THÈSE / UNIVERSITÉ DE BRETAGNE OCCIDENTALE
Sous le sceau de l'Université Européenne de Bretagne

Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE
Mention : Science et Technologie de l'Information et de la Communication
École Doctorale Santé, Information, Communication, Mathématique, Matière

présentée par
Vincent Gaudel

Préparée dans le département informatique
Laboratoire CNRS LabSTICC

Des patrons de conception pour assurer l'analyse d'architectures : un exemple avec l'analyse d'ordonnancement

Thèse soutenue le 26 novembre 2014
devant le jury composé de :

Pierre DISSAUX
Ellidiss Technologies / invité
Jérôme HUGUES
Maître de Conférences, ISAE / examinateur
Laurent PAUTET
Professeur, Télécom ParisTech / rapporteur
Alain PLANTEC
Maître de Conférences, Université de Bretagne Occidentale / examinateur (co-directeur de thèse)
José RUFINO
Enseignant-Chercheur, Université de Lisbonne / examinateur
Frank SINGHOFF
Professeur, Université de Bretagne Occidentale / examinateur (directeur de thèse)
Jean-Pierre TALPIN
Professeur, INRIA Rennes / rapporteur
Yvon TRINQUET
Professeur, Université de Nantes / examinateur

Remerciements

Résumé

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

De nos jours, les systèmes embarqués sont présents partout autour de nous. Les applications sont multiples : avionique, automobile, téléphonie, distributeurs, domotique, etc. Ces systèmes sont constitués d'un ensemble de matériels et de logiciels ayant pour but l'accomplissement d'une mission. Leur particularité réside dans la limitation de leurs ressources (processeur, mémoire, énergie, etc) [LLS07]. Le dysfonctionnement d'un téléphone portable aura des conséquences peu importantes, tandis que le dysfonctionnement d'un logiciel embarqué dans un avion ou au sein d'un réacteur nucléaire causera des dégâts humains et matériels irréversibles. On parle alors de systèmes embarqués critiques.

Lorsque la correction d'un tel système ne dépend pas uniquement de la justesse des résultats qu'il produit, mais est également contrainte par le délai avec lequel ce dernier est capable de fournir ses résultats, le système est dit temps-réel [Sta92]. Ce dernier doit alors respecter, à l'exécution, un ensemble de contraintes temporelles.

Cette thèse étudie les systèmes temps-réel embarqués critiques et des méthodes servant à leur validation.

L'objectif de ce chapitre est d'introduire brièvement les contributions de cette thèse et les motivations de leur proposition. La partie 1.1 présente le contexte de nos travaux. La problématique traitée dans cette thèse est exposée en partie 1.2, suivie de la description de nos contributions dans la partie 1.3. La partie 1.4 présente le cadre et les projets de recherche auxquels nous avons contribué. Enfin, le plan de ce manuscrit est fourni en partie 1.5.

1.1 Le développement de systèmes embarqués temps-réel critiques

Le développement de systèmes temps-réel embarqués critiques (STRECs) est une activité complexe, pouvant difficilement être réduite au suivi d'un ensemble de règles de développement. Cependant, les différentes phases de développement d'un système aident à la représentation et la manipulation des informations, et peuvent grandement aider à l'analyse de problèmes de développement.

En théorie, le processus de développement peut être structuré selon les étapes suivantes : analyse des objectifs, identification des exigences, conception architecturale, conception détaillée des composants, intégration des composants, validation du système, mise en service et recette [LLS07].

Dans la pratique, le suivi linéaire de ces étapes peut être difficilement réalisable. En effet, l'ensemble des données d'un nouveau problème de conception n'est identifiable qu'une fois le processus de développement bien avancé. Cela implique de multiples itérations et retours sur les différentes phases présentées précédemment. De surcroît, plus les problèmes sont détectés tardivement, plus le coût des corrections à apporter est important [Boe81]. On souhaite donc pouvoir détecter de telles erreurs le plus tôt possible au cours du processus de développement.

1.1.1 La validation des systèmes embarqués temps-réel critiques

Il existe de nombreuses exigences à vérifier lors du développement de STRECs : fiabilité, sûreté, maintenabilité, sécurité, ainsi que le respect de leurs exigences temporelles. Les choix de conception, même réalisés très tôt lors du développement, impactent le comportement temporel des systèmes. La détection du non-respect de contraintes temporelles lors des phases de tests peuvent entraîner des modifications lourdes au niveau architectural.

Il est donc nécessaire de vérifier les exigences temporelles des STRECs le plus tôt possible lors du processus de développement. Les trois principales méthodes de vérification des performances temporelles des STRECs sont : la simulation, le *model-checking* et les méthodes analytiques [Nas07].

La simulation consiste à exécuter un modèle du système dans des conditions les plus proches possible du système réel [CGG04, CHD+14]. Cependant, il est en général impossible d'énumérer exhaustivement tous les états potentiels d'un système. Le plus souvent, seuls les états les plus significatifs sont étudiés. L'avantage de cette approche est sa capacité à vérifier des systèmes de grande taille ; son désavantage est qu'elle ne prouve pas la conformité du système à ses exigences.

A contrario, le *model-checking* consiste à abstraire le système à l'aide d'un langage formel. Des outils permettent alors d'énumérer exhaustivement les états du modèle. Un parcours de cet ensemble d'états couplé aux exigences du STREC réalise une vérification formelle du système. Parmi les approches les plus généralement appliquées, on trouve, entre autres, les réseaux de Petri [GCG02], les automates temporisés [AFM+04] et l'algèbre des processus [SLC06]. L'avantage de cette approche est la preuve formelle des résultats de la vérification. Ses désavantages résident dans l'assurance de la correction de l'abstraction du système et les difficultés liées au passage à l'échelle.

Lors de cette thèse, nous nous sommes concentrés sur les méthodes analytiques appelées tests de faisabilité. Les tests de faisabilité reposent sur un modèle du STREC analysé, associé à un ensemble d'équations et d'algorithmes. Ces derniers permettent de valider ou d'invalider le respect des exigences temporelles. L'ensemble de ces méthodes forme la théorie de l'ordonnancement [SAa+].

Les trois approches de vérification présentées ci-dessus utilisent un modèle du STREC

que l'on souhaite analyser. Le modèle abstrait le système analysé afin de ne manipuler que les informations nécessaires à la validation de son comportement temporel. La modélisation de STRECs est une activité complexe que nous introduisons dans la partie suivante.

1.1.2 La modélisation de systèmes temps-réel embarqués critiques

Krob définit l'architecture d'un système comme la partie invariante d'un système, i.e., qui est considérée comme fixe au cours du temps [Kro09]. Elle sert également de point d'appui lors du développement d'un système pour la représentation de ce dernier. En effet, l'architecture va servir de modèle du système sur lequel on peut agir afin de gérer son évolution. Une architecture peut être décrite à l'aide d'un ensemble de composants et de leurs interactions. Ici, un composant est défini comme une unité destinée à être assemblée et à fonctionner avec d'autres [MT00]. Ces composants modélisent donc les aspects matériels et logiciels : processeurs, bus, buffers, tâches, espaces d'adressage, etc.

L'architecture permet de spécifier et manipuler les aspects structurels (tant logiciels que matériels) de la solution choisie pour implanter un système [Ker05]. Cette spécification a pour objectif de supporter un raisonnement sur ces derniers [Gol13a]. La façon dont les ressources matérielles sont affectées aux composants logiciels impacte les performances temporelles du STREC. Les systèmes embarqués fournissent, par définition, une quantité de ressources limitées et parfois dédiées à des opérations spécifiques. De plus, des conflits dûs à l'utilisation des ressources matérielles peuvent également apparaître, c'est par exemple le cas lors de l'utilisation de multiples processeurs. La prise en compte de ces deux types de composants dans le processus de développement est donc primordiale.

L'architecture logicielle

L'architecture logicielle d'un STREC est modélisée dans l'optique de diverses analyses et de la génération de code. Elle identifie les différentes tâches que le système doit réaliser, ainsi que leurs interactions et interdépendances. L'architecture logicielle est la pièce centrale de la conception de STRECs. Afin de pouvoir réaliser l'analyse d'un STREC, les entités logicielles sont caractérisées par des informations sur leur comportement (temps d'exécution et politique d'activation d'une tâche par exemple). On parle alors de modèle de tâches [OGRR12].

L'architecture matérielle

La conception des systèmes embarqués temps-réel ne peut se faire sans prendre en compte leur partie matérielle. Les éléments modélisant l'architecture matérielle sont, entre autres, les processeurs, leurs cœurs, les bus de communication et les espaces d'adressage. En effet, le bon comportement des STRECs est défini par l'exécution de l'architecture logicielle sur la plate-forme matérielle. Cette dernière impacte fortement le comportement du système en termes de performances (de par ses capacités de calcul par exemple) et de résistance aux fautes. À titre d'exemple, le partitionnement de l'application sur différents cœurs de processeurs permet de réaliser une isolation temporelle des différentes parties logicielles du

système et de contenir la propagation des erreurs. Par ailleurs, les changements de contextes lors de la suspension de l'exécution d'une tâche et de son redéploiement sur un autre cœur peuvent être assez longs pour être pris en compte lors de l'étude des performances temporelles d'un STREC.

Les langages de description d'architectures fournissent les outils nécessaires à la modélisation de ces architectures. Au cours des dernières décennies, de nombreux langages de modélisation d'architectures ont été proposés. Ces derniers s'imposent de plus en plus comme une approche courante [HB14].

1.1.3 L'utilisation des langages de description d'architecture au sein de chaînes d'outils de développement

Dans cette partie, nous présentons les langages de description d'architectures et la façon dont ils peuvent être utilisés dans les chaînes d'outils pour le développement et l'analyse de STRECs.

Les langages de description d'architectures

Les langages de description d'architectures (Architecture Description Languages ou ADLs) sont des langages de modélisation spécifiques à un domaine. De nombreux ADLs permettent aux développeurs de STRECs de spécifier, formellement ou non, l'architecture d'un STREC. Habituellement, les ADLs fournissent une abstraction de composants, de connexions et de déploiements [MT00, Ves93] et permettent la spécification de contraintes temporelles. De nombreux ADLs modélisent les parties logicielles et matérielles d'un STREC à l'aide de composants spécifiques. Les connexions modélisent des relations entre différents composants et les déploiements spécifient comment les composants logiciels sont déployés sur les composants matériels, i.e., comment les ressources du système sont allouées. Il existe de nombreux ADLs spécifiques à la modélisation de systèmes temps-réel embarqués critiques. Nous avons identifié deux approches dans leur conception.

La première catégorie d'ADLs rassemble ceux pouvant servir de langage pivot entre les différentes utilisations que le concepteur aura de son modèle d'architecture tout au long du processus de développement. Ces ADLs sont donc des langages riches, permettant la spécification d'informations très diverses qui ne sont pas toujours corrélées (le temps d'exécution des tâches et le mécanisme de propagation d'erreurs par exemple). Parmi les ADLs les plus connus dans cette catégorie, on retrouve AADL [Fei04], OMG SysML [H+06], MARTE [Obj05] ou encore AUTOSAR [Bun11], entre autres. Ces ADLs proposent un niveau de détails permettant de nombreuses utilisations et sont donc plus complexes à appréhender.

La seconde catégorie comprend les ADLs dédiés à une analyse ou à une pratique particulière. Ces derniers ne permettent de modéliser les systèmes que dans l'optique d'un objectif précis, comme la vérification du respect des exigences temporelles d'un STREC par exemple. Ces ADLs sont donc dépendants d'une approche ou d'un outil. Ils permettent la modélisation des systèmes à un niveau d'abstraction spécifique à l'utilisation pour la-

quelle ils ont été conçus. Cheddar ADL [FT13] et MOSART [OGRR12] sont deux exemples d'ADLs dédiés à l'utilisation de tests de faisabilité.

L'utilisation de chaînes d'outils de développement

Une des approches possibles pour le développement de systèmes temps-réel embarqués critiques est la définition de chaînes d'outils, basées sur un seul et unique ADL pivot. Un modèle d'architecture est alors défini et raffiné au cours des différentes étapes du processus de développement. Ce modèle est partagé entre les différents outils d'analyse, d'édition de modèles et de génération de code. Le modèle exprimé à l'aide de l'ADL pivot est ensuite traduit à l'aide de transformations de modèles vers les ADLs dédiés, manipulés par chacun des outils de la chaîne [MMPT10, KBG12a].

Du point de vue de la modélisation, le challenge est de garantir que le modèle pivot se prête aux différentes utilisations qu'en font les outils. En effet, un outil d'analyse d'ordonnancement requiert que le modèle contienne l'ensemble des informations du modèle de tâche manipulé par l'analyse. Par exemple, l'application des politiques de sécurité se concentre sur les patrons de communication, tandis que la gestion des erreurs traite des mécanismes de propagation d'erreurs. Un des enjeux de cette approche est donc la correction de la spécification des concepts nécessaires au bon fonctionnement des différents outils et méthodes d'analyse.

1.2 Applicabilité des méthodes d'analyse et interopérabilité des outils au cours du cycle de développement

Comme nous l'exposions précédemment, la validation d'un STREC est très importante et peut reposer sur l'utilisation de chaînes d'outils spécifiques. Par exemple, la théorie de l'ordonnancement est maintenant relativement mature. Il existe des outils assistant les concepteurs tout au long des phases de développement des STRECs. Cependant, l'utilisation de la théorie de l'ordonnancement et des différents outils reste peu présente dans les pratiques industrielles. Les problématiques traitées dans ce mémoire s'articulent selon deux axes. Le premier axe consiste en l'étude de l'applicabilité des tests de faisabilité proposés par les outils d'analyse d'ordonnancement ; le second axe traite de l'interopérabilité des outils manipulant un même ADL pivot. Ces deux problématiques sont exposées ci-dessous.

1.2.1 Applicabilité des méthodes d'analyse d'ordonnancement

Depuis les années 70, la théorie de l'ordonnancement temps-réel propose des méthodes analytiques appelées tests de faisabilité [LL73b, SAa+]. Un test de faisabilité permet à un concepteur de vérifier *a priori* l'ordonnançabilité d'un système. Ce domaine a donné lieu à de très nombreuses recherches et expérimentations [G+96a, CDKM02, SRL90a, SAa+].

Cependant, les tests de faisabilité sont spécifiques à certains types de systèmes. En effet, chaque test considère que le STREC qu'il analyse, respecte un ensemble donné d'hy-

pothèses. Dans le cas contraire, l'application du test de faisabilité résulte en une erreur, menant à l'absence de résultat ou à un résultat erroné.

De plus, la quantité de tests existants est très importante, par conséquent, devenir expert du domaine est une tâche longue et coûteuse [SPDL09].

Par ailleurs, il existe des architectures pour lesquelles cette théorie n'est actuellement pas applicable, ou tout du moins pas assez mature (les architectures multiprocesseurs particulièrement) [DB11].

Ainsi, déterminer quels tests de faisabilité sont applicables est une tâche complexe et coûteuse. C'est une des raisons pour lesquelles la théorie de l'ordonnancement reste globalement inappliquée par le monde industriel, même si cela pourrait être très profitable [DLP+10].

Il est donc nécessaire d'assister les concepteurs de STRECs lors du processus de sélection de tests de faisabilité adaptés. Une solution est d'expliciter des associations entre des ensembles de STRECs et les tests de faisabilité leur étant applicables.

1.2.2 Interopérabilité des outils

Dans les chaînes d'outils de développement qui manipulent des modèles d'architectures exprimés à l'aide d'ADLs pivots, les différents outils peuvent reposer sur des ADLs dédiés modélisant le système au niveau d'abstraction spécifique à leurs objectifs. Ces outils implantent des méthodes d'analyse complexes nécessitant un haut niveau d'expertise dans leurs domaines respectifs. Ils sont souvent développés par des équipes indépendantes, avec une compréhension (et donc une utilisation) des ADLs pivots qui leur est propre [HB14].

De ce fait plusieurs équipes peuvent modéliser un même concept de façons différentes à l'aide d'un même ADL. Les règles de transformation de modèles d'un ADL pivot vers un ADL dédié à un outil, varient d'un outil à l'autre. Les ADLs pivots sont donc sujets à diverses interprétations. Cela implique que même si les ADLs pivots fournissent une syntaxe ainsi qu'une sémantique communes et précises, cela n'implique pas nécessairement l'interopérabilité des outils.

De plus, la grande variété des STRECs implique que chacun des outils ne supporte pas l'ensemble des systèmes pouvant être modélisés, mais un sous-ensemble de cas identifiés comme caractéristiques, ou plus simplement traitables par la méthode d'analyse. Les sous-ensembles de systèmes supportés varient donc d'un outil à l'autre, et sont caractérisés par l'utilisation faite des modèles. Les informations définissant ces sous-ensembles sont donc définis par l'implantation des outils.

Un concepteur de STRECs doit donc prendre en compte les sous-ensembles supportés pas les différents outils. Cela crée un paradoxe : les modèles ne sont plus conçus uniquement pour remplir les exigences du système, mais également pour que le modèle réponde aux besoins spécifiques des outils.

La spécification de ces besoins permet d'expliciter l'utilisation faite de l'ADL pivot par les outils. Cette spécification doit être indépendante des outils qu'elle caractérise. Elle doit également définir les sous-ensembles du langage de description d'architecture pivot que ces outils supportent. Du point de vue des concepteurs de chaînes d'outils, la comparaison de

ces spécifications est également un enjeu important. En effet, être capable de comparer les sous-ensembles d'ADL supportés permet de déterminer quels outils peuvent être utilisés conjointement ou non.

1.3 Contributions

Les contributions de cette thèse peuvent être résumées en quatre points. Premièrement, nous proposons d'expliciter les relations entre des modèles de STRECs et les tests de faisabilité leur étant applicables à l'aide de patrons de conception architecturaux spécifiques aux STRECs déployés sur des environnements d'exécution mono-processeur. Ces patrons permettent la sélection automatique de tests de faisabilité. Deuxièmement, nous étendons cette approche à des STRECs comprenant une architecture plus complexe : les STRECs à ordonnancement hiérarchique. Troisièmement, nous proposons un logiciel de sélection de tests de faisabilité. Ce logiciel implante nos deux premières contributions et est intégré à Cheddar, un environnement d'analyse de STRECs [SPDL09]. La dernière contribution est une ouverture de notre approche au problème de l'interopérabilité entre les outils. Pour ce faire, nous proposons un langage dédié permettant de spécifier les sous-ensembles d'un ADL supportés par des outils. Une méthode de comparaison des différents sous-ensembles ainsi spécifiés est également fournie.

Chacune des contributions énumérées ci-dessus est présentée dans la suite de cette partie.

Les patrons de conception architecturaux pour le cas mono-processeur non-hiérarchique

La sélection de tests de faisabilité pour un STREC donné n'est pas chose aisée. Afin d'assister le concepteur d'un STREC dans cette tâche, Singhoff *et al.* ont proposé quatre patrons de conception basés sur des pratiques industrielles : *Time-Triggered Uniprocessor*, *Ravenscar Uniprocessor*, *Blackboard Uniprocessor* et *Queuedbuffer Uniprocessor* [SPDL09]. Ces patrons de conception sont constitués d'un ensemble de contraintes caractérisant les STRECs. Pour chacun de ces patrons, nous sommes capables de déterminer une liste de tests de faisabilité applicables.

Dans le cadre de cette thèse, nous définissons les patrons de conception architecturaux. Nous les positionnons vis-à-vis des autres approches de spécification de patrons de conception.

Nous proposons de modéliser nos patrons comme deux ensembles distincts de contraintes. Le premier ensemble de contraintes caractérise l'environnement d'exécution, tandis que le second caractérise les protocoles de communication et de synchronisation entre les entités logicielles des architectures de STRECs.

Nous spécifions un patron de conception supplémentaire : *Unplugged Uniprocessor*.

Nous proposons une approche de composition de patrons de conception permettant l'analyse de STRECs combinant l'utilisation de différents protocoles de communication et de synchronisation, sous réserve que le système soit analysable.

Un algorithme de sélection automatique de tests de faisabilité est également fourni. Nous montrons qu'il est possible de générer du code intégrable à Cheddar.

Extension de l'approche à l'analyse des STRECs à ordonnancement hiérarchique

L'automatisation de la sélection des tests de faisabilité pour le cas de STRECs mono-processeur n'est pas suffisante pour pallier aux besoins des concepteurs d'architectures d'aujourd'hui. Les architectures multi-processeurs et à ordonnancement hiérarchique sont désormais utilisées fréquemment, on pourra citer, par exemple, le cas du standard pour l'avionique ARINC653 [Ari97].

Nous proposons de généraliser l'approche précédente. Nous exposons comment combiner les ensembles de contraintes utilisés pour le cas mono-processeur, en graphes de contraintes. Ces graphes permettent de caractériser des architectures plus complexes que dans le cas précédent. À titre d'exemple, nous proposons trois patrons de conception modélisés par des graphes de contraintes et spécifiques aux STRECs à ordonnancement hiérarchique.

Nous proposons alors, un algorithme de sélection de tests de faisabilité prenant en compte la modélisation des patrons par des graphes de contraintes.

Cette approche nous a permis de mener une réflexion sur la modélisation générique des STRECs à ordonnancement hiérarchique à l'aide de langages de description d'architecture. Cela nous a conduit à modifier le langage d'architecture dédié de Cheddar.

La réalisation d'un logiciel de sélection de tests de faisabilité

Les deux contributions présentées précédemment ont été évaluées par la réalisation d'un prototype. Ce logiciel est intégré à Cheddar et a été un moteur dans l'évolution de ce dernier.

Les travaux traitant des STRECs à ordonnancement hiérarchique ont également donné lieu à l'implantation de nouveaux tests de faisabilité, étendant ainsi les capacités d'analyse de Cheddar. Cette réalisation est susceptible d'être intégrée à AADLInspector, un logiciel d'édition et d'analyse de modèles. AADLInspector est centré sur le langage AADL et proposé par Ellidiss Technologies [Ell].

De l'interopérabilité entre méthodes d'analyse vers l'interopérabilité entre outils : les *subsets* AADL

Afin de poursuivre les travaux précédemment présentés, nous avons étudié un autre exemple d'utilisation des contraintes constituant les patrons de conception. Il ne s'agit désormais plus d'expliciter une relation entre des tests de faisabilité et un patron de conception, mais de définir la portée d'outils basés sur un même ADL pivot.

Ces outils sont ceux utilisés par les concepteurs de systèmes temps-réel tout au long du cycle de développement : éditeurs de modèles, outils d'analyse, méthodes formelles ou encore générateurs de code.

Nous proposons de spécifier la portée des outils à l'aide de sous-ensembles du langage pivot considéré. Nous nommons ces sous-ensembles les *subsets*. Un *subset* est constitué de contraintes similaires à celles caractérisant les patrons de conception.

Nous proposons ASSET, un langage dédié à la spécification des *subsets* d'ADLs.

Nous proposons également quatre opérateurs de comparaison de ces derniers. Ces opérateurs sont utiles à l'étude de la compatibilité des différents outils.

Cette approche est appliquée au langage d'architecture AADL (Architecture Analysis & Design Language). Ces travaux ont eu lieu dans le cadre d'une collaboration avec le comité de standardisation AADL et a conduit à la proposition d'une annexe au langage.

1.4 Cadre et projets de recherche

Au cours de ma thèse, mes travaux se sont intégrés aux projets SAPIENT [CSR+12] et SMART [DMR+14] ainsi qu'au comité de standardisation SAE-AADL.

Le projet SAPIENT (Scheduling Analysis Principles and Tool for Time- and Space-Partitioned System) a été financé par l'EGIDE. Il résulte d'un partenariat entre les laboratoires LaSIGE/FCUL (Université de Lisbonne) et l'UBO de 2012 à 2013. L'objectif de ce projet a été de faire évoluer les capacités analytiques de Cheddar, afin de supporter les systèmes temps-réel partitionnés hiérarchiques. Les travaux que nous avons réalisés lors de ce partenariat sont, pour la partie concernant cette thèse, décrits dans le chapitre 5.

Le projet SMART (Simulation Multi-Agent d'ARchitectures Temps-réel) [DMR+14] est financé par la région Bretagne et résulte d'un partenariat entre le Lab-STICC et les industriels Ellidiss et Virtualys. Ce projet s'est concentré sur l'analyse temporelle et la simulation des architectures temps-réel multi-processeurs. Son objectif est d'émuler le contrôle commande de ces architectures et d'en animer une représentation graphique tri-dimensionnelle.

Enfin, j'ai eu l'opportunité de participer à de multiples reprises au comité de standardisation SAE-AADL. Les travaux relatifs aux subsets AADL (cf. chapitre 6) y ont été présentés. Les nombreuses présentations et discussions menées au sein de ce groupe de travail ont débouché sur la proposition d'une première version d'annexe au standard, soumise lors du comité d'octobre 2013.

1.5 Plan

La suite de ce mémoire de thèse est organisée de la façon suivante.

Chapitre 2 - Vérification des spécifications des architectures temps-réel

Le chapitre 2 introduit l'ensemble des notions nécessaires à la bonne compréhension de nos travaux par le biais d'un état de l'art. Nous y présentons les systèmes temps-réel embarqués critiques ainsi que leur processus de développement. Une présentation de différents ADLs est ensuite proposée.

Chapitre 3 - Survol de l'approche

Le chapitre 3 présente un survol de notre approche. Nous y exposons la problématique abordée par cette thèse. Une présentation de nos contributions est ensuite fournie. Elle inclut la description de notre approche du point de vue du concepteur de STRECs, la définition et le modèle de nos patrons de conception. Les hypothèses de travail de chacune de nos contributions sont exposées. Enfin, nous positionnons nos travaux vis-à-vis des travaux déjà existants.

Chapitre 4 - Exemples de patrons de conception pour l'analyse des STRECs mono-processeur

Le chapitre 4 présente les patrons de conception architecturaux dédiés aux STRECs mono-processeur. Nous y expliquons également la façon dont nous modélisons les associations entre patrons de conception architecturaux et tests de faisabilité. L'approche de composition des patrons de conception est décrite. Nous y proposons ensuite, un algorithme de sélection de tests de faisabilité.

Chapitre 5 - Extension de l'approche à l'analyse des STRECs à ordonnancement hiérarchique

Le chapitre 5 traite de l'extension de l'approche précédente aux systèmes utilisant de l'ordonnancement hiérarchique. Cette ouverture vers un nouveau type de système permet de présenter nos travaux dans le cas de systèmes ayant différents types d'environnements d'exécution et des approches d'analyse différentes (l'analyse compositionnelle particulièrement). Après avoir décrit les propriétés de ce type de STREC, nous détaillons un modèle de patron de conception adapté. Nous proposons trois patrons spécifiques aux STRECs à ordonnancement hiérarchique ainsi qu'un algorithme de sélection de tests de faisabilité pour le cas hiérarchique.

Chapitre 6 - Mises en œuvre

Le Chapitre 6 est dédié aux mises en œuvre réalisées dans le cadre du projet Cheddar et ayant pour but de valider notre approche. Après avoir décrit le processus d'ingénierie de l'environnement d'analyse Cheddar, nous expliquons comment nous en avons tiré parti. Les différentes modifications apportées à l'environnement sont alors détaillées.

Chapitre 7 - Exemple d'application des patrons de conception architecturaux : la définition de subsets d'AADL

Enfin, dans le chapitre 7, nous généralisons notre approche de spécification de l'interopérabilité des outils, centrés autour des langages de description d'architectures. Pour ce faire, nous proposons la définition de sous-ensembles de ces langages à l'aide de *subsets*. Cette approche est appliquée dans le cas du langage de description d'architectures AADL. Nous proposons ASSET, un langage dédié à la spécifiaction des *subsets* AADL.

Chapitre 8 - Bilan et perspectives

Ce mémoire est conclu par le chapitre 8. Nous y résumons les contributions de cette thèse. Nous dressons un bilan du travail effectué et discutons ses perspectives.

Chapitre 2

Spécification et vérification des architectures temps-réel

Les contributions majeures de cette thèse traitent de l'applicabilité des méthodes d'analyse d'ordonnancement des systèmes temps-réel embarqués critiques et de l'interopérabilité des outils utiles lors de leur processus de développement. L'objectif de cette partie est de présenter les domaines abordés dans ce manuscrit. Dans un premier temps, nous introduisons les définitions des concepts relatifs aux systèmes temps-réel embarqués critiques (STRECs) dans la partie 2.1. Les tâches sont abordées dans la partie 2.2. La partie 2.3 définit ce qu'est l'ordonnancement et expose quelques algorithmes. La partie 2.4 présente la théorie de l'ordonnancement, un ensemble de méthodes d'analyse du comportement temporel des STRECs. La partie 2.5 discute du processus de développement des STRECs et plus particulièrement du cycle en V. Enfin, la partie 2.6 introduit brièvement l'ingénierie dirigée par les modèles et décrit trois langages de description d'architecture utilisés lors du développement de STRECs.

2.1 Généralités à propos des systèmes temps-réel

Nous définissons, ici, ce qu'est un système temps-réel embarqué critique. Cette présentation est faite à l'aide d'une liste de définitions des concepts de base de ce domaine.

Définition 2.1 ***(Système temps-réel).*** *Un système temps-réel est un système dont la correction ne dépend pas uniquement de ses résultats fonctionnels, mais également du temps que lui prend la production de ces résultats [Sta88, JP86a].*

Un résultat correct fonctionnellement, mais ne respectant pas les contraintes temporelles du système, est alors considéré comme incorrect. La correction temporelle est généralement représentée sous forme d'échéances devant être respectées.

Définition 2.2 ***(Système embarqué).*** *Un système embarqué est un système disposant de ressources limitées. Ces ressources peuvent être : le processeur, la mémoire, l'énergie, etc.*

Un tel système est composé d'un ensemble d'éléments matériels et logiciels ayant pour objectif la réalisation d'une mission réalisée en interaction avec leur environnement [LLS07].

Il existe de nombreuses classifications pour les systèmes temps-réel embarqués. Classiquement, nous les classons selon leur niveau de criticité [CDKM00]. La figure 2.1 positionne, en ordre croissant, les niveaux de criticité présentés ci-dessous.

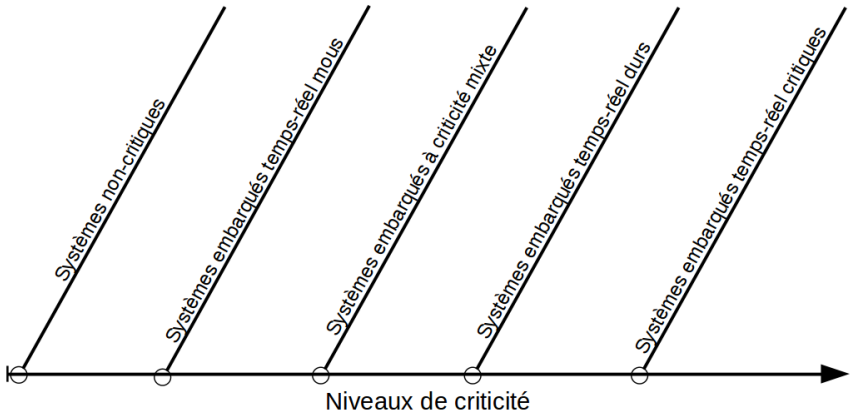

FIGURE 2.1 – Les niveaux de criticité triés par ordre croissant de criticité.

1. Les systèmes embarqués temps-réel mous : ce type de système temps-réel peut tolérer des dépassements d'échéances. Le système peut s'en accommoder dans certaines limites, au delà desquelles il devient inutilisable. C'est le cas des systèmes de visio-conférence, des jeux en réseaux, etc [GL99].
2. Les systèmes embarqués temps-réel à criticité mixte : ce type de système est contraint par deux types d'échéances, celles ne pouvant pas être dépassées et d'autres dont la violation peut être tolérée [BBD11]. Cependant, il y a des limites sur le nombre de fois où les échéances peuvent ne pas être respectées. Les systèmes gérant, conjointement, le contrôle de vol et le système d'information de vol au sein d'un avion, en sont une bonne illustration. En effet, le système de contrôle ne tolère aucun dépassement d'échéance, au contraire de celui du système d'information.
3. Les systèmes embarqués temps-réel durs : ce type de système ne supporte aucun dépassement d'échéances [Moi85]. Parmi ces systèmes, on retrouve typiquement les chaînes de montage automatisées.

4. Les systèmes embarqués temps-réel critiques : un système temps-réel dur est dit critique si son dysfonctionnement peut avoir pour conséquence d'entraîner des dégâts sur les personnes ou l'environnement [Bur91]. On y retrouve les systèmes de contrôle aérien ou les centrales nucléaires. La validation d'un tel système est donc très importante.

Les définitions présentées ci-dessus spécifient le type de système que nous considérons : les systèmes temps-réel embarqués critiques (STRECs dans la suite de cette thèse).

Un STREC est le plus souvent modélisé comme un ensemble d'éléments architecturaux logiciels et matériels. Cottet et al. identifient les éléments architecturaux suivants [CDKM00] :

1. les éléments passifs tels que les ressources physiques (périphériques, capteurs, moteurs) ou logiques (tampons mémoire, fichiers, etc) ;
2. les éléments actifs que sont les tâches ;
3. les éléments de communication (messages, données partagées, ports et canaux de communication) et de synchronisation (évènements, sémaphores, objets protégés, etc) ;
4. les éléments matériels comme les processeurs, mémoires cache et périphériques réseaux.

2.2 Les tâche dans les STRECs

Liu & Layland ont proposé, dès 1973 [LL73a], un modèle de tâche permettant de représenter les entités constituant la partie logicielle des STRECs. Ce modèle de tâche a ensuite été enrichi au fil des décennies suivantes [SAa+].

Définition 2.3 ***(Tâche).*** *Une tâche τ est une unité d'exécution de la partie logicielle d'un STREC. À chaque fois qu'une tâche est activée, elle doit exécuter un ensemble d'instructions séquentielles. Chaque exécution de cet ensemble d'instructions est appelée travail de la tâche τ [AB90].*

Le cycle de vie d'une tâche est généralement représentée par la machine à états schématisée par la figure 2.2.

1. Inactive : une fois créée, une tâche commence par être *inactive*, elle ne s'exécute pas. La tâche est en attente d'un message d'activation. Elle n'effectue aucun traitement ou calcul. Lorsqu'un évènement indiquant son activation intervient, la tâche devient *prête*.
2. Prête : la tâche dispose de toutes les ressources pour s'exécuter sauf le processeur. Elle attend alors d'être élue, parmi l'ensemble des tâches du système, afin de pouvoir s'exécuter. Elle obtient alors l'accès à la ressource processeur et devient *active*.

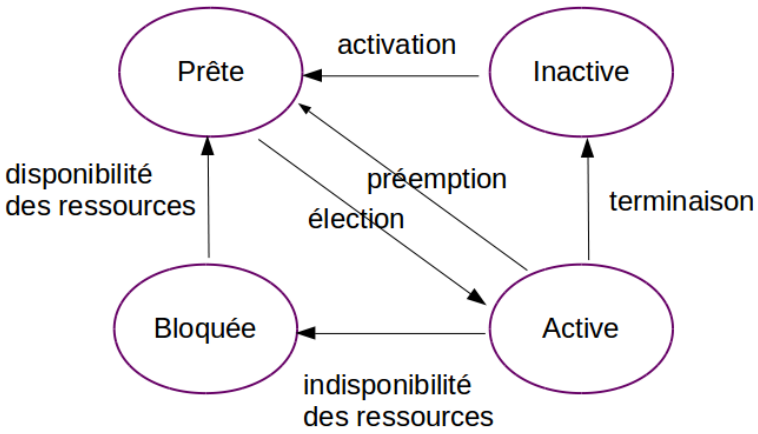

FIGURE 2.2 – Représentation de la sémantique d'une tâche à l'aide d'une machine à états.

3. Active : la tâche est en train de s'exécuter sur le processeur. Si, lors de son exécution, la tâche requiert un accès à une ressource non-disponible, elle devient *bloquée*. De plus, si une autre tâche est élue sur le même processeur, la tâche est suspendue et retourne dans l'état *prête*, en attente d'être de nouveau élue afin de reprendre l'exécution là où elle s'est arrêtée. On parle alors de préemption. Si la tâche arrive au terme de son exécution, elle redevient *inactive*.
4. Bloquée : la tâche est en attente de disponibilité d'une ressource autre que le processeur (donnée partagée, bus de communication, fichier, etc). Une fois la ressource libérée, la tâche redevient *prête*, en attente d'être élue à nouveau.

Les types de tâches

Il existe différents types de tâches, caractérisées par les occurrences de leurs travaux. Dans cette partie, nous allons décrire trois types de tâches : les périodiques, les sporadiques et les apériodiques. Il existe de nombreuses autres façons de modéliser les tâches. Nous nous limitons, ici, à ces trois types.

Définition 2.4 ***(Tâche périodique).*** *Une tâche périodique τ est une tâche pour laquelle l'activation est déclenchée à intervalles réguliers. Cet intervalle est appelé la période de τ [CDKM00].*

Définition 2.5 ***(Tâche sporadique).*** *Une tâche sporadique est une tâche pour laquelle on ne connaît pas la période, mais dont on connaît l'intervalle minimal séparant deux activations successives [CDKM00].*

Définition 2.6 ***(Tâche apériodique).*** *Une tâche apériodique est une tâche activée une seule fois [CDKM00].*

D'autre part, les tâches peuvent également être classées selon les échanges qu'elles entretiennent. On distingue alors deux types de tâches : les tâches dépendantes et les tâches indépendantes.

Définition 2.7 ***(Tâches dépendantes).*** *Deux tâches sont dites dépendantes si elles peuvent se synchroniser durant leur exécution [CDKM00, AB90].*

Définition 2.8 ***(Tâches indépendantes).*** *Les tâches indépendantes sont des tâches qui n'interagissent pas les unes avec les autres. [CDKM00, AB90].*

En plus des occurrences de leurs travaux et de leurs interactions, les tâches sont caractérisées par l'instant de leur première activation.

Définition 2.9 ***(Tâches synchrones).*** *Des tâches sont dites synchrones si leurs premiers travaux sont activés au même instant [CDKM00].*

Définition 2.10 ***(Tâches asynchrones).*** *Des tâches sont dites asynchrones s'il y a au moins une tâche dont le premier travail n'est pas activé au même instant que les premiers travaux des autres tâches [CDKM00].*

En plus de ces paramètres temporels et de leurs dépendances, les tâches sont également caractérisées par leur préemptivité.

Définition 2.11 ***(Préemptivité).*** *La préemptivité correspond à la possibilité de reprendre une ressource à une tâche sans que celle-ci ait libéré cette ressource [CDKM00].*

Certaines tâches, après leur élection (début de leur exécution effective), ne doivent pas être suspendues jusqu'à la fin de leur exécution : on parle alors de tâches non-préemptibles. A contrario, lorsqu'une tâche peut être suspendue au cours de son exécution et retourner dans l'état *Prête* (cf. figure 2.2), dans le but d'allouer le processeur à une autre tâche, on parle de tâche préemptible.

Les opérations d'élection et de suspension des tâches sont réalisées par un *ordonnanceur* et dépendent de multiples caractéristiques des tâches. L'objectif de la partie suivante est d'introduire ces caractéristiques. Nous proposons, ensuite, les définitions relatives aux ordonnanceurs et à l'ordonnancement.

Caractéristiques des tâches

Les tâches sont caractérisées par, entre autres, leur priorité, leur temps d'exécution, leur temps de réponse, leur échéance et leur gigue.

Les opérations d'élection et de suspension des tâches se basent sur un mécanisme de priorités. Ce mécanisme permet de déterminer, à tout instant, quelle tâche doit avoir accès à la ressource processeur.

Définition 2.12 ***(Priorité).*** *La priorité d'une tâche détermine son ordre d'importance pour l'accès à la ressource processeur [AB90].*

Soit τ_1 et τ_2 deux tâches *prêtes* ainsi que μ_1 et μ_2 leurs priorités respectives, avec $\mu_1 > \mu_2$. La tâche élue est celle possédant la priorité la plus élevée, soit τ_1. Il existe divers algorithmes d'affectation de priorité. Nous en décrivons plusieurs dans la partie 2.3.

Le nombre d'instructions pouvant être exécutées par une tâche permet de définir son temps d'exécution appelé capacité.

Définition 2.13 ***(Capacité).*** *La capacité d'une tâche est son temps d'exécution pire-cas [Bar06].*

Le temps de réponse pris par une tâche pour exécuter l'ensemble de ses instructions ne correspond pas à sa capacité.

Définition 2.14 ***(Temps de réponse).*** *Le temps entre l'activation d'une tâche et sa terminaison est appelé temps de réponse [WEE+08].*

Dans le cas des STRECs, une tâche doit toujours avoir un temps de réponse inférieur à son échéance.

Définition 2.15 ***(Échéance).*** *L'échéance d'une tâche est son temps de réponse maximal autorisé [AB90].*

Enfin, lorsqu'une tâche est activée, elle peut être sujette à une gigue.

Définition 2.16 ***(Gigue).*** *Une tâche activée au plus tôt à un instant t, et subissant une gigue de G, peut être activée à tout instant de l'intervalle $[t; t + G]$ [AB90].*

La gigue peut être dûe à de multiples facteurs comme lors d'un changement de contexte quand on affecte une nouvelle tâche au processeur.

2.3 L'ordonnancement : définition et exemples d'algorithmes

Les tâches constituant les STRECs sont soumises à des contraintes temporelles : leur exécution est limitée dans le temps par une échéance. Afin que ces contraintes soient respectées, il est nécessaire d'organiser dans le temps, l'affectation des ressources aux tâches constituant un STREC. Organiser ces affectations, c'est définir l'ordre d'exécution des tâches. C'est la responsabilité de l'ordonnanceur. L'objectif de cette partie est de décrire ce qu'est l'ordonnancement et les algorithmes d'ordonnancement les plus connus.

Définition 2.17 ***(Ordonnancement).*** *L'ordonnancement est une méthode d'affectation des ressources aux tâches, et plus particulièrement de la ressource processeur. Un ordonnancement a pour objectif de respecter les contraintes temporelles de toutes les tâches. Un ordonnancement est réalisé selon une algorithme d'ordonnancement [AB90, SAa+, Riv98].*

Définition 2.18 ***(Algorithme d'ordonnancement).*** *Un algorithme d'ordonnancement affecte des tâches au processeur et fournit une liste ordonnée de tâches appelée séquence d'ordonnancement.*

Il existe de nombreuses façons de classifier les algorithmes d'ordonnancement [SK93, Rot94,CK88,CFH+04,MW11]. Dans la suite de cette partie, nous présentons ce que sont les algorithmes d'ordonnancement hors-ligne, en-ligne, à priorités fixes, à priorités dynamiques, préemptifs et non-préemptifs.

La différence entre un ordonnancement hors-ligne et un ordonnancement en-ligne réside dans le fait que l'ordonnancement des tâches est calculé a priori, ou lors de l'exécution.

Définition 2.19 ***(Ordonnancement hors-ligne).*** *Ordonnancer hors-ligne, un jeu de tâches Γ, consiste à calculer une séquence complète d'ordonnancement des tâches de Γ, en tenant compte de ses paramètres. L'ordonnancement est connu avant l'exécution des tâches [CDKM00].*

Cette approche statique est rigide et suppose que tous les paramètres impactant l'exécution, comme les instants d'activation des tâches par exemple, soient fixes, limitant ainsi l'adaptabilité du système aux variations de l'environnement avec lequel il interagit.

Définition 2.20 ***(Ordonnancement en-ligne).*** *L'ordonnancement en-ligne permet de choisir, à tout instant, la prochaine tâche à exécuter en se basant sur les paramètres courants des tâches actives. Il peut alors choisir de changer la tâche s'exécutant en fonction du déroulement de l'exécution, sans connaître l'instant de ce changement a priori [CDKM00].*

Ce type d'ordonnancement supporte des activations de tâches imprévisibles (c'est le cas des tâches apériodiques par exemple), ce que ne permet pas l'ordonnancement statique. Un algorithme d'ordonnancement en-ligne suppose que chaque tâche se voie assigner une priorité, dérivée de ses paramètres temporels. Les priorités peuvent être fixes ou dynamiques.

Définition 2.21 ***(Ordonnancement à priorités fixes).*** *Une valeur de priorité constante est affectée à chaque tâche à ordonnancer. À tout instant, la tâche élue est toujours, parmi les tâches prêtes, celle avec la valeur de priorité la plus importante [CDKM00].*

Il existe de multiples algorithmes d'affectation de priorités fixes aux tâches [HKL94] dont, entre autres :

1. Rate Monotonic (RM) : dans le cas d'un jeu de tâches périodiques, les priorités sont assignées de façon inversement proportionnelle à leur période. Les tâches avec les périodes les plus courtes se verront affecter les priorités les plus hautes.
2. Deadline Monotonic (DM) : les priorité sont assignées de façon inversement proportionnelle à leur échéance. Les tâches avec les échéances les plus courtes se verront affecter les priorités les plus hautes.

Définition 2.22 ***(Ordonnancement à priorités dynamiques).*** *La valeur de la priorité des tâches évolue dynamiquement au cours de l'exécution du STREC [CDKM00].*

Les algorithmes d'ordonnancement à priorités dynamiques les plus courants sont [ZRS87] :

1. Earliest Deadline First (EDF) : cet algorithme assigne les priorités en fonction de leur échéance. La tâche dont l'échéance arrive à son terme le plus tôt se verra affecter la priorité la plus haute.
2. Least Laxity First (LLF) : cet algorithme assigne les priorités selon leur laxité. La laxité d'une tâche est le résultat de la différence entre son échéance relative et le temps d'exécution restant sur sa période courante.

Définition 2.23 ***(Ordonnancement préemptif et non-préemptif).*** *Dans le cas de l'ordonnancement préemptif, une tâche s'exécutant peut être suspendue afin que le processeur soit alloué à une tâche plus prioritaire. A contrario, lors d'un ordonnancement non-préemptif, toute tâche ayant commencé l'exécution d'un travail ne peut pas être suspendue avant la terminaison de ce dernier [CDKM00, AB90].*

Lorsqu'une préemption intervient, la tâche suspendue retourne dans l'état *Prête* (cf. figure 2.2) dans l'attente de la reprise de son exécution. Ce type d'ordonnancement n'est utilisable qu'avec des tâches préemptibles.

L'objectif d'un algorithme d'ordonnancement est garantir le respect des contraintes temporelles des tâches. Cela n'est pas faisable pour tout jeu de tâches. Des méthodes d'analyse permettent de déterminer si un jeu de tâches peut être ordonnancé de façon à respecter les contraintes temporelles. La partie suivante est dédiée à la présentation de ces méthodes.

2.4 Analyse de l'ordonnancement d'un STREC

Dans cette partie, nous fournissons les concepts et définitions nécessaires à la compréhension de l'analyse d'ordonnancement d'un STREC.

2.4.1 Faisabilité et ordonnançabilité

L'objectif de toute analyse d'ordonnancement est de déterminer si un ordonnancement est faisable, étant donné un jeu de tâches et un algorithme d'ordonnancement.

Définition 2.24 ***(Faisabilité).*** *Un jeu de tâches est dit faisable, s'il existe un algorithme d'ordonnancement pouvant ordonnancer toute séquence de travaux générable pour le jeu de tâches sans manquer aucune échéance [DB11].*

Définition 2.25 ***(Ordonnançabilité).*** *Une tâche est dite ordonnançable selon un algorithme d'ordonnancement si le temps de réponse pire-cas de cette dernière est inférieur ou égal à son échéance. Un jeu de tâches est dit ordonnançable selon un algorithme d'ordonnancement si toutes les tâches le constituant sont ordonnançables [DB11].*

Comme nous le mentionnons dans la partie 1.1, il existe diverses façons d'analyser la faisabilité de l'ordonnancement d'un jeu de tâches : la simulation, le *model-checking* et la théorie de l'ordonnancement temps-réel. Nous nous limitons, ici, à présenter la théorie de l'ordonnancement. La théorie de l'ordonnancement est constituée d'un ensemble de méthodes d'analyse appelées tests de faisabilité. Chaque test de faisabilité évalue un critère de performance. Dans la suite de cette partie, nous introduisons chacun de ces trois concepts.

2.4.2 La théorie de l'ordonnancement

La théorie de l'ordonnancement temps-réel propose des méthodes analytiques et des algorithmes qui peuvent être employés lors de simulations, et qui permettent la validation du comportement temporel d'un système temps-réel embarqué critique.

Définition 2.26 ***(Test de faisabilité).*** *Un test de faisabilité est une méthode d'analyse qui permet d'effectuer une vérification de l'ordonnançabilité d'un jeu de tâches par un ordonnanceur donné [DB11, Bar03].*

Un test de faisabilité consiste en un ensemble d'équations et d'algorithmes évaluant un critère de performance. Ces équations manipulent des informations sur le système analysé. Ces informations sont capturées par un modèle de tâche. Afin d'être applicable, un test de faisabilité suppose que l'architecture qu'il analyse soit conforme à un ensemble d'hypothèses. Un test de faisabilité peut être modélisé de la façon suivante :

1. par une condition d'ordonnançabilité,
2. par un modèle de tâche,
3. par le critère de performance évalué,
4. par un ensemble d'hypothèses.

2.4.3 Condition d'ordonnançabilité

Dans cette partie, nous définissons ce que nous appelons une condition d'ordonnançabilité et ses caractéristiques. Une condition d'ordonnançabilité est définie par un test de faisabilité. Si cette condition est respectée, le système est ordonnançable.

Sha et al. caractérisent les tests de faisabilité selon *un ensemble d'hypothèses* sur le modèle de calcul du système, un condition d'ordonnançabilité et sa précision [SAa$^+$]. La précision d'un test de faisabilité spécifie si la condition d'ordonnançabilité qu'il évalue est nécessaire et suffisante, suffisante ou nécessaire afin d'assurer la faisabilité d'un jeu de tâches :

(i) nécessaire et suffisante : un jeu de tâches est faisable si et seulement si il respecte cette condition. On parle alors de condition exacte ;

(ii) suffisante : tout jeu de tâches respectant cette condition est faisable, mais ceux ne la respectant pas peuvent tout de même être faisables. On parle alors de condition pessimiste ;

(iii) nécessaire : aucun des jeux de tâches ne respectant pas cette condition n'est faisable. Cela signifie que des jeux de tâches respectant cette condition peuvent ne pas être faisables quand même.

Un test de faisabilité prend en entrée un système décrit par un modèle appelé modèle de tâche.

2.4.4 Les modèles de tâche

Les équations et les algorithmes utilisés par les tests de faisabilité manipulent des informations sur les tâches du système à analyser. Ces informations sont contenues dans une représentation du système nommée modèle de tâche. Il existe de nombreux modèles de tâche manipulés par un ou plusieurs tests de faisabilité.

Un modèle de tâche a pour objectif de contenir les informations nécessaires à l'analyse d'un STREC, i.e., les caractéristiques des tâches ayant un impact sur leur ordonnancement. L'occurrence des travaux d'une tâche et le fait qu'elle soit préemptible ou non font partie de ces caractéristiques.

La figure 2.3 donne une représentation graphique des attributs du modèle de tâche périodique proposé par Liu et Layland [LL73a]. Une tâche périodique τ est définie par un

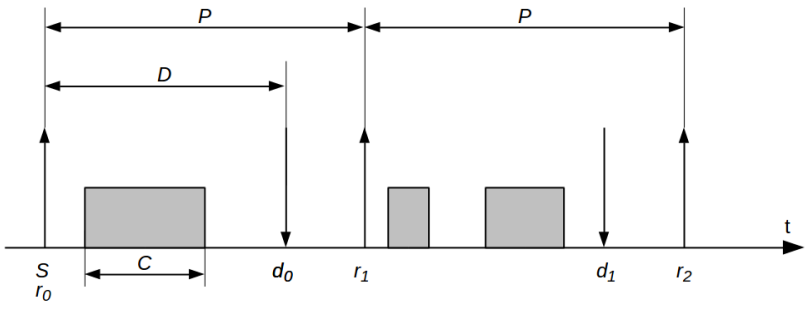

FIGURE 2.3 – Les paramètres d'une tâche périodique.

ensemble de paramètres dont les plus courants sont [CDKM00] :

— La date de première activation de la tâche, c'est-à-dire le premier instant où la tâche τ requiert le processeur, noté S.

— La capacité de la tâche, c'est la durée d'exécution maximale quand elle dispose du processeur, noté C. La capacité d'une tâche est généralement supposée constante. La capacité peut être obtenue par une analyse statique du code de la tâche ou par l'établissement de mesures sur la plate-forme d'exécution.

— La période d'activation, ou P. P est un délai fixe entre deux réveils successifs de la tâche τ. Pour chaque réveil, la tâche τ doit exécuter un travail de C unités de temps.

— L'échéance D d'une tâche est le délai maximum autorisé pour son exécution. C'est la contrainte temporelle que la tâche doit respecter.
— La priorité de la tâche permet à l'ordonnanceur de choisir la prochaine tâche prête à exécuter.

Un test de faisabilité peut évaluer divers critères de performance.

2.4.5 Les critères de performance

Un test de faisabilité donné évalue, le plus souvent, un unique critère de performance. Selon les propriétés du système à analyser, les critères de performance à évaluer varient. Nous en présentons cinq :

(i) le facteur d'utilisation processeur,

(ii) le temps de réponse pire-cas,

(iii) le calcul d'ordonnancement sur l'intervalle de faisabilité,

(iv) l'analyse d'empreinte mémoire,

(v) la détection d'interblocage et d'inversion de priorité.

(i) Facteur d'utilisation processeur

Le facteur d'utilisation processeur détermine la quantité de temps de calcul, fournie par le processeur, qui est utilisée par le jeu de tâches.

Liu et Layland proposent un moyen de calculer le facteur d'utilisation processeur [LL73b] conduisant à l'inéquation 2.1 :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{2.1}$$

La somme des rapports de la capacité des tâches sur leur période doit être inférieure ou égale à 1. Dans le cas où cette somme est supérieure à 1, la charge de calcul est trop importante vis-à-vis de la capacité du processeur. Le système ne peut donc pas être ordonnancé de façon à respecter les contraintes temporelles des tâches.

L'application d'un test de faisabilité à un STREC requiert que ce dernier respecte un ensemble d'hypothèses. Liu et Layland supposent que le STREC est conforme aux hypothèses suivantes :

- toutes les tâches sont périodiques ;
- les travaux des tâches sont activés en début de période et doivent respecter une échéance égale à la période de leur tâche ;
- toutes les tâches sont indépendantes ;
- la capacité de chaque tâche est inférieure ou égale à sa période ;
- les tâches ne peuvent pas se suspendre volontairement ;
- les tâches sont préemptibles ;

- les coûts de préemption sont supposés nuls ;
- le système est déployé sur une architecture mono-processeur ;
- l'algorithme d'ordonnancement est *Earliest Deadline First* ;
- les tâches sont synchrones.

(ii) Calcul du temps de réponse pire-cas

Le principe des tests de faisabilité évaluant ce critère de performance est de calculer les temps de réponse pire-cas de chacune des tâches du système et de les comparer avec leurs échéances [JP86a]. Les temps de réponse des tâches dépendent, entre autres, des temps d'attente dûs à leurs accès aux données, et leurs méthodes calcul varient d'un système à l'autre [SRL90b, Bak91, GLDN01].

(iii) Calcul d'ordonnancement sur l'intervalle de faisabilité

La condition de ce type de test de faisabilité est de vérifier l'ordonnancement sur un intervalle de temps étant le plus court possible et permettant d'affirmer l'ordonnançabilité du système. Le principe consiste à calculer l'ordonnancement sur l'intervalle de faisabilité par une simulation. Il existe plusieurs méthodes de calcul d'intervalle de faisabilité.

Par exemple, Cucu et Gossens ont prouvé, que pour un jeu de tâches périodiques à priorités fixes, synchrones, possédant des échéances égales ou inférieures à leurs périodes l'intervalle de faisabilité, est égal à l'hyper-période : l'intervalle minimal dont l'ordonnancement sera répété périodiquement [CG06]. L'hyper-période est de durée égale au plus grand multiple commun des périodes de toutes les tâches constituant le système.

(iv) Détection d'interblocage et d'inversion de priorité

Le partage de données entre les tâches d'un système peut mener à des phénomènes d'interblocage et d'inversion de priorité.

Définition 2.27 ***(Inversion de priorité).*** *On parle d'inversion de priorité lorsqu'une tâche est bloquée par une autre tâche de priorité inférieure [SRL90a].*

Définition 2.28 ***(Interblocage).*** *Un ensemble de tâches est en interblocage si et seulement si tout processus de l'ensemble est en attente d'un évènement qui ne peut être réalisé que par un autre processus de l'ensemble. [TT92].*

Ces deux phénomènes conduisent au non-respect d'échéances. Il est donc nécessaire de vérifier que les STRECs que l'on cherche à analyser ne peuvent pas être dans de telles situations [BW95, TT92].

(v) Analyse d'empreinte mémoire

L'analyse d'empreinte mémoire consiste à vérifier la cohérence entre la production et la consommation de messages lorsque les tâches communiquent à l'aide d'un tampon de communication. Cette analyse permet de calculer le temps d'attente moyen et maximum d'un message. Ces temps d'attente sont utiles au calcul de temps de réponse de bout en bout, ainsi que pour la méthode holistique [TC94, GSYY09, LSNM04a].

Comme nous l'exposons dans cette partie, de nombreux tests de faisabilité permettent l'analyse du comportement temporel des STRECs. Ces tests permettent de réduire les coûts de production grâce à la détection des problèmes d'ordonnancement en amont du cycle de vérification des STRECs.

Les tests de faisabilité sont donc conçus pour être intégrés aux processus de développement de STRECs.

2.5 Ingénierie des modèles d'architectures

Les modèles d'architectures sont plus particulièrement spécifiés et affinés au cours des étapes de conception architecturale et de conception détaillée du cycle de vie. Le modèle de gestion de projet le plus utilisé dans l'industrie et en particulier pour les systèmes critiques est le cycle en V [TFR05]. Le cycle en V est un modèle de gestion de projet très mature. Le périmètre de chaque étape de ce processus est clairement défini et la transition à une étape suivante ne s'éffectue pas tant que l'étape en cours n'est pas considérée comme stable et valide. La figure 2.4 présente le schéma classique des étapes constituant ce processus.

Le cycle en V est globalement constitué de deux phases : une descendante et une ascendante. Chaque étape de la phase descendante est couplée avec une étape de la phase ascendante. Les étapes de la phase ascendante ont pour objectif de valider l'étape de la phase descendante avec laquelle elle est couplée. Par exemple, l'étape de *Tests Unitaires* a pour objectif de valider l'étape de *Conception détaillée*. Les étapes du cycle en V sont les suivantes :

1. La phase descendante :
   - Analyse des besoins et faisabilité : le client décrit ses besoins et les usages correspondant au système tel qu'il peut l'imaginer.
   - Analyse des exigences et spécification fonctionnelle : la spécification fonctionnelle décrit les processus métiers dans lesquels le système devra intervenir. Les fonctions, à implanter, prises en charge par le système et son interaction avec les autres intervenants (utilisateurs et autres systèmes). Les exigences spécifiées ici peuvent être fonctionnelles ou non-fonctionnelles. De façon parallèle à cette étape, les tests de validation peuvent être définis.
   - Conception architecturale : la phase de conception architecturale consiste en la conception de l'organisation des différents éléments du système (logiciels, matériels et/ou humains) et des relations entre ces éléments. La conception est faite

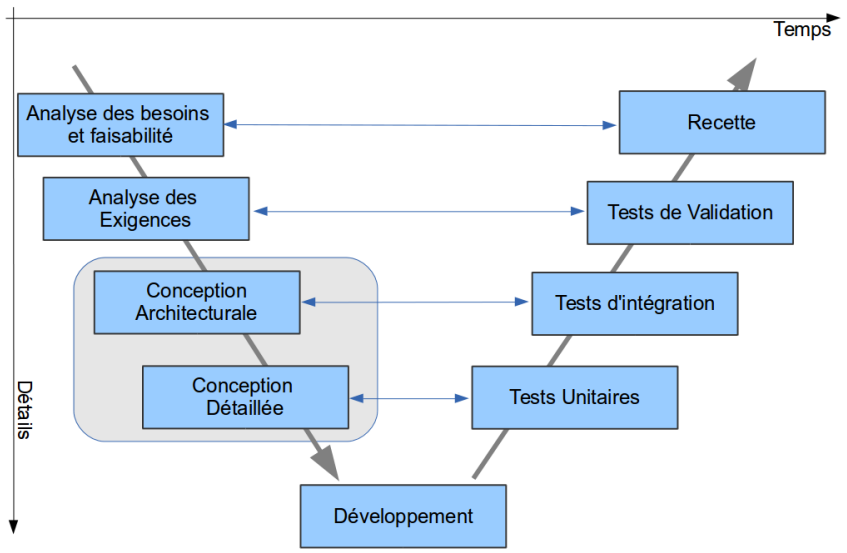

FIGURE 2.4 – Représentation classique du cycle en V.

de façon à remplir les exigences spécifiées lors de l'étape précédente. Les tests d'intégration peuvent être spécifiés parallèlement à cette étape.

— Conception détaillée : la conception détaillée consiste en la spécification des choix d'implémentation des éléments identifiés au niveau architectural. Les tests à développer en parallèle de cette étape sont les tests unitaires.

— Développement : le développement est la phase de réalisation effective du code implémentant le système. Cette étape est la dernière avant le début des tests évaluant la correction du système en cours de réalisation.

2. La phase montante :
   — Tests unitaires : les tests unitaires consistent en la vérification des fonctionnalités de chacun des composants de façon indépendante.
   — Tests d'intégration : les test d'intégration consistent en la vérification des fonctionnalités de l'architecture dans son ensemble.
   — Tests de validation : cette phase de tests a pour objectif d'assurer formellement que le produit est conforme aux spécifications.
   — Recette : cette dernière phase de tests se déroule en deux étapes, une première chez le fournisseur et une seconde chez le client. À l'issue de cette étape, le système est livré au client.

La qualité des modèles d'architecture est un point crucial et les méthodes d'analyse étudiées dans cette thèse concernent tout particulièrement les étapes de *Conception Ar-*

chitecturale et de *Conception détaillée* [HB14, Boe81, Fei04]. On parle de vérification et de validation précoce car l'idée est de minimiser les erreurs détectées lors des étapes suivantes, de celle de *Développement* à celle de *Recette*.

Les techniques génératives de l'IDM peuvent être exploitées non seulement pour produire une partie du code du système (transition de l'étape de *Conception détaillée* à celle de *Développement*) mais aussi pour produire des artefacts utiles aux étapes de la phase ascendante. La qualité des artefacts produits dépend donc en grande partie de celle des modèles d'architecture.

2.6 Langages de description d'architectures

La conception d'un système temps réel se base en premier lieu sur sa description. Cette description est double. Elle comprend tout d'abord celle de ses composants logiciels, des ressources matériels supportant l'exécution des composants logiciels et celle des relations et interactions entre composants. Elle décrit aussi comment les ressources matérielles du système sont allouées. On parle de déploiement des composants logiciels sur les composants matériels. L'ensemble de ces descriptions constitue l'architecture du système.

Définition 2.29 ***(Architecture).*** *L'architecture d'un système modélise les parties invariantes d'un système. Elle sert de point d'appui lors du développement d'un système pour la représentation de ce dernier. Une architecture est constituée de composants et de leurs interactions [Kro09].*

Pour décrire une architecture, le concepteur dispose de langages de description d'architecture.

Définition 2.30 ***(Les langages de description d'architecture).*** *Les langages de description d'architecture fournissent une abstraction de composants, de leurs connections et de leurs déploiements [Med09].*

A l'aide d'un langage de description d'architecture, pour un système à mettre en œuvre, le concepteur développe un modèle d'architecture. Outre l'aspect documentation précise du système, l'intérêt est double puisqu'à partir d'un modèle d'architecture, il est possible (1) d'effectuer des vérifications précoces sur la description des composants et sur les relations entre composants mais aussi (2) de raisonner sur le déploiement envisagé avant la mise en œuvre réelle [Gol13b]. Une partie importante des vérifications précoces concerne notamment l'analyse d'ordonnançabilité.

Les concepteurs utilisant des langages de description d'architectures exploitent leurs modèles d'architecture au travers d'outils. De façon très générales, ces outils assistent les concepteurs pour la vérification précoce des systèmes (avant leur mise en œuvre concrète) et la production automatique ou semi-automatique du code des systèmes. Les techniques appliquées sont celles de l'ingénierie dirigée par les modèles.

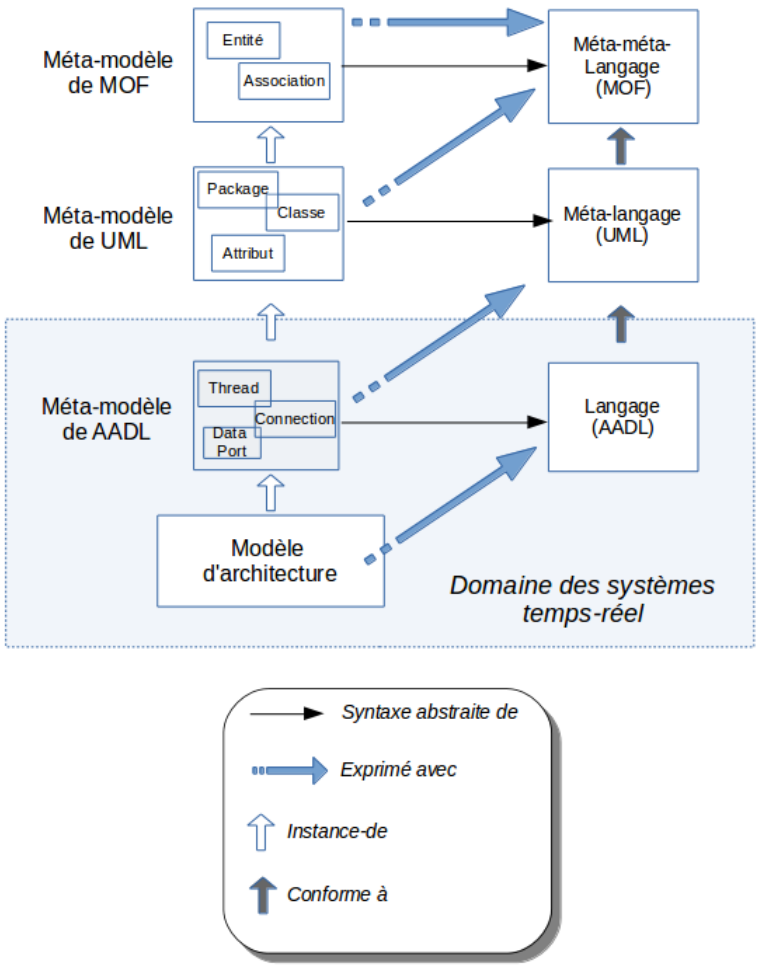

FIGURE 2.5 – Représentation des niveaux de modélisation présents dans l'ingénierie dirigée par les modèles (inspiré de [Pla]).

Définition 2.31 ***(Ingénierie dirigée par les modèles).*** *L'ingénierie dirigée par les modèles (IDM) est une méthodologie de développement de systèmes basé sur le concept de modèle. Chaque étape consomme et traite un ou des modèles sources et produit un ou des modèles résultats. Un système est comme un modèle, ainsi obtenu par étapes successives automatiques ou semi-automatiques [Ken02].*

La figure 2.5 présente brièvement les principaux concepts de l'IDM utiles à la suite de ce rapport. Cette figure montre qu'un modèle d'architecture est décrit avec un langage de description d'architectures, par exemple AADL. La langage AADL est décrit en partie par sa syntaxe abstraite nommé le méta-modèle de AADL. Dans l'IDM, ces deux relations de description sont récurrentes. Le méta-modèle d'AADL et le méta-modèle d'UML sont eux même décrits à l'aide d'un autre langage (par exemple UML et MOF, respectivement). Le dernier niveau est lui auto-descriptif.

Les outils peuvent aussi être combinés pour constituer des chaînes d'outils. Un apport très important de l'IDM est la normalisation des langages de description des modèles. Cette normalisation facilite l'échange et le partage des modèles et facilite donc l'intéropérabilité entre outils. Dans le contexte d'une chaine d'outils dédiée aux architectures temps-réel, un ADL peut ainsi servir de langage pivot définissant non seulement la syntaxe concrète et la syntaxe abstraite mais aussi la sémantique des modèles partagés par les différents outils.

De nombreux langages de description d'architecture (ADLs) sont actuellement disponibles. Parmis les langages de description d'architectures existants, on distingue les langages généralistes des langages plus spécifiques, dédiés à une utilisation bien particulière. Actuellement, deux langages de description d'architectures généralistes sont majoritairement utilisés : AADL [SAE09] et Marte [Obj05]. Le langage Cheddar [FT13] constitue un exemple représentatif de langage spécifique puisqu'il est plus particulièrement dédié à l'analyse d'ordonnancement. La partie suivante présente ces trois langages.

2.6.1 Trois approches de modélisation : AADL, MARTE et Cheddar ADL

Dans la suite de cette partie, nous présentons trois ADLs conçus selon des approches différentes. Nous identifions deux catégories d'ADLs : les ADLs pivots et les ADLs spécifiques à une utilisation donnée.

Les ADLs pivots

AADL et MARTE entrent dans la catégorie des ADLs pivots. Ce sont des langages riches, dont l'objectif est de couvrir une grande variété d'utilisations. Ils permettent donc la spécification d'informations très diverses et pouvant être dé-corrélées.

En effet, l'objectif de ces langages est la modélisation d'un système tout au long du processus de développement. Pour ce faire, il faut modéliser un même système à différents niveaux d'abstraction.

Les ADLs spécifiques

Cheddar ADL est un langage de description d'architecture spécifique à Cheddar [FT13], un logiciel libre d'analyse d'ordonnançabilité [SPDL09]. L'unique objectif de Cheddar ADL est de modéliser des STRECs de façon à pouvoir y appliquer des tests de faisabilité et réaliser des simulations.

De ce fait, les informations spécifiables avec cet ADL sont celles contenues dans les modèles de tâches des tests de faisabilité proposés par Cheddar. L'objectif de ce langage est de rester le plus proche possible des concepts capturés par la théorie de l'ordonnancement.

Il ne partage donc pas l'objectif de couvrir plusieurs étapes du processus de développement et ne propose qu'un seul niveau d'abstraction : celui des modèles de tâches de la théorie de l'ordonnancement.

Nous allons maintenant présenter successivement ces trois ADLs.

2.6.2 Architecture Analysis & Design Language (AADL)

AADL est un ADL supportant la conception, l'analyse et l'intégration de systèmes temps-réel distribués [SAE09]. Ce langage permet la spécification de systèmes comme un ensemble de composants logiciels et matériels. AADL intègre la modélisation d'interfaces de composants fonctionnels inter-connectés avec des aspects non-fonctionnels tels que le temps de réponse, la sûreté et des propriétés de certification, entre autres. Ces interfaces sont ensuite raffinées par des implémentations de composants.

Un modèle AADL est constitué de plusieurs catégories de composants logiciels et matériels. (1) Les catégories de *composants logiciels* sont *thread*, *process* et *subprogram*. Un *thread* modélise une unité d'exécution concurrente ordonnançable. Un *process* représente un espace d'adressage virtuel. Un *subprogram* modélise un morceau de programme s'exécutant séquentiellement. (2) Les catégories de *composants matériels* modélisent la plate-forme d'exécution. AADL en propose quatre : *processor*, *memory*, *device* et *bus*. Un composant *processor* modélise un processeur. Un composant *memory* abstrait des espaces mémoire permettant de stocker des données et du code et un composant *bus* représente les périphériques physiques de communication entre les divers composants matériels. Les composants AADL peuvent être définis en librairies ou organisés hiérarchiquement au sein des composants de la catégorie *system*. Un composant *system* modélise également le déploiement des composants logiciels sur les composants matériels.

Les composants AADL interagissent au travers de *connections*. Les *connections* modélisent des flots de contrôle et des flots de données entre les composants AADL. Ces *connections* peuvent modéliser une grande variété de flots grâce à différents protocoles de communication et de synchronisation tels que l'échange de messages, avec ou sans file d'attente, le rendez-vous ou encore l'accès à une donnée partagée.

Tout composant AADL peut être enrichi à l'aide de propriétés. Une propriété explicite de nombreux concepts tels que le protocole d'activation d'un *thread*, un fichier contenant le code source d'un *subprogram*, ou la bande passante d'un *bus*. Le standard AADL inclut un large ensemble de propriétés définies au sein d'ensembles de propriétés standards. Il est

```
package Example                                                                    1
public                                                                             2
system S end S;                                                                    3
system implementation S.i                                                          4
subcomponents                                                                      5
   A1 : process A.i;                                                               6
   P1 : processor P.i;                                                             7
properties                                                                         8
   Actual_Processor_Binding => (reference(P1)) applies to A1;                      9
   Scheduling_Protocol => (EARLIEST_DEADLINE_FIRST_PROTOCOL) applies to P1;         10
end S.i;                                                                           11
                                                                                   12
processor P end P;                                                                 13
processor implementation P.i end P.i;                                              14
                                                                                   15
process A end A;                                                                   16
process implementation A.i                                                         17
subcomponents                                                                      18
   T1 : thread T.i;                                                                19
   T2 : thread TT.i;                                                               20
end A.i;                                                                           21
                                                                                   22
thread T                                                                           23
properties                                                                         24
   Dispatch_Protocol => Periodic;                                                  25
   Period => 10 ms;                                                                26
   Compute_Execution_Time => 5 ms..5 ms;                                           27
end T;                                                                             28
thread implementation T.i end T.i;                                                 29
                                                                                   30
thread TT                                                                          31
properties                                                                         32
   Dispatch_Protocol => Hybrid;                                                    33
   Period => 25 ms;                                                                34
   Compute_Execution_Time => 3 ms..4 ms;                                           35
end TT;                                                                            36
thread implementation TT.i end TT.i;                                               37
end Example;                                                                       38
```

FIGURE 2.6 – Modélisation d'un STREC avec AADL.

également possible de définir des propriétés jusque là inexistantes. Ces propriétés spécifiques à une application ou un système peuvent être requises par une méthode d'analyse donnée.

Un modèle AADL doit être conforme au langage AADL noyau. Ce langage noyau est étendu par un ensemble d'annexes. Une annexe peut étendre le langage noyau afin de modéliser des erreurs ou de raffiner le comportement des composants. Ce sont, respectivement, les annexes de modélisation des erreurs (*Error Modeling Annex*) et comportementale (*Behavior Annex*). Les éléments supplémentaires relatifs à ces deux annexes peuvent être ignorés par les outils s'ils n'utilisent pas les données proposées par ces extensions. Une annexe peut également définir un cadre d'utilisation spécifique du langage noyau. C'est le cas de l'annexe de modélisation des données (*Data Modeling Annex*) et de l'annexe pour la

modélisation de systèmes conformes au standard pour l'avionique ARINC653 (*ARINC563 Systems Annex*).

La figure 2.6 propose un exemple simple de modèle AADL. Il représente un système constitué de deux *threads* indépendants $T1$ et $T2$. $T1$ est périodique, alors que $T2$ est hybride (activé périodiquement ou à l'aide d'un évènement). Ces deux *threads* sont déployés sur un environnement mono-processeur. Chacun des *threads* contient ses propres propriétés spécifiant :

— son protocole d'activation (*Dispatch_ Protocol*),
— sa période (*Period*),
— son temps d'exécution (*Computing_ Execution_ Time*).

L'algorithme d'ordonnancement est *Earliest_ Deadline_ First* (EDF). Il est spécifié à l'aide d'une propriété déclarée au sein du composant S.

2.6.3 Modeling and Analysis of Real-Time and Embedded systems (UML MARTE)

Dans la partie précédente, nous avons présenté un ADL conçu dans la perspective de l'analyse des STRECs.

Nous présentons, dans cette partie, un autre ADL, défini comme un profil UML (Unified Modelling Language) [Obj13]. UML MARTE est un profil UML standardisé et dédié à la modélisation et l'analyse de STRECs.

Un profil UML étend le langage UML pour un domaine spécifique. Un profil est défini par des stéréotypes, des valeurs étiquetées et des contraintes sur des entités d'UML.

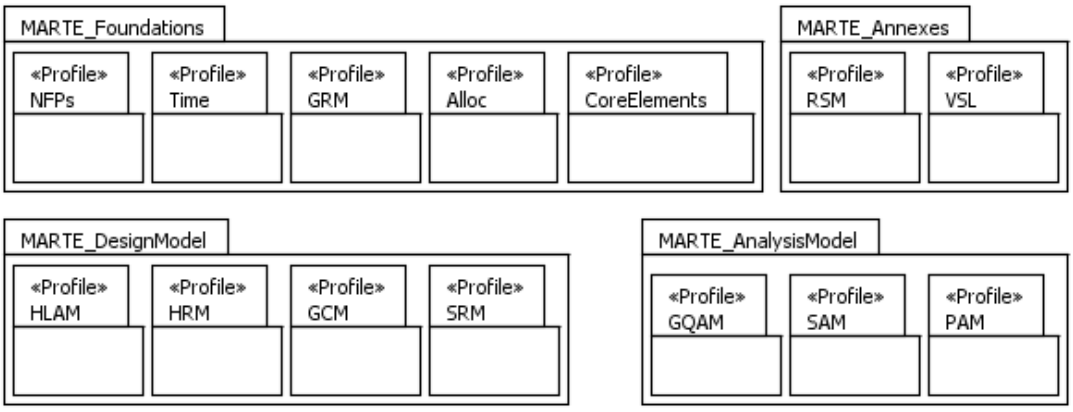

FIGURE 2.7 – Paquetages de MARTE.

UML MARTE permet à UML de supporter le développement dirigé par les modèles de STRECs. MARTE fournit les moyens de modélisation de tels systèmes et permet également d'annoter le modèle pour permettre différents types d'analyse. MARTE est basé sur UML version 2 [Obj11]. Nous considérons que les stéréotypes correspondent aux entités définies

par un ADL et que les valeurs étiquetées sont leurs attributs (ou propriétés dans le cas d'AADL).

Le profil MARTE est composé de plusieurs sous-profils pour représenter les concepts inhérents aux STRECs, représentés dans la figure 2.7. Entre autres, le sous-profil de modélisation des ressources matérielles (HRM) est utilisé afin de modéliser les entités matérielles et le modèle d'application de haut niveau (HLAM) qui modélise l'ensemble des entités logicielles.

Les sous-profils sont rassemblés au sein des quatre paquetages suivants :

— le paquetage *Foundations* contient les concepts fondamentaux des STRECs. Il est utile aux autres paquetages ;
— le paquetage *Design Model* étend les entités du paquetage *Foundations*. Il est dédié à la modélisation des aspects architecturaux des STRECs ;
— le paquetage *Analysis Model* étend également le paquetage *Foundations* et propose trois sous-profils dédiés à l'analyse ;
— le paquetage *Annexes* contient les entités permettant la spécification des aspects non-fonctionnels des STRECs.

2.6.4 Étude comparative de MARTE et AADL

AADL et MARTE ont été conçus selon deux philosophies différentes.

AADL propose une description de toutes les entités qu'il permet de définir. Cette description, fournie par le standard, établit une sémantique pour chaque entité, ainsi que des règles de validité. Un utilisateur ne pourra pas modifier les catégories, ni en créer de nouvelles. La stabilité d'AADL et sa sémantique précise permettent aux concepteurs d'outils de partager un savoir commun, ce qui est particulièrement utile pour la conception d'outils et leur interopérabilité. Néanmoins, il est possible d'étendre AADL grâce aux différentes annexes au standard et à la possibilité de définition de $Property_Set$ par les utilisateurs.

A contrario, MARTE constitue une base commune pour la définition de nouveaux langages de modélisation (les sous-profils). Un des objectifs de MARTE est la définition de nouveaux langages et outils. Il existe, pour ce faire, de nombreux outils de développement supportant UML. MARTE est donc flexible et permet, à partir d'un ensemble de concepts de base relativement réduit (en comparaison avec AADL), la définition de langages et d'outils pour des STRECs particuliers.

2.6.5 Le langage Cheddar ADL

Le langage de description d'architecture Cheddar ADL [FT13] a été développé dans le cadre du projet Cheddar [SPDL09]. Ce langage, dont certains aspects sont inspirés d'AADL, permet la modélisation d'une variété importante de STRECs. Cheddar ADL définit les entités et concepts de base habituels de la théorie de l'ordonnancement temps-réel. Chaque entité possède un nom unique et des attributs. Avec Cheddar ADL, il existe deux types d'entités : les composants matériels et les composants logiciels. Les composants maté-

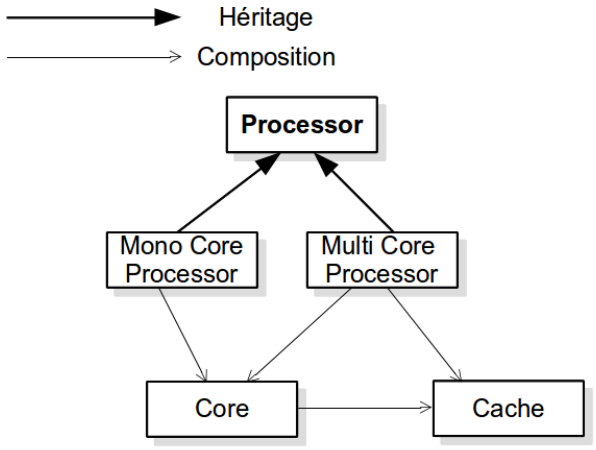

FIGURE 2.8 – Diagramme de classes UML des entités matérielles de Cheddar ADL.

riels représentent les ressources fournies par l'environnement. Les composants logiciels sont déployés sur les composants matériels. Voici quelques exemples de composants matériels :

Les composants matériels

Cheddar ADL identifie les entités matérielles suivantes : *Core*, *Cache*, *Processor*, *Memory* et *Network*. Ces cinq entités permettent de présenter les composants matériels nécessaires à l'analyse d'ordonnancement. Seules les entités *Core* et *Processor* sont présentées ci-après. Une description complète de ces six entités matérielles est fournie dans [FT13].

La figure 2.8 propose une vue graphique des différentes entités matérielles de Cheddar ADL.

Entité 2.1 ***(Core).*** *Une entité Core est une cible de déploiement pour les entités logicielles. Elle modélise une entité fournissant une ressource capable d'exécuter séquentiellement un flot de contrôle. Une entité Core est l'abstraction d'un cœur de processeur.*

Cette entité est constituée, entre autres, de l'attribut *Scheduling*. Cet attribut contient les paramètres d'ordonnancement.

Entité 2.2 ***(Processor).*** *Une entité Processor est constituée d'un ensemble d'entités Cores et, de façon optionnelle, d'entités Caches. Cette entité comporte trois attributs : Network, Processor_Type et Migration_type.*

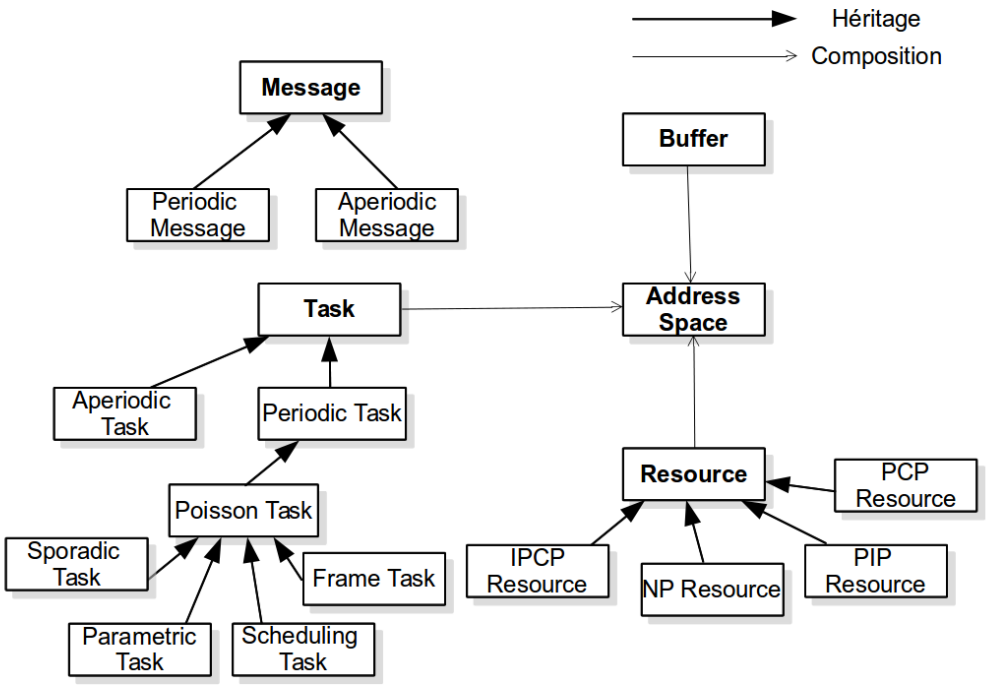

FIGURE 2.9 – Diagramme de classes UML des entités logicielles de Cheddar ADL.

Tout comme les entités *Cores*, les entités *Processors* sont également des cibles de déploiement pour les entités logicielles. L'attribut *Processor_Type* définit deux types de *Processor* :

— **Mono_Core_Processor** : Ce type de processeur n'exécute qu'un seul flot d'instructions à la fois. Il n'est alors composé que d'un cœur.

— **Multi_Cores_Processor** : Ce type de processeur référence plusieurs unités de calcul, modélisées par de multiples entités *Cores*. Il permet l'ordonnancement global de plusieurs flots de contrôle par un ensemble de cœurs.
L'attribut *Migration_Type* spécifie si les travaux des tâches peuvent migrer d'un cœur à un autre ou pas.

Dans la partie suivante, nous nous concentrons sur la description des entités logicielles du langage.

Les composants logiciels

Cheddar ADL identifie sept entités logicielles : *Address_ space*, *Task*, *Resource*, *Buffer*, *Message*, *Dependency*et *Task_Group*. L'entité *Task_Group* n'est pas présentée ici [LSRB14]. Une description complète de ces sept entités logicielles est fournie par le rapport technique suivant [FT13]. Les six autres entités sont présentées successivement ci-après.

La figure 2.9 propose une vue graphique des différentes entités logicielles de Cheddar ADL.

Entité 2.3 ***(Address_ space).*** *L'entité Address_ space représente les espaces d'adressage. Elle permet de modéliser des unités de mémoire physique ou logique. Elle définit un ensemble d'adresses mémoire.*

Une entité *Address_ space* peut correspondre à un périphérique, un secteur de disque ou mémoire et tout autre entité de mémoire physique ou logique. Elles peuvent être associées à un mécanisme de protection d'adressage.

Par exemple, cette entité peut représenter un espace d'adressage d'un système RTEMS [Ben11] ou d'une partition ARINC 653 [Ari97].

Chaque espace d'adressage est lié à un processeur à l'aide d'un attribut, i.e., un processeur accueille un ou plusieurs espaces d'adressage. Des attributs supplémentaires permettent de définir plus précisément leur sémantique.

Entité 2.4 ***(Task).*** *Une entité Task modélise un flot de contrôle. Ce dernier permet d'exécuter tout type de programme (incluant ainsi toute fonctionnalité système telle qu'un ordonnanceur).*

Une tâche est définie de façon statique dans une entité *Address_ space*.

Les attributs des tâches de Cheddar ADl sont ceux classiquement utilisés par la théorie de l'ordonnancement temps-réel : capacité, échéance, mais également la priorité qui permet à l'ordonnanceur de choisir quelle *Task* doit être exécutée, ou encore le temps d'attente dû à l'accès à une ressource partagée et les $offsets$ pour le cas où les tâches sont organisées en transactions [TC94].

Lorsque plusieurs tâches ont le même niveau de priorité, une *politique*, définissant l'algorithme d'ordonnancement, décrit la façon dont l'ordonnanceur choisit une entité *Task* à exécuter.

Enfin, Cheddar ADL définit divers types de tâches selon leur activation de ces dernières. Cette sémantique est spécifiée grâce à l'attribut *task_type* pouvant prendre les valeurs suivantes :

— Une tâche *apériodique* est activée une seule fois.

— Une tâche *périodique* est activée de multiples fois, et le délai entre deux activations est fixe (et est stocké dans l'attribut *period*).
Pour les tâches périodiques, une gigue peut être spécifiée par l'attribut *jitter*. La gigue est la latence maximale sur l'instant de réveil de la tâche. Cette dernière peut être utilisée pour exprimer des précédences entre tâches et pour l'application des méthodes holistiques de calcul de temps de réponse pire-cas [REW04].

— Une tâche *poisson* est également activée à plusieurs reprises avec un délai aléatoire entre deux activations successives. Une loi de poisson est utilisée pour générer ces délais.
— Une tâche *sporadique* est activée plusieurs fois, mais avec un délai minimal entre deux activations successives.

Les dépendances modélisant les interactions entre les tâches sont abstraites à l'aide de l'entité *Dependency*.

Entité 2.5 ***(Dependency).*** *Une entité Dependency permet de modéliser une interaction entre deux entités logicielles ayant un impact sur l'ordonnancement du système. Ces dernières peuvent être des tâches, des ressources ou des tampons.*

Une entité *Dependency* est caractérisée par le type de dépendance qu'elle modélise. Les types autorisés sont les suivants :

— *Precedence* lorsque l'entité *Dependency* modélise une relation de précédence entre deux *Task*,
— *Communication* modélise une dépendance entre une *Task* et un *Message*.
— $Time\_Triggered$ modélise une communication déclenchée de façon temporelle entre deux *Tasks*.
— *Resource* modélise la dépendance d'une tâche à une *Resource*.
— $Queuing\_Buffer$ lorsque la dépendance modélise une dépendance entre une tâche et un tampon de communication, avec un protocole de communication de type FIFO,
— $Black\_Board\_Buffer$ lorsque que la dépendance modélise une dépendance entre une *Task* et un *Buffer* avec un protocole de communication de de type *Blackboard*.

Entité 2.6 ***(Buffer).*** *Un Buffer modélise l'échange de données à l'aide d'un tampon, entre des Tasks d'un même Address_space. Il se définit de façon statique au sein d'un Address_space.*

La taille d'un *Buffer* est bornée. La sémantique de lecture et d'écriture d'une entité *Buffer*est spécifiée à l'aide des deux attributs suivants :

— $Queueing\_system$ décrit les occurrences des opérations de lecture et d'écriture et précise si une tâche peut être bloquée lors de ces opérations. Les valeurs possibles de l'attribut $Queueing\_System$ sont ceux décrits par les modèles de files d'attente [Kle76]. Les modèles classiques peuvent être référencés par cet attribut, e.g. les valeurs Qs_Mm1 ou Qs_Md1 spécifient que les lectures/écritures sur l'entité *Buffer* sont conformes aux modèles de files d'attente M/M/1 et M/D/1. Des modèles dédiés aux tâches périodiques peuvent également être utilisés, e.g. le modèle P/P/1 [LSNM04b].
— *Roles* décrit le comportement des tâches vis-à-vis du tampon. Une tâche peut produire ou consommer des données. On suppose que les producteurs et consommateurs écrivent et lisent chacun une quantité fixe de données dans le tampon. La taille de la donnée produite ou consommée est définie par cet attribut. Les instants auxquels

les opérations de lecture/écriture interviennent sont également spécifiés. Ces instants sont spécifiés relativement à la capacité de la tâche concernée Si cet attribut spécifie qu'une entité *Task* consomme une donnée à l'instant 2, cela signifie que la donnée sera retirée de la file d'attente lorsque s'exécute la seconde unité de temps de sa capacité.

Entité 2.7 ***(Resource).*** *Une entité Resource modélise une structure de données, partagée, ou non, par plusieurs tâches et pouvant être synchronisée ou pas.*

Elle se définit statiquement au sein d'une entité *Address_ space*. Une entité *Resource* peut être vue comme un sémaphore et possède une valeur initiale [Dij65].

Une *Resource* modélise une donnée partagée entre des tâches d'un même espace d'adressage.

Les principaux attributs caractérisant la sémantique d'une entité *Resource* sont *state*, *protocol* et *critical_sections* : *state* contient la valeur initiale de la donnée partagée (similaire à la valeur du compteur du sémaphore). Si sa valeur est inférieure ou égale à zéro, les tâches requérant l'accès à cette entité *Resource* sont bloquées jusqu'à sa libération.

protocol précise la façon dont les données sont réservées et libérées. Les protocoles pouvant être choisis avec Cheddar ADL sont : PCP (*Priority_Ceiling_Protocol*), PIP (*Priority_Inheritance_Protocol*), ICPP ou *FIFO* [SRL90b]. L'utilisation des protocoles PCP, PIP et ICPP implique qu'accéder à une entité *Resource* peut modifier les priorités des tâches impliquées.

Le dernier attribut *critical_sections*, spécifie quand chaque *Task* doit réserver ou libérer une ressources. Il explicite les sections critiques définies pour toute *Task* et toute *Resource*.

Entité 2.8 ***(Message).*** *Une entité Message modélise un échange de messages entre des tâches déployées au sein d'espaces d'adressage différents.*

Un *Message* peut être périodique ou apériodique. Un message périodique est envoyé lors de chaque travail, contrairement aux messages apériodiques pour lesquels il n'y a pas d'information sur l'instant d'envoi.

Un message est caractérisé par un *type* de la donnée transmise (*boolean*, *integer*, *double* ou *string*) et sa valeur *value*. Aux entités *Messages* sont aussi joints leurs temps de réponse (attribut *Response_Time*), qui représentent le délai de bout en bout entre l'émission et la réception du message. L'attribut *Communication_Time* exprime la durée pendant laquelle le message reste dans la file d'attente de communication.

2.7 Conclusion

L'objectif de ce chapitre a été d'introduire les notions et concepts nécessaires à la bonne compréhension de cette thèse. Nous avons introduit des concepts fondamentaux comme les systèmes temps-réel embarqués critiques, les éléments composant ces derniers (logiciels et

matériels), la théorie de l'ordonnancement temps-réel et les tests de faisabilité qui la composent, la notion d'architecture, de modèle et de méta-modèle, l'ingénierie dirigée par les modèles.

Nous avons ensuite décrit des langages de description d'architectures pouvant être utilisés pour la conception de STRECs : MARTE, AADL et Cheddar ADL. Les deux derniers ont été centraux lors de ces travaux de thèse et seront abordés à de nombreuses reprises dans ce manuscrit.

Toutes ces notions servent d'assise à la présentation des problématiques et contributions de cette thèse dans le chapitre suivant.

Chapitre 3

Survol de l'approche

Durant le processus de développement de systèmes embarqués temps-réel critiques (STRECs), un des objectifs des concepteurs est de pouvoir détecter, le plus tôt possible, les problèmes pouvant apparaître à l'exécution [Boe81]. Parmi ces problèmes, le non respect des contraintes temporelles du système tient une place importante et peut être, dans de nombreux cas, repéré dès les premières phases de conception grâce à la théorie de l'ordonnancement.

Comme nous le précisions dans le chapitre précédent, le modèle d'architecture doit respecter plusieurs hypothèses afin d'être analysable à l'aide de la théorie de l'ordonnancement. Le modèle doit non seulement contenir les informations nécessaires à l'analyse, mais également respecter des contraintes structurelles et sémantiques pour qu'une analyse soit applicable.

La sélection des tests de faisabilité adaptés à un STREC en cours de développement est une tâche ardue, ce qui peut expliquer qu'ils ne sont pas utilisés, alors que cela pourrait être profitable.

Pour pallier ce problème, nous proposons d'automatiser la sélection de tests de faisabilité à l'aide de patrons de conception architecturaux.

L'objectif de ce chapitre est de détailler les problématiques traitées dans cette thèse et nos propositions pour y répondre, ainsi que leur positionnement vis-à-vis des solutions existantes.

Ce chapitre se divise en trois parties. La partie 3.1 présente les problématiques de cette thèse. La partie 3.2 expose la solution que nous proposons. Enfin, un positionnement vis-à-vis des approches existantes est fourni dans la partie 3.4 avant de conclure en partie 3.5.

3.1 Le problème de l'analyse d'ordonnançabilité des STRECs

L'objectif de cette partie est de présenter les problématiques traitées par cette thèse.

3.1.1 Comment déterminer si un test de faisabilité est applicable à un STREC ?

Comme il l'a été établi précédemment, le comportement temporel des systèmes temps-réel embarqués critiques (STRECs) doit être analysé le plus tôt possible lors du processus de développement.

La théorie de l'ordonnancement permet de réaliser des analyses d'ordonnançabilité précoces sur des modèles d'architecture de STREC. Cependant, un test de faisabilité suppose que le modèle d'architecture qu'il analyse, respecte un ensemble d'hypothèses. Nous nommons ces hypothèses contraintes d'applicabilité [GSP+11b].

Définition 3.1 ***(Contrainte d'applicabilité).*** *Une contrainte d'applicabilité est une hypothèse faite par un test de faisabilité qu'un modèle architecture devra respecter pour qu'un test de faisabilité puisse s'appliquer [GSP+11b].*

Par exemple, le test de Liu et Layland présenté en partie 2.4.5 suppose le respect de dix contraintes d'applicabilité portant sur les aspects matériels et logiciels des STRECs. Lorsque ces contraintes ne sont pas respectées dans leur intégralité, l'utilisation du test peut mener à deux issues :

— il manque des informations requises par le test. L'application du test échoue et aucun résultat n'est produit ;

— le test ne prend pas en compte, ou interprète de façon erronée, des informations contenues dans le modèle d'architecture. Le résultat produit est faux. Un STREC pourra être évalué comme non-ordonançable alors qu'il l'est ; ou évalué comme ordonnançable alors qu'il ne l'est pas, ce qui n'est pas toléré pour le cas de systèmes temps-réel embarqués critiques.

La sélection de tests de faisabilité adaptés au STRECs que l'on souhaite analyser est donc une tâche cruciale.

3.1.2 Les approches de conception de tests de faisabilité

De nombreux tests de faisabilité ont été conçus au cours des quarante dernières années. Nous avons identifié trois approches pour la conception d'un nouveau test de faisabilité pour une architecture donnée.

La première approche consiste à créer, de toutes pièces, un test pour un système non-couvert jusqu'alors. Le test conçu par Liu et Layland (cf. partie 2.4.5) en est un bon exemple.

La seconde approche consiste à relâcher une partie des contraintes d'applicabilité d'un test existant, et donc adapter le test de faisabilité à de nouvelles architectures [SAa+]. On pourra citer les travaux de Leung et al. [LM80] qui proposent une adaptation du test de Liu et Layland permettant l'utilisation de tâches ayant une échéance inférieure à leur période. La surpression de contraintes d'applicabilité a souvent un prix en termes de complexité de calcul et de pessimisme du résultat.

La dernière approche est, pour un ensemble de contraintes d'applicabilité donné, d'améliorer les performances d'un test existant. Pour exemple, Pellizoni et al. [PL04] proposent une amélioration d'un test de faisabilité proposé par Ripoll et al. [RCM]. Cette amélioration consiste à réduire le pessimisme du test de Ripoll et al. grâce à la prise en compte des offsets. Les auteurs améliorent également la complexité de l'algorithme de test.

De ces trois approches de conception de tests de faisabilité découlent

3.1.3 Les difficultés à sélectionner les tests de faisabilité

Les choix de conception d'un déterminent si un modèle d'architecture de celui-ci respectera les contraintes d'applicabilité, déterminant ainsi quels tests de faisabilité sont applicables. Choisir ces tests de faisabilité, parmi le grand nombre de tests existant, est une tâche ardue, qui nécessite une expertise du domaine de la théorie de l'ordonnancement temps-réel [SPDL09].

Afin d'illustrer ce propos, nous avons décompté les travaux proposant des tests de faisabilité [SAa$^+$, DB11].

Dans [SAa$^+$], Sha et al. présentent un historique de la théorie de l'ordonnancement. Les auteurs parcourent les différents types de systèmes temps-réel embarqués et les tests de faisabilité ayant fait date. Plus de 200 articles proposant chacun au moins un test de faisabilité (et souvent plusieurs) sont cités.

Dans [DB11], Davis et al. dressent un état de l'art des tests de faisabilité dédiés aux systèmes multiprocesseurs. Les auteurs citent environ 120 travaux traitant de l'analyse de STRECs multiprocesseurs uniquement.

Être capable de trouver, parmi la multitude de tests de faisabilité existant, les tests de faisabilité applicables à un STREC en particulier nécessite donc une connaissance pointue de la théorie de l'ordonnancement.

Par ailleurs, supposons qu'un concepteur de STREC soit capable de sélectionner une première liste de tests de faisabilité par lui-même, à chaque modification de son architecture (utilisation d'un autre protocole de communication d'ordonnancement par exemple). il sera nécessaire de sélectionner de nouveaux tests.

Sélectionner des tests de faisabilité est donc une tâche complexe, qui doit, de surcroît, être réalisée plusieurs fois au fil des évolutions du système en cours de développement.

De plus, sélectionner tous les tests de faisabilité applicables à un système ne suffit pas. En effet, plusieurs tests peuvent être applicables à un même système. Dans ce cas, on ne souhaite généralement pas appliquer l'intégralité des tests sélectionnés et il sera nécessaire de choisir les tests à appliquer. L'application des tests doit donc être réalisée en fonction des objectifs du concepteur : faible coût de calcul, exactitude des résultats, pessimisme des résultats, etc.

3.1.4 Intégration dans le processus de développement

L'intégration de l'analyse d'ordonnancement dans le processus de développement peut aussi s'avérer compliquée. Par exemple, peu d'ingénieurs savent à quelles étapes de leur

processus de développement il est nécessaire d'effectuer ces vérifications.

En outre, le nombre d'outils disponibles implantant la théorie de l'ordonnancement. On pourra citer, entre autres, Rapid-TMA [RAP08], Timewiz [Tim02], STORM [UDT10], Cheddar [SPDL09] ou MAST [HGG+02]. Ces outils ne sont pas tous, de surcroît, librement accessibles (Rapid-TMA et Timewiz par exemple).

Par ailleurs, l'usage d'un outil d'analyse d'ordonnancement seul reste difficile pour un ingénieur s'il n'est pas intégré dans une chaîne d'outils de développement. En effet, les outils d'analyse possèdent souvent un ADL qui leur est propre. La transformation du modèle de l'architecte vers le modèle utilisé par l'outil implique une compréhension fine des concepts liés à l'ordonnancement et manipulés par l'outil. Par exemple, certains tests de faisabilité utilisent la gigue pour modéliser les changements de contexte ou les attentes diverses.

Il est donc nécessaire d'assister les concepteurs à l'aide d'outils de modélisation et en explicitant les relations entre les caractéristiques des modèles d'architectures et la théorie de l'ordonnancement.

3.2 Présentation de notre solution : les patrons de conception architecturaux

L'objectif de cette partie est de présenter la solution proposée par cette thèse. Dans un premier temps, nous exposons la façon dont nous souhaitons assister les concepteurs dans leur utilisation de la théorie de l'ordonnancement : utiliser des patrons de conception architecturaux pour automatiser la sélection de tests de faisabilité. Puis, nous exposons le modèle de patrons de conception architecturaux que nous définissons. Nous présentons, alors, notre approche du point de vue d'un concepteur de STRECs.

3.2.1 Assister les concepteurs au long du cycle de développement

Afin d'assister l'intégration de ces tests de faisabilité au sein du processus de développement, nous proposons une caractérisation des architectures par le biais de patrons de conception architecturaux qui permettent d'automatiser la sélection des tests de faisabilité [DS08, PSDL10]. Ces patrons de conception sont modélisés par des ensembles de contraintes d'applicabilité portant sur les propriétés des modèles d'architectures.

On distingue, pour chaque patron, deux types de contraintes distincts. Le premier comprend les contraintes portant sur la partie matérielle de l'architecture, i.e., l'environnement d'exécution. Le second type de contraintes porte sur la partie logicielle de l'architecture, et plus particulièrement sur les protocoles de communication et de synchronisation entre les tâches du système.

La figure 3.1 schématise l'utilisation des patrons de conception architecturaux pour la sélection automatique de tests de faisabilité. Nous proposons d'analyser les modèles d'architecture pour vérifier le respect de contraintes d'applicabilité : nous vérifions que les modèles d'architecture sont conformes aux patrons. Dans le cas où la conformité à un patron de conception est confirmée, nous sommes capables de sélectionner une liste de

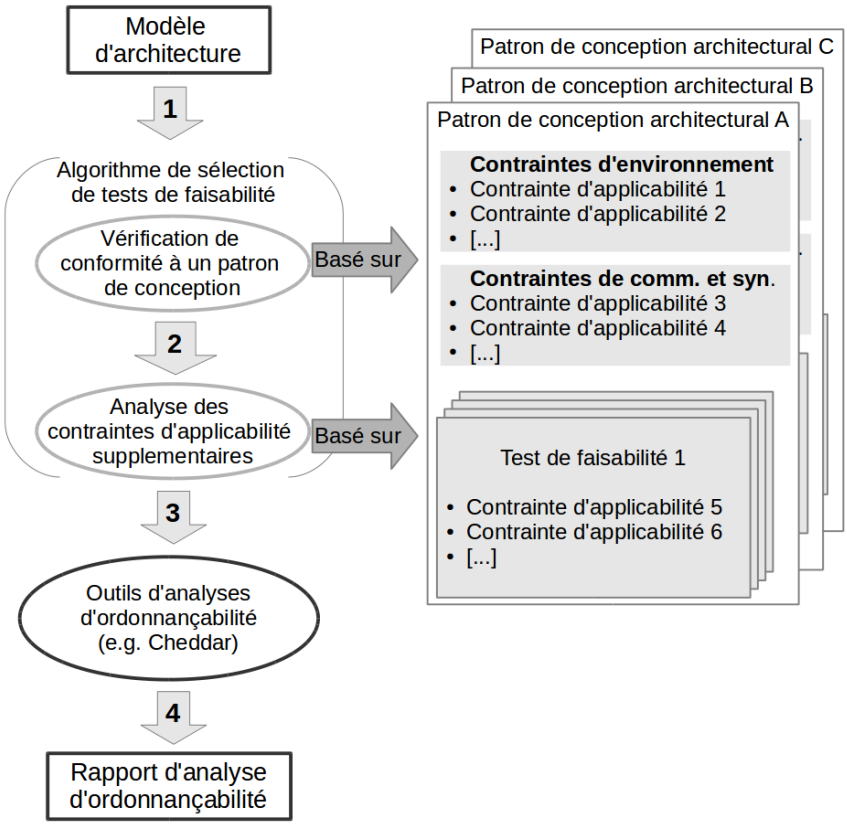

FIGURE 3.1 – Représentation graphique du processus de sélection de tests de faisabilité et de sa relation avec les patrons de conception architecturaux.

tests de faisabilité applicables. Cette liste est dépendante du patron de conception utilisé ainsi que des contraintes sur les propriétés de l'architecture. Les tests contenus dans la liste peuvent ensuite être appliqués à l'aide d'un outil d'analyse d'ordonnancement comme Cheddar.

3.2.2 Définition des patrons de conception architecturaux

Cette partie donne notre définition de patron de conception architectural.

Le *GOF*[1] définit un patron de conception comme *"une solution à un problème récurrent dans un certain contexte"* [GHJV95]. Alexander et al. le définissent comme *"une règle en trois parties qui exprime une relation entre un contexte, un problème et une solution"* [Ale79].

Dans le domaine des systèmes temps-réel, les auteurs de [Dou02] et [MKMG97] classifient les patrons de conception selon trois niveaux, correspondant chacun à une phase de conception : architecturale, mécanique et détaillée.

- **Conception architecturale :** décisions stratégiques de grande échelle, telles que la politique d'ordonnancement, les interactions inter-composants, le comportement global de l'application, etc.
- **Conception mécanique :** définition de groupes de composants à concevoir pour réaliser un mécanisme ou une fonctionnalité (un buffer de communication par exemple).
- **Conception détaillée :** considérations de bas niveau telles que le typage des données, l'algorithmique interne aux composants, etc.

Les patrons de conception mécaniques peuvent être vus comme de potentielles instanciations de solutions des patrons de conception architecturaux. De façon similaire, les patrons de conception détaillés peuvent faire office d'instanciations de solutions des patrons mécaniques.

Les patrons de conception que nous proposons font partie du niveau architectural. En effet, nos hypothèses ne portent pas sur la façon dont le système est implanté. Pour chacun de nos patrons, leur contexte est défini par le protocole de communication ou de synchronisation entre les tâches qui est utilisé. Le problème traité par chaque patron est l'analyse devant être effectuée dans le but de valider le comportement temporel du système traité. Enfin, la solution de chaque problème est un ensemble de contraintes sur l'architecture du système. Le respect de ces contraintes permet une sélection de tests de faisabilité appropriée à l'analyse du système à implanter.

Définition 3.2 ***(Patron de conception architectural).*** *Un patron de conception architectural explicite une relation entre un ensemble de modèles d'architecture et des tests de faisabilité leur étant applicables. Un patron de conception architectural est constitué d'un ensemble de contraintes d'applicabilité associé à une liste de tests de faisabilité.*

3.2.3 Spécification des patrons de conception architecturaux

Dans cette thèse, nous proposons huit patrons de conception architecturaux. Les cinq premiers sont spécifiques au cas mono-processeur sans ordonnancement hiérarchique, tandis que les trois autres sont dédiés aux systèmes mono-processeurs à ordonnancement hiérarchiques.

1. Gang of Four : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

Le cas mono-processeur

Singhoff et al. ont défini quatre patrons de conception [Sin08, DS08, PSDL10]. Ces patrons ont été définis pour le cas de STRECs mono-processeur sans ordonnancement hiérarchique. Chacun de ces patrons explicite une relation de correspondance entre un environnement d'exécution, un type de communication ou de synchronisation entre les tâches et une liste de tests de faisabilité.

Nous proposons de décrire les patrons sous la forme de deux ensembles de contraintes sur les entités définies par le langage de description d'architecture Cheddar ADL (cf. partie 2.6.5). Nous identifions deux types d'entités :

(a) Les entités matérielles : *processor*, *core*, *memory*, *network*, etc.

(b) Les entités logicielles : *task*, *dependency*, *address space*, *resource*, etc.

Les contraintes portant sur les entités matérielles constituent un ensemble de contraintes dites d'environnement. Les autres contraintes portant sur les entités logicielles forment un ensemble de contraintes dites de communication et de synchronisation.

Les ensembles de contraintes d'environnement décrivent les caractéristiques des environnements d'exécution pour lesquelles nous sommes capables de sélectionner des tests de faisabilité. Dans un premier temps, un seul ensemble de contraintes d'environnement est proposé. L'ensemble est nommé *uniprocessor* dans la suite du document. Il correspond à l'environnement d'exécution mono-processeur sans ordonnancement hiérarchique.

Nous avons défini cinq ensembles de contraintes de communication et de synchronisation : *Time-triggered*, *Ravenscar*, *Blackboard*, *Queued buffer* et *Unplugged*. Chacun de ces ensembles propose une solution architecturale à un problème de synchronisation entre les tâches en définissant un protocole de communication inter-tâches [DS08, PSDL10].

Un patron étant la combinaison d'ensembles de contraintes d'environnement ainsi que de communication et de synchronisation, nous proposons les cinq patrons suivants : *Time-triggered uniprocessor*, *Ravenscar uniprocessor*, *Blackboard uniprocessor*, *Queued buffer uniprocessor* et *Unplugged uniprocessor*.

Le cas mono-processeur à ordonnancement hiérarchique

Dans un second temps, nous avons étendu cette approche aux STRECs mono-processeur à ordonnancement hiérarchique. Nous proposons alors une nouvelle façon de généraliser la modélisation des patrons pour des architectures matérielles plus complexes.

Cette ouverture nous a menés à proposer un second modèle de patron sous la forme de graphes de contraintes.

Nous proposons alors trois nouveaux patrons de conception : le patron *ARINC653*, le patron *Two-levels-independent-applications* et le patron *N-levels-independent-workloads*. Cette contribution est présentée dans le chapitre 5.

3.2.4 Notre approche du point de vue du concepteur de STRECs

Nous exposons, ici, l'utilisation de notre approche du point de vue du concepteur de STRECs voulant vérifier de façon automatique l'ordonnançabilité du système en cours de

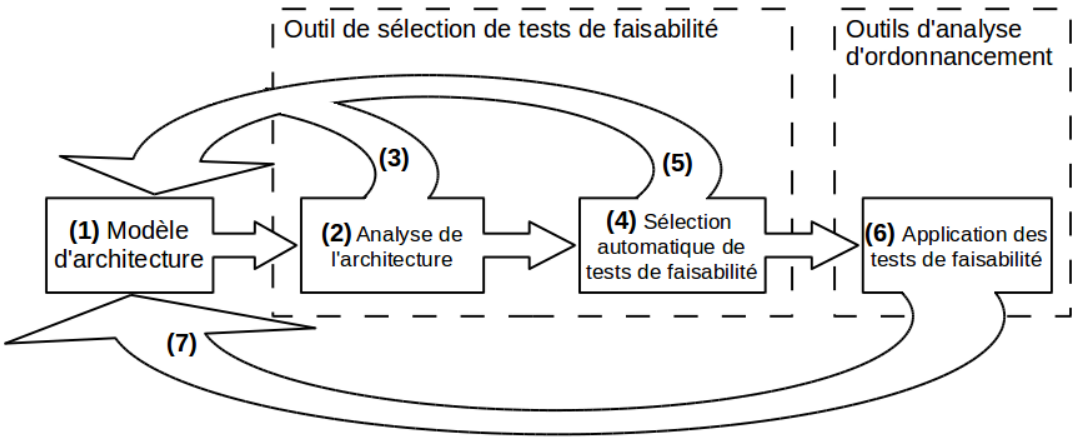

FIGURE 3.2 – La méthode de sélection d'une liste de tests de faisabilité du point de vue du concepteur.

développement.

La figure 3.2 décrit le cycle de conception de modèles d'architectures temps-réel incluant notre approche de sélection de tests :

— le concepteur propose un modèle d'architecture (1) ;
— l'outil de sélection analyse le modèle d'architecture et vérifie sa conformité à un patron (2). Nous présentons l'algorithme de reconnaissance de patrons de conceptions dans la partie 4.4 ;
— l'outil propose alors une liste de tests de faisabilité spécifique au système (4), avant de les exécuter (6).

Lors de chacune des étapes 2, 4 et 6, le concepteur est susceptible de revenir sur la phase de conception de l'architecture. Dans un premier temps lors de la vérification de la conformité du modèle d'architecture à un patron, si le système n'est pas conforme à un patron, l'outil n'est donc pas en mesure de fournir une liste de tests de faisabilité, et présente les contraintes auxquelles le modèle d'architecture n'est pas conforme(3).

Dans un second temps, lors de l'étape (4), la liste de tests de faisabilité proposée peut ne satisfaire le besoin qu'a l'architecte, d'appliquer un test en particulier par exemple. L'outil propose alors un ensemble de tests de faisabilité différents et leurs contraintes d'applicabilité non-respectées (5).

Enfin, après l'analyse d'ordonnançabilité, dans le cas où le système n'est pas ordonnançable ou si les tests faisabilité ne sont pas en mesure de valider l'ordonnancement (7), les résultats de l'analyse sont retournés à l'utilisateur qui peut alors modifier son modèle d'architecture.

Comme nous venons de le montrer, outre le choix des tests de faisabilité à appliquer, il est utile de raisonner sur le modèle d'architecture afin de pouvoir fournir au concepteur des informations sur des modifications potentielles permettant une meilleure vérification de son modèle d'architecture. Un outil intégrant ces contrôles de conformité à un patron

de conception doit donc répondre à plusieurs questions.

— Le modèle d'architecture est-il conforme à un patron de conception ?
  - Si non, quelle est l'importance des modifications à apporter au modèle d'architecture pour l'être ?
  - Si oui, quelle est la liste des tests de faisabilité applicables ?
— Quelles sont les autres listes de tests de faisabilité non retenues et les modifications à apporter au modèle d'architecture pour pouvoir les appliquer ?
— Les tests de faisabilité sélectionnés suffisent-ils à prouver l'ordonnançabilité ?
— Le STREC est-il ordonnançable ?

3.3 Hypothèses de travail

L'ensemble de tests de faisabilité étant très important, nous avons restreint notre approche à une quantité limitée de STRECs et de tests de faisabilité.

Lors de la présentation de nos patrons de conception architecturaux, nous prenons comme base de travail un ensemble de tests de faisabilité conçus pour un type d'environnement d'exécution donné.

Deux ensembles de tests de faisabilité délimitent le cadre de nos travaux. Toutefois, notre approche peut être adaptée à des contextes où le nombre de tests de faisabilité est bien plus grand, comme c'est le cas pour les architectures multiprocesseurs [RFS+14].

Nous listons, ci-après, les tests de faisabilité considérés pour l'approche mono-processeur sans ordonnancement hiérarchique et auxquels nous nous limitons dans le cadre de cette thèse.

(i) Le facteur d'utilisation processeur : calcule le taux d'occupation processeur. Dans le cadre de cette thèse, nous considérons neuf tests de faisabilité, permettant l'analyse d'ordonnancement à priorités fixes [LL73b, Car96, SRL90b, LSD89], et à priorités dynamiques [LL73b, LW82, JSM91, BHR90, BMR90].

(ii) Le temps de réponse pire cas : calcule le temps de réponse dans le pire cas pour chaque tâche du jeu de tâches à ordonnancer. Dans ce mémoire, nous considérons cinq tests de faisabilité évaluant ce critère de performance [JP86b, ABR+93a, TBW94, G+96b, Spu96a]. De plus, dans le cas où les tâches accèdent à des ressources partagées, nous considérons trois méthodes de calcul des temps d'attente pire-cas [SRL90b, Bak91, GLDN01].

(iii) L'analyse d'empreinte mémoire : l'analyse d'empreinte mémoire consiste à vérifier la cohérence entre la production et la consommation de messages dans le cas de communication à l'aide d'un tampon de communication. Nous considérons trois tests de faisabilité réalisant ce type d'analyse [TC94, GSYY09, LSNM04a].

(iv) Intervalle de faisabilité : cette méthode consiste à vérifier l'ordonnancement sur un intervalle de temps afin d'affirmer l'ordonnançabilité du système. Nous considérons deux méthodes de calcul permettant de définir l'intervalle de faisabilité d'un système [CGG04] [RRC03].

Dans le cas des STRECs mono-processeur à ordonnancement hiérarchique, la quantité

de tests considérés est moindre. Elle est, cependant, représentative des méthodes d'analyse conçues pour de tels systèmes.

Les quatre tests de faisabilité considérés sont ceux proposés par Mok et al. [MFC01], Easwaran et al. [EAL07] [ELSV09] et Davis et al. [DB05].

Nous revenons plus en détails sur la description de ces tests de faisabilité dans le chapitre 5.

3.4 Positionnement vis-à-vis des solutions existantes

Il existe de multiples approches étudiant également l'application des méthodes d'analyse à des modèles d'architecture de STRECs.

Gilles *et al.* définissent un langage d'expression de contraintes pour AADL nommé REAL (Requirement Enforcement Analysis Language) [GH10]. Ce dernier est développé par Telecom-Paris-Tech et l'ISAE et a été adopté en tant qu'annexe au standard AADL. REAL permet la spécification de différents types de contraintes directement sur les modèles d'architecture AADL. Les auteurs ont montré que REAL peut être employé pour l'expression de contraintes d'applicabilités similaires à celles que nous souhaitons modéliser.

OCL (Object Constraint Language) permet également la spécification de contraintes similaires aux nôtres, mais pour des modèles UML [WK03].

Une autre approche de spécification comparable à nos patrons de conception est définie par la méthode HOOD et HRT-HOOD, dont l'objectif est la réalisation de systèmes conformes au profil Ada Ravenscar [BW95].

Panunzio *et al.* définissent un processus d'ingénierie basé sur le méta-modèle RCM (Ravenscar Computation Model) [PV07]. Les vérifications de performances sont réalisées avec MAST [HGG$^+$02]. Comme Cheddar, MAST implante des tests de faisabilité. Enfin, PPOOA propose une approche similaire à nos patrons de conception [FSMA09].

PPOOA est implémenté comme une extension d'UML et fournit un ensemble prédéfini de mécanisme de synchronisation. De plus, les auteurs soulignent l'importance de l'application des tests de faisabilité le plus tôt possible lors du processus de développement et utilisent Cheddar pour ce faire.

Les méthodes présentées précédemment étudient la validation d'un ensemble de STREC avec un nombre prédéfini de patrons de conception. Dans le cas de notre approche, nous nous efforçons à permettre à un utilisateur de spécifier lui même un patron à l'aide du modèle que nous proposons. En effet, nous montrons comment spécifier des patrons de conception permettant la sélection de tests de faisabilité parmi un ensemble de tests donné. L'algorithme de sélection de tests et sa mise en œuvre ont été conçus de façon à pouvoir être étendus à de nouveau patrons.

Notre approche peut être employée pour traiter le problème de l'interopérabilité entre les outils qui manipulent des ADLs communs. Ce problème a été le sujet de nombreuses études. On peut relever deux approches principales :

— les transformations de modèles entre des modèles spécifiques à leur outil et
— la définition de sous-ensembles des langages de description d'architecture ;

Diverses approches de transformation de modèles ont été étudiées. Garlan et al. [GMW00] proposent un langage de spécification de transformation entre des paires d'outils. Cette approche implique que la cohérence des données entre les paires d'outils soit assurée, ce qui peut être difficile à garantir lorsque les outils sont mis à jour. Malavolta et al. proposent DUALLY [MMPT10], un canevas pour des transformations de modèles sur un méta-modèle qui associe les éléments possédant une sémantique équivalente dans les deux modèles. Enfin, la plate-forme d'ingénierie AMMA offre des transformations bijectives depuis les méta-modèles des outils vers un méta-modèle pivot [BBJK05].

Cependant, ces exemples ne permettent pas de résoudre le problème lié à la présence de l'intégralité des informations nécessaires aux différents outils. En effet, permettre la mise en correspondance entre des concepts similaires représentés différemment n'est pas suffisant pour assurer le fait qu'un modèle contienne l'ensemble des informations requises au traitement que veut lui appliquer un outil, contrairement à nos patrons de conception.

3.5 Conclusion

L'analyse d'ordonnançabilité des STRECs est une tâche difficile. Dans ce chapitre, nous analysons pourquoi les méthodes d'analyse d'ordonnancement restent peu utilisées, bien que cela puisse être profitable.

Ce problème est dû, selon nous, à deux raisons principales :

— la difficulté de sélectionner les méthodes d'analyse adaptées au système en cours de développement et
— le problème de l'intégration de ces méthodes d'analyse dans le processus de développement et donc dans les chaînes d'outils.

Nous proposons de pallier à ces problèmes à l'aide de patrons de conception architecturaux, que nous définissons. Ces patrons sont utilisés pour sélectionner automatiquement des tests de faisabilité adaptés à un modèle d'architecture de STREC donné. Notre démarche est adaptable à tout type d'architecture par la création de nouveaux patrons de conception.

Nous expérimentons notre démarche avec deux types de STRECs : les systèmes monoprocesseur avec et sans ordonnancement hiérarchique. Les patrons relatifs au premier type sont présentés dans le chapitre 4, et ceux spécifiques au second sont décrits dans le chapitre 5.

Chapitre 4

Exemples de patrons de conception pour l'analyse des STRECs mono-processeur

Dans le chapitre 2, nous avons présenté l'importance de l'analyse temporelle des systèmes temps-réel embarqués critiques (STRECs) à l'aide de la théorie de l'ordonnancement. Cependant, l'utilisation de cette dernière implique une expertise approfondie, et on constate qu'elle est peu utilisée par le monde industriel. Nous proposons une approche de sélection automatique de tests de faisabilité reposant sur des patrons de conceptions architecturaux. Ces patrons explicitent l'association entre une liste de tests de faisabilité et un ensemble de STRECs. Les ensembles de STRECs, spécifiés par les patrons, sont modélisés par deux ensembles de contraintes sur leurs modèles d'architectures. Le premier ensemble caractérise l'environnement d'exécution, tandis que le second restreint les protocoles de communication et de synchronisation entre les entités logicielles du modèle d'architecture.

Dans la partie 4.1, nous spécifions cinq patrons de conception pour des STRECs mono-processeur. Pour ce faire, nous proposons un ensemble de contraintes d'environnement et cinq ensembles de contraintes de communication et de synchronisation entre les tâches.

La partie 4.2 décrit le modèle d'association entre un patron de conception et des listes de tests de faisabilité.

Puis en partie 4.3, nous proposons une méthode d'analyse de la composition de patrons. Elle permet la sélection de tests de faisabilité pour des STRECs utilisant différents protocoles de communication et de synchronisation.

Un algorithme de sélection de tests de faisabilité est proposé dans la partie 4.4. Ce dernier vérifie la conformité d'un modèle d'architecture à un patron, ou à une composition de patrons. Une fois la conformité du STREC à un patron est confirmée, l'algorithme sélectionne parmi la liste de tests de faisabilité associée au patron les tests pouvant être appliqués.

La partie 4.5 présente l'évaluation de notre approche. Nous avons réalisé un prototype capable de sélectionner les tests de faisabilité applicables à un modèle de STREC conforme

à un patron de conception. Enfin, nous concluons ce chapitre dans la partie 4.6.

4.1 Proposition de patrons de conception architecturaux pour l'analyse d'ordonnançabilité mono-processeur

Dans cette partie, nous décrivons les cinq patrons pris en considération dans notre approche du cas mono-processeur. Pour ce faire, nous commençons par donner la grille de lecture de nos tableaux récapitulant les ensembles de contraintes. Puis nous décrivons l'ensemble de contraintes d'environnement communes aux patrons présentés ici. Par la suite, nous présentons chacun des cinq patrons de conception *à la* Alexander (cf. partie 3.2.2). Les ensembles de contraintes de communication et de synchronisation des patrons sont alors présentés.

4.1.1 Grille de lecture des tableaux récapitulatifs d'ensembles de contraintes

L'objectif de cette partie est d'expliquer la façon dont nous présentons les ensembles de contraintes constituant nos patrons. Le tableau 4.1 donne la grille de lecture d'un tableau récapitulant un ensemble de contraintes. Chaque patron est modélisé par deux ensembles de contraintes, chacun récapitulé au sein d'un tableau (cf. partie 3.2.3).

TABLE 4.1 – Grille de lecture des tableaux récapitulant les ensembles de contraintes constituant les patrons.

Id1	La plate-forme d'exécution est mono-processeur.	Cst$i$
Id2	La politique d'ordonnancement peut être : *Earliest Deadline First*, *Least Laxity First* ou à priorités fixes.	Cst$j$

La première colonne de chaque tableau donne le numéro de la contrainte spécifiée sur la ligne associée. Toutes les contraintes ont un numéro unique au sein d'un ensemble de contraintes, mais peuvent être référencées dans plusieurs ensembles de contraintes sous un numéro différent.

La seconde colonne spécifie la contrainte en langage naturel. Cette spécification est réalisée à l'aide des concepts inhérents aux STRECs et à la théorie de l'ordonnancement définis lors du chapitre 2.

Enfin, la dernière colonne donne l'identifiant de la contrainte, i.e., deux contraintes identiques ont un numéro distinct au sein de chaque ensemble de contraintes mais sont référencées par un identifiant global. Un identifiant référence une contrainte exprimée avec les entités de Cheddar ADL.

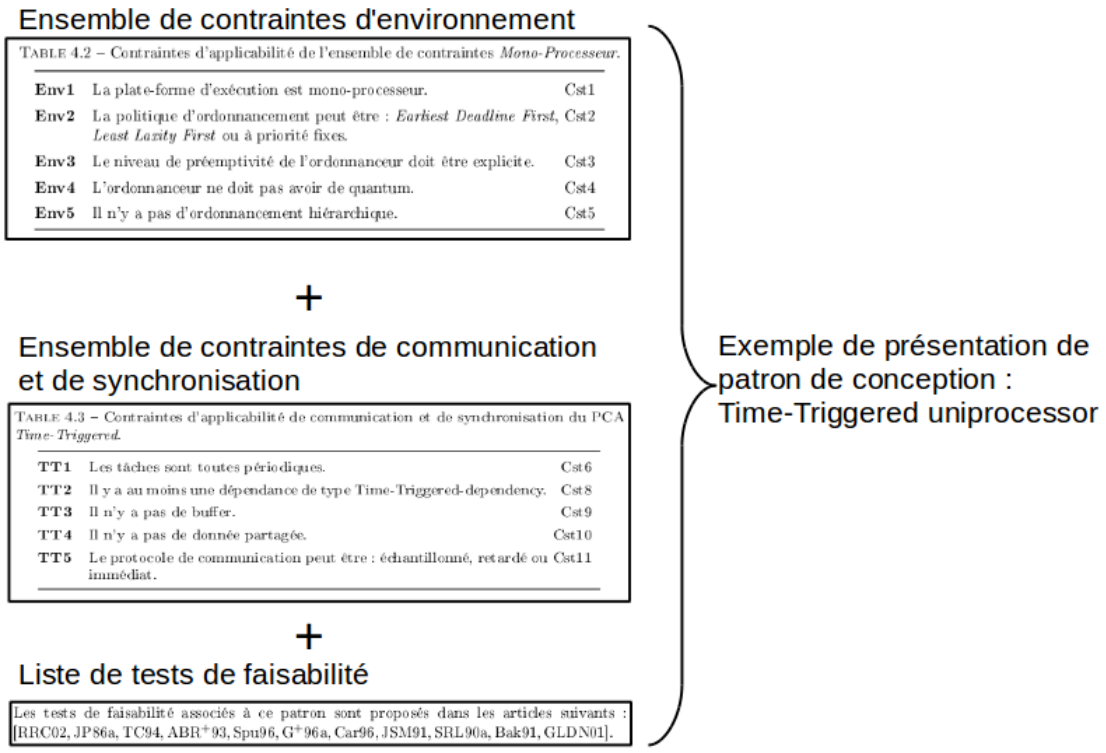

FIGURE 4.1 – Les éléments constituant la présentation d'un patron : exemple de Time-Triggered

4.1.2 L'ensemble de contraintes communes aux cinq patrons de conception : uniprocessor

Dans le cadre de ce chapitre, nous ne considérons qu'un seul type de plate-forme d'exécution : une architecture mono-processeur. Cette plate-forme est modélisée par un ensemble de contraintes communes à tous les patrons dédiés aux systèmes mono-processeur.

Comme nous le précisions dans la partie 3.2.3, nous distinguons deux types de contraintes : les contraintes d'environnement et les contraintes de communication et de synchronisation. L'ensemble de contraintes, présenté ci dessous, modélise un environnement d'exécution mono-processeur. Ces contraintes sont partagées par les cinq patrons présentés dans ce chapitre.

Pour plus de clarté, nous ne présentons cet ensemble de contraintes qu'une seule fois. La figure 4.1 résume les éléments constituant la présentation d'un patron au travers de l'exemple de *Time-Triggered Uniprocessor*. Ces éléments sont : un ensemble de contraintes d'environnement, un ensemble de contraintes de communication et de synchronisation et une liste de tests de faisabilité.

L'ensemble de contraintes d'environnement *Uniprocessor* :

Le cas des STRECs mono-processeur ayant été le premier étudié, il bénéficie d'une quantité importante de tests de faisabilité [SAa+]. On considère ici des systèmes sans ordonnancement hiérarchique. Le système comporte donc un seul ordonnanceur. Les systèmes déployés sur un unique processeur peuvent être analysés de multiples façons. Les critères de performances évalués pour ce type de système sont multiples et dépendent de sa partie logicielle. Les cinq contraintes sur le modèle d'architecture, spécifiant cet environnement, sont récapitulées dans la table 4.2.

TABLE 4.2 – Contraintes d'applicabilité de l'ensemble de contraintes *Uniprocessor*.

Env1	La plate-forme d'exécution est mono-processeur.	Cst1
Env2	La politique d'ordonnancement peut être : *Earliest Deadline First*, *Least Laxity First* ou à priorités fixes.	Cst2
Env3	Le niveau de préemptivité de l'ordonnanceur doit être explicite.	Cst3
Env4	L'ordonnanceur ne doit pas utiliser de quantum.	Cst4
Env5	Il n'y a pas d'ordonnancement hiérarchique.	Cst5

4.1.3 Les patrons de conception architecturaux dédiés aux STRECs mono-processeur

Dans cette partie, nous spécifions les cinq patrons de conception : *Time-triggered uniprocessor*, *Ravenscar uniprocessor*, *Unplugged uniprocessor*, *Blackboard uniprocessor* et *Queuedbuffer uniprocessor*.

Pour ce faire, nous présentons chacun des patrons selon le formalise d'Alexander [Ale79], i.e., sous la forme d'un contexte, d'un problème et d'une solution.

Afin d'illustrer chaque patron, un exemple de scénario d'ordonnancement est fourni. Tous les exemples présentent l'ordonnancement du même jeu de tâches, mais utilisant les différents protocoles de communication et de synchronisation.

Le jeu de tâches est décrit par le tableau 4.3. Nous considérons trois tâches périodiques τ_i avec $i \in \{1, 2, 3\}$. Les tâches τ_i sont synchrones et à échéances sur requêtes. L'algorithme

TABLE 4.3 – Jeu de tâches des exemples illustrant les patrons

Tâche	S_i	C_i	D_i	P_i
τ_1	0	2	6	6
τ_2	0	2	8	8
τ_3	0	4	12	12

d'ordonnancement est Rate Monotonic. Les paramètres des tâches sont les suivants : S_i donne l'instant d'activation de la tâche, C_i est sa capacité, D_i est son échéance relative et P_i est sa période.

Le patron de conception architectural *Time-Triggered uniprocessor*

Contexte : les communications de type *time-triggered* sont réalisées à l'aide d'une mémoire partagée. Les instants auxquels sont produites et reçues les données sont connus *a priori*. Par exemple, une tâche va recevoir un message lors de son activation et en envoyer un autre lors de sa terminaison [KB03, CRB+13, Ves98]. On s'assure, ainsi, que les envois et réceptions de messages n'interviennent jamais au même instant. Il n'y a donc pas besoin de mécanisme de synchronisation (pas de mutex par exemple). Ce protocole de communication est directement inspiré des connections *delayed* et *sampled* d'AADL [SAE09].

Problème : les systèmes conformes à ce patron sont relativement simples à analyser. Du point de vue de la théorie de l'ordonnancement, les tâches peuvent être considérées comme indépendantes. Ce patron autorise l'utilisation de nombreux tests de faisabilité, simples et efficaces. Ces derniers sont basés sur le facteur d'utilisation processeur et le calcul du temps de réponse pire cas.

Solution : les cinq contraintes de l'ensemble de contraintes de communication et de synchronisation *Time- triggered* sont récapitulées dans la table 4.4. Les tests de faisabilité associés à ce patron sont proposés dans les articles suivants : [RRC02, JP86a, TC94, ABR+93a, Spu96b, G+96a, LL73a, Car96, LSD89, BHR90, JSM91].

TABLE 4.4 – Ensemble de contraintes de communication et de synchronisation : *Time-Triggered*.

TT1	Les tâches sont toutes périodiques.	Cst6
TT2	Il y a au moins une communication de type Time-Triggered.	Cst8
TT3	Il n'y a pas de buffer.	Cst9
TT4	Il n'y a pas de donnée partagée.	Cst10
TT5	Le protocole de communication peut être : échantillonné ou retardé.	Cst11

Exemple d'ordonnancement d'un système conforme à *Time-Triggered uniprocessor*

Considérons que la tâche τ_1 envoie deux données : une à la tâche τ_2 avec le protocole de communication *retardé*, une à la tâche τ_3 avec le protocole de communication *échantillonné*.

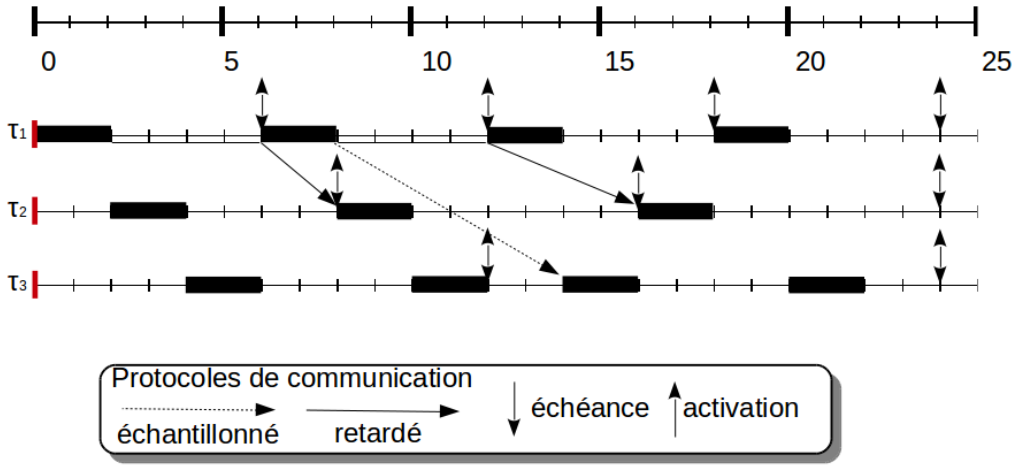

FIGURE 4.2 – Ordonnancement des trois tâches avec l'algorithme RM, avec trois communications de type *Time-Triggered*.

La figure 4.2 propose l'ordonnancement de ce jeu de tâches. Seuls les productions de messages reçus lors des 25 instants représentés dans le graphique sont apparentes.

La donnée produite par la tâche τ_1 et lue par la tâche τ_2 est retardée. Les données produites aux instants 2 et 8 sont lues lors des réveils de τ_2 consécutifs à la fin de la période de τ_1 durant laquelle la donnée est produite, i.e., lors des instants 8 et 16.

La donnée produite par la tâche τ_1 et lue par τ_3 est produite selon le protocole de communication *échantillonnée*. Elle est donc lue par τ_3 à l'instant 14. La donnée produite à l'instant 2 est écrasée par celle produite à l'instant 8.

Le patron de conception architectural *Ravenscar uniprocessor*

Contexte : le second patron de conception modélise les communications asynchrones entre les tâches du système. Avec *Ravenscar*, les tâches communiquent à l'aide de données partagées et dont l'accès est réalisé à l'aide d'un protocole d'héritage de priorité parmi PIP, PCP et ICPP. *Ravenscar* suppose donc que les données partagées sont protégées par des sémaphores. Ces dernières peuvent être utilisées pour implanter de multiples protocoles de synchronisation comme les sections critiques, lecteurs/écrivains, producteurs/consommateurs, etc. Pour le patron *Ravenscar*, l'utilisation des sémaphores est réduite à l'exclusion mutuelle. On peut noter que ce patron est inspiré du profil Ravenscar pour Ada [BDR98]. Cependant, nous considérons un ensemble plus large d'architectures.

Problème : les tâches ne sont plus indépendantes : leur temps de réponse pire cas doit considérer les temps de blocage dus aux sections critiques lors de l'accès aux données partagées. Les temps de blocage sont donc intégrés dans les calculs de temps de réponse pire-cas, ainsi que dans le facteur d'utilisation processeur.

Solution : les sept contraintes de l'ensemble de contraintes de communication et de synchronisation *Ravenscar* sont récapitulées dans la table 4.5. Les tests de faisabilité associés à ce patron sont proposés dans les articles suivants : [RRC02, JP86a, TC94, ABR+93a, Spu96b, G+96a, Car96, JSM91, SRL90a, Bak91, GLDN01].

TABLE 4.5 – Ensemble de contraintes de communication et de synchronisation : *Ravenscar*.

Rav1	Les tâches sont toutes périodiques ou sporadiques.	Cst7
Rav2	Il y a au moins une donnée partagée.	Cst12
Rav3	Il n'y a pas de buffer.	Cst9
Rav4	Il y a au moins deux tâches connectées à chaque donnée partagée et protégée par un sémaphore.	Cst13
Rav5	Le protocole de partage des données partagées doit être un des protocoles suivants : PIP, ICPP ou PCP.	Cst14
Rav6	Si le protocole choisi est PCP ou ICPP, la priorité affectée à la donnée partagée doit être supérieure ou égale aux priorités fixes de toutes les tâches accédant à la donnée.	Cst15
Rav7	Si le protocole choisi est PIP, deux tâches ne peuvent pas partager plus d'une donnée partagée.	Cst16

Exemple d'ordonnancement d'un système conforme à *Ravenscar uniprocessor*

Considérons que les deux tâches τ_1 et τ_3 partagent une donnée R. Nous supposons une seule section critique par tâche. Afin de décrire les sections critiques de τ_1 et τ_3, nous ajoutons de nouveaux paramètres au tableau récapitulant notre jeu de tâches :

TABLE 4.6 – Jeu de tâches illustrant le patron *Ravenscar Uniprocessor*.

Tâche	S_i	C_i	D_i	P_i	C_i^α	C_i^β	C_i^γ
τ_1	0	2	6	6	1	1	0
τ_2	0	2	8	8	2	0	0
τ_3	0	4	12	12	0	4	0

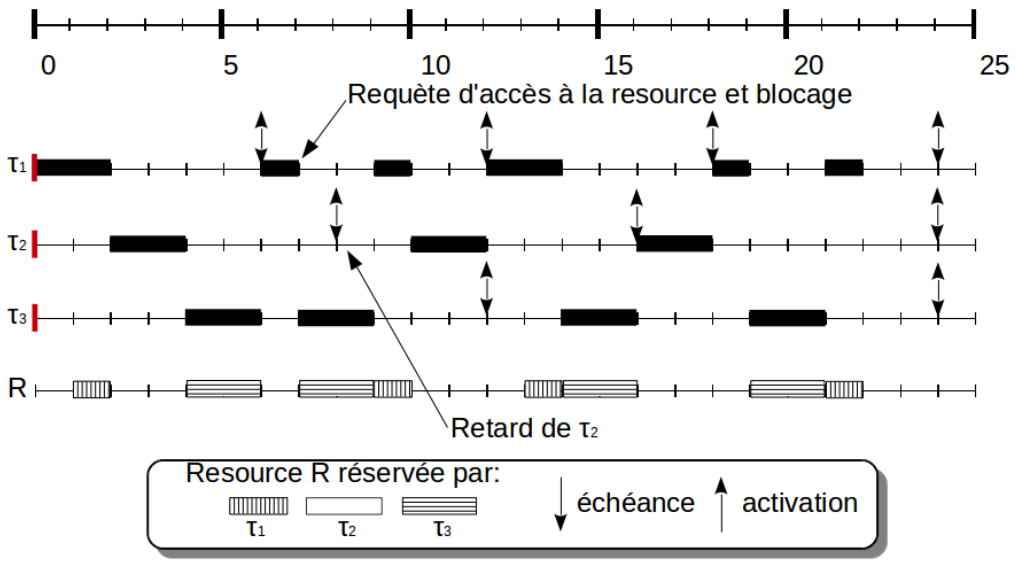

FIGURE 4.3 – Ordonnancement des trois tâches avec l'algorithme RM, avec un partage de donnée et le protocole d'héritage de priorité PIP.

— C_i^{α} : durée d'exécution de la tâche avant d'entrer dans la section critique,
— C_i^{β} : durée de la section critique,
— C_i^{γ} : durée d'exécution de la tâche après la section critique.

Le protocole de partage de la donnée est PIP. La figure 4.3 propose l'ordonnancement de ce jeu de tâches. Lors de l'instant 6, la tâche τ_1 requière l'accès à la donnée R. Cette donnée est alors bloquée par la tâche τ_3, qui hérite donc de la priorité de τ_1. La tâche τ_3 possède alors le plus haut niveau de priorité du jeu de tâche. C'est pourquoi la tâche τ_2 n'est plus prioritaire sur τ_3 lors des instants 8 et 9.

Le patron de conception architectural *Unplugged uniprocessor*

Contexte : ce patron modélise les STRECs constitués de tâches indépendantes uniquement. Les tâches n'effectuent ni communication, ni synchronisation. Le patron *Unplugged uniprocessor* a été spécifié à des fins d'analyses de composition de patrons (cf. partie 5.2.2).

Problème : l'analyse de ce type de système est identique à celle requise par *Time-Triggered uniprocessor*. En effet, les analyses relatives à ce dernier considèrent les tâches comme indépendantes.

Solution : les deux contraintes de l'ensemble de contraintes de communication et de syn-

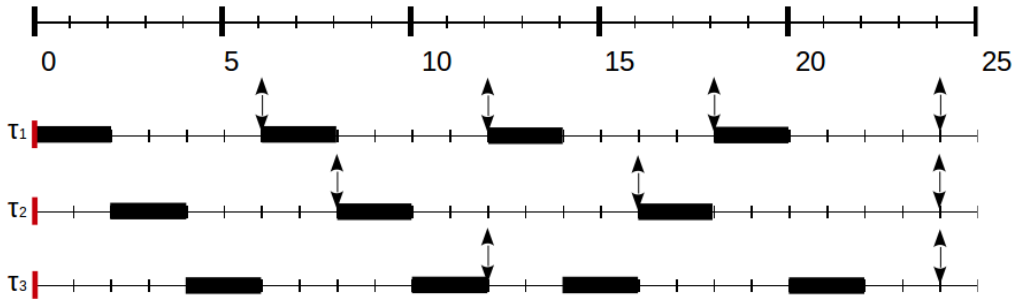

FIGURE 4.4 – Ordonnancement des trois tâches sans mécanisme de communication ou de synchronisation.

chronisation *Unplugged* sont récapitulées dans la table 4.7. Les tests de faisabilité associés à ce patron sont proposés dans les articles suivants : [RRC02, JP86a, TC94, ABR+93a, Spu96b, G+96a, LL73a, Car96, LSD89, BHR90, JSM91].

TABLE 4.7 – Ensemble de contraintes de communication et de synchronisation : *Unplugged*.

Unp1	Les tâches sont toutes périodiques.	Cst6
Unp2	Les tâches sont indépendantes.	Cst17

Exemple d'ordonnancement d'un système conforme à *Unplugged uniprocessor*

la figure 4.4 propose l'ordonnancement du jeu de tâches avec un algorithme d'ordonnancement à priorités fixes et l'algorithme Rate Monotonic pour l'affectation des priorités.

Le patron de conception architectural *Blackboard uniprocessor*

Contexte : *Blackboard* implante un protocole de communication lecteurs/écrivains. Ce type de communication est également inspiré de celui proposé dans le standard pour l'avionique ARINC653 [Ari97].

Pour *Blackboard*, tout message écrit sur une donnée partagée de ce type reste disponible jusqu'à ce que la donnée soit écrasée par un nouveau message ou supprimée.

La mise en file d'attente de messages n'est pas autorisée. Une tâche peut soit écrire un message sur le *blackboard*, soit lire un message ou soit supprimer un message écrit sur un *blackboard*.

Un délai d'attente maximal est spécifié pour les tâches en attente d'accès au *blackboard*.

Problème : Dans ce cas, le calcul des temps de blocage dûs aux accès aux données partagées est plus complexe que dans celui de *Ravenscar* (différents sémaphores pour les lecteurs et les écrivains par exemple). Il est également nécessaire de procéder à des analyses de détection de certains interblocages.

Solution : Les quatre contraintes de l'ensemble de contraintes de communication et de synchronisation *Blackboard* sont récapitulées dans la table 4.8. En plus de ceux utiles à l'analyse de *Ravencar Uniprocessor*, les tests de faisabilité associés à ce patron sont proposés dans les articles suivants : [BW95, TT92].

TABLE 4.8 – Ensemble de contraintes de communication et de synchronisation : *Blackboard*.

BB1	Les tâches sont toutes périodiques ou sporadiques.	Cst7
BB2	Il y a au moins un buffer de communication.	Cst18
BB3	La taille des buffers doit être de 1.	Cst30
BB4	Pour tous les buffers, le protocole de production/consommation de données est de type Blackboard.	Cst19
BB5	Deux tâches ne peuvent partager qu'un seul tampon de communication.	Cst20
BB6	Un délai d'attente maximal doit être spécifié.	Cst29

Exemple d'ordonnancement d'un système conforme à *Blackboard uniprocessor*

Considérons que les deux tâches τ_1 et τ_2 écrivent des données et que la tâche τ_3 lit des données sur un blackboard *BB*. Afin de spécifier les instants de production et de

TABLE 4.9 – Jeu de tâches illustrant le patron *BlackBoard Uniprocessor*.

Tâche	S_i	C_i	D_i	P_i	C_i^α	K_i
τ_1	0	2	6	6	0	écriture
τ_2	0	2	8	8	1	écriture
τ_3	0	4	12	12	2	lecture

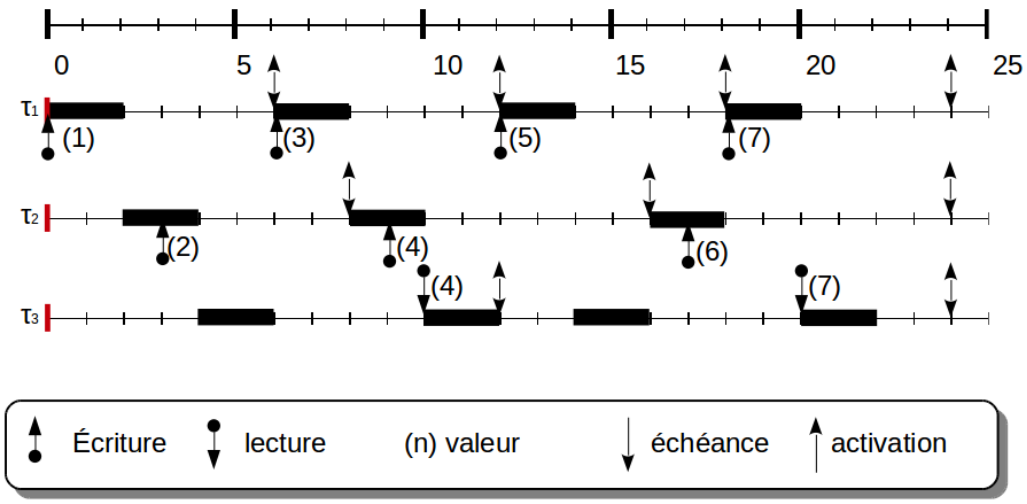

FIGURE 4.5 – Ordonnancement des trois tâches partageant des données via un Blackboard avec l'algorithme RM.

consommation, nous ajoutons de nouveaux paramètres au tableau récapitulant notre jeu de tâches :

— C_i^{α} : durée d'exécution de la tâche avant l'écriture ou la lecture d'une donnée,

— K_i : spécifie le type d'accès au Blackboard.

La figure 4.5 propose l'ordonnancement de ce jeu de tâches. Lors des instants 0, 6 ainsi que 3 et 9, les tâches τ_1 et τ_2 écrivent, respectivement les valeurs *(1)*, *(3)* et *(2)*, *(4)* sur le *Blackboard*. À l'instant 10, la tâche τ_3 lit la dernière valeur écrite, soit *(4)*. Cette valeur est ensuite écrasée par τ_1, puis τ_2, et à nouveau par τ_1. Enfin, lors de l'instant 20, l la tâche τ_3 lit à nouveau le blackboard. La dernière valeur écrite est alors *(7)*.

Le patron de conception architectural *Queued Buffer uniprocessor*

Contexte : Queued Buffer implante le protocole de communication producteurs/consommateurs. Ce type de communication est inspiré de celui proposé dans le standard pour l'avionique ARINC653 [Ari97].

Avec QueudBuffer, aucune donnée présente dans un message ne peut être perdue. Il n'est donc pas possible d'écraser les messages précédents lors du transfert.

Les buffers peuvent stocker de multiples messages dans une file. Ces derniers sont stockés dans la file dans l'ordre FIFO. Le nombre maximal de messages pouvant être stockés dans un buffer est déterminé par la taille du buffer, qui doit-être spécifiée.

Les tâches en attente d'accès à un buffer sont ordonnés selon leur priorité ou un protocole FIFO également. Si deux tâches ont la même priorité, le protocole FIFO est appliqué.

Un délai d'attente maximal est spécifié pour les tâches en attente de réception (lorsque le buffer est vide) ou d'émission de message (lorsque le buffer est plein). Une fois ce délai dépassé, les tâches concernées sont retirées de la file d'attente et remise dans l'état *prête*.

Problème : Une analyse d'empreinte mémoire des tampons doit être réalisée. Elle permet de prévenir les pertes de données (fréquence de production supérieure à celle de consommation) ou les famines (fréquence de consommation supérieure à la fréquence de production). L'analyse de l'empreinte mémoire des tampons peut être réalisée à l'aide de la théorie des files d'attente [SLNM05].

Solution : Les cinq contraintes de l'ensemble de contraintes de communication et de synchronisation *Queued buffer* sont récapitulées dans la table 4.10. En plus de ceux utiles à l'analyse de *Blackboard Uniprocessor*, les tests de faisabilité associés à ce patron sont proposés dans les articles suivants : [Tin94, GH98, TC94, PH03].

TABLE 4.10 – Ensemble de contraintes de communication et de synchronisation : *Queued Buffer*.

QB1	Les tâches sont toutes périodiques ou sporadiques.	Cst7
QB2	Il y a au moins un buffer de communication.	Cst18
QB3	La taille des buffers doit être spécifiée.	Cst21
QB4	Le nombre de productions et de consommations de messages effectuées par les tâches lors de leurs activations doit être spécifié.	Cst22
QB5	Les instants de production et de consommation doivent être spécifiés.	Cst23
QB6	Un délai d'attente maximal doit être spécifié.	Cst29

TABLE 4.11 – Jeu de tâches illustrant le patron *Queued Buffer uniprocessor*.

Tâche	r_i	C_i	D_i	T_i	C_i^α	K_i	Q_i
τ_1	0	2	6	6	0	production	1
τ_2	0	2	8	8	1	production	1
τ_3	0	4	12	12	2	consommation	1

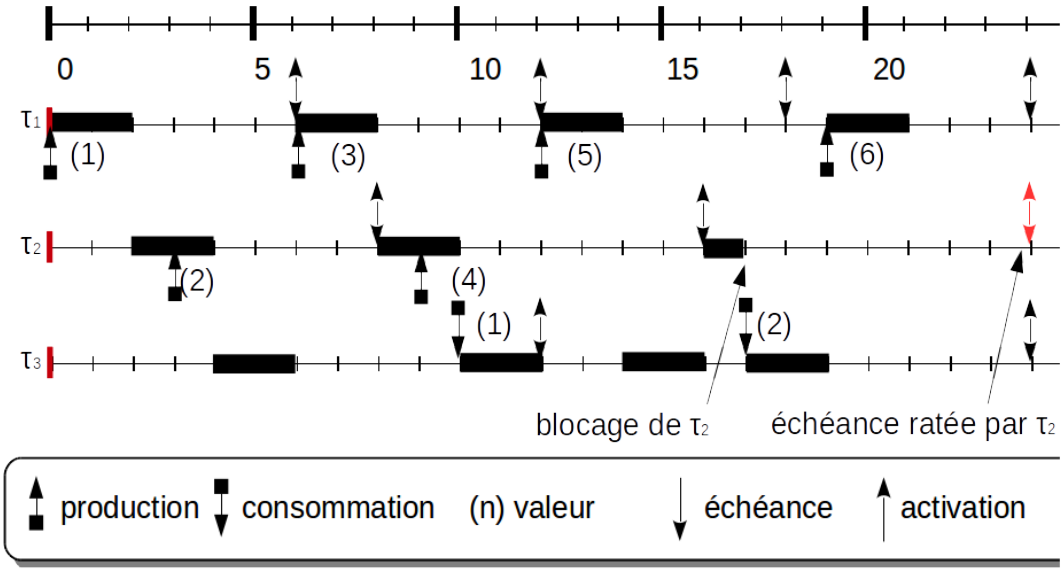

FIGURE 4.6 – Ordonnancement des trois tâches partageant des données via un Queued Buffer avec l'algorithme RM.

Exemple d'ordonnancement d'un système conforme à *Queued Buffer uniprocessor*

Considérons que les deux tâches τ_1 et τ_2 écrivent des données et que la tâche τ_3 lit des données sur un Queued buffer QB de taille 4. Afin de spécifier les instants de production et de consommation, nous ajoutons de nouveaux paramètres au tableau récapitulant notre jeu de tâches :

— C_i^α : durée d'exécution de la tâche avant la production ou la consommation d'une donnée,
— K_i : spécifie le type d'accès au Queued Buffer.
— Q_i : spécifie la quantité de données produites ou lues.

Entre les instants 0 et 9, les tâches τ_1 et τ_2 produisent deux données chacune. À cet instant, le buffer est plein. Lors de l'instant 10, la tâche τ_3 consomme la première valeur à avoir été produite, libérant ainsi une place dans le buffer. La tâche τ_1 produit une nouvelle valeur deux instants plus tard.

À l'instant 17, la tâche τ_2 est bloquée car elle ne peut pas produire de donnée. Lors de l'instant suivant, un seconde donnée est consommée par τ_3. La tâche τ_1 a alors débuté un nouveau travail et est élue. comme aucune autre donnée ne sera consommée avant l'instant 24, τ_2 est bloquée jusqu'à rater son échéance.

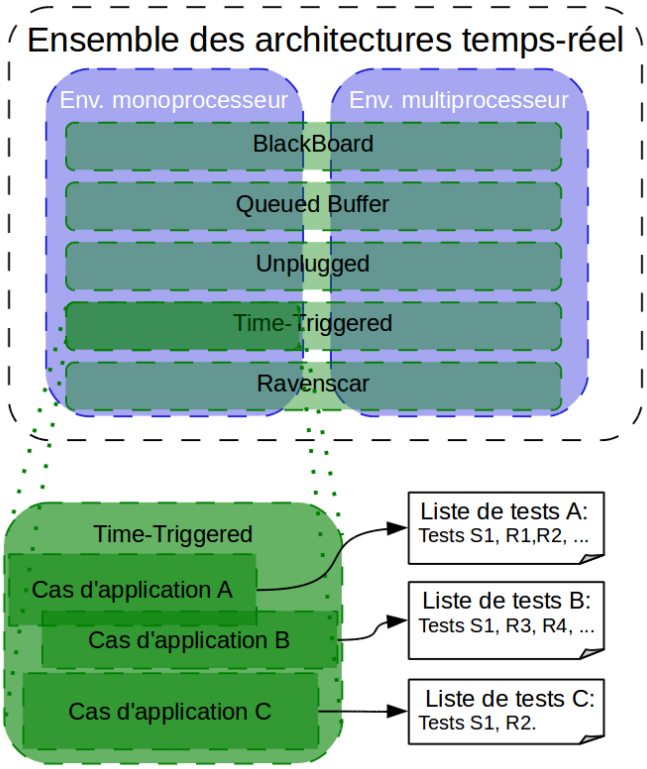

FIGURE 4.7 – Représentation ensembliste des modèles d'architectures conformes aux patrons.

Dans cette partie, nous vous avons exposé les contraintes spécifiant nos patrons de conception. Nous allons maintenant expliquer comment nous les associons à des tests de faisabilité.

4.2 Comment associer les patrons de conception architecturaux à des tests de faisabilité ?

Dans cette partie, nous proposons de modéliser les relations entre les patrons et les tests de faisabilité qui leur sont applicables. Dans un premier temps, nous présentons les associations entre un patron et les listes de tests de faisabilité applicables : appelées *cas d'application*. Nous proposons, ensuite, un modèle de cas d'application, illustré par un exemple.

Définition 4.1 ***(Cas d'application).*** *Un cas d'application modélise l'association entre un test de faisabilité et un ensemble de contraintes d'applicabilité suffisant à garantir son applicabilité.*

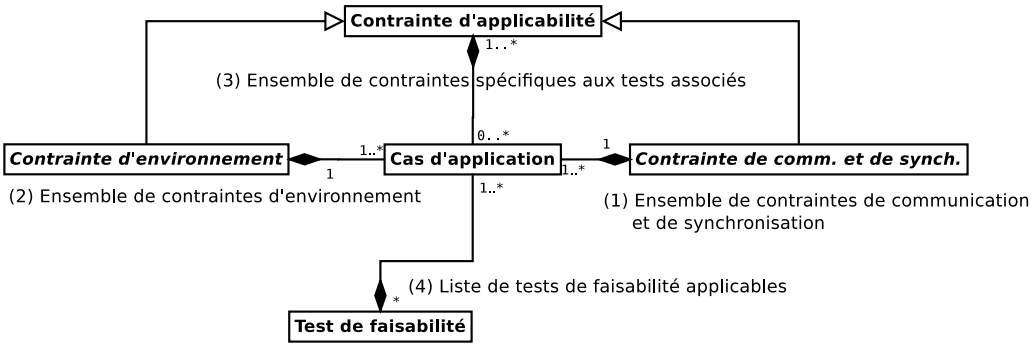

FIGURE 4.8 – Modèle d'association entre un modèles d'architecture et listes tests de faisabilité.

Les patrons permettent de définir des sous-ensembles de systèmes pouvant être analysés. La figure 4.7 représente l'ensemble de tous les types d'architectures temps-réel modélisables à l'aide d'un ADL dans cette thèse. Dans cette figure, chaque boîte cerclée de pointillés représente l'ensemble des modèles d'architecture conformes aux contraintes spécifiant le patron nommé. Plus le fond de la boîte est foncé, plus le nombre de contraintes à respecter est important. Une intersection entre un ensemble de contraintes d'environnement et et un ensemble de contraintes de communication délimite un ensemble de modèles d'architecture analysables.

Pour chacune de ces intersections, il existe un ensemble de tests de faisabilité applicables. Tous ces tests de faisabilités ne sont pas applicables à l'intégralité des modèles d'architecture conformes à un patron, mais à un sous-ensemble plus fin. Par exemple, le sous-ensemble de modèles d'architecture conformes au patron *Time-Triggered uniprocessor* supporte divers niveaux de préemptivité. Cependant, les méthodes d'analyse varient selon le niveaux préemptivité des tâches, la politique d'ordonnancement ou les instants d'activation des tâches, entre autres. Un cas d'application est donc joint d'un ensemble de contraintes d'applicabilité précisant les propriétés temporelles des tâches et le protocole d'ordonnancement des architectures conformes au patron de conception concerné.

Il existe donc de multiples associations entre un patron et les tests qui lui sont applicables. Chaque association est définie par un *cas d'application*. Un cas d'application est constitué d'une liste de tests de faisabilité applicables et d'une liste de contraintes d'applicabilité spécifiques aux tests de la liste. Par exemple, 128 cas d'application distincts existent pour le patron *time-triggered uniprocessor*. Il existe un cas d'application pour chaque modèle d'architecture pouvant être conforme à un patron.

La figure 4.8 décrit ce modèle d'association. Un cas d'application est constitué de quatre éléments :

1. un ensemble de contraintes de communication et de synchronisation parmi la liste des cinq présentés dans les tableaux 4.4, 4.5, 4.7, 4.10 et 4.8 ;
2. un ensemble de contraintes d'environnement ;

- Ensemble de contraintes d'environnement : *Uniprocessor*.
- Ensemble de contraintes de communication et de synchronisation : *Time-Triggered*.
- Ensemble de contraintes d'applicabilité précisant les propriétés temporelles des tâches et le protocole d'ordonnancement :
  — l'ordonnancement est préemptif ;
  — l'ordonnancement est a priorités fixes ;
  — les tâches sont synchrones ;
  — les échéances sont contraintes, i.e., les échéances des tâches sont inférieures ou égales à leur période.
- Liste de tests de faisabilité : [RRC02, JP86b, TBW94, ABR+93a].

FIGURE 4.9 – Exemple de cas d'application : *Time-Triggered uniprocessor* utilisant le protocole d'ordonnancement à priorité fixe Rate Monotonic et préemptif. Les tâches doivent être synchrones à échéance contrainte.

3. un ensemble de contraintes d'applicabilité associées au cas d'application et précisant les propriétés temporelles des tâches et le protocole d'ordonnancement ;
4. une liste de tests de faisabilité applicables si le modèle d'architecture à analyser est conforme aux trois ensembles de contraintes.

La figure 4.9 présente un cas d'application. Ce cas d'application associe le patron de conception *Time-Triggered uniprocessor* avec une liste de tests de faisabilité et les contraintes d'applicabilité qui leur sont spécifiques.

Pour ce cas d'application, la liste de tests de faisabilité comprend :
— un test proposé par Richard et al. [RRC02], qui calcule un intervalle de faisabilité.
— deux tests qui évaluent le temps de réponse pire cas des tâches de l'architecture [JP86b, TBW94, ABR+93a].

Afin que les tests de faisabilité soient applicables, le modèle d'architecture doit être conforme à *Time-Triggered Uniprocessor* et composé de tâches synchrones à échéances contraintes. Le protocole d'ordonnancement utilisé est à priorités fixes et préemptif.

4.3 Composition de patrons de conception architecturaux

Dans la partie précédente, nous avons présenté les patrons de conception architecturaux. Dans la pratique, une application temps-réel peut utiliser de multiples protocoles de communication et de synchronisation. Elle est alors constituée d'une composition d'entités conformes à des patrons différents, comme un système utilisant des communications de type *Time-Triggered* et d'autres de type *Ravenscar* par exemple.

Dans cette partie, nous introduisons la notion de composition de patrons. Nous proposons, par la suite, une méthode d'analyse statique permettant de définir les compositions compatibles avec notre objectif de sélection de tests de faisabilité.

4.3.1 La *composabilité* des patrons de conception architecturaux

Les tests de faisabilité, étudiés dans le cadre de ce chapitre, prennent en compte l'ensemble des entités du modèle d'architecture. Afin d'analyser un système, toutes les tâches et dépendances doivent être prises en compte simultanément, et par conséquent l'ensemble des protocoles de communication et de synchronisation.

Un STREC utilisant différents protocoles de communication et de synchronisation est constitué de plusieurs éléments du modèle d'architecture pouvant être conformes à des patrons différents.

Définition 4.2 ***(Modèle de STREC composite).*** *Un modèle STREC composite comprend plusieurs entités conformes à des patrons différents.*

Afin qu'un modèle de STREC composite soit analysable, il est nécessaire que les patrons auxquels sont conformes ses entités soient composables, au sens de l'analyse d'ordonnancement.

Définition 4.3 ***(Composabilité).*** *Deux patrons sont dits composables s'il est possible d'appliquer une stratégie d'analyse d'ordonnançabilité permettant d'analyser un système composé de l'ensemble des entités conformes aux deux patrons de conception.*

Une solution pour la sélection de tests de faisabilité avec un modèle de STREC composite est de considérer que le modèle d'architecture est conforme, non pas à un ensemble de patrons, mais à un seul parmi les patrons auxquels sont conformes les entités du modèle de STREC composite. On parle alors de patron dominant.

Définition 4.4 ***(Patron de conception dominant).*** *Soit S, le modèle d'architecture d'un STREC composite dont les entités sont conformes aux patrons pca_1 et pca_2. Le patron pca_1 est dit dominant vis-à-vis du patron pca_2 si S peut être analysé comme s'il était conforme à pca_1 uniquement.*

Les tests de faisabilité sont sélectionnés comme si le système était conforme au patron dominant. Par exemple, un système composé de deux parties, respectivement conformes aux patrons *Time-Triggered uniprocessor* et *Ravenscar uniprocessor*, est analysé comme globalement conforme à *Ravenscar uniprocessor*. *Ravenscar uniprocessor* est dit dominant vis-à-vis de *Time-Triggered uniprocessor*.

4.3.2 Analyse de la composabilité de patrons de conception architecturaux

On souhaite prouver la domination d'un patron envers un autre. Soit pca_1 et pca_2 deux patrons de conception. Nous supposons le patron pca_1 dominant vis-à-vis de pca_2. Nous prouvons qu'un modèle d'architecture S composé de deux ensembles d'entités E_1 et E_2,

respectivement conformes à pca_1 et pca_2, est conforme à pca_1 ; i.e., le modèle d'architecture composite S doit respecter l'ensemble des contraintes du patron dominant pca_1.

Pour ce faire, nous utilisons un raisonnement par l'absurde afin de prouver que chacune des contraintes de pca_1 est respectée. Dans cette partie, nous donnons un exemple de preuve par l'absurde du respect de la contrainte *"Rav1 : les tâches sont périodiques ou sporadiques."* par un modèle d'architecture composé d'un ensemble d'entités conforme à *Ravenscar Uniprocessor* et un autre conforme à *Time-Triggered Uniprocessor*. Notons que pour prouver la domination de pca_1 sur pca_2, ce type de démonstration doit être fait pour toutes les contraintes de pca_1.

Eléments de preuve :

[Preuve Rav1] Soit $S = E_1 \cup E_2$ un modèle d'architecture comprenant m tâches $\{T_1, ..., T_n\}$.

Soit $E_1 = \{T_1, ..., T_n\}$ les entités de S conformes à *Time-Triggered Uniprocessor*.

Soit $E_2 = \{T_{n+1}, ..., T_m\}$ les entités de S conformes à *Ravenscar Uniprocessor*.

Considérons que la contrainte *"Rav1 : les tâches sont périodiques ou sporadiques."* n'est pas respectée par S.

Étant donné que E_2 est conforme à *Ravenscar*, elle respecte *Rav1*.

Alors, $\exists\ T_{fail} \in E_1$ une tâche n'étant ni périodique ni sporadique.

Or, E_1 est conforme à *Time-Triggered*. Elle respecte donc la contrainte *"TT1 : les tâches sont périodiques."*.

Donc, E_1 contient la tâche T_{fail} et respecte *TT1*, ce qui est contradictoire. La contrainte *Rav1* est donc bien respectée par S dans son intégralité. □

Le tableau 4.12 récapitule les compositions des patrons dont la faisabilité a été prouvée. La conformité des modèles de STREC composites à toutes les contraintes d'applicabilité du patron dominant a été étudiée. Les premières lignes et colonnes contiennent les patrons composés, tandis que les intersections contiennent le patron dominant, résultant de leur composition. Les cases vides du tableau sont les combinaisons pour lesquelles les patrons ne sont pas composables.

On peut noter que les compositions autorisées sont restreintes à un type de communication *Time-Triggered* (ou l'absence de communication) avec un type de communication *Ravenscar*, *BlackBoard* ou *Queued Buffer*. En effet, les trois derniers types de communication ont un impact sur l'ordonnancement selon le protocole utilisé. Ainsi, les attentes dûes aux accès aux données partagées se calculent différemment selon le protocole d'accès aux données. Dans le cas de *Time-Triggered* et *Unplugged*, il n'y a pas de temps d'attente, et les tâches peuvent donc être considérées comme indépendantes.

Lors des deux dernières parties, nous avons présenté les patrons et notre approche pour les associer à des tests de faisabilité applicables aux modèles d'architecture leur étant conforme. Nous avons ensuite étudié la composition de patrons de conception.

Patrons uniproc.	Unplugged	Time-Trig.	Ravenscar	Blackboard	QueuedBuf.
Unplugged	Unplugged	Time-Trig.	Ravenscar	Blackboard	QueuedBuf.
Time-Triggered	Time-Trig.	Time-Trig.	Ravenscar	Blackboard	QueuedBuf.
Ravenscar	Ravenscar	Ravenscar	Ravenscar	∅	∅
Blackboard	Blackboard	Blackboard	∅	Blackboard	∅
QueuedBuffer	QueuedBuf.	QueuedBuf.	∅	∅	QueuedBuf.

TABLE 4.12 – Tableau récapitulatif des patrons dominants pour toutes les compositions autorisées.

4.4 Algorithme de sélection de tests de faisabilité

Dans cette partie, nous présentons un algorithme permettant la sélection automatique de tests de faisabilité, ainsi que son application à une étude de cas. Cet algorithme, en cinq étapes, consiste à détecter des patrons auxquels un modèle d'architecture est conforme, et à analyser leur composition.

Dans un premier temps, nous présentons une étude de cas qui servira à illustrer notre algorithme. Ensuite, nous proposons un modèle de graphe représentant les tâches d'un modèle d'architecture et leurs dépendances. Enfin, l'algorithme est présenté étape par étape.

4.4.1 Étude de cas : un système automobile simplifié

Dans cette partie, nous présentons un système conforme au patron *Time-Triggered uniprocessor*. L'architecture proposée modélise une version simplifiée d'un système temps-réel embarqué pour l'automobile. Le modèle d'architecture est produit à l'aide du langage d'AADL. La figure 4.10 propose la représentation graphique du modèle d'architecture. Ce système implante trois fonctionnalités : le contrôle des phares, le contrôle des essuie-glaces et l'ESP (Électro-Stabilisateur Programmé), un système de correction de trajectoire. Il est composé de 12 threads, communicant à travers des data ports AADL. Le data port AADL est utilisé pour modéliser les communication de type Time-Triggered.

Quatre threads lisent les données produites par des capteurs (détection de pluie, de vitesse, de direction et de lumière) lors de leur réveil. Cinq autres threads envoient des instructions aux actuateurs du véhicule (essuie-glaces, injection, système de freinage, direction et phares).

Un premier thread de contrôle des essuie-glaces reçoit les données des threads lecteurs des capteurs de vitesse et de pluie, puis transmet le résultat de son traitement au thread en charge de l'actuateur des essuie-glaces.

Un second thread de contrôle des phares reçoit les données de la tâche lectrice du capteur de lumière, puis transmet le résultat de son traitement au thread en charge de l'actuateur des phares.

Un thread de contrôle de l'ESP reçoit les données des tâches associées aux capteurs de

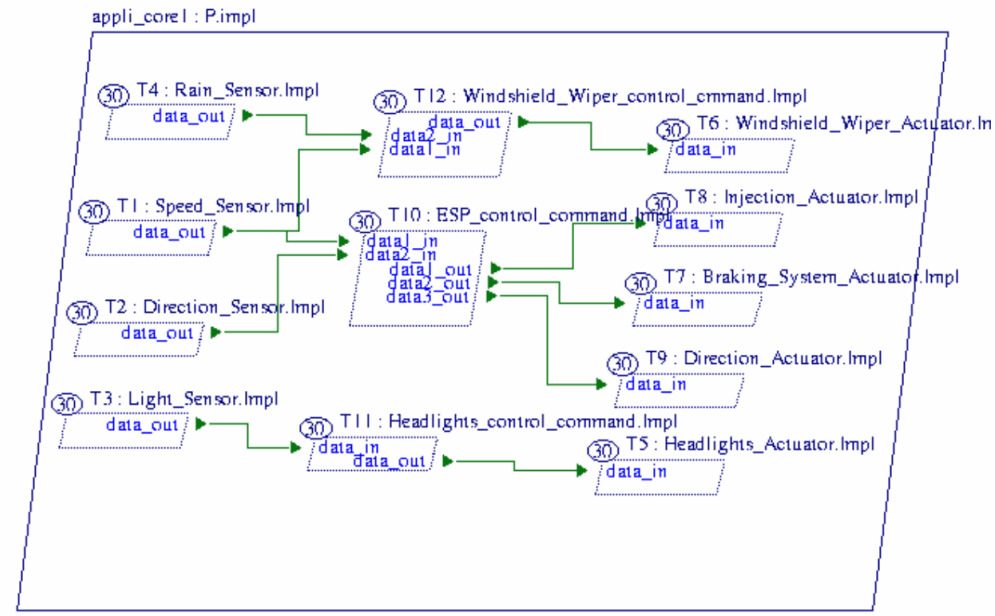

FIGURE 4.10 – Représentation graphique de l'étude de cas modélisée à l'aide d'AADL.

vitesse et de direction, et transmet le résultat de son traitement aux threads en charge des actuateurs des systèmes d'injection et de freinage, ainsi qu'à la direction.

Tous les threads ont une période de 30ms, une échéance sur requête et une capacité de 2ms. Le système est déployé sur un environnement d'exécution conforme à l'ensemble de contraintes *Uniprocessor*. L'ensemble des threads sont synchrones. La politique d'ordonnancement est préemptive et à priorités fixes. Les priorités sont affectées aux threads selon la méthode *deadline monotonic*. Par construction, l'ensemble des contraintes du patron *Time-Triggered Uniprocessor* sont respectées.

Cette partie a présenté une étude de cas servant à illustrer l'algorithme permettant de sélectionner les tests de faisabilité applicables à un modèle d'architecture mono-processeur. Nous allons, à présent, détailler ce dernier.

4.4.2 Le graphe de dépendances d'un modèle d'architecture

Les modèles d'architectures contiennent une quantité importante d'information. En effet, plus les utilisations potentielles du modèle sont variées, plus la quantité d'information requise est importante. Dans le cas de la sélection de tests de faisabilité, seule une quantité limitée d'information est nécessaire : les dépendances entre les tâches. Nous proposons donc d'abstraire un modèle d'architecture à l'aide d'un graphe de dépendances. Ce graphe de dépendances nous permet d'identifier les groupes de tâches interagissant les unes avec les autres.

Définition 4.5 ***(Graphe de dépendances).*** *Le graphe de dépendances d'une application temps-réel est un triplet $G = (N, A, \mu)$, où :*

- *N est un ensemble fini de nœuds, chaque nœud est une tâche du modèle d'architecture ;*
- *$A \in N \times N$ est l'ensemble des arcs du graphe, c'est à dire une dépendance entre deux tâches ;*
- *$\mu : A \rightarrow Dependency\_Type$ est une fonction retournant le type de dépendance modélisée par un arc, avec $Dependency\_Type \in \{Precedence, Queuing\_Buffer, Communication, Time\_Triggered\_Communication, Resource, BlackBoard\_Buffer\}$*
- *Si $N = \emptyset$ alors le graphe est vide.*

Par ailleurs, nous définissons la notion de sous-graphe comme suit.

Définition 4.6 ***(Sous-graphe).*** *Étant donné un graphe de dépendances $G = (N, A, \mu)$, un sous-graphe de G est un graphe $S = (N_S, A_S, \mu_S)$ tel que :*

- *$N_S \subseteqq N$,*
- *$A_S = A \cap (N_S \times N_S)$ est l'ensemble des arcs du graphe,*
- *μ_S est la restriction de μ à A_S, tel que*

$$\mu_S(a) = \begin{cases} \mu(a) & si\ a \in A_S \\ non\ défini & sinon \end{cases}$$

La notation $S \subseteqq G$ indique que S est un sous-graphe de G.

Nous définissons également trois opérateurs ensemblistes sur ce modèle de graphe : l'union, l'intersection et la différence.

Définition 4.7 ***(Intersection de deux graphes de dépendances).*** *L'intersection de deux graphes $G_1 = (N_1, A_1, \mu_1)$ et $G_2 = (N_2, A_2, \mu_2)$ retourne le graphe $G_\cap = (N_\cap, A_\cap, \mu_\cap)$ tel que :*

- *$\forall n \in N_\cap, n \in N_1$ et $\exists n \in N_2$,,*
- *$\forall a \in A_\cap, a \in A_1$ et $a \in A_2$,,*
- *$\mu_\cap$ est la restriction de μ à $A_\cap$.*

La notation $G_1 \cap G_2 = G_\cap$, indique que $G_\cap$ est l'intersection de G_1 et G_2.

Définition 4.8 ***(Union de deux graphes de dépendances).*** *L'union de deux graphes $G_1 = (N_1, A_1, \mu_1)$ et $G_2 = (N_2, A_2, \mu_2)$ retourne le graphe $G_\cup = (N_\cup, A_\cup, \mu_\cup)$ tel que :*

- *$\forall n \in N_\cup, n \in N_1$ ou $n \in N_2$,*
- *$\forall a \in A_\cup, a \in A_1$ ou $a \in A_2$,*

- $\mu_\cup$ *est la restriction de* μ *à* $A_\cup$.

La notation $G_1 \cup G_2 = G_\cup$, indique que $G_\cup$ est l'union de G_1 et G_2.

Définition 4.9 ***(Différence entre deux graphes de dépendances).*** *La différence entre deux graphes* $G_1 = (N_1, A_1, \mu_1)$ *et* $G_2 = (N_2, A_2, \mu_2)$ *retourne le graphe* $G_- = (N_-, A_-, \mu_-)$ *tel que :*

- $\forall n \in N_-, (n \in N_1$ *et* $n \notin N_2)$ *ou* $(n \notin N_1$ *et* $n \in N_2)$,
- $\forall a \in A_-, (a \in A_1$ *et* $a \notin A_2)$ *ou* $(a \notin A_1$ *et* $a \in A_2)$,
- μ_- *est l'extension de* μ *à* A_-.

La notation $G_1 \setminus G_2 = G_-$, indique que G_- est la différence entre G_1 et G_2.

Enfin, nous proposons un ensemble de *vues*. Ces dernières sont utilisés pour l'identification de composantes connexes par type de dépendances présentes dans le modèle d'architecture. Nous définissons deux types de vues sur le graphe de dépendances : les vues simples et les vues multiples.

Une vue simple est une fonction prenant un graphe de dépendances en entrée et retournant un de ses sous-graphes.

Définition 4.10 ***(Vue simple).*** *Une vue simple sur le graphe* G *est une fonction* $f_{view} : G \to G'$ *avec* $G' \subseteqq G$.

Une vue multiple est une fonction prenant un graphe de dépendances en entrée et retournant plusieurs de ses sous-graphes.

Définition 4.11 ***(Vue multiple).*** *Une vue multiple sur le graphe* G *est une fonction* $f_{view} : G \to G^*$ *avec* G^* *une liste de graphes et* $\forall G_i \in G^*, G_i \subseteqq G$.

Pour la mise en œuvre de notre algorithme de sélection de tests de faisabilité, nous avons besoin d'élaborer une vue par dépendance et une vue par composantes connexes. Ces vues nous permettent d'extraire du graphe des composantes connexes par types de dépendances. Pour ce faire nous définissons deux vues.

Définition 4.12 ***(Vue simple par type de dépendances).*** *Une vue simple par type de dépendance sur un graphe* G *est une fonction* $f_d : G \to G'$, *avec* $G = (N, A, \mu)$ *et* $G' = (N', A', \mu')$, *telle que :*

- $G' \subseteqq G$,
- $\forall a_1, a_2 \in (A' \times A')$, $\mu'(a_1) = \mu'(a_2)$

Définition 4.13 ***(Vue multiple par composantes connexes).*** *Une vue multiple par composantes connexes sur un graphe* G *est une fonction* $f_d : G \to G^*$, *avec* $G = (N, A, \mu)$ *et* $\forall G_i \in G^*$, $G_i = (N_i, A_i, \mu_i)$, *telle que :*

- $\forall G_i \in G^*, G_i \subsetneqq G$,
- $\forall n_1, n_2 \in (N_i \times N_i)$, $\exists$ *un chemin* $c = (n_1, ..., n_2)$ *, i.e., pour tout couple de deux sommets consécutifs* $u, v \in c$*, il existe un arc* $a \in A_i$ *tel que* $a = (u, v)$.

Le graphe de dépendances occupe une place centrale dans le processus de sélection de tests de faisabilité. La partie suivante présente l'algorithme de sélection.

4.4.3 Les cinq étapes de l'algorithme de sélection de tests de faisabilité

L'algorithme de sélection de tests de faisabilité est constitué de cinq étapes. La première étape consiste à construire le graphe de dépendances G des tâche du modèle d'architecture. Ensuite, on analyse G afin d'identifier les ensembles d'entités pouvant être conformes aux différents patrons de conception. Chaque ensemble d'entités est une composante connexe par type de dépendance du graphe construit à l'étape précédente. On vérifie alors que ces ensembles d'entités respectent bien les contraintes du patron caractérisant l'utilisation du type de dépendance de la composante connexe. Une fois que la conformité à un patron de chaque ensemble d'entités est confirmée, nous analysons leur composition. La conformité du modèle d'architecture à un unique patron de conception *pca* est alors déterminée. Lors de l'étape suivante, les cas d'application relatifs à *pca* sont évalués. Les tests de faisabilité associés au premier cas d'application auquel est conforme le modèle d'architecture sont sélectionnés.

Ci après, nous présentons les cinq étapes de l'algorithme de sélection de tests de faisabilité.

Étape 1 : construction d'un graphe de dépendances des tâches

Dans cette étape, à partir du modèle d'architecture, on construit le graphe G des dépendances entre les tâches.

Soit $S = (T, Res, Buf, Dep)$ un modèle d'architecture modélisant un système où :

- $T = \{t_1, ..., t_i\}$ est l'ensemble des tâche du modèle,
- $Res = \{res_1, ..., res_j\}$ est l'ensemble des données partagées du modèle,
- $Buf = \{buf_1, ..., buf_k\}$ est l'ensemble des buffers de communication du modèle,
- $Dep = (TT, SD, BU)$ est un triplet de dépendances tel que :
  - $TT = \{tt_1, ..., tt_l\}$ est l'ensemble des communications synchrones du modèle, avec $tt_x = (t_y, t_z)$ un couple de tâches appartenant à T ;
  - $SD = \{sd_1, ..., sd_m\}$ est l'ensemble des données partagées du modèle, avec $sd_u = (t_v, dep_w)$ où t_v est une tâche de T et dep_w est une donnée de Dep ;
  - $BU = \{bu_1, ..., bu_n\}$ est l'ensemble des buffers du modèle, avec $bu_r = (t_s, buf_t)$ où t_s est une tâche de T et buf_t est un buffer de Buf.

On instancie un graphe de dépendances $G = (N, A, \mu)$ tel que :

- $\forall t_a \in T$, on ajoute un nœud n_a à N ;

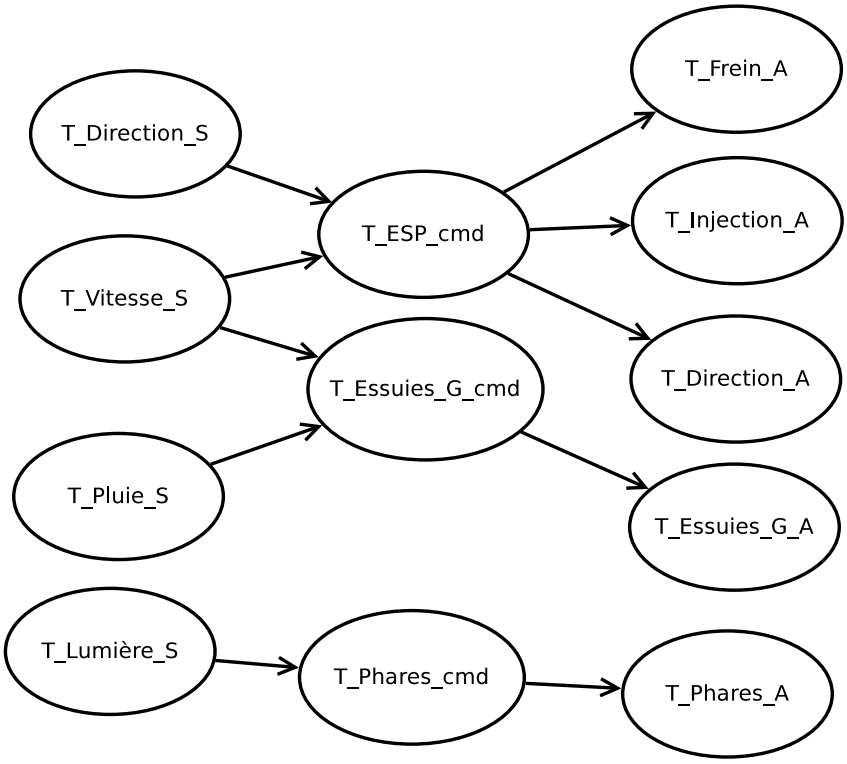

FIGURE 4.11 – Représentation graphique du graphe de dépendances instancié lors de l'étape 1.

- $\forall tt_b = (t_y, t_z) \in TT$, on ajoute un arc a_b à A avec $a_b = (n_y, n_z)$ et $\mu(a_b) = Time\_Triggered\_Communication$;
- $\forall sd_c = (t_u, dep_w)$ et $sd_d = (t_v, dep_w) \in SD$, on ajoute un arc a_e à A avec $a_e = (n_u, n_v)$ et $\mu(a_e) = Resource$;
- $\forall bu_f = (t_r, buf_t)$ et $bu_g = (t_s, buf_t) \in BU$, on ajoute un arc a_h à A avec $a_h = (n_r, n_s)$ et $\mu(a_h) = Queuing\_Buffer$ ou $\mu(a_h) = BlackBoard\_Buffer$ (en fonction du protocole de communication implanté par le buffer).

Les autres types de dépendances (précédence, message, etc) ne sont pas décrites ici, pour plus de concision. Elles sont ajoutés au graphe de la même façon que les autres dépendances.

La figure 4.11 propose la représentation graphique du graphe de dépendances correspondant à l'étude de cas de la partie 4.4.1. Le graphe est composé de 12 nœuds et de 10 dépendances de type *Time-Triggered*.

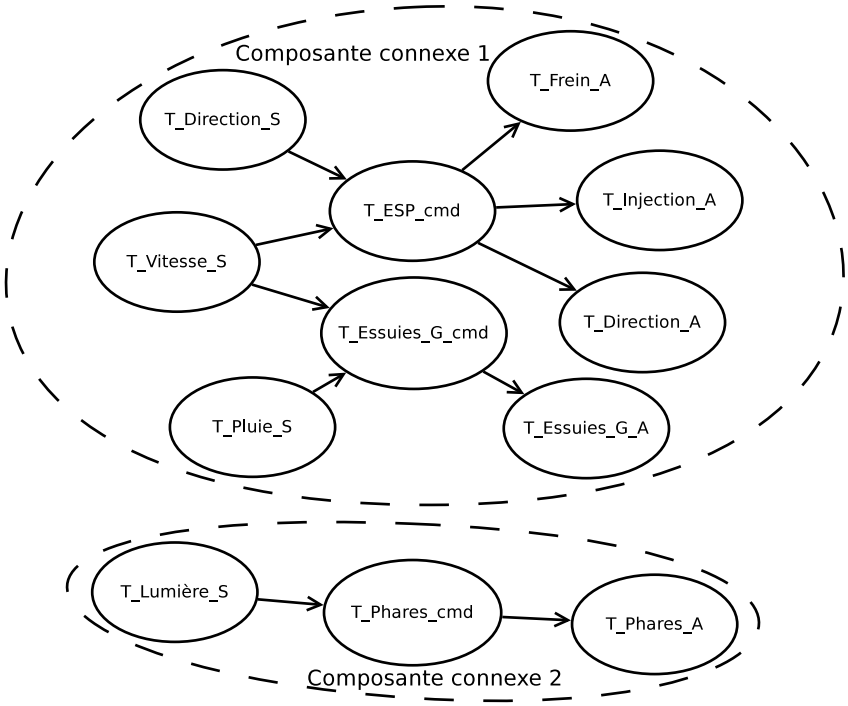

FIGURE 4.12 – Graphe de dépendances : deux composantes connexes avec pour type de dépendance des communications conformes à *Time-Triggered*.

Étape 2 : analyse du graphe de dépendances

Une fois le graphe de dépendances construit, on cherche à identifier les sous-graphes pouvant être conformes à un patron de conception. Ces sous graphes sont les composantes connexes par type de dépendances, i.e., une composante connexe pour laquelle tous les arcs partagent un même type de dépendance. On construit donc ces sous-graphes.

La figure 4.12 représente les deux composantes connexes de notre étude de cas. La première est constituée de trois nœuds correspondant aux tâches de contrôle des phares : *T_Lumière_S*, *T_Phares_cmd* et *T_Phares_A*. La seconde contient le reste du graphe. Elle représente les tâches implantant les deux fonctionnalités de contrôle des essuie-glaces et de l'ESP.

Étape 3 : Vérification de la conformité aux patrons

Les patrons sont définis par des ensembles de contraintes. Ainsi, la conformité des composantes connexes extraites lors de l'étape précédente vis-à-vis des contraintes doit être vérifiée, afin de déterminer si elles sont conformes à un patron.

Si l'un de ces sous-ensembles d'entités ne respecte pas l'intégralité des contraintes du patron concerné, il ne sera pas possible de sélectionner des tests de faisabilité. Á la fin de

cette étape, l'ensemble des instances de patrons ont été identifiées.

Les deux composantes connexes de notre étude de cas sont conformes au patron *Time-Triggered uniprocessor*. Le modèle d'architecture est donc composé de deux parties conformes à *Time-Triggered uniprocessor*.

Étape 4 : Analyse de composition de patron

Une fois l'intégralité du système identifié comme un ensemble d'instances de patrons, on analyse leur composition afin de pouvoir déterminer les tests de faisabilité qui vont être appliqués à l'ensemble du modèle. Pour ce faire, on s'appuie sur les règles de composition (cf. partie 4.3).

On évalue la composition des instances de patrons deux à deux.Si cette composition est autorisée par la table de composition (cf. table 4.12), on ne considère alors plus qu'un unique patron : celui qui est dominant.

Dans le cas où deux patrons ne sont pas composables, il ne sera pas possible de sélectionner des tests de faisabilité.

Le système automobile simplifié est composé de deux parties conformes au patron *Time-Triggered Uniprocessor*. Il est donc, à la fin de cette étape, considéré comme étant conforme au patron *Time-Triggered Uniprocessor* (cf. tableau 4.12, colonne 2, ligne 2).

Étape 5 : recherche du cas d'application spécifique au modèle d'architecture et sélection des tests de faisabilité

Lors des quatre étapes précédentes, nous avons déterminé la conformité du modèle d'architecture considéré à un de nos patrons. Comme nous l'avons précisé précédemment, il existe de multiples listes de tests de faisabilité pouvant être sélectionnées pour un seul et même patron. Chacune de ces listes est associée à un patron, ainsi qu'à une liste de contraintes d'applicabilité supplémentaires : ce sont les cas d'application.

La conformité du modèle d'architecture à un cas d'application est détectée par la vérification du respect de chacune des contraintes d'applicabilité lui étant associées. Les cas d'application sont étudiés de façon itérative jusqu'au premier cas d'application conforme au modèle d'architecture.

Notre étude de cas donne lieu à la sélection de cinq tests de faisabilité : un basé sur le calcul de l'intervalle de faisabilité [RRC02], deux basés sur le facteur d'utilisation processeur [LSD89, LL73a] et deux basés sur le calcul du pire temps de réponse [ABR+93b, JP86a].

4.5 Validation et expérimentation

L'objectif de cette partie est de présenter le processus d'évaluation de notre approche. On peut noter que, dans cette partie, nous nous bornons à exposer les aspects méthodologiques de notre évaluation. La mise en œuvre a été réalisée à l'aide du processus de développement de Cheddar et assisté par l'ingénierie dirigée par les modèles. Les détails de ce processus sont décrits dans le chapitre 6.

Cette partie comporte deux sous-parties : une présentation succincte du prototype, suivie du processus d'évaluation.

4.5.1 Prototypage assisté par l'ingénierie dirigée par les modèles

Afin d'évaluer notre approche, nous avons réalisé un prototype intégré à Cheddar. Cheddar implante de nombreux tests de faisabilité. Notons que Cheddar est en partie constitué de code généré à partir du méta modèle des entités qu'il manipule [PS07]. La modélisation et la génération de code sont réalisées grâce au méta-atelier Platypus [Pla07].

Le code de l'outil a été produit manuellement pour les patrons de conception *Time-Triggered uniprocessor*, *Ravenscar uniprocessor* et *Unplugged uniprocessor*. L'objectif est de générer la partie du prototype propre aux patrons de conception et à leur détection (l'implantation du graphe de dépendances particulièrement). De cette façon nous facilitons l'extension du prototype avec de nouveaux patrons pour lesquels le code peut être généré automatiquement.

Un générateur d'architectures dédié

Dans l'optique de valider notre approche, nous avons conçu un générateur de modèles d'architecture. La génération d'architectures est paramétrable par le nombre d'entités de chaque type proposé par Cheddar ADL. Les types d'entités pris en charge par la génération sont les suivants : *processor*, *core*, *task*, *buffer*, *dependency*, *message*, *resource* et *address space*.

Les propriétés des entités générées prises en compte dans la sélection des tests de faisabilité sont générées aléatoirement, soit dans un ensemble de valeurs et de types conformes aux patrons, soit dans l'ensemble global des possibilités proposées par Cheddar ADL. Le tirage aléatoire est réalisé selon une loi uniforme.

Le générateur d'architecture propose également la génération d'architectures non-conformes aux patrons.

4.5.2 Processus d'évaluation

L'objectif de notre évaluation est de tester la correction, la robustesse et le passage à l'échelle de notre prototype. Nous procédons à trois expérimentations. La première consiste à vérifier la détection du non-respect de chacune des contraintes individuellement. Pour ce faire, des architectures conformes et non-conformes à chacune des contraintes, sont générées. La seconde phase comprend la sélection de tests de faisabilité pour des études de cas. La troisième et dernière phase consiste en la sélection de tests de faisabilité pour des architectures générées. Ces dernières sont générées avec un nombre d'entités variant de 5 tâches et 5 dépendances à 1000 tâches et 1000 dépendances.

TABLE 4.13 – Propriétés temporelles des tâches du système automobile simplifié

Nom	Type	P_i	C_i	D_i	S_i
Toute tâche du modèle d'architecture	Périodique	30	2	30	0

4.5.3 Évaluation de la correction de l'implantation de nos contraintes à l'aide de modèles d'architectures générés

L'objectif de cette expérimentation est de vérifier la correction de l'implantation des contraintes contenues dans les patrons de conception et les cas d'application.

Pour ce faire, nous générons un nombre de modèles d'architecture égal à deux fois le nombre de contraintes implantées. Pour chaque contrainte c, une architecture conforme A_c et une non-conforme A_{sc} à c sont créées. Ensuite, le respect de chaque contrainte c est évalué sur les deux architecture A_c et A_{sc}.

Lors de cette première phase, les évaluations des contraintes ont toutes retourné le résultat attendu. Nos contraintes ont donc été correctement implantées.

4.5.4 Évaluation de la correction de l'algorithme à l'aide d'études de cas

L'objectif de cette évaluation est la correction de l'algorithme de sélection de tests de faisabilité.

Pour ce faire, nous avons appliqué notre algorithme de sélection de tests de faisabilité à deux études de cas : Le système automobile simplifié présenté en partie 4.4.1 et l'architecture du *Mars Pathfinder* [CDKM02].

Le système automobile simplifié

Le modèle d'architecture du système automobile simplifié est constitué de 12 tâches possédant les mêmes propriétés temporelles. Les propriétés temporelles communes à toutes les tâches du modèle sont proposées dans le tableau 4.13.

Les cinq tests de faisabilité sélectionnés pour le système automobile simplifié sont décrits dans la partie 4.4.3.

Ces cinq tests suffisent à prouver l'ordonnançabilité du système.

Mars Pathfinder

Mars Pathfinder est un projet de la NASA lancé en 1997 et ayant pour objectif d'explorer la surface de la planète Mars [CDKM02].

Cottet et al. ont présenté une étude de cas décrivant l'architecture de cet engin. Le STREC constituant l'architecture de contrôle de l'appareil est constitué de sept tâches décrites dans le tableau 4.14.

L'architecture du Mars Pathfinder est une instance du patron *Ravenscar uniprocessor*. En effet, les quatre tâches *Data Distribution*, *Measure Task*, *Radio Task* et *Meteo*

TABLE 4.14 – Jeu de tâches de l'application Mars Pathfinder

i	Nom	Type	P_i	C_i	D_i	S_i
1	Bus scheduling.	Périodique	200	2.13	200	0
2	Data Distribution	Périodique	200	1.43	22	0
3	Control Task	Périodique	200	52.84	100	0
4	Radio Task	Périodique	200	1.43	17	0
5	Camera Task	Sporadique	100	4.08	100	0
6	Measure Task	Périodique	100	1.43	24	0
7	Meteo Task	Périodique	100	1.43	24	0

patrons	Unplugged	Time-Triggered	Ravenscar	Blackboard	Queued Buffer
Unplugged	25	25	25	25	25
Time-Triggered	∅	25	50	50	50
Ravenscar	∅	∅	25	50	50
Blackboard	∅	∅	∅	25	50
Queued Buffer	∅	∅	∅	∅	25
Non-conforme	10	10	10	10	10

FIGURE 4.13 – Quantité d'architectures générées pour l'évaluation de notre approche.

Task accèdent à une même donnée partagée. Cet accès est réalisé à l'aide d'une exclusion mutuelle et selon le protocole *PIP*. Les priorités des tâches sont statiques et affectées à l'aide de l'algorithme *Rate Monotonic*. Cinq tests de faisabilité sont sélectionnés [RRC02, LSD89, LL73a, ABR+93a, JP86a].

Les tests sélectionnés nous ont permis d'effectuer l'analyse d'ordonnançabilité des deux modèles d'architecture présentés ci-dessus. Nous montrons, ainsi, la correction de notre algorithme de sélection de tests de faisabilité.

4.5.5 Évaluation à l'aide d'architectures générées

Afin d'évaluer la robustesse et le passage à l'échelle de notre prototype, nous avons généré des modèles d'architecture auxquels nous avons appliqué notre algorithme de sélection de tests de faisabilité.

La figure 4.13 résume le nombre d'architectures générées. On sélectionne les tests de faisabilité pour chacune d'entre elles. Le nom des tests de faisabilité sélectionnés, ou les contraintes non respectées sont stockées dans un fichier. Cette sélection (ou non-sélection) est ensuite vérifiée pour chacune des architectures.

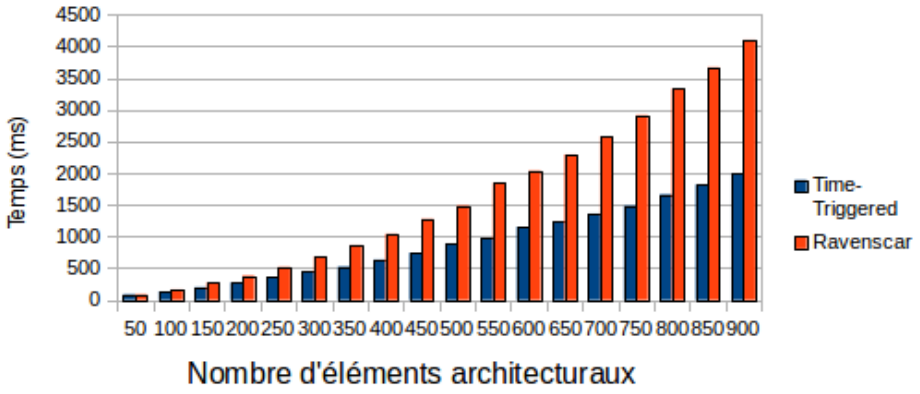

FIGURE 4.14 – Temps de réponse de l'algorithme de sélection de tests de faisabilité, en fonction du nombre de tâches et de dépendances du modèle d'architecture analysé (nombre de tâches = nombre de dépendances).

La figure 4.14 présente le temps de réponse de l'algorithme de sélection de tests de faisabilité, en fonction du nombre de tâches et de dépendances du modèle d'architecture analysé.

Cette évaluation a été exécutée sur une plate-forme Ubuntu 12.04. La machine utilisée possède un processeur Intel(R) Dual CPU T2330 1.60GHz avec 2,0GiB de mémoire RAM.

Cette évaluation montre que notre prototype est robuste et passe à l'échelle.

On peut noter que le temps de réponse de notre prototype varie de quelques millisecondes à quatre secondes pour de grands systèmes (1000 tâches + 1000 dépendances). Les guides d'ergonomie pour les interfaces homme-machine conseillent un temps de réponse de deux secondes maximum, avec une variation tolérée de moins d'une seconde autour de cette borne. Nous en déduisons donc que notre approche semble adaptée à l'utilisation de notre prototype intégré à un environnement de développement interactif, comme Eclipse par exemple.

De plus, le temps de réponse nécessaire à la détection de patron et la sélection de tests de faisabilité, augmente de façon linéaire vis-à-vis de la somme du nombre de tâches et de dépendances du système.

Enfin, toutes les contraintes d'applicabilité non-respectées par le second ensemble d'ar-

chitectures générées ont été détectées.

4.6 Conclusion

L'analyse d'ordonnançabilité est une étape cruciale du processus de développement des STRECs. Cette analyse dépend des caractéristiques de l'architecture du système, et son application est une tâche ardue. Dans ce chapitre, nous avons proposé une approche de modélisation des relations entre architectures et tests de faisabilité à l'aide de patrons de conception architecturaux. Les patrons sont spécifiques à un protocole de communication ou de synchronisation entre tâches. Nous montrons comment les patrons peuvent être utilisés pour la sélection automatique de tests de faisabilité spécifiques aux architectures leur étant conforme. Nous expliquons comment les patrons peuvent être composés afin de permettre l'analyse de système utilisant plusieurs protocoles de communication. Par la suite, un algorithme de sélection automatique de tests de faisabilité est détaillé. Enfin, notre approche est évaluée à l'aide d'études de cas et d'architectures générées.

Ce chapitre a présenté notre approche dans le cas d'un environnement d'exécution mono-processeur. Lors du chapitre suivant, nous adaptons notre approche aux STRECs utilisant un ordonnancement hiérarchique.

Chapitre 5

Extension de l'approche à l'analyse des STRECs à ordonnancement hiérarchique

Lors du chapitre précédent, nous avons présenté une approche de sélection de tests de faisabilité à l'aide de patrons de conception architecturaux. Notre proposition suppose un environnement d'exécution mono-processeur non-hiérarchique. Cet environnement a été modélisé à l'aide de l'ensemble de contraintes listées dans le tableau 4.2.

Nous proposons de généraliser cette approche en étendant le cadre de nos travaux avec le cas des systèmes utilisant un ordonnancement hiérarchique.

Nous présentons un modèle de patron de conception architectural permettant d'associer des ensembles de contraintes et des tests de faisabilité spécifiques à l'analyse de STRECs à ordonnancement hiérarchique.

Nous spécifions trois patrons : *ARINC653*, *two-levels-independent-applications*, *n-levels-independent-workloads*. Pour ce faire, nous proposons quatre ensembles de contraintes d'environnement spécifiques aux STRECs à ordonnancement hiérarchique.

La multiplicité des ensembles de contraintes d'environnement et le modèle d'association présentés dans ce chapitre impliquent l'extension de l'algorithme de sélection de tests de faisabilité. Cette extension réalise l'analyse de conformité d'un STREC aux ensembles de contraintes d'environnement et permet la sélection des nouvelles méthodes d'analyse d'ordonnancement spécifiques aux STRECs à ordonnancement hiérarchique. L'ensemble de ces contributions est illustré à l'aide d'une étude de cas conforme au standard ARINC653 [Ari97].

Ce chapitre est organisé de la façon suivante. Dans la partie 5.1, nous présentons les STRECs à ordonnancement hiérarchique. La partie 5.2 présente les deux approches pour l'analyse d'ordonnançabilité de systèmes à ordonnancement hiérarchique que nous considérons. La partie 5.3 propose le modèle de patrons de conception architecturaux adapté à la sélection de tests de faisabilité pour les STRECs à ordonnancement hiérarchique, tandis que la partie 5.4 liste les ensembles de contraintes d'environnement que nous avons définis.

Nous spécifions, au sein de la partie 5.5, trois nouveaux patrons spécifiques aux STRECs à ordonnancement hiérarchique. Ensuite, une étude de cas conforme au standard ARINC653 est décrite en partie 5.6. L'extension de l'algorithme de sélection de tests de faisabilité est présenté partie 5.7. Enfin, l'évaluation de ces travaux est proposée en partie 5.8, avant de conclure en partie 5.9.

5.1 Modélisation des systèmes temps-réel embarqués critiques à ordonnancement hiérarchique

Il existe de nombreux formalismes pour décrire les STRECs à ordonnancement hiérarchique. Dans ce chapitre, nous utilisons le formalisme et le vocabulaire de Davis et Burns [DB05].

L'ordonnancement hiérarchique permet le partage hiérarchique de ressources entre différentes applications utilisant chacune leur propre protocole d'ordonnancement. Dans ce contexte, une application est un composant logiciel assumant la responsabilité de l'ordonnancement des tâches et des autres applications le constituant.

Définition 5.1 ***(Ordonnancement hiérarchique).*** *L'ordonnancement hiérarchique consiste à utiliser plusieurs ordonnanceurs au sein d'un même système : un ordonnanceur global et des ordonnanceurs locaux [DB05].*

Définition 5.2 ***(Application).*** *Une application est un composant logiciel comportant des tâches ainsi que des applications et étant responsable de leur ordonnancement [DB05].*

L'ensemble des tâches et applications qu'une application A a la responsabilité d'ordonnancer est appelé la charge de travail de A. Le modèle d'architecture exemple donné à la figure 5.1, est composé de cinq applications : R, A, B, C et D.

Définition 5.3 ***(Charge de travail).*** *La charge de travail d'une application A modélise l'ensemble des tâches et applications que A a la responsabilité d'ordonnancer [DB05].*

Par exemple, au sein de notre modèle, les applications C et D, ainsi que la tâche *1* constituent la charge de travail de l'application A.

On distingue deux types d'ordonnanceurs : l'ordonnanceur global et les ordonnanceurs locaux. La distinction entre ces deux rôles découle de la nature de la charge de travail à ordonnancer, ainsi que des ressources à allouer.

Un ordonnanceur global a la responsabilité de répartir la ressource processeur entre les applications formant un système.

Définition 5.4 ***(Ordonnanceur global).*** *Ce type d'ordonnanceur détermine, à tout instants à quelle application doit être allouée la ressource processeur [DB05].*

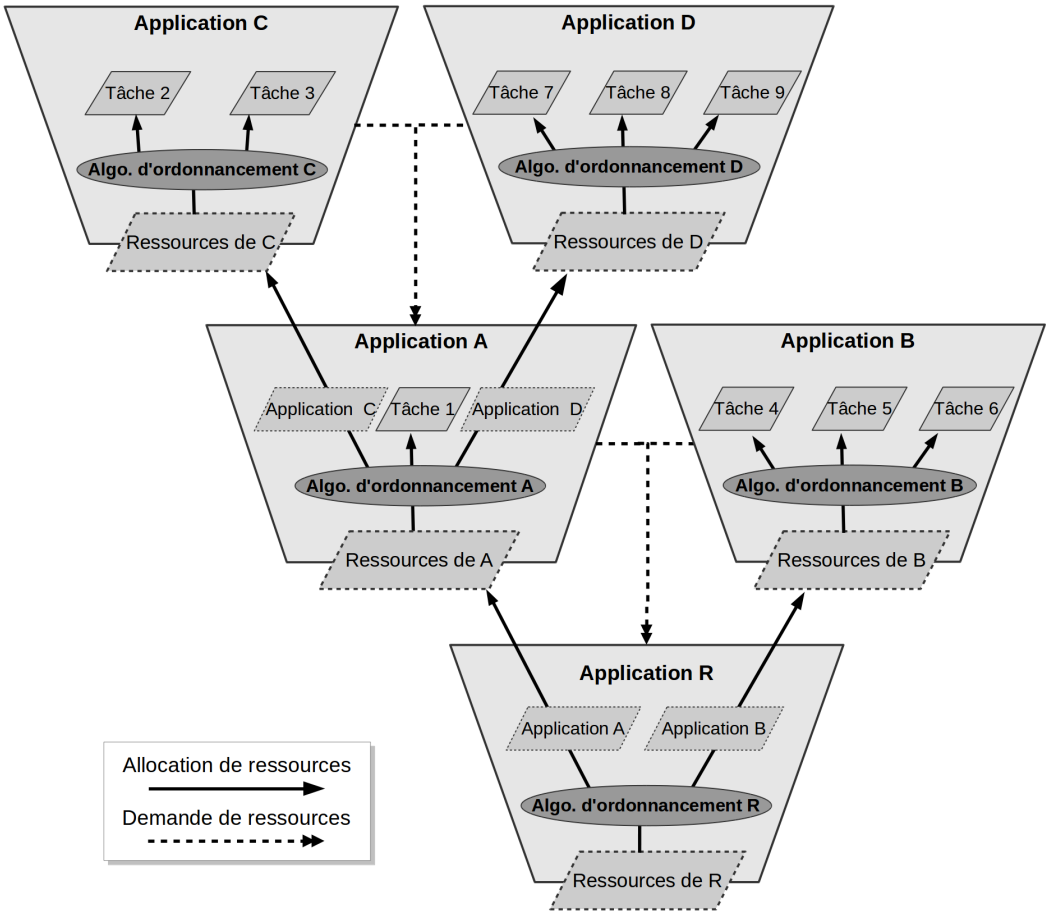

FIGURE 5.1 – Modèle d'architecture d'un STREC à ordonnancement hiérarchique.

Dans le système donné en exemple à la figure 5.1, il y a deux ordonnanceurs globaux, celui de l'application R ainsi que celui de l'application A.

Un ordonnanceur local est inclus dans une application. Il a pour rôle de déterminer quelle tâche ou application de la charge de travail du composant qu'il ordonnance doit effectivement s'exécuter, lorsque son application se voit allouer la ressource processeur par l'ordonnanceur global.

Définition 5.5 ***(Ordonnanceur local).*** *Un ordonnanceur local est utilisé pour déterminer quelle tâche ou application de la charge de travail de l'application choisie par l'ordonnanceur global doit effectivement s'exécuter [DB05].*

Au sein de l'exemple fourni dans la figure 5.1, les quatre ordonnanceurs (A, B, C et D) sont des ordonnanceurs locaux. L'ordonnanceur A est donc à la fois local et global.

Un STREC à ordonnancement hiérarchique peut être modélisé comme un arbre d'applications.

Les ressources sont allouées par Une application alloue des ressources aux applications et aux tâches constituant sa charge de travail. La quantité de ressources allouées est déterminée par son protocole d'ordonnancement.

La figure 5.1 schématise ce type de système. Chaque trapèze décrit une application. Ici, le système est constitué de cinq applications : A, B, C, D et R. L'application R est l'application approvisionnant en ressource processeur de toutes les autres applications du système. Elle approvisionne les deux applications des sa charge de travail A et B en ressource processeur, selon l'algorithme d'ordonnancement R. Les deux applications A, B répartissent ensuite les quantités de ressource leur ayant été allouées selon leurs propres algorithmes d'ordonnancement.

Les applications C et D ainsi que la tâche *1* forment la charge de travail de A. Les groupes de tâches [2, 3], [4, 5, 6] et [7, 8, 9] sont les charges de travail respectives des applications C, B et D.

Le standard ARINC653 est un exemple de standardisation de ces principes, dans le cadre du développement de systèmes avioniques [Pri08]. La profondeur de l'arbre est alors limitée à deux.

L'ordonnancement hiérarchique peut être utilisé pour permettre l'exécution de multiples applications, tout en garantissant leur isolation temporelle et spatiale. Ceci peut être réalisé correctement lorsque le système réalise un partitionnement, i.e., les applications sont indépendantes d'un point de vue fonctionnel. Cela permet l'isolation des erreurs : une erreur apparaissant lors de l'exécution d'une application n'aura pas de conséquence sur l'exécution des autres applications du système.

5.2 Les différentes approches d'analyse des STRECs à ordonnancement hiérarchique

Dans cette partie, nous identifions deux approches pour l'analyse d'ordonnançabilité de systèmes utilisant l'ordonnancement hiérarchique : l'approche globale et l'approche compositionnelle. L'approche globale ne considère que les systèmes avec un nombre connu de niveaux hiérarchiques, tandis que l'approche compositionnelle permet l'analyse de systèmes comportant un nombre de niveaux hiérarchiques quelconques.

Dans un premier temps, un exemple de méthode d'analyse globale est présenté. Puis, l'approche compositionnelle est décrite et illustrée.

5.2.1 Les méthodes d'analyse globale

L'approche globale d'analyse d'ordonnançabilité est l'adaptation des méthodes d'analyses dédiées aux systèmes sans ordonnancement hiérarchique aux systèmes comportant un nombre connu de niveaux hiérarchiques.

Dans le cas où seuls deux niveaux sont considérés, les tests de faisabilité traitent un niveau local constitué de toutes les applications réalisant un ordonnancement local et le niveau global modélisé par une unique application réalisant l'ordonnancement global. Ces tests ont pour principe de vérifier l'ordonnançabilité de chaque application du niveau local. L'analyse d'une seule application prend en compte les interférences et temps de blocage causés par le niveau global, i.e., l'ordonnancement des autres applications du système. Il est donc nécessaire de connaître les caractéristiques internes de toutes les applications constituant le STREC.

Les méthodes d'analyse existantes permettent l'évaluation de divers critères de performances, similaires au cas non-hiérarchique. Dans la suite de cette partie, nous décrivons un exemple d'analyse : la recherche de l'intervalle de faisabilité pour le calcul d'ordonnancement.

Exemple d'analyse globale : le calcul d'ordonnancement sur un intervalle de faisabilité

Le critère de performance évalué par ce test de faisabilité est le calcul d'ordonnancement sur un intervalle de faisabilité.

Ce test de faisabilité est spécifique aux STRECs à ordonnancement hiérarchique à deux niveaux de hiérarchie avec un ordonnancement hors-ligne au niveau global.

La séquence d'ordonnancement du niveau global pour une unique application du niveau local est définie comme suit.

Définition 5.6 ***(Séquence d'ordonnancement pour une application*** Γ***).*** *Une séquence d'ordonnancement* Ξ *est un couple* (φ,P) *où* φ *est une liste de* N *paires d'instants* $\{ (\varsigma_1,\varepsilon_1), (\varsigma_2,\varepsilon_2),\ldots, (\varsigma_N,\varepsilon_N) \}$ *avec* $(0 < \varsigma_1 < \varepsilon_1 < \varsigma_2 < \varepsilon_2 < \ldots < \varsigma_N < \varepsilon_N < P)$ *où* $N \geq 1$, *et* P *est la période de l'application* Γ. *La ressource processeur est disponible pour la charge de travail de* Γ *sur les intervalles de temps* $(\varsigma_i + j \times P)$, *avec* $1 \leq i \leq N$ *et* $j \geq 0$.

L'ordonnancement est à priorité fixe au niveau local avec des tâches à échéances contraintes, i.e., l'échéance d'une tâche est inférieure à sa période.

Un jeu de tâches est ordonnançable par l'application Γ si et seulement si les travaux réalisés lors de la première activation des tâches sont ordonnançables pour tous les instants critiques basés sur intervalles IBCIs (Interval-Based Critical Instance) [MFC01].

Définition 5.7 ***(Instants critiques basés sur intervalles).*** *On appelle* $\varepsilon_1, \varepsilon_2, \ldots \varepsilon_N$ *d'une partition* $\Gamma = \{ (\varsigma_1,\varepsilon_1), (\varsigma_2,\varepsilon_2),\ldots, (\varsigma_N,\varepsilon_N) \}$ *les points critiques basés sur intervalles (IBCPs). Si une tâche est activée simultanément avec toutes les autres tâches de la charge de travail dont elle fait partie, lors d'un IBCP, cet instant est appelé instant critique basé sur intervalle [MFC01].*

On détermine l'ensemble des IBCIs, et on calcule les ordonnancements pour toutes les tâches de toutes les applications et pour tous les IBCIs. Mok et al. montrent que si les

échéances de l'ensemble des tâches du système sont respectées pour tous les ordonnancements ainsi calculés, le STREC est ordonnançable.

Theorem 1 *Pour une application à ordonnancement à priorité fixes et dont la charge de travail est constituée de tâche ayant des échéances contraintes, l'application est ordonnançable si et seulement si son premier travail est ordonnançable pour tous les IBCIs [MFC01].*

Les contraintes d'applicabilité spécifiques à ce test sont les suivantes :
— il y a deux niveaux de hiérarchie uniquement ;
— les tâches sont périodiques ;
— les tâches ne sont pas synchrones ;
— l'ordonnancement est préemptif ;
— l'environnement d'exécution est mono-processeur ;
— l'ordonnancement est hors-ligne au niveau global et en-ligne à priorités fixes au niveau local ;
— les tâches sont indépendantes.

5.2.2 Les méthodes d'analyse compositionnelle

L'approche compositionnelle permet l'analyse de STRECs à ordonnancement hiérarchiques constitués d'applications dont on ne connaît pas le fonctionnement interne.

Le principe général par l'exemple

Le principe de ce type d'analyse est de ramener le problème de l'analyse d'ordonnançabilité d'un ensemble d'applications diverses en un unique problème, identique pour toutes les applications. Pour ce faire, on abstrait les applications par des tâches, et on analyse chacune des applications comme si leur charge de travail était uniquement composés de tâches.

On ramène ainsi un problème d'analyse d'un STREC, contenant m applications organisées selon $\mathbf{n}$ niveaux de hiérarchie, à m problèmes d'analyse d'une application dont la charge de travail n'est constituée que de tâches.

Supposons que l'on applique un méthode d'analyse compositionnelle à l'exemple présenté dans la figure 5.1. La première étape consiste à calculer la quantité de ressources minimales requise par les applications réalisant un ordonnancement local uniquement, i.e., les applications B, C et D.

Les quantités de ressources minimales requises par les applications C et D sont alors utilisées pour ne plus considérer ces deux application que comme des tâches.

Du point de vue de l'analyse à cette étape, les applications C et D sont désormais considérées comme des tâches temps-réel et l'application A ne réalise donc plus qu'un ordonnancement local. On calcule alors la quantité de ressources qu'elle requière, de façon identique à l'analyse précédemment appliquée à C et D.

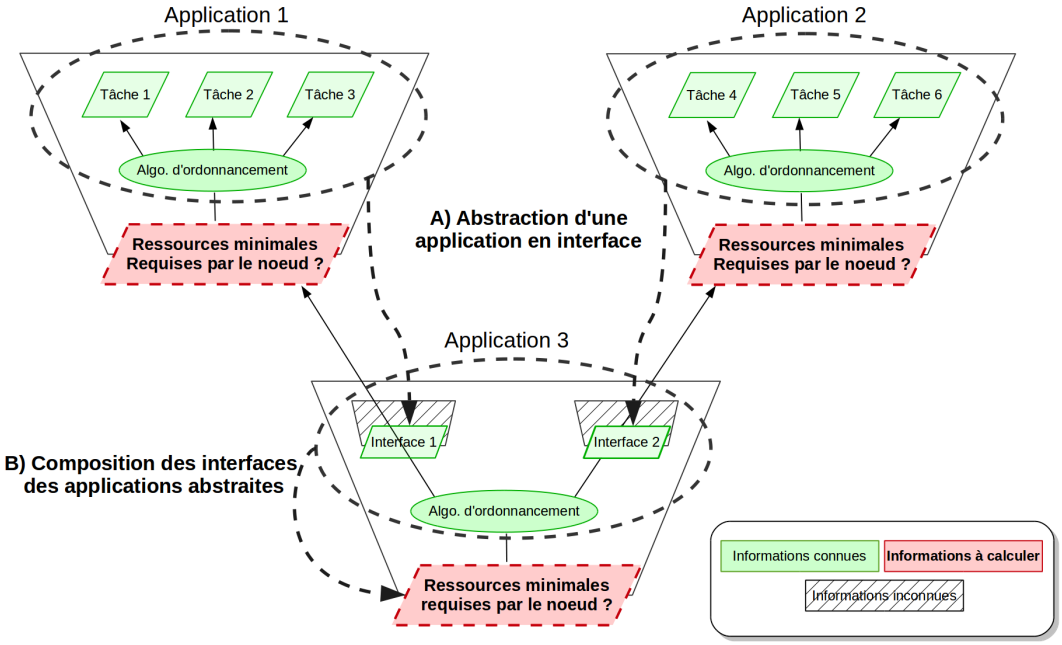

FIGURE 5.2 – Les deux principes de l'analyse compositionnelle : l'abstraction et la composition d'applications.

Les deux applications A et B n'étant plus considérées que comme des tâches, c'est désormais l'application R qui ne réalise plus qu'un ordonnancement local.

L'application R peut donc être analysée avec de la même méthode d'analyse que celle utilisée pour toutes les autres applications du système.

Après avoir expliqué le principe général de l'analyse compositionnelle, nous allons présenter les deux principes sur lesquels elle repose.

Abstraction et composition d'applications

L'analyse peut être résumée selon deux principes présentés dans la figure 5.2.2 : l'abstraction d'applications en interfaces temps-réel et la composition d'applications abstraites en interfaces.

Le premier principe permet d'abstraire les applications dont la charge de travail n'est constituée que de tâches afin de pouvoir les considérer comme des tâches pour la suite de l'analyse. Cette abstraction consiste à calculer la quantité de ressource minimale requise

par une application et à la modéliser selon un *modèle de ressource*.

Définition 5.8 ***(Modèle de ressources temps-réel).*** *Modèle de ressources temps-réel spécifie la quantité de ressource temps-réel minimale qu'une application alloue à une des applications de sa charge de travail, dans un système à ordonnancement hiérarchique [LLS07] [MFC01].*

Un modèle de ressources temps-réel permet de spécifier des *interfaces temps-réel*.

Définition 5.9 ***(Interface temps-réel).*** *Une interface temps-réel abstrait les contraintes temporelles des éléments constituant la charge de travail d'une application comme une unique contrainte temporelle, sans révéler ses informations internes comme sa charge de travail ou son algorithme d'ordonnancement [LLS07].*

Une fois que toutes les applications d'une charge de travail sont abstraites sous la forme d'*interfaces temps-réel*, on réitère la même analyse, mais pour le niveau $n+1$ de l'architecture hiérarchique. Les applications abstraites sont alors considérées comme des tâches et analysées comme des boîtes noires. On parle alors de composition d'application abstraites.

Définition 5.10 ***(Composition d'applications abstraites).*** *Composer des applications, du point de vue de l'analyse compositionnelle revient à combiner différentes applications comme une seule application de plus haut niveau, à l'aide de leurs interfaces, tout en préservant les propriétés temporelles de ces derniers [LLS07].*

Les étapes A et B, sont donc équivalentes, et les deux applications de la charge de travail de l'application 3 sont alors traitées comme des tâches lors du calcul de la quantité de ressources minimale qu'elle requière.

L'analyse d'ordonnancement compositionnelle peut être définie comme l'utilisation conjointe des deux principes exposés ci-dessus.

Définition 5.11 ***(Analyse d'ordonnancement compositionnelle).*** *L'analyse d'ordonnancement compositionnelle est un type d'analyse basé sur l'abstraction d'applications en interfaces et la composition de ces interfaces [LLS07].*

Interfaces temps-réel et modèles de ressource

Il existe plusieurs types d'interface permettant de modéliser des contraintes temporelles différentes. Le type d'interface est déterminé par son modèle de ressource.

Il existe de multiples modèles de ressources permettant d'abstraire les contraintes de temps d'une application en interface temps-réel. Par exemple, Easrawan et al. [EAL07] utilisent le modèle de ressource périodique [SL03], correspondant à l'abstraction d'une application en tâche périodique. Ce modèle définit une interface temps-réel Γ_i d'une application i, comme un couple (Π_i, Θ_i) avec $\Theta_i < \Pi_i$, où Θ_i exprime la quantité de ressources

disponibles pour i toutes les Π_i unités de temps . La capacité de la ressource $U_{\Gamma i}$ est définie comme $U_{\Gamma i} = \Theta_i/\Pi_i$.

Le problème de la composition d'interfaces est le suivant : soit l'application $R = (\Pi_R, \Theta_R)$ constitué de deux applications A et B avec un algorithme d'ordonnancement R, le problème de composition consiste à trouver une interface périodique optimale (Π_R, Θ_R). L'approche consiste à déterminer l'ordonnancement interfaces périodiques de Γ_A et Γ_B de A et B, respectivement, et à considérer R comme étant constituée de deux tâches périodiques, c'est-à-dire, $\Pi_R = \Gamma_A, \Gamma_B$. Le problème de composition de R devient alors équivalent au problème d'abstraction d'une application.

Exemple d'analyse compositionelle : le modèle de ressource périodique

Le modèle d'interface utilisé par Easwaran et al. [ELSV09] est le modèle de ressource périodique [SL03].

Dans ce modèle, l'interface $\Gamma_i = (\Pi_i, \Theta_i)$ (avec $\Theta_i < \Pi_i$) de l'application i, exprime l'approvisionnement en ressource de Θ_i unités de temps pour toutes les périodes de Π_i unités de temps. La vérification de l'ordonnancement des applications est réalisée via interface temps-réel exprimée à l'aide du modèle de ressource périodique, puis transformées en tâche périodiques et en utilisant la politique d'affectation des priorités Deadline Monotonic (DM).

Au sein de chaque application i, les tâches contenues dans la charge de travail $\mathcal{T}_i$ sont périodiques et sont également ordonnancées avec DM. Toute tâche $\tau_{i,j} \in \mathcal{T}_i$ est modélisée à l'aide d'un tuple $(T_{i,j}, C_{i,j}, D_{i,j}, O_{i,j}, J_{i,j})$ (respectivement : période, capacité, échéance, offset et gigue).

L'analyse d'ordonnancement de chaque application est basée sur la relation entre la quantité de ressource processeur nécessaire à l'ensemble de tâches de chaque application, d'après leur interface, et la quantité de ressources disponibles.

Les auteurs prouvent la condition d'ordonnançabilité exacte suivante.

Theorem 2 *Une application peut ordonnancer, en respectant son échéance, sa charge de travail $\mathcal{T}_i$ (avec DM) en utilisant un interface Γ_i si et seulement si, pour toutes les tâches $\tau_{i,j} \in \mathcal{T}_i$ et pour tout instant t_x conforme aux conditions suivantes :*

— *$t_x + D_{i,j} - O_{i,j} - J_{i,j} < \mathrm{lcm}_{\tau_{i,j} \in \mathcal{T}_i} T_{i,j}$; et*

— *$t_x = O_i + J_i + xT_i$ avec x un entier positif,*

il existe un instant $t \in]t_x, t_x + D_{i,j} - O_{i,j} - J_{i,j}]$, tel que

$$rf_{\tau_{i,j}}(0,t) \leq sbf_{\Gamma_i}(t) \quad et \quad rf_{\tau_{i,j}}(t_x,t) \leq sbf_{\Gamma_i}(t - t_x)$$

La fonction de requête $rf(t_1, t_2)$ de chaque tâche retourne la quantité de ressource maximale pouvant être requise par une tâche entre les deux instants t_1 et t_2. Cette fonction prend en compte son offset, sa gigue ainsi que l'influence de ces paramètres sur les interférences entre les tâches. Elle est définie comme suit :

$$rf_{\tau_{i,j}}(t_1,t_2) = \sum_{\ell=1}^{j} \left(\left\lfloor \frac{t_2 - O_{i,\ell}}{T_{i,\ell}} \right\rfloor - \left\lfloor \frac{t_1 - O_{i,\ell} - J_{i,\ell}}{T_{i,\ell}} \right\rfloor \right) C_{i,\ell}$$

La fonction de borne d'approvisionnement (*supply bound function*) correspond à la quantité minimale de ressource processeur garantie pour toute intervalle de longueur t unités de temps. Dans le cas du modèle de ressource périodique, on l'appelle approvisionnement de ressource pire-cas (*worst case resource supply*) [ELSV09] [SL03].

La fonction de borne d'approvisionnement sbf est définie comme suit.

$$sbf_{\Gamma_i}(t) = \left\lfloor \frac{t}{\Pi_i} \right\rfloor \Theta_i + \max(0, t - (\Pi_i - \Theta_i) - \left\lfloor \frac{t}{\Pi_i} \right\rfloor \Pi_i) \,.$$

L'abstraction des applications en interfaces est réalisée par la sélection d'une valeur de Π et la recherche du Θ minimal satisfaisant la condition d'ordonnançabilité. Comme les applications sont ordonnancées avec *Deadline Monotonic*, la transformation de chaque interface en tâche périodique et un nouvelle réalisation de cette méthode d'abstraction permet la réalisation de composition d'interfaces (cf. figure 5.2.2).

Les contraintes d'applicabilité spécifiques à ce test de faisabilité sont les suivantes :
- au sein d'une même application, les tâches sont harmoniques ;
- tous les ordonnanceurs sont préemptifs ;
- l'environnement d'exécution est mono-processeur ;
- les échéances des tâches et applications sont contraintes, i.e., inférieures à leur périodes ;
- l'ordonnancement est hors-ligne au niveau parent (priorités fixe DM) ;
- les priorités peuvent être hors-ligne ou en-ligne au niveau local ;
- les applications sont indépendantes ;
- les charges de travail des applications peuvent utiliser des données partagées si elles sont constituées de tâches uniquement.

On peut noter que ces contraintes d'applicabilité ne donnent pas de limite au nombre de niveaux de hiérarchie.

5.3 Modélisation des patrons de conception architecturaux pour les STRECs mono-processeurs à ordonnancement hiérarchique

Dans cette partie, nous présentons le modèle des patrons de conception permettant la sélection de tests de faisabilité pour les STRECs à ordonnancement hiérarchique. Nous introduisons les concepts de nœud d'association et de graphe de contraintes.

5.3.1 Proposition de nœuds d'association

L'objectif de cette partie est d'exposer la façon dont on explicite la relation entre les modèles d'architecture de systèmes temps-réel et les tests de faisabilité spécifiques aux systèmes à ordonnancement hiérarchique. Nous rappelons que, pour être applicable, un

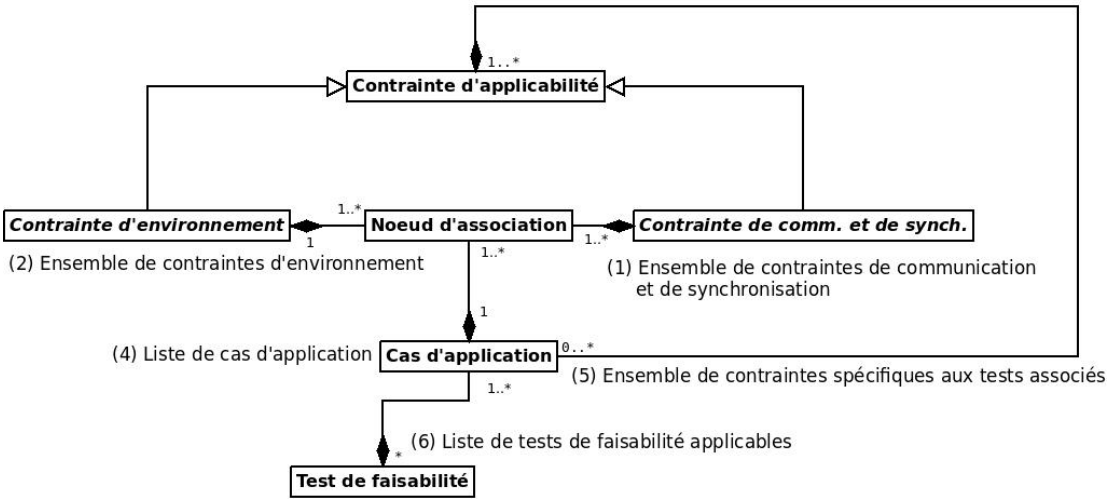

FIGURE 5.3 – Modèle de nœud d'association entre des tests de faisabilité et un ensemble de modèles d'architectures.

test de faisabilité suppose que le système qu'il analyse respecte un ensemble d'hypothèses appelées contraintes d'applicabilité [GSP+11b].

Nous avons défini deux types d'ensembles de contraintes : les *ensembles d'environnement* et les *ensembles de communication et de synchronisation.*

Comme les différentes applications d'un système à ordonnancement hiérarchique peuvent être analysées indépendamment, notamment avec l'approche compositionnelle, le modèle d'association présenté dans la partie 4.2 n'est plus suffisant pour la modélisation de nos patrons. Ce modèle ne permet pas de distinguer les différentes applications du système et donc de les traiter de façon distincte.

Pour pallier ce problème, nous définissons le concept de nœud d'association.

Définition 5.12 ***(Nœud d'association).*** *Un nœud d'association est constitué d'un ensemble de contraintes d'environnement, d'un ensemble de contraintes de communication et de synchronisation entre les tâches et/ou applications ainsi qu'une d'une liste de tests de faisabilité.*

Un nœud d'association a pour objectif d'expliciter des contraintes portant sur une application uniquement (et non plus sur tout le système), ainsi que de référencer les autres nœuds d'association auxquels les applications constituant sa charge de travail doivent être conformes. La figure 5.3 donne une représentation graphique de notre modèle d'association. Une nœud d'association comprend :

— un ensemble de contraintes d'environnement,
— un ensemble de contraintes de communication et de synchronisation,
— un ensemble de cas d'application,
— un ensemble de contraintes d'applicabilité associées au cas d'application

— un ensemble de tests de faisabilité.

5.3.2 Proposition de graphes de contraintes

Nous définissons maintenant la notion de *graphe de contraintes.*

Un graphe de contraintes modélise un patron de conception architectural, i.e., l'association entre un ensemble de STRECs analysables et des tests de faisabilité permettant l'analyse de leur d'ordonnançabilité.

Un graphe de contraintes est constitué d'un ensemble fini de nœuds d'associations et d'un ensemble d'arcs orientés.

Définition 5.13 ***(Graphe de contraintes).*** *Le graphe de contraintes est un couple* $G = (N, A)$*, où :*

- N *est un ensemble fini de nœuds d'association,*
- $A \in N \times N$ *est l'ensemble des arcs orientés du graphe. Les arcs permettent de hiérarchiser les nœuds d'association afin de spécifier des architectures de nœuds utiles à la modélisation des patrons.*
- *Si* $N = \emptyset$ *alors le graphe est vide.*

Nous définissons les cinq types de nœuds d'association suivants : prédécesseur, successeur, racine, feuille et nœud intermédiaire.

Définition 5.14 ***(Prédécesseur au sein du graphe de contraintes).*** *Soit* $G = (N, A)$ *un graphe de contraintes et deux nœuds d'association* $N_1 \in N$ *et* $N_2 \in N$*. Le nœud d'association* N_1 *est le prédécesseur du nœud d'association* N_2*, si et seulement si*

- $\exists\ A_{1,2} \in A$ *tel que* $A_{1,2} = (N_1, N_2)$*.*

Définition 5.15 ***(Successeur au sein du graphe de contraintes).*** *Soit* $G = (N, A)$ *un graphe de contraintes et deux nœuds d'association* $N_1 \in N$ *et* $N_2 \in N$*. Le nœud d'association* N_2 *est le successeur du nœud d'association* N_1*, si et seulement si*

- $\exists\ A_{1,2} \in A$ *tel que* $A_{1,2} = (N_1, N_2)$*.*

Définition 5.16 ***(Racine du graphe de contraintes).*** *Soit* $G = (N, A)$ *un graphe de contraintes. Une racine du graphe de contraintes est :*

- *tout nœud d'association* $N_r \in N$ *n'ayant pas de prédécesseur ou*
- *tous les nœud d'association* $N_i \in N$ *si tous les nœuds ont un prédécesseur.*

Définition 5.17 ***(Feuille du graphe de contraintes).*** *Soit* $G = (N, A)$ *un graphe de contraintes. Une feuille du graphe de contraintes est :*

- *tout nœud d'association* $N_f \in N$ *n'ayant pas de successeur.*

Définition 5.18 ***(nœud intermédiaire du graphe de contraintes).*** *Soit $G = (N, A)$ un graphe de contraintes. Un nœud intermédiaire du graphe de contraintes est tout nœud du graphe n'étant ni racine, ni feuille.*

- *tout nœud d'association $N_f \in N$ ayant au moins un successeur et au moins un prédécesseur.*

Un patron de conception supportant les STRECs à ordonnancement hiérarchique est donc modélisé par un graphe de nœuds d'association appelé graphe de contraintes. L'objectif de la partie suivante est d'exposer l'utilisation de ce nouveau modèle dans l'objectif de la sélection de tests de faisabilité.

5.3.3 Modélisation des patrons en tant que graphes de contraintes

L'objectif de cette partie est de décrire l'utilisation des graphes de contraintes pour la modélisation des patrons de conception et de définir la notion de conformité d'un STREC à ces derniers.

Comme nous le précisions précédemment, avec l'inclusion de multiples environnements d'exécution, la méthode d'association présenté précédemment (cf. chapitre 4) n'est plus valide. On expose ici comment déduire d'un patron de conception, modélisé à l'aide d'un graphe de contraintes, un ensemble de tests de faisabilité applicables.

Nous avons identifié deux types de tests de faisabilité distincts : ceux effectuant une analyse globale, et ceux réalisant une analyse compositionnelle (cf. partie 5.2).

Dans le cas de l'analyse globale, les contraintes de faisabilité relatives aux tests de faisabilité restreignent l'ensemble des éléments composant le système à analyser. On aurait donc pu, si nous nous étions bornés à cette seule approche, conserver un modèle d'association proche de celui défini dans le chapitre 4. En effet, la distinction entre application et tâche est claire dans ce cas, et les charges de travail sont généralement constituées uniquement d'un ensemble de l'une ou de l'autre.

Si l'on souhaite pouvoir prendre en compte l'analyse compositionnelle, il est donc nécessaire de pouvoir modéliser, à l'aide de nos patrons, à la fois des systèmes comportant un nombre de niveaux connu et des système comportant un nombre de niveaux inconnu.

Faire cette distinction est réalisable grâce à différents ensembles de contraintes d'environnement.

Les ensembles de contraintes d'environnement peuvent être classés selon s'ils peuvent être utilisé lors de la spécification de nœuds feuilles ou non.

Les ensembles de contraintes d'environnement dédiés à la spécification de nœuds feuilles caractérisent des applications du système ayant une charge de travail uniquement constituée de tâches.

Les ensembles de contraintes d'environnement dédiés à la spécification de nœuds intermédiaires caractérisent des applications du système ayant une charge de travail constituées de tâches, d'applications ou d'un mélange des deux.

Pour modéliser une association de tests de faisabilité avec un ensemble de systèmes à deux niveaux de hiérarchie, le graphe de contraintes comprendra un ou plusieurs nœuds

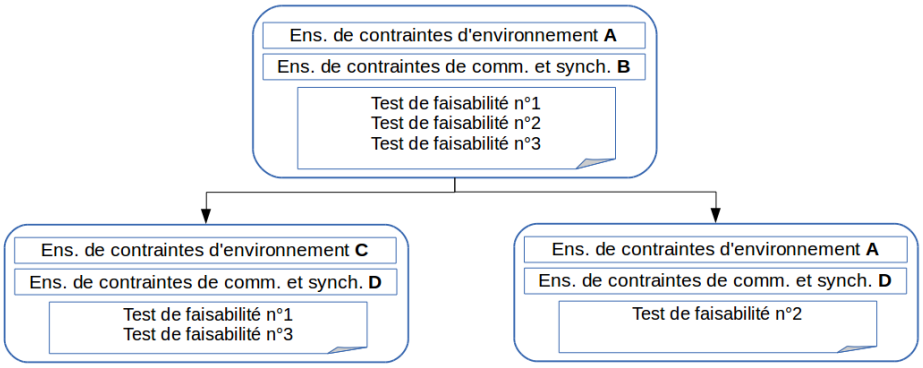

FIGURE 5.4 – Grille de lecture de la représentation graphique des modèles de patrons sous la forme de graphes de contraintes.

racines constitués d'un couple d'ensemble de contraintes comprenant un ensemble de contraintes d'environnement de type intermédiaire et d'un ensemble de contraintes de communication et de synchronisation. L'ensemble des nœuds successeurs de ces racines seront des feuilles et ne pourront donc pas avoir de nœud successeurs.

Par ailleurs, les systèmes à nombre de niveaux de hiérarchie quelconque sont modélisés par un graphe comprenant au moins un cycle, i.e., comprenant un nœud étant son propre successeur ou le successeur d'un de ses successeurs. Ce cycle permet d'étendre l'ensemble des systèmes potentiellement conformes à un nombre de niveaux illimité.

Avant de définir la notion de conformité d'un STREC à un patron modélisé à l'aide d'un graphe de contraintes, nous fournissons une représentation graphique des graphes de contraintes modélisant les patrons.

5.3.4 Grille de lecture de la spécification de patrons architecturaux adaptés aux systèmes à ordonnancement hiérarchique

Dans cette sous-partie, nous proposons une notation graphique pour la spécification de patrons architecturaux. La figure 5.4 donne la grille de lecture d'un schéma modélisant un patron. Chaque rectangle représente un ensemble de contraintes spécifié dans ce manuscrit. Les rectangles à bords arrondis modélisent un nœud d'association. Chaque nœud d'association contient les deux ensembles de contraintes le constituant et la liste de tests de faisabilité associés. Les flèches noires modélisent un arc orienté entre deux nœuds d'association.

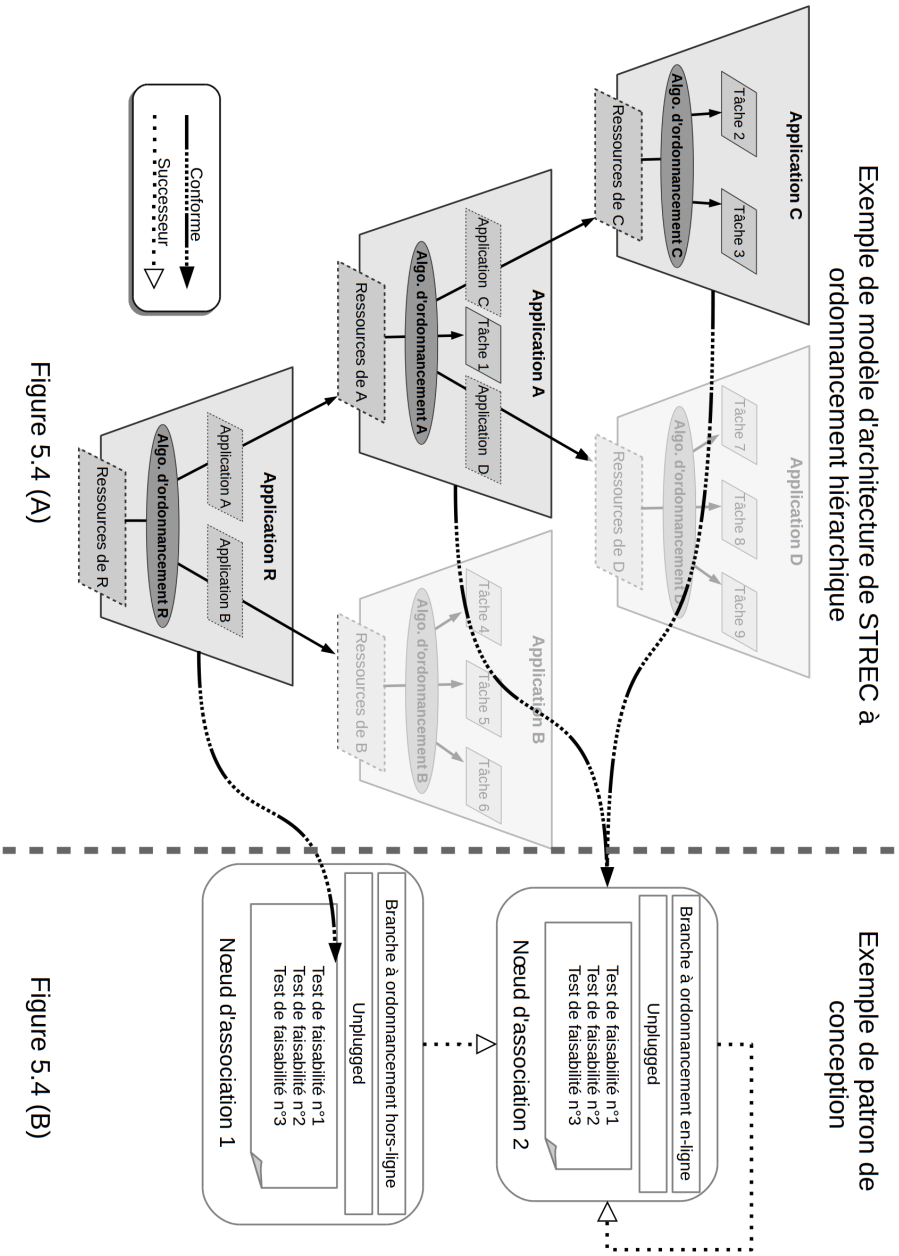

FIGURE 5.5 – Conformité d'un système à ordonnancement hiérarchique (A) avec un patron de conception modélisé par un graphe de contraintes (B).

5.3.5 Conformité d'un STREC à un patron de conception architectural

La figure 5.5 schématise la notion de conformité à un patron de conception défini comme un graphe de contraintes.

Définition 5.19 ***(Conformité d'un STREC à un patron de conception architectural).*** *Un STREC à ordonnancement hiérarchique est conforme à un patron de conception si et seulement si,*

- *l'application réalisant l'ordonnancement global est conforme aux contraintes d'une des racines du graphe de contraintes,*
- *toute application A du modèle d'architecture est conforme à un nœud P_A du graphe de contrainte modélisant le patron et*
- *chacune des applications de la charge de travail de A (s'il en existe) sont conformes à un nœud successeur de P_A.*

La figure 5.5 illustre ce concept. La figure 5.5(A) comprend le modèle d'architecture présenté en partie 5.1, tandis que la figure 5.5(B) propose un exemple patron de conception modélisé par un graphe de contraintes constitué de deux nœuds d'association numérotés de 1 et 2.

Afin que le STREC soit conforme au patron, il est nécessaire que :
— l'application R respecte les contraintes du nœud d'association racine, i.e., le nœud 1 ;
— l'application A respecte les contraintes du successeur du nœud 1, i.e., le nœud 2 ;
— l'application C respecte les contraintes du successeur du nœud 2, i.e., le nœud 2.

.

La partie suivante discute de la sélection des tests de faisabilité dans le cas des STRECs à ordonnancement hiérarchique conformes à un parton modélisé comme un graphe de contraintes.

5.3.6 Patron de conception et sélection de tests de faisabilité

Le fait qu'un patron associe un STREC lui étant conforme à de multiples méthodes d'analyse compositionnelle peut poser problème lors de la sélection des tests de faisabilité.

Comme nous le mentionnons dans la partie 5.2.2, l'analyse compositionnelle se repose sur l'abstraction des applications à l'aide d'interface temps-réel. Pour que l'analyse puisse être menée, il est nécessaire que toutes les applications puissent êtres abstraites à l'aide d'interfaces reposant sur le même modèle de ressources. Par exemple, si une des applications doit être abstraite avec le modèle de ressource à délai borné [MFC01] uniquement et une autre avec le modèle de ressource périodique [ELSV09], la composition des deux interfaces ne sera pas possible.

Il est donc nécessaire, pour qu'un test de faisabilité soit applicable, que le même test ait été sélectionné pour tous les nœuds du modèle d'architecture.

Nous allons désormais décrire les ensembles de contraintes d'environnement que nous proposons, avant de présenter les trois exemples de patrons de conception spécifiques aux systèmes à ordonnancement hiérarchique développés lors de cette thèse.

TABLE 5.1 – Ensemble de contraintes d'environnement d'un nœud feuille à ordonnancement hors-ligne.

leaf-S1 :	Il y a un unique processeur.	Cst1
leaf-S2 :	La politique d'ordonnancement doit être hors-ligne.	Cst24
leaf-S3 :	Le niveau de préemptivité de l'ordonnanceur doit être explicite.	Cst3
leaf-S4 :	L'ordonnanceur ne doit pas utiliser de quantum.	Cst4
leaf-S5 :	La charge de travail est uniquement constituée de tâches.	Cst25

TABLE 5.2 – Ensemble de contraintes d'environnement d'un nœud feuille à ordonnancement en-ligne.

leaf-D1 :	Il y a un unique processeur.	Cst1
leaf-D2 :	La politique d'ordonnancement peut être : *Earliest Deadline First.*	Cst26
leaf-D3 :	Le niveau de préemptivité de l'ordonnanceur doit être explicite.	Cst3
leaf-D4 :	L'ordonnanceur ne doit pas utiliser de quantum.	Cst4
leaf-D5 :	La charge de travail est uniquement constituée de tâches.	Cst25

5.4 Proposition de quatre ensembles de contraintes d'environnement pour le cas hiérarchique

Cette partie est consacrée à la proposition de quatre ensembles de contraintes d'environnement, spécifiques à l'analyse de STRECs à ordonnancement hiérarchique.

Ces ensembles de contraintes permettent de modéliser les patrons de conception architecturaux utiles à la sélection des quatre tests de faisabilité proposés par : Mok et al. [MFC01], Davis et al. [DB05] et Easwaran et al. [EAL07, ELSV09]. Dans [MFC01] et [DB05], deux tests de faisabilité utilisant une approche globale sont proposés. Les deux autres tests présentés dans [EAL07, ELSV09] réalisent des analyses compositionelles.

Nous proposons quatre nouveaux ensembles de contraintes d'environnement. Nous les nommons : nœud feuille à ordonnancement hors-ligne (cf. tableau 5.1), nœud feuille à ordonnancement en-ligne (cf. tableau 5.2), nœud intermédiaire à ordonnancement hors-ligne (cf. tableau 5.4), nœud intermédiaire à ordonnancement en-ligne (cf. tableau 5.3).

On peut noter que les ensembles de contraintes présentés ici varient peu. Cependant, le nombre de combinaisons potentielles d'ensembles de contraintes, et donc de nœuds d'association, est important.

En effet, seules les deux contraintes relatives à la composition de la charge de travail de l'application et à l'algorithme d'ordonnancement varient. Les trois contraintes Cst1, Cst3 et Cst4 sont communes aux quatre ensembles de contraintes.

TABLE 5.3 – Ensemble de contraintes d'environnement d'un nœud intermédiaire à ordonnancement en-ligne.

node-D1 :	Il y a un unique processeur.	Cst1
node-D2 :	La politique d'ordonnancement peut être : *Earliest Deadline First.*	Cst26
node-D3 :	Le niveau de préemptivité de l'ordonnanceur doit être explicite.	Cst3
node-D4 :	L'ordonnanceur ne doit pas utiliser de quantum.	Cst4
node-D5 :	La charge de travail peut être constituée de tâches et d'applications.	Cst27

TABLE 5.4 – Ensemble de contraintes d'environnement d'un nœud intermédiaire à ordonnancement hors-ligne.

node-S1 :	Il y a un unique processeur.	Cst1
node-S2 :	La politique d'ordonnancement doit être hors-ligne.	Cst24
node-S3 :	Le niveau de préemptivité de l'ordonnanceur doit être explicite.	Cst3
node-S4 :	L'ordonnanceur ne doit pas utiliser de quantum.	Cst4
node-S5 :	La charge de travail peut être constituée de tâches et d'applications.	Cst27

5.5 Exemples de patrons de conception pour les STRECs mono-processeurs à ordonnancement hiérarchique

L'objectif de cette partie est de présenter trois exemples de patron de conception architecturaux spécifiques à l'analyse de STRECs mono-processeurs à ordonnancement hiérarchique.

Les trois patrons sont les suivants : le patron *ARINC653*, le patron *two-levels-independent-applications* et le patron *n-levels-independent-workloads*. Le patron ARINC653 est directement inspiré du standard pour l'avionique du même nom. Ce dernier suppose exactement deux niveaux de hiérarchie et des applications indépendantes, du point de vue de l'ordonnancement, au niveau global. Au niveau local, les tâches peuvent interagir à l'aide de communication de type *Time-Triggered* ou *Ravenscar*. Le patron *two-levels-independent-applications* spécifie des STRECs à deux niveaux d'ordonnancement hiérarchique exactement. Il permet d'utiliser de l'ordonnancement hors-ligne au niveau local. La politique d'ordonnancement au niveau global est en-ligne. Enfin, le patron *n-levels-independent-workloads* autorise l'utilisation d'un nombre de niveaux de hiérarchie quelconque, mais n'autorise pas de communication, que ce soit entre des tâches ou entre des applications.

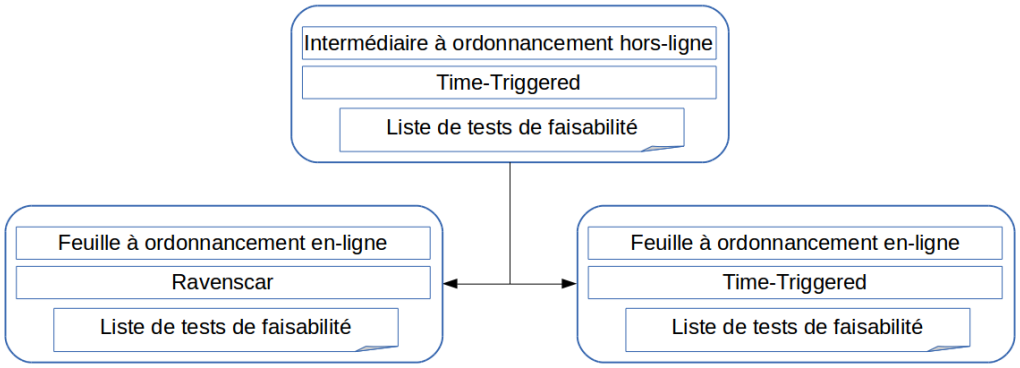

FIGURE 5.6 – Représentation graphique du patron *ARINC653*.

5.5.1 Le patron ARINC653

Contexte : Les systèmes conformes à ce patron peuvent être conformes aux principes considérés dans le standard ARINC653. Cependant, ce patron ne représente pas tous les systèmes pouvant être conformes à ARINC653 : nous nous restreignons aux systèmes mono-processeurs.

Ce patron est constitué de trois nœuds. Le nœud racine du graphe est composé d'un environnement d'exécution à ordonnancement hors-ligne et permettant des communications de type *Time-Triggered* entre les différentes applications. Ce dernier a deux successeurs, représentant les deux nœuds d'association auxquels les applications ordonnancées par le niveau global doivent être conformes pour que le système soit analysable avec les tests associés à ce patron.

Problème : Les systèmes à ordonnancement hiérarchique déployés sur un unique processeur peuvent être analysés de multiples façons. Deux approches différentes (globale et compositionnelle) sont présentées dans la partie 5.2. De nombreux tests de faisabilité ont spécialement été conçus pour ce type de système bien précis. Les critères de performance nécessitant d'être évalués pour ce type de système sont multiples et dépendent de sa partie logicielle.
Solution : Le graphe de contraintes du patron de conception ARINC653 est présenté dans figure 5.6. Les ensembles de contraintes utilisés sont récapitulés dans les tableaux 4.4, 4.5, 5.2 et 5.4.

5.5.2 Le patron de conception two-levels-independent-applications

Contexte : On considère ici des STRECs à deux niveaux de hiérarchie permettant l'utilisation de protocoles de communication et de synchronisation entre les tâches du niveau

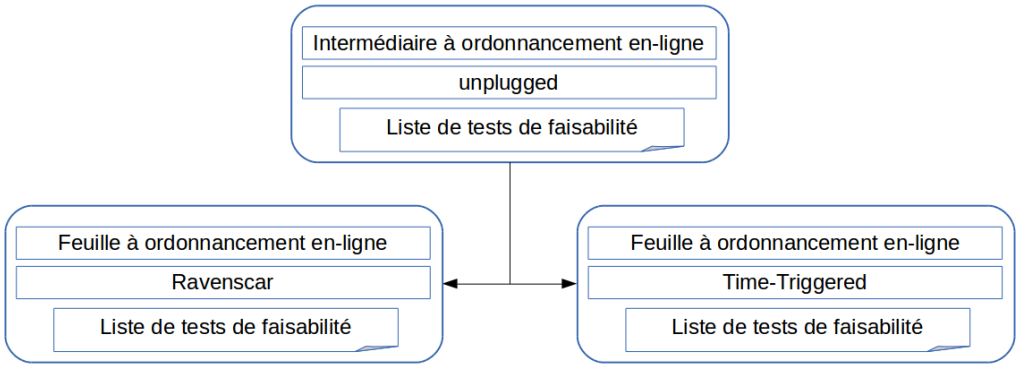

FIGURE 5.7 – Représentation graphique du patron *two-levels-independent-applications*

local d'une même application, mais pas entre les applications. La principale différence entre ce patron et le patron ARINC653 est que le protocole d'ordonnancement du niveau global est en-ligne.
Problème : Ce type de système est celui pour lequel les tests de faisabilité sont les plus nombreux. Ils utilisent une approche globale capable de traiter ce type de système. Il est possible d'évaluer un grand nombre de critères de performance. La plupart des tests pris en considération lors de ces travaux autorisent l'utilisation d'un unique protocole de communication et de synchronisation. La difficulté réside donc dans le fait de vérifier que tous les nœuds de graphe modélisant le STREC à ordonnancement hiérarchique peuvent bien être analysés à l'aide d'un même test de faisabilité.
Solution : Le graphe de contraintes du patron de conception *two-levels-independent-applications* est présenté figure 5.7. Les ensembles de constraintes utilisés sont récapitulés dans les tableaux 5.3, 5.2, 4.4 et 4.5.

5.5.3 Le patron de conception n-levels-independent-workloads

Contexte : On considère ici des STRECs avec un nombre de niveaux de hiérarchie quelconque, mais n'utilisant pas de protocoles de communication ou de synchronisation entre les tâches et les applications. Ces hypothèses sont celles assumées par les premiers tests de faisabilité réalisant de l'analyse compositionnelle. On peut noter que ce patron autorise l'utilisation d'ordonnancement hors-ligne et en-ligne.
Problème : Les systèmes à ordonnancement hiérarchique à nombre niveaux de hiérarchie quelconque requiert l'utilisation de l'approche d'analyse compositionnelle. La principale difficulté pour sélectionner des tests de faisabilité leur étant applicables, est de vérifier que toutes les applications peuvent être abstraites selon le même modèle de ressources.
Solution : Le graphe de contraintes du patron de conception *n-levels-independent-workloads* est présenté figure 5.8. Les ensembles de constraintes utilisés sont récapitulés dans les ta-

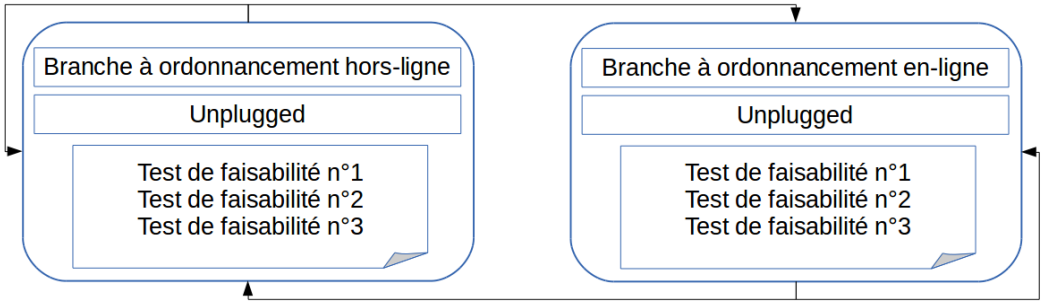

FIGURE 5.8 – Représentation graphique du patron *n-levels-independent-workloads*.

bleaux 5.4, 5.3 et 4.7.

5.6 Étude de cas ARINC653

Dans cette partie, nous présentons une étude de cas. Cette étude de cas a été proposée par l'équipe de recherche du LaSIGE [ACPR13]. Elle décrit l'architecture de contrôle d'un satellite, conçue de façon conforme au standard ARINC653.

Cette architecture de contrôle est composée de quatre applications responsables des tâches suivantes : le contrôle de l'orbite et de l'altitude, la gestion des erreurs et la récupération, la télémétrie et la télécommande et enfin l'application spécifique à la mission (scientifique ou industrielle).

5.6.1 Description de l'architecture de contrôle d'un satellite

Nous présentons, ici, les quatre applications composant une architecture de contrôle d'un satellite

Système de contrôle d'orbite et d'altitude (AOCS)

L'orientation d'un satellite est cruciale, que ce soit pour les télécommunications, l'observation terrestre, ou encore des missions de recueil de données astronomiques.

Le système de contrôle d'orbite et d'altitude (Attitude and Orbit Control System : AOCS) est responsable de la gestion de la hauteur d'orbite et garantit que le satellite est orienté de façon à remplir ses objectifs. Cette partie du système est constituée de capteurs et d'actuateurs redondants, tels que des capteurs d'étoiles et terrestres, gyroscopes, volants d'inertie (M-WHEELS), volants à réaction (R-WHEELS), magnéto coupleurs, propulseurs, de panneaux solaires et d'un mécanisme de finition de positionnement. Les points suivants décrivent les fonctionnalités de chaque tâche.

— **Command Actuator** : réalise les commandes définies par la tâche control-law afin de maintenir l'orientation du véhicule spatial.
— **Control Law** : implante les lois de contrôle de base et le maintien de l'orientation de l'engin vers un point donné.
— **RW Data** : l'actuateur *Reaction Wheels* (RW) contrôle les mouvements de l'engin spatial.
— **DSS Data** : le capteur *Digital Sun Sensor* (DSS) surveille l'orientation de l'engin relativement à la position du soleil.
— **Gyro Data** : le capteur *Rate Gyro Sensor* détecte la rotation de l'engin spatial ; ce capteur produit des données de façon sporadique.
— **IRES Data** : le capteur infrarouge *InfraRed Earth Sensor* (IRES) scanne un large champ de fréquences et détecte les signaux de transition Terre/Espace.

Détection des fautes, isolation et récupération (FDIR)

L'application FDIR est responsable de la surveillance du système, afin de détecter efficacement, isoler les erreurs et effectuer l'action de récupération adéquate.
— **Fault Detection** : implante les différents algorithmes de détection d'erreurs dans le système.
— **Fault Isolation** : réalise les actions d'isolation d'erreurs prédéfinies.
— **Fault Recovery** : contient les actions de récupération nécessaires au composant au sein duquel l'erreur à été détectée.

Télémétrie et Télécommande (TMTC)

Une connexion spatiale est un lien de communication entre un engin spatial et un système associé se trouvant au sol, ou dans un autre engin spatial. Le flot de données de base sur ce type de connexion est composé des données de télémétrie (TM) et de télécommande (TC). Les flots de données entrants (TC) et sortants (TM) constituent un canal de communication entre l'engin spatial et les opérateurs au sol. Cette application gère toutes les données envoyées et reçues par l'engin spatial. Ce sous-système gère l'envoi et la réception de toutes les données, y compris les données scientifiques et l'exploitation du satellite.
— **Telecommands** : reçoit les commandes envoyées depuis la terre au satellite. Ces dernières servent à la reconfiguration du satellite au cours de la mission.
— **Telemetry** : envoie des données depuis le satellite vers la terre grâce au flux TM. Les types de données sont variables :
  — *payload data* : données relatives au fonctionnement de l'engin ;
  — *orbit position data* : données décrivant la position actuelle du satellite ;
  — *telecommand reception status (CLCW)* : réponses envoyées au système sur terre, signifiant la bonne réception d'une commande de téléopération ; et
  — *memory dump data* : vidage de la mémoire de l'engin spatial.

Payload

L'application payload implante les fonctionnalités scientifiques et/ou industrielles du satellite. Elle contient donc l'ensemble des fonctions nécessaires à la réalisation de sa mission.

— **Payload Data** : objectifs scientifiques ou industriels de la mission dans sa globalité. (e.g., cartographie de la surface de planètes, communications GPS pour les satellites y étant dédiés, ...).

Nous allons maintenant présenter l'extension de l'algorithme de sélection de tests de faisabilité, tout en l'illustrant à l'aide de cette étude de cas.

5.7 Proposition d'un algorithme de sélection de tests de faisabilité supportant les STRECs à ordonnancement hiérarchique

Dans cette partie, nous présentons un algorithme de sélection de tests de faisabilité pouvant être utilisé pour des systèmes à ordonnancement hiérarchique.

L'algorithme présenté partie 4.4 est capable de sélectionner les tests de faisabilité applicables à un STREC mono-processeur conforme à un des patrons de conception présentés dans le chapitre 4. Cependant, une des hypothèses supposées lors de la conception de cet algorithme est que l'environnement d'exécution est conforme à l'ensemble de contraintes *uniprocessor* (cf. table 4.2).

Nous proposons, dans cette partie, un algorithme de sélection de tests de faisabilité capable d'analyser la conformité d'un STREC à un patron de conception modélisé comme un graphe de contraintes d'analyse. La figure 5.9 présente cet algorithme constitué de quatre étapes. Il réalise l'analyse de l'environnement d'exécution d'un STREC par construction d'un graphe de contraintes qui sera ensuite comparé aux graphes définies par les patrons. Pour chacune des applications, on vérifie sa conformité à un ensemble de contraintes d'environnement. Ensuite, on applique l'algorithme présenté chapitre 4 à chacune des applications afin de déterminer quel protocole de communication et de synchronisation est utilisé. Des tests de faisabilité adaptés à chaque application sont alors sélectionnés. Enfin, les ensembles de tests de faisabilité sélectionnés sont comparés pour en extraire les tests de faisabilité communs à tous ces ensembles. Les tests de faisabilité ainsi extraits sont applicables au modèle d'architecture en entrée de l'algorithme.

Cet algorithme est constitué de 4 étapes, que nous présentons successivement ici.

5.7.1 Étape 1 : Reconnaissance des environnements d'exécution à partir du modèle d'architecture et construction du graphe de contraintes

Pour chacun des déploiements présents dans le modèle d'architecture exprimé à l'aide de Cheddar ADL, nous déterminons l'environnement d'exécution en vérifiant la conformité du déploiement à l'un des ensembles de contraintes d'environnement que nous avons proposé.

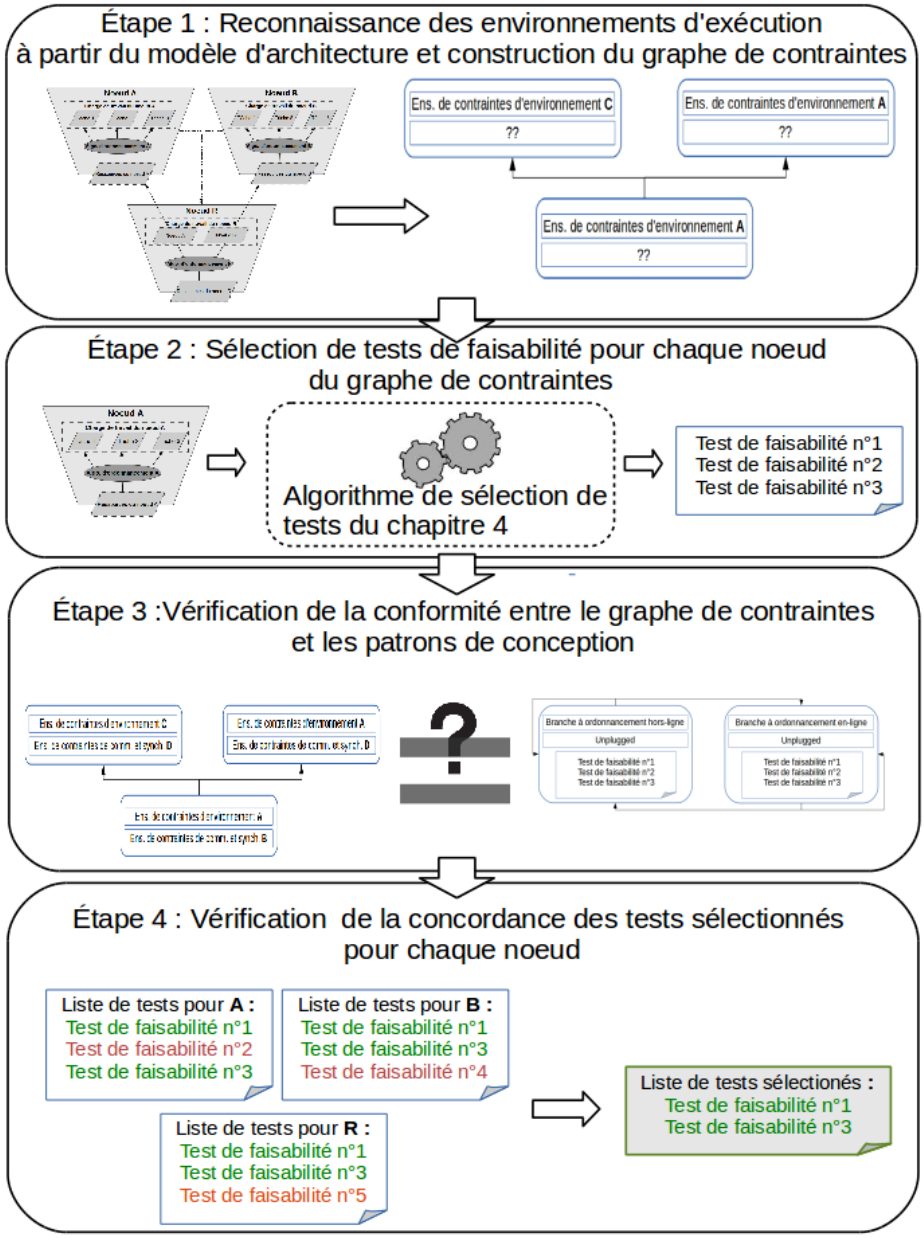

FIGURE 5.9 – Représentation graphique de l'algorithme de sélection de tests de faisabilité adapté aux STRECs à ordonnancement hiérarchique.

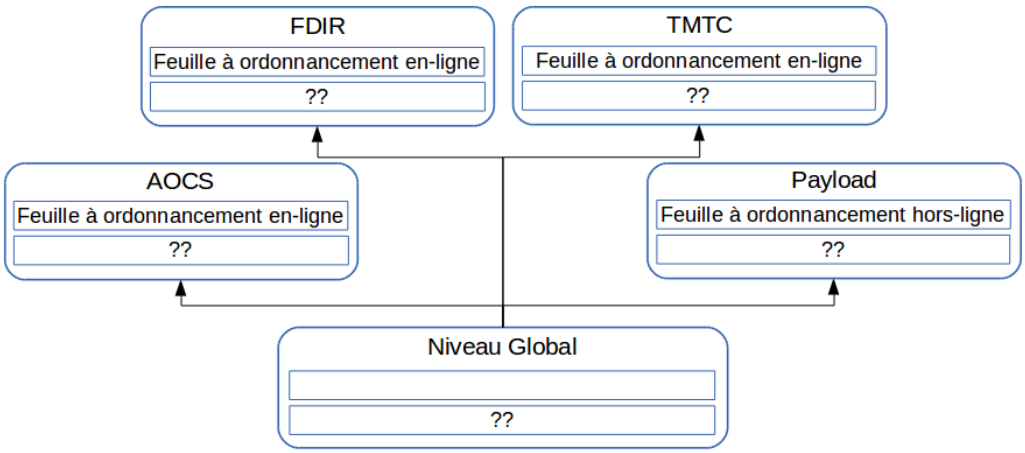

FIGURE 5.10 – Représentation graphique du graphe construit pendant l'application de l'étape 1 à notre étude de cas.

Une fois l'environnement déterminé, on instancie un nœud d'association modélisant le type d'environnement détecté.

Lorsque tous les déploiements ont été vérifiés conformes à un ensemble de contraintes d'environnement, on créé un graphe contenant les différents ensembles de contraintes d'environnement de chaque application et respectant la structure du modèle d'architecture.

L'étude de cas de l'architecture de contrôle d'un satellite comporte cinq applications. Le graphe de contraintes construit est présenté à la figure 5.10. Il comporte trois nœuds contenant un ensemble de contraintes *feuille à ordonnancement en-ligne*, un nœud contenant un ensemble de contraintes *feuille à ordonnancement hors-ligne* et un dernier contenant un ensemble de type *intermédiaire à ordonnancement hors-ligne*.

5.7.2 Étape 2 : Sélection des tests de faisabilité pour chaque nœud du graphe de contraintes

Maintenant que les environnements d'exécution de chaque nœud de l'architecture hiérarchique sont détectés, on applique, aux charges de travail des nœuds, l'algorithme de sélection de tests de faisabilité basé sur l'analyse des ensembles de contraintes de communication et de synchronisation. Il s'agit, ici, d'appliquer l'algorithme décrit dans la partie 4.4. On associe, alors, à chacun des nœuds du graphe de contraintes, la liste de tests de faisabilité sélectionnés. On peut noter que dans le cas où un nœud n'est conforme à aucun patron de conception existant, la liste associée est vide.

Le graphe de contraintes construit lors de l'application de cet algorithme à l'étude de cas est présenté à la figure 5.11. Pour plus de lisibilité, les tests de faisabilité sélectionnés

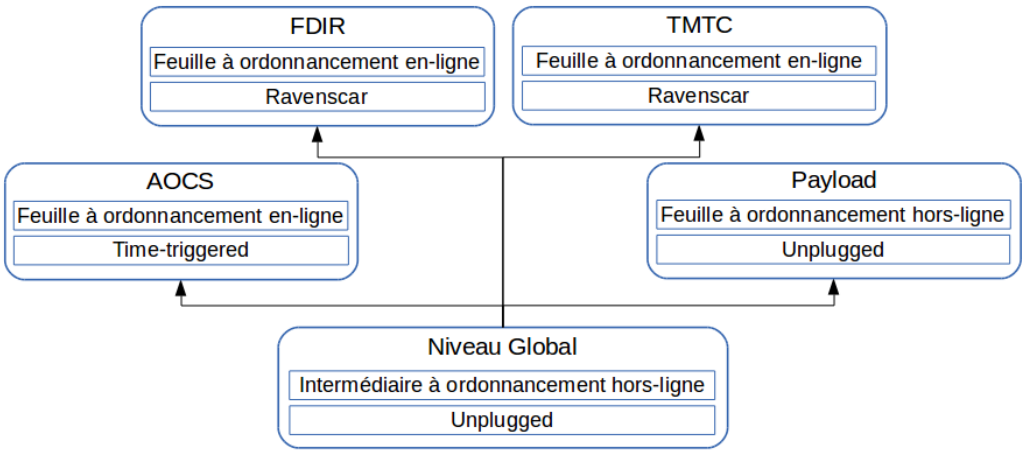

FIGURE 5.11 – Représentation graphique du graphe construit pendant l'application de l'étape 1 après l'application de l'algorithme présenté dans la partie 4.4 à notre étude de cas.

nœuds	Tests de faisabilité sélectionnés
Niveau Global	Mok et al. [MFC01], Easwaran et al. [EAL07] [ELSV09], Davis et al. [DB05]
AOCS	Mok et al. [MFC01]
FDIR	Mok et al. [MFC01], Easwaran et al. [ELSV09]
TMTC	Mok et al. [MFC01], Easwaran et al. [ELSV09]
Payload	Mok et al. [MFC01]

TABLE 5.5 – Tests de faisabilité sélectionnés pour chacun des nœuds de notre étude de cas lors de l'étape 2 de notre algorithme.

pour chacun des nœuds sont listés dans le tableau 5.5.

5.7.3 Étape 3 : Vérification de la conformité entre le graphe de contraintes et les patrons de conception

On vérifie, ensuite, la conformité du graphe de contraintes d'analyse à un patron de conception, selon la définition proposée dans la partie 5.3.5

Dans le cas où le graphe construit n'est conforme à aucun des patrons de conception, le concepteur devra, pour pouvoir sélectionner des tests de faisabilité, modifier son architecture. Sinon, le système est bien conforme à un patron, l'algorithme exécute l'étape suivante pour finaliser la sélection de tests de faisabilité.

5.7.4 Étape 4 : Vérification de la concordance des tests sélectionnés pour chaque nœud

Avant de cette étape, des tests de faisabilité ont été sélectionnés pour chacune des applications du système. On compare une à une les listes de tests de faisabilité sélectionnés pour ne retenir uniquement ceux communs à toutes les listes. Les tests retenus sont ceux applicables au système.

Pour notre étude de cas d'architecture de contrôle de satellite, seul un test de faisabilité est commun à tous les nœuds d'association du graphe construit par l'algorithme : le test de Mok et al. [MFC01] présenté dans la partie 5.2.

5.8 Validation/Expérimentation

Dans cette partie, nous présentons le processus d'évaluation de l'approche présentée dans ce chapitre, sans rentrer dans les détails de la mise en œuvre. De la même façon que pour le chapitre 4, la mise en œuvre est décrite dans le chapitre 6.

5.8.1 Outillage

Cette partie décrit l'outillage implanté afin d'évaluer la contribution présentée dans ce chapitre. Dans un premier temps, nous listons les tests de faisabilité spécifiques aux systèmes à ordonnancement hiérarchique intégrés à Cheddar. Nous présentons ensuite le prototype de l'algorithme de sélection de tests de faisabilité.

Afin de pouvoir évaluer notre proposition de modélisation à l'aide de déploiements, nous avons implanté deux tests de faisabilité spécifiques aux systèmes à ordonnancement hiérarchique. Ces tests correspondent à ceux présentés dans la partie 5.2.2. Ces tests ont été intégrés à Cheddar.

Afin d'évaluer notre approche de sélection de tests de faisabilité adaptée aux systèmes à ordonnancement hiérarchique, nous avons réalisé un second prototype [1] intégré à Cheddar.

Le processus de développement de ce second prototype est également exposé dans le chapitre 6.

Lors de l'évaluation de l'approche exposée dans le chapitre 4, nous avons proposé un générateur de modèles d'architectures. L'ensemble des paramètres de la nouvelle version du générateur reste identique, i.e., le nombre d'éléments architecturaux de chaque type souhaité dans le modèle d'architecture généré. Les types d'éléments déjà présents dans la première version restent disponibles, e.g., *processor*, *core*, *task*, *buffer*, *dependency*, *message*, *resource* et *address_space*. Deux nouveaux concepts ont été ajoutés : les déploiements statiques ainsi que les déploiements dynamiques.

Les propriétés de ces éléments architecturaux sont toujours générées aléatoirement selon une loi normale. Les affectations des tâches et ressources aux déploiements sont également générées aléatoirement.

1. Le prototype est disponible à l'adresse : `http://beru.univ-brest.fr/svn/CHEDDAR/`

Enfin, l'affectation des dépendances est désormais paramétrable selon deux critères : affectation entre deux tâches appartenant au même déploiement ou affectation entre deux tâches appartenant à deux déploiements distincts.

5.8.2 Processus d'évaluation

L'objectif de notre évaluation est de tester la correction, la robustesse et le passage à l'échelle de notre second prototype. Pour ce faire, nous procédons à trois phases de tests. La première consiste à vérifier la détection du non-respect de chacune des contraintes individuellement. Des architectures, non-conformes à chacune des contraintes spécifiées dans le cadre de l'approche présentée dans ce chapitre, sont générées. La seconde phase évalue la sélection de tests de faisabilité pour les études de cas présentées dans les chapitres 4 et 5 de ce manuscrit : le système automobile simplifié, l'architecture de contrôle de *Mars Pathfinder* ainsi que l'architecture de contrôle du satellite exposée en partie 5.6. La troisième et dernière phase consiste en la sélection de tests de faisabilité pour des architectures générées.

5.8.3 Évaluation d'architectures non-conformes à nos patrons

L'objectif de cette expérimentation est de vérifier la correction de l'implantation des contraintes contenues dans les patrons de conception et les cas d'application. Pour ce faire, nous générons un nombre de modèles d'architecture égal à deux fois le nombre de contraintes implantées. Pour chaque contrainte c, une architecture conforme et une non-conforme à c sont créées.

De plus, un ensemble de modèles d'architecture pour lesquels chaque déploiement est conforme à un nœud d'association, mais dont l'arbre de hiérarchie ne correspond à aucun patron a été généré. Nous en avons créé cinq par patron de conception proposé dans ce chapitre, soit quinze au total.

Lors de cette première phase, les évaluations des contraintes ont toutes retourné le résultat attendu. Nos contraintes et la méthode d'analyse de conformité d'un modèle d'architecture à un patron modélisé comme un graphe de contraintes ont donc été correctement implantées.

5.8.4 Évaluation à l'aide d'études de cas

L'objectif de cette évaluation est de vérifier si notre algorithme est capable de sélectionner des tests de faisabilité pour notre étude de cas.

Nous avons, tout d'abord, modélisé l'architecture de contrôle de satellite Puis, de façon analogue au chapitre 4,nous avons appliqué notre algorithme à l'étude de cas présenté dans la partie 5.6.

Modélisation de l'étude de cas à l'aide de Cheddar ADL

La Figure 5.12 fournit la représentation graphique de la modélisation de l'étude de cas

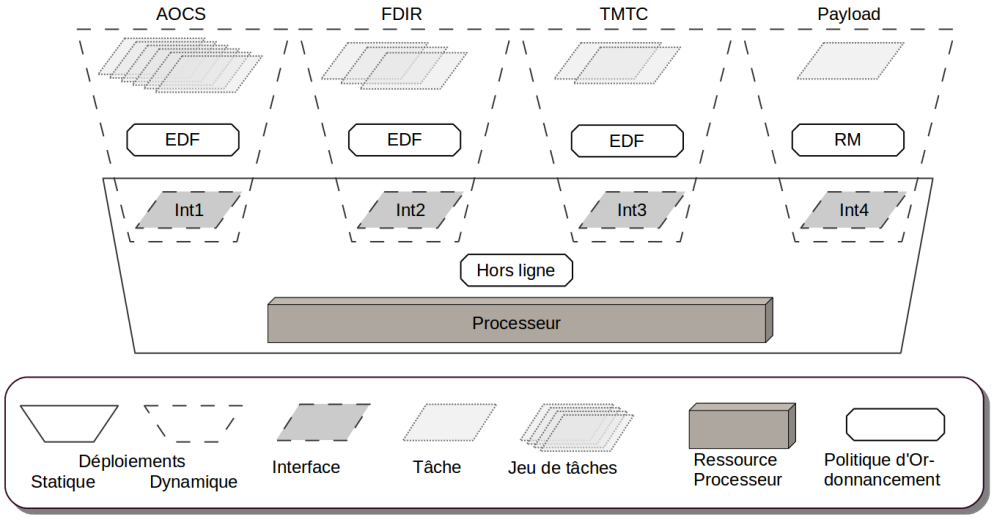

FIGURE 5.12 – Modélisation de l'étude de cas à l'aide de Cheddar ADL.

Le niveau global est modélisé à l'aide d'un déploiement statique. La charge de travail, i.e., les quatre applications, est modélisée à l'aide de quatre tâches d'ordonnancement. Ces dernières sont déployées sur le processeur à l'aide d'un déploiement statique. Chacune des applications du niveau local est modélisée à l'aide d'une tâche d'ordonnancement (celle déployée sur le processeur) et d'un ensemble de tâches. Les tâches sont affectées à leur application à l'aide d'un déploiement dynamique.

Paramètres temporels des applications de l'étude de cas

Toute application i est constituée d'un ensemble de n tâches $\mathcal{T}_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,n}\}$. Les contraintes temporelles de chaque tâche sont exprimées à l'aide d'un tuple $\tau_{i,j} = (P_{i,j}, C_{i,j}, D_{i,j}, S_{i,j})$. $P_{i,j}$ représente la période de la tâche j de l'application i ou l'intervalle minimum entre deux de ses activations (si la tâche i,j est respectivement périodique ou sporadique). $C_{i,j}$ est le temps d'exécution pire cas de la tâche (WCET). $D_{i,j}$ spécifie l'échéance, et $\S_{i,j}$ (pour les tâches périodiques uniquement) est l'instant auquel la tâche est activée pour la première fois (activation du premier travail).

Nous fournissons, ici, les paramètres temporels de chacune des quatre application composant notre étude de cas.

Le jeu de tâches de l'application AOCS est présenté dans le tableau 5.6.

Le jeu de tâches de l'application FDIR est spécifié dans le tableau 5.7

Le jeu de tâches de l'application TMTC est spécifié dans le tableau 5.8.

Le jeu de tâches de l'application Payload est spécifié dans le tableau 5.9.

TABLE 5.6 – Jeu de tâches de l'application AOCS ($\mathcal{T}_{\mathsf{AOCS}}$)

j	Nom	Type	$P_{i,j}$	$C_{i,j}$	$D_{i,j}$	$S_{i,j}$
1	Command Act.	périodique	200	2.13	200	100
2	RW Data	périodique	200	1.43	22	0
3	Control Law	périodique	200	52.84	100	25
4	DSS Data	périodique	200	1.43	17	0
5	Gyro Data	sporadique	100	4.08	100	
6	IRES Data	périodique	100	1.43	24	0

TABLE 5.7 – Jeu de tâches de l'application FDIR ($\mathcal{T}_{\mathsf{FDIR}}$)

j	Nom	Type	$P_{i,j}$	$C_{i,j}$	$D_{i,j}$	$S_{i,j}$
1	Fault Detection	périodique	100	12.57	100	0
2	Fault Isolation	périodique	200	7.5	200	5
3	Fault Recovery	périodique	200	15	200	10

TABLE 5.8 – Jeu de tâches de l'application TMTC ($\mathcal{T}_{\mathsf{TMTC}}$)

j	Nom	Type	$P_{i,j}$	$C_{i,j}$	$D_{i,j}$	$S_{i,j}$
1	Telecommands	sporadique	187	2.5	187	
2	Telemetry	sporadique	62.5	3.19	30	

TABLE 5.9 – Jeu de tâche de l'application Payload ($\mathcal{T}_{\mathsf{Payload}}$)

j	Nom	Type	$P_{i,j}$	$C_{i,j}$	$D_{i,j}$	$S_{i,j}$
1	Payload Data	périodique	150	40	150	0

Restrictions temporelles globales

Les contraintes temporelles globales de chaque application sont exprimées comme une interface de ressource, en utilisant le modèle de ressources périodiques : $\Gamma_i \triangleq (\Pi_i, \Theta_i)$ représente la mise à disposition de Θ_i unités de ressource tous les Π_i unités de temps [SL03]. Dans le cadre de cette étude de cas, nous considérons que l'unité des ressources et de temps est la milli-seconde ; une provision de ressource (Π_i, Θ_i) indique que toutes les Π_i milli-secondes, la somme des durées où l'application i accède au processeur a pour borne

TABLE 5.10 – Interfaces à ressources périodiques dans l'étude de cas présentée ici

$\Gamma_i \triangleq (\ \Pi_i\ ,\ \Theta_i)$
$\Gamma_{\mathsf{AOCS}} = (100\ ,\ 35)$
$\Gamma_{\mathsf{FDIR}} = (100\ ,\ 30)$
$\Gamma_{\mathsf{TMTC}} = (\ 62\ ,\ 10)$
$\Gamma_{\mathsf{Payload}} = (150\ ,\ 40)$
$\sum \Theta_i/\Pi_i \simeq 1.078$

On peut noter que les applications présentées ici ne sont pas ordonnançables sur un unique processeur ($\sum \Theta_i/\Pi_i > 1$).

supérieure Θ_i milli-secondes.

Cette dernière donne lieu à la sélection d'un unique test de faisabilité : le test de faisabilité proposé par le test de Mok et al. [MFC01] (cf. partie 5.2.1). La sélection de tests de faisabilité a également été réalisée pour les deux études de cas du chapitre 4 avec des résultats similaires aux précédents (cf. partie 4.5).

Les tableaux 5.10 présentent les interfaces (exprimées à l'aide du modèle de ressource périodique) garantissant l'ordonnançabilité de chaque tâche comme si elle était ordonnancée localement à l'aide de la politique EDF.

Le test de faisabilité sélectionné est bien applicable à l'étude de cas. Nous montrons, ainsi, la correction de notre algorithme de sélection de tests de faisabilité.

5.8.5 Évaluation à l'aide d'architectures générées

L'objectif de cette évaluation est d'évaluer la robustesse et le passage à l'échelle de notre prototype.

De même que pour notre évaluation présentée dans le chapitre 4, cette évaluation est réalisée à l'aide d'architectures générées.

Pour chacun des trois patrons de conception présentés dans ce chapitre, nous avons généré vingt architectures leur étant conformes. Nous avons également modifié cinq architectures conformes à chacun de ces trois patrons, formant ainsi quinze architectures non-conformes. Ces modifications consistent en l'inclusion de nœuds non-conformes à un patron de conception dans un système lui étant conforme. Les nœuds inclus de la sorte sont, par construction, conformes à un nœud présent dans un autre patron de conception. Cela permet de vérifier le bon fonctionnement de la comparaison entre le graphe de contraintes construit par l'algorithme et les graphes de contraintes définissant les patrons.

Le processus de vérification des résultats reste le même que pour l'évaluation précédente. Le nom des tests sélectionnés, les contraintes non-respectées et les échecs lors de la comparaison des deux graphes de contraintes sont stockés dans un fichier. Cette sélection (ou non-sélection) est ensuite vérifiée pour chacune des architectures.

Patrons	ARINC653	two-levels-independent-application
Quantité	20	20
Patrons	n-levels-independent-workloads	non-conforme
Nombre	20	15

TABLE 5.11 – Quantité d'architectures générées pour l'évaluation de notre approche.

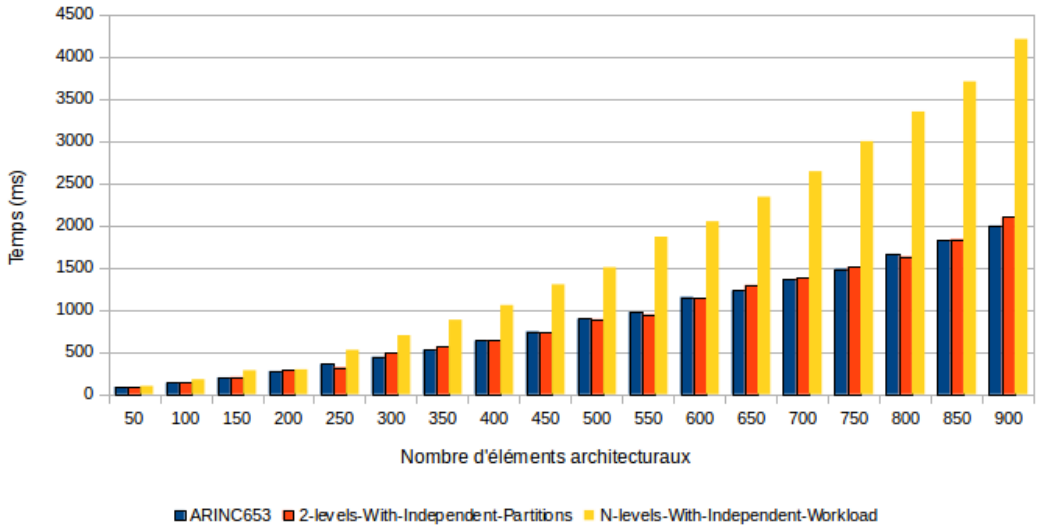

FIGURE 5.13 – Temps de réponse de l'algorithme étendu de sélection de tests de faisabilité, en fonction du nombre de tâches et de dépendances du modèle d'architecture analysé (nombre de tâches = nombre de dépendances). Toutes les architectures évaluées comportent 5 applications.

La figure 5.13 présente le temps de réponse de l'algorithme de sélection de tests de faisabilité, en fonction du nombre de tâches et de dépendances du système analysé.

Cette évaluation a été exécutée sur une plate-forme Ubuntu 12.04. La machine utilisée possède un processeur Intel Core i5-2430M CPU @ 2.40GHz × 4 avec 4,0GiB de mémoire RAM.

Cette évaluation permet de montrer que notre prototype est robuste : toutes les contraintes d'applicabilité non-respectées par les modèles de STRECs générés ont été détectées et signalées à l'utilisateur.

Pour les modèles d'architecture de grande taille, i.e., dont la quantité d'éléments architecturaux (tâches + dépendances) dépasse le millier, le temps de réponse de l'algorithme reste de quelques secondes. Cela nous permet de valider le passage à l'échelle de notre prototype. On peut noter que le temps de réponse de notre prototype varie de quelques

milli-secondes à quatre secondes pour de grands systèmes (1000 tâches + 1000 dépendances). Nous en déduisons donc que notre approche reste adaptée à l'utilisation de notre prototype intégré à un environnement de développement intéractif, comme Eclipse par exemple.

5.9 Conclusion

Les systèmes temps-réel à ordonnancement hiérarchique sont souvent utilisés pour le développement de systèmes avioniques. Plus complexes que les systèmes mono-processeurs étudiés chapitre 4, ils conservent néanmoins les mêmes exigences du point de vue de l'analyse temporelle.

Dans ce chapitre, nous proposons d'étendre notre approche de modélisation des relations entre architectures et tests de faisabilité au cas des systèmes embarqués temps-réel critiques à ordonnancement hiérarchique. Nous présentons ces derniers et les méthodes d'analyse d'ordonnancement qui leur sont spécifiques.

Le modèle de patron de conception architectural présenté dans le chapitre 4 n'est pas suffisant pour associer de tels systèmes à des tests de faisabilité adaptés.

Nous proposons donc un modèle de patrons de conception basé sur le concept de nœuds d'association pouvant être hiérarchisés au sein d'un graphe de contraintes. Ce modèle permet la représentation des caractéristiques de systèmes comprenant des environnements d'exécution complexes. Nous traitons le cas de l'ordonnancement hiérarchique ici, mais notre approche est applicable aux systèmes multiprocesseurs par exemple.

Trois nouveaux patrons de conception sont proposés. Pour ce faire nous définissons quatre nouveaux ensembles de contraintes d'environnement.

Par la suite, l'algorithme de sélection automatique de tests de faisabilité présenté chapitre 4 est étendu par une analyse des systèmes à ordonnancement hiérarchique. Enfin, notre approche est évaluée à l'aide d'études de cas et d'architectures générées.

Ce chapitre a présenté l'extension de notre approche dans le cas d'un environnement d'exécution complexe : les systèmes à ordonnancement hiérarchique.

Le chapitre suivant est consacré à la description de notre processus de validation des propositions exposées dans les chapitres 4 et 5.

Chapitre 6

Mises en œuvre

Au cours de cette thèse, les deux contributions présentées précédemment ont été évaluées par la réalisation de prototypes intégrés à l'environnement d'analyse Cheddar. La mise en place de ces prototypes a donné lieu à la création de quatre nouveaux composants de Cheddar, de deux nouveaux tests de faisabilité, ainsi qu'à la modification et l'extension de son langage de description d'architecture : Cheddar ADL.

Ces travaux ont été réalisés selon le processus d'ingénierie de Cheddar proposé par Plantec et al. [PS07]. Ce procédé permet la génération de code d'une partie de Cheddar. L'objectif de ce chapitre est de présenter l'ensemble des mises en œuvre relatives aux patrons de conception architecturaux et aux deux algorithmes de sélection de tests de faisabilité.

Nous commençons par présenter le processus d'ingénierie de Cheddar dans la partie 6.1 avant d'exposer un survol de nos réalisations en partie 6.2. Elles sont ensuite détaillées dans les parties suivantes. Dans un premier temps, nous décrivons, en partie 6.3, les modifications apportées au langage d'architecture de Cheddar : les dépendances et les déploiements. La modélisation des contraintes d'applicabilité, en vue de leur génération automatique est proposé à la partie 6.4. La partie 6.5 est dédiée à la présentation des deux prototypes d'algorithme de sélection de tests de faisabilité. Enfin, le générateur d'architecture utilisé pour leurs évaluations est détaillé en partie 6.6.

6.1 Le processus d'ingénierie de Cheddar

L'objectif de cette partie est de présenter le processus d'ingénierie de Cheddar. Nous commençons par décrire les composants architecturaux de Cheddar, puis le processus d'ingénierie utilisé pour sa mise en œuvre.

6.1.1 Les composants architecturaux standards de Cheddar

Cheddar est constitué de sept composants principaux. Ces composants sont présentés à la figure 6.1. Ils se répartissent en deux couches logicielles :

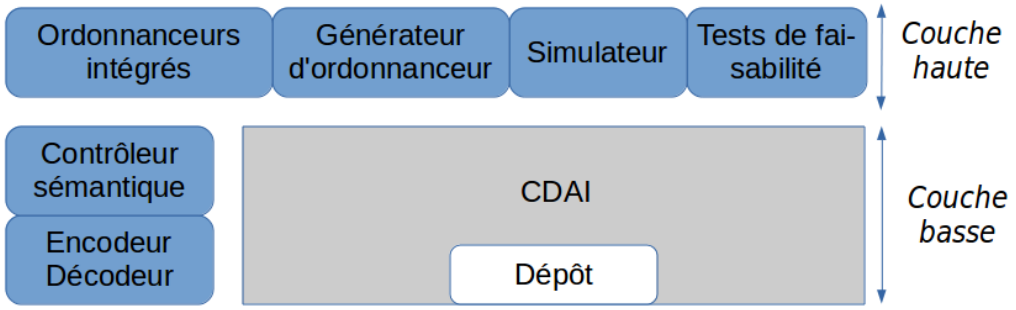

FIGURE 6.1 – Architecture logicielle à deux couches de Cheddar.

— la couche basse est dédiée à la gestion des données. Elle dépend des types de données manipulées par Cheddar et des contraintes associées. Ces contraintes établissent des règles structurelles et sémantiques sur les données représentées. Cette couche de nature générique est fortement réutilisable et évolutive ;
— la couche haute est liée au domaine de la théorie de l'ordonnancement temps-réel et de la simulation de systèmes temps-réel. Cette couche repose sur la couche basse pour l'accès aux données. Elle est évolutive de par l'utilisation d'un procédé de mise en œuvre dirigée par les modèles. Cette évolutivité est rendue possible par l'utilisation d'un générateur de code spécifique.

La couche basse

La couche basse de Cheddar comprend un dépôt centralisé pour le stockage des données. Les entités du langage d'architecture Cheddar ADL, présentées dans la partie 2.6.5, sont des exemples de données. Le dépôt est un composant de stockage de données. Dans la version actuelle de Cheddar, il s'agit de la mémoire.

Le dépôt est encapsulé dans un composant d'accès aux données (CDAI pour *Cheddar Data Acces Interface*). Tous les accès en lecture ou écriture de données ou méta-données s'effectuent au travers de ce composant. Le composant d'accès aux données est l'élément architectural central autour duquel s'articulent tous les autres composants de Cheddar.

La couche basse est également constituée de deux composants supplémentaires liés à la définition des données manipulées au travers de la CDAI :

— le contrôleur sémantique : il permet la validation des données et des méta-données ;
— le composant d'encodage et de décodage : permet une interopérabilité par échange de données entre Cheddar et des langages d'architecture autre que Cheddar ADL. Ce composant met en œuvre un protocole standard pour l'encodage et le décodage des données et des méta-données vers ou depuis XML.

La couche haute

Les composants constituant la couche haute sont les suivants :
— les ordonnanceurs intégrés mettent en œuvre des algorithmes d'ordonnancement reconnus comme classiques par la communauté ;
— le générateur d'ordonnanceurs permet la réalisation de versions spécifiques de Cheddar comprenant des ordonnanceurs définis par des utilisateurs à l'aide d'un langage dédié [PS07] ;
— le simulateur réalise la simulation de l'ordonnancement de systèmes à partir des ordonnanceurs intégrés ou de ceux générés par un utilisateur ;
— une bibliothèque de tests de faisabilité ; Cheddar propose un ensemble de tests de faisabilité pouvant être appliqués à des modèles de STRECs. L'ensemble de ces tests de faisabilité est présenté dans [Sin10].

6.1.2 Le processus d'ingénierie de Cheddar

La réalisation de la mise en œuvre des deux couches logicielles présentées en partie 6.1 est assurée par un procédé dirigé par les modèles à deux niveaux. Ce procédé permet une évolutivité incrémentale du système. La figure 6.2 présente le procédé utilisé.

Pour la couche basse, les objets manipulés, comprenant les données et les méta-données, sont spécifiés par le modèle de données de Cheddar. Ce modèle intègre la définition structurelle des hiérarchies et la définition des contraintes des entités du système. À une version particulière de ce modèle correspond une version de la couche basse puisque tous les constituants de cette couche sont automatiquement produits à partir de ce modèle par un générateur de couche basse qui produit les paquetages Ada correspondants.

Prenons l'exemple du générateur d'ordonnanceur. Cheddar propose un langage dédié de spécification d'ordonnanceur à l'aide d'automates [Sin08]. Ce générateur permet de traduire la représentation interne du programme en un ordonnanceur sous la forme d'un paquetage Ada spécifique intégré au canevas Cheddar. Cette traduction met en jeu deux processus de génération. Tout d'abord, le code de l'ordonnanceur est généré. Il comprend la mise en œuvre de l'automate et les calculs associés aux états. Ce paquetage est produit dans la couche haute par le générateur d'ordonnanceurs sous la forme d'un ordonnanceur intégré. En complément, le nouvel ordonnanceur doit être déclaré dans le modèle de données de Cheddar, ce qui implique la génération d'une entité complémentaire. Le processus de production automatique de la couche basse est alors exploité. Après recompilation de l'ensemble, le nouvel ordonnanceur fait partie intégrante du canevas Cheddar. L'intégration d'un ordonnanceur spécifique représente un incrément. À chaque incrément, une nouvelle version de Cheddar spécifiquement adaptée au contexte d'utilisation est ainsi produite.

6.1.3 Génération d'une partie du code de Cheddar

Le processus dirigé par les modèles exploite deux générateurs complémentaires, un pour générer la couche basse et l'autre pour produire les ordonnanceurs de la couche haute. Classiquement, dans le cadre de l'ingéniérie dirigée par les modèles, la construction d'un

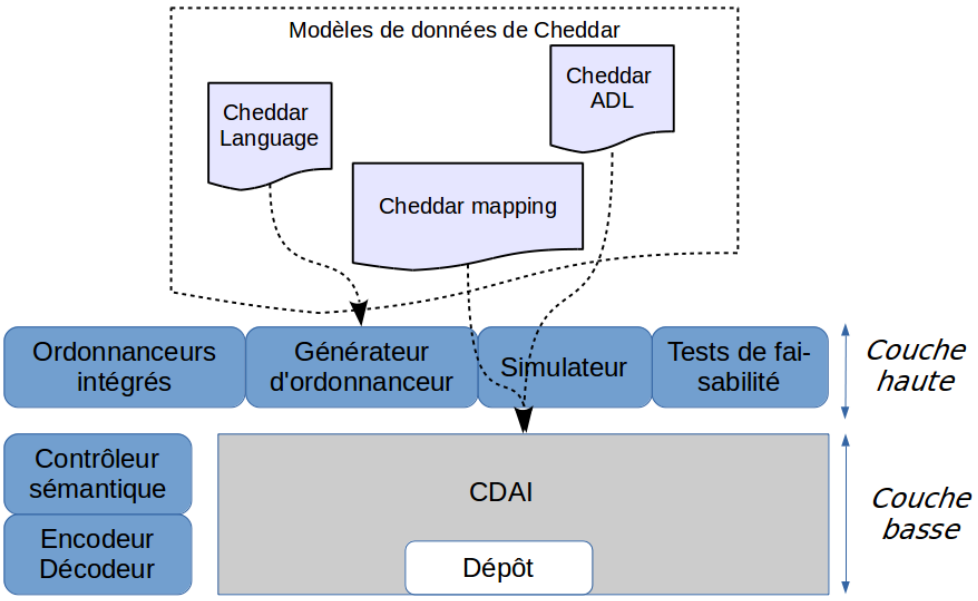

FIGURE 6.2 – Évolution incrémentale de Cheddar : un exemple avec le générateur d'ordonnanceur.

générateur de code nécessite la déclaration d'une syntaxe abstraite associée à la déclaration des règles de traduction. La génération de code proprement dite est mise en œuvre comme un processus de compilation.

Pour chaque générateur, un méta-modèle associé à des règles de génération de code est développé en EXPRESS :

— Le méta-modèle est un modèle de données. Il spécifie la syntaxe abstraite du langage source utilisé pour modéliser. Cette déclaration s'abstrait complètement de la syntaxe concrète du langage source.

— Les règles de traduction sont encapsulées dans les entités du méta-modèle sous la forme d'attributs dérivés. Ce sont ces attributs dérivés qui permettent de générer du code.

6.1.4 STEP, EXPRESS et Platypus

Le processus d'ingénierie de Cheddar se base sur l'utilisation de générateurs de code. Ces générateurs utilisent des modèles spécifiés à l'aide d'EXPRESS, un langage de modélisation conforme à la norme STEP.

Le standard STEP et le langage EXPRESS

Le standard STEP (*STandard for the Exchange of Product model data*) est la norme ISO 10303 élaborée par le sous-comité ISO TC 184/SC4. L'objectif de cette norme est de permettre aux applications informatiques industrielles d'échanger et de partager des données indépendamment des spécificités des différents systèmes informatiques.

Les travaux de normalisation comprennent, d'une part le développement d'une technologie fournissant des méthodes et outils informatiques neutres pour la description, la validation et la manipulation des informations de définitions de produits, et d'autre part la spécification de modèles de données standards, appelés protocoles d'application et organisés par métiers industriels. Ces protocoles d'application sont développés dans le cadre strict de la technologie STEP. Ils sont spécifiés dans le langage de modélisation EXPRESS et constituent une bibliothèque de concepts réutilisables pour développer des modèles de données [Bou95].

Le langage EXPRESS permet la spécification de modèles de données. Son originalité est de permettre la spécification des contraintes locales aux types ou aux entités mais aussi des contraintes globales.

Génération de code avec Platypus

Ce processus d'ingénierie dirigée par les modèles est mis en œuvre grâce à Platypus, un méta-environnement basé sur l'utilisation de la technologie STEP, notamment le langage de modélisation EXPRESS. Platypus [PS06] est l'environnement STEP utilisé pour construire des générateurs de code pour Cheddar.

Avec Platypus, le développement d'un compilateur pour le langage source de la génération de code n'est pas nécessaire si le langage de modélisation source est EXPRESS et si le méta-modèle est une spécialisation du méta-modèle de Platypus lui-même. La spécification d'un lien de conformité entre les entités du modèle source et celle du méta-modèle (syntaxe abstraite) permet la génération automatique de code.

Nous allons, dans la suite de ce chapitre, détailler nos différentes réalisations.

6.2 Intégration des réalisations dans le canevas Cheddar

L'objectif de cette partie est de présenter de façon globale l'ensemble des réalisations produites lors de cette thèse. Ces réalisations ont été intégrées au processus d'ingénierie de Cheddar et sont présentées à la figure 6.3. Nous identifions cinq réalisations majeures, liées à l'approche exposée dans les chapitres 4 et 5. Ces dernières sont présentées successivement ci-dessous.

Extension du méta-modèle de Cheddar ADL

Comme nous l'exposons dans la partie 2.6.5, Cheddar propose un langage de description d'architecture permettant la modélisation de systèmes temps-réel embarqués critiques.

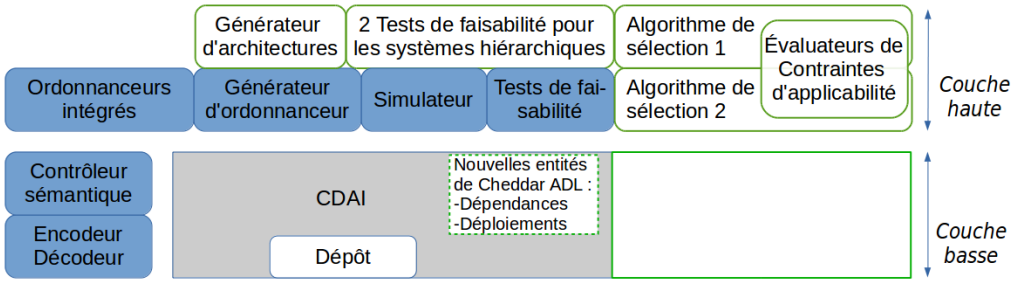

FIGURE 6.3 – Intégration des nouveaux composants dans l'architecture logicielle à deux couches de Cheddar.

Afin de pouvoir modéliser les STRECs de façon à être analysés de façon efficace par nos algorithmes de sélection de tests de faisabilité, nous avons proposé de nouvelles entités de type *dependency*.

Par ailleurs, Cheddar ne supportait la modélisation de systèmes à ordonnancement hiérarchiques que de façon limitée. Nous avons donc proposé une nouvelle façon de modéliser les architectures de STRECs à l'aide des entités *Deployments*. Ces deux extensions impactent la couche basse de Cheddar et ont conduit à une génération automatique de code. Cette réalisation est décrite dans la partie 6.3.

Modélisation des contraintes d'applicabilité

Un des objectifs de notre approche est de permettre à des utilisateurs de définir leurs propres patrons de conception. Pour ce faire, nous avons modélisé l'intégralité des contraintes d'applicabilité présentées dans ce manuscrit, afin de pouvoir générer automatiquement de nouvelles versions de nos algorithmes de tests de faisabilité.

Les contraintes sont modélisées à l'aide du langage EXPRESS, dans l'optique de générer automatiquement le code vérifiant la conformité d'un modèle de STREC à ces contraintes. La modélisation des contraintes d'applicabilité est décrite et discutée en partie 6.4.

Implantation de deux nouveaux tests de faisabilité

Afin de pouvoir prendre en main les modèles de systèmes à ordonnancement hiérarchique et valider notre approche pour ces systèmes, nous avons implanté deux tests de faisabilité spécifiques aux systèmes à ordonnancement hiérarchique. Ces tests sont ceux proposés par Davis et al. [DB05] et Mok et al. [MFC01]. Nous ne revenons pas plus en détails sur ces tests ici, leur description a été faite dans le chapitre 5.

Réalisation des deux algorithmes de sélection de tests de faisabilité

Afin d'évaluer l'approche présentée dans les chapitres 4 et 5, nous avons réalisé deux prototypes implantant l'algorithme de sélection de tests de faisabilité.

La réalisation de ces prototypes a impliqué la modification des deux couches de Cheddar. Dans un premier temps, la couche basse a été étendue avec les modèles de données manipulés par les algorithmes. Dans un second temps, la couche haute a été enrichie de deux composants, produits manuellement, réalisant les deux algorithmes. Ces derniers utilisent les paquetages dédiés à la vérification de conformité à des contraintes d'applicabilité.

Nous revenons plus en détails sur l'implantation des algorithmes dans la partie 6.5.3.

Proposition d'un générateur d'architectures

Comme précisé lors de la partie précédente, les évaluations de nos prototypes ont été réalisées à l'aide d'un générateur d'architectures. En effet, pour chacune des étapes de conception des prototypes, nous avons généré des ensembles d'architectures aux propriétés diverses.

Le générateur propose une cinquantaine de procédures, dont onze sont dédiées à la génération de modèles d'architecture conformes ou non à nos patrons de conception.

Ce générateur à été implanté manuellement et n'a pas bénéficié de l'utilisation de l'ingénierie dirigée par les modèles, contrairement aux parties de prototype décrites ci-après.

Le générateur est décrit plus en détails dans la partie 6.6.

Quelques métriques à propos de ces réalisations

Nous donnons ici quelques métriques à propos de nos réalisations au sein du canevas Cheddar. Le tableau 6.1 résume les quantités de code EXPRESS (en termes de lignes) et d'Ada (en termes de lignes et de paquetages) produits pour chacun des composants.

La figure 6.4 représente les modèles de données proposés et les paquetages générés. Toutes les réalisations effectuées au cours de cette thèse sont colorées en blanc au sein de la figure. Nous avons étendu l'ensemble de modèles de données de Cheddar de cinq nouveaux modèles EXPRESS suivants :

— un modèle pour les contraintes d'applicabilité,
— un modèle pour le graphe de dépendances,
— un modèle pour les structures de contraintes d'applicabilité,
— un modèle pour le graphe de contraintes et les nœuds d'association et
— un modèle pour les cas d'application.

Des instances de cas d'application (modélisées en STEP) ont également été ajoutées.

Nous allons, dans la suite de ce chapitre, détailler nos réalisations selon le plan présenté dans cette partie.

TABLE 6.1 – Récapitulatif de la quantité de lignes EXPRESS, de paquetage Ada et de lignes d'Ada pour chacun des composants proposés.

Composants	Lignes EXPRESS	Paquetages Ada	Lignes Ada
Générateur d'architectures	-	1	2630
Algorithme de sélection 1	196	10	15970
Algorithme de sélection 2	128	4	8620
Contraintes d'applicabilité	165	36	4560
Tests de faisabilité	-	2	1210
Sommes des métriques	489	53	32990

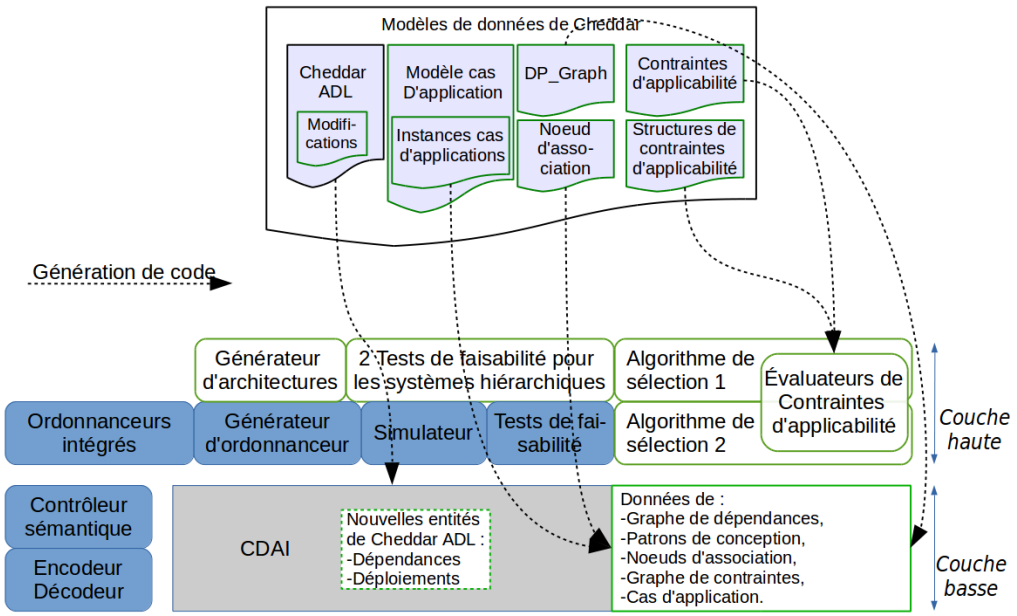

FIGURE 6.4 – Génération de code pour les composants réalisés dans le cadre de cette thèse.

6.3 Extension du méta-modèle de Cheddar ADL

Lors de mes travaux de thèse, j'ai été amené à prendre part à l'évolution du langage d'architecture Cheddar ADL.

La première modification a été d'étendre les dépendances entre les tâches. Précédemment, il en existait trois : *Precedence_ Dependency*, *Communication_ Dependency*, et *Resource_ Dependency*. Ces trois dépendances modélisent, respectivement, une contrainte

de précédence entre deux tâches, l'échange d'un message ou évènement entre deux tâches et l'accès à une donnée partagée.

La seconde modification consiste à introduire le concept de déploiement. Ces entités permettent la modélisation de STRECs à ordonnancement hiérarchique de façon générique.

6.3.1 Extension de Cheddar ADL avec les dépendances

Nous avons introduit trois nouvelles dépendances modélisant chacune un protocole de communication spécifique :

— *Time_ Triggered_ Communication_ Dependency*,
— *Queuing_ Buffer_ Dependency*,
— *Black_ Board_ Buffer_ Dependency*,

La dépendance *Time_ Triggered_ Communication_ Dependency* représente une communication de type *time-triggered*, tandis que les deux autres modélisent des communications qui utilisent un *Buffer* et conformes aux communications de type *Queued-buffer* et *Blackboard*.

Ces modifications impactent le langage Cheddar ADL. Les modifications découlant de ces nouveaux modèles impactent le CDAI, qui est généré automatiquement à l'aide du modèle du langage.

Un extrait du modèle de Cheddar ADL est fourni dans la figure 6.5. Nous y fournissons le code EXPRESS des entités décrites ci-dessus.

L'extrait du modèle propose cinq entités définies au sein d'un même *SCHEMA*.

Les clauses *USE FROM* permettent l'inclusion des entités nécessaires à la définitions des entités spécifiées dans ce schéma. Ici, les entités concernées sont celles impliquées dans les mécanismes de communication et de synchronisation : *Messages*, *Tasks*, *Buffers* et *Resources*.

Dans ce modèle, nous définissons deux types énumérés :

— *Time_Triggered_Communication_Timing_Property_Type* modélise les trois protocoles de communication Time-Triggerd, i.e., échantilloné, retardé ou immédiat ;
— *Orientation_Dependency_Type* modélise le type d'accès à un buffer de communication.

Ensuite, les cinq entités définissant les différentes dépendances sont spécifiées. Une entité est identifiée par son nom et les attributs qui la composent. Par exemple, la dépendance *Time_Triggered_Communication_Timing_Property_Type* est constituée de trois attributs. Les deux premiers sont les tâches concernées par la dépendance et le troisième précise le type de protocole utilisé.

6.3.2 Extension de Cheddar ADL avec les déploiements et la tâche d'ordonnancement

Afin de modéliser les systèmes hiérarchiques, nous avons proposé une modification de Cheddar ADL.

```
SCHEMA Dependencies;
  USE FROM Basic_Types;
  USE FROM Messages;
  USE FROM Tasks;
  USE FROM Buffers;
  USE FROM Resources;

  ...

  TYPE Time_Triggered_Communication_Timing_Property_Type = ENUMERATION
    OF ( Sampled_Timing, Immediate_Timing, Delayed_Timing );
  END_TYPE;

  TYPE Orientation_Dependency_Type = ENUMERATION
    OF ( From_Object_To_Task, From_Task_To_Object );
  END_TYPE;

  ENTITY Time_Triggered_Communication_Dependency_Type;
    time_triggered_communication_sink : Generic_Task;
    time_triggered_communication_source : Generic_Task;
    timing_property : Time_Triggered_Communication_Timing_Property_Type;
  END_ENTITY;

  ENTITY Precedence_Dependency_Type;
    precedence_sink : Generic_Task;
    precedence_source : Generic_Task;
  END_ENTITY;

  ENTITY Queuing_Buffer_Dependency_Type;
    buffer_dependent_task : Generic_Task;
    buffer_orientation : Orientation_Dependency_Type;
    buffer_dependency_object : Buffer;
  END_ENTITY;

  ENTITY Black_board_Buffer_Dependency_Type;
    black_board_dependent_task : Generic_Task;
    black_board_orientation : Orientation_Dependency_Type;
    black_board_dependency_object : Buffer;
  END_ENTITY;

  ENTITY Resource_Dependency_Type;
    resource_dependency_resource : Generic_Resource;
    resource_dependency_task : Generic_Task;
  END_ENTITY;

  ...

END_SCHEMA;
```

FIGURE 6.5 – Extrait du modèle de Cheddar ADL en EXPRESS : les dépendances.

Dans la version précédente de l'ADL, les jeux de tâches étaient directement affectées à un processeur. Il était possible de modéliser certains systèmes hiérarchiques à deux niveaux, et de façon restreinte [Sin08].

L'objectif de cette extension du langage d'architecture est de permettre la modélisation de systèmes hiérarchiques à plusieurs niveaux, ainsi que les nouveaux concepts introduits

```
SCHEMA Deployments;
  USE FROM Basic_Types;
  USE FROM Objects;
  USE FROM Scheduling_Analysis;
  USE FROM Scheduler_Interface;

  ENTITY Generic_Deployment
    SUBTYPE OF ( Named_Object );
    consumer_entities : Generic_Objects_Set;
    resource_entities : Generic_Objects_Set;
  DERIVE
    SELF\Generic_Object.object_type : Objects_Type := Deployment_Type;
  END_ENTITY;

  ENTITY Static_Deployment
    SUBTYPE OF ( Generic_Deployment );
    allocation_description : STRING;
  END_ENTITY;

  ENTITY Dynamic_Deployment
    SUBTYPE OF ( Generic_Deployment );
    allocation_parameters : Scheduling_Parameters;
  END_ENTITY;
END_SCHEMA;
```

FIGURE 6.6 – Extrait du méta-modèle de Cheddar ADL en EXPRESS : les déploiements.

par l'analyse de ces système : les interfaces [LLS07]. Pour ce faire, il est nécessaire d'introduire une entité modélisant les applications et leurs interfaces, tout en conservant l'information de chaque tâche. Le maintien des concepts et entités manipulés par les méthodes d'analyse préalablement implantées dans Cheddar est également un aspect important. Il existe des systèmes où l'affectation des ressources est statique et d'autres où l'affectation est dynamique.

Nous proposons d'étendre le méta-modèle de Cheddar ADL avec deux nouvelles entitées :

— le concept de déploiement statique,

— le concept de déploiement dynamique.

Un déploiement modélise une relation d'association entre deux types d'entités : des ressources et leurs utilisateurs. La figure 6.6 présente l'extrait du méta-modèle de Cheddar ADL pour les déploiements.

Nous distinguons deux types de déploiements :

1. le déploiement statique : il permet de modéliser l'affectation statique de ressources à des utilisateurs. Il peut modéliser l'ordonnancement hors-ligne de tâches ou d'applications, comme défini pour les architectures conformes à ARINC653. L'attribut *allocation_description* permet de définir une séquence d'ordonnancement.
2. le déploiement dynamique : il permet de modéliser l'affectation dynamique de ressource aux utilisateurs. Il peut modéliser un algorithme d'ordonnancement en-ligne

```
SCHEMA all_tasks_are_periodic_constraint;
  USE FROM Tasks;

  RULE all_tasks_are_periodic FOR ( generic_task );
  WHERE
    R4 : SIZEOF ( QUERY ( t <* generic_task | NOT ( 'TASKS.PERIODIC_TASK' IN TYPEOF ( t ) ) ) ) =
         0;
  END_RULE;
END_SCHEMA;


SCHEMA no_shared_resources_constraint;
  USE FROM Resources;

  RULE no_shared_resources FOR ( Generic_Resource );
  WHERE
    R6 : ( SIZEOF ( Generic_Resource ) = 0 );
  END_RULE;
END_SCHEMA;


SCHEMA No_Buffer_constraint;
  USE FROM Buffers;

  RULE No_Buffer FOR ( Buffer );
  WHERE
    R5 : SIZEOF ( Buffer ) = 0;
  END_RULE;
END_SCHEMA;

SCHEMA At_least_One_Time_Triggered_communication_constraint;
  USE FROM Dependencies;

  RULE At_least_One_Time_Triggered FOR ( Time_Triggered_Dependency_Type );
  WHERE
    R13 : SIZEOF ( Time_Triggered_Dependency_Type ) > 1;
  END_RULE;
END_SCHEMA;
```

FIGURE 6.7 – Exemple de modèles de contraintes : l'ensemble de contraintes Time-Triggered.

pour des tâches ou partitions. Il peut également modéliser un algorithme type *malloc* pour un ensemble de composants au sein d'un espace d'adressage. L'attribut *allocation_parameters* contient les caractéristiques de l'algorithme d'affectation des ressources.

6.4 Modélisation des contraintes d'applicabilité

Afin d'évaluer la correction des contraintes vis-à-vis du méta-modèle de Cheddar ADL, nous avons modélisé l'intégralité des contraintes rencontrées lors de nos travaux en EXPRESS. La figure 6.7 propose quatre exemples de modélisation de contraintes.

Les contraintes sont modélisées par des schémas EXPRESS (un par contrainte). Chaque schéma inclut le schéma définissant les entités qu'il restreint ou les relations entre les entités qu'il restreint. Chaque contrainte est modélisée par une règle EXPRESS. Une règle EXPRESS est une contrainte globale contraignant l'utilisation d'entités du méta-modèle.

Une règle est modélisée par une *RULE*, qui contient le type d'entité contraint et la contrainte elle-même.

La contrainte est appliquée de façon identique à toutes les instances d'entités contenant le type référencé par la *RULE*.

La figure 6.7 présente les contraintes du patron de conception *Time-Triggered Uniprocessor*. Il est constitué de cinq schémas restreignant, respectivement, les entités $Task$, les entités *Resource*, les entités $Buffer$, et les entités *Dependency*.

Nous avons identifié 3 opérateurs servant à la construction des contraintes. Pour chacun de ces opérateurs, nous avons implanté son équivalent en Ada. Les opérateurs sont : *QUERY*, *TYPEOF* et *SIZEOF*. L'opérateur *QUERY* est un accesseur ensembliste dont les paramètres sont un type d'entité et une contrainte. L'opérateur *TYPEOF* retourne la liste des types de l'entité passée en paramètre. L'opérateur *SIZEOF* retourne le cardinal de l'ensemble passé en paramètre.

Par exemple, la règle *all_tasks_are_periodic* restreint toutes les entités *generic_task*. Cette règle impose que le cardinal de l'ensemble de tâches n'étant pas de type périodique doit être égal à 0. Cette règle EXPRESS modélise la contrainte *cst6*. La figure 6.8 présente le code généré pour la contrainte *cst6*.

Par la suite, toutes les contraintes d'applicabilité rencontrées lors de nos travaux, ont été modélisées de la sorte et implantées par 36 paquetages Ada indépendants.

6.5 Réalisation des deux algorithmes de sélection de tests de faisabilité

L'objectif de cette partie est de présenter comment s'intègre nos travaux aux deux couches du canevas de Cheddar.

6.5.1 Extension de la couche basse de Cheddar : les modèles de données

Comme pour le reste de Cheddar, nous avons bénéficié des fonctionnalités de génération de code de Platypus afin de générer les structures de données utilisées par nos algorithmes.

Nous avons modélisé les quatre structures de données suivantes :

1. Le graphe de dépendances ;
2. Les patrons de conception ;
3. Les nœuds d'association ;
4. Les cas d'application.

Le modèle du graphe de dépendances, et celui du cas d'application sont présentés ci-dessous. Nous exposons, également, la génération de code à partir de morceaux choisis.

```
package body applicability_constraint.all_tasks_are_periodic is

   function R1_QUERY1_Condition (t : Generic_Task_Ptr) return Boolean is
   begin

      return not (Element_In_List
                     (To_Unbounded_String ("TASKS.PERIODIC_TASK"),
                      type_of (t)));

   end R1_QUERY1_Condition;

   function R1 (Sys : System) return Boolean is
   begin
      Context := Sys;
      return (Get_Number_Of_Elements
                 (Select_And_Copy (Sys.Tasks, R1_QUERY1_Condition'Access)) =
              Generic_Task_Set.Element_Range (0));

   end R1;

   function R1_Txt return Unbounded_String is
   begin

      return ("The constraint R1 is not met:")&unbounded_lf &
        ("all tasks must be periodic.") &unbounded_lf ;

   end R1_Txt;

end applicability_constraint.all_tasks_are_periodic;
```

FIGURE 6.8 – Exemple de modèles de contraintes de l'ensemble *Time-Triggered*.

Le modèle du graphe de dépendances

Les figures 6.9 et 6.10 fournissent les modèles EXPRESS du graphe de dépendances proposé dans le chapitre 4 (cf. partie 4.4). Ce modèle est constitué de deux schémas. Le premier modélise un graphe classique. Le second étend le premier et propose des arcs étiquetés par le type de dépendances qu'ils représentent.

Les paquetages pour ces modèles sont générés automatiquement selon le processus de génération de code de Cheddar. La figure 6.11 propose un extrait du paquetage généré pour l'entité *Graph* du modèle EXPRESS de la figure 6.10.

Dans la partie 4.4.2, nous définissons différentes vues sur le graphe de contraintes. L'implantation de ces vues a été réalisée manuellement.

6.5.2 Modélisation des cas d'application

Comme nous le précisions dans le chapitre 4, pour un même patron de conception, de nombreux cas d'application peuvent être associés. Afin de pouvoir générer le code Ada permettant le parcours et l'évaluation de la conformité d'un modèle d'architecture à un cas d'application, nous avons proposé un modèle EXPRESS.

```
SCHEMA Generic_Graph;                                                    1
  USE FROM basic_types;                                                  2
                                                                         3
  ENTITY Generic_Node                                                    4
    ABSTRACT SUPERTYPE;                                                  5
    Id : STRING;                                                         6
  END_ENTITY;                                                            7
                                                                         8
  ENTITY Node_Lists_Package;                                             9
    List_ : Node_Lists;                                                  10
    Iterator : Node_Lists_Iterator;                                      11
  END_ENTITY;                                                            12
                                                                         13
  TYPE Node_Lists = LIST OF Generic_Node;                                14
  END_TYPE;                                                              15
                                                                         16
  TYPE Node_Lists_Iterator = Iterator;                                   17
  END_TYPE;                                                              18
                                                                         19
  ENTITY Generic_Edge                                                    20
    ABSTRACT SUPERTYPE;                                                  21
    Id : STRING;                                                         22
    Node_1 : STRING;                                                     23
    Node_2 : STRING;                                                     24
  END_ENTITY;                                                            25
                                                                         26
  ENTITY Edge_Lists_Package;                                             27
    List_ : Edge_Lists;                                                  28
    Iterator : Edge_Lists_Iterator;                                      29
  END_ENTITY;                                                            30
                                                                         31
  TYPE Edge_Lists = LIST OF Generic_Edge;                                32
  END_TYPE;                                                              33
                                                                         34
  TYPE Edge_Lists_Iterator = Iterator;                                   35
  END_TYPE;                                                              36
                                                                         37
  ENTITY Graph;                                                          38
    Nodes : Node_Lists;                                                  39
    Edges : Edge_Lists;                                                  40
  END_ENTITY;                                                            41
END_SCHEMA;                                                              42
```

FIGURE 6.9 – Modèle du graphe de dépendances (1/2).

```
SCHEMA DP_Graph;
  USE FROM Generic_Graph;
  USE FROM Tasks;
  USE FROM Buffers;
  USE FROM Messages;
  USE FROM Dependencies;
  USE FROM Resources;
  USE FROM Processors;


  ENTITY Task_Node
    SUBTYPE OF ( Generic_Node );
    TaskRef : Generic_Task;
    Kind : Tasks_Type;
  END_ENTITY;

  ENTITY Time_Triggered_Communication_Edge
    SUBTYPE OF ( Generic_Edge );
    Timing_Property : Time_Triggered_Communication_Timing_Property_Type;
    Time_Triggered_Communication_Sink : Task_Node;
    Time_Triggered_Communication_Source : Task_Node;
  END_ENTITY;

  ENTITY Resource_Edge
    SUBTYPE OF ( Generic_Edge );
    Resource_Dependency_Resource : Generic_Resource;
  END_ENTITY;

  ENTITY Precedence_Edge
    SUBTYPE OF ( Generic_Edge );
  END_ENTITY;

  ENTITY Communication_Edge
    SUBTYPE OF ( Generic_Edge );
    Communication_Dependency_Object : Generic_Message;
  END_ENTITY;

  ENTITY Buffer_Edge
    SUBTYPE OF ( Generic_Edge );
    Buffer_Dependency_Object : Buffer;
  END_ENTITY;
END_SCHEMA;
```

FIGURE 6.10 – Modèle du graphe de dépendances (2/2).

```
-- --------= Graph =--------
type Graph;
type Graph_Ptr is access all Graph'Class;
type Graph is new Ada.Finalization.Controlled with
record
 cheddar_private_id : Unbounded_String;
 Nodes : Node_Lists;
 Edges : Edge_Lists;
end record;

procedure Initialize(obj : in out Graph);
procedure Put(obj : in Graph);
procedure Put(obj : in Graph_Ptr);
procedure Put_Name(obj : in Graph_Ptr);
procedure Build_Attributes_XML_String(obj : in Graph; level : in natural := 0; result : in out
    Unbounded_String);
function XML_String(obj : in Graph; level : in natural := 0) return Unbounded_String;
function XML_String(obj : in Graph_Ptr; level : in natural := 0) return Unbounded_String;
function XML_Ref_String(obj : in Graph; level : in natural := 0) return Unbounded_String;
function XML_Ref_String(obj : in Graph_Ptr; level : in natural := 0) return Unbounded_String;
function Copy(obj : in Graph_Ptr) return Graph_Ptr;
function Copy(obj : in Graph) return Graph_Ptr;
function type_of(obj : in Graph) return unbounded_string_list;
function type_of(obj : in Graph_Ptr) return unbounded_string_list;
procedure Free  is new Unchecked_Deallocation (Graph'Class, Graph_Ptr);
```

FIGURE 6.11 – Extrait de l'implantation, générée en Ada, du graphe de dépendances.

Le modèle EXPRESS permet d'accueillir des fonctions implantant la vérification de conformité d'un modèle d'architecture à une contrainte d'applicabilité.

Le modèle est constitué de quatre entités :

- *Applicability_Constraint_Function* modélise une fonction de vérification de conformité à une contrainte d'applicabilité ;
- *Applicability_Constraint* contient un nom de contrainte, un résultat booléen et une fonction de vérification ;
- *Applicability_Constraint_Case* modélise un cas d'application comme une liste de tests de contraintes d'applicabilité et une liste de noms de tests de faisabilité ;
- *All_Cases_Structure* contient un ensemble d'entités *Applicability_Constraint_Case* et modélise l'ensemble des cas d'application pour un patron de conception donné.

Les paquetages permettant la gestion de ces structures de données sont générés automatiquement. L'instanciation des différents cas d'application et l'écriture des fonctions de parcours de cette structure de données a été implanté manuellement. Les différents cas d'application ont été modélisés en EXPRESS. Un exemple de modèle de cas d'application est fourni dans la figure 6.13. La figure 6.14 présente le code Ada correspondant à ce modèle de cas d'application.

```
SCHEMA Applicability_Constraints_Main_Structure;
  USE FROM basic_types;

  ENTITY Applicability_Constraint_Function;
  END_ENTITY;

  ENTITY Applicability_Constraint;
    Name : STRING;
    Result : BOOLEAN;
    Corresponding_Function : Applicability_Constraint_Function;
  END_ENTITY;

  ENTITY Applicability_Constraints_List_Package;
    List_ : Applicability_Constraints_List;
    Iterator : Applicability_Constraints_List_Iterator;
  END_ENTITY;

  TYPE Applicability_Constraints_List = LIST OF Applicability_Constraint_Case;
  END_TYPE;

  TYPE Applicability_Constraints_List_Iterator = Iterator;
  END_TYPE;

  ENTITY Applicability_Constraint_Case;
    Name : STRING;
    Feasibility_Test_Names : STRING;
    Applicability_Constraints : Applicability_Constraints_List;
  END_ENTITY;

  ENTITY Applicability_Constraint_Cases_List_Package;
    List_ : Applicability_Constraint_Cases_List;
    Iterator : Applicability_Constraint_Cases_List_Iterator;
  END_ENTITY;

  TYPE Applicability_Constraint_Cases_List = LIST OF Applicability_Constraint_Case;
  END_TYPE;

  TYPE Applicability_Constraint_Cases_List_Iterator = Iterator;
  END_TYPE;

  ENTITY All_Cases_Structure;
    Cases : Applicability_Constraint_Cases_List;
  END_ENTITY;
END_SCHEMA; _Constraint_Cases_List;
  END_ENTITY;
END_SCHEMA;
```

FIGURE 6.12 – Modèle de structure de cas d'application utilisé pour la mise en œuvre de l'étape 5 de l'algorithme de sélection de tests de faisabilité.

```
SCHEMA CASE_1;

  (* meta-model de cheddar pour la regle preemptive_rate_monotonic *)
  USE FROM Schedulers;
  (* Environnement *)
  USE FROM Mono_Processor_Environment;
  (* Contraintes d'applicabilite pour tous les cas de Time-Triggered-Uniprocessor *)
  USE FROM Time-Triggered-Uniprocessor;
  (* Contraintes specifiques pour ces tests de faisabilite *)
  USE FROM Simultaneous_Release_Time_Constraint;
  USE FROM Period_Equal_Deadline_Constraint;
  (* Enumeration des references sur les tests applicables *)
  USE FROM feasibility_tests_taxonomy ( test_s1, test_R1, test_R2, test_C7, test_C1 );


  RULE preemptive_rate_monotonic FOR ( Mono_Processor_Environment );
  WHERE
    ( 'SCHEDULERS.RATE_MONOTONIC_PROTOCOL' IN TYPEOF ( SELF\environment.scheduler ) ) AND
     ( SELF\environment.scheduler.preemptivity = preemptive );
  END_RULE;
END_SCHEMA;
```

FIGURE 6.13 – Exemple d'instance de modèle de cas d'application.

6.5.3 Extension de la couche haute de Cheddar : algorithmes de sélection de tests de faisabilité

Afin d'évaluer l'approche présentée dans les chapitres 4 et 5, nous avons réalisé deux prototypes implantant l'algorithme de sélection de tests de faisabilité décrit en partie 4.4 et son extension en partie 5.7.

Dans un premier temps, la couche basse a été étendue avec les modèles de données manipulés par les algorithmes de sélection. La couche haute a ensuite été enrichie de deux composants, produits manuellement, réalisant les deux algorithmes. Ces derniers utilisent des paquetages dédiés à la vérification de conformité des contraintes d'applicabilité et de parcours des cas d'application.

La sous partie précédente a présenté les différentes structures de données que nous avons générées afin de réaliser ces algorithmes. L'objectif de cette génération est de permettre la génération du code conséquent à l'ajout d'un nouveau patron de conception.

6.6 Proposition d'un générateur d'architecture

Comme précisé lors de la partie précédente, l'évaluation du prototype présenté ci-dessous a été réalisée à l'aide d'un générateur d'architectures. En effet, pour chacune des étapes de conception du prototype, nous avons généré des ensembles d'architectures aux propriétés diverses.

Le générateur propose une cinquantaine de procédures, dont onze sont dédiées à la génération de modèles d'architecture conformes ou non avec un patron de conception donné. Les autres procédures consistent en l'ajout d'un ou plusieurs éléments d'architecture dans

```
--CASE_1
   --("CASE 1");
   constraint_case_temp.all.Name                    :=
      To_Unbounded_String ("Case_1");
   constraint_case_temp.all.Feasibility_Test_Names :=
      To_Unbounded_String
        ("Test_S1 ; Test_R1 ; Test_R2 ; Test_C7 ; Test_C1 ");
   --Simultaneous_Release_Time_Constraint
   constraint_temp.all.Name                    :=
      To_Unbounded_String ("Simultaneous_Release_Time_Constraint");
   constraint_temp.all.Corresponding_Function :=
     applicability_constraint.Simultaneous_Release_Time_Constraint.apply'
     Access;

   Applicability_Constraints_List_Package.Add
     (constraint_case_temp.all.Applicability_Constraints,
      Copy (constraint_temp));
   --Period_Equal_Deadline_Constraint
   constraint_temp.all.Name                    :=
      To_Unbounded_String ("Period_Equal_Deadline_Constraint");
   constraint_temp.all.Corresponding_Function :=
     applicability_constraint.Period_Equal_Deadline_Constraint.apply'Access;

   Applicability_Constraints_List_Package.Add
     (constraint_case_temp.all.Applicability_Constraints,
      Copy (constraint_temp));
   --CASE_1_Constraint
   constraint_temp.all.Name                    :=
      To_Unbounded_String ("CASE_1_Constraint");
   constraint_temp.all.Corresponding_Function := Function_Case_1'Access;

   Applicability_Constraints_List_Package.Add
     (constraint_case_temp.all.Applicability_Constraints,
      Copy (constraint_temp));

   --adding to CASE_1 to All_Cases
   Applicability_Constraint_Cases_List_Package.Add
     (All_Cases.Cases,
      Copy (constraint_case_temp));
   Applicability_Constraints_List_Package.Initialize
     (constraint_case_temp.all.Applicability_Constraints);
```

FIGURE 6.14 – Extrait du code généré en Ada pour l'étape 5 de l'algorithme de sélection de tests de faisabilité.

un modèle d'architecture, dont les caractéristiques peuvent être précisées manuellement ou déterminées de façon aléatoire.

On peut noter, qu'à la différence des travaux de Martineau *et al.* [Mar94], notre générateur ne se concentre pas sur les caractéristiques déterminant l'ordonnançabilité du sytème, mais sur celles déterminant quelles méthodes peuvent la valider. Ainsi, la difficulté ne réside pas en la génération des propriétés telles que le temps de réponse pire cas d'une tâche, son échéance ou encore sa période. En revanche, le protocole d'ordonnancement, la préemptivité du système, les dépendances entre les différentes tâches, leur type et leur nombre doivent pouvoir être définis aléatoirement. De cette façon, nous pouvons faire varier les paramètres déterminants lors de la sélection de tests de faisabilité.

Nous avons donc proposé un ensemble de fonctions pour le tirage aléatoire des propriétés d'un modèle d'architecture selon une loi uniforme. Pour chaque propriété, nous définissons deux fonctions : une retournant une valeur parmi la liste exhaustive des valeurs potentielles de la propriété et une autre parmi une liste réduite aux valeurs connues comme valides pour nos patrons.

Ce générateur à été implanté manuellement et n'a pas bénéficié de l'utilisation de l'ingénierie dirigée par les modèles. Cependant, il repose, comme tout paquetage du canevas Cheddar sur la CDAI, qui est produite automatiquement.

6.7 Conclusion

Dans cette partie, nous avons présenté l'ensemble des mises en œuvre relatives aux patrons de conception et à nos algorithmes de sélection de tests de faisabilité (cf. chapitres 4 et 5).

Tout d'abord, nous avons détaillé Platypus, un méta-atelier utilisé pour l'élaboration et l'évaluation de méta-modèles, ainsi que pour de la génération de code. Platypus est omniprésent dans le développement de Cheddar depuis plusieurs années [PS07].

Le processus de développement du logiciel de vérification Cheddar a été exposé. Cheddar bénéficie de l'ingénierie dirigée par les modèles afin de supporter des modifications du langage d'architecture sur lequel il repose.

Nous avons explicité de nouveaux concepts dans Cheddar ADL. Ces nouveaux concepts ont permis d'expliciter l'ensemble des comportements nécessaires à l'analyse d'ordonnançabilité. On lève ainsi toute ambiguïté dans l'analyse. Par exemple, le niveau d'abstraction des dépendances correspond à celui considéré dans la théorie de l'ordonnancement temps-réel. Ce niveau d'abstraction permet de ne pas considérer les différentes façons d'implanter, et donc de modéliser les mécanismes impliquant ces dépendances.

Nous avons exposé la structure du canevas mis en place pour la modélisation des patrons de conception et la sélection automatique de tests de faisabilité. Nous avons généré le code pour quatre mécanismes nécessaires à nos travaux :

1. Le graphe de dépendances ;
2. Les contraintes d'applicabilité ;
3. Les cas d'application ;

4. Les nœuds d'association et le graphe de contraintes.

Cette implémentation nous a permis de valider notre approche au travers des évaluations présentées dans les deux chapitres précédents (chapitres 4 et 5). Nous avons aussi montré la flexibilité et l'extensibilité du logiciel Cheddar, ainsi que de nos patrons.

Enfin, une présentation du générateur d'architecture utilisé pour la conduite de nos évaluations a également été proposé.

Chapitre 7

Un exemple d'application des patrons de conception architecturaux : la définition de subsets d'AADL

Bien que l'IDM et les ADLs soient une aide pour les ingénieurs, le développement de systèmes temps-réel embarqués critiques (STRECs) reste une tâche difficile. Les concepteurs doivent explorer les solutions architecturales et les analyser tout au long du processus de développement. Ces différentes étapes peuvent être réalisées à l'aide d'une combinaison particulière d'outils, couvrant chacun un domaine d'analyse spécifique. L'un des principaux défis à venir est de définir un ensemble d'outils cohérent, compatible avec les contraintes imposées par l'industrie ou le système en cours de construction.

En guise d'ouverture à nos travaux, nous appliquons notre approche des patrons de conception à ce problème. Nous étudions ainsi un autre exemple d'utilisation des contraintes constituant nos patrons de conception afin de garantir l'intéropérablité entre divers outils basés sur un même ADL pivot. Nous proposons d'expliciter les restrictions et exigences des outils, à l'aide de modèles dédiés nommés subsets. Les subsets sont modélisés comme une combinaison logique de contraintes sur le méta-modèle AADL en utilisant un langage dédié (DSL).

Ce chapitre est organisé de la façon suivante. La partie 7.1 présente les chaînes d'outils de développement de systèmes temps-réel critiques centrées sur AADL. Elle détaille, également, la problématique de l'interopérabilité entre des outils utilisant AADL comme langage pivot. Nous y identifions les problèmes d'interopérabilité dans ce cas précis. La définition des subsets que nous proposons est donnée dans la partie 7.2. La partie 7.3 décrit leur modèle et présente les trois subsets d'AADL à partir desquels nous avons réalisé nos travaux. Le DSL pour la spécification des subsets est ensuite proposé dans la partie 7.4. La partie 7.5 aborde la comparaison de subsets. Enfin, la partie 7.6 est dédiée à l'évaluation de notre approche.

7.1 Chaînes d'outils et langages d'architecture pivots

Dans cette partie, nous exposons la problématique traitée dans ce chapitre. Pour ce faire, nous commençons par décrire les chaînes d'outils existant pour AADL, avant de discuter de leur interopérabilité.

7.1.1 Les chaînes d'outils AADL

AADL permet de modéliser les composants matériels et logiciels des systèmes embarqués temps-réel [SAE09]. Comme tout langage d'architecture temps-réel, les modèles AADL sont constitués de composants réutilisables et inter-connectés.

La figure 7.1 présente des combinaisons potentielles de chaînes d'outils comprenant un outil de conception de modèle, un outil d'analyse et un générateur de code.

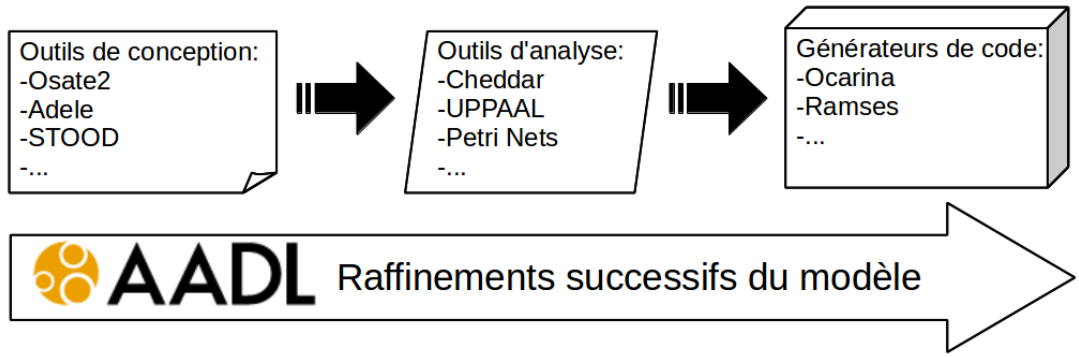

FIGURE 7.1 – Chaînes d'outils centrées autour de l'utilisation d'AADL.

Un modèle AADL peut être utilisé à des fins multiples, telles que la conception, diverses vérifications, l'analyse ou la génération de code. Il existe différents outils AADL couvrant la totalité du cycle en V : des éditeurs de modèles tels que OSATE2 [FG06], ADELE ou STOOD [DS08], des outils d'analyse comme Cheddar [DS08], mais également de nombreuses passerelles depuis AADL vers des outils pré-existants comme UPPAAL [LPY97], des réseaux de Petri [BFLM06], des automates temporisés [BBDZ+10], etc [1].

Enfin, les générateurs de code facilitent le passage de la conception détaillée à l'implémentation. Parmi les générateurs de code dédiés à AADL, on retrouve Ocarina [HZPK08] ou RAMSES [CBGP12].

On peut noter que chaque outil cité ci-dessus, couvre une étape du cycle d'ingénierie. Idéalement, le concepteur itère sur le même modèle d'architecture tout au long des activités du cycle de développement. Par conséquent, le même modèle sera manipulé par différents outils.

1. Une liste complète des outils de vérification de modèles pour AADL est présentée dans [Hug11] et sur http ://www.aadl.info

Du point de vue de la modélisation, la problématique est de garantir que le modèle se prête à toutes les analyses, i.e., remplit les pré-requis de tous les outils.

Dans les faits, un outil d'analyse d'ordonnancement ne supporte qu'un ensemble de modèles de tâches. Par exemple, l'application des politiques de sécurité se concentre sur les patrons de communication, tandis que la gestion des erreurs traite des mécanismes de propagation d'erreurs. Ainsi un concepteur proposant un modèle et souhaitant l'analyser, doit se concentrer sur les concepts acceptés par un outil ou une théorie. Ces mécanismes appartiennent à des niveaux de détail différents, pouvant mener à la spécification de plusieurs modèles, ce qui nuit à l'interopérabilité.

Définition 7.1 ***(Interopérabilité).*** *L'interopérabilité est la capacité de plusieurs composants logiciels, à coopérer malgré des différences de langage, d'interface et de plateforme d'exécution [Weg96].*

7.1.2 Problèmes d'interopérabilité : le cas d'AADL

Dans cette partie, nous détaillons les problèmes d'interopérabilité entre les outils centrés autour d'AADL. Nous en identifions quatre : la complexité du langage, les transformations de modèles appliquées en entrée des outils et l'hétérogénéité des spécifications de sous-ensembles du langage.

La complexité du langage AADL

AADL est un langage riche avec pour ambition de permettre la modélisation d'une large palette d'architectures et de systèmes. Comme tout langage d'architecture, AADL modélise les aspects structurels des composants logiciels et matériels d'une architecture. La partie comportementale de l'architecture est spécifiée par la sémantique propre à chaque composant de l'architecture, leurs propriétés, leurs inter-connexions, la façon dont les ressources d'exécution leur sont allouées, ainsi que dans les annexes. Cependant, la caractérisation complète d'un système, ou d'une architecture, requiert une compréhension poussée des concepts AADL. Le méta-modèle d'AADLv2 comprend plus de 100 entités. Par ailleurs, la BNF d'AADL est composée de 185 règles syntaxiques. La partie du standard dédiée au langage de base décrit environ 250 règles d'héritage et plus de 500 règles sémantiques. Le langage étant complexe et volumineux, les outils ne prennent pas en compte tous ses composants. Par exemple, Cheddar ne considère pas les composants de la catégorie *subprogram*.

Plusieurs modèles et niveaux de détail pour un même système

Feiler et al. identifient deux façon d'utiliser AADL pour l'analyse [Fei04] : (1) des analyses peu coûteuses de patrons architecturaux détectant des problèmes systémiques et (2) des analyses plus poussées, basées sur des travaux théoriques : l'analyse de sensitivité, d'ordonnançabilité ou de sûreté, la génération de code, etc.

Cependant, la richesse d'AADL permet de modéliser des informations similaires de façons différentes (le PCA Blackboard peut être modélisé avec l'annexe comportementale ou

à l'aide de propriétés spécifiques par exemple) [DS08, GSP+14]. Cette diversité dans l'utilisation des modèles peut provenir des pratiques d'équipes de concepteurs, et apparaître au cours des différents raffinements lors du cycle de développement ou des objectifs d'analyse.

Par exemple, un *data_component* utilisé par un seul *thread* sera inutile à l'analyse d'ordonnancement et ne sera donc pas modélisé dans Cheddar. Par contre, il pourra être nécessaire à l'analyse d'aspects fonctionnels ou à la génération de code.

L'impact des transformations de modèles sur un ADL pivot

Certains outils AADL se basent sur des méthodes de transformations de modèles, afin de transformer un modèle AADL en modèle compatible avec leur analyse [KBG12b]. Ces transformations sont basées sur des stratégies de reconnaissance de patrons, afin d'extraire les informations nécessaires et de les représenter au niveau d'abstraction supporté par un outil d'analyse. Par exemple, AADLInspector *traduit* un modèle AADL en modèle Cheddar ADL afin de permettre l'analyse de son ordonnançabilité.

Par conséquent, si un outil ne supporte qu'un ensemble limité de patrons, ou inversement, si un modèle en utilise un de façon alambiquée, la transformation peut être incomplète ou incorrecte. Cela mènerait à l'instanciation d'un modèle d'analyse inexact, et donc à des résultats d'analyse inexacts également. En effet, les transformations de modèles sont ad-hoc par rapport à l'outil que l'on souhaite utiliser.

À titre d'exemple, Muëller étudie l'analyse d'ordonnancement de messages sur un bus de communication entre des processeurs [Mue14]. Pour ce faire, il abstrait un bus CAN comme un processeur utilisant un ordonnanceur non-préemptif, et les messages comme des tâches périodiques. Ensuite, il utilise Cheddar pour appliquer des tests de faisabilité évaluant la charge processeur et les temps de réponse pire-cas des tâches.

Pour conclure, on voit que les modèles AADL doivent être conçus en concordance avec les restrictions et capacités des outils utilisés. C'est pourquoi, les choix de conceptions dépendent fortement des outils utilisés. On peut noter que ces problématiques sont communes à toutes les chaînes d'outils reposant sur un ADL pivot comme AADL, mais aussi MARTE [Obj05] ou AUTOSAR [Bun11].

7.2 Proposition : les subsets AADL

L'approche qui consiste à formaliser un sous-ensemble de langage a été abordée par de nombreux travaux [DPP+09, HGPDM01].

Par exemple, Burns *et al.* définissent Ravenscar, un sous-ensemble du langage de programmation généraliste Ada [BDR98]. Ce sous-ensemble d'Ada permet à des développeurs d'écrire des programmes Ada conformes aux exigences des STRECs.

Delange et al. [DPP+09] ont proposé des directives pour la modélisation de systèmes ARINC653 avec AADL. Ces directives spécifient un sous-ensemble d'AADL modélisant tous les concepts d'une architecture de type ARINC 653.

Enfin, un ensemble de restrictions pour l'application des analyses d'ordonnancement proposées par l'outil MAST a également été rédigé [HGPDM01]. Ces restrictions sont

rédigées comme un ensemble de règles exprimées en langage naturel.

Chacune de ces solutions propose de spécifier les types d'architecture qu'elles supportent à l'aide de sous-ensembles de différents langages, ce qui complique leur utilisation pour résoudre le problème de l'interopérabilité entre ces outils.

Afin de faciliter la conception de chaînes d'outils AADL, les modèles doivent être accordés pour se conformer à des capacités de l'outil spécifiques ou des modèles reconnus et de ne pas appliquer les exigences spécifiques au système. Pour résoudre ce problème, nous visons à rendre explicite le sous-ensemble du langage de modélisation qui est accepté par chaque outil, et de fournir une validation d'un modèle vis-à-vis d'un outil. Les bénéfices attendus sont nombreux :

— déterminer à l'avance si un modèle est compatible avec une analyse ou un outil,
— comparer la puissance d'expression de sous-ensembles reconnus par les outils,
— détecter les situations où un outil accepte à tort un modèle (en dépit de l'absence d'informations cruciales pour l'analyse par exemple).

Nous proposons d'appliquer notre approche des patrons de conception, limitée jusqu'ici à l'analyse d'ordonnançabilité, aux outils AADL. C'est-à-dire, nous proposons d'expliciter les hypothèses faites par les outils sur les modèles d'architectures à l'aide d'ensembles de contraintes. Ces contraintes modélisent un sous-ensemble des modèles représentables à l'aide du langage AADL. Nous nommons ces modèles les *subsets*. Les contraintes constituant les *subsets* peuvent alors être évaluées, par exemple afin de construire un outil de test de conformité d'un modèle à un outil AADL.

Définition 7.2 ***(Subset AADL).*** *Un subset AADL modélise l'ensemble des modèles AADL supportés par un outil utilisant ce langage d'architecture [GPS+13].*

Cette modélisation sert deux objectifs complémentaires :

— on sait précisément si un modèle est conforme à une analyse donnée, i.e., toutes les informations nécessaires à l'analyse sont présentes dans le modèle ; ou, au contraire, détecter les utilisations interdites du langage d'architecture ;
— en modélisant un système conforme à de multiples subsets, on étend le nombre d'analyses pouvant y être appliquées.

7.3 Comment modéliser les subsets AADL ?

Dans cette partie, nous étudions comment modéliser les subsets AADL à l'aide d'une étude de cas.

Dans un premier temps, nous présentons trois subsets nous ayant été fournis par des concepteurs d'outils AADL. Ces subsets ont été rédigés sous la forme de contraintes exprimées en langage naturel. Nous étudions leurs différences et identifions les différents types de contraintes utilisées. Enfin, nous expliquons comment et pourquoi ces contraintes peuvent être normalisées sous la forme de contraintes que nous appelons *contraintes de cardinalité*.

7.3.1 Grille de lecture des subsets exprimés en langage naturel

Les trois subsets présentés ci-dessous sont exprimés à l'aide du langage naturel. Afin d'aider l'utilisateur de subsets lors de leur lecture, ils sont présentés de la façon suivante : (1) les subsets et chacune de leurs contraintes se voient arbitrairement affecter un identifiant unique, (2) les contraintes sont classées par les entités qu'elles restreignent.
Dans la suite de ce chapitre, chacun des exemples de l'étude de cas suivante est présenté sous cette forme.

Entité AADL 1	
Identifiant de contrainte-1 :	description de la contrainte en langage naturel.
...	
Identifiant de contrainte-N :	description de la contrainte en langage naturel.
Entité AADL 2	
...	

7.3.2 Étude de cas : trois exemples de subsets d'outils AADL

Nous présentons, ici, les subsets relatifs à trois outils : (1) MarzhinV1 [Ell], (2) BLESS [LCH] et (3) Cheddar. Pour Cheddar, les subsets modélisent nos patrons de conception [GSP+11b]. Ces trois outils sont respectivement dédiés à la simulation d'ordonnancement (1), à la vérification formelle (2) et à l'analyse d'ordonnançabilité (3).

Le subset de MarzhinV1

MarzhinV1 est la première version d'un outil de simulation d'ordonnancement pour des modèles AADL [DMR+14]. Cet outil est intégré à AADLInspector et n'implante pas l'intégralité du standard AADL. L'objectif de ce subset est donc d'assurer qu'un modèle AADL donné est simulable avec MarzhinV1. Ce subset est constitué de douze contraintes. Ces contraintes restreignent l'utilisation de trois ensembles de propriétés AADL : les propriétés contenues dans l'ensemble AADL_Project, l'ensemble des propriétés ARINC653 ainsi que l'ensemble de propriétés Cheddar (inclus dans le standard AADL) [Sin]. Elles limitent également l'utilisation de l'annexe comportementale à un ensemble limité d'opérateurs comportementaux. Les contraintes spécifiant le subset de MarzhinV1 sont les suivantes :

Processor	
MA1	Il y a un unique composant processeur.
MA2	La propriété *Actual_Processor_Binding* doit être spécifiée.
MA3	Pour tous les composants processeurs, la propriété *Scheduling_Protocol* doit avoir une des valeurs suivantes : *POSIX_Fixed_Priority_Scheduling_Protocol*, *Rate_Monotonic_Protocol* ou *Deadline_Monotonic_Protocol*.
Thread	
MA4	La propriété *Dispatch_Protocol* doit avoir une des valeurs suivantes : *Periodic*, *Aperiodic*, *Timed*, *Hybrid*, *Background*.
MA5	Pour les *threads*, seules les propriétés suivantes sont autorisées : *Dispatch_Protocol*, *Period*, *Deadline*, *Priority* et *Compute_Execution_Time*.
MA6	Seules les *features* suivantes sont autorisées : *Event_Port*, *Event_Data_Port*, *Provides_Subprogram_Access*, *Requires_Subprogram_Access*, *Requires_Data_Access*.
MA7	Pour les *features*, seules les propriétés suivantes sont autorisées : *Queue_Size*, *Dequeue_Protocol*
MA8	La propriété *Dequeue_Protocol* doit avoir une des valeurs suivantes : *OneItem* ou *AllItems*.
Data	
MA9	La propriété *Concurrency_Control_Protocol* doit avoir la valeur suivante : *Priority_Ceiling_Protocol*
Subprogram	
MA10	Les appels à des *subprograms* doivent s'effectuer entre deux *threads*
MA11	Un appel à un *subprogram* doit être exprimé au sein du *classifier* à l'aide d'une des propriétés de *subprogram* suivantes : *Behavior_specification* ou *Compute_Execution_Time*
Behavior Annex	
MA12	Les actions de l'annexe comportementale autorisées sont : Computation, ! <, >!, !

Le subset de BLESS

L'outil BLESS est un outil de vérification formelle conçu pour l'analyse de modèles AADL dédiés dans le domaine médical. Cet outil suppose que, pour pouvoir être analysé, un

modèle AADL n'utilise que les constructions contenues dans AADL-Light [LCH]. AADL-Light capture les parties les plus utilisées d'AADL, dans une optique d'apprentissage pour utilisateurs débutants. Il interdit l'utilisation de pans entiers d'AADL (les *subprograms* par exemple). Ce subset est constitué de 11 contraintes :

Toute catégorie	**de composant**
BL1	Il n'y a pas d'héritage entre les composants.
BL2	Il n'y a pas de refinement.
BL3	Il n'y a pas de composant générique (*generic component*).
BL4	Il n'y a pas de composant abstrait (*abstract component*).
BL5	Il n'y a pas d'in-binding.
BL6	Il n'y a pas d'association de priorités.
Threads	
BL7	Les threads n'ont pas de modes.
Features	
BL8	Il n'y a pas de *features* d'accès à des *subcomponents*.
Subprograms	
BL9	Il n'y a pas de séquences d'appel à des subprograms.
BL10	Il n'y a pas de flows.

Les subsets de Cheddar

Les subsets définis pour Cheddar sont, en fait, les différents patrons de conception présentés dans le chapitre 4 de ce manuscrit. L'objectif de ces subsets est d'assurer l'applicabilité et la correction de l'analyse d'ordonnaçabilité sur des modèles AADL. Dans ce chapitre, nous ne présentons que le patron *Time-Triggered uniprocessor*. Il est donc constitué de 11 contraintes :

Processor	
TT1	Il y a un unique processeur.
TT8	La propriété *Scheduling_Protocol* doit être spécifiée et avoir une valeur parmi la liste suivante : *POSIX_Fixed_Priority_Scheduling*, *Rate_Monotonic*, *Earliest_Deadline_First* ou *Deadline_Monotonic* .
TT9	La propriété *Preemptivity* doit être spécifiée.
TT10	La valeur de la propriété *Quantum* doit être de 0.
TT11	Il n'y a pas de *Virtual_Processor* (il n'y a pas d'ordonnancement hiérarchique).
Thread	
TT2	Les threads sont périodiques.
Feature	
TT3	Toutes les connexions doivent être des *Data_Port_Connections*.
TT4	Il n'y a pas de *Data_Component*.
TT5	Les seules *features* autorisées sont des *Data_Port*.
TT6	Il y a au moins une connexion entre deux *Data_Ports*.
TT7	Pour tous les *Data_Ports*, la propriété *Timing* doit avoir une valeur parmi la liste suivante : *sampled*, ou *delayed*.

On peut noter que l'exemple de modèle AADL présenté dans la partie 2.6.2 est conforme au subset MarzhinV1, mais n'est pas analysable avec Cheddar : il n'est conforme à aucun des subsets Cheddar. En effet, le thread $T2$ n'est ni périodique, ni sporadique. De plus, il n'y a pas de *Data_Port* modélisant une communication de type $Time - Triggered$. Cela illustre deux types de problèmes d'interopérabilité : l'absence d'un élément requis par une analyse et la présence d'un élément qui interdit une analyse.

7.3.3 Analyse des subsets de l'étude de cas

Dans cette partie, nous identifions les besoins relatifs à la définition de susbets, en se basant sur l'étude des trois exemples présentés précédemment.

Dans ces spécifications, les contraintes sont exprimées à l'aide du langage naturel. Cette forme de spécification peut mener à de sérieux problèmes d'interprétation, et donc d'implantation. En effet, une même contrainte peut être exprimée de différentes façons. De plus, il est difficile de mener une étude comparative de ces subsets.

Par exemple, les restrictions MA4 et TT2 limitent toutes les deux les types de comportements temporels sur l'activation des threads. MA4 donne une liste de valeurs potentielles de propriétés, alors que TT2 requiert qu'un thread ait un comportement donné.

On peut également noter que si un modèle AADL est conforme à TT2, cela implique qu'il soit également conforme à MA4. Cette information est cruciale pour identifier qu'un outil requérant TT2 puisse interopérer avec un outil requérant MA4. Afin de permettre une spécification non-ambiguë des subsets, nous proposons un langage dédié nommé ASSET.

Notre proposition se base sur les observations faites sur l'étude de cas.

Premièrement, la diversité des objectifs pour lesquels les subsets ont été définis induit des restrictions sur des concepts distincts. Les subsets de MarzhinV1 et de Cheddar restreignent les modèles d'instances d'AADL, alors que le subset de BLESS interdit l'utilisation de constructions du langage AADL (l'héritage, le raffinement, etc). Le subset de BLESS restreint donc le modèle déclaratif d'AADL. Le modèle d'instances peut être déduit du modèle déclaratif : les instances de composants sont des sous-composants d'un composant *system*. Nous en déduisons donc que toutes les restrictions doivent porter sur le modèle déclaratif d'AADL.

Deuxièmement, nous avons identifié deux types de contraintes :

— certaines contraintes interdisent ou requièrent la présence d'une entité AADL. Nous les nommons *contraintes globales*, car elles doivent être respectées par le modèle AADL dans son intégralité (les contraintes TT1, TT3 et MA1 par exemple).

— D'autres contraintes interdisent ou requièrent une valeur particulière pour un attribut, ou portent sur une relation entre deux entités AADL. Nous les nommons *contraintes locales* (les contraintes TT2, TT6 et MA10 par exemple).

Dans le but d'exprimer ces deux types de contraintes de façon uniforme, nous proposons de les spécifier comme des *contraintes de cardinalité* sur le méta-modèle d'AADL.

Définition 7.3 ***(Contrainte de cardinalité).*** *Une contrainte de cardinalité C_c est une équation booléenne qui restreint le cardinal d'un ensemble d'éléments donné. Ces élements peuvent être des entités d'un ADL ou des relations entre plusieurs entités d'un ADL.*

La transformation de contraintes globales en contraintes de cardinalité est naturelle, étant donné qu'elles requièrent la présence ou l'absence d'entités AADL. Les contraintes locales restreignent une relation entre des entités AADL, ou entre un entité AADL et un ou plusieurs de ses attributs. On peut alors exprimer ces contraintes comme un contrainte de cardinalité, non pas sur un entité, mais sur le cardinal d'une relation entre plusieurs entités AADL.

7.4 ASSET : un langage dédié pour la spécification des subsets AADL

Dans cette partie, nous présentons le langage de spécification de subsets AADL : ASSET (AADL SubSET). Une spécification ASSET est constituée d'un nom associé à toutes les contraintes qu'un modèle AADL doit respecter pour être conforme au subset, ainsi qu'à une description littérale du subset et de ses objectifs.

La figure 7.2 présente la BNF du langage ASSET. Elle est constituée de 16 règles syntaxiques. Lors de la conception d'ASSET, nous nous sommes efforcés à proposer des constructions proches du langage naturel afin que les contraintes puissent être lisibles sans nécessiter un long apprentissage du langage.

```
subsetDef::= subset subsetIdentifier ( {constraintDef}+ )

constraintDef::=  constraintIdentifier : constraintBody ;

constraintBody::=
 there must be cardinalityExpression setDef
| the category of setDef must be typeDef
| for all elementIdentifier in setDef : ( constraintBodyAux )

constraintBodyAux::=
 constraintBody
| the value of elementIdentifier must be valueOperator
| the value of elementIdentifier must be specified
| ( constraintBodyAux and constraintBodyAux )
| ( constraintBodyAux or constraintBodyAux )

cardinalityExpression::= cardinalityOp cardinalityNumber

cardinalityNumber::= <integer>

cardinalityOp::= enumeration (exactly, at most, at least)

setDef::=
typeDef
| ownerIdentifier typeDef
| ( setDef elementIdentifier such that constraintBodyAux )

valueOperator::=
 valueContent
| less than valueContent
| greater than valueContent

valueContent::=
 <integer>
| <real>
| <string>
| <identifier>
| the value of elementIdentifier

ownerIdentifier::= <identifier>.

simpleIdentifier::= <identifier>

typeDef::= <identifier>

subsetIdentifier::= <identifier>

elementIdentifier::= <identifier>| ownerIdentifier elementIdentifier

constraintIdentifier::= <identifier>
```

FIGURE 7.2 – BNF du langage ASSET.

Toute contrainte exprimée à l'aide d'ASSET peut être transformée en contrainte de cardinalité sur des éléments du méta-modèle AADL (cf figure 2.5). Une contrainte de cardinalité peut être construite à l'aide des cinq constructions suivantes uniquement :

— Un test sur l'égalité entre la valeur d'une propriété ou d'un attribut et une valeur, ou entre les valeurs de deux attributs/propriétés.
— Une fonction retournant le cardinal d'un ensemble E d'entités AADL : $\| E \|$.
— Un test sur la comparaison entre deux entiers. Les opérateurs autorisés sont (nommés ops par la suite) : $<, >, =$.
— Un test sur le type T des entités d'un ensemble S : $hasType(T, S)$.
— Un requête sur un ensemble d'entités S qui retourne le sous-ensemble d'éléments $e_i \in S'$, avec $S' \subset S$ et avec tout e_i respectant une contrainte C : $S' = query(S, C)$

Les opérateurs sur les entiers $\{+, -, *, /\}$ et sur les booléens : union, intersection et négation sont autorisés.

À l'aide de ces cinq constructions, il est possible de produire deux types de contraintes.

7.4.1 Contraintes globales et contraintes locales

Il y a deux types de contraintes pouvant être définies au sein d'un subset : des contraintes globales et des contraintes locales.

Définition 7.4 ***(Contrainte globales).*** *Les contraintes globales permettent trois types de restriction :*

- *interdire la présence d'une entité ou d'une quantité d'entités ;*
- *requérir la présence d'une entité ou d'une quantité d'entités ;*
- *imposer sous-type pour toutes les entités partageant un type donné.*

Cela peut être la présence d'un seul et unique processeur, ou le fait que toutes les *features* doivent être des data access par exemple.

Définition 7.5 ***(Contrainte locales).*** *Les contraintes locales permettent quatre types de restriction :*

- *interdire une valeur particulière pour un attribut d'une entité ;*
- *requérir une valeur particulière pour un attribut d'une entité ;*
- *interdire la présence d'une relation entre deux entités ;*
- *requérir la présence d'une relation entre deux entités.*

Elles peuvent, par exemple, contraindre la valeur de la préemptivité d'un processeur, ou le fait que chaque donnée partagée doit être reliée à au moins deux threads.

Les parties suivantes détaillent la spécification de ces deux types de contraintes à l'aide d'ASSET.

7.4.2 Les contraintes globales

Toute contrainte est définie à l'aide d'un *constraintBody*. Les deux premières constructions de *constraintBody* permettent de spécifier une contrainte globale.

Un *ConstraintBodyAux* définit une contrainte qui doit être contenue dans la déclaration d'une autre contrainte. Elle peut soit être une construction de type constraintBody ou une des constructions suivantes :

— l'exigence de la spécification d'une valeur pour un attribut ;
— l'exigence d'une valeur donnée pour un attribut ;
— l'union de deux constraintBodyAux ;
— l'intersection de deux constraintBodyAux ;

Un *setDef* spécifie un ensemble d'entités AADL. Il peut être défini de trois façons différentes :

— un catégorie AADL, l'ensemble correspondant est alors constitué de tous les éléments AADL de cette catégories ;
— Un attribut d'un élément AADL ou d'une catégorie AADL ;
— un ensemble d'élément AADL dont les membres respectent tous une contrainte ;

Exemples de spécifications de contraintes globales

Dans l'exemple suivant, la contrainte globale de MarzhinV1 *MA1* est présentée en détails. Afin d'être conforme à cette contrainte, un modèle AADL ne doit contenir qu'un seul processeur, ce qui est exprimé comme suit avec ASSET :

```
Subset MarzhinV1 (
  MA1 : There must be exactly 1 Processor;
...)
```

Ce type de contrainte permet une restriction sur le cardinal d'un ensemble d'élément AADL. Par exemple, *MA1* contraint la taille de l'ensemble de tous les processeurs qui doit être égale à un. En dehors de *'exactly'*, il exsite deux autres restrictions sur le cardinal d'un ensemble : 'at most' et 'at least' qui spécifient, respectivement, un cardinal minimal et un cardinal maximal pour un ensemble d"élement d'AADL.

Ainsi, la contrainte *TT5* se spécifie de la façon suivante :

```
Subset Time_Triggered (
  ...
  TT5 : There must be at least 1 Port_Connection;
...)
```

Le second type de contrainte globale consiste en la spécification d'un sous type pour un ensemble d'entités AADL. Par exemple, la contrainte Time-Triggered *TT4* spécifie que les *features* doivent toutes être des *data_ports* :

```
Subset Time_Triggered (
  ...
  TT4 : the category of Feature must be Data_Port;
...)
```

Enfin, des opérateurs logiques binaires *AND* et *OR* peuvent être utilisés pour spécifier une contrainte plus complexe. La contrainte MarzhinV1 *MA6* peut donc être déclarée comme suit :

```
Subset MarzhinV1 ( ...
  MA6 : (((( the category of Feature must be Event_Port
   or the category of Feature must be Event_Data_Port)
   or the category of Feature must be Provides_Subprogram_Access)
   or the category of Feature must be Requires_Subprogram_Access)
   or the category of Feature must be Requires_Data_Access);
...)
```

7.4.3 Les contraintes locales

Les contraintes locales sont construites à l'aide de la dernière construction de constraintBody : for all variableDeclaration in setDef : (constraintBodyAux). Une contrainte locale restreint le cardinal d'une relation entre plusieurs entités AADL.

Exemples de spécifications de contraintes locales

Nous donnons maintenant des exemples de contraintes locales. Une contrainte locale est déclarée, au sein d'un subset AADL, de la même façon qu'une contrainte globale.

```
Subset Time_Triggered ( ...
  TT2 : For all T in Thread : ( the value of
    T.Dispatch_Protocol must be Periodic );
     ...)
```

La contrainte spécifiée ci-dessus impose une valeur d'attribut à un ensemble d'entités AADL. La contrainte TT2 requière que tous les threads soient périodiques en n'autorisant qu'une valeur possible pour la propriété *Dispatch_Protocol*.

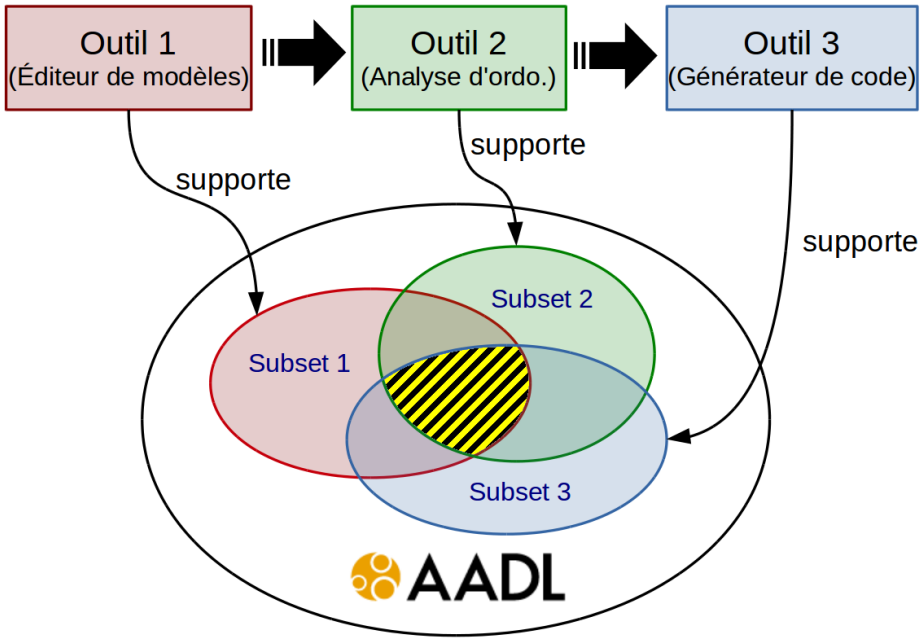

FIGURE 7.3 – Illustration d'un subset commun à une chaîne d'outils.

Une contrainte locale peut également restreindre le modèle déclaratif d'AADL. L'exemple suivant interdit l'héritage d'un classifier de composant AADL, comme spécifié dans le subset BLESS.

```
Subset BLESS ( ...
  BL2 : For all C in Component_Classifier :
    ( C.inherits must be empty );
...)
```

La spécification complète du subset MarzhinV1 à l'aide du langage ASSET, est fournie en fin de chapitre dans les figures 7.10 et 7.11.

Dans cette partie, nous avons exposé comment spécifier un subset AADL à l'aide d'ASSET. La partie suivante est consacrée à la comparaison des subsets.

7.5 La comparaison des subsets AADL

Cette partie est dédié à la description de la comparaison de ces subsets.

Pour qu'un modèle de système soit supporté par tous les outils dans une chaîne d'outils, il doit être conforme aux subsets de tous les outils la composant. La figure 7.3 présente une chaîne d'outils constituée d'un éditeur de modèle, d'un outil d'analyse d'ordonnancement et d'un générateur de code. Les subsets 1, 2 et 3 expriment respectivement les ensembles de modèles AADL supportés par les outils 1, 2 et 3. Les ellipses représentent des ensembles de modèles AADL. L'ellipse blanche correspond à l'intégralité des modèles exprimables à l'aide du langage. Chacune des trois autres ellipses représente le sous-ensemble des modèles conformes à un des trois subsets, i.e., respectant l'intégralité de ses contraintes.

Afin d'être utilisable avec chacun des outils de la chaîne, un modèle AADL doit respecter la globalité des contraintes spécifiées par les trois susbets. Dans la figure, cet ensemble de modèles est représenté par la zone hachurée. Pour pouvoir raisonner sur le subset de la chaîne d'outils, il est nécessaire de pouvoir comparer les subsets des outils la constituant. Comparer les subsets revient alors à comparer les contraintes qu'ils spécifient.

Dans la suite de cette partie, nous exposons une approche de comparaison de contraintes puis de comparaison de susbsets.

7.5.1 La comparaison de contraintes

Il existe quatre types de relations entre les contraintes comparées deux à deux :
- les contraintes sont identiques (e.g. *TT1* et *MA1*)
- une contrainte c_1 implique la seconde contrainte c_2 (e.g. *TT2* implique *MA4*)
- une contrainte implique le non respect d'une autre contrainte : les contraintes sont dites incompatibles (e.g. *TT5* et *MA6* sont incompatibles)
- les contraintes sont indépendantes (e.g. *BL2* et *MA1*)

À l'aide de ces quatre relations, nous définissons des opérateurs de comparaison de subsets.

7.5.2 La comparaison de subsets

Nous définissons quatre opérateurs de comparaison de subsets :

1. l'équivalence entre deux subsets,
2. l'inclusion d'un subset dans un autre subset,
3. l'incompatibilité entre deux subsets et
4. l'intersection de deux susbets.

Définition 7.6 ***(Équivalence entre deux subsets).*** *Un subset* A *et un subset* B *sont dit équivalents si et seulement si* A *et* B *sont spécifiés à l'aide de deux ensembles identiques de contraintes.*

L'équivalence entre les subsets de deux outils signifie qu'il sont complètement inter-opérable. C'est-à-dire que tous les systèmes supportés par l'un des deux outils le seront également par l'autre.

Définition 7.7 ***(Inclusion d'un subset dans un autre subset).*** *Un subset* A *est inclu dans un subset* B *si, pour toute contrainte* c_B *appartenant à* B*, il existe une contrainte* c_A *appartenant au subset* A *telle que le respect de* c_A *implique le respect de* c_B*.*

Ainsi, tout modèle conforme au subset A sera conforme au subset B. L'inclusion du subset d'un outil dans celui d'un autre outil implique que tous les systèmes supportés par le premier outil le seront également par le deuxième.

Définition 7.8 ***(Incompatibilité entre deux subsets).*** *Un subset* A *et un subset* B *sont dits incompatibles s'il existe au moins une contrainte* c_B *appartenant à* B*, et qu'il existe au moins une contrainte* c_A *appartenant à* A*, telles que le respect de* c_B *implique le non-respect d'une contrainte* c_A*.*

Tout modèle conforme au subset A ne pourra pas être conforme au subset B. Une incompatibilité entre les subsets de deux outils signifie que les outils supportent des ensembles disjoints de modèles AADL, et ne pourront donc pas être utilisés au sein d'une même chaîne d'outils.

Définition 7.9 ***(Intersection de deux subsets).*** *Le subset* I *est l'intersection d'un subset* A *et d'un subset* B *si toute contrainte* c_I *de* I*, appartient à* A *et/ou* B*. Le subset* I *est donc inclus dans* A *et dans* B*.*

L'ensemble des modèles conformes à I est donc un sous-ensemble commun aux ensembles de modèles conformes à A et B. Le subset d'une chaîne d'outils est l'intersection des subsets de tous les outils la composant.

La partie suivante présente l'évaluation de notre approche.

7.6 Implantation et évaluation

Dans les parties précédentes, nous avons proposé ASSET, un DSL pour la spécification de subsets AADL, ainsi que des opérateurs de comparaison des subsets et des contraintes qui les composent.

L'évaluation de notre approche vise trois objectifs :

— vérifier qu'ASSET permet bien de spécifier l'ensemble des contraintes des subsets de notre étude de cas,

— étudier la génération du code d'un outil permettant la vérification de conformité d'un modèle AADL à un subset, et

— valider les opérateurs de comparaison de contraintes que nous avons spécifié.

On peut noter que les résultats proposés ici ne portent que sur les subsets de notre étude de cas.

Dans un premier temps, nous présentons notre implantation d'ASSET et nous évaluons cette réalisation au travers de la spécification des contraintes des subsets de l'étude de cas puis la génération de contraintes de cardinalité en EXPRESS et en REAL.

Dans un second temps, nous détaillons les résultats d'une étude de compatibilité entre les susbets de notre étude de cas.

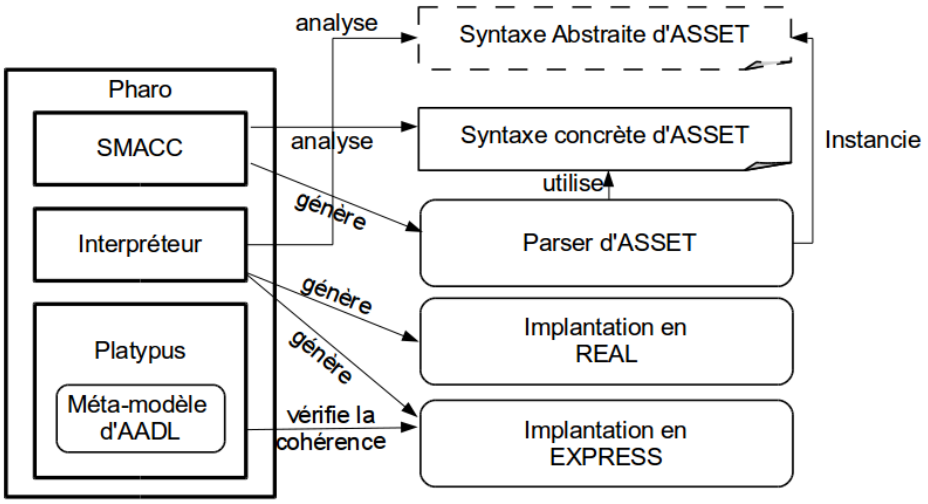

FIGURE 7.4 – Rôles joués par les différents outils lors de l'implantation d'ASSET.

7.6.1 Mise en œuvre d'ASSET

L'implantation du langage ASSET a été réalisée à l'aide de l'outil SmaCC (Smalltalk Compiler-Compiler). SmaCC est un générateur d'interpréteurs libre disponible pour Smalltalk. SmaCC peut générer des analyseurs de grammaires.

la figure 7.4 présente les outils utilisés lors de l'implantation d'ASSET.

SmaCC permet, à partir de la définition de la syntaxe concrète d'un langage de générer un interpréteur. La syntaxe concrète est étendue de règles d'instanciation, permettant d'instancier la syntaxe abstraite du langage lors de l'interprétation. La syntaxe abstraite a été spécifiée à l'aide de Platypus et du langage EXPRESS. La figure 7.5 présente la syntaxe abstraite d'ASSET.

La figure 7.5, quant à elle, présente la syntaxe abstraite du langage ASSET que nous avons modélisé dans Platypus.

Afin de vérifier que les contraintes spécifiées en ASSET peuvent utilisées pour générer le code d'outils de vérification de conformité d'un modèle AADL au subset d'un outil, nous avons généré le code des contraintes pour deux langages cibles : EXPRESS et REAL. Real est un langage de [GH10].

La cohérence des contraintes générées en EXPRESS est réalisée par Platypus vérifiée à l'aide d'un méta-modèle d'AADL. Nous avons utilisé Platypus pour son élaboration [PR06].

La figure 7.6 fournit un exemple de contrainte EXPRESS généré à partir d'ASSET. Les contraintes présentées sont les deux premières contraintes du subset MarzhinV1.

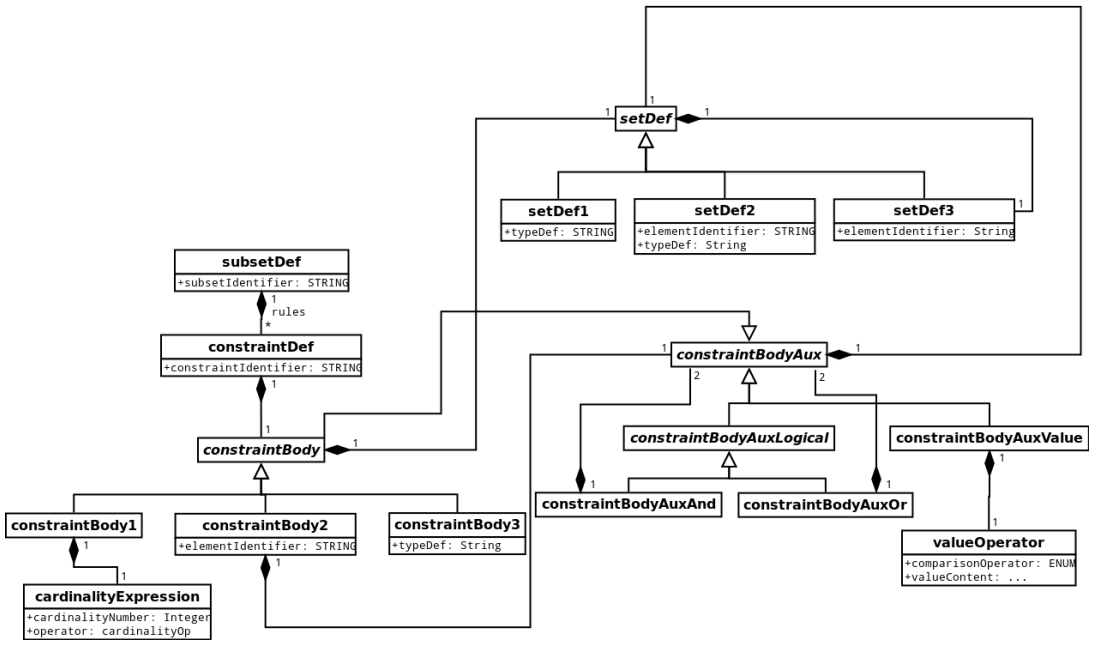

FIGURE 7.5 – Syntaxe Abstraite d'ASSET.

7.6.2 Évaluation d'ASSET

Le premier objectif de cette évaluation est de vérifier qu'ASSET permet de spécifier l'intégralité des contraintes de notre étude de cas. Le second est de montrer que nous sommes capables, à partir d'un spécification ASSET de générer automatiquement le code implantant la vérification des contraintes à l'aide de contraintes de cardinalité.

Pour ce faire, nous avons spécifié les différents subsets proposés par les concepteurs d'outils AADL ainsi que venant de la littérature. Nous avons spécifié les cinq subsets suivants : BLESS, MarzhinV1, Cheddar Time-Triggered, Cheddar Ravenscar et Cheddar Unplugged. Ces trois subsets correspondent, respectivement aux patrons *Time-Triggered uniprocessor*, *Ravenscar uniprocessor* et *Unplugged uniprocessor*.

Pour chacun de ces subsets, ASSET a permis la spécification de toutes leurs contraintes et la génération de leurs formes EXPRESS et REAL.

7.6.3 Comparaison des subsets de l'étude de cas

Nous avons ensuite comparé chacun des subsets ainsi conçus afin de réaliser une étude comparative des différents subsets ainsi spécifiés. Les tableaux 7.7, 7.8 et 7.9 résument les résultats des comparaisons. Les trois tableaux contiennent respectivement le nombre de contraintes identiques, impliquées par une contrainte de l'autre subset et incompatibles entre chaque couple de subsets.

Les diagonales contiennent le nombre de contraintes de chaque subset. On peut noter

```
SCHEMA Subset_MarzhinV1;
  RULE Only_One_Processor FOR ( Processor );
  WHERE
    MA1 : SIZEOF ( Processor ) = 1;
  END_RULE;

  RULE Actual_Processor_Binding_Must_Be_Specified FOR ( Property );
  WHERE
    MA2 : SIZEOF ( QUERY ( p <* Property |
        ( p.Property_Name = 'Actual_Processor_Binding' ) ) ) > 0;
  END_RULE;
END_SCHEMA;
```

FIGURE 7.6 – Exemples de contraintes spécifiées avec une règle globale EXPRESS

Subsets	*BLESS*	*MarzhinV1*	*Time-Triggered*	*Ravenscar*	*Unplugged*
BLESS	11	**0**	**0**	**0**	**2**
MarzhinV1		12	**2**	**2**	**2**
Time-Triggered			10	**5**	**5**
Ravenscar				13	**6**
Unplugged					12

FIGURE 7.7 – Nombre de contraintes identiques entre les subsets.

Subsets	*BLESS*	*MarzhinV1*	*Time-Triggered*	*Ravenscar*	*Unplugged*
BLESS	11	**1**	**4**	**3**	**2**
MarzhinV1		12	**2**	**4**	**6**
Time-Triggered			10	**1**	**1**
Ravenscar				13	**1**
Unplugged					12

FIGURE 7.8 – Nombre de contraintes impliquées par une contrainte d'un autre subset.

Subsets	*BLESS*	*MarzhinV1*	*Time-Triggered*	*Ravenscar*	*Unplugged*
BLESS	11				
MarzhinV1	*0*	12			
Time-Triggered	*0*	*1*	10		
Ravenscar	*1*	*1*	*3*	13	
Unplugged	*0*	*0*	*1*	*1*	12

FIGURE 7.9 – Nombre de contraintes incompatibles entre les subsets.

que les contraintes identiques sont nombreuses pour les subsets spécifiques à Cheddar. En effet, ces subsets partagent tous les mêmes cinq contraintes d'environnement *uniprocessor*.

Résultats de l'évaluation

Cette évaluation montre trois choses.

La première est qu'ASSET permet bien la spécification de subsets AADL réels, conçus avec des perspectives très différentes (BLESS et MarzhinV1 par exemple).

La seconde est que cette évaluation a montré que nous somme capable de spécifier, comparer et générer différentes implantations pour un groupe représentatif de subsets.

La troisième est que cette approche nous a également permis d'identifier des problèmes d'interopérabilité qui n'étaient pas connus a-priori.

7.7 Conclusion

Le développement de systèmes embarqués temps-réel critiques est une tâche difficile. Il existe de nombreux outils dédiés à la conception de tels systèmes. Pourtant, les outils sont habituellement dédiés à une utilisation particulière et possèdent chacun des exigences et restrictions spécifiques sur les modèles qu'ils supportent. Cela conduit à des problèmes d'interopérabilité lors de leur utilisation au sein d'un même processus de développement.

En guise d'ouverture à nos travaux, nous proposons d'appliquer notre approche des patrons de conception à ce problèm. Nous proposons une approche de modélisation des restrictions et exigences de chaque outil avec des modèles dédiés nommés *subsets*. Un subset est défini comme une liste de contraintes sur le méta-modèle d'un ADL. Nous illustrons notre approche avec AADL, un langage standard de conception de systèmes embarqués temps-réel. Nous avons spécifié plusieurs subsets proposés par des concepteurs d'outlis et modélisant leurs restrictions et exigences sur l'utilisation d'AADL. Pour ce faire, nous proposons ASSET, un DSL de spécification des susbsets.

Nous définissons des opérateurs de comparaison de subsets reposant sur la comparaison des contraintes les composant. Nous effectuons, ensuite une étude de compatibilité entre les subsets proposés par les concepteurs d'outils.

Par la mise en œuvre de ce langage, la spécification de cinq subsets et leur comparaison, nous avons montré que notre approche est bien adaptée à la spécification des capacités des outils et de leurs restrictions. Ces travaux ont permis d'identifier des incompatibilités entre ces outils.

```
Subset Marzhin (
—Marzhin is a simulation tool for AADL models. It is integrated to AADLInspector. The intention
    of this —subset is to ensure that an AADL model can be simulated with Marzhin 1.0. Its subset
     is composed of —twelve restrictions. The restrictions are related to three property sets:
    properties defined in the —AADL_Project property set, the ARINC653 property set and some
    properties and in the Cheddar Property —Set. They also restrict the use of the Behavior annex
     to a limited set of behavioral operators. The —restrictions for the Marzhin tool are
    describe below.

—MA1 : There is only one Processor component
Marzhin1 : there must be exactly 1 Processor;

— MA2 : The property Actual_Processor_Binding must be specified
MA2 : for all p in Processor : (the value of P.Actual_Processor_Binding must be specified);
— MA3 : For all processors, property Scheduling_Protocol must have the following values only :
POSIX_Fixed_Priority_Scheduling_Protocol, Rate_Monotonic_Protocol or Deadline_Monotonic_Protocol
MA3 : for all p in Processor : ( ( (  the value of p.schedulingpolicy must be
        Posix_Fixed_Priority_Scheduling_Protocol
        or the value of p.schedulingpolicy must be Rate_Monotonic_Protocol )
        or the value of p.schedulingpolicy must be Deadline_Monotonic_Protocol ) ) ;

— MA4 : The property Dispatch_Protocol must have one of the following values : Periodic,
Aperiodic, Sporadic, Timed, Hybrid, Background.
MA4 : for all t in Thread : ( (((((  the value of t.Dispatch_Protocol must be Periodic
        or the value of t.Dispatch_Protocol must be Aperiodic )
        or the value of t.Dispatch_Protocol must be Sporadic )
        or the value of t.Dispatch_Protocol must be Timed )
        or the value of t.Dispatch_Protocol must be Hybrid )
        or the value of t.Dispatch_Protocol must be Background )) ;
— MA5 : Properties must be one of the following : Dispatch_Protocol, Period, Deadline, Priority,
Compute_Execution_Time
MA5 : for all t in Thread : (   for all p in t.Properties : (((((  the value of p.Name must be
    Dispatch_Protocol
        or the value of p.Name must be Deadline )
        or the value of p.Name must be Period )
        or the value of p.Name must be Priority )
        or the value of p.Name must be Compute_Execution_Time )));
```

FIGURE 7.10 – Spécification complète du subset MARZHINV1 (1/2)

```
-- MA6 : Features must be one of the following : Event Port, Event Data Port, Provides Sub-
program Access, Requires Subprogram Access, Requires Data Access
MA6 : for all f in feature : ((((( the category of f must be Event_Port
        or the category of f must be Event_Data_Port)
        or the category of f must be Provides_Subprogram_Access)
        or the category of f must be Requires_Subprogram_Access)
        or the category of f must be Requires_Data_Access));

-- MA7 : Features Properties must be one of the following : Queue_Size, Dequeue_Protocol
MA7 :    for all f in Feature : ( for all p in f.Property : (( the value of p.Name must be
     Queue_Size
        or the value of p.Name must be Dequeue_Protocol )));

--MA8 : Dequeue_Protocol must have one of the following values : OneItem or AllItems
MA8 : for all f in Feature : ( for all p in
    ( f.Property p2 such that  the value of p2.Name must be Dequeue_Protocol ):
    (( the value of p must be OneItem or the value of p must be AllItem )) );

-- MA9 : The property Concurrency_Control_Protocol must be set to Priority_Ceiling_Protocol
MA9 : for all s in Shared_Data :
  ( the value of s.Concurrency_Control_Protocol must be Priority_Ceiling_Protocol );
)
```

FIGURE 7.11 – Spécification complète du subset MARZHINV1 (2/2)

Chapitre 8

Bilan et perspectives

La thèse présentée dans ce manuscrit traite de l'applicabilité des méthodes d'analyses et de l'interopérabilité des outils dédiés au développement de systèmes temps-réel embarqués critiques (STRECs). Les STRECs possèdent, par définition, une quantité de ressources limitée et doivent produire des résultats corrects tout en respectant des contraintes temporelles strictes. En effet, le respect de ces contraintes est une condition *sine qua non* à leur correction. Dans le cas contraire, les conséquences seront des dégâts irréversibles sur les personnes ou l'environnement.

La théorie de l'ordonnancement temps-réel fournit des méthodes d'analyses appelées tests de faisabilité permettant de valider l'ordonnançabilité de STRECs. Afin d'être analysable avec un test de faisabilité donné, un STREC doit respecter un ensemble d'hypothèses sur ses propriétés architecturales. Nous nommons ces hypothèses contraintes d'applicabilité.

L'ingénierie dirigée par les modèles et les langages de description d'architectures plus particulièrement, permettent la spécification de modèles d'architecture des STRECs en cours de développement. Ces modèles d'architecture peuvent être utilisés pour réaliser les différentes analyses permettant de valider la conception du système et pour générer une partie du code implantant le système réel. Les travaux décrits dans ce manuscrit se concentrent sur deux langages de description d'architecture (ADLs) : AADL et Cheddar ADL.

Bien que mature pour certains types de STRECs, la théorie de l'ordonnancement est peu utilisée par les industriels alors que cela pourrait être profitable. Nous pensons que ce problème est dû à un manque de connaissances des industriels à propos de la théorie de l'ordonnancement et à des lacunes quant à son intégration dans le processus de développement.

En effet, les tests de faisabilité sont spécifiques à certains types de STRECs et la sélection d'un test applicable à un STREC en cours de développement est une tâche ardue.

Notre objectif est donc d'assister les concepteurs de STRECs lors du processus de développement en sélectionnant automatiquement des tests de faisabilité.

Par ailleurs, les chaînes d'outils de développement sont constituées, entre autres, d'éditeurs de modèles, de logiciels d'analyse et de générateurs de code. Afin de permettre leur interopérabilité, ces outils peuvent reposer sur un ADL pivot commun. Cependant, ils sont développés de façon indépendante et ne supportent pas, dans la plupart des cas, l'intégralité du langage pivot.

Un concepteur de STRECs doit donc prendre en compte les sous-ensembles de l'ADL supportés par les différents outils, lors de la modélisation de son système.

Nous montrons dans cette thèse comment notre approche peut être utilisée pour améliorer l'interopérabilité des outils reposant sur un ADL commun.

Afin d'assister les concepteurs de STRECs au cours du processus de développement, nous avons proposé d'expliciter les relations entre les modèles d'architectures et les méthodes d'analyse à l'aide de patrons de conception architecturaux.

Nous proposons des patrons de conception architecturaux permettant la sélection automatique de tests de faisabilité pour des STRECs déployés sur un environnement d'exécution mono-processeur (cf chapitre 4). Un patron de conception architectural est spécifié à l'aide de deux ensembles de contraintes. Le premier restreint la plate-forme d'exécution. Les contraintes le constituant portent sur les entités matérielles du langage d'architecture. Le second ensemble de contraintes restreint les protocoles de communication et de synchronisation entre les tâches. Les contraintes le constituant portent donc sur les entités logicielles du langage d'architecture. Chacune de ces contraintes est exprimée à l'aide du langage de description d'architectures de Cheddar.

Les deux ensembles de contraintes sont associés à une liste de tests de faisabilité applicables aux STRECs y étant conformes.

Un algorithme de sélection de tests de faisabilité basé sur la vérification de la conformité d'un système à un patron a été proposé.

Nous avons ensuite étendu notre approche à des STRECs utilisant des environnements d'exécution plus complexes. Cette extension a été étudiée dans le cas de systèmes temps-réel utilisant un ordonnancement hiérarchique.

Nous proposons alors un modèle de patron sous la forme d'un graphe de contraintes. Un graphe de contraintes est constitué de nœuds d'association, eux-mêmes constitués de deux ensembles de contraintes d'environnement ainsi que de contraintes sur les communications et les synchronisations entre les tâches.

Ce faisant, nous enrichissons les associations entre propriétés architecturales et méthodes d'analyses.

Avec cette proposition, nous avons pu supprimer une hypothèse forte de la problématique traitée précédemment : restrictions de nos propositions à un environnement d'exécution mono-processeur. Cela constitue une première étape vers la prise en compte d'un ensemble plus large de systèmes temps-réel.

Une adaptation de l'algorithme de sélection de tests de faisabilité au cas hiérarchique est également proposée.

Afin de valider nos travaux, nous avons conçu deux prototypes. Ces prototypes ont été intégrés à l'environnement d'analyse d'ordonnancement Cheddar. Ces prototypes permettent la sélection automatique de tests de faisabilité pour des STRECs conformes à nos patrons et modélisés à l'aide de Cheddar ADL.

Au cours de ces travaux, nous avons également implanté deux tests de faisabilité spécifiques aux STRECs à ordonnancement hiérarchiques Nous avons participé à l'évolution de Cheddar ADL par l'ajout de nouveaux concepts afin de modéliser des STRECs à ordonnancement hiérarchique.

Ces réalisations sont donc intégrées à Cheddar et devraient l'être dans AADLInspector, un logiciel de vérification de modèles AADL proposé par la société Ellidiss Technologies.

En guise d'ouverture nous avons montré que notre approche pouvait contribuer à l'interopérabilité des outils intégrés à des chaînes d'outils dédiées au développement de STRECs.

Pour ce faire, nous introduisons la notion de subset de langages de description d'architectures, spécifiés à l'aide de contraintes structurelles semblables à celles utilisées pour la spécification de nos patrons de conception architecturaux.

Nous avons étudié trois exemples de subsets. Les deux premiers subsets sont ceux concernant les outils MarzhinV1 et Cheddar. Le troisième définit AADL-light, un sous-ensemble d'AADL ayant pour objectif de simplifier AADL afin d'en faciliter l'apprentissage.

Nous avons proposé ASSET, un langage dédié à la spécification de sous-ensembles de langages d'architectures. ASSET permet la spécification de contraintes définissant l'ensemble des systèmes supportés par un outil centré autour d'un ADL.

Dans le but de pouvoir raisonner sur les subsets, nous avons proposé des opérateurs de comparaison des subsets et des contraintes les composant. À l'aide de ces opérateurs, nous avons réalisé une étude comparative des trois subsets de Cheddar, AADL-Light et MarzhinV1. Cela nous a permis de détecter des incompatibilités entre les susbets, jusque-là inconnues.

Afin de pérenniser les définitions de subsets réalisées et l'utilisation d'ASSET, une première version d'une annexe au langage AADL, a été proposée au comité de standardisation pour discussion.

Perspectives

Le travail décrit dans ce mémoire peut conduire à de nombreuses perspectives.

La principale perspective de nos travaux est la poursuite de l'extension de notre approche aux types de systèmes temps-réel n'ayant pas encore été pris en compte. On peut citer, tout particulièrement, le cas des systèmes multi-processeurs. Le domaine de recherche traitant de l'analyse temporelle de tels systèmes est très actif et mériterait que l'on s'y penche. La méthode proposée au chapitre 5 consistant à calquer la structure des patrons sur l'architecture matérielle des systèmes concernés est une bonne piste de départ. Cependant, les principes propres aux systèmes mutliprocesseur tels que la migration des tâches peut cacher des problématiques de modélisation non rencontrées jusqu'ici.

Le seconde perspective est d'ouvrir le champ des analyses considérées à des méthodes différentes comme la simulation ou le *model-checking*. En effet, le bilan de l'utilisation de ces approches est le même que celui de la théorie de l'ordonnancement temps-réel. Notre approche devrait pouvoir être appliquée à ces méthodes et aux propriétés bien précises qu'elles requièrent.

Une troisième perspective des travaux relatifs aux patrons de conception architecturaux réside principalement dans la portabilité de nos patrons (exprimés à l'aide de Cheddar ADL) vers d'autres langages de description d'architectures. En effet, l'utilisation de Cheddar ADL, langage enrichi itérativement par les concepts des modèles de tâches manipulés par la théorie de l'ordonnancement, facilite la définition des patrons. La sémantique des entités utilisées est non ambiguë et concorde avec le niveau d'abstraction de la théorie de l'ordonnancement temps-réel.

Dans les faits, les langages d'architectures utilisés pour la conception des systèmes embarqués temps-réel comportent un niveau de détail plus fin et une sémantique pouvant être ambiguë. Un même protocole de communication pourra être modélisé de différentes façons. Détecter de façon sûre quel protocole est utilisé n'est pas une tâche facile.

Les transformations de modèles d'un langage de description d'architecture vers un autre, sont le plus souvent ad-hoc et spécifiques à un outil donné (AADL vers Cheddar ADL par exemple). La définition de patrons de conception mécaniques implémentant les protocoles de communication manipulés au niveau architectural peut fournir une solution à ce problème.

Une quatrième perspective à ces travaux est la conception d'une taxinomie des tests de faisabilité. Il existe de nombreux travaux de classification des algorithmes d'ordonnancement. Cependant, il n'existe pas, à notre connaissance, de classification des tests de faisabilité. Cette classification peut être construite selon deux aspects.

Le premier aspect consiste en une classification des contraintes d'applicabilité des tests de faisabilité considérés. Ces tests sont, quant à eux, classifiés selon des critères différents des contraintes, constituant ainsi le second aspect de la classification. Les critères à prendre en compte pour la classification des tests sont leur complexité et leur pessimisme. L'objectif de cette classification est de pouvoir comparer un ensemble de tests sélectionnés pour un STREC donné, afin de pouvoir raffiner la sélection selon des objectifs de l'utilisateur (analyser le STREC de façon rapide ou précise par exemple).

Enfin, une cinquième perspective concerne les travaux relatifs aux subsets. La création de chaînes d'outils supportant l'ensemble des étapes du processus de développement est une tâche complexe. De plus, la chaîne d'outils est susceptible d'évoluer au cours du processus.

Dans le chapitre 7, nous avons abordé la comparaison de subsets, et donc des contraintes les composant, à l'aide des quatre opérateurs. La conception d'un algorithme de comparaison automatique des contraintes exprimées à l'aide d'ASSET permettrait l'implantation d'un outil de comparaison automatique des subsets.

Les potentielles contributions d'un tel outil sont multiples. Il permettrait la détection d'incompatibilités entre des outils jusqu'à lors non connues, comme ça a été le cas lors de l'évaluation présentée dans le chapitre 7. Une autre utilisation de cet outil serait la vérification automatique du support d'un outil pour un standard donné, déterminer si AADLInspector est-il capable d'analyser des systèmes conformes au standard ARINC653 par exemple. Enfin, il pourrait être utilisé pour détecter des types de systèmes non supportés par un outil, aidant ainsi le concepteur de l'outil à le compléter afin de garantir sa conformité au standard.

Par ailleurs, nous avons généré de l'EXPRESS, afin d'évaluer la génération de code à partir de spécifications ASSET. L'étape suivante est désormais de générer automatiquement un outil d'analyse de conformité d'un modèle AADL au subset d'un outil. Une solution serait de permettre la génération de code vers de multiples langages cibles, afin que chaque outil puisse générer et intégrer un mécanisme d'évaluation de conformité des modèles au subset qu'il supporte.

Publications

Les travaux effectués pendant cette thèse ont donné lieu aux publications listées ci-dessous.

Revues

[SPR+14] F. Singhoff, A. Plantec, S. Rubini, V. Gaudel, S. Li, C. Fotsing, P. Dissaux, J. Legrand, and L. Lemarchand, "How architecture description languages help schedulability analysis : a return of experience from the Cheddar project," soumise à *Science of Computer Programming*, 2014.

[GSP+14] V. Gaudel, F. Singhoff, A. Plantec, P. Dissaux, J. Legrand, Composition of Design Patterns : from the modeling of RTOS synchronization tools to schedulability analysis, *ACM SIGBED Review* 11(1) :44-49. *Special issue of EWiLi'13, The 3rd Embedded Operating Systems Workshop* 26 - 27 August 2013, ENSEEIHT, Toulouse, France.

[GSP+11b] V. Gaudel, F. Singhoff, A. Plantec, P. Dissaux, J. Legrand, An Ada design pattern recognition tool for AADL performance analysis. In : *ACM SIGAda Ada Letters*, volume 31, number 3, pages 61-68. ACM New York, USA, November 2011, ISSN :1094-3641. *Special issue of the proceedings of the international ACM SIGAda conference*, Denver, Colorado, USA. *Best student paper award*

Conférences internationales avec comité de lecture et actes

[DMR+14] Pierre Dissaux, Olivier Marc, Stéphane Rubini, Christian Fotsing, Vincent Gaudel, Frank Singhoff, Alain Plantec, Vuong Nguyen-Hong, Hai-Nam Tran. The SMART project : Multi-agent scheduling simulation of real-time architectures. In*7th European Congress ERTSS Embedded Real Time Software and System*, Toulouse, France, February 2014.

[GPS+13] Vincent Gaudel, Alain Plantec, Frank Singhoff, Jérôme Hugues, Pierre Dissaux, and Jérôme Legrand, Enforcing software engineering tools interoperability : An example with aadl subsets. In *Rapid System Prototyping (RSP), 2013 International Symposium*, pp 59-65. IEEE, Montréal, Canada, 2013.

[KBG12b] Kerboeuf, M., Babau, J.-P., Gaudel, V., A two-steps model transformation

to extend the scope of an analysis framework to standard modeling languages. In *Conférence : 6th MoDELS workshop on Models and Evolution.*, pp. 9-14, Innsbruck, Autriche, Octobre 2012.

[PSG+11] Alain Plantec, Frank Singhoff, V Gaudel, Vincent Ribaud, et al. Forward engineering and early model validation with smalltalk. In *5th Argentine Smalltalk Conference.*, 2011.

[DLG+11] Pierre Dissaux, J Legrand, Vincent Gaudel, Alain Plantec, Stéphane Rubini, Frank Singhoff. AADL real-time design-pattern automatic recognition. In *SAE Technical Paper*, 2011.

Conférences nationales et workshops avec comité de lecture et actes

[CSR+12] Craveiro, J., and Souza, J., Rufino, J., Gaudel, V., Lemarchand, L., Plantec, A., Rubini, S., Singhoff, F., Scheduling Analysis Principles and Tool for Time-and Space-Partitioned Systems. In :*INFORUM 2012 symposium*, pp. 582-585, Lisbonne, Portugal, 2012.

[GSP+11a] Vincent Gaudel, Frank Singhoff, Alain Plantec, Pierre Dissaux, and Jérôme Legrand. Sélection automatique de tests de faisabilitéa l'aide de patrons de conception. In : *Ecole d'été Temps Réel 2011 (ETR'11)*, pages 185–188, 2011.

Standards

[Gau14] Gaudel, V., Dissaux, P., Plantec, A., P., Singhoff, F., Hugues, J., Legrand, J., AADL Subset Annex (First Draft) présenté au meeting AADL Standard, User Day, Toulouse (France) February, 2014.

Séminaires et communications sans actes

— AADL Subset Annex Update. V. Gaudel, P. Dissaux, A. Plantec, F. Singhoff, J. Hugues, J. Legrand. AS-2C/AADL SAE Summer working group meeting. Orlando, USA, Juillet, 2014.

— AADL Subset Annex. V. Gaudel, P. Dissaux, A. Plantec, F. Singhoff, J. Hugues, J. Legrand. AS-2C/AADL SAE Autumn working group meeting. Montréal, Canada, Octobre, 2014.

— What is an AADL Subset ? V. Gaudel, P. Dissaux, A. Plantec, F. Singhoff, J. Hugues, J. Legrand. AS-2C/AADL SAE Winter working group meeting. Valencia, Espagne, Février, 2013.

— Automatic Selection of Feasibility Tests With the Use of AADL Design Patterns. V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, J. Legrand. AS-2C/AADL SAE Spring working group meeting. Paris, France, june 2011.

Bibliographie

[AB90] N. Audsley and A. Burns. Real-time system scheduling. *Computer Science Department Technical Report No. YCS134, York University, UK*, 1990.

[ABR+93a] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5) :284–292, 1993.

[ABR+93b] N. Audsley, A. Burns, M. Richarson, K. Tindel, and J. Wellings. Applying New Scheduling Theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5) :284–292, 1993.

[ACPR13] K. Almeida, J. Craveiro, R. Pinto, and J. Rufino. Spaceborne software : typical spacecraft use-case and preliminary analysis of its timing requirements. In *Technical Report READAPT Project TR-13-01*, Lisbon, Portugal, 2013.

[AFM+04] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times : a tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems*, pages 60–72. Springer, 2004.

[Ale79] C. Alexander. *The timeless way of building*, volume 1. Oxford University Press, USA, 1979.

[Ari97] Arinc. *Avionics Application Software Standard Interface*. The Arinc Committee, January 1997.

[Bak91] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1) :67–99, 1991.

[Bar03] Sanjoy K Baruah. Dynamic-and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1) :93–128, 2003.

[Bar06] S. K. Baruah. Resource Sharing in EDF-Scheduled Systems : A Closer Look. pages 379–387. 27th IEEE International Real-Time Systems Symposium, December 2006.

[BBD11] Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34–43. IEEE, 2011.

[BBDZ+10] Bernard Berthomieu, Jean-Paul Bodeveix, Silvano Dal Zilio, Pierre Dissaux, Mamoun Filali, P Gaufillet, S Heim, and Franois Vernadat. Formal verification of aadl models with fiacre and tina. In *5th International Congress and exhibition ERTS2*, 2010.

[BBJK05] Jean Bézivin, Hugo Bruneliere, Frédéric Jouault, and Ivan Kurtev. Model engineering support for tool interoperability. In *Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica*, volume 2, 2005.

[BDR98] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies-Ada-Europe*, pages 263–275. Springer, 1998.

[Ben11] P. Benes. Porting of resource reservation framework to rtems executive. Master's thesis, Lulea University of Technology, 2011.

[BFLM06] J.P. Bodeveix, M. Filali, J. Lawall, and G. Muller. Vérification automatique de propriétés d'ordonnanceurs Bossa. In *AFADL, ENST Paris*, pages 95–109. ENST, 15-17 mars 2006.

[BHR90] S.K. Baruah, R. R. Howell, and L. E. Rosier. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real Time Tasks on one Processor. *Real Time Systems journal*, 2 :301–324, 1990.

[BMR90] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE, 1990.

[Boe81] Barry W Boehm. Software engineering economics. 1981.

[Bou95] Med Bouazza. *La norme STEP*. Hermès, 1995.

[Bun11] Stefan Bunzel. Autosar–the standardized software architecture. *Informatik-Spektrum*, 34(1) :79–83, 2011.

[Bur91] A. Burns. Scheduling hard real-time systems : a review. *Software Engineering Journal*, 6(3) :116–128, 1991.

[BW95] A. Burns and A.J. Wellings. *HRT-HOOD : a structured design method for hard real-time Ada systems*, volume 3. Elsevier Science, 1995.

[Car96] Francisco Vasques de Carvalho. *Sur l'integration de mecanismes d'ordonnancement et de communication dans la sous-couche mac de reseaux locaux temps-reel*. PhD thesis, 1996.

[CBGP12] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet. Design patterns for rule-based refinement of safety critical embedded systems models. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pages 67–76, 2012.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Ordonnancement temps réel*. Hermès, 2000.

[CDKM02] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in real-time systems*. Wiley Online Library, 2002.

[CFH+04] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. 2004.

[CG06] Liliana Cucu and Joël Goossens. Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors. In *Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on*, pages 397–404. IEEE, 2006.

[CGG04] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real-time systems of periodic tasks with offsets. *Theoretical computer science*, 310(1-3) :117–134, 2004.

[CHD+14] Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, et al. Simso : A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *Proceedings of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2014.

[CK88] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2) :141 –154, feb 1988.

[CRB+13] Fabien Cadoret, Thomas Robert, Etienne Borde, Laurent Pautet, and Frank Singhoff. Deterministic implementation of periodic-delayed communications and experimentation in aadl. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–8. IEEE, 2013.

[CSR+12] João Pedro Craveiro, Jeferson L. R. Souza, José Rufino, Vincent Gaudel, Laurent Lemarchand, Alain Plantec, Stéphane Rubini, and Frank Singhoff. Scheduling analysis principles and tool for time-and space-partitioned systems. In *INFORUM 2012 symposium*, pages 582–585, Lisbon, Portugal, 2012.

[DB05] R.I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp. –398, dec. 2005.

[DB11] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4) :35, 2011.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9) :569, 1965.

[DLG+11] Pierre Dissaux, J Legrand, Vincent Gaudel, Alain Plantec, Stéphane Rubini, Frank Singhoff, et al. Aadl real-time design-pattern automatic recognition. *SAE 2011*, 2011.

[DLP+10] P. Dissaux, J. Legrand, A. Plantec, M. Kerboeuf, and F. Singhoff. AADL Design-Patterns and Tools for Modelling and Performance Analysis of Real-Time systems. 2010.

[DMR+14] Pierre Dissaux, Olivier Marc, Stéphane Rubini, Christian Fotsing, Vincent Gaudel, Frank Singhoff, Alain Plantec, Vuong Nguyen-Hong, Hai-Nam Tran, et al. The smart project : Multi-agent scheduling simulation of real-time architectures. In *Proceedings of the ERTSS 2014 conference*, 2014.

[Dou02] Bruce Powell Douglass. *Real-Time Design Patterns : Robust Scalable Architecture for Real-Time Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[DPP+09] Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, simulate, and implement arinc653 systems using the aadl. In *ACM SIGAda Ada Letters*, volume 29, pages 31–44. ACM, 2009.

[DS08] P. Dissaux and F. Singhoff. Stood and cheddar : Aadl as a pivot language for analysing performances of real time architectures. jan 2008.

[EAL07] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 129 –138, dec. 2007.

[Ell] Ellidiss web site. http ://www.ellidiss.fr.

[ELSV09] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal. A compositional scheduling framework for digital avionics systems. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 371 –380, aug. 2009.

[Fei04] P.H. Feiler. Embedded system architecture analysis using sae aadl. Technical report, DTIC Document, 2004.

[FG06] Peter H Feiler and Aaron Greenhouse. Osate plug-in development guide. *CMU. Pittsburgh*, 2006.

[FSMA09] J. Fernández Sánchez and G. Mármol Acitores. Modelling and Evaluating Real-Time Software Architectures. *Reliable Software Technologies–Ada-Europe 2009*, pages 164–176, 2009.

[FT13] Singhoff F. Plantec A. Gaudel V. Rubini S. Li Shuai Fotsing, C. and Lemarchand L. Dissaux P. Legrand J. Tran, H. Cheddar Architecture Description Language. Lab-STICC Technical report, number fotsing-2014 Available at http ://beru.univ-brest.fr/svn/CHEDDAR/trunk/docs/cheddar_adl/cheddar_adl.pdf, 2013.

[G+96a] L. George et al. Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling. 1996.

[G+96b] L. George et al. Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling. 1996.

[Gau14] Dissaux P. Plantec A. Singhoff F. Hugues J. Legrand J. Gaudel, V. Subset Annex (First Draft). Available at http ://beru.univ-brest.fr/svn/CCHEDDAR/trunk/docs/Subset_Annex_Draft/SubsetAnnex-Draft.odt, 2014.

[GCG02] Emmanuel Grolleau and Annie Choquet-Geniet. Off-line computation of real-time schedules using petri nets. *Discrete Event Dynamic Systems*, 12(3) :311–333, 2002.

[GH98] J.C. Palencia Gutiérrez and M. González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. Proceedings of the 18th. IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998.

[GH10] O. Gilles and J. Hugues. Expressing and enforcing user-defined constraints of AADL models. In *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 337–342. IEEE, 2010.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns : Elements of reusable object-oriented design, 1995.

[GL99] Mark K Gardner and Jane WS Liu. Analyzing stochastic fixed-priority real-time systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 44–58. Springer, 1999.

[GLDN01] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 73–83. IEEE, 2001.

[GMW00] David Garlan, Robert T Monroe, and David Wile. Acme : Architectural description of component-based systems. *Foundations of component-based systems*, 68 :47–68, 2000.

[Gol13a] Boris Golden. *A Unified Formalism for Complex Systems Architecture*. PhD thesis, Ecole Polytechnique, 2013.

[Gol13b] Boris Golden. *A unified formalism for complex systems architecture*. PhD thesis, Palaiseau, Ecole polytechnique, 2013.

[GPS+13] Vincent Gaudel, Alain Plantec, Frank Singhoff, Jérôme Hugues, Pierre Dissaux, and Jérôme Legrand. Enforcing software engineering tools interoperability : An example with aadl subsets. In *Rapid System Prototyping (RSP), 2013 International Symposium on*, pages 59–65. IEEE, 2013.

[GSP+11a] Vincent Gaudel, Frank Singhoff, Alain Plantec, Pierre Dissaux, and Jérôme Legrand. Sélection automatique de tests de faisabilitéa l'aide de patrons de conception. *ETR'11*, pages 185–188, 2011.

[GSP+11b] Vincent Gaudel, Frank Singhoff, Alain Plantec, Stéphane Rubini, Pierre Dissaux, and Jérôme Legrand. An ada design pattern recognition tool for aadl performance analysis. In *ACM SIGAda Ada Letters*, volume 31, pages 61–68. ACM, 2011.

[GSP+14] Vincent Gaudel, Frank Singhoff, Alain Plantec, Pierre Dissaux, and Jérôme Legrand. Composition of design patterns : from the modeling of rtos synchronization tools to schedulability analysis. *ACM SIGBED Review*, 11(1) :44–49, 2014.

[GSYY09] Nan Guan, M. Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 387 –397, dec. 2009.

[H+06] Matthew Hause et al. The sysml modelling language. Citeseer, 2006.

[HB14] J. Hugues and G. Brau. Analysis as first-class citizens – an application to architecture description languages. June 2014.

[HGG+02] G. Harbour, G. Garcia, P. Gutierrez, D. Moyano, et al. MAST : Modeling and analysis suite for real time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125–134. IEEE, 2002.

[HGPDM01] Michael Gonzalez Harbour, Javier Gutierrez Garcia, J.C. Palencia, and J.M. Drake Moyano. MAST : modeling and analysis suite for real-time applications. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 125–134. IEEE Comput. Soc, 2001.

[HKL94] M.G. Harbour, M.H. Klein, and J.P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *Software Engineering, IEEE Transactions on*, 20(1) :13–28, Jan 1994.

[Hug11] J. Hugues. Analytic virtual integration of cyber-physical systems & AADL : challenges, threats and opportunities. In *Proceedings of the second Analytic Virtual Integration of Cyber-Physical Systems Workshop*, Vienna, Austria, November 2011.

[HZPK08] Jerome Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4) :42, 2008.

[JP86a] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5) :390, 1986.

[JP86b] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5) :390, 1986.

[JSM91] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129 –139, dec 1991.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, 2003.

[KBG12a] Mickael Kerboeuf, Jean-Philippe Babau, and V. Gaudel. A two-steps model transformation to extend the scope of an analysis framework to standard modeling languages. In *Conférence : 6th MoDELS workshop on Models and Evolution.*, pages P.P1–6, Innsbruck, Autriche, October 2012.

[KBG12b] Mickaël Kerboeuf, Jean-Philippe Babau, and Vincent Gaudel. A two-steps model transformation to extend the scope of an analysis framework to standard modeling languages. In *Proceedings of the 6th International Workshop on Models and Evolution*, pages 9–14. ACM, 2012.

[Ken02] Stuart Kent. Model driven engineering. In *Integrated formal methods*, pages 286–298. Springer, 2002.

[Ker05] Y Kermarrec. Approches et expérimentations autour des composants applications aux composants logiciels, aux objets d'apprentissages et aux services distribués. *Habilitation à diriger des recherches de l'Université de Bretagne Occidentale*, 2005.

[Kle76] L. Kleinrock. *Queueing Systems : Theory*, volume 1. Wiley, 1976.

[Kro09] Daniel Krob. Eléments d'architecture des systèmes complexes. *Gestion de la complexité et de l'information dans les grands systèmes critiques*, pages 179–207, 2009.

[LCH] B. Larson, P. Chalin, and J. Hatcliff. Bless : Formal specification and verification of behaviors for embedded systems with software. In N. Rungta G. Brat and A. Venet, editors, *NASA Formal Methods Symposium, SNFM 2013*, volume 7871 of *Lecture Notes in Computer Science*.

[LL73a] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environnment. *Journal of the Association for Computing Machinery*, 20(1) :46–61, January 1973.

[LL73b] CL Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[LLS07] Insup Lee, Joseph YT Leung, and Sang H Son. *Handbook of real-time and embedded systems*. CRC Press, 2007.

[LM80] J.Y.T. Leung and ML Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3) :115–118, 1980.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1) :134–152, 1997.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166 –171, dec 1989.

[LSNM04a] J. Legrand, F. Singhoff, L. Nana, and L. Marcé. Performance analysis of buffers shared by independent periodic tasks. Technical report, LISYC Technical report number legrand-02-2004, 2004.

[LSNM04b] Jérôme Legrand, Frank Singhoff, Laurent tchamnda Nana, and Lionel Marcé. Performance analysis of buffers shared by independent periodic tasks. In *LISYC Technical report number legrand-02-2004*, January 2004.

[LSRB14] Shuai Li, Frank Singhoff, Stéphane Rubini, and Michel Bourdelles. Extending schedulability tests of tree-shaped transactions for tdma radio protocols. pages 1–8, 2014.

[LW82] J.Y.T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks* 1. *Performance Evaluation*, 2(4) :237–250, 1982.

[Mar94] Patrick Martineau. *Ordonnancement en-ligne dans les systèmes informatiques temps réel.* PhD thesis, 1994.

[Med09] Nenad Medvidovic. A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering*, 26(1) :70–93, 2009.

[MFC01] A.K. Mok, X. Feng, and Deji Chen. Resource partition for real-time systems. In *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*, pages 75 –84, 2001.

[MKMG97] R.T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *Software, IEEE*, 14(1) :43 –52, jan/feb 1997.

[MMPT10] Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Damien Andrew Tamburri. Providing architectural languages and tools interoperability through model transformation technologies. *Software Engineering, IEEE Transactions on*, 36(1) :119–140, 2010.

[Moi85] Abha Moitra. Analysis of hard real-time systems. Technical report, Cornell University, 1985.

[MT00] Nenad Medvidovic and Richard N Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1) :70–93, 2000.

[Mue14] C. Mueller. Modélisation et analyse de systèmes distribués temps-réel. Master report, Available at http ://beru.univ-brest.fr/~singhoff/cheddar, September 2014.

[MW11] Dirk Müller and Matthias Werner. Genealogy of hard real-time preemptive scheduling algorithms for identical multiprocessors. *Central European Journal of Computer Science*, 1 :253–265, 2011.

[Nas07] Odile Nasr. *Spécification et vérification des ordonnanceurs Temps Réel en B.* PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2007.

[Obj05] Object Management Group. MARTE specification, 2005.

[Obj11] Object Management Group. UML specification, 2011.

[Obj13] Object Management Group. OMG website, 2013.

[OGRR12] Yassine Ouhammou, Emmanuel Grolleau, Pascal Richard, and Michael Richard. Reducing the gap between design and scheduling. In *Proceedings of*

the 20th International Conference on Real-Time and Network Systems, pages 21–30. ACM, 2012.

[PH03] J.C. Palencia and M. González Harbour. Offset-Based Response Time Analysis of Distributed Systems Scheduled under EDF. Proceedings of the Euromicro conference on real-time systems, Porto, Portugal, June 2003.

[PL04] R. Pellizzoni and G. Lipari. A new sufficient feasibility test for asynchronous real-time periodic task sets. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 204–211. IEEE, 2004.

[Pla] A Plantec. Faciliter la vérification et la validation de méta-modèles dans le cadre de l'ingénierie dirigée par les modèles : une approche agile, outillée et orientée données.

[Pla07] Platypus Technical Summary and download. http ://cassoulet.univ-brest.fr/mme/, 2007.

[PR06] Alain Plantec and Vincent Ribaud. PLATYPUS : A STEP-based Integration Framework. In *14th Interdisciplinary Information Management Talks (IDIMT-2006)*, pages 261–274, République Tchèque, September 2006.

[Pri08] Paul J Prisaznuk. Arinc 653 role in integrated modular avionics (ima). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1–E. IEEE, 2008.

[PS06] A. Plantec and F. Singhoff. Refactoring of an Ada 95 Library with a Meta CASE Tool. *ACM SIGAda Ada Letters, ACM Press, New York, USA*, 26(3) :61–70, November 2006.

[PS07] A. Plantec and F. Singhoff. Un processus d'ingenierie de Cheddar pour la simulation de systemes temps reel a grande echelle. *GENIE LOGICIEL-TOULOUSE THEN PARIS-*, 83 :26, 2007.

[PSDL10] A. Plantec, F. Singhoff, P. Dissaux, and J. Legrand. Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation-Volume Part I*, pages 4–17. Springer-Verlag, 2010.

[PSG+11] Alain Plantec, Frank Singhoff, V Gaudel, Vincent Ribaud, et al. Forward engineering and early model validation with smalltalk. In *5th Argentine Smalltalk Conference.*, 2011.

[PV07] M. Panunzio and T. Vardanega. A metamodel-driven process featuring advanced model-based timing analysis. *Reliable Software Technologies–Ada Europe 2007*, pages 128–141, 2007.

[RAP08] RMA RAPID. The art of modeling real-time systems. http ://www.tripac.com/html/prod-fact-rrm.html, 2008.

[RCM] Ismael Ripoll, Alfons Crespo, and Aloysius K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11 :19–39.

[REW04] J.B. Reinder, F.M. Elisabeth, and F.J. Wim. Best-case response time and jitter analysis of real-time tasks. *Journal of Scheduling*, 7 :133–144, 2004.

[RFS+14] Stéphane Rubini, Christian Fotsing, Frank Singhoff, Hai Nam Tran, and Pierre Dissaux. Scheduling analysis from architectural models of embedded multi-processor systems. *ACM SIGBED Review*, 11(1) :68–73, 2014.

[Riv98] Nicolas Rivierre. *Ordonnancement temps réel centralisé, les cas préemptifs et non-préemptifs.* PhD dissertation, University of Versailles - St Quentin, 1998.

[Rot94] H.G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *Computers and Digital Techniques, IEE Proceedings* -, 141(1) :1 –10, jan 1994.

[RRC02] P. Richard, M. Richard, and F. Cottet. *Analyse holistique des systèmes temps réel distribués : principes et algorithmes.* Ordonnancement dans les systèmes distribués, Hermès, 2002.

[RRC03] P. Richard, M. Richard, and F. Cottet. Analyse holistique des systèmes temps réel distribués : principes et algorithmes. *ordonnancement pour l'informatique parallèle, Traité IC2, Hermès, page*, 2003.

[SAa+] Lui Sha, Tarek Abdelzaher, Karl-Erik arzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory : A historical perspective. *Real-Time Systems*, 28 :101–155.

[SAE09] SAE. Architecture Analysis and Design Language (AADL) AS-5506A. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 2.0, January 2009.

[Sin] F. Singhoff. The cheddar aadl property sets (release 2.x). Technical report, LISyC Technical report number singhoff-03-2007, February 200.

[Sin08] F Singhoff. à propos de l'applicabilité de la théorie de l'ordonnancement temps-réel : le projet cheddar. *Habilitation à diriger des recherches de l'Université de Bretagne Occidentale*, 2008.

[Sin10] F. Singhoff. A taxonomy of real-time scheduling theory feasibility tests. LISyC Technical report, number singhoff-01-2010, Available at http ://beru.univ-brest.fr/~singhoff/cheddar, February 2010.

[SK93] J. Silcock and S. Kutti. Taxonomy of real-time scheduling. *Deakin University, Geelong, Australia*, 1993.

[SL03] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 2 – 13, dec. 2003.

[SLC06] Oleg Sokolsky, Insup Lee, and Duncan Clarke. Schedulability analysis of aadl models. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[SLNM05] F Singhoff, J Legrand, L Nana, and L Marcé. Scheduling and memory requirements analysis with aadl. In *ACM SIGAda Ada Letters*, volume 25, pages 1–10. ACM, 2005.

[SPDL09] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand. Investigating the usability of real-time scheduling theory with the Cheddar project. *Real-Time Systems*, 43(3) :259–295, 2009.

[SPR+14] Frank Singhoff, Alain Plantec, St'ephane Rubini, Vincent Gaudel, Shuai Li, Christian Fotsing, Pierre Dissaux, J'erôme Legrand, and Laurent Lemarchand. How architecture description languages help schedulability analysis : a return of experience from the cheddar project. *Science of Computer Programming*, 2014.

[Spu96a] M. Spuri. Analysis of deadline scheduled real-time systems. *Rapport De Recherche-Institut National De Recherche En Informatique Et En Automatique*, 1996.

[Spu96b] M. Spuri. Analysis of deadline scheduled real-time systems. *Rapport De Recherche-Institut National De Recherche En Informatique Et En Automatique*, 1996.

[SRL90a] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Transactions on computers*, pages 1175–1185, 1990.

[SRL90b] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9) :1175–1185, 1990.

[Sta88] JA Stancovic. Misconceptions about real-time computing. *IEEE Computer*, 21 :10–19, 1988.

[Sta92] John A Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 1992.

[TBW94] K. Tindell, A. Burns, and A. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9 :147–171, 1994.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2-3) :117–134, 1994.

[TFR05] Daniel E. Turk, Robert B. France, and Bernhard Rumpe. Assumptions underlying agile software-development processes. *J. Database Manag.*, 16(4) :62–87, 2005.

[Tim02] TimeSys. *Using TimeWiz to Understand System Timing before you Build or Buy.* White paper, http ://www.timesys.com/index.cfm ?bdy=home_bdy_library.cfm, 2002.

[Tin94] K. Tindell. Adding Time-Offsets to Schedulability Analysis. Technical report, YCS-94-221, University of York, 1994.

[TT92] Andrew S Tanenbaum and Andrew Tannenbaum. *Modern operating systems*, volume 2. Prentice hall Englewood Cliffs, 1992.

[UDT10] Richard Urunuela, A Deplanche, and Yvon Trinquet. Storm a simulation tool for real-time multiprocessor scheduling evaluation. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–8. IEEE, 2010.

[Ves93] Steve Vestal. A cursory overview and comparison of four architecture description languages. Technical report, Citeseer, 1993.

[Ves98] S. Vestal. Meta-H User's Manual, Version 1.27. Technical report, download at http ://www.htc.honeywell.com/metah/uguide.pdf, 1998.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3) :36, 2008.

[Weg96] Peter Wegner. Interoperability. *ACM Computing Surveys (CSUR)*, 28(1) :285–287, 1996.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[ZRS87] Wei Zhao, Krithi Ramamritham, and John A. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *Software Engineering, IEEE Transactions on*, (5) :564–577, 1987.