d'analyse, assurant la faisabilité du système et l'absence d'erreur au niveau de ses spécifications.
2. **Une implantation et une certification automatique (chapitre 5)**. L'étape d'implantation traduit en code exécutable et l'intègre à une plate-forme d'exécution partitionnée qui fournit des services de sécurité et de sûreté. Ce processus de génération assure la bonne implantation des caractéristiques du système et facilite son analyse. L'étape de certification supervise l'exécution de l'application générée et valide le respect des exigences de ses spécifications.

Objectifs du chapitre

Les sections qui suivent mettent en pratique l'approche développée au cours des chapitres 4 et 5. Elles s'attachent à montrer son utilité dans le contexte des systèmes critiques au travers de deux cas d'études : l'un dédié à l'assurance de sûreté, l'autre à la sécurité.
La sûreté de fonctionnement est mise en valeur au travers d'une étude définissant un système de pilotage d'avion (cas d'étude *« integrated »*). Celui-ci spécifie une architecture avionique modulaire (IMA) composée de plusieurs partitions isolées pour des raisons de sûreté. Cette étude consiste à montrer que notre approche est capable de spécifier, valider et implanter des systèmes avioniques de taille conséquente. Enfin, l'architecture proposée est semblable à celles conçues dans un contexte industriel comme le montre [45].
Les aspects sécurité sont mis en pratique dans un cas d'étude (nommé *MILS*) composé de deux machines échangeant des données à différents niveaux de sécurité via un réseau non sécurisé (tel un lien Ethernet). Chaque niveau définit ses mécanismes de protection des données (protocole de chiffrement, etc.) et est isolé dans des partitions dédiées. Au cours de cette étude, le confinement de l'information est validé au niveau de spécifications et les mécanismes de protection sont validés à l'exécution.

6.1 Mise en avant de l'approche, éléments étudiés

Au cours de cette section, nous détaillons l'utilisation de notre méthode pour la spécification, la validation, l'implantation et la certification d'architectures sûres et sécurisées. Les sections suivantes détaillent les éléments propres à chaque exigence.

6.1.1 Aspect sûreté

Les aspects sûreté du cas d'étude *integrated* mettent en pratique les éléments suivants de notre approche :

1. L'utilisation de patrons de modélisation dédiés à la spécification d'architectures partitionnées et de leur politique de recouvrement.
2. La validation de l'architecture partitionnée et de ses mécanismes de recouvrement d'erreurs.
3. La génération automatique de l'implantation en respectant les mécanismes de sûreté spécifiés.
4. La validation du système généré et de l'implantation de sa politique de recouvrement par une analyse comportementale de l'exécution.
5. Le respect du standard de certification DO178B par une analyse de couverture de code.

6.1.2 Aspect sécurité

Le cas d'étude dédié à la sécurité met en valeur notre approche par :

1. L'utilisation des patrons de modélisation pour la spécification des niveaux de sécurité et de leurs mécanismes associés.
2. La validation de l'isolation des niveaux de sécurité par analyse de modèles AADL.
3. La génération automatique de l'implantation et de ses mécanismes de sécurité.
4. La validation de l'implantation des mécanismes de sécurité par une analyse du système à l'exécution.
5. Une analyse de couverture de code, élément requis par les standards applicables aux systèmes critiques (tel DO178B).

6.1.3 Plan de l'étude

Dans un premier temps, chaque cas d'étude est présenté séparément (sections 6.2 et 6.3). Nous décrivons le contexte d'utilisation de ces systèmes, détaillons leurs exigences et les modèles AADL associés. Chacune de ces sections abordent les outils de certification utilisés pour analyser leur exécution (validation mécanismes de sûreté pour le cas d'étude *« integrated »* , supervision de l'isolation des données pour le cas d'étude *MILS*).

La section 6.4 rapporte les taux de couverture de code des composants générés (noyau, partitions) dans chaque cas d'étude et met en relation les

résultats obtenus avec l'approche globale. En particulier, il explique la pertinence de la génération de code pour obtenir un haut niveau de couverture de code. Enfin, la section 6.5 présente une analyse de l'empreinte mémoire des applications générées et compare les résultats avec les travaux existants.

6.2 Cas d'étude *« integrated avionics »* : gestionnaire de pilotage

Les sections suivantes décrivent le cas d'étude *« integrated »* , dédié à la sûreté. Il présente ces spécifications sous une forme textuelle, leur traduction à l'aide de nos patrons de modélisation et la validation des exigences de sûreté à partir du modèle.

6.2.1 Introduction

Ce cas d'étude définit une application avionique de contrôle de vol, dirigeant la trajectoire de l'avion, gère le matériel de vol, affiche les informations de navigation au pilote et détecte les erreurs potentielles. Son architecture suit la philosophie *« Integrated Modular Avionics » (IMA)* qui consiste à séparer les fonctions dans des partitions pour les isoler les unes des autres.

Aperçu de l'architecture

L'architecture proposée comprend 6 partitions inter-connectées, chacune remplissant une fonction bien définie :

1. Le *Flight Manager* calcule le plan de vol de l'avion.
2. Le *Flight Director* gère le matériel contrôlant l'avion et répond aux demandes de contrôle du contrôle de vol (partition 1 — *Flight Manager*).
3. Le *Page Content Manager* contient toutes les informations potentiellement affichables sur l'écran du pilote et les organise en différentes pages. Les informations proviennent de la partition 1 (*Flight Manager*).
4. Le *Display Manager* affiche une page d'information de navigation parmi celles disponibles au sein de la partition 3.
5. Le *Hardware Display* contrôle le périphérique affichant les pages (carte vidéo + écran) et affiche la page choisie par la partition 4.
6. Le *Warning Annunciation Manager* détecte de erreurs potentielles survenant lors du vol.

Les connections entre les partitions utilisent le principe de la mémoire partagée : aucune valeur n'est mise en attente. Ce concept de communication correspond à celui des *sampling port* du standard ARINC653. Chaque partition est isolée temporellement et spatialement.

Origine de ce cas d'étude

Cet exemple a été défini dans un rapport technique [45] publié par le *Software Engineering Institute* pour la conception et l'analyse de systèmes avioniques avec le standard AADL. Un rapport [45] détaille les patrons de modélisation utilisés ainsi qu'une analyse des flots de communication inter-partitions

(vérification que la latence de communication est inférieure au temps de communication demandé par le concepteur d'application). La version initiale de ce modèle a été créé en 2007. Il utilise donc la première version du langage AADL (la seconde version ayant été publiée un an après), peu adapté à la spécification d'architecture partitionnée (composants et propriétés manquants — par exemple les `virtual processor`).

Problèmes de la spécification initiale

En particulier, cette première version de la spécification ne permet pas d'analyser plusieurs éléments du système de par le manque de précision des patrons de modélisation utilisés. En particulier :

1. l'environnement d'exécution des partitions n'est pas spécifié, empêchant leur analyse (simulation de l'ordonnancement des partitions et du système, fonctionnalités proposées suffisantes par rapport aux besoins de la couche applicative, etc.).
2. les connections entre partitions ne spécifient pas leurs contraintes de configuration (délai d'arrivée des données, etc.), ce qui restreint l'analyse de l'architecture (ces éléments ont un impact sur l'ordonnançabilité) et empêche la génération de code (ces mécanismes doivent être spécifiés pour générer la configuration de chaque canal de communication).
3. les mécanismes de *health monitoring* ne sont pas spécifiés, ce qui empêche de générer la politique de détection et de reprise d'erreur.

Afin de pallier ces problèmes, nous avons adapté cet exemple à l'aide de nos patrons de modélisation afin de mieux décrire l'architecture et ses exigences.

Utilisation de nos patrons de modélisation

Afin de pallier ces manques, nous avons donc proposé une nouvelle version de cette spécification en utilisant les patrons de modélisation du chapitre 4. En particulier, les éléments suivants ont été apportés :

- Des composants `virtual processor` et `memory` ont été ajoutés afin de modéliser l'isolation temporelle et spatiale :
    1. Chaque partition est isolée dans des segments mémoire distincts ayant une taille minimale de 100 Ko (estimation de l'empreinte mémoire de chaque partition).
    2. Les partitions sont exécutées périodiquement par un algorithme *« Round-Robin »* . La *Major Frame* du système (période à laquelle le cycle d'ordonnancement est répété) est de 120ms. Pendant une période, chaque partition est exécutée pendant 20ms.
- Les connections, initialement spécifiées via des groupes de ports (`port group`, type d'interface inutilisée dans le cas des architectures partitionnées), ont été séparées en plusieurs `data ports` inter-connectés. Par ailleurs, la configuration de chaque port a été spécifiée afin d'être en mesure de générer leur code d'initialisation. Ainsi, le patron de modélisation utilisé correspond à un type de communication utilisé dans les systèmes partitionnés (*sampling port*).

– La politique de *health monitoring* a été spécifiée à chaque niveau de l'architecture :

1. Au niveau noyau (module ARINC653), pour chaque faute potentielle (erreur d'ordonnancement, de configuration ou d'initialisation), la politique de recouvrement redémarre le système.
2. Pour chaque partition, les fautes potentielles (erreur d'ordonnancement, de configuration ou d'initialisation), utilisent une politique de recouvrement qui arrête la partition concernée.
3. Au niveau tâche, par défaut, toute erreur détectée redémarre la partition qui l'exécute.

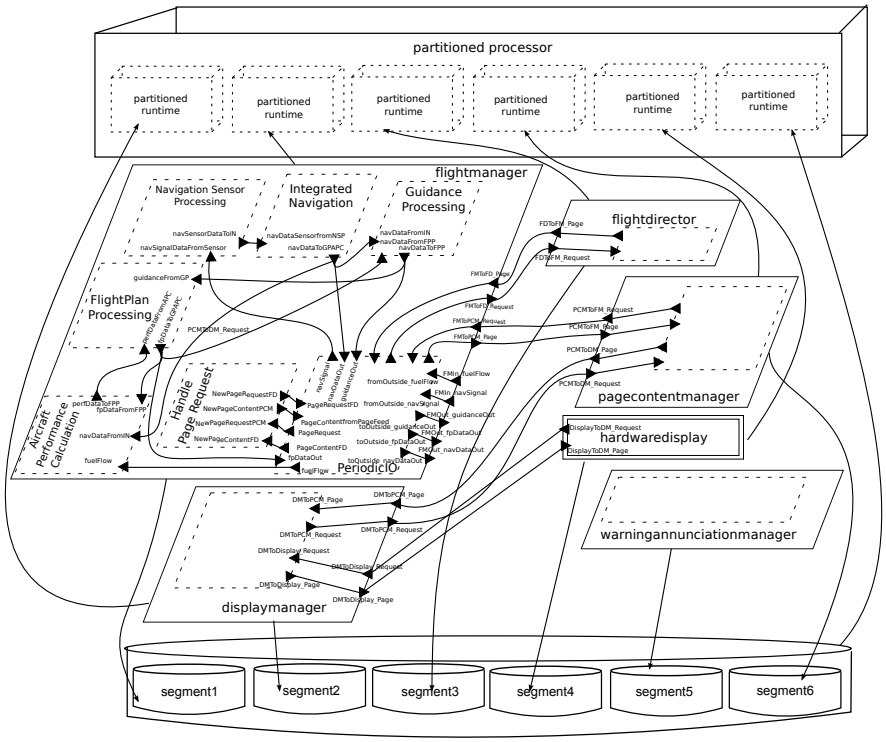


FIGURE 6.1 – Cas d'étude *« integrated »*

6.2.2 Spécifications AADL

La spécification AADL correspondante à l'architecture est illustrée dans la figure 6.1, la représentation textuelle associée est donnée en annexe (section B.4.1 page 247). Les paragraphes suivants détaillent l'utilisation de nos patrons de modélisation (4) pour spécifier cette architecture.

Ce cas d'étude décrivant une architecture avionique, le modèle associé utilise les patrons de modélisation dédiés à la description d'architecture ARINC653. Nous utilisons donc les propriétés spécifiques qui ont été proposés dans le cadre de la définition de l'annexe ARINC653 au standard AADL [34].

Noyau partitionné

Le composant AADL `processor` représente le noyau partitionné garantissant l'isolation spatiale et temporelle. Il contient six sous-composants `virtual processor`, chacun représentant l'environnement d'exécution d'une partition qu'il exécute (le composant `virtual processor` associé à la partition *Display Manager* est présenté en listing 6.2). Les caractéristiques (politique d'ordonnancement, politique de *health monitoring*, etc.) de chaque environnement est spécifié au moyen de propriétés AADL (`Scheduling_Protocol`, `HM_Errors`, etc.).

```
processor implementation partitioned.x86
subcomponents
   part1 : virtual processor
           partitions::runtime.warningannunciationmanager;
   part2 : virtual processor partitions::runtime.displaymanager;
   part3 : virtual processor partitions::runtime.pagecontentmanager;
   part4 : virtual processor partitions::runtime.flightmanager;
   part5 : virtual processor partitions::runtime.flightdirector;
properties
   ARINC653::Module_Major_Frame  => 100ms;
   ARINC653::Partition_Slots     => (20ms, 20ms, 20ms, 20ms, 20ms);
   ARINC653::Slots_Allocation    => (reference (part1),
                                     reference (part2),
                                     reference (part3),
                                     reference (part4),
                                     reference (part5));
   ARINC653::HM_Errors  =>
      (Module_Config, Module_Init, Module_Scheduling);
   ARINC653::HM_Actions =>
      (Module_Restart, Module_Restart, Module_Restart);
   POK::Architecture => x86;
   POK::BSP => x86_qemu;
end partitioned.x86;
```

Listing 6.1 – Spécification du noyau du cas d'étude *Integrated*

Partitions

Chaque partition est spécifiée au moyen d'un composant `process` et `virtual processor`. Les composants `virtual processor` (environnement d'exécution de la partition) sont contenus au sein d'un `processor` (listing 6.1) alors que les composants `process` sont ajoutés dans un composant hybride `system`. Ces deux composants sont associés par la propriété `Actual_Processor_Binding`. Le listing 6.2 présente la définition de ces deux composants pour la partition *Display Manager*.

```
virtual processor implementation runtime.displaymanager
properties
   ARINC653::HM_Errors  => (Partition_Scheduling, Partition_Config,
                            Partition_Handler, Partition_Init);
   ARINC653::HM_Actions => (Partition_Stop, Partition_Stop,
                            Partition_Stop, Partition_Stop);
   Scheduling_Protocol  => RMS;
end runtime.displaymanager;

process process_displaymanager
```

```
features
   DMToDisplay_Request   : in data port types::PageRequestCmd;
   DMToDisplay_Page      : out data port types::PageContent;
   DMToPCM_Request       : out data port types::PageRequestCmd;
   DMToPCM_Page          : in data port types::PageContent;
end process_displaymanager;

process implementation process_displaymanager.impl
subcomponents
   thr : thread threads::thread_displaymanager.impl;
connections
   port DMToDisplay_Request    -> thr.DMToDisplay_Request;
   port thr.DMToDisplay_Page   -> DMToDisplay_Page;
   port DMToPCM_Page           -> thr.DMToPCM_Page;
   port thr.DMToPCM_Request    -> DMToPCM_Request;
end process_displaymanager.impl;
```

Listing 6.2 – Spécification de la partition *DisplayManager* du cas d'étude *Integrated*

Isolation spatiale et mémoire

Les partitions définissent leur contenu (tâches, ressources, couche applicative sous-jacente, etc.) dans un composant `process`. La localisation au sein de la mémoire doit cependant être spécifiée. Pour cela, nous décrivons l'organisation de la mémoire principale, sa division en segments et leur association avec les partitions.

La mémoire principale (mémoire RAM) est spécifiée à l'aide d'un unique composant de type `memory` divisé en sous-composants `memory` représentant les segments. Leur taille (100Ko) est définie au moyen de la propriété standard `Byte_Count`. Le listing 6.3 présente la décomposition de la mémoire en segments : le composant `ram` contient plusieurs sous-composants de type `memory`, chacun d'eux décrit une partie de la mémoire pour stocker le code ou les données de chaque partition.

Chaque composant `process` est associé à un segment mémoire pour spécifier la stratégie d'isolation spatiale (section 4.2.3). La propriété standard `Actual_Memory_Binding` est définie sur chaque composant AADL `process` et référence le segment mémoire (composant `memory`) qui la contient.

```
memory implementation segment.data_segment
properties
   ARINC653::Memory_Kind => memory_data;
   ARINC653::Access_Type => read_write;
   Byte_Count => 3000;
end segment.data_segment;

memory implementation segment.code_segment
properties
   ARINC653::Memory_Kind => memory_code;
   ARINC653::Access_Type => read;
   Byte_Count => 5000;
end segment.code_segment;


memory implementation ram.i
subcomponents
   segment1_data : memory segment.data_segment;
```

```
    segment1_code : memory segment.code_segment;
    segment2_data : memory segment.data_segment;
    segment2_code : memory segment.code_segment;
    segment3_data : memory segment.data_segment;
    segment3_code : memory segment.code_segment;
    segment4_data : memory segment.data_segment;
    segment4_code : memory segment.code_segment;
    segment5_data : memory segment.data_segment;
    segment5_code : memory segment.code_segment;
    segment6_data : memory segment.data_segment;
    segment6_code : memory segment.code_segment;
end ram.i;
```

Listing 6.3 – Spécification de la mémoire du cas d'étude *Integrated*

Isolation temporelle

Le noyau de partitionnement isole et contrôle le temps d'exécution des partitions, assurant au bon respect de la période qui leur est allouée. Cette caractéristique du système est spécifiée (patron de modélisation 4.2.4) en associant les propriétés `ARINC653::Slots`, `ARINC653::Slots_Allocation` et `ARINC653::Module_Major_Frame` au composant AADL `processor` (noyau). Le listing 6.1 montre la définition de ces propriétés pour le composant `processor` de ce cas d'étude.

La propriété `ARINC653::Slots` définit une liste de six valeurs de 20ms, décrivant les six tranches de temps que le noyau alloue pour l'exécution de ses partitions. La propriété `ARINC653::Slots_Allocation` spécifie l'allocation de ces périodes d'exécution à chaque partition. Dans notre exemple, chaque partition utilise une période d'exécution (20ms).

Enfin, la *« Major Frame »* du module est spécifiée en définissant la propriété `ARINC653::Major_Frame` à 120ms.

Politique de reprise des fautes

La politique de détection et de reprise des fautes doit être spécifiée à chaque niveau de l'architecture : noyau (composant AADL `processor`), partition (composant AADL `virtual processor`) et tâche (composant AADL `thread`).

Les patrons de modélisation dédiés à la description des fautes et des politiques de recouvrement (sections 4.4.2 et 4.4.3) associent les propriétés (`ARINC653::HM_Errors` et `ARINC653::HM_Actions`) à chaque niveau de l'architecture : composants `processor` (listing 6.1), `virtual processor` (listing 6.2) et `thread` (listing 6.4). Ces propriétés sont définies composants conformément aux spécifications de la politique de reprise des fautes (spécifications de la section 6.2.1) : en cas de faute, le noyau redémarre (valeurs de la propriété `HM_Errors` à `Module_Restart`), les partitions s'arrêtent (`Partition_Stop`) et les tâches arrêtent la partition (`Partition_Restart`).

```
thread thread_displaymanager
features
    DMToDisplay_Request   : in data port types::PageRequestCmd;
    DMToDisplay_Page      : out data port types::PageContent;
```

```
   DMToPCM_Page          : in data port types::PageContent;
   DMToPCM_Request       : out data port types::PageRequestCmd;
properties
   Dispatch_Protocol        => Periodic;
   Period                   => 50 Ms;
   Compute_Execution_Time   => 5 Ms .. 15 Ms;
   Source_Data_Size         => 10 Kbyte;
   Source_Code_Size         => 10 Kbyte;
   ARINC653::HM_Errors     => (Deadline_Miss, Application_Error,
                               Numeric_Error, Illegal_Request,
                               Stack_Overflow, Memory_Violation,
                               Hardware_Fault, Power_Fail);
   ARINC653::HM_Actions    => (Partition_Restart, Partition_Restart,
                               Partition_Restart, Partition_Restart,
                               Partition_Restart, Partition_Restart,
                               Partition_Restart, Partition_Restart);
end thread_displaymanager;

thread implementation thread_displaymanager.impl
calls
   call1: {pspg : subprogram subprograms::spg_displaymanager;};
connections
   parameter pspg.request -> DMToPCM_Request;
   parameter DmToPCM_Page -> pspg.page;
end thread_displaymanager.impl;
```

Listing 6.4 – Spécification de la tâche de la partition *DisplayManager* du cas d'étude *Integrated*

Autres éléments de l'architecture

La spécification réalisée introduit d'autres éléments qui ne sont pas relatifs aux aspects sûreté mais sont toutefois nécessaires à l'analyse ou la génération de l'implantation. Les paragraphes suivants donnent un aperçu de leur définition.

Le modèle définit les types de données utilisés dans les communications inter-partitions. Afin de simplifier l'aspect applicatif de l'application, seuls des entiers sont utilisés. La description de ces types peut être trouvée en annexe (section B.4.1.8).

Les sous-programmes de ce système sont implantés manuellement en `C`. Les composants spécifiant les sous-programmes référencent (section B.4.1.5) ces routines afin de les intégrer au système lors de sa production.

6.2.3 Validation de la spécification

Les paragraphes suivants détaillent l'utilisation des règles proposées pour la validation de l'architecture (section 4.3) et de sa politique de sûreté (section 4.5).

Conformité du modèle

La première étape de validation analyse le modèle AADL et vérifie le respect de l'utilisation de nos patrons de conception. Elle assure (section 4.3) assure que le contenu de chaque partition (composant AADL `process`) est as-

socié à un segment mémoire (composant `memory`) et un environnement d'exécution (composant `virtual processor`).

Une autre règle assure que la mémoire principale est divisée en plusieurs sous-composants `memory`, spécifiant ainsi ses segments dans lesquels sont isolées les partitions.

L'architecture de ce cas d'étude remplissant toutes ces contraintes, ces règles sont donc validées.

Isolation temporelle

La règle validant l'isolation temporelle (section 4.3.2) garantit l'exécution des partitions à chaque cycle d'ordonnancement et vérifie la conformité de la *Major Frame* associée du noyau.

Pour cela, elle analyse chaque noyau (composant `processor`) de l'architecture et assure que chaque partition (composant `virtual processor`) qu'il exécute est référencée dans la liste des partitions ordonnancées (propriété `ARINC653::Slots_Allocation`).

La validation du *Major Frame* assure que la valeur (propriété `ARINC653::-Module_Major_Frame` est cohérente avec les tranches de temps allouées aux partitions : la valeur du *Major Frame* doit être égale à la somme des tranches de temps allouées aux partitions (propriété `ARINC653::Partitions_Slots`). Dans ce cas d'étude, six tranches de temps de 20 ms sont allouées et le *Major Frame* est défini à 120 ms. Cette règle est donc vérifiée.

Isolation spatiale

La validation de l'isolation spatiale assure que chaque partition dispose d'un unique segment mémoire pour le stockage de ses ressources (code, données, etc.). Elle vérifie également le bon dimenssionnement des segments en fonction du contenu de la partition.

Dans ce cas d'étude, chaque partition (composant `process`) est associée à un unique segment mémoire (composant `memory`) et spécifie ses caractéristiques (taille de la pile, du code, etc.). Une règle valide la taille de la partition en assurant que les besoins mémoire de la partition sont inférieurs à la taille du segment qui lui est alloué. La partition qui consomme le plus de mémoire requiert 87Ko alors que chaque partition dispose de 100Ko. Cette règle est donc validée.

6.2.4 Validation des mécanismes de *Health Monitoring*

Une fois le système généré, nous pouvons valider la bonne implantation de l'architecture et de ses mécanismes de sûreté. Pour cela, nous utilisons l'outil SPOQ présenté en section 5.5.2. Celui-ci supervise le système et reporte les appels systèmes réalisés par l'application.

Le but de cette validation consiste à assurer la bonne implantation des mécanismes de *Health Monitoring* en vérifiant que la politique spécifiée par l'utilisateur (au niveau du modèle AADL) est bien mise en œuvre par le système généré.

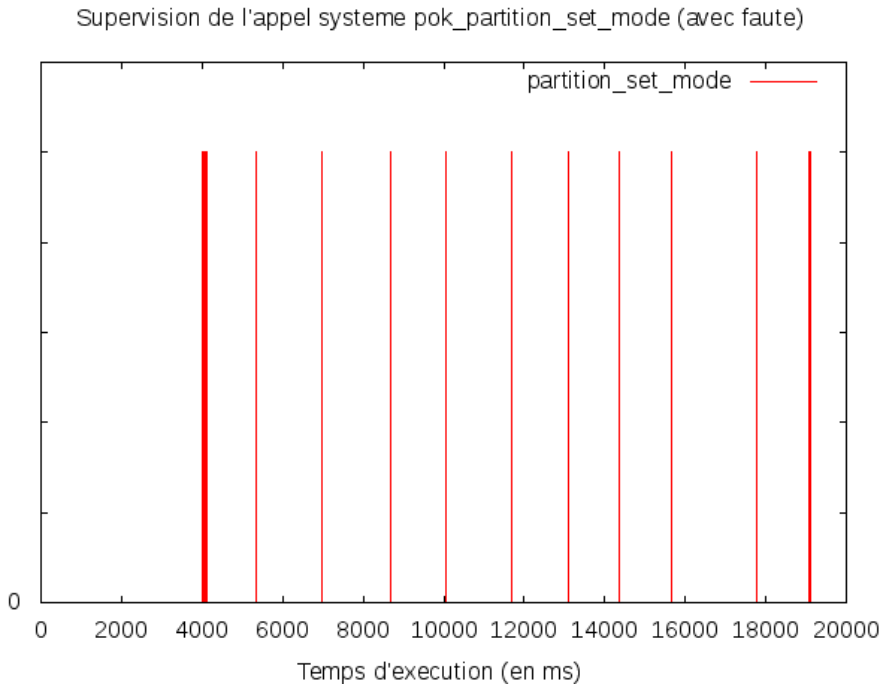


FIGURE 6.2 – Supervision de l'appel système `partition_set_mode` pour le cas d'étude *integrated* avec introduction de faute dans une tâche

A chaque faute survenant au sein d'une tâche, la partition associée est redémarrée. Nous avons donc introduit volontairement une faute dans une tâche : l'unique tâche de la partition *flight director* réalise une division par zéro au bout de dix itérations et entraîne donc son redémarrage.

Au niveau de l'implantation, l'appel système `pok_partition_set_mode` demande un changement d'état de la partition. Il est obligatoirement déclenché par la tâche d'initialisation pour basculer la partition en mode normal (les ressources sont alors initialisées et les autres tâches peuvent s'exécuter), mais est également utilisé pour changer de mode en cours d'exécution. En particulier, lorsqu'une partition doit être redémarrée suite à une erreur, son mode passe de l'état normal à initialisation.

Nous avons donc utilisé SPOQ (section 5.5.2) afin de tracer l'utilisation de cet appel système pendant les 20 premières secondes d'exploitation du système. La figure 6.2 illustre l'utilisation de cet appel système au cours cette supervision. On constate qu'à l'initialisation du système (environ 4 secondes), celui-ci est détecté plusieurs fois. Ceci s'explique par le fait que toutes les partitions ont terminé leur phase d'initialisation et entrent en mode nominal. Cet appel système est de nouveau intercepté au cours de l'exécution. Cette fois-ci, il s'agit du redémarrage de la partition contenant la tâche fautive : celle-ci générant une faute périodiquement (division par zéro), la partition redémarre à intervalles réguliers. On observe des différences dans les intervalles de temps, cela est principalement dû à la latence de redémarrage de la partition.

Nous avons réalisé la même analyse en supprimant la faute introduite. Le graphique correspondant est illustré en figure 6.3. On observe ici que l'appel

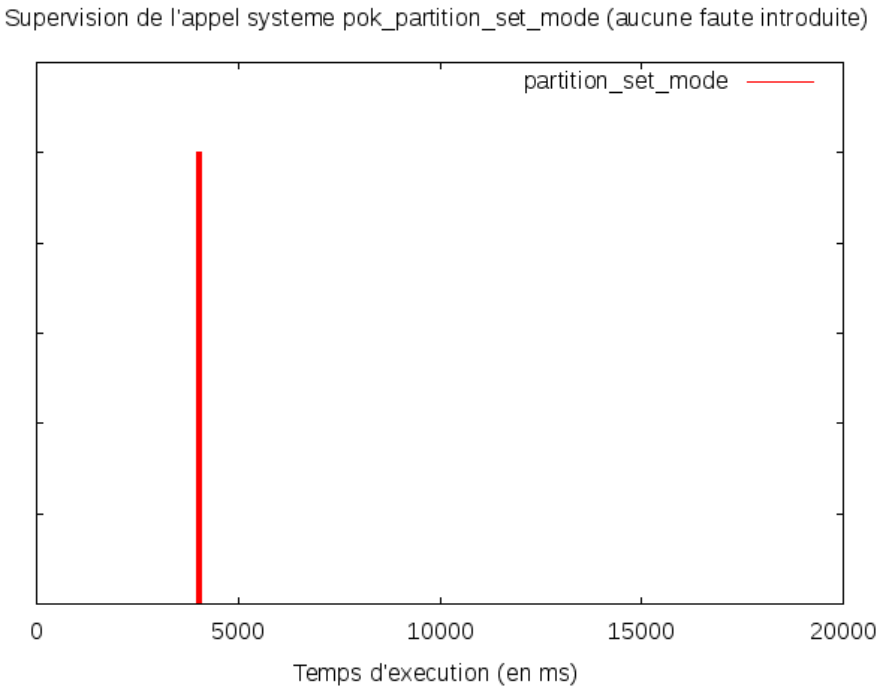


FIGURE 6.3 – Supervision de l'appel système `partition_set_mode` pour le cas d'étude *integrated* sans introduction de faute

système est appelé à l'initialisation du système (basculement du mode d'initialisation au mode normal), mais n'est jamais déclenché par la suite. Cela montre bien que la faute introduite déclenchait alors cet appel système et le redémarrage de la partition.

6.3 Cas d'étude *« MILS »*

Les paragraphes suivants présentent le cas d'étude *« MILS »* dédié à la sécurité. Dans un premier temps, nous décrivons les spécifications sous une forme textuelle, puis détaillons leur transposition en AADL en utilisant nos patrons de modélisation. Nous expliquons également quelles sont les règles de validation utilisées pour l'assurance de l'isolation de la sécurité à partir des modèles AADL créés. Enfin, la dernière section détaille le processus de vérification de l'isolation des données lors de l'exécution du système, assurant la bonne implantation des mécanismes de sécurité spécifiés.

6.3.1 Introduction, spécification

Cette architecture échange des données classifiées à différents niveaux de sécurité entre deux machines. Ce type de système peut être trouvé dans des applications militaires (comme le montre la figure 6.4) où le recours à plusieurs niveaux de classification est courant pour séparer les données qui transitent entre plusieurs grades d'une armée. Notre cas d'étude définit trois niveaux de sécurité, nous considérons que celles-ci seront destinées à trois types de de

personnes : les Aspirants (niveau non-classifié), les Colonels (niveau secret) et les Généraux (niveau top-secret).

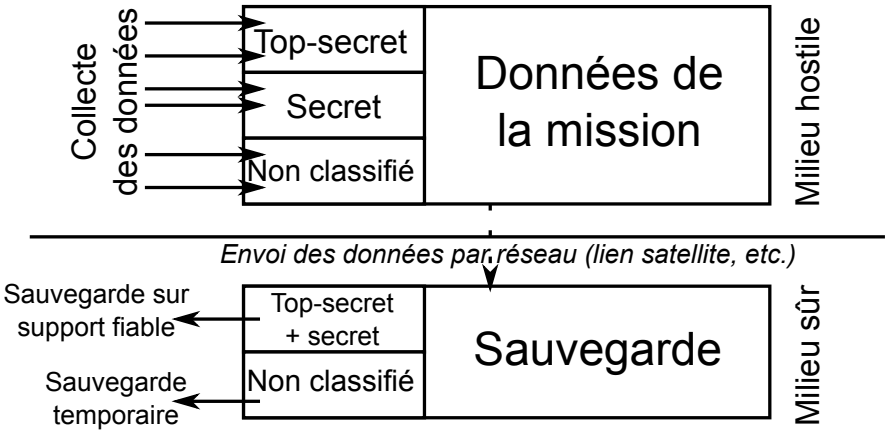


FIGURE 6.4 – Exemple d'application utilisant l'architecture du cas d'étude *« MILS »*

L'application de ce cas d'étude a pour but de sauvegarder les données d'une mission (figure 6.4) : une machine récolte les données (par exemple, l'ordre des Généraux, la position des Aspirants, etc.) et les envoie à une autre machine au moyen d'une connection réseau afin de les sauvegarder. La machine source réalise une triple séparation des données (chaque niveau de sécurité est confiné dans une partition) alors que la machine de sauvegarde ne sépare que les données non-classifiées (Aspirants) d'un côté et les données top-secrètes et secrètes (Colonels et Généraux) d'un autre. La machine de sauvegarde étant localisée dans un endroit sûr et non hostile, les deux niveaux sont confinés dans la même partition qui sauvegarde leurs données sur le même support. Les données des Aspirants sont quant à elles sauvegardées sur un support différent, de moins bonne qualité.

Le lien reliant les deux machines est non sécurisé. Nous devons considérer que les données qu'il transporte peuvent être lues par simple écoute, comme toute technique de reniflage (*sniffing*) sur un lien Ethernet. Par conséquent, chaque niveau de sécurité doit définir des mécanismes pour protéger (chiffrer) ses données lorsque celles-ci transitent par ce lien et plus le niveau de sécurité est élevé, plus ces mécanismes doivent être difficiles à analyser/casser.

Objectifs de ce cas d'étude

Ce cas d'étude a trois objectifs :

1. mettre en œuvre nos patrons de modélisation dédiés à la sécurité (section 4.6).
2. valider l'architecture à partir du modèle AADL au moyen de nos règles de validation (section 4.7). AADL.
3. assurer la bonne implantation des mécanismes de sécurité spécifiés.

Ces trois étapes montrent que la sécurité du système est assurée tout au cours du processus : à partir spécifications (utilisation de patrons de conception

et de règles de validation) jusqu'à l'implantation/exécution (génération des mécanismes de sécurité et vérification de leur conformité par rapport à la spécification).

Niveaux de sécurité

Trois niveaux de sécurité sont définis :

1. *top-secret* : niveau de sécurité le plus fort, il chiffre les données en utilisant l'algorithme *Blowfish* [103].
2. *secret* : niveau de sécurité intermédiaire, il utilise l'algorithme de chiffrement *DES* [19].
3. *non-classifié* : niveau de sécurité le moins élevé, il n'utilise aucun mécanisme de protection des données.

Pour faciliter l'analyse du système et des mécanismes de chiffrement, les données échangées par les partitions sont des entiers incrémentés à chaque itération des tâches. Les communications réseaux sont réalisées par des partitions dédiées contrôlant les interfaces connectées au réseau physique. Celles-ci sont évaluées au niveau non-classifié (aucun chiffrement).

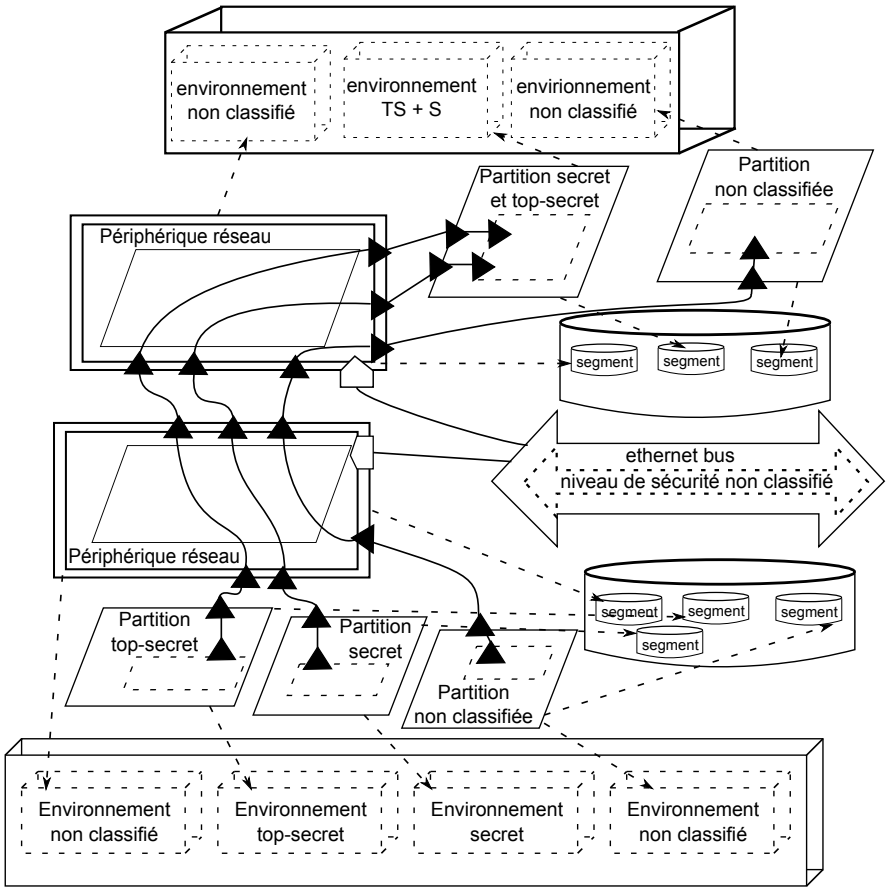


FIGURE 6.5 – Cas d'étude *« MILS »*

6.3.2 Spécifications AADL

La représentation graphique de la spécification AADL est illustrée dans la figure 6.5. La section B.4.2 des annexes (page 265) fournit la version textuelle incluant toutes les propriétés appropriées. Les paragraphes suivants présentent toutefois quelques composants afin d'illustrer l'utilisation de nos patrons de conception.

Niveaux de sécurité

La spécification des niveaux de sécurité (section 4.6) est réalisée par l'utilisation de composants `virtual bus`. Notre modèle contient trois composants de ce type, un par niveau de sécurité (listing 6.5). Chaque niveau est différencié par la propriété `POK::Security_Level` : plus le niveau de sécurité est important, plus sa valeur est grande.

Ces composants définissent leurs mécanismes de protection de données (algorithmes de chiffrement) en héritant de composants prédéfinis dans notre chaîne de développement (section B.3, page 240). Des propriétés (`POK::DES_Init`, `POK::DES_Key`, `POK::Blowfish_Init` et `POK::Blowfish_Key`) permettent de configurer les algorithmes de chiffrement DES et Blowfish associés aux niveaux secret et top-secret :

- `POK::DES_Init` indique la valeur d'initialisation du protocole DES. Elle est utilisée comme valeur aléatoire par le protocole.
- `POK::DES_Key` correspond à la clé de chiffrement symétrique utilisée par le protocole.
- `POK::Blowfish_Init` est similaire à la propriété `POK::DES_Init` pour l'algorithme Blowfish (initialisation de l'algorithme de chiffrement).
- `POK::DES_Key` est la clé de chiffrement symétrique de l'algorithme.

```
virtual bus implementation secret.i extends poklib::des.i
properties
   POK::Security_Level => 2;

   POK::DES_Key =>
          "{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
            0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87}";
   POK::DES_Init =>
         "{0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10}";
end secret.i;

virtual bus implementation topsecret.i extends poklib::blowfish.i
properties
   POK::Security_Level => 3;
   POK::Blowfish_Key =>
   "{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
           0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87}";
   POK::Blowfish_Init =>
         "{0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10}";
end topsecret.i;

virtual bus implementation unclassified.i extends poklib::ceasar.i
properties
   POK::Security_Level => 0;
end unclassified.i;
```

Listing 6.5 – Niveaux de sécurité du cas d'étude MILS

Noyau

Conformément au patron de modélisation (section 4.2.1), le noyau du système est représenté à l'aide d'un composant `processor`. Ce dernier représente la plate-forme qui exécute et assure l'isolation des partitions.

Ce cas d'étude contenant deux nœuds répartis, deux composants `processor` sont spécifiés : un pour le nœud émetteur et un pour le nœud récepteur.

Chaque composant `processor` contient plusieurs `virtual processor`, chacun représentant les environnement d'exécution des partitions. Ainsi, le `processor` du nœud émetteur contient trois composants `virtual processor` (un pour chaque niveau de sécurité) et celui du nœud récepteur (présenté en listing 6.6) n'en contient que deux (un pour les niveaux *top-secret* et *secret*, un autre pour le niveau non-classifié).

```
processor implementation kernel_receiver.i
subcomponents
   partition_topsecretandsecret  : virtual processor
                 runtime::partition_runtime.topsecret;
   partition_unclassified        : virtual processor
                 runtime::partition_runtime.unclassified;
   partition_network             : virtual processor
                 runtime::partition_runtime.unclassified
  {POK::Additional_Features => (stdlib, io, pci, stdio);};
properties
   POK::Major_Frame => 2000ms;
   POK::Scheduler => static;
   POK::Slots => (1000ms, 500ms, 500ms);
   POK::Slots_Allocation => (reference (partition_topsecretandsecret),
                             reference (partition_unclassified),
                             reference (partition_network));
end kernel_receiver.i;
```

Listing 6.6 – Définition en AADL du noyau de la partition réceptrice cas d'étude MILS

Partitions

Les partitions sont spécifiées (section 4.2.2), à l'aide de deux composants :

1. un composant `virtual processor`, représentant la plate-forme d'exécution de la partition. Celui-ci est soumis au contrôle du noyau (composant processor). Pour modéliser cette relation de dépendance, les `virtual processor` sont définis comme sous-composants d'un `processor` (listing 6.6)
2. un composant `process` qui modélise l'espace d'adressage de la partition et ses ressources. Ce composant contient les ressources de la partition (composants `thread`, `data`, etc.). Afin de représenter le lien entre les ressources de la partition et sa plate-forme d'exécution, ces deux composants sont associés au moyen de la propriété `Actual_Processor_Binding`.

Les partitions de ce cas d'étude ne contiennent qu'une tâche (composant `thread`) chargée d'envoyer ou recevoir des données. Le listing 6.7 montre la définition de la partition réceptrice des niveaux de sécurité secret et top-secrets.

```
virtual processor implementation partition_runtime.secretandtopsecret
properties
   Provided_Virtual_Bus_Class =>
    (classifier (layers::secret.i), classifier (layers::topsecret.i));
end partition_runtime.secretandtopsecret;


process partition_secret_and_topsecret_receive
features
   valueone : in data port types::integer
      {Allowed_Connection_Binding_Class =>
         (classifier (layers::topsecret.i));};
   valuetwo : in data port types::integer
      {Allowed_Connection_Binding_Class =>
         (classifier (layers::secret.i));};
end partition_secret_and_topsecret_receive;

process implementation partition_secret_and_topsecret_receive.i
subcomponents
   thr : thread threads::thr_receive_two_int.i;
connections
   port valueone -> thr.invalue1;
   port valuetwo -> thr.invalue2;
end partition_secret_and_topsecret_receive.i;
```

Listing 6.7 – Définition en AADL de la partition réceptrice du niveau secret et top-secret du cas d'étude MILS

Niveaux de sécurité fournis par chaque partition

Un niveau de sécurité est associé avec une partition (section 4.6.1) par l'association de la propriété `Provided_Virtual_Bus_Class` sur les composants `virtual processor` (environnement d'exécution des partitions). La valeur de cette propriété est une liste de composants `virtual bus` représentant les niveaux de sécurité supportés.

Dans le modèle présenté, chaque composant `virtual processor` définit une liste à un élément. Seule la partition réceptrice des niveaux *top-secret* et *secret* (composant `partition_runtime.secretandtopsecret`, listing 6.7) supporte deux niveaux de sécurité et définit une liste avec les deux `virtual bus`.

Niveaux de sécurité des interfaces de communication

Il est nécessaire de spécifier le niveau de sécurité associé à chaque interface de communication afin que le générateur de code soit en mesure de créer les appels de fonction chiffrant les données avant leur envoi/réception. Cette association entre l'interface de communication et son niveau de sécurité (section 4.6.1) requiert que la propriété AADL standard `Allowed_Connection_Binding_Class` soit associée à chaque `port` de la partition. Comme pour la propriété `Provided_Virtual_Bus_Class`, celle-ci prend pour valeur une liste de niveaux de sécurité (composant `virtual bus`). Toutefois, le patron de modélisation requiert que cette liste ne contienne qu'une seule valeur, un port n'utilisant qu'un seul niveau de sécurité (section 4.6.1).

Le listing 6.7 montre la définition de cette propriété pour les deux `ports` de la partition réceptrice des niveaux top-secret et secret : l'un reçoit les données

au niveau top-secret et est associée au `virtual bus layers::topsecret.i` tandis que l'autre reçoit les données au niveau secret est est associé à `layers::secret.i`.

Isolation spatiale

Le noyau de partitionnement (composant `processor`) assure l'isolation spatiale entre les partitions qu'il exécute. L'utilisateur doit cependant spécifier les segments mémoire et le déploiement des partitions sur chacun d'eux.

Le modèle de ce cas d'étude définit deux composants `memory` représentant la mémoire `RAM` de chaque nœud : `ram.sender` et `ram.receiver`. Chacun de ces composants contient des sous-composants `memory` qui définissent leur division en segments.

Le confinement de chaque partition dans un segment mémoire est spécifié (section 4.2.3) par la propriété `Actual_Memory_Binding` qui associe le contenu d'une partition (composant `process`) à son segment mémoire (composant `memory`).

Isolation temporelle

Dans un contexte de sécurité, l'isolation temporelle garantit que chaque partition est exécutée pendant un temps fixe et prédéfini, empêchant les attaques par inférence sur le temps d'exécution (un attaquant peut deviner le comportement d'un système à partir du temps d'exécution des sous-programmes).

La politique d'isolation temporelle (section 4.2.4) est assurée par le noyau de partitionnement, sa spécification est donc décrite par la modification du composant `processor` au moyen de deux propriétés :

- `POK::Slots` spécifie le nombre de tranches de temps allouées par le noyau.
- `POK::Slots_Allocation` définit l'allocation de ces tranches de temps aux partitions.

Le nœud émetteur (composant `kernel_sender.i`) exécute chacune de ses 4 partitions pendant 500ms. Le nœud récepteur (composant `kernel_receiver.i`, illustré dans le listing 6.6 en page 195) ne contient que trois partitions : il exécute la première pendant 1s et les deux autres pendant 500ms. Ces propriétés sont définies dans la version textuelle du modèle (sections B.4.2.6 et B.4.2.7 des annexes, page 270 et 270).

Spécification AADL des périphériques réseau

Le périphérique réseau chargé d'envoyer ou recevoir des données est modélisé par un composant AADL `device` (listing 6.8 et section B.4.2 des annexes page 265). Le pilote chargé de son exécution est confiné dans une partition dédiée.

Cette spécification est réalisée au moyen d'un composant `abstract` (`driver_container`, section B.4.2 des annexes, une version simplifiée est proposée en listing 6.8). Ce dernier contient un composant `process` décrivant le contenu de la partition gérant le périphérique. Il est par ailleurs associé au composant `device` via la propriété AADL standard `Device_Driver`.

Ce composant `device` est une partition système soumise aux mêmes règles d'isolation spatiale et temporelle que les partitions applicatives. Pour cette raison, il est associé à un composant `virtual processor` (association avec une plate-forme d'exécution) et `memory` (segment dans lequel le pilote est exécuté).

```
process implementation driver.i
subcomponents
   data_handler    : thread thr_handler.i;
   poller          : thread thr_poller.i;
properties
  POK::Needed_Memory_Size => 160 Kbyte;
end driver.i;

device rtl8029
features
   incoming_topsecret       : in data port types::integer;
   incoming_secret          : in data port types::integer;
   incoming_unclassified    : in data port types::integer;
   outgoing_topsecret       : out data port types::integer;
   outgoing_secret          : out data port types::integer;
   outgoing_unclassified    : out data port types::integer;
   ethernet_access          : requires bus access
                              runtime::ethernet.unsecure;
properties
   Initialize_Entrypoint => classifier (rtl8029::init);
   POK::Device_Name => "rtl8029";
end rtl8029;

abstract driver_container
end driver_container;

abstract implementation driver_container.i
subcomponents
   p : process driver.i;
end driver_container.i;

device implementation rtl8029.i
properties
   Device_Driver => classifier (rtl8029::driver_container.i);
end rtl8029.i;
```

Listing 6.8 – Définition en AADL du périphérique réseau du cas d'étude MILS

6.3.3 Validation des spécifications

Les paragraphes suivants présentent les règles de validation qui ont été appliquées sur ce cas d'étude. En particulier, elles valident la conformité du modèle avec les patrons de modélisation de l'architecture partitionnée, la bonne spécification des mécanismes d'isolation ainsi que le confinement des niveaux de sécurité.

Conformité de l'architecture partitionnée

Tout comme pour le cas d'étude *« integrated »* , les règles de validation (section 4.3.1) analysent le modèle AADL et assurent qu'il décrit une architecture partitionnée (utilisation de `virtual processor`, `process` et `processor`, etc.).

Dans le cas présent, le modèle est validé : l'environnement d'exécution de chaque partition est spécifié au moyen d'un composant `virtual processor`,

la mémoire du système ainsi que sa division en plusieurs segments est correctement décrite et la politique de déploiement respecte les restrictions des architectures partitionnées (partition confinée dans un environnement d'exécution et segment mémoire).

Isolation temporelle

La validation de l'isolation temporelle (section 4.3.2) consiste à assurer que chaque partition est exécutée par le noyau à chaque cycle d'ordonnancement et que la *Major Frame* est cohérente par rapport aux tranches de temps allouées à l'exécution des partitions.

Les deux noyaux de partitionnement définissent une *Major Frame* de 2000 ms. Le noyau émetteur exécute chacune de ses quatre partitions pendant 500ms. Le second contient trois partitions, exécute la première pendant 1000ms et les deux restantes pendant 500ms. Par conséquent, les valeurs sont cohérentes : 4×500 (temps requis par les partitions du premier nœud) $= 1000 + 2 \times 500$ (temps requis par les partitions du second nœud) $= 2000$ (major frame des deux noyaux).

Isolation spatiale

La validation de l'isolation spatiale consiste à vérifier que chaque partition est confinée au sein d'un segment mémoire dédié (section 4.3.3).

Le modèle (figure 6.5 et listing B.4.2 des annexes, page 265) spécifie la mémoire principale (RAM) de chaque nœud et sa décomposition en segments (composants `memory` décomposés en sous-composants). Chaque segment est par ailleurs associé à une partition (composant `process`). Par conséquent, les règles valident la conformité du modèle.

Mécanismes de sécurité

Dans la description de l'architecture, chaque composant `virtual bus` représentant un niveau de sécurité étend un autre composant définissant les mécanismes de sécurité. Par exemple, le `virtual bus topsecret` (listing 6.5, page 194) hérite du `virtual bus` nommé `des` et implantant les mécanismes de chiffrement du *Data Encryption Standard* [76]. En outre, ce composant hérité défini les propriétés adéquates pour l'implantation de son algorithme de protection des données.

Le composant `virtual bus` représentant le niveau de sécurité *topsecret* hérite quant à lui du `virtual bus` nommé `blowfish` qui spécifie les mécanismes de protection de données associés au protocole de chiffrement *Blowfish* [103].

Les règles de validation assurant la bonne implantation des mécanismes de sécurité (section 4.7.1) analysent donc l'architecture sans rapporter d'erreur. En outre, cela assure la bonne configuration des mécanismes de protection des données pour chaque niveau de sécurité.

Enfin, les règles d'analyse vérifiant la conformité des niveaux de sécurité (section 4.7.1) et leurs potentiels conflits de configuration : chaque niveau de sécurité utilisant des mécanismes de protection différents, aucune erreur n'est reportée par nos outils de validation de modèles.

6.3.4 Validation de l'implantation

Une fois les spécifications validées, les programmes des deux nœuds sont automatiquement générés par notre chaîne d'outils :

1. Le générateur de code configure la politique d'isolation temporelle et spatiale. Il créé également le code responsable du chiffrement/déchiffrement des données à leur envoi/réception.
2. POK fournit les services nécessaires à l'exécution de l'architecture : chargement et isolation des partitions, algorithmes de chiffrement, etc.

Nous utilisons l'outil QEMU [13], pour exécuter les deux applications et simuler le réseau les connectant. Ce réseau *« virtuel »* émule un réseau Ethernet traditionnel au dessus du protocole TCP/IP. Le trafic échangé lors de l'exécution des deux applications générées peut être capturé à l'aide d'un outil de capture de paquets réseaux, tel *Wireshark* [86]. Afin de procéder à l'analyse des trames échangées, nous avons mis en place un protocole de test illustré dans la figure 6.6. Chaque application générée est exécutée dans sa machine virtuelle et le trafic échangé entre elles est capturé par *Wireshark*.

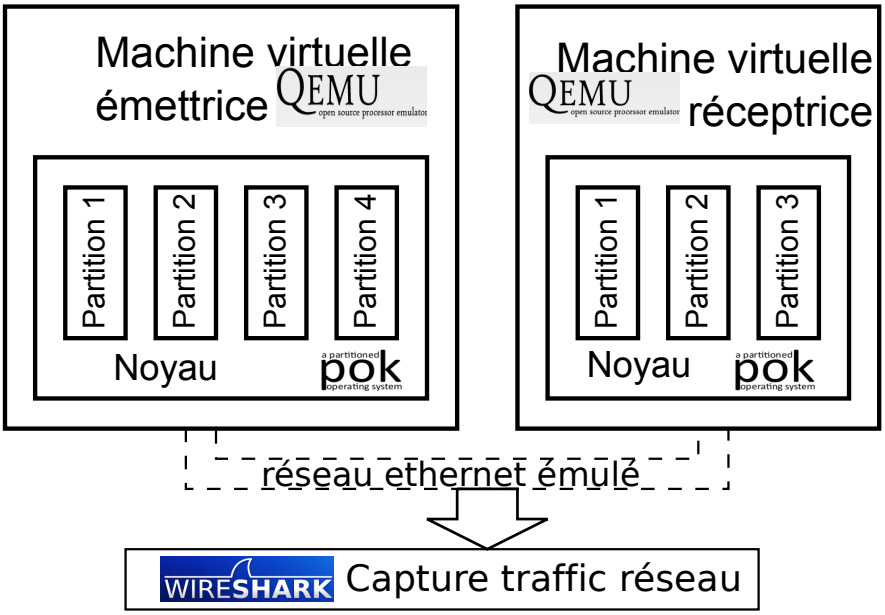


FIGURE 6.6 – Protocole de test pour la vérification de l'implantation des mécanismes de sécurité du cas d'étude *« MILS »*

Dans un premier temps, nous capturons les données envoyées par les nœuds classifiés au niveau *secret* et *topsecret*. Ces données apparaissent sur le réseau comme chiffrées. Puis, nous reproduisons le chiffrement des données par un programme indépendant (dont le code fourni en annexe, section B.5.1, page 275) en utilisant la même configuration (clés, etc.) que les partitions.

Enfin, nous comparons les données chiffrées capturées lors de l'exécution des applications générées et celles obtenues par notre programme indépendant. L'aspect applicatif étant simple (envoi d'un nombre incrémenté à chaque période), il est facile de reproduire les valeurs échangées entre les applications.

Le tableau 6.1 rapporte les données issues de la reproduction du chiffrement par notre programme et celles transmises par les applications générées. Par souci de clarté, les données relatives à la gestion des protocoles réseaux (en-tête de paquet, etc.) ont été supprimées.

Les paquets capturés sont conformes au protocole Ethernet, qui utilise la convention de représentation des données *Gros-Boutiste*. L'ordre des octets est donc différent de celui du programme exécuté sur une machine qui utilise une représentation *Petit-Boutiste*. Pour ces raisons, les octets sont inversés entre les données obtenues par capture des paquets et celles obtenues par reproduction.

Une fois une convention de représentation choisie, les données sont alors identiques : les données envoyées sur le canal chiffré (niveau de sécurité *secret*) correspondent à celles reproduites par le programme de chiffrement indépendant. De plus, les valeurs envoyées sur le canal non-chiffré (niveau de sécurité *non classifiée*) correspondent bien à un entier incrémenté à chaque fois. Enfin, le bon affichage des données applicatives dans les partitions réceptrices montrent que celles-ci font appel aux mécanismes adéquats pour déchiffrer les données avant leur transmission à la couche applicative.

Itération	**Reproduction *secret***	**Capture *secret***	**Capture *non classifié***
0	`af:26:c6:12`	`12:c6:26:af`	`00:00:00:00`
1	`46:d7:49:9f`	`9f:49:d7:46`	`01:00:00:00`
2	`fa:5f:37:fa`	`fa:37:5f:fa`	`02:00:00:00`
3	`ee:10:d3:96`	`96:d3:10:ee`	`03:00:00:00`
4	`40:b8:6f:27`	`27:6f:8b:40`	`04:00:00:00`
5	`8b:61:6d:dd`	`dd:6d:61:8b`	`05:00:00:00`

Table 6.1 – Comparaison des données obtenues par reproduction du chiffrement et des données capturées lors de l'exécution

6.4 Analyse de la couverture de code

Une analyse de couverture de code a été réalisée sur chaque cas d'étude afin d'évaluer notre approche avec les exigences de la norme DO178B.

6.4.1 Cas d'étude *integrated*

Les taux de couverture du cas d'étude *« integrated »* sont présentés dans le tableau 6.2, ceux du cas d'étude *« MILS »* dans le tableau 6.3.

Composant	**Taux de couverture**
Noyau	69%
Partition *flightmanager*	74%
Partition *flightdirector*	54%
Partition *displaymanager*	54%
Partition *pagecontentmanager*	54%
Partition *warningannunciationmanager*	47%
Partition *hardwaredisplay*	52%

Table 6.2 – Taux de couverture de chaque composant généré pour le cas d'étude *« integrated »*

Les taux relevés dans le cas d'étude *integrated* montrent des résultats équivalents pour les partitions *flightdirector*, *displaymanager* et *pagecontentmanager*, celles-ci ont toutes un taux de couverture de 54%. Ceci est dû au fait que ces partitions utilisent les mêmes services et disposent des mêmes ressources : chacune exécute une seule tâche et utilise le service de communication inter-partitions. Embarquant un nombre de ressources équivalent et utilisant les mêmes services, il est donc normal que leur taux de couverture soit similaire.

Le taux de couverture de la partition *warningannunciationmanager* est moins élevé alors qu'elle n'utilise pas le service de communication. Ce faible taux est expliqué par l'inclusion de fonctionnalités non utilisées : certaines fonctionnalités sont toujours incluses par l'étape de génération de code alors qu'elles ne sont pas utilisées par le système. Leur présence et leur non exécution réduit alors le taux de couverture. Ce résultat montre que le code généré de la configuration des services de la plate-forme peuvent être optimisés.

Le taux de couverture le plus élevé est celui de la partition *flightmanager* (74%). Ce résultat s'explique par le nombre plus important de fonctions qu'elle utilise (communications intra- et inter-partitions) et donc, une exécution plus exhaustive des services de l'environnement d'exécution.

Enfin, le code du noyau est couvert à 69%. La raison de ce faible taux de couverture vient de l'utilisation du service de *Health Monitoring*. Ce dernier active la gestion des exceptions dans le noyau. Cependant, ces dernières n'étant pas toutes levées à l'exécution, une partie du code relatif à ce service n'est donc pas exécuté et réduit le taux de couverture.

	Composant	**Taux de couverture**
Emetteur	Noyau	79%
	Partition driver	55%
	Partition *top-secret*	58%
	Partition *secret*	48%
	Partition *non-classifié*	68%
Récept.	Noyau	78%
	Partition driver	48%
	Partition *top-secret* et *secret*	52%
	Partition *non-classifié*	67%

TABLE 6.3 – Taux de couverture de chaque composant généré dans le cas d'étude *MILS*

6.4.2 Cas d'étude *MILS*

Dans le cadre du cas d'étude *MILS*, le taux de couverture du noyau est proche de 80% (79% dans le cas du nœud émetteur, 78% pour le nœud récepteur). Ce meilleur taux de couverture (par rapport au cas d'étude *« integrated »*) s'explique par l'absence du service *« Health Monitoring »* : aucune couche de l'architecture n'utilise ce service qui est ainsi désactivé par le générateur de code.

A l'inverse, le taux de couverture des partitions est plus faible, excepté pour les partitions *non classifiée*. Ces résultats s'expliquent par les couches de chiffrement de la plate-forme d'exécution : celles-ci ayant été reprises du projet *OpenSSL*, certaines parties du code ne sont pas exécutées. Une amélioration

de ces résultats demanderait alors une adaptation de ces fonctions importées aux outils de génération de code.

6.4.3 Analyse des résultats

Ces résultats montrent l'importance de la génération de code à partir des spécifications. L'étape de génération de code permet de n'inclure que les services nécessaires (figure 6.7) dans chaque couche de l'architecture. Elle adapte chaque couche de l'architecture partitionnée et supprime les services qui sont inutiles en fonctions des ses caractéristiques.

Dans le cas du cas d'étude *« MILS »* , le noyau n'utilise pas le service de gestion d'erreur, celui-ci est donc automatiquement supprimé des fonctionnalités du noyau. De la même manière, les services requis pour implanter chaque partition sont différents :

- la partition gérant la carte réseau (1 sur la figure 6.7) ne nécessite pas d'API spécifique (*libm*, *libc*, etc.), uniquement la gestion des tâches, l'interfaçage avec le noyau et la gestion des pilotes.
- les partitions utilisant les niveaux de sécurité *secret* et *top-secret* (2 sur la figure 6.7) requièrent des algorithmes de chiffrement en plus de la gestion des tâches et de l'interfaçage avec le noyau. La phase de génération de code les inclut automatiquement.
- la partition manipulant des données non classifiées (3 sur la figure 6.7 ne requiert de protocoles de chiffrement et par conséquent, ne doit contenir que le service de gestion des tâches et l'interfaçage avec le noyau.

Cette phase de sélection des fonctionnalités, en plus de limiter fortement l'empreinte mémoire (section 6.5), maximise la couverture de code. Enfin, le caractère automatique du processus permet d'éviter les écueils traditionnellement introduits par les développeurs dans la production de ces configurations.

Cependant, ces résultats peuvent sembler faibles. Il faut toutefois considérer que dans le cas présent, chaque couche du système contient un environnement d'exécution dédié contenant du code responsable de la gestion des fonctions dépendantes de la machine et difficilement désactivable. Les approches d'analyse de couverture de code traditionnelles s'appuient sur des environnement d'exécution certifiés. Par conséquent, seul le code applicatif nécessite d'être couvert, celui de la runtime étant *de facto* considéré comme couvert, correct et sûr.

6.5 Étude de l'empreinte mémoire

Le matériel embarqué utilisé dans la production de systèmes critiques a une capacité mémoire limitée : l'empreinte mémoire des applications est donc un élément déterminant de leur conception. La construction d'un système trop important en terme d'espace mémoire complique son intégration sur le matériel et implique une hausse des coûts de production (ajout de mémoire sur chaque matériel exécutant le logiciel, etc.). Pour ces raisons, la taille des composants inclus dans un système doit être la plus réduite possible.

L'empreinte mémoire de chaque composant généré pour les deux études sont rapportés dans les tableaux 6.4 (cas d'étude *« integrated »*) et 6.6 (cas

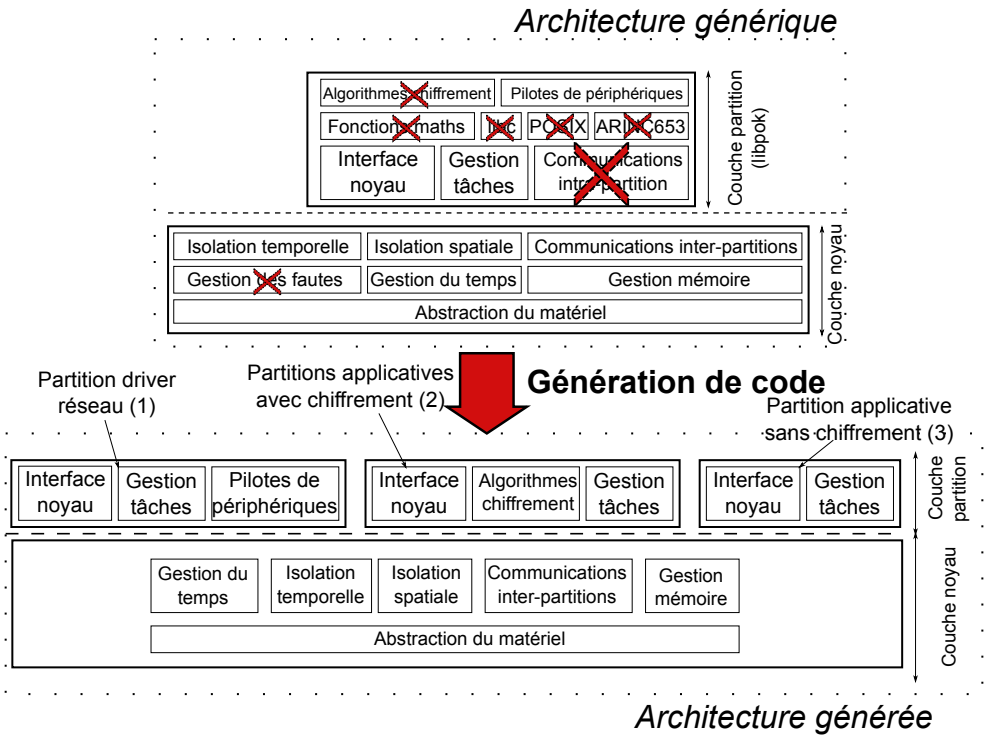


FIGURE 6.7 – Impact de la génération de code sur l'architecture et la couverture de code pour le cas d'étude *M*ILS

d'étude *« MILS »*).

6.5.1 Protocole de test

Le protocole de test suivi consiste à générer les applications de nos cas d'études, les compiler et à supprimer toute information de déboggage inclue dans les fichiers via l'outil `strip` de la chaîne d'outils `binutils`. Afin que les résultats reportés soient reproductibles, nous indiquons la version des outils utilisés et leur disponibilité :

- **GCC** issue de GNAT GPL 2009, téléchargeable sur `http://libre.adacore.com`.
- **Ocarina**, version de développement, *snapshot* issue de la révision 7222. Disponible sur `http://aadl.telecom-paristech.fr`.
- **POK**, version de développement, *snapshot* issue de la révision 20100223 téléchargeable à l'adresse `http://pok.gunnm.org`. Les tests ont été réalisés sur la plate-forme *x86*.
- **binutils** version 2.20. Disponible sur `http://www.gnu.org/software/binutils/`.
- **xcov** version de développement, issue de la révision 2488 disponible à l'adresse suivante : `http://forge.open-do.org`.

Composant	Taille totale	Taille architecture	Taille applicatif
Noyau	23 Ko	N/A	N/A
Partition *flightmanager*	22 Ko	20 Ko	1.2 Ko
Partition *flightdirector*	17 Ko	16 Ko	700 o
Partition *displaymanager*	15 Ko	14 Ko	604 o
Partition *pagecontentmanager*	15 Ko	14 Ko	612 o
Partition *warningannunciationmanager*	13 Ko	12 Ko	620 o
Partition *hardwaredisplay*	15 Ko	14 Ko	600 o

TABLE 6.4 – Taille de chaque composant généré pour le cas d'étude *« integrated »*

6.5.2 Résultats du cas d'étude *integrated*

La taille des partitions du cas d'étude *« integrated »* (tableau 6.4) est inférieure à 30 Ko. Elle correspond à la somme de la taille de leur environnement d'exécution et du code applicatif qu'elles exécutent. La taille du noyau correspond à toutes ses fonctions et ressources qu'il inclut. Dans ce cas d'étude, le noyau utilise plusieurs canaux de communication inter-partitions, ce qui a pour conséquence d'augmenter ses exigences mémoire (introduction de tampons d'émission/réception pour chaque canal).

Cependant, nous constatons que la taille des composants est dépendante de leurs exigences et leurs ressources. Pour procéder à cette analyse, le tableau 6.5 reporte les fonctions et ressources utilisées par chaque partition. Si la majorité a une taille inférieure à 20 Ko et contient qu'une tâche exécutant un sous-programme, on peut observer que la partition *flightmanager* dépasse le seuil des 20 Ko. Cela est dû au fait qu'elle utilise davantage de ressources (7 tâches) et de fonctionnalités (utilisation du mécanisme de communication intra-partition).

6.5.3 Résultats du cas d'étude *MILS*

Les résultats du cas d'étude *MILS* sont présentés dans le tableau 6.6. Le noyau est particulièrement léger (19Ko), tout comme les partitions. Ces résultats s'expliquent par le peu de fonctionnalités qu'ils utilisent (la figure 6.7 détaille les fonctionnalités embarquées dans chaque couche de l'architecture).

L'empreinte mémoire la plus importante concerne la partition exécutant les pilotes de périphériques. Celle-ci embarque davantage de code que les autres, les routines de gestion du périphérique réseau étant conséquentes en terme de taille.

Les autres partitions ont une taille dépendante des fonctionnalités qui sont incluses via le processus de génération de code. Celles utilisant des protocoles de chiffrement embarquent alors les algorithmes qui leur sont associés, leur ajoutant alors du code et augmentant leur taille.

Partition	Tâches	Fonctionnalités	Taille
Flightmanager	7	– API ARINC653 – Gestion des tâches – Communication intra-partition – Communication inter-partitions	22 Ko
Flightdirector	1	– API ARINC653 – Gestion des tâches – Communication inter-partitions	17 Ko
Displaymanager	1	– API ARINC653 – Gestion des tâches – Communication inter-partitions	15 Ko
Pagecontentmanager	1	– API ARINC653 – Gestion des tâches – Communication inter-partitions	15 Ko
Warningannunciation-manager	1	– API ARINC653 – Gestion des tâches	13 Ko
Hardwaredisplay	1	– API ARINC653 – Gestion des tâches – Communication intra-partition – Communication inter-partitions	15 Ko

TABLE 6.5 – Résumé des ressources et fonctions utilisées par chaque partition du cas d'étude *« integrated »*

6.5.4 Analyse des résultats

Ces résultats (tableaux 6.4 et 6.6) démontrent bien le caractère minimal de notre approche : il nous est possible déduire le coût mémoire associé à chaque composant. Dans le cas du cas d'étude *M*ILS (tableau 6.6), les partitions émettrices de données classifiées *top-secret* et *secret* et utilisant des mécanismes de chiffrement ont une taille mémoire supérieure de 8Ko et 12Ko par rapport à la partition n'utilisant aucun mécanisme de chiffrement (utilisation du niveau de sécurité *non classifié*). Celle-ci n'inclut que les services d'interfaçage avec le noyau, la gestion des tâches et a une taille faible (figure 6.7) : 14Ko. On peut donc en déduire qu'une partition manipulant simultanément les niveaux *top-secret* et *secret* aura une taille de 34Ko (14Ko pour les services d'interfaçage et de gestion de tâche, 8Ko pour la protection des données du niveau *top-secret* et 12Ko pour la protection des données au niveau *secret*). Ceci est confirmé par la partition réceptrice des données classifiées : celle-ci utilise simultanément ces deux niveaux de sécurité et a une taille de 35Ko (la différence provenant dans l'ajout de ressources pour gérer deux données simultanément).

Ces résultats illustrent encore une fois le caractère *« minimal »* de notre approche et le strict respect des exigences des spécifications du système.

	Composant	Taille totale	Taille architecture	Taille applicatif
Émetteur	Noyau	19 Ko	N/A	N/A
	Partition driver	53 Ko	N/A	N/A
	Partition *top-secret*	22 Ko	21 Ko	888o
	Partition *secret*	26 Ko	25 Ko	888o
	Partition *non-classifié*	14 Ko	13 Ko	888o
Récepteur	Noyau	19 Ko	N/A	N/A
	Partition driver	53 Ko	N/A	N/A
	Partition *top-secret* et *secret*	35 Ko	34 Ko	800 o
	Partition *non-classifié*	14 Ko	13 Ko	800 o

TABLE 6.6 – Taille de chaque composant généré pour le cas d'étude *MILS*

6.5.5 Comparaison avec les travaux précédents

Par rapport aux précédents travaux sur la génération de code [116], notre approche réduit considérablement la taille des applications créées. Les deux approches existantes de génération de code à partir de modèles AADL produisent du C [31] et de l'Ada [116]. Les applications qu'elles génèrent sont intégrées avec un système d'exploitation traditionnel (tel Linux ou RTEMS).

L'empreinte mémoire des applications générées était toujours supérieure à 400Ko. Cette différence de résultats s'explique par deux facteurs :

1. les programmes générés sont exécutés au-dessus d'un système d'exploitation générique et nécessite l'inclusion de bibliothèques tierces (*runtime* Ada dans le cas des travaux portant sur la génération de code Ada, libairie C compilée statiquement pour le code C généré). Au contraire, notre approche intègre le code généré avec un environnement d'exécution léger et adapté à chaque couche du système en fonction de ses exigences.
2. aucune sélection des services n'est réalisée par le générateur. Par conséquent, les applications générées incluent de nombreuses fonctionnalités qui ne sont pas utiles mais toutefois inclues dans l'application. Notre approche adresse ce problème en désactivant *par défaut* toute fonctionnalité et active chacune d'entre elle en fonction des besoins de l'application.

Enfin, à titre de comparaison, un système d'exploitation dédié au temps-réel comme RTEMS [79] a une empreinte mémoire supérieure à 100Ko même lorsqu'il n'inclut que le minimum de fonctionnalités. A l'inverse, notre approche de configuration fine de chaque niveau de l'architecture assure une empreinte mémoire optimisée qui permet de générer des noyaux dont la taille est inférieure à 20Ko.

Synthèse de ce chapitre

Ce chapitre du manuscrit a mis en pratique la méthode développée dans les chapitres précédents au travers de deux cas d'études : l'un concentré sur la sûreté, l'autre, sur la sécurité. Il détaille l'utilisation de nos patrons de conception et nos règles de validation pour l'assurance de ces deux exigences. L'aspect implantation automatique permet la bonne implantation des spécifications et la certification du système valide leur respect à l'exécution (assurance de la bonne implantation de la politique de *Health Monitoring* ou d'isolation des données).
Enfin, les études de performances ont montré la conformité des applications produites aux contraintes des systèmes embarqués/temps-réel. Les empreintes mémoires reportées sont minimales, inférieures aux travaux précédents qui constituaient déjà une avancée significative dans la réduction de la consommation mémoire. La génération de code de notre processus améliore ces résultats en réduisant les fonctionnalités des systèmes générés. Cette sélection des services de la plate-forme d'exécution réduit fortement l'empreinte mémoire et maximise la couverture de code.

Chapitre 7

Conclusions et Perspectives

Sommaire du chapitre

7.1 Conclusions 209
7.1.1 Assurance de sécurité et de sûreté de bout en bout . 209
7.1.2 Réunion de la sécurité et de la sûreté 211
7.1.3 Bénéfices de l'approche dirigée par les modèles . . . 211
7.1.4 POK, premier système d'exploitation libre pour la création de systèmes critiques sûrs et sécurisés . . . 212
7.2 Perspectives 212
7.2.1 Description plus fine de l'architecture 213
7.2.2 Vers une meilleure certification 213
7.2.3 Amélioration de notre plate-forme, POK 214
7.2.4 Développement de la communauté de POK 214

Les précédents chapitres ont décrit notre approche pour la construction de systèmes critiques sûrs et sécurisés. Cette dernière partie du manuscrit procède à un rappel de nos contributions, effectue un retour sur nos travaux et indique les potentiels axes de recherche en vue de leur amélioration.

7.1 Conclusions

Les sections qui suivent rappellent nos contributions et résultats, mettant en valeur chaque aspect de notre approche (modélisation, validation, génération de code et certification).

7.1.1 Assurance de sécurité et de sûreté de bout en bout

Les travaux présentés dans ce manuscrit proposent une approche visant à garantir les contraintes de sécurité dans la construction d'applications critiques. Celle-ci s'appuie sur :

1. des patrons de modélisation ;
2. des règles de validation ;
3. une génération automatique de code qui configure/crée le système à partir de modèles et une plate-forme fournissant les services de sécurité/sûreté requis à l'exécution ;
4. une certification des aspects sûreté et de sécurité et une assurance de conformité avec les standards applicables aux systèmes critiques.

Les patrons de modélisation proposés s'appuient sur le langage AADL et bénéficient de ses avantages (typage fort, notation graphique et textuelle, description des aspects logiciels et matériels). Ils précisent la sémantique des composants de ce langage pour la spécification de systèmes partitionnés et leurs contraintes de sécurité/sûreté. Leur utilisation définit les composants et les propriétés que le concepteur d'applications doit utiliser. En outre, ils limitent les erreurs au sein des spécifications en empêchant l'utilisateur d'utiliser de composants ou de propriétés non pertinentes. Ces patrons correspondent à la méthode de spécification requise par le premier objectif de ces travaux (*« méthode de spécification et de validation »* , section 1.4).

Les règles de validation assurent que la spécification écrite par le concepteur d'application est conforme aux règles des architectures partitionnées (définition de la politique d'isolation temporelle, spatiale, etc.). Elles analysent également les modèles afin de détecter des erreurs de sécurité ou de sûreté (par exemple : politique de détection d'erreur incomplète ne gérant pas certaines fautes, faille potentielle de sécurité, etc.). Leur utilisation garantit le respect de caractéristiques de sécurité/sûreté (isolation des niveaux de criticité/sécurité, etc.) avant de procéder à l'implantation du système, réduisant les erreurs traditionnellement découvertes lors des étapes suivantes du développement. Ces règles de validation complètent notre méthode de spécification et remplissent les exigences de notre premier objectif (*« méthode de spécification et de validation »* , section 1.4).

L'étape de génération de code produit l'application à partir des spécifications validées et configure automatiquement la plate-forme d'exécution à partir des exigences et contraintes du modèle. Un générateur de code transforme les composants AADL et leurs propriétés (caractéristiques de sécurité/-sûreté du système) en code C. L'analyse du modèle déduit les services requis à l'exécution et configure finement l'architecture pour n'inclure que les services nécessaires. Afin de rendre le système exécutable, notre chaîne d'outils intègre le code généré avec une plate-forme d'exécution qui fournit les services de sécurité et de sûreté requis (détection et recouvrement d'erreur, algorithmes de chiffrement, ...). Ces deux aspects (génération de code et plate-forme d'exécution) remplissent le second objectif de nos travaux (*« plate-forme garantissant la sécurité/sûreté et génération de code à partir des spécifications »* , section 1.4).

L'aspect certification de notre approche assure que l'application produite est conforme aux exigences des spécifications. Des outils analysent le système et assurent la bonne implantation des mécanismes de sécurité et de sûreté ainsi que la conformité aux standards tels DO178B. L'analyse des mécanismes de sécurité et de sûreté est réalisée par des programmes qui supervisent l'exécution du système : une machine virtuelle reporte les opérations effectuées par le système et l'utilisateur vérifie leur conformité avec la spécification. La validation des standards applicables aux systèmes critiques (tels DO178B) est réalisée par une chaîne d'outils (xcov, [22] qui produit automatiquement un rapport de couverture de code pour chaque niveau de l'architecture (noyau, partitions). Ces aspects remplissent le troisième objectif défini en introduction (*« certification du système et vérification du respect des contraintes de sécurité/sûreté »* , section 1.4).

7.1.2 Réunion de la sécurité et de la sûreté

L'approche proposée analyse et regroupe les services communs à la sécurité et la sûreté et montre leur complémentarité. En particulier, l'utilisation de l'architecture partitionnée apporte des bénéfices pour l'assurance de ces deux exigences :

- L'isolation temporelle garantit que chaque partition dispose d'une tranche de temps fixe et prédéterminée pour l'exécution de ses tâches (assurance de sûreté). Cette caractéristique évite également d'inférer des informations à partir de leur temps d'exécution (assurance de sécurité).
- L'isolation spatiale stocke chaque partition dans un segment de mémoire dédié. Cette propriété empêche une application fautive d'écrire dans les espaces réservés aux autres partitions en vue de générer des fautes (objectif de sûreté). Elle sépare également les partitions stockant des données classifiées à des niveaux hétérogènes (objectif de sécurité).
- Le confinement des communications garantit qu'une application fautive ne peut échanger des données qui ne respectent pas la politique de communication décrite par l'utilisateur (objectif de sûreté, seuls les canaux spécifiés sont autorisés). Cette caractéristique assure également l'isolation des données les niveaux de sécurité.

Certains services restent toutefois spécifiques à la sécurité ou la sûreté, tels l'implantation des mécanismes de recouvrement d'erreur (sûreté) ou l'utilisation d'algorithmes pour le chiffrement des données (sécurité). Pour chacun d'eux, nous proposons des patrons de modélisation, des règles de validation et de génération pour les intégrer dans notre processus de développement.

Enfin, ces aspects sont validés de bout en bout de notre chaîne de production, des spécifications jusqu'à l'exécution. Cette approche permet d'analyser l'impact des aspects sûreté sur la sécurité et de détecter le plus tôt possible toute erreur potentielle.

7.1.3 Bénéfices de l'approche dirigée par les modèles

Le cycle de développement proposé utilise le langage AADL comme langage *pivot* pour chaque aspect (modélisation, validation, implantation et certification). Les présents travaux ont montré l'impact du choix du langage de modélisation et de sa sémantique. Le typage fort et le caractère extensible d'AADL nous a permis de définir des patrons de modélisation appropriés aux systèmes appropriés, mais également d'exploiter ces modèles pour diriger chaque aspect de notre cycle de développement (validation, génération de code, certification).

La chaîne d'outils proposée montre que les modèles peuvent être exploités à chaque étape du développement d'un système critique. Elle a mis en lumière la pertinence de cette utilisation au niveau des spécifications (utilisation de patrons et validation des mécanismes de sécurité/sûreté), mais surtout au niveau de la génération automatique d'application et de leur certification.

La possibilité de définir des propriétés additionnelles au langage nous a amener à modéliser finement les exigences du système et a permis d'optimiser les applications produites. En outre, cette caractéristique impacte les performances du système en terme d'empreinte mémoire (réduction des fonc-

tionnalités) et de couverture de code. Les résultats obtenus à l'issue de nos travaux constituent une première base et gagnent toutefois à être améliorés. Les sections 7.2.1 et 7.2.2 indiquent en ce sens plusieurs axes d'évolution.

Enfin, l'utilisation d'un unique format de représentation à chaque étape du développement évite le recours à différents formalismes (documents textuels, code, etc.) et facilite l'analyse du système. En particulier, dans notre chaîne d'outils, les mêmes spécifications sont réutilisés pour les aspects validation, implantation et certification. L'utilisation d'une unique représentation garantit une consistance du développement (assurance que les mêmes exigences sont analysées à chaque étape) et évite les écueils dûs à l'utilisation de différentes spécifications parfois contradictoires (interprétation des spécifications sous forme textuelle, etc.).

7.1.4 POK, premier système d'exploitation libre pour la création de systèmes critiques sûrs et sécurisés

La plate-forme d'exécution produite à l'issue de nos travaux constitue le premier système d'exploitation libre conforme avec le standard ARINC653 (section 2.2.2) et l'approche MILS (section 2.1.1). Il peut être automatiquement configuré à partir de modèles AADL ou manuellement par du code provenant de l'utilisateur.

Cette plate-forme implémente l'API de la première partie des services du standard ARINC653 (*Required Services*) : partitionnement, services de tâches, de communication intra- et inter-partitions, implantation du *Health Monitoring*. Elle est par ailleurs conforme aux exigences des architectures MILS (partitionnement, isolation des niveaux de sécurité, etc.) et introduit plusieurs services spécifiques à la sécurité : algorithmes de chiffrement et confinement des pilotes de périphériques dans des partitions.

Au cours de nos travaux, plusieurs entreprises ont montré leur intérêt pour ce projet et nombreuses sont celles l'ayant téléchargé pour l'évaluer. A ce jour, il est utilisé par Thales dans le cadre du projet PARSEC en partenariat avec TELECOM Paristech. Il est par ailleurs en cours d'étude par l'Agence Spatiale Européenne dans le cadre de travaux sur les architectures avioniques modulaires intégrées.

7.2 Perspectives

Nos travaux ont permis de montrer la pertinence de la réunion de la sûreté et de la sécurité mais surtout les bénéfices de l'utilisation d'une spécification commune pour diriger chaque étape de la conception de systèmes critiques. Dans certains domaines (comme la certification), nos résultats montrent que l'utilisation de modèles optimise l'empreinte mémoire et la couverture de code, facilitant ainsi les efforts relatifs à l'optimisation et à la certification du système. Cependant, notre approche montre quelques limites (par exemple, taux de couverture non optimal). Les sections suivantes donnent de axes potentiels de recherche quant à son amélioration et à la correction des problèmes.

7.2.1 Description plus fine de l'architecture

Un premier axe de recherche consiste à décrire plus finement l'architecture du système et ses exigences. Ces ajouts à la spécification peuvent être mis en œuvre par une amélioration des nos patrons de modélisation et l'utilisation des langages d'annexes au langage AADL, tels l'annexe comportementale [51] ou l'annexe de modélisation des erreurs [95]. Cette méthode spécifierait ainsi chaque opération du système, explicitant plus finement les services utilisés pour la génération et l'exécution.

De tels ajouts présenteraient des bénéfices à chaque niveau de notre approche. En premier lieu, la spécification enrichie apporterait de nouvelles informations à valider et de nouvelles règles pourraient ainsi détecter davantage d'erreurs avant l'implantation du système. De plus, les nouvelles caractéristiques du système peuvent être exploitée par le générateur de code pour optimiser davantage les systèmes produits en terme d'empreinte mémoire ou de couverture de code.

7.2.2 Vers une meilleure certification

Au cours de nos travaux, nous avons proposé plusieurs solutions pour la certification des systèmes générés. Parmi celles-ci, nous avons développé une solution de supervision des évènements obtenus à l'exécution en vue de vérifier la conformité de l'exécution du système par rapport à sa spécification (section 5.5.2). Cette technique demande toutefois que l'utilisateur compare les traces obtenues avec le comportement déduit des spécifications. Un axe de recherche consisterait à automatiser cette comparaison, en créant des outils déduisant le comportement du système en fonction de ses spécifications et en les confrontant avec les évènements reportés à l'exécution.

L'aspect couverture de code (section 2.3.3) peut être également amélioré par une optimisation de la configuration de l'architecture. En effet, les patrons de modélisation proposés ne décrivent pas les fonctionnalités utilisées au sein de chaque service de l'architecture. Le générateur inclut alors automatiquement un bloc de code pour chaque service, indépendemment des réels besoins de l'application. Pour combler ce manque et supprimer les parties non requises, il est nécessaire de décrire plus finement l'architecture comme le suggère la première section de ces perspectives. Une meilleure description des exigences et du comportement de l'architecture permettrait ainsi de configurer plus finement le système, de supprimer davantage de fonctionnalités inutilisées et par conséquent, de maximiser le taux de couverture de code.

Enfin, la certification d'un système requiert que celui-ci soit testé exhaustivement (tous les cas d'exécution doivent être considérés) afin de vérifier sa robustesse. Un axe de recherche consisterait à générer automatiquement une suite de tests à partir des spécifications. Une spécification plus fine de l'architecture (comme le suggère la première partie de cette section) offrirait l'opportunité de tester chaque aspect de l'architecture et de valider la conformité de l'exécution avec les spécifications.

7.2.3 Amélioration de notre plate-forme, POK

Enfin, le dernier axe de recherche proposé concerne notre plate-forme d'exécution partitionnée, POK. La recherche autour des systèmes partitionnés est particulièrement active (nombreux projets tels que SPICES [2], PARSEC, Xtratum [29], etc.) et amène de nouvelles approches dans notre contexte.

D'un point de vue sécurité, il est pertinent d'étudier l'inclusion de méthodes de chiffrement asymétrique. Bien que plus lourd en ressources processeurs et mémoires, leur utilisation peut être pertinente lorsque peu d'entités échangent des données et que celles-ci sont hautement classifiées (meilleurs mécanismes de protection). Toutefois, cette intégration nécessite d'évaluer le coût (consommation processeur et mémoire) de leur utilisation et l'impact en terme de performance (temps de chiffrement/déchiffrement n'expirant pas avant une échéance donnée, etc.).

De plus, si l'isolation spatiale constitue une plus-value en terme de sécurité ou de sûreté, le concepteur d'application peut souhaiter introduire plus de flexibilité dans l'ordonnancement de son système [78]. Par exemple, certaines partitions nécessitent une telle adaptabilité, notamment celles exécutant des pilotes de périphériques. Pour ces dernières, il peut être pertinent de *« relâcher »* le processeur lorsque le périphérique contrôlé n'a aucune demande à traiter et de lui allouer davantage de temps d'exécution lorsque de nombreuses données lui sont transmises. Nous avons commencé à explorer ces aspects en incluant le protocole d'ordonnancement, HFPPS [30]. Ce dernier alloue du temps d'exécution supplémentaire aux partitions quand elles en ont besoin à condition qu'elles le restituent lors des périodes suivantes.

Enfin, notre plate-forme, POK, implante l'API du standard ARINC653. Toutefois, sa conformité n'a pas été validée, ces aspects relevant davantage de problématiques d'ingénierie industrielle. La politique de test mise en œuvre dans POK valide le bon fonctionnement de chaque service, sans pour autant tester exhaustivement tous les cas d'utilisation. Toutefois, une exploitation commerciale ou industrielle de POK nécessiterait d'inclure une telle campagne de test afin de valider la conformité de notre plate-forme avec le standard. Enfin, un dernier axe de développement industriel consisterait à implanter les services secondaires d'ARINC653 (*« Extended Services »* : système de fichiers, *sampling ports* étendus, etc.). L'ajout de ces services supplémentaires complèterait ainsi notre plate-forme d'exécution et offrirait davantage de possibilités aux concepteurs d'applications (stockage de fichiers, communications pouvant supporter davantage de contraintes, etc.).

7.2.4 Développement de la communauté de POK

Au cours de nos travaux, le projet POK et la suite d'outils associée (Ocarina, REAL) ont été utilisés par plusieurs partenaires, aussi bien du domaine industriels (Thales, Adacore) et académique (TELECOM ParisTech, LIP6, Ecole pour l'Informatique et les Techniques Avancées). De ces utilisations est née une synergie entre ces différents contributeurs, aboutissant à la création d'une communauté. Celle-ci s'est regroupée autour d'un site web et d'une liste de diffusion pour publier les travaux issus du projet et

discuter des développements futurs.

Une perspective de ces travaux constitue à continuer à faire vivre cette communauté pour améliorer les solutions développées autour du projet. Celui-ci ayant pris naissance à TELECOM ParisTech, c'est tout naturellement cette institution qui reprendra ce projet et sera garante de la vie future de la communauté ainsi créée. Par ailleurs, plusieurs doctorants ont déjà commencé à travailler sur des problématiques connexes à POK et un projet de recherche utilisant le projet (PARSEC, en partenariat avec Thales, le CEA, Ellidiss, INRIA, Systerel, OpenWide, Alstom et Télécom ParisTech) est en cours de réalisation. La vie du projet et de la communauté devraient donc être assurés à long terme.

Bibliographie

[1] Common Criteria for Information Technology Security Evaluation - `http://www.commoncriteriaportal.org/`.

[2] SPICES-ITEA project - `http://www.spices-itea.org/`.

[3] Static Source Code Analysis Tools for C - `http://www.spinroot.com/static/`.

[4] Airlines Electronic Engineering. Avionics Application Software Standard Interface. Technical report, Aeronautical Radio, INC, 1997.

[5] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS Architecture for High-Assurance Embedded Systems. *CROSSTALK - International Journal of Embedded Systems*, Aout 2005.

[6] J. Alves-Foss, C. Taylor, and P. Oman. A Multi-Layered Approach to Security in High Assurance Systems. In *HICSS '04 : Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90302.2, Washington, DC, USA, 2004. IEEE Computer Society.

[7] B. Ames. Real-time software goes modular. *Military & Aerospace Electronics*, 14(9), 2003.

[8] A. J. Arlow, C. J. Duffy, and J. A. McDermid. Safety Specification of the Active Traffic Management Control System for English Motorways. In *24th International System Safety Conference*, pages 54 – 63, 2006.

[9] ARTIST - Network of Excellence on Embedded Systems Design. Development of UML for Real-time Embedded Systems - `http://www.artist-embedded.org/artist/Development-of-UML-for-Real-time.html`. Technical report, ARTIST embedded, 2008.

[10] J. Barnard. The Value of a Mature Software Process. In *United Space Alliance, presentation to UK Mission on Space Software*, May 1999.

[11] W. Barnes. ARINC 653 and why is it important for a safety-critical RTOS, April 2004.

[12] D. E. Bell and L. J. LaPadula. Secure Computer System : Unified Exposition and MULTICS Interpretation. Technical report, The MITRE Corporation, 1976.

[13] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05 : Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[14] C. Beounes, M. Aguera, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. Moreira de Souza, D. Powell, and P. Spiesser. SURF-2 : A program for dependability evaluation of complex hardware and software systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 668–673, Jun 1993.

[15] Bernard Cole. Telelogic brings SysML to its TAU G2 system modeling suite - `http://www.embedded.com/news/embeddedindustry/172900319`.

[16] S. Bernardi and J. Merseguer. A UML profile for dependability analysis of real-time embedded systems. In *WOSP '07 : Proceedings of the 6th international workshop on Software and performance*, pages 115–124, New York, NY, USA, 2007. ACM.

[17] S. Bernardi, J. Merseguer, and D. C. Petriu. Adding Dependability Analysis Capabilities to the MARTE Profile. In *MoDELS '08 : Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 736–750, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical report, MITRE Corp., 04 1977.

[19] E. Biham and A. Shamir. Differential cryptanalysis of des-like cryptosystems. In *CRYPTO '90 : Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, pages 2–21, London, UK, 1991. Springer-Verlag.

[20] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The MILS Component Integration Approach to Secure Information Sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2008.

[21] E. Borde, G. Haïk, and L. Pautet. Mode-Based Reconfiguration of Critical Software Component Architectures. In *Design Automation and Test in Europe (DATE09)*, 2009.

[22] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, T. Quinot, J. Delange, J. Hugues, and L. Pautet. Couverture : an Innovative Open Framework for Coverage Analysis of Safety Critical Applications. *Ada User Journal*, pages 248–256, Dec. 2009.

[23] D. D. F. Brewer and D. M. J. Nash. The Chinese Wall Security Policy. volume 0, page 206, Los Alamitos, CA, USA, 1989. IEEE Computer Society.

[24] CEA/INRIA. Frama-C - `http://www.frama-c.cea.fr/index.html`.

[25] J. Chilenski. Aerospace Vehicle Systems Institute Systems and Software Integration Verification Overview. *AADL Safety and Security Modeling Meeting*, Nov. 2007.

[26] Common Criteria Testing Laboratory. INTEGRITY-178B Separation Kernel, certification report - `http://www.commoncriteriaportal.org/files/epfiles/st_vid10119-vr.pdf`. Technical report, 2008.

[27] Common Criteria Testing Laboratory. XTS-400/STOP 6.4 U4 - Validation Report, certification report : `http://www.commoncriteriaportal.org/files/epfiles/st_vid10293-vr.pdf`. Technical report, 2008.

[28] D. Coppersmith, D. B. Johnson, S. M. Matyas, T. J. Watson, D. B. Johnson, and S. M. Matyas. Triple DES Cipher Block Chaining with Output Feedback Masking, 1996.

[29] A. Crespo, I. Ripoll, and M. Masmano. Partitioned Embedded Architecture Based on Hypervisor : The XtratuM Approach. volume 0, pages 67–72, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[30] R. I. Davis and A. Burns. Hierarchical Fixed Priority Pre-Emptive Scheduling. In *RTSS '05 : Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[31] J. Delange. *PolyORB-HI-C user guide - `http://aadl.telecom-paristech.fr`*. Ecole Nationale Supérieure des Télécommunications, 46 rue Barrault, 2007.

[32] J. Delange. *POK Developper's Guide*. Ecole Nationale Supérieure des Télécommunications, 46 rue Barrault, 2008.

[33] J. Delange. *POK User's Guide*. Ecole Nationale Supérieure des Télécommunications, 46 rue Barrault, 2008.

[34] J. Delange. ARINC653 Annex for AADLv2 - to be published. Technical report, 2010.

[35] J. Delange, J. Hugues, L. Pautet, and D. de Niz. A MDE-based Process for the Design, Implementation and Validation of Safety Critical Systems. In *Proceedings of the 5th UML& AADL Workshop (UML&AADL 2010)*, University of Oxford, UK, Mar. 2010.

[36] J. Delange, J. Hugues, L. Pautet, and B. Zalila. Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain. In *4th European Congress ERTS*, Toulouse, January 2008.

[37] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon. Validate, simulate, and implement ARINC653 systems using the AADL. *Ada Letters*, 29(3) :31–44, 2009.

[38] 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 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 50–57, Washington, DC, USA, 2008. IEEE Computer Society.

[39] Department of Defense of the United States. Trusted Computer System Evaluation Criteria (Orange Book).

[40] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal. A Compositional Scheduling Framework for Digital Avionics Systems. In *Proceeding of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009)*, 2009.

[41] European Cooperation for Space Standardization. Standard ECSS-E-40-05 : Software Verification, Validation and Testing - http://www.ecss.nl. Technical report, 2007.
[42] European Space Agency. ASSERT Project - http://www.assert-project.net.
[43] European Space Agency. Ariane 501 - Presentation of Inquiry Board report - http://www.esa.int/esaCP/Pr_33_1996_p_EN.html. Technical Report 33-1996, 1996.
[44] P. Feiler and A. Rugina. Dependability Modeling with the Architecture Analysis & Design Language (AADL). Technical report, Software Engineering Institute, July 2007.
[45] P. H. Feiler. Evolution of an Avionics System. Technical report, Software Engineering Institute, October 2007.
[46] P. H. Feiler and J. Hansson. Flow Latency Analysis with the Architecture Analysis and Design Language (AADL). Technical report, Software Engineering Institute, 2007.
[47] P. H. Feiler, J. Hansson, D. de Niz, and L. Wrage. System Architecture Virtual Integration : An Industrial Case Study. Technical report, 2009.
[48] N. E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. Softw. Eng.*, 26(8) :797–814, 2000.
[49] M. V. Fleet. Protection Profile for Partitioning Kernels in Environments Requiring Augmented High Robustness. Technical report, National Security Agency, 2003.
[50] Flex-eWare. Flex-eWare Project - https://srcdev.lip6.fr/trac/research/flex-eware.
[51] R. Frana, J.-P. Bodeveix, M. Filali, and J.-F. Rolland. The aadl behaviour annex – experiments and roadmap. *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 377–382, July 2007.
[52] C. Georgiou and B. Cipher. Differential Cryptanalysis of the Data Encryption Standard (DES), 2000.
[53] A. German and G. Mooney. Air Vehicle Static Code Analysis - Lessons Learnt. In *Aspects of Safety Management*. Springer, 2001.
[54] O. Gilles. REAL : Requirement Enforcement Analysis. Technical report, TELECOM ParisTech, 2008.
[55] D. Greve, M. Wilding, and M. Vanfleet. A Separation Kernel Formal Security Policy. Technical report, Rockwell Collins and U.S Department of Defense.
[56] N. Halbwachs. A synchronous language at work : the story of lustre. pages 3–11, July 2005.
[57] C. Hannum and J. Kohl. The NetBSD project : A highly portable UNIX-like system. *the USENIX Association newsletter*, 20(6) :39–41, Dec. 1995.
[58] S. Harris. *CISSP Certification All-in-One Exam Guide, Fourth Edition.* McGraw-Hill, Inc., New York, NY, USA, 2008.
[59] L. Hatton. Estimating source lines of code from object code : Windows and Embedded Control Systems. Technical report, CISM, University of Kingston, August 2005.
[60] K. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage, 2001.
[61] J. Hugues, L. Pautet, B. Zalila, P. Dissaux, and M. Perrotin. Using AADL to build critical real-time systems : Experiments in the IST-ASSERT project. In *4th European Congress ERTS*, Toulouse, Paris, jan 2008.
[62] W. Jackson. Under attack - Common Criteria has loads of critics, but is it getting a bum rap ? http ://gcn.com/articles/2007/08/10/under-attack.aspx, Aug 2007.
[63] John J. Chilenski. An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion - http://www.tc.faa.gov/its/worldpac/techrpt/ar01-18.pdf. Technical report, DOT/FAA/AR, April 2001.
[64] R. M. Karcich, R. Skibbe, Aditya, and P. Garg. On Software Reliability and Code Coverage. In IEEE, editor, *Aerospace Applications Conference, 1996. Proceedings*, pages 297 – 308, February 1996.

[65] R. N. Kashi and M. Amarnathan. Perspectives on the use of model based development approach for safety critical avionics software development. In *International Conference on Aerospace Science and Technology*, 2008.

[66] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : exact characterization and average case behavior. In *Proceedings of the 10th Real Time Systems Symposium (RTSS 1989)*, pages 166–171, 1989.

[67] Lockheed-Martin and Boeing and Rockwell Collins and Green Hills Software and LynuxWorks and Objective Interface and University of Idaho. U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, `http://www.commoncriteriaportal.org/files/ppfiles/pp_skpp_hr_v1.03.pdf`, 2003.

[68] LynuxWorks. LynxSecure Embedded Hypervisor and Separation Kernel - `http://www.lynuxworks.com/virtualization/hypervisor.php`.

[69] J. Mason, K. Luecke, and J. Luke. Device drivers in time and space partitioned operating systems. In *25th Digital Avionics Systems Conference, 2006 IEEE/AIAA*, pages 1 –9, 15-19 2006.

[70] J. McDermid. Software Hazard and Safety Analysis - `ftp://ftp.cs.york.ac.uk/pub/hise/mcdermid.pdf`, January 2004.

[71] J. McDermid and T. Kelly. Software in Safety Critical Systems : Achievement and Prediction - `http://www-users.cs.york.ac.uk/tpk/inuce2004.pdf`. *Nuclear Future*, 03 :140–145, 2006.

[72] J. A. Mcdermid, M. Nicholson, D. J. Pumfrey, and P. Fenelon. Experience with the application of HAZOP to computer-based systems. In *Proceedings of the Tenth Annual Conference on Computer Assurance. COMPASS95. Systems Integrity, Software Safety and Process Security*, pages 37 – 48, 1995.

[73] Med KECHIDI (LEREPS-GRES). Aeronautical manufacturer versus architect-integrator : a new industrial model for Airbus (In French). Cahiers du GRES 2008-11, Groupement de Recherches Economiques et Sociales, 2008.

[74] A. K. Mok, X. Feng, and D. Chen. Resource Partition for Real-Time Systems. In *Seventh IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, May 2001.

[75] NASA. Mars Climate Orbiter - Mishap Investigation Board - `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`. Technical report, 1999.

[76] National Bureau of Standards of the United States of America. *Data Encryption Standard*. Number 46-1 in Federal Information Processing Standards publication. 1988. Category : ADP operations ; subcategory : computer security. Supersedes FIPS PUB 46, 1977 January 15. Shipping list no. : 88-367-P. Reaffirmed 1988 January 22.

[77] National Institute of Standards and Technology (NIST). The Economic Impacts of Inadequate Infrastructure for Software Testing - `http://www.nist.gov/director/prog-ofc/report02-3.pdf`. Technical report, 2002.

[78] T. Nolte, M. Behnam, M. Asberg, R. J. Bril, and I. Shin. Hierarchical Scheduling of Complex Embedded Real-Time Systems. In *Ecole d'Ete Temps-Reel (ETR09)*, 2009.

[79] OARCorp. RTEMS - `http://www.rtems.com`.

[80] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification - `http://www.omg.org/docs/formal/05-01-02.pdf`, January 2002.

[81] Object Management Group. OCL 2.0 Specification - `http://www.omg.org/spec/OCL/2.0/`, June 2005.

[82] Object Management Group. CORBA Component Model Specification - `http://http://www.omg.org/docs/formal/06-04-01.pdf/`. Technical report, 2006.

[83] Object Management Group. Systems Modeling Language (SysML) - `http://www.omgsysml.org`, 2007.

[84] Object Management Group. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) - `http://www.omgmarte.org`, June 2008.

[85] Open Source Initiative. BSD licence - `http://www.opensource.org/licenses/bsd-license.php`.

[86] A. Orebaugh, G. Ramirez, J. Burke, and L. Pesce. *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.
[87] R. F. Paige, L. M. Rose, X. Ge, D. S. Kolovos, and P. J. Brooke. FPTC : Automated Safety Analysis for Domain-Specific Languages. In *Models in Software Engineering*, volume Volume 5421/2009, pages 229–242. Springer Berlin / Heidelberg, 2009.
[88] S. Paynter, J. Armstrong, and J. Haveman. ADL : An Activity Description Language for Real-Time Networks. *Formal Aspects of Computing*, pages 120–144, 2000.
[89] *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). 1990.
[90] G. Quaranta and P. Mantegazza. Using Matlab-Simulink RTW To Build Real Time Control Applications In User Space With RTAI-LXRT, 2001.
[91] D. J. Reifer. Industry Software Cost, Quality and Productivity Benchmarks. `http://www.compaid.com/caiinternet/ezine/Reifer-Benchmarks.pdf`.
[92] D. W. Reinhardt. Considerations in the Preference for and Application of RTCA/DO-178B in the Australian Military Avionics Context. In T. Cant, editor, *Thirteenth Australian Conference on Safety-Related Programmable Systems (SCS 2008)*, volume 100 of *CRPIT*, pages 49–68, Canberra, Australia, 2008. ACS.
[93] M. Rhodes-Ousley, R. Bragg, and K. Strassberg. *Network Security : The Complete Reference*. McGraw-Hill Osborne Media, 2003.
[94] RTCA Inc. *Software considerations in airborne systems and equipment certification (DO178B)*.
[95] A. E. Rugina, K. Kanoun, and M. Kaaniche. The ADAPT Tool : From AADL Architectural Models to Stochastic Petri Nets through Model Transformation. Technical Report arXiv :0809.4108, Sep 2008. Comments : 6 pages.
[96] J. Rushby. *The Bell and La Padula Security Model*. Computer Science Laboratory, SRI International, Menlo Park, CA, 1986. Draft Technical Note.
[97] J. Rushby. Separation and Integration in MILS (The MILS Constitution). Technical report, SRI International, 2008.
[98] SAE Aerospace. *Architecture Analysis and Design Language (AADL)*, 2004.
[99] M. A. Sánchez-Puebla and J. Carretero. A new approach for distributed computing in avionics systems. In *ISICT '03 : Proceedings of the 1st international symposium on Information and communication technologies*, pages 579–584. Trinity College Dublin, 2003.
[100] C. Savarese and B. Hart. The Caesar Cipher - `http://starbase.trincoll.edu/~crypto/historical/caesar.html`.
[101] B. Schneier. Products that use Blowfish - `http://www.schneier.com/blowfish-products.html`.
[102] B. Schneier. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish. In *In Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, 1994.
[103] B. Schneier. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish) . In *Fast Software Encryption, Cambridge Security Workshop*, pages 191–204, London, UK, 1994. Springer-Verlag.
[104] F. Singhoff, J. Legrand, L. N. Tchamnda, and L. Marcé. Cheddar : a Flexible Real Time Scheduling Framework. *ACM Ada Letters journal, 24(4) :1-8, ACM Press. Also published in the proceedings of the ACM SIGADA International Conference, Atlanta, 15-18 November, 2004*, Nov. 2004.
[105] Software Engineering Institute. Open Source AADL Tool Environment. Technical report, Software Engineering Institute - Carnegie Mellon University, 2006.
[106] Z. Stephenson, M. Nicholson, and J. McDermid. Flexibility and Manageability of IMS Projects. In *24th International System Safety Conference*, 2006.
[107] Z. R. Stephenson, J. A. McDermid, and A. G. Ward. Health Modeling for Agility in Safety-Critical Systems Development. In *1st IET Conference on System Safety*, page 260, 2006.
[108] SysML Forum. SysML tools - `http://www.sysmlforum.com/tools.htm`.

[109] TELECOM ParisTech. TELECOM ParisTech AADL Portal - http ://aadl.telecom-paristech.fr.

[110] The OpenSSL Project. OpenSSL - `http://www.openssl.org`. Programa de computador, December 1998.

[111] G. Uchenick and M. Vanfleet. Multiple Independent Levels of Safety and Security : High Assurance Architecture for MSLS/MLS. In IEEE, editor, *Military Communications Conference, 2005. MILCOM*, 2005.

[112] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. Rapid Development Methodology for Customized Middleware. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 111–117, Montreal, Canada, June 2005. IEEE.

[113] M. Wallace. Modular Architectural Representation and Analysis of Fault Propagation and Transformation. *Electr. Notes Theor. Comput. Sci.*, 141(3) :53–71, 2005.

[114] M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. In L.-H. Eriksson and P. A. Lindsay, editors, *FME 2002 : Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe*, volume Lectur, pages 431–450. Springer Verlag, 2002.

[115] T. W. Williams, M. R. Mercer, J. P. Mucha, and R. Kapur. Code Coverage, What Does It Mean in Terms of Quality ? In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 420 – 424, 2001.

[116] 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.

[117] B. Zalila, I. Hamid, J. Hugues, and L. Pautet. Generating distributed high integrity applications from their architectural description. In *Ada-Europe'07 : Proceedings of the 12th international conference on Reliable software technologies*, pages 155–167, Berlin, Heidelberg, 2007. Springer-Verlag.

[118] B. Zalila, J. Hugues, and L. Pautet. *Ocarina user guide - `http://aadl.telecom-paristech.fr`*. TELECOM ParisTech.

[119] J. Zhou and J. Alves-Foss. Architecture-based refinements for secure computer systems design. In *PST '06 : Proceedings of the 2006 International Conference on Privacy, Security and Trust*, pages 1–11, New York, NY, USA, 2006. ACM.

[120] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4) :425–432, 1980.

[121] M. Zitser, D. E. S. Group, and T. Leek. Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. In *In Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.

Annexe A

Publications & glossaire

Sommaire

A.1 Publications 224
A.1.1 Articles de conférence 224
A.1.2 Chapitre de livre 224
A.1.3 Articles de Journaux 225
A.1.4 Documents de standardisation 225
A.1.5 Rapports techniques 225
A.2 Glossaire 226

A.1 Publications

A.1.1 Articles de conférence

- Julien Delange, Laurent Pautet et Fabrice Kordon. *Design, Verification and Implementation of MILS systems*. In Proceedings of the 21th International Symposium on Rapid System Prototyping, Fairfax, États-Unis, Juin 2010.
- Julien Delange, Laurent Pautet et Fabrice Kordon. *Modeling and Validation of ARINC653 Architectures*. In Proceedings of Embedded Real Time Software and Systems (ERTSS'10), Toulouse, France, Juin 2010.
- Gopal Raghav, Swaminathan Gopalswamy, Julien Delange, Laurent Pautet et Jérôme Hugues. *Model Based Code Generation for Distributed Embedded Systems*. In Proceedings of Embedded Real Time Software and Systems (ERTSS'10), Toulouse, France, Juin 2010.
- Julien Delange, Laurent Pautet, Jérôme Hugues et Dionisio de Niz. *A MDE-based Process for the Design, Implementation and Validation of Safety-Critical Systems*. In Proceedings of the 5th UML AADL Workshop - UML AADL 2010, Oxford, Angleterre, Mars 2010.
- Gopal Raghav, Swaminathan Gopalswamy, Karthikeyan Radhakrishnan, Julien Delange et Jérôme Hugues. *Architecture Driven Generation of Distributed Embedded Software from Functional Models*. In Proceedings of Ground Vehicle Systems Engineering and Technology Symposium (GVSETS09). Detroit, États-Unis, Août 2009.
- Julien Delange, Olivier Gilles, Jérôme Hugues et Laurent Pautet. *Model-Based Engineering for the Development of Partitioned Architectures*. In Proceedings of the AeroTech Congress & Exhibition - Avionics - Integrated Model-based System, Application and Architectures, Seattle, États-Unis, Novembre 2009.
- Julien Delange, Laurent Pautet et Peter H. Feiler. *Validating safety and security requirements for partitioned architectures*. In Proceedings of the 14th International Conference on Reliable Software Technologies - Ada Europe, pages 30-43, Brest, France, Juin 2009.
- Julien Delange, Laurent Pautet et Fabrice Kordon. *Code Generation Strategies for Partitioned Systems*. In Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS'08), IEEE Computer Society, pages 53-56, Barcelone, Espagne, Décembre 2008.
- Julien Delange, Jérôme Hugues Laurent Pautet et Bechir Zalila. *Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain*. In Proceedings of the 4th European Congress ERTS, Toulouse, Paris, Janvier 2008.

A.1.2 Chapitre de livre

- Julien Delange, Laurent Pautet et Fabrice Kordon. *Model-Based Approach for the Configuration of ARINC653 systems* dans *« Reconfigurable Embedded Control Systems : Applications for Flexibility and Agility »* . IGI Global, 2010. À paraître.

A.1.3 Articles de Journaux

- Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff et Fabrice Kordon. *Validate, simulate and implement ARINC653 systems using the AADL.* ACM SIGAda Ada Letters (from the proceedings of the ACM SigAda conference 2009), 29(3), pages 31-44, ACM Press, Novembre 2009. (Prix du meilleur papier étudiant).
- Matteo Bordin, Cyrille Comar, Tristan Gingold, Jérôme Guitton, Olivier Hainque, Thomas Quinot, Julien Delange, Jérôme Hugues et Laurent Pautet, *« Couverture : an Innovative Open Framework for Coverage Analysis of Safety Critical Applications »* , In Ada-User Journal, pages 248-256, Ada-Europe, Decembre, 2009.

A.1.4 Documents de standardisation

- Julien Delange. *ARINC653 annex for AADLv2. A paraître.* `http://standards.sae.org/wip/as5506/2/`.

A.1.5 Rapports techniques

- Julien Delange. *POK User's Guide.* `http://pok.gunnm.org/snapshots/pok-userguide-current.pdf`
- Julien Delange. *POK Kernel Manual.* `http://pok.gunnm.org/snapshots/pok-kernel-refman-current.pdf`
- Julien Delange. *POK Libpok Manual.* `http://pok.gunnm.org/snapshots/pok-libpok-refman-current.pdf`
- Julien Delange, Jérôme Hugues et Bechir Zalila *PolyORB-HI-C User's Guide.* `http://aadl.telecom-paristech.fr`

A.2 Glossaire

- **AADL** : *Architecture Analysis and Design Language* - langage de modélisation d'architecture.
- **AST** : *Abstract Syntax Tree* - structure de données pour le stockage des déclarations écrites dans un langage (tel Ada ou C).
- **Chiffrement** : procédé consistant à rendre incompréhensible une information.
- **Criticité** : La criticité représente le degré d'importance d'un composant. Dans le cas des architectures partitionnées, la criticité d'une partition spécifie l'importance des fonctions remplies par cette partition au regard des autres [99, 94].
- **Déchiffrement** : procédé consistant à retrouver la signification d'une information précédemment chiffrée (voir chiffrement).
- ***Earliest Deadline First*** : protocole d'ordonnancement consistant à élire les tâches en fonction de leur échéance.
- ***Gestionnaire d'Exceptions*** : routine du noyau chargée de traîter les erreurs reportées au processeur. Plusieurs lignes d'entrées/sorties du processeur signalent les erreurs qu'il reporte, le gestionnaire d'exception est exécuté dès que l'une d'entre elle est activée.
- ***Interruption Matérielle*** : une interruption matérielle (IRQ en anglais pour *Interrupt ReQuest*) est une ligne d'entrée/sortie entre un périphérique et le processeur. Cette ligne peut être gérée par le logiciel exécuté sur la machine afin de gérer le périphérique. Dans ce cas, à chaque émission d'interruption, le programme chargé de gérer le périphérique est exécuté, préemptant l'exécution du programme en cours.
- **Noyau** : programme dédié à la gestion et au contrôle des ressources matérielles de la machine.
- **Ordonnancement** : choix d'une entité parmi un groupe en fonction de ses caractéristiques (temporelles ou non).
- **Pagination** : mécanisme consistant à découper la mémoire de la machine en pages et change la correspondance entre mémoire virtuelle et mémoire physique. La pagination introduit souvent de l'indéterminisme : les pages étant de taille réduite (quelques Ko), toute allocation mémoire entraîne une recherche de pages libres dont le temps est difficilement borné.
- **Politique de sécurité** : ensemble de règles définissant les opérations (lecture/écriture/modification) que peut réaliser une entité d'exécution sur des données en fonction de son niveau de sécurité (secret, top-secret, etc).
- **Polling** : mode d'implantation d'un pilote de périphérique dans lequel le programme contrôlant le matériel effectue des requêtes pour accéder à ses données. Ce mode diffère de l'implantation par *interruption*, où le matériel notifie le logiciel de la présence de nouvelles données (voir définition *interruption matérielle*).
- ***Rate Monotonic Scheduling* - RMS** : protocole d'ordonnancement consistant à élire les tâches à exécuter suivant leur période d'exécution.
- ***Round-Robin*** : protocole d'ordonnancement élisant les tâches dans un

ordre fixe, prédéterminé et ne prenant pas en compte les paramètres temporels comme leur période ou échéance.

- **Segmentation** : mécanisme de division de la mémoire physique en segments caractérisés par leur adresse de début et leur taille. Contrairement à la pagination, la segmentation n'introduit qu'un simple décalage mémoire et n'entraine pas d'indéterminisme dans la gestion de la mémoire.
- **Sécurité** : Ensemble de fonctions et mécanismes destinés à protéger l'information des entités n'ayant pas l'autorisation de les manipuler.
- **Sûreté** : Ensemble de mécanismes mis en place pour assurer la continuité de fonctionnement du système.

Annexe B

Code source

Sommaire

B.1 Code de POK 230
B.1.1 Initialisation des partitions 230
B.1.2 Fonction de recouvrement du noyau 231
B.1.3 Fonction de recouvrement des erreurs des partitions 232
B.1.4 Fonction de recouvrement des erreurs des tâches dans une partition 233
B.2 Théorèmes de validation implémentés avec REAL . 233
B.2.1 Conformité de la spécification 233
B.2.2 Isolation temporelle 234
B.2.3 Isolation spatiale 234
B.2.4 Spécification du Health Monitoring 235
B.2.5 Spécification de la criticité des partitions 236
B.2.6 Couverture des fautes 236
B.2.7 Impact de la politique de recouvrement 237
B.2.8 Implémentation des mécanismes de sécurité 238
B.2.9 Implémentation de la cohérence des mécanismes de sécurité 238
B.2.10 Validation politique de sécurité Bell-Lapadula 238
B.2.11 Validation politique de sécurité Biba 239
B.2.12 Validation de la politique de sécurité MILS 239
B.3 Librairie de componsants 240
B.4 Représentation textuelle des cas d'études 247
B.4.1 Cas d'étude *« integrated »* 247
B.4.2 Cas d'étude *« MILS »* 265
B.5 Code source de test des cas d'études 275
B.5.1 Programme de chiffrement 275

B.1 Code de POK

Cette section présente plusieurs parties du code source de notre plate-forme d'exécution, POK. Tout le code du projet n'est pas inclus, nous avons souhaité montrer les parties les plus essentielles ou critiques de ses services.

B.1.1 Initialisation des partitions

```
pok_ret_t pok_partition_init ()
{
   uint8_t     i;
   uint32_t    threads_index = 0;

   const uint32_t partition_size[POK_CONFIG_NB_PARTITIONS]
                              = POK_CONFIG_PARTITIONS_SIZE;
#ifdef POK_CONFIG_PARTITIONS_LOADADDR
   const uint32_t program_loadaddr[POK_CONFIG_NB_PARTITIONS]
                              = POK_CONFIG_PROGRAM_LOADADDR;
#endif
#ifdef POK_NEEDS_LOCKOBJECTS
   uint8_t lockobj_index = 0;
#endif

   for (i = 0 ; i < POK_CONFIG_NB_PARTITIONS ; i++)
   {
        uint32_t size = partition_size[i];
#ifndef POK_CONFIG_PARTITIONS_LOADADDR
        uint32_t base_addr =
            (uint32_t)pok_bsp_mem_alloc(partition_size[i]);
#else
      uint32_t base_addr = program_loadaddr[i];
#endif
      uint32_t program_entry;
      uint32_t base_vaddr = pok_space_base_vaddr(base_addr);

      pok_partitions[i].base_addr   = base_addr;
      pok_partitions[i].size        = size;
      pok_partitions[i].sched       = POK_SCHED_RR;

      pok_partition_setup_scheduler (i);

      pok_create_space (i, base_addr, size);

      pok_partitions[i].base_vaddr = base_vaddr;
      /* Set the memory space and so on */

      pok_partitions[i].thread_index_low
                              = threads_index;
      pok_partitions[i].nthreads
               = ((uint32_t[]) POK_CONFIG_PARTITIONS_NTHREADS) [i];

#ifdef POK_NEEDS_ERROR_HANDLING
      if (pok_partitions[i].nthreads <= 1)
      {
         pok_partition_error
               (i, POK_ERROR_KIND_PARTITION_CONFIGURATION);
      }
#endif

      #ifdef POK_CONFIG_PARTITIONS_SCHEDULER
      pok_partitions[i].sched
            = ((pok_sched_t[]) POK_CONFIG_PARTITIONS_SCHEDULER) [i];
#endif

      pok_partitions[i].thread_index_high
```

```
        =   pok_partitions[i].thread_index_low + (
            (uint32_t[]) POK_CONFIG_PARTITIONS_NTHREADS) [i];
      pok_partitions[i].activation          = 0;
      pok_partitions[i].period              = 0;
      pok_partitions[i].thread_index        = 0;
      pok_partitions[i].thread_main         = 0;
      pok_partitions[i].current_thread      = IDLE_THREAD;
      pok_partitions[i].prev_current_thread     = IDLE_THREAD;

#ifdef POK_NEEDS_SCHED_HFPPS
      pok_partitions[i].payback = 0;
#endif /* POK_NEEDS_SCHED_HFPPS */

      threads_index
                = threads_index + pok_partitions[i].nthreads;
      /* Initialize the threading stuff */

      pok_partitions[i].mode
            = POK_PARTITION_MODE_INIT_WARM;

#ifdef POK_NEEDS_LOCKOBJECTS
      pok_partitions[i].lockobj_index_low
                = lockobj_index;

      pok_partitions[i].lockobj_index_high
                = lockobj_index +
                  ((uint8_t[]) POK_CONFIG_PARTITIONS_NLOCKOBJECTS[i]);

      pok_partitions[i].nlockobjs
                = ((uint8_t[]) POK_CONFIG_PARTITIONS_NLOCKOBJECTS[i]);

      lockobj_index
                = lockobj_index + pok_partitions[i].nlockobjs;
      /* Initialize mutexes stuff */
#endif

#ifdef POK_NEEDS_ERROR_HANDLING
      pok_partitions[i].thread_error        = 0;
      pok_partitions[i].error_status.failed_thread = 0;
      pok_partitions[i].error_status.failed_addr   = 0;
      pok_partitions[i].error_status.error_kind
                        = POK_ERROR_KIND_INVALID;
      pok_partitions[i].error_status.msg_size       = 0;
#endif

      pok_loader_load_partition (i, base_addr - base_vaddr, &program_entry);
      /*
       * Load the partition in its address space
       */
      pok_partitions[i].thread_main_entry = program_entry;

#ifdef POK_NEEDS_INSTRUMENTATION
      pok_instrumentation_partition_archi (i);
#endif

      pok_partition_setup_main_thread (i);
   }

   return POK_ERRNO_OK;
}
```

Listing B.1 – Initialisation des partitions

B.1.2 Fonction de recouvrement du noyau

```
void pok_kernel_error
    (uint32_t error)
{

  switch (error)
  {
    case POK_ERROR_KIND_KERNEL_INIT:
    {
      pok_kernel_stop ();

      break;
    }
    case POK_ERROR_KIND_KERNEL_SCHEDULING:
    {
      pok_kernel_restart ();

      break;
    }
  }
}
```

Listing B.2 – Fonction de recouvrement des erreurs dans la couche noyau

B.1.3 Fonction de recouvrement des erreurs des partitions

```
void pok_partition_error
    (uint8_t partition,
    uint32_t error)
{

  switch (partition)
  {
    case 0:
    {
      switch (error)
      {
        case POK_ERROR_KIND_PARTITION_INIT:
        {
          pok_partition_set_mode (0, POK_PARTITION_MODE_STOPPED);

          break;
        }
        case POK_ERROR_KIND_PARTITION_SCHEDULING:
        {
          pok_partition_set_mode (0, POK_PARTITION_MODE_STOPPED);

          break;
        }
      }

      break;
    }
    case 1:
    {
      switch (error)
      {
        case POK_ERROR_KIND_PARTITION_INIT:
        {
          pok_partition_set_mode (1, POK_PARTITION_MODE_STOPPED);

          break;
        }
        case POK_ERROR_KIND_PARTITION_SCHEDULING:
        {
          pok_partition_set_mode (1, POK_PARTITION_MODE_STOPPED);
```

```
            break;
          }
        }

        break;
      }
    }
}
```

Listing B.3 – Fonction de recouvrement des erreurs des partitions

B.1.4 Fonction de recouvrement des erreurs des tâches dans une partition

```
void pok_error_handler_worker ()
{

  pok_error_status_t error_status;
  while (1)
  {
    pok_thread_stop_self ();
    pok_error_get (&(error_status));
    switch (error_status.failed_thread)
    {
      case 1:
      {
        switch (error_status.error_kind)
        {
          case POK_ERROR_KIND_DEADLINE_MISSED:
          {
            pok_thread_restart (1);

            break;
          }
          case POK_ERROR_KIND_APPLICATION_ERROR:
          {
            pok_thread_restart (1);

            break;
          }
        }

        break;
      }

    }
  }
}
```

Listing B.4 – Fonction de recouvrement des erreurs pour les tâches

B.2 Théorèmes de validation implémentés avec REAL

Afin de faciliter la lecture, l'ordre des théorèmes implémentés suit l'organisation de la section dédiée à la validation (section 4.3).

B.2.1 Conformité de la spécification

```
theorem Process_Compliance
    foreach p in process_set do
      M  := {x in memory_set | is_bound_to (x, p)};
      VP := {x in virtual_processor_set | is_bound_to (p, x)};
    check ((cardinal (VP) = 1) and (cardinal (M) >= 1);
end Process_Compliance;
```

Listing B.5 – Théorème validant la conformité des composants process

```
theorem Contains_Memories

   foreach s in system_set do
      mainmem := {y in memory_Set | is_subcomponent_of (y, s)};
      partitionsmem := {x in memory_Set | is_subcomponent_of (x, mainmem)};

    check ( (Cardinal (mainmem) > 0) and
            (property_exists (partitionsmem, "Byte_Count")) );
end Contains_Memories;
```

Listing B.6 – Théorème validant la conformité des composants memory

B.2.2 Isolation temporelle

```
theorem partitions_execution
   foreach p in Processor_Set do
      VP := { x in Virtual_Processor_Set | Is_Subcomponent_Of (x, p)};

      check (Is_In (VP, Get_Property_Value (p, "ARINC653::Slots_Allocation")) or
             Is_In (VP, Get_Property_Value (p, "POK::Slots")));
end partitions_execution;
```

Listing B.7 – Théorème validant l'exécution de chaque partition

```
theorem scheduling_major_frame
    foreach cpu in processor_set do
    check (float (property (cpu, "ARINC653::Module_Major_Frame")) =
           sum (property (cpu, "ARINC653::Partition_Slots")));
end scheduling_major_frame;
```

Listing B.8 – Théorème validant la conformité du major frame

B.2.3 Isolation spatiale

```
theorem Partitions_Memory
   foreach p in Process_Set do
      mem := {y in Memory_Set | Is_Bound_To (y, p)};

   check (Cardinal (mem) = 0);
end Partitions_Memory;
```

Listing B.9 – Théorème validant l'assignation des partitions à un segment mémoire

```
theorem check_memory_requirements_partitions
```

```
   foreach prs in process_set do
      Thrs := {x in thread_set | is_subcomponent_of (x, prs)};
      mems := { x in memory_set | is_bound_to (Prs, x)};
      check
         (
            (sum (property (Thrs, "Source_Stack_Size")) +
             sum (property (Thrs, "Source_Data_Size")) +
             sum (property (Thrs, "Source_Code_Size")))
            < (sum (property (mems, "byte_count")))
         );
end check_memory_requirements_partitions;
```

Listing B.10 – Théorème validant la taille des segments mémoires associés aux partitions

B.2.4 Spécification du Health Monitoring

```
theorem check_processor_hm_validity
   foreach mycpu in processor_set do

      var allowed_module_errors  := list ("Module_Config",
                     "Module_Init", "Module_Scheduling");

      var allowed_module_actions := list ("Nothing",
                     "Module_Stop", "Module_Restart");

      var module_actions :=  (property
                        (mycpu, "ARINC653::HM_Actions"));

      var module_errors  :=  (property
                        (mycpu, "ARINC653::HM_Errors"));

      check (
         (property_exists (mycpu, "ARINC653::HM_Actions")) and
         (property_exists (mycpu, "ARINC653::HM_Errors" )) and
         (is_in (module_actions, allowed_module_actions )) and
         (is_in (module_errors, allowed_module_errors   )));
end check_processor_hm_validity;
```

Listing B.11 – Validation de la spécification du Health Monitoring du Noyau

```
theorem check_virtual_processor_hm_validity
   foreach vp in virtual_processor_set do

      var allowed_module_errors := list ("Partition_Config",
         "Partition_Init", "Partition_Scheduling", "Partition_Handler");

      var allowed_module_actions := list ("Nothing",
         "Partition_Stop", "Partition_Restart");

      var module_actions :=  (property (vp, "ARINC653::HM_Actions"));

      var module_errors  :=  (property (vp, "ARINC653::HM_Errors"));

      check ((property_exists (vp, "ARINC653::HM_Actions")) and
             (property_exists (vp, "ARINC653::HM_Errors")) and
             ( is_in (module_actions, allowed_module_actions)) and
             ( is_in (module_errors, allowed_module_errors)));
```

```
end check_virtual_processor_hm_validity;
```

Listing B.12 – Validation de la spécification du Health Monitoring des Partitions

```
theorem check_thread_hm_validity
   foreach thr in thread_set do

      var allowed_module_errors  := list ("Deadline_Miss",
         "Application_Error", "Numeric_Error", "Illegal_Request",
         "Stack_Overflow", "Memory_Violation", "Hardware_Fault",
         "Power_Fail");

      var allowed_module_actions := list ("Ignore", "Confirm",
      "Process_Stop", "Process_Stop_And_Start_Another",
      "Process_Restart", "Partition_Stop", "Partition_Restart");

      var module_actions :=  (property (thr, "ARINC653::HM_Actions"));

      var module_errors  :=  (property (thr, "ARINC653::HM_Errors"));

      check ((property_exists (thr, "ARINC653::HM_Actions"  )) and
             (property_exists (thr, "ARINC653::HM_Errors"   )) and
             (is_in (module_actions, allowed_module_actions )) and
             (is_in (module_errors, allowed_module_errorsi  )));
end check_thread_hm_validity;
```

Listing B.13 – Validation de la spécification du Health Monitoring des Tâches

B.2.5 Spécification de la criticité des partitions

```
theorem check_partitions_criticality
   foreach vp in virtual_processor_set do
      check ((property_exists (vp, "ARINC653::Criticality")) or
             (property_exists (vp, "POK::Criticality")));
end check_partitions_criticality;
```

Listing B.14 – Validation de la criticité des partitions

B.2.6 Couverture des fautes

```
theorem check_error_coverage
   foreach thr in thread_set do
      Prs :=  {x in Process_Set           | is_subcomponent_of (thr, x)};

      VP :=   {x in Virtual_Processor_Set | is_bound_to (Prs, x)};

      CPU :=  {x in Processor_Set         | is_subcomponent_of (VP, x)};

      var errors := list ("Module_Config", "Module_Init",
         "Module_Scheduling"   ,
         "Partition_Scheduling", "Partition_Config",
         "Partition_Handler"   , "Partition_Init",
         "Deadline_Miss"       , "Application_Error",
         "Numeric_Error"       , "Illegal_Request",
         "Stack_Overflow"      , "Memory_Violation",
         "Hardware_Fault"      , "Power_Fail");

      var actual_errors := (property (CPU, "ARINC653::HM_Errors") +
```

```
                              property (VP,  "ARINC653::HM_Errors") +
                              property (thr, "ARINC653::HM_Errors"));
      check (is_in (errors, actual_errors) and
             is_in (actual_errors, errors));
end check_error_coverage;
```

Listing B.15 – Validation de la couverture de fautes

B.2.7 Impact de la politique de recouvrement

```
theorem check_omission_transient
 foreach Src_Prs in process_set do
    Src_Thr     := {x in Thread_Set           |
                    is_subcomponent_of (x, Src_Prs)};

    Src_Runtime := {x in Virtual_Processor_Set|
                    is_bound_to (Src_Prs, x)};

    Dst_Prs     := {x in Process_Set          |
                    is_connected_to (Src_Prs, x)};

    Dst_Runtime := {x in Virtual_Processor_Set|
                    is_bound_to (Dst_Prs, x)};

      var restart_actions      := list ("Partition_Restart",
                                   "Process_Restart", "Confirm");
      var src_recovery_actions :=
            (property (Src_Runtime, "ARINC653::HM_Actions") +
             property (Src_Thr, "ARINC653::HM_Actions"));

      check (((cardinal (Src_Prs) > 0) and (cardinal (Dst_Prs) >= 0) and
               (is_in (restart_actions, src_recovery_actions)) and
               (max (property (Dst_Runtime, "POK::Criticality")) <
                max ((property (Src_Runtime,"POK::Criticality"))))
              )
              or
              (not (is_in (restart_actions, src_recovery_actions)))
             );
end check_omission_transient;
```

Listing B.16 – Théorème analysant l'impact transitoire d'une faute sur les autres partitions

```
theorem check_omission_permanent
   foreach Src_Prs in process_set do
    Src_Thr    :={x in Thread_Set           |
                  is_subcomponent_of (x, Src_Prs)};
    Src_Runtime:={x in Virtual_Processor_Set|
                  is_bound_to (Src_Prs, x)};
    Dst_Prs    :={x in Process_Set          |
                  is_connected_to (Src_Prs, x)};
    Dst_Runtime:={x in Virtual_Processor_Set|
                  is_bound_to (Dst_Prs, x)};

      var restart_actions :=
            list ("Partition_Stop",
                  "Process_Stop_And_Start_Another",
                  "Process_Stop");

      var src_recovery_actions :=
         (property (Src_Runtime, "ARINC653::HM_Actions") +
          property (Src_Thr, "ARINC653::HM_Actions"));
```

```
      check (((cardinal (Src_Prs) > 0) and
              (cardinal (Dst_Prs) >= 0) and
              (is_in (restart_actions, src_recovery_actions)) and
              (max (property (Dst_Runtime, "POK::Criticality")) <
               max ((property (Src_Runtime,"POK::Criticality")))))
            or
            (not (is_in (restart_actions, src_recovery_actions))));
end check_omission_permanent;
```

Listing B.17 – Théorème analysant l'impact permanent de la politique de reprise

B.2.8 Implémentation des mécanismes de sécurité

```
theorem Check_Virtual_Bus_Implementation
    foreach VB in virtual_bus_set do
      check (property_exists (VB, "Implemented_As"));
end Check_Virtual_Bus_Implementation;
```

Listing B.18 – Validation de l'implantation

B.2.9 Implémentation de la cohérence des mécanismes de sécurité

```
theorem Check_Virtual_Bus_Cipher_Algorithms

    foreach VB1 in virtual_bus_set do

      VB2 := {x in Virtual_Bus_Set |
                 (get_property_value (x, "POK::Security_Level") <>
                  get_property_value (VB1, "POK::Security_Level")) };

      check (
                ((not property_exists (VB1, "pok::blowfish_key")) or
                 (not property_exists (VB2, "pok::blowfish_key"))) or
                 (get_property_value(VB1, "pok::blowfish_key") <>
                  Head (get_property_value (VB2, "pok::blowfish_key")))
             );
end Check_Virtual_Bus_Cipher_Algorithms;
```

Listing B.19 – Validation de la cohérence des mécanismes de sécurité pour le protocole *Blowfish*

B.2.10 Validation politique de sécurité Bell-Lapadula

```
theorem bell_lapadula

    foreach p_src in process_set do

      VP1    := {x in virtual_processor_set |
                 is_bound_to (p_src, x)};

      B_Src  := {x in virtual_bus_set       |
                 is_provided_class (VP1, x)};

      P_Dest := {x in process_set           |
                 is_connected_to (p_src, x)};
```

```
      VP2    := {x in virtual_processor_set |
                 is_bound_to (P_Dest, x)};

      B_Dst  := {x in virtual_bus_set       |
                 is_provided_class (VP2, x)};

    check (cardinal (P_Dest) = 0
           or
           (max (property (B_Src, "POK::Security_Level")) <=
            min (property (B_Dst, "POK::Security_Level"))
           )
          );
end bell_lapadula;
```

Listing B.20 – Validation de la politique de sécurite Bell/Lapadula

B.2.11 Validation politique de sécurité Biba

```
theorem biba

    foreach p_src in process_set do

      VP1    := {x in virtual_Processor_Set |
                 is_bound_to (p_src, x)};

      B_Src  := {x in virtual_Bus_Set       |
                 is_provided_class (VP1, x)};

      P_Dest := {x in process_Set           |
                 is_connected_To (p_src, x)};

      VP2    := {x in virtual_Processor_Set |
                 is_bound_to (P_Dest, x)};

      B_Dst  := {x in virtual_Bus_Set       |
                 is_provided_class (VP2, x)};

    check (cardinal (P_Dest) = 0 or
           (min (property (B_Src, "POK::Security_Level"))
            >=
            max (property (B_Dst, "POK::Security_Level"))
           )
          );
end biba;
```

Listing B.21 – Validation de la politique de sécurite Biba

B.2.12 Validation de la politique de sécurité MILS

```
theorem MILS_1

    foreach p_src in process_set do

      VP1 := {x in Virtual_Processor_Set | is_bound_to (p_src, x)};

      B_Src := {x in Virtual_Bus_Set | is_provided_class (VP1, x)};

      P_Dest := {x in Process_Set | Is_Connected_To (p_src, x)};

      VP2 := {x in Virtual_Processor_Set | is_bound_to (P_Dest, x)};
```

```
      B_Dst := {x in Virtual_Bus_Set | is_provided_class (VP2, x)};

    check (Cardinal (P_Dest) = 0 or
            ((max (property (B_Src, "POK::Security_Level")) =
              max (property (B_Dst, "POK::Security_Level"))) and
             (min (property (B_Src, "POK::Security_Level")) =
              min (property (B_Dst, "POK::Security_Level")))));
end MILS_1;
```

Listing B.22 – Validation de la politique de sécurite MILS

B.3 Librairie de componsants

Notre chaîne de développement contient un fichier contenant plusieurs composants AADL. Ceux-ci peuvent ensuite être réutilisés par le concepteur de système pour éviter de redéfinir des concepts déjà discutés et éviter d'introduire des erreurs lors de la définition de ses modèles.

```
package poklib

public

with POK;
with Data_Model;

-------------------
-- Processor --
-------------------

processor pok_kernel
properties
   POK::Scheduler => static;
end pok_kernel;

processor implementation pok_kernel.x86_qemu
properties
   POK::Scheduler => static;
   POK::Architecture => x86;
   POK::BSP => x86_qemu;
end pok_kernel.x86_qemu;

processor implementation pok_kernel.ppc_prep
properties
   POK::Architecture => ppc;
   POK::BSP => prep;
end pok_kernel.ppc_prep;

processor implementation pok_kernel.sparc_leon3
properties
   POK::Architecture => sparc;
   POK::BSP => leon3;
end pok_kernel.sparc_leon3;

processor implementation pok_kernel.x86_qemu_two_partitions
                                 extends pok_kernel.x86_qemu
subcomponents
   partition1 : virtual processor
                  poklib::pok_partition.basic_for_example;
   partition2 : virtual processor
                  poklib::pok_partition.basic_for_example;
properties
   POK::Major_Frame => 1000ms;
   POK::Scheduler => static;
```

```
   POK::Slots => (500ms, 500ms);
   POK::Slots_Allocation =>
            (reference (partition1), reference (partition2));
end pok_kernel.x86_qemu_two_partitions;

processor implementation pok_kernel.x86_qemu_three_partitions
                                 extends pok_kernel.x86_qemu
subcomponents
   partition1 : virtual processor
                poklib::pok_partition.basic_for_example;
   partition2 : virtual processor
                poklib::pok_partition.basic_for_example;
   partition3 : virtual processor
                poklib::pok_partition.basic_for_example;
properties
   POK::Major_Frame => 1500ms;
   POK::Scheduler => static;
   POK::Slots => (500ms, 500ms, 500ms);
   POK::Slots_Allocation =>
      (reference (partition1), reference (partition2),
       reference (partition3));
end pok_kernel.x86_qemu_three_partitions;

processor implementation pok_kernel.x86_qemu_four_partitions
                                 extends pok_kernel.x86_qemu
subcomponents
   partition1 : virtual processor
                     poklib::pok_partition.basic_for_example;
   partition2 : virtual processor
                     poklib::pok_partition.basic_for_example;
   partition3 : virtual processor
                     poklib::pok_partition.basic_for_example;
   partition4 : virtual processor
                     poklib::pok_partition.basic_for_example;
properties
   POK::Major_Frame => 2000ms;
   POK::Scheduler => static;
   POK::Slots => (500ms, 500ms, 500ms, 500ms);
   POK::Slots_Allocation =>
      (reference (partition1), reference (partition2),
       reference (partition3), reference (partition4));
end pok_kernel.x86_qemu_four_partitions;

processor implementation
      pok_kernel.x86_qemu_four_partitions_with_libmath
                           extends pok_kernel.x86_qemu
subcomponents
   partition1 : virtual processor
      poklib::pok_partition.basic_for_example_with_libmath;
   partition2 : virtual processor
      poklib::pok_partition.basic_for_example_with_libmath;
   partition3 : virtual processor
      poklib::pok_partition.basic_for_example_with_libmath;
   partition4 : virtual processor
      poklib::pok_partition.basic_for_example_with_libmath;
properties
   POK::Major_Frame => 2000ms;
   POK::Scheduler => static;
   POK::Slots => (500ms, 500ms, 500ms, 500ms);
   POK::Slots_Allocation =>
      (reference (partition1), reference (partition2),
       reference (partition3), reference (partition4));
end pok_kernel.x86_qemu_four_partitions_with_libmath;

-----------------------
-- Virtual Buses --
-----------------------
```

```
-- Unclassified virtual bus

virtual bus unencrypted
end unencrypted;

virtual bus implementation unencrypted.i
end unencrypted.i;

-- blowfish virtual bus

subprogram blowfish_send
features
   datain    : in parameter poklib::pointed_void;
   countin   : in parameter poklib::integer;
   dataout   : out parameter poklib::pointed_void;
   countout  : out parameter poklib::integer;
properties
   Source_Name => "pok_protocols_blowfish_marshall";
end blowfish_send;

subprogram implementation blowfish_send.i
end blowfish_send.i;

subprogram blowfish_receive
features
   datain    : in parameter poklib::pointed_void;
   countin   : in parameter poklib::integer;
   dataout   : out parameter poklib::pointed_void;
   countout  : out parameter poklib::integer;
properties
   Source_Name => "pok_protocols_blowfish_unmarshall";
end blowfish_receive;

subprogram implementation blowfish_receive.i
end blowfish_receive.i;

data blowfish_data
end blowfish_data;

data implementation blowfish_data.i
properties
   Type_Source_Name => "pok_protocols_blowfish_data_t";
end blowfish_data.i;

abstract vbus_blowfish_wrapper
end vbus_blowfish_wrapper;

abstract implementation vbus_blowfish_wrapper.i
subcomponents
   send              : subprogram blowfish_send.i;
   receive           : subprogram blowfish_receive.i;
   marshalling_type  : data blowfish_data.i;
end vbus_blowfish_wrapper.i;

virtual bus blowfish
end blowfish;

virtual bus implementation blowfish.i
properties
   Implemented_As =>
         classifier (poklib::vbus_blowfish_wrapper.i);
end blowfish.i;

-- DES virtual bus

subprogram des_send
features
   datain       : in parameter     poklib::pointed_void;
```

```
   count_in     : in parameter    poklib::integer;
   dataout      : out parameter   poklib::pointed_void;
   count_out    : out parameter   poklib::integer;
end des_send;

subprogram implementation des_send.i
properties
   Source_Name => "pok_protocols_des_marshall";
end des_send.i;

subprogram des_receive
features
   datain   : in parameter poklib::pointed_void;
   countin  : in parameter poklib::integer;
   dataout  : out parameter poklib::pointed_void;
   countout : out parameter poklib::integer;
end des_receive;

subprogram implementation des_receive.i
properties
   Source_Name => "pok_protocols_des_unmarshall";
end des_receive.i;

data des_data
end des_data;

data implementation des_data.i
properties
   Type_Source_Name => "pok_protocols_des_data_t";
end des_data.i;

abstract vbus_des_wrapper
end vbus_des_wrapper;

abstract implementation vbus_des_wrapper.i
subcomponents
   send              : subprogram des_send.i;
   receive           : subprogram des_receive.i;
   marshalling_type  : data des_data.i;
end vbus_des_wrapper.i;

virtual bus des
end des;

virtual bus implementation des.i
properties
   Implemented_As =>
         classifier (poklib::vbus_des_wrapper.i);
end des.i;

-- ceasar virtual bus

subprogram ceasar_send
features
   datain   : in parameter poklib::pointed_void;
   countin  : in parameter poklib::integer;
   dataout  : out parameter poklib::pointed_void;
   countout : out parameter poklib::integer;
properties
   Source_Name => "pok_protocols_ceasar_marshall";
end ceasar_send;

subprogram implementation ceasar_send.i
end ceasar_send.i;

subprogram ceasar_receive
features
   datain   : in parameter poklib::pointed_void;
```

```
   countin  : in parameter poklib::integer;
   dataout  : out parameter poklib::pointed_void;
   countout : out parameter poklib::integer;
properties
   Source_Name => "pok_protocols_ceasar_unmarshall";
end ceasar_receive;

subprogram implementation ceasar_receive.i
end ceasar_receive.i;

data ceasar_data
end ceasar_data;

data implementation ceasar_data.i
properties
   Type_Source_Name => "pok_protocols_ceasar_data_t";
end ceasar_data.i;


abstract vbus_ceasar_wrapper
end vbus_ceasar_wrapper;

abstract implementation vbus_ceasar_wrapper.i
subcomponents
   send              : subprogram ceasar_send.i;
   receive           : subprogram ceasar_receive.i;
end vbus_ceasar_wrapper.i;

virtual bus ceasar
end ceasar;

virtual bus implementation ceasar.i
properties
   Implemented_As =>
         classifier (poklib::vbus_ceasar_wrapper.i);
end ceasar.i;

-- gzip virtual bus

subprogram gzip_send
features
   datain   : in parameter poklib::pointed_void;
   countin  : in parameter poklib::integer;
   dataout  : out parameter poklib::pointed_void;
   countout : out parameter poklib::integer;
properties
   Source_Name => "pok_protocols_gzip_marshall";
end gzip_send;

subprogram implementation gzip_send.i
end gzip_send.i;

subprogram gzip_receive
features
   datain   : in parameter poklib::pointed_void;
   countin  : in parameter poklib::integer;
   dataout  : out parameter poklib::pointed_void;
   countout : out parameter poklib::integer;
properties
   Source_Name => "pok_protocols_gzip_unmarshall";
end gzip_receive;

subprogram implementation gzip_receive.i
end gzip_receive.i;

data gzip_data
end gzip_data;
```

```
data implementation gzip_data.i
properties
   Type_Source_Name => "pok_protocols_gzip_data_t";
end gzip_data.i;

abstract vbus_gzip_wrapper
end vbus_gzip_wrapper;

abstract implementation vbus_gzip_wrapper.i
subcomponents
   send              : subprogram gzip_send.i;
   receive           : subprogram gzip_receive.i;
   marshalling_type  : data gzip_data.i;
end vbus_gzip_wrapper.i;

virtual bus gzip
end gzip;

virtual bus implementation gzip.i
properties
   Implemented_As =>
         classifier (poklib::vbus_gzip_wrapper.i);
end gzip.i;

--------------------------
-- Virtual Processor --
--------------------------

virtual processor pok_partition
end pok_partition;

virtual processor implementation pok_partition.basic
properties
   POK::Scheduler => RR;
end pok_partition.basic;

virtual processor implementation pok_partition.driver
properties
   POK::Scheduler => RR;
   POK::Additional_Features => (pci, io);
end pok_partition.driver;

virtual processor implementation pok_partition.application
properties
   POK::Scheduler => RR;
   POK::Additional_Features => (stdio, stdlib);
end pok_partition.application;

virtual processor implementation pok_partition.basic_for_example
                                    extends pok_partition.basic
properties
   POK::Additional_Features => (stdio, stdlib);
end pok_partition.basic_for_example;

virtual processor implementation
            pok_partition.basic_for_example_with_libmath
                              extends pok_partition.basic
properties
   POK::Additional_Features => (stdio, stdlib, libmath, console);
end pok_partition.basic_for_example_with_libmath;


----------------
-- Memories --
----------------

memory pok_memory
end pok_memory;
```

```
memory implementation pok_memory.x86_segment
end pok_memory.x86_segment;

memory implementation pok_memory.x86_main
end pok_memory.x86_main;


-----------------
-- Threads --
-----------------

thread thr_periodic
properties
   Dispatch_Protocol          => Periodic;
   Period                     => 100ms;
   Deadline                   => 100ms;
   Compute_Execution_Time     => 5ms .. 10ms;
end thr_periodic;

thread thr_sporadic
properties
   Dispatch_Protocol          => Sporadic;
   Period                     => 100ms;
   Deadline                   => 100ms;
   Compute_Execution_Time     => 5ms .. 10ms;
end thr_sporadic;

-----------------------
-- Subprograms --
-----------------------

subprogram spg_c
properties
   Source_Language  => C;
   Source_Text      => ("../../../user-functions.o");
end spg_c;

-----------------
-- Integer --
-----------------

data void
properties
   Type_Source_Name => "void";
end void;

data implementation void.i
end void.i;

data pointed_void
properties
   Type_Source_Name => "void*";
end pointed_void;

data implementation pointed_void.i
end pointed_void.i;

data char
properties
   Type_Source_Name => "char";
end char;

data implementation char.i
end char.i;

data pointed_char
properties
```

```
   Type_Source_Name => "char*";
end pointed_char;

data implementation pointed_char.i
end pointed_char.i;

data integer
properties
   Data_Model::Data_Representation => integer;
end integer;


data float
properties
   Data_Model::Data_Representation => float;
end float;

end poklib;
```

B.4 Représentation textuelle des cas d'études

B.4.1 Cas d'étude *« integrated »*

B.4.1.1 Modélisation de la mémoire

```
package memories

public

with ARINC653;

memory segment
end segment;

memory implementation segment.data_segment
properties
   ARINC653::Memory_Kind => memory_data;
   ARINC653::Access_Type => read_write;
   Byte_Count => 3000;
end segment.data_segment;

memory implementation segment.code_segment
properties
   ARINC653::Memory_Kind => memory_code;
   ARINC653::Access_Type => read;
   Byte_Count => 5000;
end segment.code_segment;


memory ram
end ram;

memory implementation ram.i
subcomponents
   segment1_data : memory segment.data_segment;
   segment1_code : memory segment.code_segment;
   segment2_data : memory segment.data_segment;
   segment2_code : memory segment.code_segment;
   segment3_data : memory segment.data_segment;
   segment3_code : memory segment.code_segment;
   segment4_data : memory segment.data_segment;
   segment4_code : memory segment.code_segment;
   segment5_data : memory segment.data_segment;
   segment5_code : memory segment.code_segment;
   segment6_data : memory segment.data_segment;
```

```
   segment6_code : memory segment.code_segment;
end ram.i;


end memories;
```

B.4.1.2 Modélisation des partitions

```
package memories

public

with ARINC653;

memory segment
end segment;

memory implementation segment.data_segment
properties
   ARINC653::Memory_Kind => memory_data;
   ARINC653::Access_Type => read_write;
   Byte_Count => 3000;
end segment.data_segment;

memory implementation segment.code_segment
properties
   ARINC653::Memory_Kind => memory_code;
   ARINC653::Access_Type => read;
   Byte_Count => 5000;
end segment.code_segment;


memory ram
end ram;

memory implementation ram.i
subcomponents
   segment1_data : memory segment.data_segment;
   segment1_code : memory segment.code_segment;
   segment2_data : memory segment.data_segment;
   segment2_code : memory segment.code_segment;
   segment3_data : memory segment.data_segment;
   segment3_code : memory segment.code_segment;
   segment4_data : memory segment.data_segment;
   segment4_code : memory segment.code_segment;
   segment5_data : memory segment.data_segment;
   segment5_code : memory segment.code_segment;
   segment6_data : memory segment.data_segment;
   segment6_code : memory segment.code_segment;
end ram.i;


end memories;
```

B.4.1.3 Modélisation des partitions

```
package partitions
public
with threads;
with types;
```

```
-------------------------------------------------------------
-- GENERIC COMPONENT
-------------------------------------------------------------
virtual processor runtime
end runtime;


-------------------------------------------------------------
-- WARNING ANNUNCIATION MANAGER
-------------------------------------------------------------

-- Model both virtual processor and process for the
-- Warning Annunciation Manager partition

virtual processor implementation
        runtime.warningannunciationmanager
end runtime.warningannunciationmanager;

process process_warningannunciationmanager
end process_warningannunciationmanager;

process implementation
        process_warningannunciationmanager.impl
subcomponents
   thr : thread
        threads::thread_warningannunciationmanager.impl;
end process_warningannunciationmanager.impl;


-------------------------------------------------------------
-- DISPLAY MANAGER
-------------------------------------------------------------

-- Model both virtual processor and process for the
-- Display Manager partition

virtual processor implementation runtime.displaymanager
end runtime.displaymanager;

process process_displaymanager
features
   DMToDisplay_Request : in data port
                         types::PageRequestCmd;
   DMToDisplay_Page    : out data port
                         types::PageContent;
   DMToPCM_Request     : out data port
                         types::PageRequestCmd;
   DMToPCM_Page        : in data port
                         types::PageContent;
end process_displaymanager;

process implementation process_displaymanager.impl
subcomponents
   thr : thread threads::thread_displaymanager.impl;
connections
   port DMToDisplay_Request    -> thr.DMToDisplay_Request;
   port thr.DMToDisplay_Page   -> DMToDisplay_Page;
   port DMToPCM_Page           -> thr.DMToPCM_Page;
   port thr.DMToPCM_Request    -> DMToPCM_Request;
end process_displaymanager.impl;


-------------------------------------------------------------
-- PAGE CONTENT MANAGER
-------------------------------------------------------------

-- Model both virtual processor and process for the
```

```
-- Page Content Manager partition

virtual processor implementation runtime.pagecontentmanager
end runtime.pagecontentmanager;

process process_pagecontentmanager
features
   PCMToFM_Request : out data port types::PageRequestCmd;
   PCMToFM_Page    : in data port types::PageContent;
   PCMToDM_Request : in data port types::PageRequestCmd;
   PCMToDM_Page    : out data port types::PageContent;
end process_pagecontentmanager;

process implementation process_pagecontentmanager.impl
subcomponents
   thr : thread threads::thread_pagecontentmanager.impl;
connections
   port PCMToDM_Request      -> thr.PCMToDM_Request;
   port thr.PCMToDM_Page     -> PCMToDM_Page;
   port thr.PCMToFM_Request -> PCMToFM_Request;
   port PCMToFM_Page         -> thr.PCMToFM_Page;
end process_pagecontentmanager.impl;


-------------------------------------------------------
-- FLIGHT MANAGER
-------------------------------------------------------

-- Model both virtual processor and process for the
-- Flight Manager partition

virtual processor implementation runtime.flightmanager
end runtime.flightmanager;

process process_flightmanager
features
   FMToPCM_Request    : in data port
                        types::PageRequestCmd;
   FMToPCM_Page       : out data port
                        types::PageContent;
   FMToFD_Request     : out data port
                        types::PageRequestCmd;
   FMToFD_Page        : in data port
                        types::PageContent;
   FMIn_fuelFlow      : in data port
                        types::FuelFlowData;
   FMIn_navSignal     : in data port
                        types::NavSignalData;
   FMOut_guidanceOut  : out data port
                        types::GuidanceData;
   FMOut_fpDataOut    : out data port
                        types::FPData;
   FMOut_navDataOut   : out data port
                        types::NavData;
end process_flightmanager;

process implementation process_flightmanager.pio
subcomponents
   NSP      : thread threads::NavigationSensorProcessing.impl;
   INav     : thread threads::IntegratedNavigation.impl;
   GP       : thread threads::GuidanceProcessing.impl;
   FPP      : thread threads::FlightPlanProcessing.impl;
   APC      : thread threads::AircraftPerformanceCalculation.impl;
   pageFeed : thread threads::HandlePageRequest.impl;
   PerIO    : thread threads::PeriodicIO.impl;
connections
   navsensorconn:    port NSP.navSensorDataToIN
      -> INav.navSensorDataFromNSP;
```

```
    navdataconn1:       port INav.navDataToGPAPC
        -> GP.navDataFromIN;
    navdataconn2:       port INav.navDataToGPAPC
        -> APC.navDataFromIN;
    guidanceconn1:      port GP.guidanceToFPP
        -> FPP.guidanceFromGP;
    performancedataconn: port APC.perfDataToFPP
        -> FPP.perfDataFromAPC;
    fpdataconn1:        port FPP.fpDataToGPAPC
        -> GP.fpDataFromFPP;
    fpdataconn2:        port FPP.fpDataToGPAPC
        -> APC.fpDataFromFPP;
    inflow:             port FMIn_fuelFlow
        -> PerIO.fromOutside_fuelFlow;
    inflow2:            port FMIn_navSignal
        -> PerIO.fromOutside_navSignal;
    outflow:            port PerIO.toOutside_guidanceOut
        -> FMOut_guidanceOut;
    outflow2:           port PerIO.toOutside_fpDataOut
        -> FMOut_fpDataOut;
    outflow3:           port PerIO.toOutside_navDataOut
        -> FMOut_navDataOut;
    fuelflowconn:       port PerIO.fuelFlow
        -> APC.fuelFlow;
    navsignalconn:      port PerIO.navSignal
        -> NSP.navSignalDataFromSensor;
    navdataconn3:       port INav.navDataToGPAPC
        -> PerIO.navDataOut;
    guidanceconn2:      port GP.guidanceToFPP
        -> PerIO.guidanceOut;
    fpdataconn3:        port FPP.fpDataToGPAPC
        -> PerIO.fpDataOut;
    inPageRequestPIO:   port FMToPCM_Request
        -> PerIO.New_Page_Request_From_PCM;
    inPageRequest:      port PerIO.New_Page_Request_To_PageFeed
        -> pageFeed.New_Page_Request_From_PCM;
    outPageRequest:     port pageFeed.New_Page_Request_To_FD
        -> PerIO.New_Page_Request_From_PageFeed;
    outPageRequestPIO:port PerIO.New_Page_Request_To_FD
        -> FMToFD_Request;
    inPageContentPIO:   port FMToFD_Page
        -> PerIO.New_Page_Content_from_FD;
    inPageContent:      port PerIO.New_Page_Content_To_PageFeed
        -> pageFeed.New_Page_Content_from_FD;
    outPageContentPIO:port PerIO.New_Page_Content_To_PCM
        -> FMToPCM_Page;
    outPageContent:     port pageFeed.New_Page_Content_To_PCM
        -> PerIO.New_Page_Content_From_PageFeed;
end process_flightmanager.pio;


-------------------------------------------------------------
-- FLIGHT DIRECTOR
-------------------------------------------------------------

-- Model both virtual processor and process for the
-- Flight Director partition

virtual processor implementation runtime.flightdirector
end runtime.flightdirector;

process process_flightdirector
features
    FDToFM_Request : in data port types::PageRequestCmd;
    FDToFM_Page    : out data port types::PageContent;
end process_flightdirector;

process implementation process_flightdirector.impl
```

```
subcomponents
   thr : thread threads::thread_flightdirector.impl;
connections
   port thr.FDToFM_Page -> FDToFM_Page;
   port FDToFM_Request  -> thr.FDToFM_Request;
end process_flightdirector.impl;


-------------------------------------------------------------
-- MFD
-------------------------------------------------------------

process mfd_driver
features
   DisplayToDM_Request   : out data port
                           types::PageRequestCmd;
   DisplayToDM_Page      : in data port
                           types::PageContent;
end mfd_driver;

process implementation mfd_driver.i
subcomponents
   thr : thread threads::thread_mfd_driver.i;
connections
   port thr.DisplayToDM_Request -> DisplayToDM_Request;
   port DisplayToDM_Page        -> thr.DisplayToDM_Page;
end mfd_driver.i;

abstract mfd_driver_wrapper
end mfd_driver_wrapper;

abstract implementation mfd_driver_wrapper.i
subcomponents
   p : process mfd_driver.i;
end mfd_driver_wrapper.i;



device mfd
features
   DisplayToDM_Request   : out data port
                           types::PageRequestCmd;
   DisplayToDM_Page      : in data port
                           types::PageContent;
end mfd;

device implementation mfd.i
properties
   Implemented_As => classifier
            (partitions::mfd_driver_wrapper.i);
end mfd.i;

end partitions;
```

B.4.1.4 Modélisation de la runtime

```
package platform
public

with ARINC653;
with POK;
with partitions;

processor partitioned
end partitioned;
```

```
processor implementation partitioned.x86
subcomponents
   part1 : virtual processor
           partitions::runtime.warningannunciationmanager;
   part2 : virtual processor
           partitions::runtime.displaymanager;
   part3 : virtual processor
           partitions::runtime.pagecontentmanager;
   part4 : virtual processor
           partitions::runtime.flightmanager;
   part5 : virtual processor
           partitions::runtime.flightdirector;
properties
   ARINC653::Module_Major_Frame  => 100ms;
   ARINC653::Partition_Slots     => (20ms, 20ms, 20ms, 20ms, 20ms);
   ARINC653::Slots_Allocation    => (reference (part1),
                                     reference (part2),
                                     reference (part3),
                                     reference (part4),
                                     reference (part5));

   POK::Architecture => x86;
   POK::BSP => x86_qemu;
end partitioned.x86;

end platform;
```

B.4.1.5 Modélisation des sous-programmes applicatifs

```
package subprograms
public

with Types;

subprogram spg_warningannunciationmanager
properties
   source_name        => "user_warning_annunciation_manager";
   source_language    => C;
   Source_Text        => ("../../../wam_code.o");
   Source_Code_Size   => 4 KByte;
   Source_Data_Size   => 1 KByte;
end spg_warningannunciationmanager;

subprogram spg_displaymanager
features
   request : out parameter types::PageRequestCmd;
   page    : in parameter types::PageContent;
properties
   source_name        => "user_display_manager";
   source_language    => C;
   Source_Text        => ("../../../dm_code.o");
   Source_Code_Size   => 40 KByte;
   Source_Data_Size   => 20 KByte;
end spg_displaymanager;

subprogram spg_pagecontentmanager
properties
   source_name        => "user_page_content_manager";
   source_language    => C;
   Source_Text        => ("../../../pcm_code.o");
   Source_Code_Size   => 8  KByte;
   Source_Data_Size   => 20 KByte;
end spg_pagecontentmanager;
```

```
subprogram spg_navigationsensorprocessing
properties
   source_name          => "user_navigation_sensor_processing";
   source_language      => C;
   Source_Text          => ("../../../fm_code.o");
   Source_Code_Size     => 2 KByte;
   Source_Data_Size     => 1 KByte;
end spg_navigationsensorprocessing;

subprogram spg_integratednavigation
properties
   source_name       => "user_integrated_navigation";
   source_language   => C;
   Source_Text       => ("../../../fm_code.o");
   Source_Code_Size  => 5 KByte;
   Source_Data_Size  => 2 KByte;
end spg_integratednavigation;

subprogram spg_guidanceprocessing
properties
   source_name       => "user_guidance_processing";
   source_language   => C;
   Source_Text       => ("../../../fm_code.o");
   Source_Code_Size  => 4 KByte;
   Source_Data_Size  => 1 KByte;
end spg_guidanceprocessing;

subprogram spg_flightplanprocessing
properties
   source_name       => "user_flight_plan_processing";
   source_language   => C;
   Source_Text       => ("../../../fm_code.o");
   Source_Code_Size  => 2 KByte;
   Source_Data_Size  => 8 KByte;
end spg_flightplanprocessing;

subprogram spg_aircraftperformancecalculation
properties
   source_name       =>
      "user_aircraft_performance_calculation";
   source_language   => C;
   Source_Text       => ("../../../fm_code.o");
   Source_Code_Size  => 4 KByte;
   Source_Data_Size  => 2 KByte;
end spg_aircraftperformancecalculation;

subprogram spg_periodicio
features
--   fromOutside: feature group types::FMInData;

   fromOutside_fuelFlow              : in  parameter
                                       types::FuelFlowData;
   fromOutside_navSignal             : in  parameter
                                       types::navSignalData;
   fuelFlow                          : out parameter
                                       types::FuelFlowData;
   navSignal                         : out parameter
                                       types::NavSignalData;
   guidanceOut                       : in  parameter
                                       types::GuidanceData;
   fpDataOut                         : in  parameter
                                       types::FPData;
   navDataOut                        : in  parameter
                                       types::NavData;
   toOutside_guidanceOut             : out parameter
                                       types::GuidanceData;
   toOutside_fpDataOut               : out parameter
                                       types::FPData;
```

```
   toOutside_navDataOut              : out parameter
                                       types::NavData;
   New_Page_Request_From_PCM         : in  parameter
                                       types::PageRequestCmd;
   New_Page_Request_To_PageFeed      : out parameter
                                       types::PageRequestCmd;
   New_Page_Request_From_PageFeed    : in  parameter
                                       types::PageRequestCmd;
   New_Page_Request_To_FD            : out parameter
                                       types::PageRequestCmd;
   New_Page_Content_from_FD          : in parameter
                                       types::PageContent;
   New_Page_Content_To_PageFeed      : out parameter
                                       types::PageContent;
   New_Page_Content_From_PageFeed    : in  parameter
                                       types::PageContent;
   New_Page_Content_To_PCM           : out parameter
                                       types::PageContent;
properties
   source_name        => "user_periodic_io";
   source_language    => C;
   Source_Text        => ("../../../fm_code.o");
   Source_Code_Size   => 12 KByte;
   Source_Data_Size   => 26 KByte;
end spg_periodicio;

subprogram spg_handlepagerequest
properties
   source_name          => "user_handle_page_request";
   source_language      => C;
   Source_Text          => ("../../../fm_code.o");
   Source_Code_Size     => 10 KByte;
   Source_Data_Size     => 5 KByte;
end spg_handlepagerequest;

subprogram spg_flightmanager
properties
   source_name          => "user_flight_manager";
   source_language      => C;
   Source_Text          => ("../../../fd_code.o");
   Source_Code_Size     => 15 KByte;
   Source_Data_Size     => 6 KByte;
end spg_flightmanager;

subprogram spg_flightdirector
properties
   source_name          => "user_flight_director";
   source_language      => C;
   Source_Text          => ("../../../fd_code.o");
   Source_Code_Size     => 8 KByte;
   Source_Data_Size     => 2 KByte;
end spg_flightdirector;

subprogram mfd_driver
features
   DisplayToDM_Request   : out parameter
                           types::PageRequestCmd;
   DisplayToDM_Page      : in  parameter
                           types::PageContent;
properties
   source_name        => "user_mfd_driver";
   source_language    => C;
   Source_Text        => ("../../../mfd_code.o");
   Source_Code_Size   => 8 KByte;
   Source_Data_Size   => 2 KByte;
end mfd_driver;
```

```
end subprograms;
```

B.4.1.6 Modélisation des partitions

```
package memories

public

with ARINC653;

memory segment
end segment;

memory implementation segment.data_segment
properties
   ARINC653::Memory_Kind => memory_data;
   ARINC653::Access_Type => read_write;
   Byte_Count => 3000;
end segment.data_segment;

memory implementation segment.code_segment
properties
   ARINC653::Memory_Kind => memory_code;
   ARINC653::Access_Type => read;
   Byte_Count => 5000;
end segment.code_segment;


memory ram
end ram;

memory implementation ram.i
subcomponents
   segment1_data : memory segment.data_segment;
   segment1_code : memory segment.code_segment;
   segment2_data : memory segment.data_segment;
   segment2_code : memory segment.code_segment;
   segment3_data : memory segment.data_segment;
   segment3_code : memory segment.code_segment;
   segment4_data : memory segment.data_segment;
   segment4_code : memory segment.code_segment;
   segment5_data : memory segment.data_segment;
   segment5_code : memory segment.code_segment;
   segment6_data : memory segment.data_segment;
   segment6_code : memory segment.code_segment;
end ram.i;


end memories;
```

B.4.1.7 Modélisation des tâches

```
package threads
public

with types;

-------------------------------------------------
-- WARNING ANNUNCIATION MANAGER
-------------------------------------------------
```

```
thread thread_warningannunciationmanager
properties
   Dispatch_Protocol        => Periodic;
   Period                   => 50 Ms;
   Compute_Execution_Time   => 5 Ms .. 15 Ms;
   Source_Data_Size         => 10 Kbyte;
   Source_Code_Size         => 10 Kbyte;
end thread_warningannunciationmanager;

thread implementation thread_warningannunciationmanager.impl
calls
   call1:
      {pspg : subprogram
       subprograms::spg_warningannunciationmanager;};
end thread_warningannunciationmanager.impl;


-------------------------------------------
-- DISPLAY MANAGER
-------------------------------------------

thread thread_displaymanager
features
   DMToDisplay_Request : in  data port
                         types::PageRequestCmd;
   DMToDisplay_Page    : out data port
                         types::PageContent;
   DMToPCM_Page        : in  data port
                         types::PageContent;
   DMToPCM_Request     : out data port
                         types::PageRequestCmd;
properties
   Dispatch_Protocol        => Periodic;
   Period                   => 50 Ms;
   Compute_Execution_Time   => 5 Ms .. 15 Ms;
   Source_Data_Size         => 10 Kbyte;
   Source_Code_Size         => 10 Kbyte;
end thread_displaymanager;

thread implementation thread_displaymanager.impl
calls
   call1:
      {pspg : subprogram
         subprograms::spg_displaymanager;};
connections
   parameter pspg.request -> DMToPCM_Request;
   parameter DmToPCM_Page -> pspg.page;
end thread_displaymanager.impl;

-------------------------------------------
-- PAGE CONTENT MANAGER
-------------------------------------------

thread thread_pagecontentmanager
features
   PCMToFM_Request  : out data port
                      types::PageRequestCmd;
   PCMToFM_Page     : in  data port
                      types::PageContent;
   PCMToDM_Request  : in  data port
                      types::PageRequestCmd;
   PCMToDM_Page     : out data port
                      types::PageContent;
properties
   Dispatch_Protocol          => Periodic;
   Period                     => 50 Ms;
   Compute_Execution_Time     => 5 Ms .. 15 Ms;
```

```
   Source_Data_Size            => 10 Kbyte;
   Source_Code_Size            => 10 Kbyte;
end thread_pagecontentmanager;

thread implementation thread_pagecontentmanager.impl
calls
   call1:
      {pspg : subprogram
       subprograms::spg_pagecontentmanager;};
end thread_pagecontentmanager.impl;


-----------------------------------------
-- FLIGHT MANAGER
-----------------------------------------

thread NavigationSensorProcessing
features
   navSignalDataFromSensor : in data port
                     types::NavSignalData;
   navSensorDataToIN       : out data port
                     types::NavSensorData;
properties
   Dispatch_Protocol           => Periodic;
   Period                      => 50 Ms;
   Compute_Execution_Time      => 5 Ms .. 15 Ms;
   Source_Data_Size            => 10 Kbyte;
   Source_Code_Size            => 10 Kbyte;
end NavigationSensorProcessing;

thread implementation NavigationSensorProcessing.impl
calls
   call1:
      {pspg : subprogram
       subprograms::spg_navigationsensorprocessing;};
end NavigationSensorProcessing.impl;


thread IntegratedNavigation
features
   navSensorDataFromNSP : in  data port
                  types::NavSensorData;
   navDataToGPAPC       : out data port
                  types::NavData;
properties
   Dispatch_Protocol           => Periodic;
   Period                      => 100 Ms;
   Compute_Execution_Time      => 10 Ms .. 40 Ms;
   Source_Data_Size            => 10 Kbyte;
   Source_Code_Size            => 10 Kbyte;
end IntegratedNavigation;

thread implementation IntegratedNavigation.impl
calls
   call1:
      {pspg : subprogram
       subprograms::spg_integratednavigation;};
end IntegratedNavigation.impl;


thread GuidanceProcessing
features
   navDataFromIN   : in  data port
                     types::NavData;
   fpDataFromFPP   : in  data port
                     types::FPData;
   guidanceToFPP   : out data port
                     types::GuidanceData;
```

```
properties
   Dispatch_Protocol        => Periodic;
   Period                   => 50 Ms;
   Compute_Execution_Time   => 8 Ms .. 30 Ms;
   Source_Data_Size         => 10 Kbyte;
   Source_Code_Size         => 10 Kbyte;
end GuidanceProcessing;

thread implementation GuidanceProcessing.impl
calls
   call1:
      {pspg : subprogram
       subprograms::spg_guidanceprocessing;};
end GuidanceProcessing.impl;

thread FlightPlanProcessing
features
   guidanceFromGP     : in  data port
                        types::GuidanceData;
   perfDataFromAPC    : in  data port
                        types::PerformanceData;
   fpDataToGPAPC      : out data port
                        types::FPData;
properties
   Dispatch_Protocol        => Periodic;
   Period                   => 200 Ms;
   Compute_Execution_Time   => 10 Ms .. 30 Ms;
   Source_Data_Size         => 10 Kbyte;
   Source_Code_Size         => 10 Kbyte;
end FlightPlanProcessing;

thread implementation FlightPlanProcessing.impl
calls
   call1:
      {pspg : subprogram
       subprograms::spg_flightplanprocessing;};
end FlightPlanProcessing.impl;


thread AircraftPerformanceCalculation
features
   fpDataFromFPP      : in  data port
                        types::FPData;
   navDataFromIN      : in  data port
                        types::NavData;
   fuelFlow           : in  data port
                        types::FuelFlowData;
   perfDataToFPP      : out data port
                        types::PerformanceData;
properties
   Dispatch_Protocol        => Periodic;
   Period                   => 500 Ms;
   Compute_Execution_Time   => 1 Ms .. 100 Ms;
   Source_Data_Size         => 10 Kbyte;
   Source_Code_Size         => 10 Kbyte;
end AircraftPerformanceCalculation;

thread implementation AircraftPerformanceCalculation.impl
calls
   call1:
      {pspg : subprogram
       subprograms::spg_aircraftperformancecalculation;};
end AircraftPerformanceCalculation.impl;

thread PeriodicIO
features
   toOutside_guidanceOut              : out data port
                                        types::GuidanceData;
```

```
   toOutside_fpDataOut                : out data port
                                        types::FPData;
   toOutside_navDataOut               : out data port
                                        types::NavData;
   fromOutside_fuelFlow               : in  data port
                                        types::FuelFlowData;
   fromOutside_navSignal              : in  data port
                                        types::navSignalData;
   fuelFlow                           : out data port
                                        types::FuelFlowData;
   navSignal                          : out data port
                                        types::NavSignalData;
   guidanceOut                        : in  data port
                                        types::GuidanceData;
   fpDataOut                          : in  data port
                                        types::FPData;
   navDataOut                         : in  data port
                                        types::NavData;
   New_Page_Request_From_PCM          : in  data port
                                        types::PageRequestCmd;
   New_Page_Request_To_PageFeed       : out data port
                                        types::PageRequestCmd;
   New_Page_Request_From_PageFeed     : in  data port
                                        types::PageRequestCmd;
   New_Page_Request_To_FD             : out data port
                                        types::PageRequestCmd;
   New_Page_Content_from_FD           : in  data port
                                        types::PageContent;
   New_Page_Content_To_PageFeed       : out data port
                                        types::PageContent;
   New_Page_Content_From_PageFeed     : in  data port
                                        types::PageContent;
   New_Page_Content_To_PCM            : out data port
                                        types::PageContent;
properties
   Dispatch_Protocol        => Periodic;
   Period                   => 50 Ms;
   Compute_Execution_Time   => 1 Ms .. 1 Ms;
   Source_Data_Size         => 10 Kbyte;
   Source_Code_Size         => 10 Kbyte;
end PeriodicIO;

thread implementation PeriodicIO.impl
calls
   call1: {pspg : subprogram subprograms::spg_periodicio;};
connections
   port pspg.fuelFlow -> fuelFlow;
   port pspg.navSignal -> navSignal;
   port guidanceOut -> pspg.guidanceOut;
   port navDataOut -> pspg.navDataOut;
   port fpDataOut -> pspg.fpDataOut;
   port fromOutside_fuelFlow
            -> pspg.fromOutside_fuelFlow;
   port fromOutside_navSignal
            -> pspg.fromOutside_navSignal;
   port pspg.toOutside_guidanceOut
            -> toOutside_guidanceOut;
   port pspg.toOutside_fpDataOut
            -> toOutside_fpDataOut;
   port pspg.toOutside_navDataOut
            -> toOutside_navDataOut;
   port New_Page_Request_From_PCM
            -> pspg.New_Page_Request_From_PCM;
   port pspg.New_Page_Request_To_PageFeed
            -> New_Page_Request_To_PageFeed;
   port New_Page_Request_From_PageFeed
            -> pspg.New_Page_Request_From_PageFeed;
   port pspg.New_Page_Request_To_FD
```

```
            -> New_Page_Request_To_FD;
   port New_Page_Content_From_FD
            -> pspg.New_Page_Content_From_FD;
   port pspg.New_Page_Content_To_PageFeed
            -> New_Page_Content_To_PageFeed;
   port New_Page_Content_From_PageFeed
            -> pspg.New_Page_Content_From_PageFeed;
   port pspg.New_Page_Content_To_PCM
            -> New_Page_Content_To_PCM;
end PeriodicIO.impl;

thread HandlePageRequest
features
   New_Page_Request_From_PCM   : in  data port
                                 types::PageRequestCmd;
   New_Page_Content_To_PCM     : out data port
                                 types::PageContent;
   New_Page_Request_To_FD      : out data port
                                 types::PageRequestCmd;
   New_Page_Content_from_FD    : in  data port
                                 types::PageContent;
properties
   Dispatch_Protocol        => Periodic;
   Compute_Execution_Time   => 1 Ms .. 1 Ms;
   Period                   => 50 Ms;
   Source_Data_Size         => 10 Kbyte;
   Source_Code_Size         => 10 Kbyte;
end HandlePageRequest;

thread implementation HandlePageRequest.impl
calls
   call1: {pspg : subprogram subprograms::spg_handlepagerequest;};
end HandlePageRequest.impl;

-------------------------------------------
-- FLIGHT DIRECTOR
-------------------------------------------

thread thread_flightdirector
features
   FDToFM_Request        : in  data port
                           types::PageRequestCmd;
   FDToFM_Page           : out data port
                           types::PageContent;
properties
   Dispatch_Protocol          => Periodic;
   Compute_Execution_Time     => 1 Ms .. 1 Ms;
   Period                     => 50 Ms;
   Source_Data_Size           => 10 Kbyte;
   Source_Code_Size           => 10 Kbyte;
end thread_flightdirector;

thread implementation thread_flightdirector.impl
calls
   call1:
      {pspg : subprogram
              subprograms::spg_flightdirector;};
end thread_flightdirector.impl;

-----------------------------------------------
-- MFD
-----------------------------------------------

thread thread_mfd_driver
features
   DisplayToDM_Request   : out data port
                           types::PageRequestCmd;
   DisplayToDM_Page      : in  data port
```

```
                                types::PageContent;
properties
   Dispatch_Protocol            => Periodic;
   Compute_Execution_Time       => 1 Ms .. 1 Ms;
   Period                       => 50 Ms;
   Source_Data_Size             => 10 Kbyte;
   Source_Code_Size             => 10 Kbyte;
end thread_mfd_driver;

thread implementation thread_mfd_driver.i
calls
   call1: {pspg : subprogram subprograms::mfd_driver;};
connections
   port pspg.DisplayToDM_Request -> DisplayToDM_Request;
   port DisplayToDM_Page -> pspg.DisplayToDM_Page;
end thread_mfd_driver.i;

end threads;
```

B.4.1.8 Modélisation des types de données

```
package types
public

   with Data_Model;

  feature group PageRequest
    features
      request: out data port PageRequestCmd;
      page: in data port PageContent;
  end PageRequest;


  feature group PageReturn
    inverse of PageRequest
  end PageReturn;

  data PageRequestCmd
      properties
         Data_Model::Data_Representation => integer;
  end PageRequestCmd;

  data MenuCmd
      properties
         Data_Model::Data_Representation => integer;
  end MenuCmd;

  data PageContent
      properties
         Data_Model::Data_Representation => integer;
  end PageContent;

  data PageImage
      properties
         Data_Model::Data_Representation => integer;
  end PageImage;

  data NavSignalData
      properties
         Data_Model::Data_Representation => integer;
  end NavSignalData;

  data NavSensorData
      properties
         Data_Model::Data_Representation => integer;
```

```
  end NavSensorData;

  data NavData
      properties
         Data_Model::Data_Representation => integer;
  end NavData;

  data FuelFlowData
      properties
         Data_Model::Data_Representation => integer;
  end FuelFlowData;

  data GuidanceData
      properties
         Data_Model::Data_Representation => integer;
  end GuidanceData;

  data FPData
      properties
         Data_Model::Data_Representation => integer;
  end FPData;

  data PerformanceData
      properties
         Data_Model::Data_Representation => integer;
  end PerformanceData;

  feature group FMInData
    features
      fuelFlow: in data port FuelFlowData;
      navSignal: in data port NavSignalData;
  end FMInData;

  feature group FMInDataInv
    inverse of FMInData
  end FMInDataInv;

  feature group FMOutData
    features
      guidanceOut: out data port GuidanceData;
      fpDataOut: out data port FPData;
      navDataOut: out data port NavData;
  end FMOutData;

  feature group FMOutDataInv
    inverse of FMOutData
  end FMOutDataInv;

   data SharedData
      properties
         Data_Model::Data_Representation => integer;
   end SharedData;
end types;
```

B.4.1.9 Intégration des composants

```
package systems
public

with platform;
with partitions;
with memories;

system mysystem
end mysystem;
```

```
system implementation mysystem.impl
subcomponents
   cpu            : processor
                    platform::partitioned.x86;
   mem            : memory
                    memories::ram.i;
   WAM            : process
     partitions::process_warningannunciationmanager.impl;
   DM             : process
     partitions::process_displaymanager.impl;
   PCM            : process
     partitions::process_pagecontentmanager.impl;
   FM             : process
     partitions::process_flightmanager.pio;
   FD             : process
     partitions::process_flightdirector.impl;
   PilotDisplay   : device     partitions::mfd.i;
connections
   port DM.DMToDisplay_Page ->
        PilotDisplay.DisplayToDM_Page;
   port PilotDisplay.DisplayToDM_Request ->
        DM.DMToDisplay_Request;
   port PCM.PCMToFM_Request ->
        FM.FMToPCM_Request;
   port FM.FMToPCM_Page ->
        PCM.PCMToFM_Page;
   port PCM.PCMToDM_Page ->
        DM.DMToPCM_Page;
   port DM.DMToPCM_Request ->
        PCM.PCMToDM_Request;
   FMFD_Request:    port FM.FMToFD_Request ->
                         FD.FDToFM_Request;
   FMFD_Page_Page  :  port FD.FDToFM_Page ->
                      FM.FMToFD_Page;
properties
   actual_processor_binding =>
      (reference (cpu.part1))       applies to WAM;
   actual_processor_binding =>
      (reference (cpu.part2))       applies to DM;
   actual_processor_binding =>
      (reference (cpu.part3))       applies to PCM;
   actual_processor_binding =>
      (reference (cpu.part4))       applies to FM;
   actual_processor_binding =>
      (reference (cpu.part5))       applies to FD;
   actual_processor_binding =>
      (reference (cpu.part5))       applies to PilotDisplay;
   actual_memory_binding     =>
      (reference (mem.segment5_data),
       reference (mem.segment5_code))
      applies to WAM;
   actual_memory_binding     =>
      (reference (mem.segment2_data),
       reference (mem.segment2_code))
      applies to DM;
   actual_memory_binding     =>
      (reference (mem.segment6_data),
       reference (mem.segment6_code))
      applies to PCM;
   actual_memory_binding     =>
      (reference (mem.segment4_data),
       reference (mem.segment1_code))
      applies to FM;
   actual_memory_binding     =>
      (reference (mem.segment1_data),
       reference (mem.segment5_code))
```

```
      applies to FD;
   actual_memory_binding     =>
      (reference (mem.segment4_data),
       reference (mem.segment4_code))
      applies to PilotDisplay;
end mysystem.impl;


end systems;
```

B.4.2 Cas d'étude « *MILS* »

B.4.2.1 Niveaux de sécurité

```
package layers

public

with poklib;
with POK;

--
--  Security Layers (topsecret, secret, etc ...)
--  Extends virtual buses from the poklib
--

virtual bus secret
end secret;

virtual bus implementation secret.i
            extends poklib::des.i
properties
   POK::Security_Level => 2;

   POK::Blowfish_Key =>
         "{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
           0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87}";
   POK::Blowfish_Init =>
        "{0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10}";
end secret.i;

virtual bus topsecret
end topsecret;

virtual bus implementation topsecret.i
        extends poklib::blowfish.i
properties
   POK::Security_Level => 3;
   POK::Blowfish_Key =>
   "{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
          0xf0,0xe1,0xd2,0xc3,0xb4,0xa5,0x96,0x87}";
   POK::Blowfish_Init =>
        "{0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10}";
end topsecret.i;

virtual bus unclassified
end unclassified;

virtual bus implementation unclassified.i
            extends poklib::ceasar.i
properties
   POK::Security_Level => 0;
end unclassified.i;

end layers;
```

B.4.2.2 Mémoire

```
package memories

public

   memory memory_segment
   end memory_segment;

   memory implementation memory_segment.i
   properties
      Word_Count => 1100000;
   end memory_segment.i;

   memory ram
   end ram;

   memory implementation ram.sender
   subcomponents
      topsecret        : memory memory_segment.i;
      secret           : memory memory_segment.i;
      unclassified     : memory memory_segment.i;
      driver           : memory memory_segment.i
                         {Word_Count => 124000;};
   end ram.sender;

   memory implementation ram.receiver
   subcomponents
      topsecretandsecret       : memory memory_segment.i;
      unclassified             : memory memory_segment.i;
      driver                   : memory memory_segment.i
                                 {Word_Count => 120000;};
   end ram.receiver;

end memories;
```

B.4.2.3 Partitions

```
package partitions
public

with types;
with threads;
with layers;

process partition_secret_send
features
   valueout : out data port types::integer
      {Allowed_Connection_Binding_Class =>
            (classifier (layers::secret.i));};
end partition_secret_send;

process implementation partition_secret_send.i
subcomponents
   thr : thread threads::thr_send_int.i;
connections
   port thr.outvalue -> valueout;
end partition_secret_send.i;

process partition_topsecret_send
features
   valueout : out data port types::integer
      {Allowed_Connection_Binding_Class =>
            (classifier (layers::topsecret.i));};
end partition_topsecret_send;
```

```
process implementation partition_topsecret_send.i
subcomponents
   thr : thread threads::thr_send_int.i;
connections
   port thr.outvalue -> valueout;
end partition_topsecret_send.i;

process partition_unclassified_send
features
   valueout : out data port types::integer
      {Allowed_Connection_Binding_Class =>
            (classifier (layers::unclassified.i));};
end partition_unclassified_send;

process implementation partition_unclassified_send.i
subcomponents
   thr : thread threads::thr_send_int.i;
connections
   port thr.outvalue -> valueout;
end partition_unclassified_send.i;

process partition_unclassified_receive
features
   valuein : in data port types::integer
      {Allowed_Connection_Binding_Class =>
            (classifier (layers::unclassified.i));};
end partition_unclassified_receive;

process implementation partition_unclassified_receive.i
subcomponents
   thr : thread threads::thr_receive_int.i;
connections
   port valuein -> thr.invalue;
end partition_unclassified_receive.i;

process partition_secret_and_topsecret_receive
features
   valueone : in data port types::integer
      {Allowed_Connection_Binding_Class =>
            (classifier (layers::topsecret.i));};
   valuetwo : in data port types::integer
      {Allowed_Connection_Binding_Class =>
            (classifier (layers::secret.i));};
end partition_secret_and_topsecret_receive;

process implementation
   partition_secret_and_topsecret_receive.i
subcomponents
   thr : thread threads::thr_receive_two_int.i;
connections
   port valueone -> thr.invalue1;
   port valuetwo -> thr.invalue2;
end partition_secret_and_topsecret_receive.i;

end partitions;
```

B.4.2.4 Carte réseau

```
package rtl8029
public
with POK;
with types;
with runtime;
```

```
data anydata
end anydata;

subprogram init
properties
   source_name => "rtl8029_init";
   source_language => C;
end init;

subprogram poll
properties
   source_name => "rtl8029_polling";
   source_language => C;
end poll;


thread thr_poller
properties
   Dispatch_Protocol => Periodic;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Period => 100 Ms;
end thr_poller;

thread implementation thr_poller.i
calls
   call1 : { pspg : subprogram poll;};
end thr_poller.i;

thread thr_handler
features
   incoming_topsecret      : in data port types::integer;
   incoming_secret         : in data port types::integer;
   incoming_unclassified   : in data port types::integer;
   outgoing_topsecret      : out data port types::integer;
   outgoing_secret         : out data port types::integer;
   outgoing_unclassified   : out data port types::integer;
properties
   Dispatch_Protocol => Periodic;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Period => 1000 Ms;
   Source_Code_Size  => 3 Kbyte;
   Source_Data_Size  => 0 Kbyte;
   Source_Stack_Size  => 3000 Kbyte;
end thr_handler;

thread implementation thr_handler.i
connections
   port incoming_topsecret    -> outgoing_topsecret;
   port incoming_secret       -> outgoing_secret;
   port incoming_unclassified -> outgoing_unclassified;
end thr_handler.i;

process driver
end driver;

process implementation driver.i
subcomponents
   data_handler    : thread thr_handler.i;
   poller          : thread thr_poller.i;
properties
  POK::Needed_Memory_Size => 160 Kbyte;
end driver.i;

device rtl8029
features
   incoming_topsecret      : in data port types::integer;
   incoming_secret         : in data port types::integer;
   incoming_unclassified   : in data port types::integer;
```

```
      outgoing_topsecret       : out data port types::integer;
      outgoing_secret          : out data port types::integer;
      outgoing_unclassified    : out data port types::integer;
      ethernet_access          : requires bus access
                                 runtime::ethernet.unsecure;
   properties
      Initialize_Entrypoint => classifier (rtl8029::init);
      POK::Device_Name => "rtl8029";
   end rtl8029;

   abstract driver_container
   end driver_container;

   abstract implementation driver_container.i
   subcomponents
      p : process driver.i;
   end driver_container.i;


   device implementation rtl8029.i
   properties
      Device_Driver => classifier (rtl8029::driver_container.i);
   end rtl8029.i;

end rtl8029;
```

B.4.2.5 Runtime générique

```
package runtime

public

with POK;
with layers;

virtual processor partition_runtime
properties
   POK::Scheduler => RR;
   POK::Additional_Features => (console);
end partition_runtime;

virtual processor implementation partition_runtime.topsecret
properties
   Provided_Virtual_Bus_Class =>
      (classifier (layers::topsecret.i));
end partition_runtime.topsecret;

virtual processor implementation partition_runtime.secret
properties
   Provided_Virtual_Bus_Class =>
            (classifier (layers::secret.i));
end partition_runtime.secret;

virtual processor implementation partition_runtime.secretandtopsecret
properties
   Provided_Virtual_Bus_Class =>
      (classifier (layers::secret.i),
       classifier (layers::topsecret.i));
end partition_runtime.secretandtopsecret;

virtual processor implementation partition_runtime.unclassified
properties
   Provided_Virtual_Bus_Class =>
      (classifier (layers::unclassified.i));
end partition_runtime.unclassified;
```

```
processor pok_kernel
properties
   POK::Architecture => x86;
   POK::BSP => x86_qemu;
end pok_kernel;

bus ethernet
end ethernet;

bus implementation ethernet.unsecure
properties
   Provided_Virtual_Bus_Class =>
      (classifier (poklib::unencrypted));
end ethernet.unsecure;

end runtime;
```

B.4.2.6 Runtime noeud emetteur

```
package runtime_sender

public
with runtime;
with POK;

processor kernel_sender extends runtime::pok_kernel
end kernel_sender;

processor implementation kernel_sender.i
subcomponents
   partition_secret        : virtual processor
      runtime::partition_runtime.secret;
   partition_topsecret     : virtual processor
      runtime::partition_runtime.topsecret;
   partition_unclassified  : virtual processor
      runtime::partition_runtime.unclassified;
   partition_network       : virtual processor
      runtime::partition_runtime.unclassified
      {POK::Additional_Features => (stdlib, io, pci, stdio);};
properties
   POK::Major_Frame => 2000ms;
   POK::Scheduler => static;
   POK::Slots => (500ms, 500ms, 500ms, 500ms);
   POK::Slots_Allocation =>
      (reference (partition_secret),
       reference (partition_topsecret),
       reference (partition_unclassified),
       reference (partition_network));
end kernel_sender.i;


end runtime_sender;
```

B.4.2.7 Runtime noeud récepteur

```
package runtime_receiver

public
with runtime;
with POK;
```

```
processor kernel_receiver extends runtime::pok_kernel
end kernel_receiver;

processor implementation kernel_receiver.i
subcomponents
   partition_topsecretandsecret  : virtual processor
      runtime::partition_runtime.topsecret;
   partition_unclassified        : virtual processor
      runtime::partition_runtime.unclassified;
   partition_network             : virtual processor
      runtime::partition_runtime.unclassified
        {POK::Additional_Features => (stdlib, io, pci, stdio);};
properties
   POK::Major_Frame => 2000ms;
   POK::Scheduler => static;
   POK::Slots => (1000ms, 500ms, 500ms);
   POK::Slots_Allocation =>
      (reference (partition_topsecretandsecret),
       reference (partition_unclassified),
       reference (partition_network));
end kernel_receiver.i;

end runtime_receiver;
```

B.4.2.8 Threads

```
package threads
public

with types;

thread thr_receive_int
features
   invalue : in data port types::integer;
properties
   Dispatch_Protocol => Periodic;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Period => 1000 Ms;
   Source_Code_Size  => 3 Kbyte;
   Source_Data_Size  => 0 Kbyte;
   Source_Stack_Size  => 3000 Kbyte;
end thr_receive_int;

thread implementation thr_receive_int.i
calls
   call1 : { pspg : subprogram subprograms::print_integer;};
connections
   port invalue -> pspg.valuetoprint;
end thr_receive_int.i;

thread thr_send_int
features
   outvalue : out data port types::integer;
properties
   Dispatch_Protocol => Periodic;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Period => 1000 Ms;
   Source_Code_Size  => 3 Kbyte;
   Source_Data_Size  => 0 Kbyte;
   Source_Stack_Size  => 3000 Kbyte;
end thr_send_int;

thread implementation thr_send_int.i
calls
   call1 : { pspg : subprogram subprograms::produce_integer;};
```

```
connections
   parameter pspg.valueproduced -> outvalue;
end thr_send_int.i;

thread thr_receive_two_int
features
   invalue1 : in data port types::integer;
   invalue2 : in data port types::integer;
properties
   Dispatch_Protocol => Periodic;
   Compute_Execution_Time => 0 ms .. 1 ms;
   Period => 1000 Ms;
   Source_Code_Size  => 3 Kbyte;
   Source_Data_Size  => 0 Kbyte;
   Source_Stack_Size  => 3000 Kbyte;
end thr_receive_two_int;

thread implementation thr_receive_two_int.i
calls
   call1 : { pspg : subprogram subprograms::print_two_integer;};
connections
   parameter invalue1 -> pspg.firstvaluetoprint;
   parameter invalue2 -> pspg.secondvaluetoprint;
end thr_receive_two_int.i;

end threads;
```

B.4.2.9 Sous-programmes applicatifs

```
package subprograms

public
with types;

subprogram print_integer
features
   valuetoprint : in parameter types::integer;
properties
   source_name => "user_print_integer";
   source_language => C;
   Source_Text => ("../../../usercode.o");
   Source_Code_Size  => 3 Kbyte;
   Source_Data_Size  => 0 Kbyte;
end print_integer;

subprogram print_two_integer
features
   firstvaluetoprint     : in parameter types::integer;
   secondvaluetoprint    : in parameter types::integer;
properties
   source_name => "user_print_two_integer";
   source_language => C;
   Source_Text => ("../../../usercode.o");
   Source_Code_Size  => 3 Kbyte;
   Source_Data_Size  => 0 Kbyte;
end print_two_integer;

subprogram produce_integer
features
   valueproduced : out parameter types::integer;
properties
   source_name => "user_produce_integer";
   source_language => C;
   Source_Text => ("../../../usercode.o");
   Source_Code_Size  => 3 Kbyte;
```

```
   Source_Data_Size  => 0 Kbyte;
end produce_integer;


end subprograms;
```

B.4.2.10 Types

```
package types
public
   with Data_Model;

   data integer
   properties
      Data_Model::Data_Representation => integer;
   end integer;

   data ret_t
   properties
      Data_Model::Data_Representation => integer;
   end ret_t;

   data size_t
   properties
      Data_Model::Data_Representation => integer;
   end size_t;

   data str_t
   properties
      Data_Model::Data_Representation => string;
   end str_t;
end types;
```

B.4.2.11 Intégration des composants

```
package main

public

with POK;
with types;
with runtime_sender;
with runtime_receiver;
with rtl8029;
with partitions;
with memories;
with runtime;

system mils_architecture_sample
end mils_architecture_sample;

system implementation mils_architecture_sample.i
subcomponents

-- First, specify sender things.
   sender_processor                : processor
           runtime_sender::kernel_sender.i;
   sender_netif                    : device
            rtl8029::rtl8029.i
      {POK::Hw_Addr => "00:1F:C6:BF:74:06";};
   sender_partition_secret         : process
            partitions::partition_secret_send.i;
```

```
   sender_partition_topsecret     : process
            partitions::partition_topsecret_send.i;
   sender_partition_unclassified  : process
            partitions::partition_unclassified_send.i;
   sender_ram                     : memory
            memories::ram.sender;

-- Here, receiver components.
   receiver_processor                        : processor
            runtime_receiver::kernel_receiver.i;
   receiver_netif                            : device
            rtl8029::rtl8029.i
               {POK::Hw_Addr => "00:0F:FE:5F:7B:2F";};
   receiver_partition_topsecretandsecret     : process
            partitions::partition_secret_and_topsecret_receive.i;
   receiver_partition_unclassified           : process
            partitions::partition_unclassified_receive.i;
   receiver_ram                              : memory
            memories::ram.receiver;

   ethernet_bus       : bus runtime::ethernet.unsecure;

connections
--  Connections INSIDE the sender node
   port sender_partition_secret.valueout
               -> sender_netif.incoming_secret;

   port sender_partition_topsecret.valueout
               -> sender_netif.incoming_topsecret;

   port sender_partition_unclassified.valueout
               -> sender_netif.incoming_unclassified;

--  Connections INSIDE the receiver node
   port receiver_netif.outgoing_topsecret
            -> receiver_partition_topsecretandsecret.valueone;

   port receiver_netif.outgoing_secret
            -> receiver_partition_topsecretandsecret.valuetwo;

   port receiver_netif.outgoing_unclassified
            -> receiver_partition_unclassified.valuein;

--  Connections across nodes
   port sender_netif.outgoing_topsecret
        -> receiver_netif.incoming_topsecret
       {Actual_Connection_Binding => (reference (ethernet_bus));};

   port sender_netif.outgoing_secret
        -> receiver_netif.incoming_secret
       {Actual_Connection_Binding => (reference (ethernet_bus));};

   port sender_netif.outgoing_unclassified
        -> receiver_netif.incoming_unclassified
       {Actual_Connection_Binding => (reference (ethernet_bus));};

-- Bus accesses
   bus access ethernet_bus -> sender_netif.ethernet_access;
   bus access ethernet_bus -> receiver_netif.ethernet_access;
properties
-- Properties for the sender
   Actual_Processor_Binding =>
      (reference (sender_processor.partition_secret))
      applies to sender_partition_secret;

   Actual_Processor_Binding =>
      (reference (sender_processor.partition_topsecret))
      applies to sender_partition_topsecret;
```

```
   Actual_Processor_Binding =>
      (reference (sender_processor.partition_unclassified))
      applies to sender_partition_unclassified;

   Actual_Processor_Binding =>
      (reference (sender_processor.partition_network))
      applies to sender_netif;

   Actual_Memory_Binding    => (reference (sender_ram.driver))
      applies to sender_netif;

   Actual_Memory_Binding    => (reference (sender_ram.unclassified))
      applies to sender_partition_unclassified;

   Actual_Memory_Binding    => (reference (sender_ram.topsecret))
      applies to sender_partition_topsecret;

   Actual_Memory_Binding    => (reference (sender_ram.secret))
      applies to sender_partition_secret;

-- Properties for the receiver
   Actual_Processor_Binding =>
      (reference
       (receiver_processor.partition_topsecretandsecret))
      applies to receiver_partition_topsecretandsecret;

   Actual_Processor_Binding =>
      (reference (receiver_processor.partition_unclassified))
      applies to receiver_partition_unclassified;

   Actual_Processor_Binding =>
      (reference (receiver_processor.partition_network))
      applies to receiver_netif;

   Actual_Memory_Binding    =>
      (reference (receiver_ram.driver))
      applies to receiver_netif;

   Actual_Memory_Binding    =>
      (reference (receiver_ram.unclassified))
      applies to receiver_partition_unclassified;

   Actual_Memory_Binding    =>
      (reference (receiver_ram.topsecretandsecret))
      applies to receiver_partition_topsecretandsecret;
end mils_architecture_sample.i;

end main;
```

B.5 Code source de test des cas d'études

B.5.1 Programme de chiffrement

Le programme suivant a été utilisé pour reproduire le chiffrement des données du niveau de sécurité *secret* du cas d'étude *« MILS »* (section 6.3). Ce code s'appuie sur les fonctions de chiffrement fournies dans POK (code source disponible dans le répertoire `libpok/protocols/` du projet.

```
#include <stdio.h>
```

```
#include <stdint.h>

int main()
{
   int i = 0;
   uint64_t crypted;
   int csize;
   for (i = 0 ; i < 20 ; i++)
   {
      pok_protocols_des_marshall (&i, sizeof (int), &crypted, &csize);
      printf ("i=%d,crypted=0x%x\n", i , crypted);
   }
   return 0;
}
```