UFR Sciences et Techniques — Département d'Informatique

École doctorale SICMA — ED 373

À propos de l'applicabilité de la théorie de l'ordonnancement temps réel : le projet Cheddar

présentée et soutenue publiquement le 12 Novembre 2008

Habilitation à diriger des recherches

par

Frank Singhoff

Composition du jury

Président :	Jacques Tisseau	Professeur des Universités (ENIB, LISyC)
Rapporteurs :	Peter H. Feiler	Senior Technical Staff (Carnegie Mellon University, SEI)
	Fabrice Kordon	Professeur des Universités (Université Paris 6, LIP6)
	Laurent Pautet	Professeur (Télécom Paris-Tech, LTCI)
Examinateurs :	Mamoun Filali	Chargé de recherche CNRS (Université Toulouse 3, IRIT)
	Yvon Kermarrec	Professeur (Télécom Bretagne, LABSTICC)
	Philippe Le Parc	Professeur des Universités (UBO, LISyC)

Laboratoire d'Informatique des Systèmes Complexes — EA 3883

Mis en page avec la classe thloria.

« Pourquoi faire simple quand on peut faire compliqué. »

« Plus ça rate, plus on a de chances que ça marche. »

Mon guide spirituel,

Je tiens à remercier les membres du jury pour leur travail. Tout d'abord merci aux trois rapporteurs de ce mémoire, Laurent Pautet Professeur à Télécom-Paris-Tech, Fabrice Kordon Professeur des Universités à Paris 6 et Peter H. Feiler Senior Technical Staff à Carnegie Mellon University. Merci pour avoir accepté ce travail d'analyse. Vos suggestions et corrections ont permis d'améliorer significativement la qualité de ce mémoire. Je tiens particulièrement à remercier Peter H. Feiler pour avoir accepté de faire un si long voyage. Merci à Jacques Tisseau qui a bien voulu présider la soutenance. Merci à Mamoun Filali et à Yvon Kermarrec pour leur participation au jury. Un grand merci aussi à Philippe Le Parc pour son assistance lors de la complexe phase d'inscription mais aussi pour son aide à la rédaction de ce mémoire.

Le projet Cheddar est un travail collectif et différentes personnes ont travaillé sur ce projet depuis 2001. Je remercie donc Jérôme Legrand, Alain Plantec et Pierre Dissaux. Merci à Jérôme Legrand pour le travail concernant le dimensionnement mémoire qu'il a effectué pendant sa thèse. Merci à Alain Plantec pour sa contribution sur la mise en œuvre du langage de modélisation de Cheddar ainsi que ses outils (interpréteur, générateur, parser ...). Merci à Pierre Dissaux et à sa société Ellidiss Technologies pour leur travail commun sur AADL mais aussi et surtout pour leur confiance et le support qu'ils fournissent au projet. Merci à tous les trois. Les nombreux articles que nous avons rédigés et les idées élaborées ensemble sont directement relatés dans ce document. Ce mémoire est donc aussi le votre.

Les outils d'analyse AADL de Cheddar utilisent le logiciel Ocarina. Je tiens à remercier l'équipe d'Ocarina pour leur aide technique, leurs conseils et aussi pour leur confiance et leur amitié : merci à vous Jérôme Hugues, Laurent Pautet, Thomas Vergnaud et Bechir Zalila.

Merci à Laurent Nana et Lionel Marcé pour leur aide pendant l'encadrement de la thèse de Jérôme Legrand.

De la même façon, je tiens à remercier tous ceux qui ont contribué à ce mémoire. Merci à Stéphane Rubini, Jérôme Legrand, Mickaël Kerboeuf, Jean-Philippe Babau, Eric Gressier et Magali Singhoff pour la relecture des versions préliminaires. Merci aussi à Philippe Le Parc, Mickaël Kerboeuf et Jean-Philippe Babau pour leur aide lors de la préparation de la soutenance.

Je remercie aussi les membres du LISyC et du département/IUP informatique. Merci aux collègues qui tous les jours, par leur amitié, leur soutien et encouragements m'ont incité à poursuivre ces activités de recherche. Merci à Vincent Ribaud et à Mickaël Kerboeuf pour m'avoir soulagé des charges administratives pendant les mois de préparation de cette HDR.

Il ne serait pas possible de clore cette page de remerciements sans une pensée à mes «parents académiques» sans qui ce projet n'aurait pas vu le jour. Merci à Isabelle Demeure et Eric Gressier pour

vos efforts durant ma formation d'ingénieur, puis de jeune chercheur.

Enfin, Cheddar est un logiciel libre distribué sous licence GPL. En dehors des participants cités ci-dessus, de nombreuses personnes ont contribué à faire avancer ce logiciel. Je tiens donc à remercier toutes les personnes qui ont contribué ou qui contribuent toujours à ce logiciel : étudiants, enseignants/-chercheurs, et de façon plus générale, tous les utilisateurs de cet outil.

Mots clefs

Théorie de l'ordonnancement temps réel, Architecture temps réel, Vérification, Performances, AADL.

Résumé

Un système embarqué est un ensemble de matériels et de logiciels coopérant à l'accomplissement d'une mission spécifique. Il est souvent invisible à l'utilisateur et dispose de ressources limitées (processeur, mémoire, énergie, ...).

Un système embarqué peut être composé de différents sous-systèmes éventuellement répartis entre plusieurs équipements. Chaque sous-système offre alors un ensemble de fonctionnalités ou services conduisant à l'accomplissement de la mission pour lequel le système embarqué a été conçu.

Lorsque ces services sont contraints par le temps, le système embarqué est dit «temps réel». Le comportement correct du système dépend alors, non seulement des résultats logiques de chaque service, mais aussi du temps auquel les résultats sont produits.

Par ailleurs, on parle de système embarqué critique lorsque le système réalise une mission dont l'échec a potentiellement un impact majeur sur la vie, la santé de personnes, sur l'état de l'environnement, ... et plus généralement sur l'accomplissement des missions critiques auxquelles il participe.

Que ce soit à cause du coût de fabrication, de la criticité, de la complexité ou encore de l'inaccessibilité pendant l'exploitation de ces systèmes, les acteurs du domaine font généralement usage de diverses pratiques méthodologiques pour la réalisation d'un système embarqué temps réel critique. Traditionnellement, la mise en œuvre d'un système suit un cycle de développement en V. La partie descendante du V comprend les phases d'expression de besoins du système, de conception générale et détaillée puis de production des composants logiciels et matériels. La partie ascendante du V est constituée des phases de tests unitaires des composants, puis des tests d'intégration du système et d'une phase de recette. Par la suite, le système peut faire l'objet de maintenance évolutive ou corrective. Ces 30 dernières années, des avancées majeures ont vues le jour, à la fois sur les aspects méthodologiques mais aussi sur les modèles et outils permettant de vérifier la correction du système et de ses composants vis-à-vis de leurs exigences.

Classiquement, les vérifications sont réalisées tardivement dans la vie du système : en phases de test ou de maintenance corrective. Or, le coût de ces phases est généralement très élevé et dans un certain nombre de cas, les anomalies de fonctionnement peuvent être issues d'erreurs de conception ou d'expression des besoins.

Aujourd'hui, les acteurs du domaine souhaitent appliquer des méthodes de vérification au plus tôt dans le cycle de vie ; c'est à dire lors des phases d'expression de besoin ou de conception. Les méthodes de vérification applicables en phases de conception et/ou d'expression des besoins, qu'elles soient formelles ou non, ont pour l'instant des répercutions encore trop limitées dans les projets industriels. C'est notamment le cas de la théorie de l'ordonnancement temps réel.

Les fondements de la théorie de l'ordonnancement temps réel ont été proposés dès les années 1970. Elle a donné lieu à de très nombreuses recherches et expérimentations. Depuis 1990, les résultats théo-

riques permettent de prédire le comportement temporel de systèmes temps réel mono-processeur composés de tâches périodiques accédant à des ressources partagées grâce à des protocoles standards. Ainsi, un algorithme d'ordonnancement à priorités fixes tel que *Rate Monotonic* et un protocole d'accès aux ressources partagées tel que *Priority Ceiling Protocol* proposent des résultats théoriques nécessaires à une vérification efficace d'architectures temps réel.

De nombreux systèmes d'exploitation offrent un environnement d'exécution compatible avec les hypothèses de ces algorithmes et protocoles. Ainsi des systèmes d'exploitation tels que VxWorks en proposent une implantation. Certains standards proposent également une interface de système d'exploitation qui soit compatible avec la théorie de l'ordonnancement temps réel ; c'est le cas notamment du standard POSIX 1003.1b. Par ailleurs, de nombreux projets ont démontré que l'on pouvait utiliser avec succès la théorie de l'ordonnancement temps réel pour effectuer des vérifications en amont dans le cycle de développement d'un système temps réel.

L'approche proposée par la théorie de l'ordonnancement temps réel semble donc séduisante. Et pourtant, dans de très nombreux cas, elle reste globalement inappliquée par le monde industriel, même si souvent, le monde industriel montre un intérêt certain pour ce type de méthodes. Plusieurs hypothèses peuvent être avancées pour expliquer cet état de fait.

Il existe des architectures où cette théorie n'est actuellement pas applicable. C'est notamment le cas pour les systèmes répartis où peu de méthodes analytiques ont été proposées. Parfois, il n'existe tout simplement aucune méthode analytique pour des architectures complexes comme celles comportant certains ordonnanceurs hiérarchiques ou encore certaines architectures multi-processeurs. Nous pensons néanmoins que l'utilisation de cette théorie est pertinente pour de nombreuses architectures. Il est donc nécessaire d'identifier les familles d'architecture où l'application de cette théorie est possible.

Bien que cette théorie soit maintenant relativement ancienne, il existe à ce jour peu d'outils qui soient disponibles à la communauté. Or, si la théorie de l'ordonnancement temps réel semble simple à acquérir, pour qu'un ingénieur puisse effectivement l'employer, il est nécessaire de l'aider à en comprendre les forces et faiblesses. L'expertise nécessaire pour employer cette théorie est finalement longue à acquérir. En effet, depuis 30 ans, de très nombreux algorithmes et méthodes analytiques ont été proposés. Chaque méthode analytique nécessite de vérifier un ensemble d'hypothèses afin que la méthode puisse effectivement être employée. Chaque méthode a aussi un niveau de complexité et d'exactitude différent. Pour un ingénieur, il devient alors difficile de choisir la bonne méthode analytique à appliquer. Il y a donc un besoin d'outils, à la fois pour aider les ingénieurs à comprendre et maîtriser cette méthode de vérification, mais aussi pour automatiser la vérification.

Une raison supplémentaire limitant l'emploi de cette théorie est liée à son manque d'intégration dans les processus d'ingénierie. D'une part, peu d'ingénieurs savent à quels moments ils peuvent ou doivent effectuer ces vérifications pendant le cycle de développement d'un système embarqué. D'autre part, il existe actuellement un couplage insuffisant entre ces méthodes de vérification, les langages de conception et les outils de modélisation. Il n'est pas possible d'automatiser une vérification à partir d'un modèle de conception. Or, notre expérience nous amène à penser que dans la majorité des cas, l'usage d'un outil de vérification de l'ordonnancement reste difficile pour un ingénieur si l'outil est employé seul : l'ingénieur doit être assisté par les outils de modélisation.

Le projet Cheddar, initié en septembre 2000 au LISyC, a pour objectif d'accroître l'applicabilité de la théorie de l'ordonnancement temps réel. Pour ce faire, nous explorons plusieurs pistes.

D'abord, nous proposons des outils d'aide à la vérification. Ces outils permettent à l'ingénieur d'automatiquement appliquer une ou plusieurs méthodes analytiques compatibles avec l'architecture cible.

Dans un deuxième temps, nous avons étendu les résultats théoriques de la théorie des files d'attente afin que ces modèles puissent être appliqués aux architectures temps réel. Deux modèles de file d'attente (P/P/1 et M/P/1) ont été proposées et étudiées dans ce contexte. Ce travail a pour objectif la conception de méthodes analytiques permettant de dimensionner l'empreinte mémoire d'architecture temps réel. Puis, nous avons étudié en quoi l'usage d'un langage de conception d'architecture pouvait faciliter le travail de vérification d'une architecture temps réel. Nous avons choisi d'étudier cette problématique au travers du langage AADL. Nous avons évalué son adéquation avec la théorie de l'ordonnancement. Nous avons alors proposé des méthodes d'analyse basées sur les files d'attente P/P/1 et M/P/1 afin d'estimer l'empreinte mémoire d'architectures modélisées avec AADL. Actuellement, nous expérimentons comment ce langage peut être employé comme langage pivot entre différents outils de modélisation et d'analyse. Pour ce faire, nous étudions différents patrons de conception classiquement employés par les milieux industriels. Ces patrons de conception assurent l'interopérabilité entre des outils d'ingénierie logicielle. Cet axe de recherche contribue également à évaluer et à faire évoluer le standard AADL.

Enfin, nous explorons une dernière piste qui consiste à proposer un langage spécifique et des outils associés permettant de modéliser des ordonnanceurs et des tâches dont le comportement est différent des modèles classiques de la théorie de l'ordonnancement temps réel. En effet, de nombreuses applications industrielles ne peuvent pas être analysées par les méthodes analytiques : la théorie de l'ordonnancement temps réel ne fournit pas nécessairement de tests de faisabilité pour les ordonnanceurs et les modèles de tâches employés par le monde industriel. La définition d'un langage spécifique pour Cheddar allège le travail de modélisation tout en assurant une réutilisation potentielle des modèles. Nous proposons un processus d'utilisation associé à ce langage afin que l'utilisateur puisse aisément concevoir et tester son ordonnanceur. Grâce à l'utilisation d'un méta-environnement, l'utilisateur peut générer son outil de simulation, qui, après compilation, est capable de conduire des simulations sur des modèles de taille importante. Ce processus d'ingénierie dirigé par les modèles est mis en œuvre grâce à Platypus.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction et présentation synthétique des travaux

Sommaire

Un système embarqué est un ensemble de matériels et de logiciels coopérant à l'accomplissement d'une mission spécifique. Il est souvent invisible à l'utilisateur et dispose de ressources limitées (processeur, mémoire, énergie, ...).

Un système embarqué peut être composé de différents sous-systèmes éventuellement répartis entre plusieurs équipements. Chaque sous-système offre alors un ensemble de fonctionnalités ou services conduisant à l'accomplissement de la mission pour lequel le système embarqué a été conçu.

Lorsque ces services sont contraints par le temps, le système embarqué est dit temps réel. Le comportement correct du système dépend alors, non seulement des résultats logiques de chaque service, mais aussi du temps auquel les résultats sont produits [Sta88].

Par ailleurs, on parle de système embarqué critique lorsque le système réalise une mission dont l'échec a potentiellement un impact majeur sur la vie, la santé de personnes, sur l'état de l'environnement, ... et plus généralement sur l'accomplissement des missions critiques auxquelles il participe.

Que ce soit à cause du coût de fabrication, de la criticité, de la complexité ou encore de l'inaccessibilité pendant l'exploitation de ces systèmes, les acteurs du domaine font généralement usage de diverses pratiques méthodologiques pour la réalisation d'un système embarqué temps réel critique.

Par exemple, la mise en œuvre d'un système peut suivre un cycle de développement en V [GMSB96]. La partie descendante du V comprend les phases d'expression de besoins du système, de conception générale et détaillée puis de production des logiciels et des équipements matériels.

La phase d'expression des besoins consiste à spécifier ce que le système doit offrir comme services. On spécifie les contraintes fonctionnelles et non fonctionnelles que le système doit assurer. Ces contraintes non fonctionnelles peuvent être des exigences en terme de disponibilité/fiabilité ou de performance.

Dans les phases de conception générale et détaillée, on définit comment le système doit être réalisé. Une architecture du système est alors proposée.

Dans la suite de ce document, nous désignerons l'ingénieur qui a en charge les phases de conception, de façon indifférente, par les termes d'architecte ou de concepteur.

L'architecture décrit les différents composants de la solution choisie pour implanter le système. Un composant peut être défini comme une unité destinée à être assemblée et à fonctionner avec d'autres [Ker05]. Ici, le terme d'architecture fait référence à l'architecture matérielle et logicielle. Un composant peut donc être un processeur, un bus, un processus ou une tâche, ... L'utilisation de composants est justifiée par les besoins de réutilisabilité. Ainsi, un composant comporte habituellement deux parties : un ensemble d'interfaces permettant de décrire les interactions possibles avec d'autres composants et son implantation exprimée avec un langage de programmation ou de description matériel. Une architecture décrit également les interactions entre les composants. Les composants peuvent être organisés hiérarchiquement.

La phase de réalisation consiste alors à implanter individuellement les différents composants.

La partie ascendante du V est constituée des phases de tests unitaires des composants, puis des tests d'intégration du système et d'une phase de recette. Les phases de tests unitaires ont pour but de vérifier que chaque composant offre individuellement les services attendus. Puis, la phase de tests d'intégration complète les tests unitaires en s'assurant que les composants interagissent correctement entre eux. Enfin, la phase de recette est celle où le client réceptionne le système et vérifie qu'il répond

bien à ses besoins. Par la suite, le système peut faire l'objet de maintenance corrective qui consiste à corriger ses composants lorsqu'une anomalie de fonctionnement est détectée.

Ces 30 dernières années, des avancées majeures ont vues le jour, à la fois sur les aspects méthodologiques mais aussi sur les modèles et outils permettant de vérifier la correction du système et de ses composants vis-à-vis de leurs exigences.

Classiquement, les vérifications sont réalisées tardivement dans la vie du système : en phases de test ou de maintenance corrective. Or, le coût de ces phases est généralement très élevé [Boe81] et dans un certain nombre de cas, les anomalies de fonctionnement peuvent être issues d'erreurs de conception ou d'expression des besoins.

Aujourd'hui, les acteurs du domaine souhaitent appliquer des méthodes de vérification au plus tôt dans le cycle de vie ; c'est à dire lors des phases d'expression de besoin ou de conception. Toutefois, les méthodes de vérification applicables en phases de conception et/ou d'expression des besoins, qu'elles soient formelles ou non, ont pour l'instant des répercutions encore trop limitées dans les projets industriels. En effet, pour diverses raisons, de très nombreux industriels n'emploient pas ces méthodes, alors que paradoxalement, les systèmes critiques nécessitent un nombre de vérifications plus élevé que les systèmes d'information traditionnels.

Dans ce mémoire, nous nous focalisons sur l'analyse de performances des systèmes embarqués temps réel à l'aide de la théorie de l'ordonnancement temps réel. Cette théorie peut être employée, par exemple, pour valider une architecture proposée lors de la phase de conception. La vérification des exigences de l'architecture consiste, entre autre, à vérifier les contraintes temporelles des tâches, et en particulier, à vérifier que le temps d'exécution de chaque tâche ne dépasse pas une échéance donnée.

1.1 La théorie de l'ordonnancement temps réel : des méthodes de vérification inapplicables ?

Il existe principalement trois méthodes de vérification des performances que l'on puisse appliquer aux systèmes temps réel : la simulation, les méthodes analytiques et le *model-checking* [Nas07].

La simulation consiste à exécuter une spécification du système et à vérifier ses performances dans les conditions les plus significatives du système. En général, il est impossible de faire des simulations qui énumèrent exhaustivement tous les états possibles du système à vérifier. La simulation consiste alors à vérifier les cas les plus représentatifs. Cette méthode de vérification ne conduit donc pas à une preuve que le système est conforme à ses exigences. Toutefois, la simulation permet de vérifier des systèmes de grandes tailles qui utilisent des méthodes d'allocation de ressources spécifiques.

L'approche analytique consiste à analyser un système grâce à un modèle qui est associé à un ensemble d'équations. Les équations permettent de calculer différents critères tels que le temps de réponse d'une tâche ou le taux d'occupation d'une ressource. Ces équations décrivent indirectement le comportement du système. Pour appliquer une méthode analytique sur un système donné, il est nécessaire que celui-ci soit conforme aux hypothèses employées pour élaborer ces équations. Les méthodes analytiques permettent généralement de vérifier des systèmes de grandes tailles. Malheureusement, il existe des

systèmes complexes sur lesquels aucune équation ne peut être appliquée.

Enfin, le *model-checking* est une approche où le système est décrit grâce à un modèle exprimé avec un langage formel permettant à des outils d'énumérer exhaustivement l'ensemble des états potentiels du modèle. La vérification de la correction du modèle vis-à-vis des propriétés attendues est effectuée par un parcours de cet ensemble d'états. On parle alors de vérification formelle. Dans le contexte de l'ordonnancement temps réel, la difficulté pour ce dernier type d'outil, est de s'assurer que le modèle est une abstraction correcte du système que l'on cherche à analyser. D'autre part, le passage à l'échelle est parfois difficile. Les modèles généralement appliqués avec cette méthode pour vérifier le respect des échéances d'un ensemble de tâches peuvent être les Réseaux de Petri [GCG00], les automates temporisés [FMPY06], l'algèbre de processus [SLC06] ...

La théorie de l'ordonnancement temps réel propose des méthodes analytiques (appelées tests de faisabilité) et des algorithmes qui peuvent être employés lors de simulations. Un test de faisabilité permet d'effectuer une vérification de l'applicabilité d'un ordonnanceur temps réel pour une architecture donnée. Elle permet donc au concepteur de vérifier *a priori* le comportement temporel de l'architecture de son système grâce à deux des approches décrites ci-dessus.

Dans le cadre de cette théorie, une architecture logicielle peut être modélisée selon différents modèles de tâches et d'ordonnanceurs. Un modèle de tâches décrit comment une tâche peut être réveillée, comment son temps d'exécution peut être représenté et quand ou comment elle peut accéder aux autres ressources de l'architecture. Un ordonnanceur décrit les règles de partage du processeur entre les différentes tâches. Pour un modèle de tâches et un ordonnanceur donné, la théorie propose parfois un ou plusieurs tests de faisabilité permettant de vérifier les contraintes temporelles

Les algorithmes d'ordonnancement permettent aussi de simuler le comportement d'un tel système. De façon générale, une simulation ne permet pas d'exhiber une preuve. Néanmoins, dans le cas d'algorithmes d'ordonnancement déterministes et de jeux de tâches dont les dates de réveil sont aussi déterministes (c'est le cas avec le modèle de tâches périodiques de Liu et Layland [LL73]), alors la simulation peut mener à une preuve du respect des contraintes temporelles par le système, à condition que l'ordonnancement soit calculé sur la période d'étude [1][LM80]. En effet, il est possible dans ce cas d'énumérer exhaustivement les différents états possibles du système. Dans la suite de ce document, nous parlerons de simulations exhaustives pour faire référence à ce type d'analyse.

1.2 Les atouts de cette théorie ...

Les fondements de la théorie de l'ordonnancement temps réel ont été proposés dès les années 1970 [LL73]. Elle a donné lieu à de très nombreuses recherches et expérimentations [CDKM02, KRP+94, GRS96].

Depuis 1990, les résultats théoriques permettent de prédire le comportement temporel de systèmes temps réel mono-processeur composés de tâches périodiques accédant à des ressource partagées grâce à des protocoles standards [SRL90]. Les algorithmes d'ordonnancement à priorités fixes tels que *Rate Mo-*

[1] La période d'étude est une séquence d'ordonnancement qui se répète indéfiniment.

notonic et les protocoles d'accès aux ressources partagées tels que *Priority Ceiling Protocol* proposent les résultats théoriques nécessaires à une vérification efficace d'architectures temps réel.

De nombreux systèmes d'exploitation offrent un environnement d'exécution compatible avec les hypothèses de ces algorithmes et protocoles. Ainsi des systèmes d'exploitation tels que VxWorks en proposent une implantation [Riv97]. Certains standards proposent également une interface de système d'exploitation qui soit compatible avec la théorie de l'ordonnancement temps réel ; c'est le cas notamment du standard POSIX 1003.1b [Gal95].

Enfin, il existe des compilateurs qui implantent des méthodes permettant de vérifier statiquement qu'un système temps réel respecte effectivement les hypothèses nécessaires à l'application de la théorie de l'ordonnancement temps réel. Le profil Ravenscar, défini dans le standard international Ada 2005, autorise ce type d'analyse statique [ISO07, TDB+06].

Ainsi, de nombreux projets ont démontré que l'on pouvait utiliser avec succès la théorie de l'ordonnancement temps réel pour effectuer des vérifications en amont dans le cycle de développement d'un système temps réel [SEI03].

1.3 ... et les faiblesses de cette théorie

L'approche proposée par la théorie de l'ordonnancement temps réel semble séduisante. Pourtant, dans de très nombreux cas, elle reste globalement inappliquée par le monde industriel, même si souvent, le monde industriel montre un intérêt certain pour ce type de méthodes. Plusieurs hypothèses peuvent être avancées pour expliquer cet état de fait.

Certaines des architectures actuelles sont trop complexes

Il existe des architectures où cette théorie n'est actuellement pas applicable. C'est notamment le cas pour les systèmes répartis où peu de méthodes analytiques ont été proposées [TC94]. Les systèmes répartis requièrent la gestion simultanée de multiples ressources interdépendantes (réseaux, mémoire, processeurs), compliquant fortement l'analyse du système. De plus, on s'attend à un déterminisme temporel plus faible avec les architectures temps réel réparties, ce qui lève la question de l'applicabilité des quelques résultats analytiques disponibles dans ce contexte [CDKM02].

Parfois, il n'existe tout simplement aucune méthode analytique pour des architectures complexes comme celles comportant certains ordonnanceurs hiérarchiques ou encore certaines architectures multi-processeurs.

Notons que même dans le cas mono-processeur, certaines hypothèses associées aux modèles proposés dans le cadre de cette théorie peuvent conduire à des erreurs d'analyse du système. Ainsi, le temps d'exécution d'une tâche est modélisé par un délai fixe. Ce délai fixe, appelé «capacité» ou WCET[2], modélise le pire temps d'exécution d'une tâche. En pratique, le temps d'exécution d'une tâche n'est bien sûr pas constant. Cette hypothèse simplificatrice peut conduire à des incohérences graves entre les résultats d'analyse issues du modèle et le fonctionnement réel du système à vérifier [RPGC02].

[2]Worst Case Execution Time.

D'autre part, l'estimation de la capacité est un problème difficile, même si des outils existent [WEE+08, CPRS03].

Nous pensons néanmoins que l'utilisation de cette théorie est pertinente pour de nombreuses architectures. Il est donc nécessaire d'identifier les familles d'architecture où l'application de cette théorie est possible.

Peu d'outils disponibles pour la formation des ingénieurs et pour l'automatisation des vérifications

Bien que cette théorie soit maintenant relativement ancienne, il existe à ce jour peu d'outils qui soient disponibles à la communauté. Des outils comme Rapid-RMA [TP03], Timewiz [Tim02] ou MAST [HGGM01] proposent néanmoins des implantations de cette théorie.

Rapid-RMA et Timewiz se focalisent sur certains environnements d'exécution et implante une partie restreinte de la théorie de l'ordonnancement temps réel. Par ailleurs, ces outils ne sont pas librement accessibles. MAST constitue un des seuls outils open-source qui intègre une partie significative de la théorie. A ce jour, il implante différents algorithmes et tests de faisabilité, pour des algorithmes d'ordonnancement statiques et dynamiques. Depuis peu, les auteurs cherchent à employer MAST pour l'analyse de modèles UML. L'approche explorée par le projet MAST est donc proche de celle décrite dans ce mémoire.

Il existe donc un nombre très restreint d'outils accessible à la communauté. Or, si la théorie de l'ordonnancement temps réel semble simple à acquérir, pour qu'un étudiant puisse effectivement l'employer, il est nécessaire de l'aider à en comprendre les forces et faiblesses.

L'expertise nécessaire pour employer ces méthodes est finalement longue à acquérir. En effet, depuis 30 ans, de très nombreux algorithmes et méthodes analytiques ont été proposés. Chaque méthode analytique nécessite de vérifier un ensemble d'hypothèses afin que la méthode puisse effectivement être employée. Chaque méthode a aussi un niveau de complexité et d'exactitude différent. Pour un concepteur, il devient alors difficile de choisir la bonne méthode analytique à appliquer.

Il y a donc un besoin d'outils, à la fois pour aider les ingénieurs à comprendre et maîtriser cette méthode de vérification, mais aussi pour automatiser la vérification. Notons qu'un bon outil de vérification, dans ce contexte, doit implanter les principaux résultats analytiques mais aussi offrir un moyen d'adapter l'outil aux contextes de vérification. C'est notamment le cas quand le système à vérifier utilise des ordonnanceurs spécifiques.

Intégration dans le processus de développement

Une raison supplémentaire limitant l'emploi de cette théorie est liée à son manque d'intégration dans les processus d'ingénierie. D'une part, peu d'ingénieurs savent à quels moments ils peuvent ou doivent effectuer ces vérifications pendant le cycle de développement d'un système embarqué. D'autre part, il existe actuellement un couplage insuffisant entre ces méthodes de vérification, les langages de conception et les outils de modélisation. Il n'est pas possible d'automatiser une vérification à partir d'un modèle de conception. Or, notre expérience nous amène à penser que dans la majorité des cas,

l'usage d'un outil de vérification de l'ordonnancement reste difficile pour un ingénieur si l'outil est employé seul : le concepteur doit être assisté par les outils de modélisation et par l'usage de patrons de conception. En effet, à chaque patron de conception, il est possible d'associer les méthodes analytiques applicables. La modélisation d'une architecture temps réel en appliquant des patrons de conception est un moyen qui garantit au concepteur que son architecture peut être vérifiée. La définition et l'usage de patrons de conception intégrant les données nécessaires à la vérification de critères de performance pour les architectures temps réel sont des pistes actuellement étudiées par de nombreuses équipes telles que [PV07].

Il est donc nécessaire de réfléchir sur la façon d'associer la théorie de l'ordonnancement temps réel aux processus de développement et aux langages de conception employés par les industriels.

1.4 Le projet Cheddar : parcours des activités de recherche et objectifs

Dans le cadre de ma thèse [Sin99], j'ai participé au projet Polka de septembre 1999 à juin 2000. Avec Isabelle Demeure (Professeur à Télécom Paris-Tech et directeur de thèse), nous avions proposé un environnement offrant un support d'exécution pour des applications multimédias réparties manipulant des flots de données continues tels que l'audio et la vidéo. Pour ces systèmes temps réel, nous avions proposé l'utilisation d'un modèle orienté composants dans lequel nous spécifiions les flots de données multimédias ainsi que les composants importants du système (composants matériels et logiciels tels que cartes graphiques, cartes audio, décodeurs logiciels MPEG, disques, coupleurs réseaux, ...). Ce modèle permettait la spécification des contraintes temporelles associées aux flots de données. Nous avions alors proposé des algorithmes qui exploitaient ce modèle et fournissaient principalement des services pour ordonnancer les tâches du système conformément aux contraintes temporelles spécifiées. Le modèle ainsi que les algorithmes d'ordonnancement furent validés par une plate-forme et par le développement de plusieurs applications multimédias. Nous montrions alors les impacts de ces propositions sur l'ordonnancement des tâches et sur l'empreinte mémoire du système.

Cette thèse fut l'occasion d'aborder la problématique de l'utilisation coordonnée de différentes ressources d'un système temps réel (mémoire, processeur, réseaux, carte audio, ...) afin de garantir des contraintes non fonctionnelles (ex : délais de bout en bout, gigue, ...). Nous avions alors constaté que peu de résultats théoriques existaient concernant l'ordonnancement temps réel et son impact sur la gestion mémoire. Cette expérience fut aussi l'occasion d'appréhender les forces et les faiblesses d'une modélisation par composants matériels et logiciels d'un système temps réel en vue d'une analyse des performances.

D'autre part, ce travail mit en évidence la difficulté de concevoir et de tester des algorithmes d'ordonnancement, puis, d'en analyser les impacts sur la gestion des ressources processeurs et mémoire à partir de modèles d'architecture orientés composants.

Le projet Cheddar a donc été initié lors de mon arrivée à l'UBO en septembre 2000. Ce projet a pour objectif d'accroître l'applicabilité de la théorie de l'ordonnancement temps réel. Le cadre de

ce projet est donc les applications concurrentes asynchrones : nous excluons les approches synchrones [HCRP91a, HCRP91b, BG92, Har87] du périmètre de ce projet Cheddar.

Pour ce faire, nous explorons plusieurs pistes.

D'abord, j'ai proposé des outils d'aide à la vérification. Ces outils permettent au concepteur d'automatiquement appliquer une ou plusieurs méthodes analytiques compatibles avec l'architecture cible.

Dans un deuxième temps, pendant la thèse de Jérôme Legrand (UBO/LISyC), nous avons étendu les résultats théoriques de la théorie des files d'attente afin que ces modèles puissent être appliqués aux architectures temps réel. Deux files d'attente (P/P/1 et M/P/1) ont été proposées et étudiées dans ce contexte. Ce travail avait pour objectif la conception de méthodes analytiques permettant de dimensionner l'empreinte mémoire d'architecture temps réel. Puis, avec Pierre Dissaux (Ellidiss Technologies), nous avons étudié en quoi l'usage d'un langage de conception d'architecture pouvait faciliter le travail de vérification d'une architecture temps réel. Nous avons choisi d'étudier cette problématique au travers du langage AADL. AADL est une norme internationale publiée par la SAE[3] sous le standard AS-5506 [SAE04, SFR+07]. AADL offre la possibilité de décrire l'architecture complète matérielle et logicielle d'un système embarqué. Ce langage autorise donc l'analyse de performances sur plusieurs ressources différentes (processeurs, empreinte mémoire, ...). Dans un premier temps, nous avons évalué son adéquation avec la théorie de l'ordonnancement. Puis, nous avons proposé des méthodes d'analyse basées sur les files d'attente P/P/1 et M/P/1 afin d'estimer l'empreinte mémoire d'architectures modélisées avec AADL [SLNM05].

Actuellement, nous expérimentons comment ce langage peut être employé comme langage pivot entre différents outils de modélisation et d'analyse. Pour ce faire, nous étudions différents patrons de conception classiquement employés par les milieux industriels. Ces patrons de conception assurent l'interopérabilité entre des outils d'ingénierie logicielle. Cet axe de recherche contribue également à évaluer et à faire évoluer le standard AADL.

Enfin, avec Alain Plantec (UBO/LISyC), nous explorons une piste qui consiste à proposer un langage spécifique et des outils associés permettant de modéliser des ordonnanceurs et des tâches dont le comportement est différent des modèles classiques de la théorie de l'ordonnancement temps réel. En effet, de nombreuses applications industrielles ne peuvent pas être analysées par les méthodes analytiques : la théorie de l'ordonnancement temps réel ne fournit pas nécessairement de tests de faisabilité pour les ordonnanceurs et les modèles de tâches employés par le monde industriel. D'autre part, élaborer ces tests de faisabilité est une activité coûteuse et généralement difficile. Les systèmes à analyser dans ce contexte sont généralement de grande taille, de sorte qu'il est pas toujours possible d'employer des méthodes de *model-checking* [BBL99]. Le monde industriel est alors réduit à vérifier ses architectures par la simulation.

Dans le cas où seule la simulation est possible, il est toutefois envisageable d'employer les fonctionnalités de simulation de certains environnements de *model-checking*. C'est notamment le cas des Réseaux de Petri [GV03] et d'outils tels que CPN Tools [Wel06], Tina [BV06] ou CPN-AMI [HHK+06]. Ces outils impliquent néanmoins la modélisation d'abstractions élémentaires du domaine telles que la

[3]Society for Automotive Enginners.

préemption, les règles de tri, le calcul de priorités. Dans sa thèse, Loïc Plassart (UBO/LISyC) illustre bien cette contrainte [Pla08].

Une autre solution consiste à développer des outils de simulation ad-hoc. Cette solution est coûteuse et n'offre, en général, qu'un faible niveau de réutilisation du logiciel de simulation. La définition d'un langage spécifique pour Cheddar allège le travail de modélisation pour le concepteur tout en lui assurant une réutilisation potentielle de ses modèles. Nous proposons un processus d'utilisation associé à ce langage afin que l'utilisateur puisse aisément concevoir et tester son ordonnanceur. Grâce à l'utilisation d'un méta-environnement, l'utilisateur peut générer son outil de simulation, qui, après compilation, est capable de conduire des simulations sur des modèles de taille importante. Ce processus d'ingénierie dirigé par les modèles est mis en œuvre grâce à Platypus [PR98].

Pour conclure, le projet Cheddar étudie donc deux approches complémentaires de vérification d'une architecture temps réel.

La première approche suppose que l'architecture à analyser est un assemblage de composants conformes à des patrons de conception. On suppose que ces patrons de conception garantissent le respect des hypothèses associées aux méthodes analytiques de vérification des performances de l'architecture. Les enjeux scientifiques et techniques soulevés pas cette approche sont nombreux. Quelles sont les méthodes analytiques efficaces et d'une précision suffisante ? Quels sont les patrons de conception offrant un niveau d'abstraction adéquate ? Comment composer différents patrons tout en conservant la prédictibilité des performances de l'architecture ? Comment lever les nombreux verrous technologiques permettant aux différents outils d'interopérer ? Etc.

La seconde approche propose une solution lorsque l'architecture à vérifier comporte des composants qui ne peuvent être modélisés avec les patrons de conception usuels. C'est notamment le cas lorsque l'architecture comporte des ordonnanceurs et modèles de tâches spécifiques. L'architecture est modélisée par le langage de Cheddar et l'analyse est effectuée par simulation. Dans certains cas, ces simulations énumèrent exhaustivement tous les états atteignables de l'architecture : il est donc parfois possible d'exhiber une forme de preuve pour la propriété que l'on a vérifiée sur les performances de l'architecture.

Là aussi, les enjeux scientifiques et techniques soulevés sont nombreux et complexes. Quel est le langage offrant le niveau d'abstraction adéquate ? Comment vérifier qu'un modèle spécifie un ordonnanceur déterministe ? Dans ces environnements déterministes, comment peut on s'assurer que la simulation conduira à une preuve ? Comment construire automatiquement des simulateurs pouvant passer à l'échelle ? Etc.

1.5 Plan du document

Dans la suite de ce document, nous détaillons plus précisément les pistes décrites ci-dessus.

Le chapitre 2 propose une introduction à la théorie de l'ordonnancement temps réel. Dans ce chapitre, nous présentons les résultats théoriques sur lesquels nous avons focalisé notre attention depuis le lancement du projet Cheddar. Puis, nous présentons un environnement permettant de modéliser un système et d'y appliquer ces résultats. Des exemples d'utilisation de cet environnement sont esquissés.

Le chapitre 3 traite des travaux que nous avons conduits autour d'AADL. Nous y décrivons comment appliquer la théorie de l'ordonnancement temps réel sur des modèles AADL. Nous présentons aussi des méthodes qui permettent l'analyse de l'empreinte mémoire d'une architecture modélisée par AADL. Enfin, nous définissons un ensemble de patrons de conception. Ces patrons de conception offrent la possibilité d'utiliser AADL comme un langage pivot entre différents outils d'ingénierie logicielle.

Le chapitre 4 est dédié aux travaux relatifs au langage spécifique et à son environnement pour la modélisation et la simulation d'algorithmes d'ordonnancement temps réel. Le langage spécifique est d'abord présenté, puis, illustré avec un modèle de l'ordonnancement hiérarchique du standard ARINC 653 [Ari97]. Par la suite, nous décrivons un processus d'utilisation des modèles d'ordonnanceur.

Enfin, dans le chapitre 5, nous concluons et dressons quelques perspectives à court et moyen termes de nos travaux.

Ce mémoire est complété par plusieurs annexes. L'annexe A décrit le référentiel de tests de faisabilité employé par nos outils lorsque les modèles d'architectures sont composés de tâches périodiques. L'annexe B contient un résumé long en anglais du projet. Enfin, l'annexe C présente un Curriculum Vitae qui résume ma carrière.

Chapitre 2

Accroître l'applicabilité de la théorie de l'ordonnancement temps réel : automatiser par l'outil

Sommaire

Une abondante littérature existe concernant la théorie de l'ordonnancement temps réel. Dans la première partie de ce chapitre, nous nous limitons à une présentation des principaux concepts et résultats de cette théorie. Ces savoirs élémentaires constituent généralement ceux présentés aux étudiants de Master dans le cadre d'un enseignement sur les systèmes temps réel. L'objectif de cette première partie n'est donc pas de présenter un état de l'art le plus exhaustif qui soit, mais plutôt d'introduire les concepts nécessaires à la compréhension des chapitres suivants. Le lecteur pourra se référer à [GRS96, KRP+94, Leb98, CDKM02] pour une présentation plus complète de la théorie de l'ordonnancement ainsi qu'à [Gal95, BW07] pour la présentation de différents environnements d'exécution compatibles avec cette théorie.

Dans la seconde partie du chapitre, nous introduisons la plate-forme de vérification Cheddar. Nous présentons sa structure, son fonctionnement ainsi que les tests de faisabilité que nous avons sélectionnés pour être implantés dans cette plate-forme. Puis, nous décrivons quelques exemples types d'utilisation de ces outils.

2.1 Introduction à la théorie de l'ordonnancement temps réel

La théorie de l'ordonnancement temps réel propose des algorithmes d'ordonnancement et des méthodes algébriques (appelées tests de faisabilité) qui permettent au concepteur de vérifier *a priori* le comportement temporel de son architecture. Pour ce faire, le concepteur doit modéliser les traitements de son système par le biais de différents modèles temporels tels que le modèle des tâches périodiques, ou le modèle des tâches sporadiques. Dans cette partie, nous présentons les modèles de tâches les plus élémentaires. Puis, nous décrivons succinctement deux algorithmes d'ordonnancement classiques adaptés à ces modèles de tâches. Chaque algorithme d'ordonnancement est expliqué avec un exemple de test de faisabilité qui peut être employé pour vérifier *a priori* les performances des tâches d'une architecture.

2.1.1 Modèles de tâches élémentaires

Une tâche peut être définie comme un ensemble de données (tel qu'une pile), une suite séquentielle d'instructions et un contexte d'exécution. Le contexte d'exécution mémorise le contexte de la tâche vis-à-vis du processeur (valeur des registres) et/ou de la mémoire (registre de la MMU[4]). Le contexte mémorise aussi l'état de la tâche. Généralement, une tâche peut être prête, bloquée ou en cours d'exécution.

Une tâche en cours d'exécution détient toutes les ressources nécessaires à l'exécution de son code. Elle exécute son code séquentiellement jusqu'à sa terminaison ou jusqu'à l'instant où l'ordonnanceur décide d'exécuter une autre tâche. On parle de préemption dans ce second cas. La préemption implique donc une commutation de contexte.

Une tâche bloquée est une tâche qui ne peut être exécutée. En plus du processeur, une tâche bloquée requiert la disponibilité d'autres ressources, ce qui conduit l'ordonnanceur à ne pas considérer la tâche

[4]Memory Management Unit.

pendant le calcul de l'ordonnancement. La demande d'une opération d'entrée/sortie ou une tentative d'accès à un sémaphore verrouillé peut conduire une tâche en cours d'exécution à être bloquée.

Enfin, une tâche prête possède toutes les ressources nécessaires à son exécution, hormis le processeur.

Une tâche peut être définie par plusieurs attributs. Une tâche peut être répétitive ou non, indépendante ou dépendante, urgente ou importante.

Une tâche importante est une tâche pour laquelle tout doit être mis en œuvre pour le respect de ses contraintes temporelles. L'échec des contraintes temporelles d'une tâche importante implique de graves conséquences sur le fonctionnement du système auquel elle appartient, ce qui peut conduire à des pertes humaines, la destruction de l'équipement, ...

Une tâche urgente est une tâche dont l'échéance est proche. L'échéance représente le délai au-delà duquel la tâche doit terminer son traitement. Notez qu'une tâche urgente n'est pas nécessairement importante, et réciproquement.

Une tâche indépendante est une tâche dont la seule condition d'exécution est la disponibilité du processeur : elle peut être exécutée par le processeur indépendamment des autres tâches. Une tâche dépendante est contrainte par l'exécution d'autres tâches, soit par les ressources qu'elles partagent (exemple : zone mémoire partagée), soit par des contraintes de précédence.

Enfin, une tâche peut être répétitive ou non. Une tâche répétitive est une tâche exécutée plusieurs fois. Une tâche répétitive fait l'objet d'activations successives. Une tâche répétitive peut être périodique si un délai fixe existe entre chaque activation. Si seul un délai minimal est connu entre chaque activation, on parle de tâche sporadique. Une tâche périodique modélise souvent un traitement critique qui peut être soit important, soit urgent. La modélisation d'un traitement par une tâche périodique permet d'évaluer précisément le besoin en processeur de la tâche. Finalement, une tâche apériodique est une tâche qui est activée de façon événementielle, est exécutée, puis disparaît du système. Il est plus difficile de prédire le respect des échéances d'une telle tâche. Une tâche apériodique ne devrait donc pas modéliser un traitement critique.

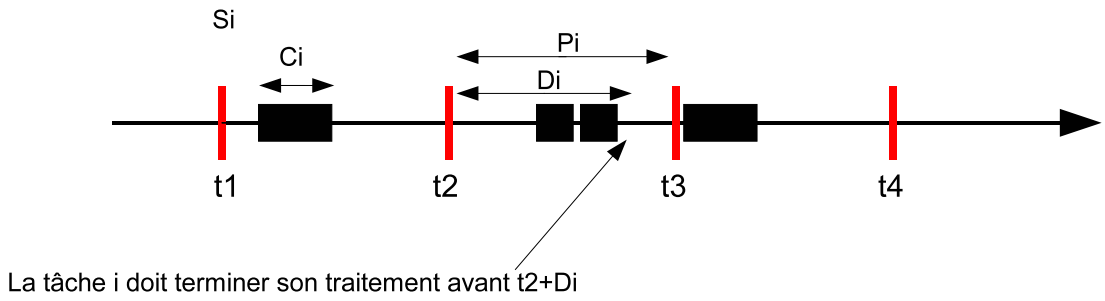

FIG. 2.1: *Une tâche périodique*

Regardons plus précisément comment une tâche périodique peut être définie. Le comportement d'une tâche périodique peut être représenté par la figure 2.1. Chaque barre verticale rouge correspond à la date d'une activation de la tâche périodique. Les rectangles noirs représentent l'exécution du code

de la tâche pour chacune de ses activations. Une tâche périodique i est définie par un ensemble de paramètres dont les plus courants sont [CDKM00] :

- La date de premier réveil (ou première activation) de la tâche, c'est à dire le premier instant où la tâche requiert le processeur. Le premier réveil de la tâche i est noté S_i.
- La durée d'exécution maximale de la tâche quand elle dispose du processeur pour elle seule. On parle aussi de capacité de la tâche, ou encore de WCET[5]. Ce paramètre est généralement noté C_i. La capacité d'une tâche est généralement supposée constante. La capacité est obtenue soit par analyse statique du code de la tâche, soit par l'établissement de mesures sur la plate-forme d'exécution.
- La période d'activation, ou P_i. P_i est un délai fixe entre deux réveils successifs de la tâche i. Pour chaque réveil, la tâche i doit exécuter un travail de C_i unités de temps.
- Le délai critique D_i. Le délai critique d'une tâche est le délai maximum acceptable pour son exécution. Il s'agit donc de la contrainte temporelle que la tâche doit respecter.
- Enfin la priorité de la tâche. La priorité est une information qui permet à l'ordonnanceur de choisir la prochaine tâche prête à exécuter. Il s'agit d'un entier dont la valeur est généralement comprise entre 0 et 255, parfois moins. L'affectation des priorités pour chaque tâche s'effectue selon différents critères : l'urgence (affectation Rate Monotonic ou Deadline Monotonic), l'importance, les contraintes de précédence, le placement des tâches sur les différents processeurs (on parle alors de partitionnement), la plage de priorité mise à disposition par l'environnement d'exécution, ...

Notez que pour les tâches périodiques, on parle de jeu de tâches à échéances sur requêtes lorsque $\forall i : P_i = D_i$.

On suppose en outre que $\forall i : S_i = 0$. Ce qui signifie que toutes les tâches démarrent au même instant, communément appelé l'instant critique. On parle alors de tâches à départs simultanés. L'instant critique permet d'établir des propriétés du système dans une situation de pire cas : en effet, la contention sur la ressource processeur la plus défavorable intervient lorsque toutes les tâches sont réveillées simultanément. Si l'instant critique intervient tôt dans l'exécution du système, on peut vérifier tout aussi tôt que les contraintes temporelles sont effectivement vérifiées par l'application. De nombreux tests de faisabilité sont construits sur l'hypothèse que cet instant critique existe.

2.1.2 Quelques généralités sur les algorithmes d'ordonnancement temps réel

Un ordonnanceur temps réel doit permettre le partage du processeur tout en tenant compte des besoins d'urgence et d'importance de l'application, c'est à dire des tâches qui la composent.

Il existe de très nombreux algorithmes d'ordonnancement temps réel et ils peuvent être classés selon d'innombrables critères [PI76]. Nous retiendrons néanmoins qu'un algorithme d'ordonnancement peut être préemptif ou non, hors ligne ou en ligne, à priorité fixe ou à priorité dynamique.

Un algorithme d'ordonnancement à priorité fixe est un algorithme dont les priorités sont statiquement affectées : les priorités ne changent pas durant la vie de l'application. À l'inverse, un algorithme à priorité dynamique peut modifier la priorité des tâches durant l'exécution de l'application.

[5]Worst Case Execution Time.

Un algorithme d'ordonnancement hors ligne est un algorithme où les décisions d'ordonnancement sont prises avant l'exécution des tâches. Pour de nombreuses applications très critiques, ce type d'ordonnancement est séduisant car le déterminisme de l'application est élevé. Avec un ordonnancement en ligne, les choix d'allocation des ressources sont effectués pendant l'exécution de l'application. Cette solution produit un ordonnancement plus flexible et adaptable, mais aussi parfois moins prédictible.

Enfin, un algorithme préemptif est un algorithme qui est autorisé à suspendre à tout moment la tâche en cours d'exécution si une tâche plus prioritaire devient prête. En général, un algorithme préemptif est plus à même de respecter les échéances d'un jeu de tâches que ne l'est un algorithme non préemptif. Néanmoins, un algorithme non préemptif facilite la gestion des ressources partagées car il n'est plus utile d'employer des sémaphores. D'autre part, le surcoût d'un algorithme non préemptif est moins élevé car il génère moins de commutation de contexte.

Pour une architecture donnée, le choix de l'algorithme à employer peut être motivé par de nombreuses contraintes telles que l'environnement d'exécution, les besoins de prédictibilité ou d'efficacité. Bien sûr, un algorithme d'ordonnancement doit pouvoir être aisément implantable dans un environnement d'exécution (s'il s'agit d'un algorithme en ligne). Le choix d'un algorithme peut aussi être motivé par sa capacité à respecter les contraintes temporelles. Pour désigner l'algorithme le meilleur, on parle d'algorithme optimal. Un ordonnanceur optimal est un ordonnanceur qui peut produire un ordonnancement pour tout ensemble faisable de tâches. Un ensemble de tâches est dit faisable s'il existe un ordonnancement de ces tâches qui respecte toutes les contraintes temporelles des tâches. On parle aussi ordonnancement valide. Enfin, un ordonnanceur temps réel doit disposer de tests de faisabilité permettant de vérifier *a priori* la validité d'un ordonnancement pour un jeu de tâches donné.

Dans la suite, nous regardons deux exemples d'algorithme d'ordonnancement. Le premier est un algorithme à priorité fixe préemptif et en ligne : il s'agit de Rate Monotonic. Le second est un algorithme à priorité dynamique préemptif et en ligne : il s'agit d'Earliest Deadline First.

2.1.3 Un algorithme d'ordonnancement à priorité fixe : Rate Monotonic

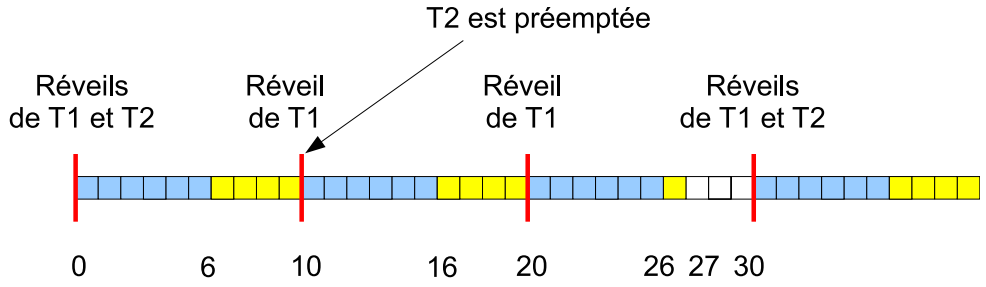

FIG. 2.2: *Exemple d'ordonnancement avec Rate Monotonic préemptif*

Les algorithmes à priorité fixe sont particulièrement bien adaptés aux applications statiques et critiques du fait des méthodes d'analyse statique qui leur sont associées. Une grande majorité des

systèmes d'exploitation temps réel offrent un ordonnanceur à priorité fixe : ce sont des algorithmes simples de mise en œuvre.

L'algorithme à priorité fixe le plus connu est Rate Monotonic. Rate Monotonic peut être présenté à la fois comme un algorithme d'ordonnancement, une méthode d'analyse ou plus simplement, une méthode d'affectation des priorités. C'est l'algorithme optimal dans la classe des algorithmes à priorité fixe dans le contexte de tâches périodiques indépendantes à départs simultanés et à échéances sur requêtes. Bien qu'il existe des solutions pour le support d'autres modèles de tâches tels que les tâches apériodiques [SSL89], Rate Monotonic est essentiellement conçu pour l'ordonnancement de tâches périodiques.

Rate Monotonic peut être présenté comme un algorithme fonctionnant en deux phases : la phase d'affectation de priorité, puis la phase d'élection. La phase d'affectation consiste à associer à chaque tâche une priorité fixe inversement proportionnelle à sa périodicité. Cette première phase est effectuée hors-ligne, lors de la conception détaillée de l'architecture logicielle. La phase d'élection consiste à élire la tâche prête de plus forte priorité, ce qui équivaut à privilégier les tâches de plus petite période, autrement dit la tâche dont la contrainte temporelle est la plus forte (dans l'hypothèse de tâches à échéances sur requêtes).

La figure 2.2 présente un ordonnancement de deux tâches périodiques à échéances sur requêtes. On suppose, dans cet exemple, que les deux tâches sont ordonnancées de façon préemptive. La tâche T1 (en bleu) possède une période de 10 unités de temps et une capacité de 6 unités de temps. La tâche T2 (en jaune) possède une période de 30 unités de temps et une capacité de 9 unités de temps. La tâche périodique T1 est réveillée aux instants 0, 10, 20 et 30. Compte tenu de sa période, la tâche périodique T2 est réveillée aux instants 0 et 30. Cet exemple montre qu'à tout moment, la tâche T1, qui est la tâche de plus forte priorité, dispose du processeur à chaque instant où elle devient prête.

Les algorithmes à priorité fixe sont particulièrement intéressants si l'on cherche à vérifier *a priori* le comportement temporel d'une application. En effet, ces algorithmes ont un comportement déterministe et il existe plusieurs tests de faisabilité de complexité polynomiale permettant de valider un jeu de tâches périodiques. Plusieurs familles de tests de faisabilité existent : test sur le taux d'occupation du processeur, test sur le calcul du pire temps de réponse, ...

Le test sur le pire temps de réponse consiste à comparer le délai critique d'une tâche au plus grand temps de réponse que la tâche subira pendant son exécution. À titre d'exemple, [JP86] propose la méthode suivante pour calculer le pire temps de réponse dans le cas d'ordonnanceur préemptif à priorité fixe avec des tâches indépendantes à échéances sur requêtes à départs simultanés :

$$\begin{cases} \forall i : r_i \leq D_i \\ avec\ r_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil \cdot C_j \end{cases} \tag{2.1}$$

Avec r_i, le pire temps de réponse de la tâche i et $hp(i)$ l'ensemble des tâches plus prioritaires que i. L'équation 2.1 additionne deux délais : le temps d'exécution de la tâche i (c'est à dire C_i) et le temps d'interférence qui est constitué de la somme des délais d'attente impliqués par les tâches plus prioritaires que la tâche i (c'est à dire $\sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil \cdot C_j$).

Dans les conditions ci-dessus, le test de faisabilité est une condition nécessaire et suffisante de validité

d'un jeu de tâches. La validité d'un jeu de tâches est assurée si, pour toute tâche i nous avons $r_i \leq D_i$. Ce test est souvent étendu pour être appliqué avec des modèles de tâches plus réalistes. Ces extensions modélisent les temps d'attente sur les ressources partagées, les latences impliquées par les contraintes de précédence entre les tâches, des retards divers et variés, ... Notez que ce test sur le temps de réponse ne requiert pas que les priorités soient affectées selon Rate Monotonic : ce test est applicable avec toutes méthodes d'affectation de priorité, pour peu que toutes les tâches aient un niveau de priorité différent.

2.1.4 Un algorithme d'ordonnancement à priorité dynamique : Earliest Deadline First

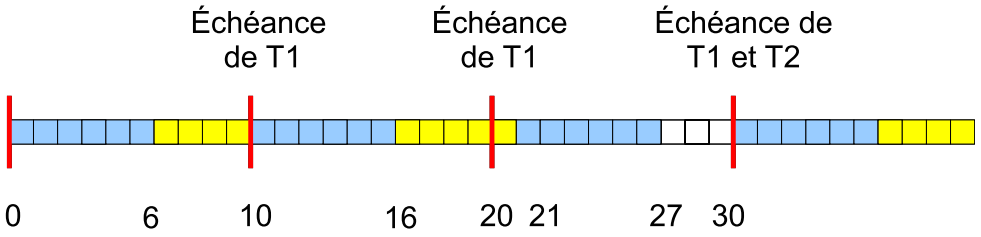

FIG. 2.3: *Exemple d'ordonnancement avec Earliest Deadline First préemptif*

L'algorithme Earliest Deadline First (ou EDF) est un algorithme mieux adapté aux applications dynamiques que Rate Monotonic. EDF ordonnance conjointement les tâches périodiques et les tâches apériodiques de façon naturelle selon une échéance associée à chaque tâche. Cet algorithme est optimal pour les tâches périodiques et pour toutes les classes d'ordonnanceur : EDF est capable de calculer un ordonnancement valide même pour les jeux de tâches occupant le processeur jusqu'à 100 pourcent de sa capacité. Toutefois, la mise en œuvre de cet algorithme est plus difficile dans un système d'exploitation. D'autre part, cet algorithme est instable en surcharge. En cas de surcharge, il peut être difficile de prédire quelle tâche manquera sa contrainte temporelle. Ce n'est donc pas un algorithme adapté aux applications critiques. Cet algorithme peut néanmoins être utilisé avec succès pour calculer un ordonnancement hors-ligne optimal pour ce type d'applications.

Comme Rate Monotonic, EDF peut être présenté comme un algorithme fonctionnant en deux phases : la phase de calcul des priorités, puis la phase d'élection. La phase de calcul des priorités consiste à calculer une échéance pour chaque tâche. Soit $D_i(t)$, l'échéance de la tâche périodique i à l'instant t. L'échéance est calculée par $D_i(t) = k + D_i$ où k est la dernière date de réveil de la tâche i égale ou inférieure à l'instant t. La phase d'élection consiste à exécuter d'abord la tâche de plus courte échéance.

La figure 2.3 présente un ordonnancement de deux tâches périodiques à échéances sur requêtes et à départs simultanés. On suppose, dans cet exemple, que les deux tâches sont ordonnancées de façon préemptive. La tâche T1 (en bleu) possède une période de 10 unités de temps et une capacité de 6 unités de temps. La tâche T2 (en jaune) possède une période de 30 unités de temps et une capacité

de 9 unités de temps. Cet exemple illustre le fait qu'à certains moments, plusieurs tâches ayant des périodes différentes peuvent avoir des échéances identiques. Ainsi, à l'instant 20, les deux tâches ont une échéance égale à 30. De ce fait, il est possible à cet instant d'exécuter soit T1, soit T2. Le fait que certaines tâches puissent avoir à certains instants une échéance identique constitue la cause de non stabilité d'EDF en cas de surcharge.

Même si EDF est moins prédictible que d'autres algorithmes d'ordonnancement, il existe plusieurs tests de faisabilité permettant de valider un jeu de tâches avec cet algorithme d'ordonnancement. Un test sur le temps de réponse a été proposé par [Spu96]. Il existe aussi un test de faisabilité exploitant le taux d'utilisation du processeur. Ce test consiste à comparer le taux d'utilisation U du processeur à une borne. Si le taux d'utilisation U est inférieur ou égal à la borne, alors le jeu de tâches respectera ses contraintes temporelles. Pour EDF, ce test est calculé de la façon suivante :

$$\begin{cases} U \leq 1 \\ avec\ U = \sum_{i=1}^{n} \frac{C_i}{P_i} \end{cases} \tag{2.2}$$

Où n est le nombre de tâches. On suppose ici que les tâches sont périodiques, à départs simultanés et à échéances sur requêtes. Avec EDF, la borne sur le taux d'utilisation est de 1. Dans les conditions ci-dessus, le test de faisabilité est une condition nécessaire et suffisante de validité d'un jeu de tâches.

2.1.5 Extension des tests de faisabilité

À partir des tests de faisabilité que nous avons décrits ci-dessus, de nombreuses extensions ont été proposées afin d'adapter chaque test à un type d'architecture donné. Ainsi, nous avons supposé les tâches indépendantes. Mais en pratique, il est bien rare qu'une tâche soit réellement indépendante. Classiquement, il existe deux types de dépendance entre tâches.

La première famille de dépendance est relative au partage de données. Par exemple, deux tâches peuvent partager une information par le biais d'une zone de mémoire commune. La donnée est généralement protégée par un sémaphore [Tan01]. Cette dépendance de données peut conduire l'une des deux tâches dans l'état bloqué lorsque celle-ci demande l'accès à la donnée et que le sémaphore associé a déjà été verrouillé par la seconde tâche. L'ordonnancement de la première tâche ne peut plus être élaboré indépendamment de la seconde.

Le second type de dépendance entre tâches classiquement rencontré est la contrainte de précédence. Ici, une tâche ne peut être exécutée qu'à la terminaison d'une seconde. Ce type de contrainte est fréquent dans les systèmes répartis où, par exemple, une tâche restera bloquée tant qu'elle ne recevra pas un message d'une tâche située sur un processeur distant.

Nous regardons par la suite deux exemples d'extensions des tests de faisabilité illustrant ces dépendances entre tâches.

2.1.5.1 Illustration avec le partage de ressources

Lorsqu'une tâche souhaite accéder à une ressource protégée par un sémaphore, il est possible que la tâche attende pour obtenir cette ressource. Il est alors nécessaire d'évaluer ce temps d'attente afin de

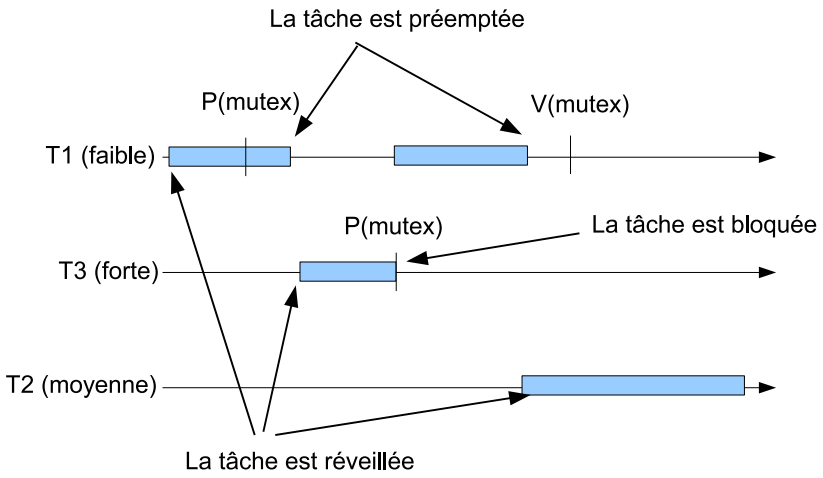

FIG. 2.4: *Scénario d'ordonnancement avec inversion de priorité*

l'intégrer dans le pire temps de réponse de la tâche, comme nous l'avions fait avec l'équation 2.1 pour l'interférence des tâches de plus forte priorité.

Ce délai d'attente, noté B_i, est parfois difficile à évaluer finement de façon statique, en particulier lorsque les sémaphores utilisés ne tiennent pas compte du phénomène d'inversion de priorité.

La figure 2.4 présente ce qu'est une inversion de priorité. Dans cet exemple, deux tâches accèdent à une ressource partagée protégée par un sémaphore. Le sémaphore, nommé *mutex*, est accédé par la tâche T1 et T3. On suppose que ces tâches sont ordonnancées par un algorithme à priorité fixe. La tâche T3 est la tâche de plus forte priorité. On assigne à la tâche T1 la priorité la plus faible. Enfin, on suppose que la troisième tâche T2 ne requiert pas d'accès à la ressource partagée et qu'elle est d'un niveau de priorité intermédiaire. La figure 2.4 présente un ordonnancement de ces trois tâches qui conduit à une inversion de priorité. Cette inversion de priorité se manifeste par le fait que la tâche T3 soit bloquée par la tâche T2. Ce blocage de T3 par T2 est provoqué par la préemption de T1 lors du réveil de T2. T1 ayant précédemment verrouillé le sémaphore, la tâche T3 ne peut être réveillée tant que T1 n'a pas libéré le sémaphore *mutex*. La tâche T2 induit donc un délai d'attente supplémentaire qui risque d'impliquer un échec du respect des contraintes temporelles de T3.

Pour réduire le délai d'attente induit par les inversions de priorité, un système d'exploitation temps réel propose généralement une implémentation spécifique des sémaphores. Ces implémentations mettent en œuvre des protocoles tels que PCP[6] [SRL90], PIP[7] ou SRP[8] [Bak91].

Le protocole le plus souvent implanté dans les environnements d'exécution temps réel est PCP. Ainsi, Ada 2005 propose d'utiliser une variante de PCP dénommée ICPP et qui fonctionne comme suit :

– On suppose qu'une priorité statique est affectée à chaque tâche. Les tâches sont ordonnancées par

[6]Priority Ceiling Protocol.
[7]Priority Inheritance Protocol.
[8]Stack Based Resource Protocol.

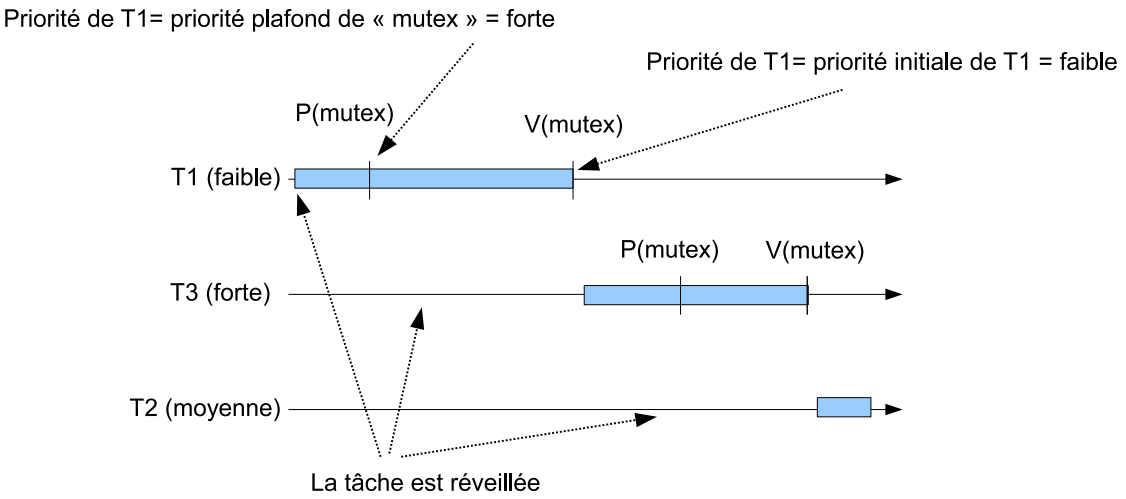

FIG. 2.5: *Scénario d'ordonnancement avec ICPP*

un algorithme à priorité fixe.

- Une priorité plafond est associée à chaque ressource. Cette priorité plafond est égale à la priorité statique maximale de toutes les tâches qui utilisent la ressource.
- À tout moment pendant l'exécution de l'application, la priorité de chaque tâche est égale au maximum de sa priorité statique et des priorités plafonds de toutes les ressources allouées par la tâche.
- Le délai d'attente B_i peut être évalué facilement car il s'agit de la plus grande section critique associée à la ressource manipulée par la tâche i.

La figure 2.5 présente un second scénario d'ordonnancement pour nos deux tâches qui accèdent à une ressource partagée grâce au sémaphore *mutex*. Cette fois, nous supposons que le sémaphore *mutex* est manipulé conformément au protocole ICPP. La priorité plafond du sémaphore et donc égale à la priorité de la tâche T3. Grâce à cette priorité plafond élevé, la tâche T2 ne peut plus préempter la tâche T1. En effet, lors de l'accès au sémaphore, T1 hérite du niveau de priorité du sémaphore *mutex*. Cet héritage de priorité garantit à T3 une libération rapide du sémaphore par T1.

À partir d'une évaluation de B_i, il est possible d'étendre les tests de faisabilité tels que ceux des équations 2.1 et 2.2. Ainsi, avec EDF, le test sur le taux d'occupation du processeur devient :

$$\begin{cases} \forall\, i, 1 \leq i \leq n : U \leq 1 \\ avec\ U = \sum_{k=1}^{i-1} \frac{C_k}{P_k} + \frac{C_i + B_i}{P_i} \end{cases} \tag{2.3}$$

Avec le test de l'équation 2.3, on suppose que les tâches sont ordonnées de façon décroissante selon leur priorité : la tâche $i-1$ est donc moins prioritaire que la tâche i.

De même, le test sur le temps de réponse dans le cas d'ordonnanceur à priorité fixe peut être adapté comme suit :

$$\begin{cases} r_i \leq D_i \\ avec\ r_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil \cdot C_j \end{cases} \tag{2.4}$$

2.1.5.2 Illustration avec les contraintes de précédence

Une deuxième famille d'extension des tests de faisabilité permet au concepteur de vérifier un jeu de tâches avec contraintes de précédence. Il existe différentes familles de contraintes de précédence. Dans cette partie, nous regardons comment vérifier les contraintes temporelles de tâches qui ne peuvent être réveillées avant la terminaison d'autres. Avec le test de faisabilité sur le temps de réponse, si une tâche i ne peut démarrer avant la terminaison d'une tâche j, le problème consiste alors à calculer le temps de réponse de i afin qu'il englobe ce nouveau délai d'attente.

Pour ce faire, [ABRT93] ont proposé l'utilisation d'un paramètre : la gigue sur le réveil d'une tâche, paramètre traditionnellement noté J_i. J_i est une borne maximale sur le retard qu'une tâche peut subir lors de son réveil.

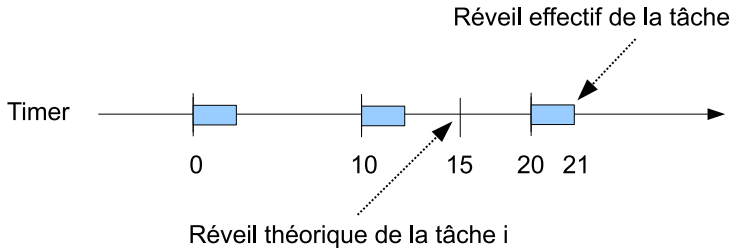

FIG. 2.6: *Retard sur réveil*

Initialement, ce paramètre fut proposé afin d'incorporer dans le modèle de tâches certaines perturbations dues à l'environnement d'exécution. La figure 2.6 illustre l'utilisation de la gigue lors du réveil d'une tâche périodique. Dans ce scénario, ou souhaite réveiller une tâche i à l'instant $t = 15\ ms$. Le réveil de la tâche est effectué par le *timer* du système qui est modélisé comme une tâche périodique dont les paramètres sont $P_{timer} = 10\ ms$, $C_{timer} = 1\ ms$. Ce *timer* est déclenché par une interruption émise par le circuit d'horloge toutes les 10 millisecondes. À chaque interruption, le *timer* met à jour l'horloge du système, puis vérifie et éventuellement réveille les tâches qui doivent l'être. Si l'on regarde précisément la figure 2.6, on constate que la date effective de réveil de la tâche i est $t = 21\ ms$. Elle a donc subi un retard de 6 ms dû au fonctionnement du *timer*. En effet, le *timer* est réveillé aux instants 0, 10 et 20. À l'instant 20, le *timer* détecte qu'il est nécessaire de réveiller la tâche i. Puisque la capacité du *timer* est d'une milliseconde, nous sommes assuré que la tâche i est effectivement réveillée en $t = 21$ ms.

Si l'on ne prend pas garde à ce retard lors du calcul du temps de réponse, alors le temps de réponse sera faux. Pour ce faire, le paramètre gigue peut être employé pour modéliser ce retard. Le retard est alors intégré dans le temps de réponse de la tâche i en posant $J_i = 6\ ms$ et en calculant le temps de réponse comme suit :

$$\begin{cases} r_i \leq D_i \\ r_i = w_i + J_i \end{cases} \tag{2.5}$$

avec :

$$w_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i + J_j}{P_j} \right\rceil \cdot C_j$$

De la même façon, l'équation 2.5 permet de calculer un temps de réponse qui tienne compte d'une contrainte de précédence. Si une tâche i doit démarrer après la terminaison d'un tâche j, on peut appliquer le principe qui consiste à considérer le temps de réponse de la tâche j comme la gigue sur l'activation de la tâche i.

La méthode de calcul holistique exploite cette façon d'exprimer une contrainte de précédence. Avec le calcul holistique, les tâches sont organisées en chaînes de traitements. Une chaîne de traitements est constituée de plusieurs tâches exécutées successivement. Ces tâches sont donc sujettes à des contraintes de précédence. Les tâches sont éventuellement placées sur des processeurs différents et communiquent par échange de messages. Ici, il ne s'agit plus de vérifier le respect d'une contrainte qui est locale à un processeur donné, mais de vérifier un délai de bout en bout : on cherchera à contrôler, par exemple, que le délai entre le démarrage de la première tâche et la terminaison de la dernière tâche d'une chaîne de traitements est inférieur à un délai critique donné. Le calcul du pire délai de bout en bout est réalisé par un calcul de proche en proche des temps de réponse des tâches : c'est le calcul holistique proposé initialement par [TC94].

$\forall i : J_i := 0, r_i := 0, r'_i := 0;$
$\forall i :$ Calculer_temps_de_réponse (r_i) ;
Tant que $(\exists i : r_i \neq r'_i)$ {
  $\forall i : J_i := max(J_i, \forall j \; avec \; j \prec i : r_j + M_{ji});$
  $\forall i : r'_i := r_i;$
  $\forall i :$ Calculer_temps_de_réponse (r_i) ;
}

FIG. 2.7: *Calcul holistique*

La figure 2.7 décrit la méthode de calcul des délais de bout en bout utilisée par le calcul holistique. La modification des gigues et le calcul des temps de réponse sont réalisés jusqu'à convergence de ces derniers. Dans la figure 2.7, M_{ji} désigne le pire délai d'acheminement d'un message expédié par la tâche j et à destination de la tâche i. L'opérateur $\prec$ exprime l'existence d'une contrainte de précédence entre deux tâches successives dans une chaîne de traitements.

2.1.6 Conclusion

Dans cette première partie du chapitre, nous avons décrit deux exemples d'ordonnanceur. Puis, plusieurs tests de faisabilité ont été présentés. La littérature propose d'innombrables tests de faisabilité pour de nombreux ordonnanceurs et modèles de tâches. L'annexe A énumère un échantillon de ces tests pour le modèle de tâches périodiques. Cet échantillon est celui sélectionné pour être implanté dans la plate-forme de vérification Cheddar que nous décrivons dans la suite de ce chapitre.

2.2 Survol de l'approche : présentation de l'environnement de vérification Cheddar

Dans cette partie, nous présentons l'environnement de vérification Cheddar. En prenant l'exemple des méthodes analytiques, nous illustrons pourquoi la vérification des performances d'une architecture temps réel est une tâche difficile avec la théorie de l'ordonnancement temps réel. Nous décrivons succinctement la solution proposée par l'environnement Cheddar. La structure et le fonctionnement de l'environnement sont d'abord décrits. Puis, nous expliquons quelques exemples types d'utilisation de l'environnement.

2.2.1 La vérification des performances d'une architecture temps réel : une activité difficile

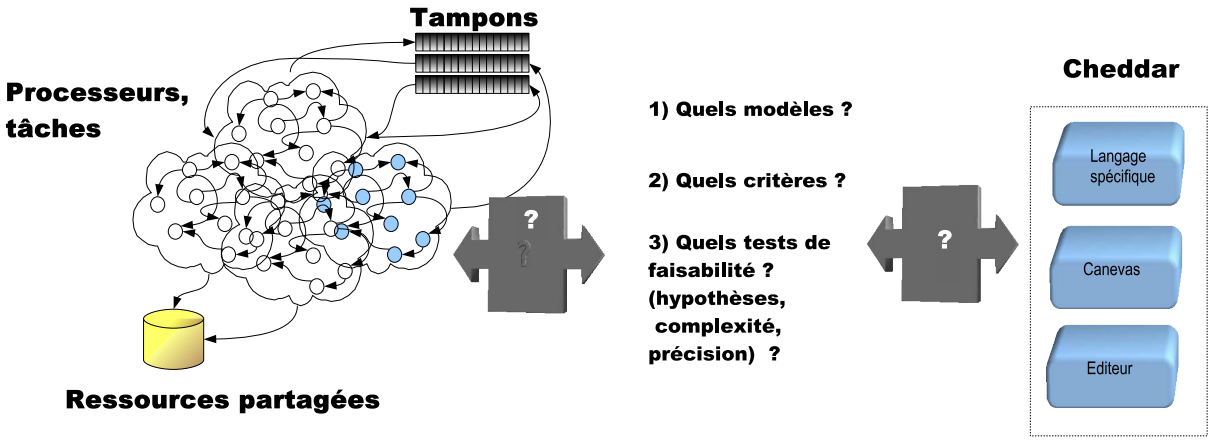

FIG. 2.8: *De la modélisation à l'analyse de performances*

Dans les parties 2.1.3 et 2.1.4, nous avons décrit quelques exemples de test de faisabilité qui permettent de valider *a priori* les contraintes temporelles d'un jeu de tâches. En pratique, pour appliquer un test de faisabilité, le concepteur devra vérifier toute une série d'hypothèses sur l'environnement d'exécution, la conception du système, les caractéristiques des tâches, ... Ainsi, à titre d'exemple, l'équation

2.1 suppose que $\forall i : D_i \leq P_i$ et que toutes les tâches périodiques sont initialement réveillées simultanément (hypothèse de l'instant critique). Par ailleurs, il existe de très nombreux tests de faisabilité selon :

- Le modèle de tâches : les tâches sont elles périodiques/apériodiques/sporadiques ? harmoniques ? indépendantes/dépendantes ? à échéances sur requêtes/à échéances quelconques ? à départs simultanés/départs différés ? Comment les priorités sont elles affectées aux tâches ? ...
- L'ordonnanceur : est il préemptif/non préemptif ? oisif/non oisif ?
- Les ressources partagées : quel protocole d'accès utilise-t-on ? est on capable de déterminer un pire temps d'attente sur la ressource avec le protocole employé ?
- La qualité du test de faisabilité : délivre-t-il une condition de validité suffisante seulement, nécessaire seulement ou nécessaire et suffisante ?

Pour le concepteur, il devient alors difficile, s'il n'est pas un expert de la théorie de l'ordonnancement temps réel, d'utiliser le bon test de faisabilité, car pour chaque entité de son architecture à vérifier (cf. figure 2.8), il devra :

1. Choisir le critère de performance à vérifier.
2. Choisir le bon modèle pour chaque entité de son architecture. Par exemple, le concepteur doit s'interroger sur la pertinence de modéliser ses traitements par des tâches périodiques ou par des tâches sporadiques. Il doit sélectionner le niveau d'abstraction permettant d'appliquer un test de faisabilité tout en garantissant que le modèle reste proche de la réalité. Ce compromis nécessite du concepteur qu'il connaisse un certain nombre de modèles de tâches et qu'il comprenne leurs avantages et leurs inconvénients respectifs.
3. Choisir un test de faisabilité qui puisse calculer le critère sélectionné en (1) et qui soit compatible avec le modèle sélectionné en (2). Le concepteur doit connaître les hypothèses d'application du test de faisabilité et être capable de vérifier que son modèle et son environnement d'exécution les respectent. Il devra aussi savoir si ce test est pessimiste ou s'il délivre un résultat exact.

Enfin, l'analyse des performances d'une architecture peut être rendue complexe car les ressources ne peuvent pas toujours être analysées indépendamment les unes des autres. Il existe bien souvent des compromis entre les différentes ressources.

Bien entendu, dans de nombreux cas, ce travail d'analyse est simple car l'architecture à vérifier l'est aussi. Un outil de vérification basé sur la théorie de l'ordonnancement temps réel doit donc offrir plusieurs niveaux d'utilisation.

Une première solution pour aider le concepteur dans sa tâche de vérification avec la théorie de l'ordonnancement temps réel consiste, par exemple, à automatiser le calcul de ces tests de faisabilité. Différents outils basés sur cette théorie existent aujourd'hui : MAST [HGGM01], Rapid-RMA [TP03], Timewiz [Tim02], TIMES [FMPY06, AFM+03] , TkRTS [Mig99], YASA [BoHT03], ... Depuis 2001, le LISyC met à disposition de la communauté un ensemble d'outils : l'environnement Cheddar [SLNM04].

2.2.2 L'environnement de vérification Cheddar : structure et fonctionnement

Cheddar est un ensemble d'outils composé d'un canevas logiciel (ou *framework*) de vérification, d'un éditeur simplifié et d'un langage spécifique de modélisation (cf. figure 2.8).

L'éditeur simplifié de Cheddar permet de décrire l'architecture à analyser. Néanmoins, la démarche naturelle consiste à utiliser un outil de modélisation pour l'édition d'un tel modèle. À ce jour, Cheddar est interopérable avec des outils de modélisation tels que Stood [DS08], TOPCASED [FGC+06], IBM Rational Software Architect [Mae07] ou PPOOA-Visio [FM08].

Le canevas logiciel est constitué d'un ensemble de paquetages Ada implantant les principales méthodes analytiques et les algorithmes d'ordonnancement temps réel classiques. On propose ici une bibliothèque de méthodes disponibles au concepteur pour la vérification de son architecture.

Le langage spécifique ainsi que ses outils associés (interpréteur, parseur, ...) permet de modéliser un ordonnanceur temps réel qui n'est actuellement pas implanté dans le canevas logiciel et d'en étudier le comportement par simulation. S'il est possible d'effectuer un simulation exhaustive, alors la simulation peut conduire à une preuve de la propriété recherchée. Dans la suite de ce mémoire, on parlera de programme Cheddar pour désigner ces modèles d'ordonnanceur.

Nous décrivons succinctement l'utilisation de Cheddar lorsque l'on souhaite effectuer une analyse de performance :

1. Il est d'abord nécessaire de modéliser l'architecture que l'on souhaite analyser.

   L'architecture à analyser peut être décrite, soit dans un format propriétaire en XML, soit par un modèle AADL[9]. AADL est un langage graphique et textuel normalisé par la SAE[10] sous le standard AS-5506 [SAE04].

   On suppose qu'une architecture est un ensemble hiérarchique de composants logiciels et matériels tels que des processeurs, des tâches, des tampons, des ressources partagées, ... La figure 2.9 propose un diagramme de classe spécifiant le modèle de données de Cheddar. Ce modèle de données décrit ce qu'est une architecture pouvant être analysée par Cheddar. Dans le cas d'une vérification d'un modèle AADL, il s'agit alors de décrire le système en terme de composants AADL tels que des composants *processor*, *thread* ou *data*. Chaque composant AADL est décoré de propriétés décrivant son comportement logique et temporel. Ainsi, il est possible de spécifier la période d'une tâche, sa capacité, le type d'ordonnanceur associé à un processeur donné, ...

   Bien qu'il existe des propriétés standardisées par AADL, Cheddar propose un certain nombre d'extensions autorisant l'application des méthodes d'analyse proposées par la théorie de l'ordonnancement temps réel [Sin07b]. Ces extensions sont issues du modèle de données de Cheddar.

2. L'utilisateur peut alors utiliser deux types de services offerts par le canevas logiciel de Cheddar : la simulation ou l'application de tests de faisabilité. Le choix entre l'un, l'autre ou les deux types d'outils dépend des caractéristiques du modèle à analyser, de l'expertise du concepteur, ... Après

[9] Architecture Analysis and Design Language.
[10] Society of Automotive Engineers.

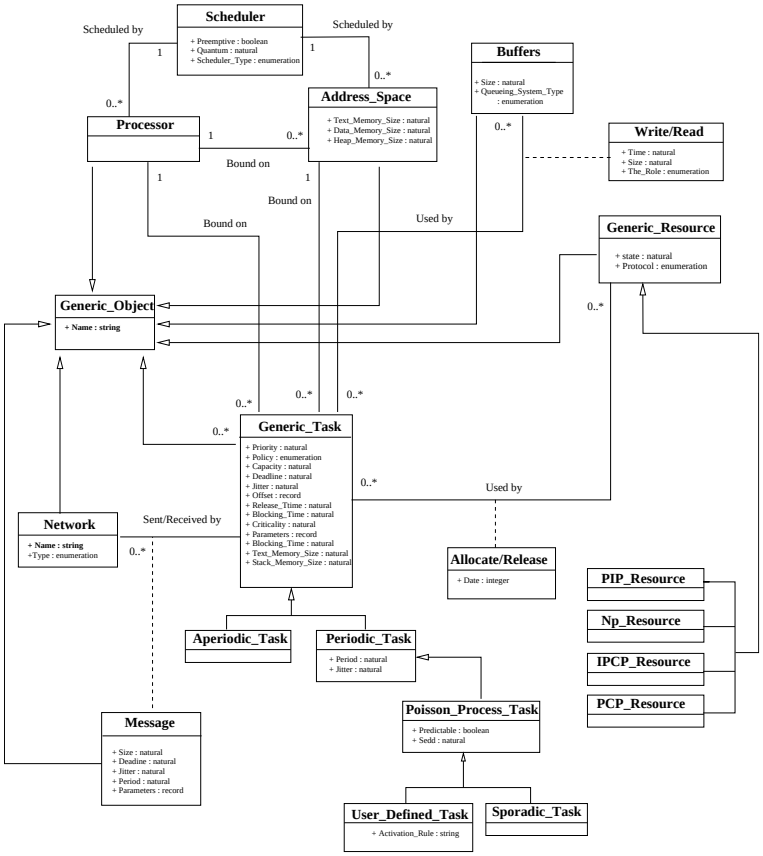

FIG. 2.9: *Diagramme de classes du modèle de données de Cheddar*

analyse, les résultats peuvent être soit affichés dans l'éditeur simplifié de Cheddar, soit exportés vers d'autres outils grâce à un flux XML propriétaire.

2.2.3 Quelques usages possibles de l'environnement Cheddar

Cheddar offre différents niveaux d'utilisation, selon le type d'architecture à analyser, selon l'outil de modélisation conjointement utilisé, selon le niveau d'expertise du concepteur, ... Nous décrivons ici quelques exemples typiques d'utilisation qui doivent aider le lecteur à comprendre comment un tel environnement peut être employé.

2.2.3.1 Usage de patrons de conception

Lorsque le modèle à analyser comporte des algorithmes d'ordonnancement classiques et déjà implantés dans le canevas logiciel, l'utilisateur exploite l'interface homme/machine pour éditer son modèle et en effectuer une analyse de performances. Cette première façon d'utiliser l'environnement ne demande pas d'expertise particulière de la part du concepteur concernant l'utilisation de Cheddar : il suffit

simplement de charger un modèle d'architecture et de réaliser l'analyse en pressant un bouton de l'éditeur de Cheddar. Dans ce cas, Cheddar choisit le critère de performance, choisit le test de faisabilité et contrôle automatiquement si les hypothèses du test sont vérifiées. On suppose dans ce cas que le concepteur utilise un patron de conception qui soit supporté par Cheddar afin de décrire son architecture. Chaque patron de conception propose une solution à une famille de problèmes de concurrence. À titre d'exemple, le concepteur peut employer le patron de conception «Ravenscar». Ravenscar est défini dans le standard international Ada 2005 [ISO07, TDB+06]. Il s'agit d'un sous ensemble du langage Ada 2005, et en particulier de ses fonctionnalités ayant attrait à la concurrence. Ce sous-ensemble d'Ada assure que tout programme compilé est compatible avec les méthodes algébriques de vérification de la théorie de l'ordonnancement temps réel. Cette première méthode d'utilisation est la plus adaptée dans le cadre d'un cours d'initiation à la théorie de l'ordonnancement temps réel.

2.2.3.2 Une bibliothèque de méthodes analytiques

Un deuxième usage classique de l'environnement Cheddar consiste à laisser le concepteur choisir le critère de performance et le test de faisabilité à calculer. Le concepteur doit alors configurer le fonctionnement du canevas logiciel via les menus de l'éditeur de Cheddar. Le canevas logiciel assure alors, toujours de façon automatique, la vérification des hypothèses du test qu'il doit calculer. Toutefois, le concepteur doit s'assurer que le critère de performance est pertinent et que la méthode de calcul (c'est à dire le test de faisabilité) constitue la méthode adéquate pour évaluer le critère avec l'architecture qu'il souhaite analyser.

Le canevas logiciel de cheddar est alors employé comme une bibliothèque de tests de faisabilité. Dans les parties 2.1.3 et 2.1.4 de ce chapitre, nous avons vu quelques exemples de tests de faisabilité. Il existe toutefois de nombreux autres tests. Afin de simplifier le travail d'analyse du concepteur, deux ensembles de tests de faisabilité ont été sélectionnés : un ensemble de tests basé sur le taux d'occupation du processeur et un ensemble de tests sur le temps de réponse des tâches.

Ces deux familles de tests sont complémentaires. Par exemple, les tests basés sur le taux d'occupation du processeur passent plus facilement à l'échelle lorsque le nombre de tâches est grand, mais sont limités à des architectures simples. De même, les tests sur le temps de réponse sont plus complexes à calculer, mais permettent au concepteur de se focaliser sur certaines tâches de l'architecture. Par ailleurs, de nombreux tests sont pessimistes et ne fournissent pas un résultat nécessaire et suffisant. L'utilisation de plusieurs tests peut alors être nécessaire.

Les tests que nous avons sélectionnés pour le canevas logiciel de Cheddar sont décrits dans l'annexe A. Les tests sélectionnés supposent que :

1. Les tâches de l'architecture sont périodiques uniquement. En effet, les résultats les plus simples et les plus généraux ont été élaborés pour ce modèle de tâches. C'est aussi le modèle de tâches le mieux adapté pour les architectures critiques.
2. L'architecture emploie les ordonnanceurs classiques suivants :
   - Rate Monotonic, Deadline Monotonic et l'ordonnancement hiérarchique POSIX 1003.1b pour les ordonnanceurs à priorité fixe. Ces algorithmes constituent ceux actuellement employés dans

les plates-formes d'exécution temps réel.
- Earliest Deadline First et Least Laxity First pour les ordonnanceurs à priorité dynamique. Ces algorithmes ont été choisis pour leur optimalité. Ils constituent donc de bons candidats lorsque des ordonnancements hors-ligne doivent être calculés.

3. Les architectures sont préalablement partitionnées. Bien que des méthodes de partitionnement sont implantées dans le canevas logiciel de Cheddar, on suppose que le déploiement des composants logiciels sur l'architecture matérielle a été effectué avant vérification. On suppose donc que les priorités sont affectées aux tâches et que les tâches sont affectées aux processeurs. Notez que cette hypothèse ne signifie pas que l'on ne puisse pas vérifier des propriétés réparties par nature. Ainsi, le calcul d'un pire délai de bout en bout réalisé par la méthode holistique est possible avec cette hypothèse.

4. Les ressources sont partagées grâce aux protocoles PIP, PCP ou SRP. Ces protocoles sont les plus couramment utilisés et référencés.

2.2.3.3 Vérification par simulation : usage du langage spécifique

Lorsque les tests de faisabilité implantés dans le canevas logiciel de Cheddar ne peuvent pas être appliqués, une architecture peut être vérifiée grâce à la simulation.

Si les ordonnanceurs ou les modèles de tâches spécifiques de son architecture ne sont pas préalablement implantés dans le canevas logiciel, le concepteur doit alors étendre le canevas logiciel.

Pour ce faire, l'environnement Cheddar offre un langage spécifique ainsi qu'un ensemble d'outils associés pour la modélisation et la mise en œuvre d'algorithmes d'ordonnancement temps réel. L'utilisation de ce langage permet au concepteur de définir un algorithme et de le tester rapidement, avant d'exploiter son modèle pour générer son simulateur et effectuer une analyse de performances. En plus de comprendre le fonctionnement du moteur de simulation, le concepteur doit alors maîtriser le langage de modélisation d'ordonnanceurs temps réel.

2.2.3.4 Modification directe du canevas logiciel : ajout d'une méthode de simulation ou d'une méthode analytique

Enfin, un dernier cas d'utilisation possible est celui où l'algorithme d'ordonnancement, le modèle de tâches ou, plus simplement, l'architecture à analyser est trop spécifique pour être exprimée avec les différents langages présentés ci-dessus. Dans ce cas, le concepteur doit modifier le canevas logiciel de Cheddar afin de les implanter. Cette dernière alternative est la plus coûteuse pour le concepteur car il lui est alors nécessaire de comprendre et modifier l'architecture et la mise en œuvre du canevas logiciel. L'existence du langage spécifique à pour objet d'éviter, tant que faire se peut, la modification directe du canevas logiciel par le concepteur afin que son utilisation reste la plus simple possible.

2.3 Conclusion

Dans ce chapitre, nous avons décrit les fondements de la théorie de l'ordonnancement temps réel. Nous avons présenté les modèles de tâches et les ordonnanceurs les plus courants. Nous avons expliqué ce que sont les tests de faisabilité et comment ils peuvent être utilisés.

Puis, nous avons brièvement décrit l'environnement de vérification Cheddar. Celui-ci implante les principales méthodes analytiques et algorithmiques constituant la théorie de l'ordonnancement temps réel. De ce point de vue, il constitue un outil intéressant pour la formation des ingénieurs logiciels.

Nous avons finalement décrit quelques usages possibles de cet environnement. Son usage dépend du processus d'ingénierie logicielle appliqué par l'organisation du concepteur d'architectures. Un environnement comme Cheddar doit donc être adapté au processus employé dans l'entreprise. La liste des usages présentés dans ce chapitre n'est donc pas exhaustive.

Dans la suite de ce mémoire, nous nous focalisons sur deux des usages décrits précédemment. Dans le premier, le concepteur emploie des méthodes analytiques en modélisant son architecture avec un ou plusieurs patrons de conception. Ce premier usage est l'objet du chapitre 3. Dans le second, le concepteur étend le canevas logiciel grâce à un langage spécifique permettant la modélisation d'ordonnanceurs ou de modèles de tâches spécifiques. Ce second usage est l'objet du chapitre 4.

Chapitre 3

Accroître l'applicabilité de la théorie de l'ordonnancement temps réel : du langage d'architecture à l'analyse de performance

Sommaire

Afin de pouvoir appliquer la théorie de l'ordonnancement temps réel, le concepteur doit comprendre comment fonctionne cette théorie et dans quelles conditions elle peut être appliquée. En effet, pour chacun des tests de faisabilité présentés dans le chapitre précédent, un ensemble d'hypothèses doit être vérifié sur l'architecture à analyser afin de garantir l'applicabilité et la validité du test. Un moyen pour aider le concepteur consiste à encapsuler ce savoir dans le processus d'ingénierie aux travers de patrons de conception et d'un langage de conception.

Dans le contexte de l'analyse de performances, chaque patron de conception décrit une solution architecturale adaptée à une famille de problèmes récurrents de concurrence tels que le partage des ressources ou la synchronisation des tâches. À chaque patron est associé un ensemble de tests de faisabilité permettant d'en vérifier les performances. Par construction, l'emploi d'un patron de conception pour la modélisation d'une architecture garantit que celle-ci est conforme aux hypothèses des tests de faisabilité.

Par le passé, la méthode HRT-HOOD fut l'un des exemples les plus significatif de cette approche [BW94]. Plus récemment, [PV07] ont proposé un méta-modèle basé sur Ravenscar autorisant l'analyse de performance.

Plusieurs langages en cours d'élaboration ou de normalisation permettent de préciser, dans un modèle de conception, les données nécessaires à l'application de tests de faisabilité. C'est le cas du profil UML MARTE récemment adopté par l'OMG [FGD06]. Depuis 2004, la SAE [11] propose de son coté un langage graphique et textuel : le langage AADL [12]. AADL offre un support pour l'ingénierie dirigée par les modèles adapté aux systèmes embarqués temps réel. Ce langage de conception a été publié en tant que norme internationale sous le standard SAE AS-5506 [SAE04, SFR+07]. Ce standard comprend différentes annexes qui décrivent, par exemple, les règles de traduction d'un modèle vers un langage de programmation. Une de ses annexes, l'annexe comportementale, décrit un langage basé sur les automates temporisés destiné à l'expression du comportement de certains composants d'un modèle AADL [FBFR07, BDF+06, DBF+06]. Dans le contexte du projet Cheddar, nous étudions comment l'utilisation d'un langage de conception peut automatiser l'application de la théorie de l'ordonnancement temps réel. Dans ce chapitre, nous regardons comment il est possible d'appliquer cette théorie sur un modèle d'architecture avec le langage AADL. Le choix d'un standard international tel qu'AADL est motivé par le souhait de favoriser, tant que faire se peut, l'interopérabilité entre les outils de modélisation et d'analyse.

Dans un premier temps, puisque l'outil Cheddar implante les tests classiques de faisabilité et les principaux algorithmes d'ordonnancement, nous avons employé Cheddar afin de vérifier que la première version du standard AADL était compatible avec la théorie de l'ordonnancement temps réel. Puis, nous avons étudié un ensemble de patrons de conception compatibles avec ces tests de faisabilité [DS08]. D'une part, ces tests permettent de vérifier des contraintes temporelles élémentaires d'un modèle AADL. D'autre part, nous avons proposé des méthodes analytiques basées sur la théorie des files d'attente qui permettent d'estimer l'empreinte mémoire d'une architecture spécifiée avec AADL [SLNM05].

Ce chapitre présente les principaux travaux que nous avons conduits autour d'AADL. Dans la

[11] Society for Automotive Enginners.
[12] Architecture Analysis and Design Language.

partie 3.1, une brève présentation du standard AADL est proposée. La partie 3.2 décrit les différentes familles d'architecture logicielle exprimées avec AADL qui sont vérifiables avec Cheddar. Ces familles sont décrites avec AADL par des patrons de conception proposant des solutions architecturales à des problèmes classiques de concurrence. Nous introduisons quatre patrons de conception : «Synchronous data-flows», «Ravenscar», «BlackBoard» et «Queued buffer».

Le choix de ces patrons est justifié par les pratiques industrielles, mais aussi pour leur complémentarité. En effet, ils expriment différents niveaux de prédictibilité, de flexibilité et de compléxité d'analyse. Ainsi, le patron «Synchronous data-flows» est celui qui est le plus prédictible mais aussi le moins flexible. Ce patron correspond aux systèmes qui doivent être certifiés et où toute inprédictibilité ne peut pas être tolérée. Ce patron est relatif à un système dont les ressources sont généralement très limitées et dont le niveau de criticité peut être très élevé. A l'autre extrême, le patron «Queued buffer» tente de fournir des moyens d'analyse pour des systèmes dont certaines fonctions sont critiques et d'autres moins. Les fonctions non critiques sont alors caractérisées par des lois stochastiques (ex : loi d'arrivée des tâches apériodiques). Les patrons «Ravenscar» et «BlackBoard» constituent des patrons dont la prédicitibilité et la flexibilité sont intermédiaires.

Les patrons «Ravenscar» et «Queued buffer» sont présentés de façon plus détaillée dans les deux parties suivantes de ce chapitre. Pendant la présentation du patron «Ravenscar», nous discutons également de l'adéquation de la première version du standard AADL avec la théorie de l'ordonnancement temps réel. Nous énumérons rapidement les paramètres associés au patron «Ravenscar» qui ne peuvent pas être spécifiés avec AADL version 1. Enfin, la partie dédiée au patron «Queued buffer» décrit des méthodes analytiques développées pour le dimensionnement mémoire d'une architecture temps réel. Finalement, nous concluons avec un bref bilan et quelques perspectives sur l'approche explorée dans ce chapitre.

3.1 AADL : un langage d'architecture adapté aux systèmes temps réel

Un langage d'architecture est un langage qui permet de définir, formellement ou non, un modèle de l'architecture d'un système que l'on cherche à mettre en œuvre [Ver06]. Un langage d'architecture introduit trois concepts : les concepts de composant, de connecteur et de déploiement. Un composant peut être défini comme une unité destinée à être assemblée et à fonctionner avec d'autres [Ker05]. L'architecture logicielle et matérielle d'un système peut être décrite comme une hiérarchie de composants. Les connecteurs permettent de spécifier les interactions entre ces composants. On parle de «déploiement» lorsqu'un modèle de l'architecture logicielle est associé à un modèle de l'architecture matérielle. Cette association est nécessaire pour la vérification du respect des exigences du système à implanter, ou tout simplement, pour générer le logiciel et le matériel requis pour sa mise en œuvre.

Un modèle AADL est un ensemble hiérarchisé de composants qui modélisent des entités logicielles et/ou matérielles d'une architecture. Le standard propose différents types de composants matériels et logiciels : les composants *data*, *thread*, *process* qui sont des composants logiciels ; et les composants

processor, *device* et *bus* qui sont des composants matériels. La mise en œuvre de ces composants est régie par des règles de traduction de ces entités vers des langages de programmation.

Un composant *data* modélise n'importe quelle structure de données de l'architecture : un composant *data* peut être implanté par une classe Java, une structure C ou un enregistrement Ada. Un *thread* AADL modélise un flot de contrôle qui possède la capacité d'exécuter un programme. Ce type de composant peut être implanté par une tâche Ada, un *thread* POSIX ou une tâche VxWorks. Un *thread* AADL peut être périodique, apériodique ou sporadique (cf. chapitre 2). Un *process* représente un espace d'adressage. Un *process* permet de modéliser une application et d'assurer une isolation mémoire entre les différentes applications d'un système. Ces différents composants logiciels doivent être déployés sur une architecture matérielle comportant un ou plusieurs processeurs, des périphériques, des unités de mémoire et des bus. Le composant *processor* permet de modéliser une entité capable d'ordonnancer des *threads*. Les composants *device* permettent de modéliser un actionneur ou un capteur. Les composants *bus* peuvent modéliser un bus mémoire, un bus de terrain ou toute entité de communication entre des composants matériels AADL.

Un modèle AADL est complété par la définition de connexions entre les composants et la définition de propriétés associées à chaque entité du modèle.

Une propriété est définie par un nom, une valeur et un type de donnée. Une propriété mémorise une information en vue de l'implémentation ou de l'analyse du composant pour lequel la propriété est déclarée.

Une connexion explicite une interaction entre des composants. Ces interactions s'effectuent grâce à des points de connexion appelés *ports*. AADL propose plusieurs types de ports modélisant différents types d'interactions. Les *data ports* permettent de connecter des composants qui partagent des données. Ils modélisent un accès concurrentiel à des composants *data*. Les *event ports* sont dédiés au transfert d'exceptions entre composants. Une exception indique l'occurrence d'anomalies dans le fonctionnement du système. Seule la dernière exception reçue dans un *event port* est accessible par un composant. Enfin les *event data ports* sont dédiés à l'échange asynchrone de messages entre composants. Dans le cas d'un *event data port*, tous les messages reçus sont conservés et accessibles jusqu'à leur consommation par un composant. L'ordre de lecture des messages est spécifié par une propriété du modèle. Par défaut, les messages sont accédés selon une politique FIFO[13].

La figure 3.1 est un exemple de modèle AADL. Cette spécification comprend deux ressources partagées modélisées par les composants *data* $r1$ et $r2$. Les ressources partagées sont accédées par deux composants *threads* : $th1$ et $th2$. Ces quatre composants sont encapsulés au sein du processus $proc0$. Enfin, cette architecture logicielle est associée à une architecture matérielle qui est ici constituée d'un processeur : le processeur $cpu0$.

AADL est un langage typé. Ainsi, *task_type* et *shared_resource_type* définissent des types de composants qui peuvent être implantés de différentes manières. Ces différentes mises en œuvre peuvent être employées dans le même modèle AADL. Un type de composants peut définir différentes propriétés qui seront alors communes à toutes ses implantations.

[13] First In, First out.

```
PROCESS a_proc
END a_proc;

PROCESS IMPLEMENTATION a_proc.Impl
   SUBCOMPONENTS
      th1 : THREAD task_type.Impl;
      th2 : THREAD task_type.Impl;
      r1 : DATA shared_resource_type.Impl;
      r2 : DATA shared_resource_type.Impl;
   CONNECTIONS
      DATA ACCESS r1 -> th1.can_access;
      DATA ACCESS r2 -> th2.can_access;
END a_proc.Impl;

DATA shared_resource_type
END shared_resource_type;

DATA IMPLEMENTATION shared_resource_type.Impl
   PROPERTIES
      Concurrency_Control_Protocol =>
         Priority_Ceiling_Protocol;
END shared_resource_type.Impl;

THREAD task_type
   FEATURES
      can_access : REQUIRES DATA ACCESS shared_resource_type.Impl;
END task_type;

THREAD IMPLEMENTATION task_type.Impl
   PROPERTIES
      Source_Text => "task_bodies.adb";
      Source_Stack_Size => 4092 kb;
      Dispatch_Protocol => Periodic;
      Period => 50 ms;
      Compute_Execution_Time => 3 ms;
END task_type.Impl;

PROCESSOR a_cpu
END a_cpu;

PROCESSOR IMPLEMENTATION a_cpu.Impl
   PROPERTIES
      Scheduling_Protocol => Rate_Monotonic_Protocol;
END a_cpu.Impl;

SYSTEM a_system
END a_system;

SYSTEM IMPLEMENTATION a_system.Impl
   SUBCOMPONENTS
      cpu0 : PROCESSOR a_cpu.Impl;
      proc0 : PROCESS a_proc.Impl;
   PROPERTIES
      Actual_Processor_Binding => REFERENCE cpu0 APPLIES TO proc0;
END a_system.Impl;
```

FIG. 3.1: *Un exemple de modèle AADL*

Dans notre exemple, le type *task_type* définit des propriétés qui seront communes aux composants *th*1 et *th*2.

Ce modèle contient des exemples de propriétés sur plusieurs composants :

- La propriété *Source_Text* du composant *task_type.Impl* indique le nom du fichier du code source nécessaire à la mise en œuvre du composant. Ainsi, si les *threads th*1 et *th*2 sont programmés en Ada, le fichier «task_bodies.adb» est supposé contenir le corps des tâches Ada pour *th*1 et *th*2. Ce type de propriété est essentiel pour les outils d'ingénierie logicielle qui doivent pouvoir assembler le code source fourni par l'utilisateur et celui généré à partir du modèle AADL.
- La propriété *Source_Stack_Size* du composant *task_type.Impl* indique la taille de la pile d'exécution pour les *threads th*1 et *th*2. Cette propriété peut être exploitée lors de l'implantation de chaque composant *thread* sur la plate-forme d'exécution. La création d'un flot de contrôle tel qu'un *thread* POSIX 1003.1b ou une tâche VxWorks s'effectue grâce à une primitive. Parfois, ces primitives imposent de spécifier une taille maximale de la pile d'exécution associée au flot de contrôle. C'est notamment le cas pour l'exécutif temps réel VxWorks et sa primitive *taskSpawn*. Cette catégorie de propriété peut également être utile pour l'analyse de l'empreinte mémoire.
- La propriété *Scheduling_Protocol* du composant *a_cpu.Impl* définit comment le processeur doit être partagé entre les différents *threads* que le processeur héberge. En d'autres termes, on indique ici quel ordonnancement temps réel on s'attend à calculer lors de l'exécution de l'application.
- Enfin, le déploiement des composants logiciels sur les composants matériels est spécifié grâce à la propriété *Actual_Processor_Binding* du composant *a_system.Impl* qui associe chaque processus à un processeur.

À partir d'un modèle AADL comme celui de la figure 3.1, il est possible de partiellement générer l'application ou d'en effectuer différentes analyses. Regardons maintenant quelques exemples d'outils offrant des services d'édition, d'analyse ou de génération de code pour des modèles AADL :

- ADELE [Ell07] est un éditeur graphique pour modèles à base de composants. ADELE est implémenté sous la forme d'un ensemble de *plugins* Eclipse s'intégrant dans l'environnement TOPCASED. Les développements actuels sont réalisés dans le cadre du programme européen ITEA SPICES et visent à gérer la saisie graphique de modèles AADL sous forme de bibliothèques de composants ou d'assemblages dans un système opérationnel. Le méta-modèle sous-jacent est structuré en couches pour permettre une meilleure flexibilité dans les évolutions ou variantes possibles de l'éditeur. La première couche, appelée «noyau», définit un modèle de composant élémentaire. La couche intermédiaire permet d'apporter la sémantique spécifique à chaque catégorie de composants et de supporter les différentiations graphiques correspondantes. Enfin, la dernière couche décrit le détail des propriétés attendues pour chacune des entités manipulées dans le modèle. ADELE est développé par Ellidiss Technologies.
- Stood [Dis04] est un outil de conception logiciel qui offre un support pour différents langages de conception tels que UML et AADL. L'outil est aussi compatible avec la méthode de conception HOOD [BW94]. Il est particulièrement bien adapté à la conception d'application temps réel grâce à son méta-modèle qui contient les propriétés nécessaires à ce type d'applications.

Il propose de nombreux générateurs de code implantés en Prolog qui peuvent être aisément adaptés aux besoins des utilisateurs. Stood est largement représenté dans les sociétés du domaine de l'avionique, le spatial ou l'armement (Airbus, CNES, ESA, Eurocopter, Alcatel, ...). À partir d'un modèle graphique AADL, Stood génère la représentation textuelle du modèle AADL. Cette représentation textuelle peut être chargée dans Cheddar grâce à l'outil Ocarina de Télécom Paris-Tech [VPK05, HZPK07]. Stood est développé par Ellidiss Technologies.

- L'usage du langage AADL a été exploré dans le cadre du projet européen IST ASSERT. L'objectif du programme ASSERT était de définir et d'expérimenter un processus de construction d'applications embarquées temps réel critiques basé sur des preuves. Dans le cadre de ce programme de recherche, Télécom Paris-Tech a conduit plusieurs expérimentations avec leur outil Ocarina. Ocarina est une bibliothèque qui offre des services pour la manipulation des modèles AADL. Cet outil propose des services pour lire, analyser et contrôler la correction sémantique d'un modèle AADL. Ocarina propose des générateurs de code Ada et C pour des applications réparties éventuellement critiques. Cette bibliothèque peut finalement être employée par des applications tierces. C'est notamment le cas de Cheddar qui exploite cet outil afin de traduire un modèle AADL vers une instance de son méta-modèle propriétaire. Par ailleurs, Télécom Paris-Tech a montré qu'il était possible d'appliquer un processus permettant, à partir d'un modèle AADL conforme au profil «Ravenscar», de générer l'exécutable avec PolyOrb et d'effectuer l'analyse du modèle avec CPN-AMI [HHK+06] et Cheddar [HZPK08].
- OSATE[14] développé par le Software Engineering Institute est un ensemble de *plugins* Eclipse [SEI04]. OSATE propose des fonctionnalités permettant d'éditer un modèle AADL graphiquement ou textuellement et de le traduire sous différents formats (exemple : XML). OSATE constitue l'outil de référence du standard AADL pour la communauté. Le projet TOPCASED [FGC+06] propose aussi des outils d'édition AADL. Par ailleurs, OSATE et TOPCASED sont interopérables et partagent les mêmes *plugins* tels que ceux distribués par la société Fremont Associates [SLC06]. Ces *plugins* offrent des outils d'analyse d'ordonnancement temps réel basés sur l'algèbre de processus temporisée [CLX95].
- Finalement, Axlog a développé ADes [Til05]. ADes est un simulateur AADL qui est capable de produire une trace d'exécution d'un modèle AADL afin d'en étudier le comportement. ADes a été initialement développé dans le cadre d'une étude conjointe entre Axlog et l'ESA[15]. Cette étude consistait à évaluer AADL et son utilisation dans l'ingénierie de systèmes spatiaux.

Le tableau 3.1 présente ces différents outils avec leurs fonctionnalités principales.

3.2 Patrons de conception : AADL comme langage pivot

Pour automatiser l'analyse des performances d'un modèle d'architecture, il est nécessaire que le concepteur soit aidé dans son choix des méthodes analytiques à appliquer. Pour ce faire, il faut offrir

[14] Open Source AADL Tool Environment.
[15] European Space Agency.

Outils	Edition	Analyse	Génération de code
ADELE	X		
Stood	X		X
Ocarina		X	X
OSATE	X	X	
ADes		X	
Cheddar		X	

TAB. 3.1: *Exemples d'outils AADL*

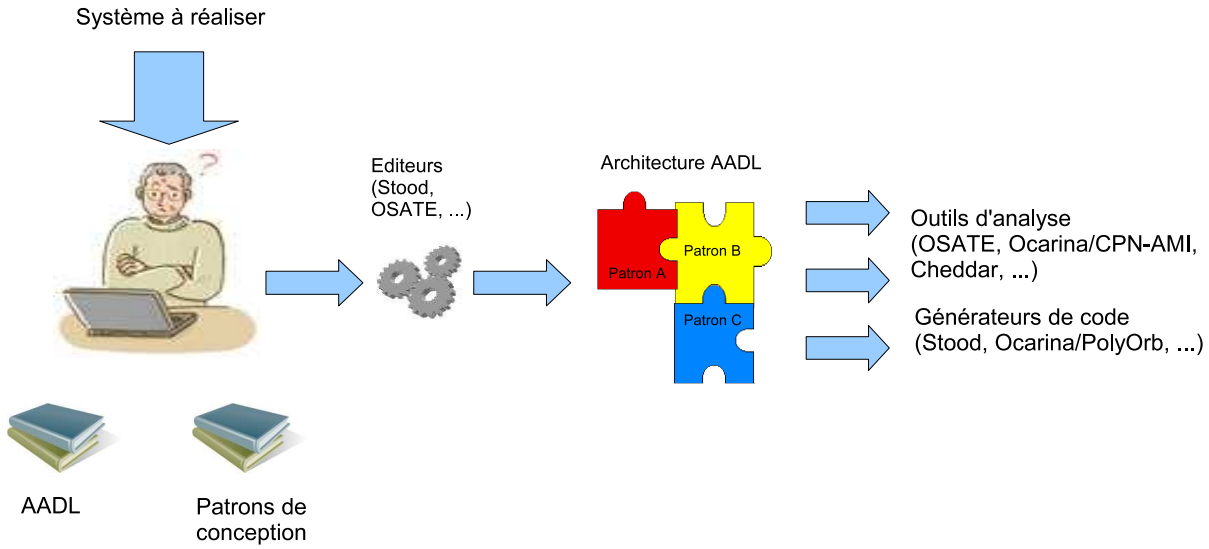

FIG. 3.2: *Approche par patron de conception*

des moyens d'interopérabilité entre les différents outils d'ingénierie logicielle. Une approche possible consiste à employer un langage pivot pour l'échange de données entre les outils. Nous avons choisi de considérer AADL comme un langage pivot pour exprimer finement la sémantique d'un ensemble de modèles d'architecture de sorte qu'ils puissent être interprétés sans ambiguïté par différents outils. Deux outils AADL sont considérés dans cette étude : l'outil de modélisation Stood et l'outil de vérification Cheddar.

La figure 3.2 présente l'approche considérée. Elle s'inspire de celle élaborée pour la méthode HRT-HOOD [BW94]. Elle consiste à définir des patrons de conception. Un patron de conception propose une solution à une famille de problèmes de concurrence : un patron décrit une solution architecturale avec AADL. La modélisation de chaque patron est complétée par une spécification rédigée avec l'annexe comportementale d'AADL. Chaque patron de conception est associé à diverses hypothèses concernant l'environnement d'exécution (ordonnanceurs utilisés, propriétés des mécanismes de synchronisation, paramètres des tâches, ...). Ces hypothèses garantissent qu'il est possible d'appliquer un ensemble de

tests de faisabilité donné sur le patron de conception. Ils permettent de vérifier les performances d'un modèle conforme au patron de conception. L'analyse de performances est simplifiée car le concepteur n'a pas à vérifier que son modèle respecte les hypothèses des tests de faisabilité. En effet, la modélisation d'une architecture temps réel avec ces patrons de conception est un moyen qui garantit au concepteur que son architecture pourra être vérifiée. Un modèle édité grâce à Stood ou OSATE peut alors être analysé automatiquement par Cheddar, ou être exploité par un générateur de code tel qu'Ocarina.

Nous avons exprimé quatre patrons de conception avec AADL : «Synchronous data-flow», «Ravenscar», «Blackboard» et «Queued Buffer». Chaque patron de conception est illustré par une étude de cas décrite dans [DS08]. Ces patrons décrivent quatre paradigmes classiques de communication et/ou de synchronisation entre des flots de contrôle tels que des *threads* AADL. Il existe bien sûr de nombreux autres paradigmes possibles de synchronisation qui puissent faire l'objet d'une telle spécification. Néanmoins, dans un premier temps, nous estimons que ces quatre patrons sont représentatifs du domaine applicatif ciblé (applications avioniques et spatiales), mais aussi suffisant pour valider l'usage d'AADL comme un langage pivot possible entre Stood et Cheddar. En effet, ces patrons de conception :

1. Correspondent à un usage fréquent dans les domaines applicatifs ciblés ; c'est à dire les domaines actuels d'utilisation de Stood. Certains de ces patrons ont d'ailleurs été initialement proposés par Ellidiss Technologies et par l'IRIT [DBF+06].
2. Emploient des concepts et fonctionnalités issues des standards du domaine applicatif [Ari97].

Pour chaque patron, nous déterminons alors quels sont les critères de performance qu'il est possible de calculer automatiquement. Ces critères sont calculés grâce à des méthodes analytiques issues, soit de la théorie de l'ordonnancement temps réel, soit de la théorie des files d'attente. Un échantillon de ces critères est énuméré ci-dessous :

A) Le pire temps de réponse d'un *thread*.

B) La borne maximale sur le temps d'attente d'un *thread* pour l'accès à une ressource partagée.

C) La détection d'inversion de priorité et d'interblocage.

D) Le nombre moyen et maximum de messages dans un *event data port*. Ces critères permettent de calculer le temps d'attente moyen et maximum d'un message dans un *event data port*. Le calcul de ce délai est nécessaire si l'on souhaite calculer des temps de réponse de bout en bout selon la méthode holistique [TC94] ou selon la méthode des offsets [Tin94, GH98, PH03].

E) Le nombre de commutations de contexte, de préemptions.

F) Le taux d'utilisation de certaines ressources telles que le processeur.

G) ...

Le tableau 3.2 résume les différents critères que nous cherchons à établir pour chacun des patrons de conception. Dans ce mémoire, nous nous limitons aux critères A, B, C et D. Ces critères ne sont pas toujours évaluables indépendamment. Ainsi pour le patron «Ravenscar», le critère B doit être évalué avant le calcul du critère A.

Patrons de conception	Critères
Synchronous data-flows	A
Ravenscar	A, B
Blackboard	A, B et C
Queued buffer	A, B, C et D

TAB. 3.2: *Critères de performance par patron de conception*

Dans la suite de ce chapitre, nous décrivons brièvement ces quatre patrons. Une description plus détaillée est proposée dans les parties 3.3 et 3.4 pour les patrons «Ravenscar» et «Queued buffer».

3.2.1 Le patron de conception «Synchronous data-flows»

Ce premier patron est celui employé par l'environnement d'exécution de Meta-H [SAE04].

Meta-H est à la base de la sémantique d'exécution du standard AADL. Avec ce patron de conception, la communication inter-thread est assurée par une synchronisation temporelle entre les *threads*. En effet, lorsque deux *threads* périodiques a et b doivent partager une donnée, l'exécution de a et de b sont planifiées statiquement afin de garantir un accès séquentiel à la donnée sans requérir l'usage d'un outil de synchronisation tel qu'un sémaphore. L'environnement d'exécution pour une application de ce type est donc rudimentaire. Si l'ordonnancement est calculé hors-ligne, il ne contient pas nécessairement d'ordonnanceur. Dans le cas contraire, un ordonnanceur à priorité fixe peut être suffisant.

Puisque le patron de conception impose un ordonnancement statique et déterministe, soit hors-ligne, soit à l'aide d'un algorithme à priorité fixe, cela implique une analyse de performances de l'architecture très simplifiée. Les interactions sur les ressources partagées peuvent être ignorées lors de la vérification. Les *threads* sont donc ordonnancés de façon indépendante.

La vérification de ce patron peut être effectuée par l'application des tests les plus simples sur le pire temps de réponse ou sur le taux d'utilisation du processeur. Seul le respect des échéances de chaque *thread* doit être vérifié (calcul du critère A). Par exemple, si l'on suppose que l'ordonnanceur est préemptif, que les tâches sont périodiques à échéances sur requêtes et à départs simultanés, il est possible d'employer les tests de faisabilité suivants :

- Si un ordonnancement hors-ligne est calculé avec EDF, le test C2 de l'annexe A (cf. page 125) peut être appliqué. Le test C2 appliqué au patron «Synchronous data-flows» permet d'obtenir une condition nécessaire et suffisante de validité des échéances des *threads*.
- Dans le cas d'un ordonnancement à priorité fixe en ligne ou hors-ligne tel que Rate Monotonic, les tests C1 (cf. page 124) ou R1 (cf. page 129) peuvent être appliqués. Le test R1 est une condition nécessaire et suffisante de validité des échéances des *threads*. Selon le jeu de *threads* considéré, le test C1 peut être une condition nécessaire et suffisante ou une condition suffisante seulement.

3.2.2 Le patron de conception «Ravenscar»

Si le patron de conception précédent dispose de méthodes d'analyse simples, précises et applicables à des architectures de grande taille, l'inconvénient principal de cette approche est l'absence de flexibilité sur l'ordonnancement des *threads*. Afin d'augmenter la flexibilité de l'ordonnancement et la possibilité d'asynchronisme lors de communications inter-thread, un moyen consiste à s'inspirer du profil Ravenscar. Le profil Ravenscar fait partie du standard international Ada 2005 [BW07, ISO07, TDB+06]. Ravenscar est un sous ensemble du langage Ada 2005, et en particulier de ses fonctionnalités ayant trait à la concurrence. Ce profil est un ensemble de restrictions vérifiées lors de la compilation d'une application Ada. Il permet de garantir qu'à l'exécution, l'application sera analysable par des méthodes analytiques. Grâce au profil, l'application Ada implémentée sera bien celle qui aura été vérifiée en phase amont du cycle de vie de l'architecture logicielle.

Ravenscar suppose que les *threads* sont ordonnancées selon un algorithme à priorité fixe. L'affectation des priorités peut être réalisée selon Rate Monotonic ou tout autre algorithme. Les *threads* peuvent partager différentes ressources grâce à des sémaphores. Ravenscar requiert l'usage du protocole ICPP pour l'usage des sémaphores.

Afin de faciliter l'analyse de performances, nous rajoutons, aux contraintes spécifiées par le standard «Ravenscar», des restrictions supplémentaires concernant l'usage des outils de synchronisation. Les *threads* partagent des données grâce à des composants *data*. On suppose que le seul outil de synchronisation autorisé pour la protection de ces composants *data* est la section critique. Tout autre modèle de synchronisation basé sur les sémaphores est interdit. D'autre part, si plusieurs *threads* utilisent plusieurs ressources communes, nous supposons que l'ordre d'accès à ces composants *data* est identique pour tous les *threads*. On suppose donc que les accès à un ensemble de composants *data* soit correctement imbriqués.

Par l'usage du protocole ICPP, il n'est pas nécessaire de vérifier l'absence d'inversion de priorités [BW07]. Par l'usage exclusif de sections critiques qui soient correctement imbriquées, il n'est pas nécessaire de vérifier l'absence d'interblocage [Tan01].

La vérification d'une architecture logicielle conforme à notre patron «Ravenscar» consiste uniquement à s'assurer que les échéances des *threads* sont respectées. La partie 3.3 énumère les tests de faisabilité applicables ici.

La complexité de la vérification du patron «Ravenscar» reste donc raisonnable puisque seuls les critères A et B doivent être vérifiés. L'évaluation du critère B est nécessaire si l'on souhaite calculer le pire temps de réponse (critère A). L'usage de sections critiques uniquement, peut faciliter l'évaluation de ces délais d'attente.

La précision des méthodes analytiques pour ce patron est toutefois inférieure à celle obtenue pour le patron «Synchronous data-flows». En effet, les délais d'attente sur les ressources partagées sont généralement évalués par une borne maximale. Cette borne maximale est calculée grâce à la durée de chaque section critique. Puisque le temps d'attente est évalué par une borne maximale, il en découle que les pires temps de réponse constituent des conditions suffisantes mais non nécessaires.

3.2.3 Le patron de conception «Blackboard»

Dans le patron de conception précédent, la vérification du modèle reste simple grâce aux restrictions sur l'accès aux composants *data*. Mais en pratique, il existe de nombreux paradigmes de synchronisation qui peuvent être construits avec des sémaphores. Ainsi, des systèmes d'exploitation ou des langages de programmation proposent des mécanismes de synchronisation tels que les barrières, les lecteurs/rédacteurs, différentes formes de producteurs/consommateurs, les rendez-vous, les sémaphores privés ou encore les sémaphores collectifs dont les méthodes synchronisées de Java en sont un exemple d'implémentation [Tan01, BW07, Kra85, ZHR+01].

Chaque patron de synchronisation nécessite d'évaluer le plus finement possible les délais d'attente sur les sémaphores employés, mais aussi de s'assurer de l'absence d'interblocage et d'inversion de priorités. Il serait long et fastidieux d'exprimer chacun de ces mécanismes de synchronisation avec AADL et son annexe comportementale. Nous avons donc choisi de nous concentrer sur le mécanisme des lecteurs/rédacteurs. Le patron de conception «Blackboard» modélise ce mécanisme de lecteurs/rédacteurs. Il s'agit d'un mécanisme classique de communication inter-thread que l'on trouve dans certaines normes internationales pour l'avionique telles qu'ARINC 653 [Ari97]. À tout instant, on garantit que la ressource partagée ne peut pas être écrite par plus d'un *thread*. Plusieurs lecteurs de la ressource partagée peuvent néanmoins accéder simultanément à la ressource, à condition qu'aucun rédacteur ne l'utilise. Pour chaque paradigme de synchronisation tel que le patron «BlackBoard», il est nécessaire de vérifier les critères A, B et C du tableau 3.2 page 50.

3.2.4 Le patron de conception «Queued Buffer»

Dans le cas du patron «Blackboard», à tout instant, seule la dernière information écrite est disponible aux lecteurs. Certains environnements d'exécution tels qu'ARINC 653 ou VxWorks proposent des services de communication inter-thread où plusieurs messages sont conservés et mémorisés jusqu'à leur consommation.

AADL propose aussi cette fonctionnalité par le biais des *event data ports*. Le patron «Queued Buffer» propose de modéliser et d'analyser ce type de fonctionnalité. On suppose pour ce patron que les messages d'un port sont mémorisés dans un tampon et consommés selon la politique FIFO : le premier message inséré dans le port/tampon et celui qui sera le premier consommé.

Comme les trois patrons précédents, le patron de conception «Queued Buffer» nécessite de vérifier les échéances des *threads* qui produisent ou consomment des messages (critère A). Selon que l'on considère indépendant ou non les producteurs et les consommateurs, le temps de réponse des *threads* nécessite ou non l'application du calcul holistique [TC94].

Chaque tampon étant une ressource partagée protégée par un ou plusieurs sémaphores, il est aussi nécessaire d'évaluer l'impact des délais d'attente dus aux sémaphores utilisés pour la mise en œuvre de ce tampon (critère B). L'absence d'interblocage et d'inversion de priorité sur ces sémaphores doit aussi être assurée (critère C).

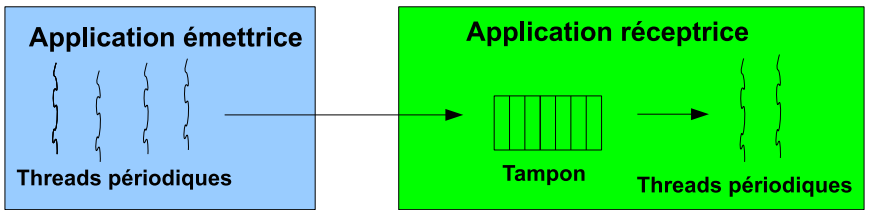

FIG. 3.3: *Deux applications qui s'échangent des messages*

Enfin, l'utilisation de ce patron de conception implique la vérification d'une nouvelle catégorie de propriété : le dimensionnement du tampon vis-à-vis des lois de production et de consommation des messages (critère D). Ce patron de conception illustre l'un des points forts d'AADL en terme d'analyse de performances : AADL offre un moyen de modéliser différentes ressources de l'architecture. Il est alors possible de réaliser des analyses de performances simultanément sur plusieurs ressources de l'architecture, afin, par exemple, d'explorer différents compromis.

La figure 3.3 illustre un compromis classique concernant l'utilisation de la mémoire et de la ressource processeur. La figure montre un exemple de système composé de deux applications qui s'échangent des messages asynchrones via un mécanisme classique de communication inter-thread tel que les IPC Unix [J.M03]. Avec ce type d'architecture, il est souvent souhaitable d'analyser les besoins en processeurs et en mémoire de façon conjointe car :

- Si les *threads* périodiques qui émettent ou reçoivent des données ont un niveau de priorité élevé, cela implique que leur pire temps de réponse est court et qu'il varie peu. Dans ce cas de figure, les besoins en mémoire pour le tampon sont peu élevés car les deux ensembles de *threads* sont fortement synchronisés.
- Dans le cas contraire, c'est à dire lorsque les *threads* périodiques qui émettent ou reçoivent des données ont un niveau de priorité bas, leur pire temps de réponse est grand et le temps de réponse d'un *thread* peut fortement varier. Les besoins en mémoire pour le tampon seront alors plus élevés car les deux ensembles de *threads* sont bien plus faiblement synchronisés.

Par la modélisation de toutes les entités du système, AADL autorise la réalisation d'analyses de performances qui peuvent prendre en considération ce type de compromis entre occupation du processeur et empreinte mémoire. Ainsi dans sa thèse [Leg04], Jérôme Legrand a proposé des tests de faisabilité basés sur la théorie des files d'attente. Ces tests ont été adaptés à des modèles AADL composés d'*event data ports* [SLNM05] et permettent de calculer le critère D du tableau 3.2. La partie 3.4 explique en détail comment ces analyses peuvent être conduites.

3.3 Patron de conception «Ravenscar» : analyse élémentaire de l'ordonnançabilité

Dans cette partie, nous revenons sur le patron «Ravenscar». Nous regardons comment conduire une analyse d'ordonnancement sur un modèle AADL qui soit conforme à ce patron.

Le profil Ravenscar, issue du standard Ada 2005, impose un ensemble de restrictions qui assurent qu'une architecture puisse être vérifiée par des méthodes analytiques. Ces restrictions lors de la compilation sont explicitées grâce à des directives de compilation ou *pragma*. Certaines de ces restrictions sont spécifiques au langage Ada et d'autres portent sur son environnement d'exécution, c'est à dire le fonctionnement du *runtime* Ada et du système d'exploitation sous-jacent. Il est possible d'extraire du standard les restrictions qui sont pertinentes du point de vue de l'analyse de performances :

1. Utilisation d'un ordonnancement à priorité fixe préemptif associé à une politique FIFO. Avec Ada 2005, cette restriction est imposée par le *pragma* $Task\_Dispatching\_Policy(FIFO\_Within\_Priorities)$ ainsi que par la restriction $No\_Dynamic\_Priorities$. Le modèle d'ordonnancement d'Ada 2005 est identique à celui proposé par le standard POSIX 1003.1b. On garantit à tout moment que la tâche de plus forte priorité soit exécutée. Une file d'attente est associée à chaque niveau de priorité et une politique de choix est appliquée lorsque plusieurs tâches de même niveau de priorité sont prêtes. Le calcul de l'ordonnancement consiste donc à élire une tâche parmi la file d'attente de plus forte priorité qui contient une à plusieurs tâches prêtes. La politique la plus classique, la politique FIFO, consiste à élire la tâche prête en tête de la file d'attente. Nous décrivons plus en détail ce modèle d'ordonnancement dans la partie 4.1.1.
2. Utilisation du protocole ICPP afin de borner les temps d'attente sur les ressources partagées. Cette restriction est imposée par le *pragma* $Locking\_Policy(Ceiling\_Locking)$.
3. Départs simultanés de toutes les tâches (ce qui est imposé par les *pragmas* $No\_Task\_Hierarchy$ et $No\_Task\_Allocators$). Cette restriction garantit l'occurrence de l'instant critique.
4. Nombre maximal de tâches connu à la conception/compilation (restriction Max_Task).
5. Interdiction d'utiliser des instructions ou services qui impliquent un non déterminisme temporel. Ainsi, l'allocation dynamique de mémoire est interdite (restrictions $No\_Task\_Allocators$, $No\_Protected\_Type\_Allocators$ ou $No\_Implicit\_Heap\_Allocation$). De même, les opérations de synchronisation trop complexes à analyser sont aussi interdites (*pragmas* $No\_Requeue\_Statements$ ou $No\_Select\_Statement$).

À ces contraintes du standard Ada 2005, nous rajoutons deux contraintes qu'une architecture AADL doit respecter afin d'en simplifier l'analyse de performances :

1. Contraintes concernant l'usage des sémaphores : on suppose que les sémaphores ne peuvent être employés que pour la construction de sections critiques sur des composants *data* par exemple. On suppose, en outre, que les accès à ces sections critiques sont correctement imbriqués. Si plusieurs *threads* utilisent plusieurs ressources communes, l'ordre d'accès à ces composants *data* est identique pour tous les *threads*.
2. Contraintes concernant l'affectation des priorités : on suppose qu'un niveau de priorité différent est affecté à chaque tâche.

Grâce à ces différentes contraintes qui définissent le patron «Ravenscar» dans le contexte du projet Cheddar, il est possible d'analyser une architecture à l'aide des tests R2 et P2 (cf. pages 129 et 132).

Bien qu'issue de la communauté Ada, l'utilisation du patron «Ravenscar» n'est pas limitée aux applications écrites en Ada. Il est tout à fait possible d'employer ce patron pour une application écrite en langage C avec l'interface POSIX 1003.1b [Gal95]. De même, une adaptation du profil Ravenscar existe aussi pour Java [KWK02]. En fait, de nombreux environnements d'exécution sont potentiellement compatibles avec ces contraintes car la quasi-majorité des exécutifs temps réel proposent un ordonnancement à priorité fixe associé à une variante du protocole PCP [SRL90].

Enfin, des méthodes d'analyse équivalentes existent aussi pour l'algorithme EDF en mode préemptif, à condition que l'accès aux ressources soit effectué par les protocoles PLCP ou SRP [ISO07, TDB+06, Bar06] (cf. tests R3 et P3, pages 130 et 132). Il est donc envisageable de concevoir un patron de conception inspiré de «Ravenscar», mais adapté à cet algorithme à priorité dynamique.

La première version du standard AADL proposait des propriétés permettant l'application des méthodes d'analyse les plus simples de la théorie de l'ordonnancement temps réel. En particulier, toutes les propriétés nécessaires pour l'analyse du patron «Synchronous data-flows» sont définies dans la version 1 de ce standard. Il n'en est pas de même pour le patron «Ravenscar».

En effet, certaines informations essentielles pour appliquer les techniques d'analyse référencées dans l'annexe A manquent à cette version. AADL propose un moyen pour étendre les propriétés prédéfinies par le standard. Tout utilisateur peut définir un ensemble spécifique de propriétés adaptées à une méthode, un outil spécifique de vérification ou de génération de code. Grâce à ce mécanisme d'extension, nous avons donc proposé un nouvel ensemble de propriétés nécessaires au patron «Ravenscar» [Sin07b].

Cet ensemble contient principalement :

- Des propriétés associées aux ordonnanceurs temps réel. On précise ici la valeur du quantum (propriété $Scheduler\_Quantum$), si l'ordonnanceur est préemptif ou non (propriété $Preemptive\_Scheduler$), les politiques d'ordonnancement POSIX 1003.1b disponibles (propriété $POSIX\_Scheduling\_Policy$), ...
- Des propriétés associées aux *threads*. On précise ici la valeur de la priorité fixe (propriété $Fixed\_Priority$), le retard sur la date de réveil (propriété $Dispatch\_Jitter$), une valeur pour un éventuel $offset$, la date du premier réveil (propriété $Dispatch\_Absolute\_Time$), le temps de blocage sur les ressources partagées (propriété $Bound\_On\_Data\_Blocking\_Time$), ...
- Des propriétés pour définir précisément quand les *threads* accèdent aux ressources partagées (propriété $Critical\_Section$). En effet, la version 1 d'AADL ne permet pas d'exprimer aisément ces informations.

Cet ensemble de propriétés propose aussi des propriétés qui ne sont pas associées au patron «Ravenscar» telles que :

- Des propriétés pour exprimer certaines contraintes de précédence qui ne pouvaient pas être déduites depuis une spécification AADL pour des raisons d'ambiguïté de la norme (propriété $Thread\_Precedence$).
- Des propriétés permettant de décrire comment un modèle AADL doit être simulé, vis-à-vis des générateurs de nombres aléatoires par exemple (propriétés $Dispatch\_Seed\_Value$ ou $Dispatch\_Seed\_is\_Predictable$).
- Enfin, certaines propriétés AADL ont été introduites afin de faciliter l'interaction du langage

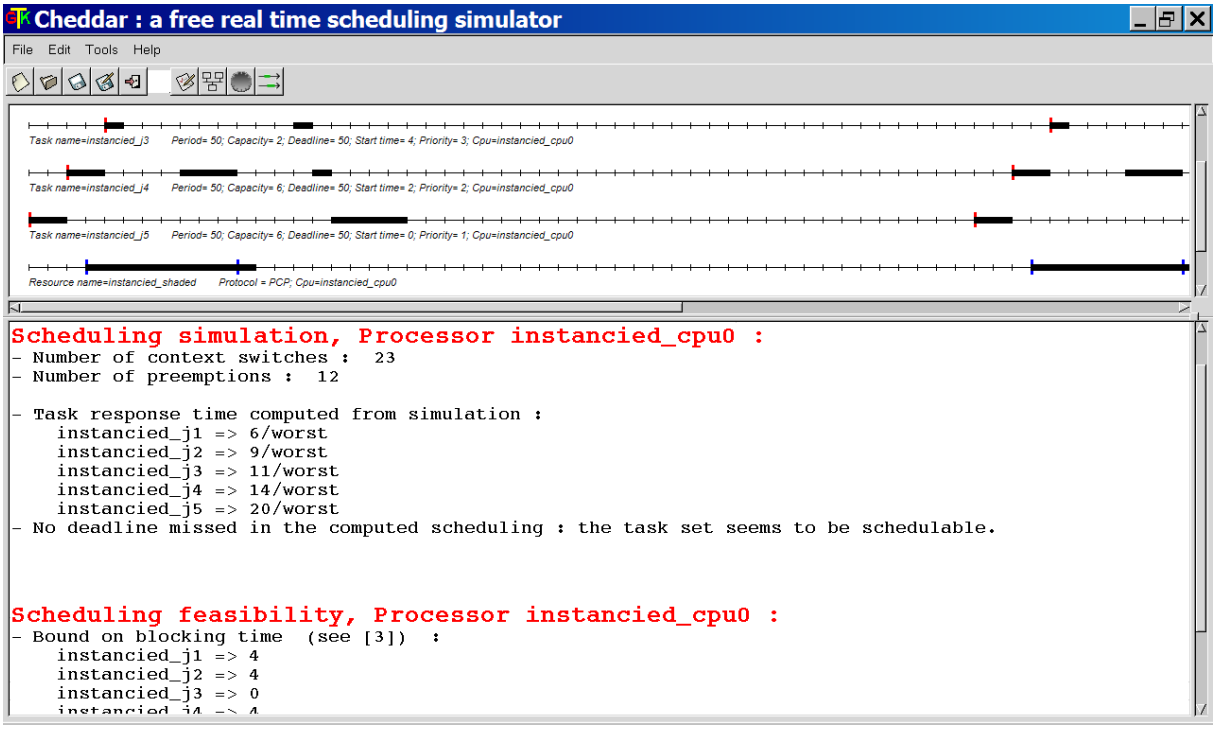

FIG. 3.4: *Analyse d'ordonnançabilité du patron «Ravenscar»*

spécifique de Cheddar avec des modèles AADL (propriétés *Automaton_Name* ou *Source_Text*). Le langage spécifique de Cheddar est décrit dans la chapitre 4.

La figure 3.5 présente un exemple de modèle AADL conforme au patron «Ravenscar» qui fait usage des propriétés définies pour Cheddar.

La figure 3.4 est une analyse d'ordonnancement effectuée par l'environnement Cheddar pour ce modèle. La fenêtre est divisée en deux parties. La partie haute de la fenêtre présente un ensemble de chronogrammes figurant l'ordonnancement des *threads* qui sera effectué lors de l'exécution du système. D'autres évènements peuvent être présentés par des chronogrammes (échanges de messages, accès aux ressources partagées, ...). La partie basse de la fenêtre affiche les différents critères de performance. Ces critères peuvent être calculés, soit à partir des chronogrammes produits par simulation, soit grâce aux tests de faisabilité de l'annexe A. La figure 3.4 affiche :

- Les temps de réponse des *threads* calculés à partir de simulations.
- Les délais maximaux d'attente pour l'accès aux composants *data*, calculés à partir des tests de faisabilité. La traçabilité des tests de faisabilité appliqués sur le modèle est assurée en affichant à l'utilisateur la publication décrivant le test de faisabilité appliqué.

```
PROCESSOR IMPLEMENTATION cpu0.Impl
   PROPERTIES
      Scheduling_Protocol => Posix_1003_Highest_Priority_First_Protocol;
      Cheddar_Properties::Preemptive_Scheduler => True;
      Cheddar_Properties::Scheduler_Quantum => 2 ms;
END cpu0.Impl;

SYSTEM IMPLEMENTATION Ravenscar.Impl
   SUBCOMPONENTS
      instancied_cpu0 : PROCESSOR cpu0.Impl;
      instancied_ea1 : PROCESS ea1.Impl;
   PROPERTIES
      Actual_Processor_Binding => REFERENCE instancied_cpu0 APPLIES TO instancied_ea1;
END Ravenscar.Impl;

DATA IMPLEMENTATION shaded.Impl
   PROPERTIES
      Cheddar_Properties::Data_Concurrency_State => 1;
      Concurrency_Control_Protocol => Priority_Ceiling_Protocol;
END shaded.Impl;

DATA IMPLEMENTATION black.Impl ...

THREAD J1
   FEATURES
      shaded_features : REQUIRES DATA ACCESS shaded.Impl;
END J1;

THREAD IMPLEMENTATION J1.Impl
   PROPERTIES
      Dispatch_Protocol => Periodic;
      Compute_Execution_Time => 0 ms .. 3 ms;
      Deadline => 50 ms;
      Period => 50 ms;
      Cheddar_Properties::Dispatch_Absolute_Time => 7 ms;
      Cheddar_Properties::POSIX_Scheduling_Policy => SCHED_FIFO;
      Cheddar_Properties::Bound_On_Data_Blocking_Time => 0 ms;
      Cheddar_Properties::Fixed_Priority => 5;
      Cheddar_Properties::Dispatch_Jitter => 0 ms;
END J1.Impl;

THREAD IMPLEMENTATION J2 ...
THREAD IMPLEMENTATION J3 ...

PROCESS IMPLEMENTATION ea1.Impl
   SUBCOMPONENTS
      instancied_J1 : THREAD J1.Impl;
      instancied_J2 : THREAD J2.Impl;
      instancied_J4 : THREAD J4.Impl;
      ...
      instancied_shaded : DATA shaded.Impl;
      instancied_black : DATA black.Impl;
   CONNECTIONS
      DATA ACCESS instancied_shaded -> instancied_J1.shaded_features;
      DATA ACCESS instancied_black -> instancied_J2.black_features;
      DATA ACCESS instancied_shaded -> instancied_J4.shaded_features;
      ...
   PROPERTIES
      Cheddar_Properties::Critical_Section => (
      "instancied_shaded",
      "instancied_J1","2","2",
      "instancied_shaded",
      "instancied_J4","2","5",
      "instancied_black",
      ... );
END ea1.Impl;
```

FIG. 3.5: *Patron «Ravenscar» exprimé avec AADL*

3.4 Patron de conception «Queued Buffer» : analyse de l'empreinte mémoire d'un modèle AADL

Nous revenons maintenant sur le patron employant le mécanisme de communication inter-thread basé sur les *event data ports*. Les *event data ports* offrent un service de communication qui permet à des *threads* d'échanger des messages. On suppose pour ce patron que les messages d'un port sont consommés selon la politique FIFO. Pour chaque *event data port*, il existe donc (au moins) un tampon qui mémorise les messages émis en attente de leur consommation.

Comme les trois patrons précédents, le patron de conception «Queued Buffer» nécessite de vérifier les échéances des *threads* qui produisent ou consomment des messages (critère A). Par ailleurs, puisque chaque tampon est une ressource partagée protégée par un ou plusieurs sémaphores, il est aussi nécessaire d'évaluer l'impact des délais d'attente induits par les sémaphores (critère B), ainsi que l'absence d'interblocage et d'inversion de priorité (critère C). Enfin, dans cette partie nous focalisons notre attention sur un critère supplémentaire. En effet, il est maintenant nécessaire de vérifier que le dimensionnement du tampon est correct vis-à-vis des lois de production et de consommation des messages (critère D). Pour ce faire, nous employons la théorie des files d'attente [Kle75b, Kle75a, Rob90].

Dans ce patron, on considère que les *threads* sont périodiques, apériodiques ou sporadiques. Nous n'émettons aucune hypothèse sur l'ordonnanceur temps réel employé. Toutefois, on suppose que tous les délais critiques sont respectés et que donc les critères A et B ont été préliminairement vérifiés. On suppose, en outre, qu'un seul message peut être consommé ou produit à chaque réveil d'un *thread* périodique ou sporadique.

Enfin, bien qu'une production et une consommation de message suggère l'existence d'une contrainte de précédence entre producteur et consommateur, on suppose que les *threads* sont ordonnancés sans tenir compte de cette contrainte. Ceci modélise, par exemple, un mécanisme de *polling* où une tâche périodique est réveillée pour traiter une opération d'entrée/sortie. Cela implique qu'un *thread* périodique consommateur peut être réveillé sans message à consommer. Dans ce cas, on suppose qu'il suspend son exécution jusqu'à son prochain réveil.

Il existe en fait très peu de tests de faisabilité capables de tenir compte de contraintes de précédence. Par ailleurs, ces tests fournissent des résultats très pessimistes. Nous explorons une voie différente qui consiste à privilégier les jeux de tâches pour lesquels des tests de faisabilité simples et efficaces existent. La difficulté est alors reportée sur le dimensionnement des tampons car il n'est plus possible d'appliquer directement les résultats classiques de la théorie des files d'attente.

La théorie des files d'attente permet d'analyser les performances de systèmes composés de serveurs et de clients qui sont soumis à des phénomènes d'attente. Par la définition du rythme d'arrivées des clients et de la capacité de traitement des serveurs, la théorie des files d'attente permet de calculer différents critères de performance. Ces critères peuvent être employés pour dimensionner un tampon. Malheureusement, les modèles classiques de cette théorie supposent implicitement qu'une contrainte de précédence existe entre l'arrivée d'un client et le réveil du serveur. Nous proposons donc une nouvelle famille de files d'attente qui ont pour objectif de modéliser le comportement d'un serveur périodique

ordonnancé par un algorithme tel que Rate Monotonic ou Earliest Deadline First. Puis, nous proposons d'étudier les tampons d'un port AADL selon l'approche suivante :

- Lorsque la durée minimum entre deux arrivées de messages dans l'*event data port* est connue, des critères maximums de performance permettent d'obtenir une borne maximale sur la taille du tampon associé au port. Dans ce cas, les *threads* qui produisent des messages dans le port sont périodiques et l'analyse de performance est conduite à l'aide d'un modèle de file d'attente nommé P/P/1.
- Lorsque seule la durée moyenne entre deux arrivées de messages est connue, des critères moyens de performance permettent d'évaluer les caractéristiques du tampon de manière à diminuer le risque de débordement. Dans ce second cas, les *threads* qui produisent des messages dans le port peuvent être apériodiques ou sporadiques et l'analyse de performance est conduite à l'aide d'un modèle de file d'attente nommé M/P/1.

Après un bref rappel sur la théorie des files d'attente, nous exposons les caractéristiques de la loi de service P. Cette loi modélise le comportement d'un serveur de file d'attente lorsqu'il s'agit d'un *thread* périodique ordonnancé par un algorithme temps réel. Puis, nous étudions les files d'attente M/P/1 et P/P/1. Enfin, nous illustrons leur emploi possible avec un cas d'école constitué d'un modèle AADL élémentaire.

3.4.1 Rappels sur la théorie des files d'attente

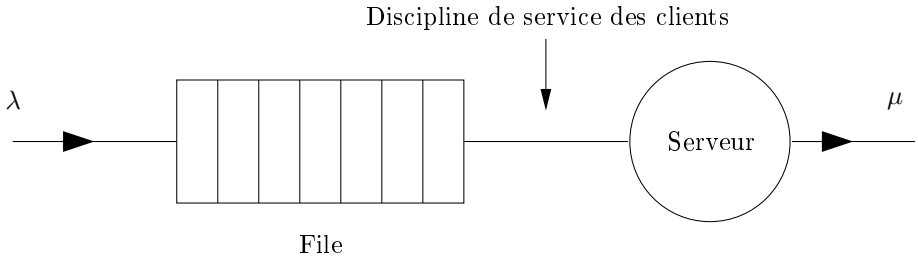

FIG. 3.6: *Un modèle de file d'attente*

La théorie des files d'attente permet d'analyser les performances de systèmes composés de serveurs et de clients qui sont soumis à des phénomènes d'attente : des personnes dans la salle d'attente d'un médecin, un routeur aiguillant des paquets de données, ...

Un modèle de file d'attente est composé d'une file qui stocke les clients en attente de service ainsi que d'un serveur (cf. figure 3.6). Le fonctionnement d'une file d'attente est le suivant :

1. Les clients arrivent dans le système à une certaine fréquence.
2. Lorsqu'un client arrive alors que le serveur est inoccupé et que la file est vide, le serveur exécute immédiatement le traitement associé au client. Si un client arrive dans le système alors que le serveur est déjà occupé, le client est placé dans la file. Le serveur prend alors en compte le client lorsqu'il a terminé le traitement en cours.

En général, les clients sont servis suivant une discipline FIFO. D'autres disciplines de service ont été étudiées telles que la discipline LIFO[16] ou les disciplines basées sur la priorité des clients [Sta92, AVS02].

3. Si la taille de la file est bornée, différentes politiques peuvent être appliquées lorsqu'un client arrive et que la file est pleine[17]. Par la suite, on suppose qu'une file d'attente est de taille infinie.

Une file d'attente est généralement décrite par au moins trois paramètres selon la notation de KENDALL : $\lambda/\mu/c$. λ est le taux d'arrivée moyen des clients. μ est le taux de service moyen. Enfin, c est le nombre de serveurs de la file d'attente. Par la suite, on suppose que toute file d'attente respecte la loi de conservation de débit :

Définition 1 (Loi de conservation du débit) *Soit $\rho = \frac{\lambda}{\mu}$, le taux d'occupation du serveur, on suppose que :*

$$\rho < 1$$

La loi de conservation du débit est une condition nécessaire pour qu'un modèle de file d'attente puisse être analysé.

Selon le taux d'arrivée des clients et le temps de service, de nombreuses files d'attente peuvent être définies. En effet, le taux d'arrivée et le temps de service peuvent suivre différentes distributions telles que les distributions Markoviennes, Déterministes ou Générales.

Une distribution Markovienne, notée M, est décrite par un processus de Poisson de paramètre λ lorsqu'elle représente le taux d'arrivée des clients, ou par une loi exponentielle dont le temps de service moyen est $1/\mu$ lorsqu'il s'agit de la distribution des services. La variance sur le délai inter-arrivée moyen est égale à $\frac{1}{\lambda}$. La variance sur le temps de service moyen est de $\frac{1}{\mu}$.

Une distribution déterministe, notée D, décrit un taux d'arrivée constant λ de clients ou un taux de service constant μ. Le temps de service moyen est de $\frac{1}{\mu}$. Le délai inter-arrivée moyen est de $\frac{1}{\lambda}$. Les variances associées au temps de service moyen et au délai moyen d'inter-arrivée sont nulles.

Finalement, une distribution générale, notée G, est décrite par un taux moyen associé à une variance σ^2.

À partir de ces distributions classiques, il est possible de définir différentes files d'attente telles que les files d'attente M/M/1, M/D/1, M/G/1 ou D/D/1 [Kle75b, Rob90]. Ces quatre files d'attente possèdent un seul serveur. La file d'attente M/M/1 est la file d'attente dont le comportement a été le plus étudié. Le taux d'arrivée et les temps de service sont markoviens. Les critères de performance de M/M/1 sont obtenus grâce à la résolution d'un processus de naissance et de mort. La file d'attente M/G/1 est semi-markovienne. Les arrivées sont décrites par un processus de Poisson. La loi de service est générale. L'obtention des critères de performance nécessite l'utilisation de chaînes de Markov immergées [CGL94]. Contrairement à la file M/G/1, la file d'attente M/D/1 possède un temps de service constant. Les critères de performance sont obtenus à partir des résultats de M/G/1 en appliquant une variance

[16] Last In First Out.

[17] À titre d'exemple, une politique possible consiste à ne pas tenir compte des clients qui arrivent lorsque le file d'attente est pleine. Les clients sont donc perdus.

nulle. Enfin, la file d'attente D/D/1 est complètement déterministe : le temps entre 2 arrivées et le temps de service sont constants.

À partir d'un modèle de file d'attente, le concepteur d'un système peut obtenir différents critères qui lui indiquent les performances de son système. On cherche ici à prédire, entre autre, le nombre moyen de client dans la file d'attente, ou le temps d'attente moyen d'un client, ou encore la probabilité d'avoir un nombre donné de clients dans la file d'attente.

Nous présentons maintenant les critères de performance les plus classiques pour les files d'attente M/M/1, M/D/1 et M/G/1 :

- W est le délai moyen d'attente d'un client dans le système. W est la somme du temps de service moyen (noté W_s) et du temps d'attente moyen dans la file (noté W_q) :

$$W = W_q + W_s \tag{3.1}$$

- L est le nombre moyen de clients dans le système. L est la somme du nombre moyen de clients dans la file d'attente (noté L_q) et du nombre moyen de clients dans le serveur (noté L_s) :

$$L = L_q + L_s \tag{3.2}$$

  Par ailleurs, il existe une relation entre W et L qui est connue sous le nom de la loi de Little [Kle75b] :

  Théorème 1 (Loi de Little) *Le nombre moyen de clients dans la file d'attente est égal au nombre moyen de clients arrivant dans la file d'attente pendant le temps d'attente moyen d'un client dans la file d'attente. Ou encore :*

$$L = \lambda \cdot W$$

- Enfin, P_n est la probabilité qu'il y ait n clients dans le système.

File d'attente	L	L_q	W	W_q	P_n
M/M/1	$\frac{\lambda \cdot W_s}{1-\rho}$	$\frac{\lambda^2 \cdot W_s^2}{1-\rho}$	$\frac{W_s}{1-\rho}$	$\frac{\lambda \cdot W_s^2}{1-\rho}$	$(1-\rho) \cdot \rho^n$
M/G/1	$\lambda \cdot W_s + \frac{\lambda^2 \cdot (W_s^2 + \sigma^2)}{2 \cdot (1-\rho)}$	$\frac{\lambda^2 \cdot (W_s^2 + \sigma^2)}{2 \cdot (1-\rho)}$	$W_s + \frac{\lambda \cdot (W_s^2 + \sigma^2)}{2 \cdot (1-\rho)}$	$\frac{\lambda \cdot (W_s^2 + \sigma^2)}{2 \cdot (1-\rho)}$	-
M/D/1	$\lambda \cdot W_s + \frac{\lambda^2 \cdot W_s^2}{2 \cdot (1-\rho)}$	$\frac{\lambda^2 \cdot W_s^2}{2 \cdot (1-\rho)}$	$W_s + \frac{\lambda \cdot W_s^2}{2 \cdot (1-\rho)}$	$\frac{\lambda \cdot W_s^2}{2 \cdot (1-\rho)}$	-

TAB. 3.3: *Critères de performance des files d'attente M/M/1, M/D/1 et M/G/1*

Le tableau 3.3 présente les principaux critères de performance, pour les files d'attente M/M/1, M/D/1 et M/G/1 (avec $W_s = \frac{1}{\mu}$ et $\rho = \lambda \cdot W_s = L_s$). Les résultats de M/D/1 peuvent être retrouvés à partir des résultats de M/G/1 lorsque $\sigma^2 = 0$. De même, les résultats de M/M/1 peuvent être retrouvés à partir des résultats de M/G/1 lorsque $\sigma^2 = \frac{1}{\mu}$.

3.4.2 La loi de service P

Appliqué au contexte d'une spécification AADL, un système constitué d'un ensemble de *threads* périodiques produisant ou consommant des messages dans le tampon d'un *event data port* peut être abstrait par une file d'attente dont les temps de service modélisent l'ordonnancement d'un ensemble de *threads* périodiques.

La loi de service P modélise donc un serveur de file d'attente dont le comportement est celui d'un *thread* périodique ordonnancé par un algorithme tel que Rate Monotonic ou Earliest Deadline First. Ainsi :

- Le temps de service de la file est variable et dépend de l'algorithme d'ordonnancement temps réel. Le temps de réponse du serveur peut varier car il peut être préempté à certains instants par des *threads* de plus fortes priorités.
- Par moment, le serveur est inactif, en particulier lorsqu'il doit attendre son prochain réveil périodique.
- D'autre part, on suppose que ce *thread* consomme un message au plus lors de son activation périodique. Lors d'un réveil périodique du *thread*/serveur de file d'attente, il est possible qu'aucun message ne soit disponible dans le port. Dans ce cas, le *thread* attend le réveil périodique suivant.

Décrire une loi de service consiste à décrire le temps moyen de service (noté W_s) ainsi que sa variance (noté σ_s^2). Nous rappelons la définition générale du temps de service d'une file d'attente :

Définition 2 (Temps de service) *Le temps de service S_i est le temps passé par un client à être traité par un serveur [Kle75b]. S_i est le délai entre l'activation du serveur et la fin du traitement du client.*

- *Si le système est vide, le serveur est activé dès qu'un nouveau client arrive.*
- *Si un ou plusieurs clients sont en attente dans la file, le serveur est activé immédiatement après la fin du traitement du client précédent.*

De cette définition, le temps de service moyen W_s et la variance du temps de service σ_s^2 peuvent alors être définis comme suit :

$$W_s = \frac{1}{n} \sum_{i=1}^{n} S_i \tag{3.3}$$

Et

$$\sigma_s^2 = \frac{1}{n} \sum_{i=1}^{n} S_i^2 - W_s^2 \tag{3.4}$$

Où n est le nombre de temps de service.

3.4.3 La file d'attente M/P/1 : établissement des critères moyens de performance

Avec cette première file d'attente, nous proposons une méthode pour étudier le dimensionnement du tampon d'un *event data port* dont les messages sont consommés par un *thread* périodique et dont

les arrivées de messages sont modélisées par une loi exponentielle. Nous pouvons modéliser ce tampon par une file d'attente M/P/1. La discipline de service des messages est la politique FIFO.

Nous cherchons à calculer la file d'attente M/P/1. Calculer la file d'attente consiste à déterminer le nombre de messages L et le temps d'attente moyen d'un message W dans la file d'attente (cf. équations 3.1 et 3.2). Il est donc nécessaire de caractériser le temps de service moyen W_s et sa variance σ_s^2. La démarche consiste à déterminer ces deux paramètres de la file d'attente M/P/1. Puis, il est alors possible de calculer les critères de performances W et L à partir des résultats théoriques de la file d'attente M/G/1 [Rob90, Kle75b].

3.4.3.1 Notion de consommation effective

Préalablement à la définition d'un temps de service et de sa variance pour la file M/P/1, nous devons définir la notion de consommation effective.

En effet, dans la partie précédente, nous avons défini la loi de service P comme une loi qui réveille le serveur périodiquement, quelque soit le nombre de clients présents dans la file d'attente. Il est donc possible qu'un serveur puisse être réveillé sans message à traiter. Pour évaluer les critères W et L, il est nécessaire de précisément quantifier le nombre de clients traités par le serveur par unité de temps. Pour ce faire, nous définissons la notion de consommation effective :

Définition 3 (Consommation effective) *Une consommation est dite :*
- *Effective si le serveur est activé et si au moins un client est présent dans la file d'attente.*
- *Non-effective lorsque le serveur est activé et qu'il n'y a pas de client dans la file d'attente.*

Il est alors possible de définir le taux de consommation effective :

Théorème 2 *Le taux de consommations effectives U_c de la file d'attente M/P/1 est de :*

$$U_c = 1 - P_0 = \rho$$

Où ρ est le taux d'utilisation de la file d'attente et P_0, la probabilité que la file soit vide.

En effet, $1 - P_0$ est la probabilité qu'il y ait au moins un client dans la file d'attente. D'autre part, pour une file G/G/1, $\rho = 1 - P_0$ [Kle75b]. La file d'attente M/P/1 étant un cas particulier de la file G/G/1, nous avons donc $U_c = \rho$.

3.4.3.2 Expression du temps de service moyen et de sa variance pour M/P/1

Maintenant que nous avons défini la notion de consommation effective, nous étudions le temps de service S_i. L'équation 3.3 permet de calculer le temps de service moyen. Cette équation requiert que soit évalué l'ensemble des valeurs des temps de service S_i.

Dans le cas de la file d'attente M/P/1, cet ensemble est difficile à déterminer. En effet, à un temps de service S_i correspond une consommation effective. Puisque les messages arrivent dans le tampon de manière aléatoire, il est difficile de prédire qu'une activation périodique du serveur (ou *thread*

consommateur) implique une consommation effective qui doit être comptabilisé dans le temps de service moyen (nouveau S_i).

C'est pourquoi nous proposons une résolution approchée de la file d'attente. Pour ce faire, nous étudions le cas où une consommation effective intervient à chaque activation périodique du serveur/-consommateur (c'est à dire quand U_c tend vers 1) :

Théorème 3 *Lorsque ρ tend vers 1, le temps de service moyen et sa variance sont :*

$$W_s = P_{cons}$$

$$\sigma_s^2 = \frac{1}{n}\sum_{i=1}^{n} S_i^2 - W_s^2$$

Où cons est le serveur de la file d'attente M/P/1.

Puis, nous étudions également la file d'attente lorsque U_c tend vers 0. Dans ce second cas de figure, seules quelques rares activations du serveur périodique impliquent la consommation d'un message depuis le tampon. Le nombre de consommations effectives tend vers 0 et le temps de service moyen est défini par le théorème 4.

Théorème 4 *Lorsque ρ tend vers 0, le temps de service moyen et sa variance est de :*

$$W_s = \frac{P_{cons}}{2}$$

$$\sigma_s^2 = \frac{P_{cons}^2}{12}$$

Enfin, nous approximons le temps de service moyen et sa variance (théorème 5) par une espérance mathématique des deux cas précédents qui sont des extrêmes ($U_c \to 0$ et $U_c \to 1$).

Théorème 5 *Le temps de service moyen est alors égale à :*

$$W_s = \frac{P_{cons}}{2} \cdot (1+\rho) = \frac{P_{cons}}{2 \cdot (1 - \lambda \cdot \frac{P_{cons}}{2})}$$

Et la variance de ce temps de service moyen est de :

$$\sigma_s^2 = \rho \cdot \left(\frac{1}{n}\sum_{i=1}^{n} S_i - W_s^2 \right) + (1-\rho) \cdot \frac{P_{cons}^2}{12}$$

Les preuves détaillées des théorèmes 3, 4 et 5 sont proposées dans la thèse de Jérôme Legrand [Leg04].

Finalement, grâce au temps de service moyen défini par le théorème 5, l'analyse de performances d'un *event data port* peut être conduite en calculant le nombre moyen de messages présents dans le port (critère L) et le temps d'attente moyen d'un message dans le port (critère W) grâce aux formules de la file d'attente M/G/1 du tableau 3.3.

3.4.4 La file d'attente P/P/1 : établissement des critères maximums de performance

Cette partie présente un second modèle de file d'attente adapté aux applications temps réel : la file d'attente P/P/1. Cette file d'attente permet de modéliser et de dimensionner le tampon d'un *event data port* dont les productions et les consommations de messages sont effectuées par des *threads* périodiques. Les productions et les consommations de messages sont conformes à la loi P. La discipline de service des messages est la politique FIFO. Dans cette partie, nous proposons une résolution exacte de cette file d'attente lorsque n *threads* produisent des messages qui sont consommés par un *thread* uniquement. On suppose, en outre, que pour tout *thread* i nous avons $D_i \leq P_i$.

Résoudre la file d'attente consiste à calculer le nombre de messages et le temps d'attente d'un message dans la file d'attente (cf. équations 3.1 et 3.2). On cherche ici à établir des bornes sur ces critères. Nous notons L_{max} et W_{max} les bornes maximales sur le nombre de messages et sur le temps d'attente d'un message dans la file d'attente.

La résolution de cette file d'attente est basée sur un résultat classique exploité, par exemple, dans les réseaux ATM et plus particulièrement dans la couche d'adaptation AAL1 [18] [GK96]. Nous commençons par décrire ce résultat. Puis, nous montrons qu'il s'agit d'une illustration de la loi de Little [Kle75a]. Grâce à Little, une expression de L_{max} est proposée. Nous en déclinons un exemple de test de faisabilité.

Pour des exemples supplémentaires de tests de faisabilité et la démonstration des propositions de cette partie, le lecteur pourra se référer à [Leg04, LSNM04, LSN+03].

3.4.4.1 Taille des tampons dans la couche de transport à débit constant des réseaux ATM

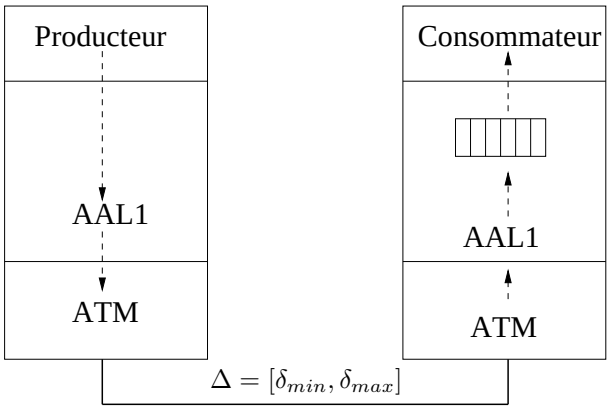

FIG. 3.7: *La couche de transport à débit fixe d'ATM*

La couche d'adaptation 1 d'ATM a pour objet le transfert à débit fixe de la voix. Le transport de la voix est sujet à des contraintes temporelles fortes : chez le récepteur, les paquets de données doivent être livrés à la cadence d'émission, afin d'assurer une présentation sans interruption de la voix.

[18] ATM Adaptation Layer 1.

Regardons une architecture constituée de deux processeurs connectés par un réseau ATM (cf. figure 3.7). L'émetteur transmet des données audio à une cadence fixe. Nous noterons ce débit d'émission d. Le débit est exprimé en cellule par seconde. La cellule est l'unité de transfert dans un réseau ATM.

Le temps de transport de ces cellules de l'émetteur au récepteur est variable et est compris entre δ_{min} et δ_{max}, respectivement le délai minimum et maximum de transport. Cette variation du délai de transmission, ou *gigue*, est due à la traversée des différents noeuds (commutateurs ou brasseurs) situés entre l'émetteur et le récepteur. Cette gigue est notée $\triangle = \delta_{max} - \delta_{min}$.

On cherche alors à restituer, chez le récepteur, le flux de données audio à la cadence de l'émetteur. La solution classique consiste à mémoriser les informations audio dans un tampon afin de compenser la gigue produite par le réseau de transmission. Dans [GK96], il est démontré qu'un tampon pouvant mémoriser L_{max} messages est nécessaire et suffisant pour compenser la gigue induite par la transmission tout en évitant un débordement du tampon :

$$L_{max} = \left\lceil \frac{W_{max}}{d} \right\rceil \tag{3.5}$$

où $W_{max} = 2 \cdot \triangle$.

W_{max} est le temps d'attente de la première cellule dans le tampon avant sa consommation. W_{max} est aussi le plus grand délai sans consommation de cellule. L'équation 3.5 est en fait une illustration de la loi de Little :

$$L_{max} = \lambda_{max} \cdot W_{max} \tag{3.6}$$

Avec, pour un réseau ATM/AAL1, $\lambda_{max} = \frac{1}{d}$. La loi de Little dit ici que la taille maximale du tampon est égale à un taux de production maximum pendant le délai d'attente maximum d'un message.

3.4.4.2 Établissement de L_{max} pour la file d'attente P/P/1

Nous cherchons maintenant à appliquer la loi de Little à notre file d'attente P/P/1.

D'une part, le délai d'attente d'un message particulier dans la file d'attente peut être établi de la façon suivante :

$$W_{max} = (y + 1) \cdot P_{cons} + D_{cons} \tag{3.7}$$

Où P_{cons} est la période du consommateur (ou serveur de la file d'attente) et D_{cons} le délai critique du consommateur. y est le nombre de messages déjà présents dans le tampon au moment où le message étudié arrive. En effet, selon la configuration du jeu de *threads* et compte tenu du fait que le tampon fonctionne de façon FIFO, entre l'instant de production du message et l'instant de sa consommation, y activations du consommateur sont nécessaires pour consommer les messages déjà présents dans le tampon.

D'autre part, puisque nous avons fait l'hypothèse qu'un *thread* produit un seul message par activation, le taux maximum de production est donc d'un message par période, soit :

$$\lambda_{max} = \frac{1}{P_{prod}} \tag{3.8}$$

Avec P_{prod}, la période du producteur *prod*. Nous pouvons alors déterminer le nombre maximum de messages L_{max} pour une file d'attente P/P/1 qui est de :

$$L_{max} = \max_{\forall y \geq 0} \left(\sum_{prod \in PROD} \left\lceil \frac{W_{max} + O_{prod}}{P_{prod}} \right\rceil - y \right) \tag{3.9}$$

Cette équation diffère de l'équation 3.5 en trois points :

1. Il existe plusieurs producteurs. Il est donc nécessaire de tenir compte des messages produits par chaque producteur. $PROD$ est l'ensemble des producteurs et P_{prod}, la période du producteur *prod*.

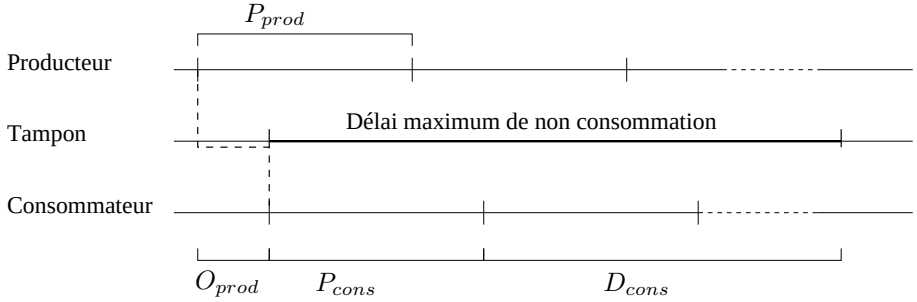

FIG. 3.8: O_{prod} *: désynchronisation des producteurs et consommateurs*

2. Dans ATM/AAL1, le délai maximum d'attente est évalué à partir de l'envoi de la première cellule du flux. Or, dans notre cas, le début du délai maximum d'attente peut intervenir à tout moment et correspondre à l'envoi de n'importe quel message. Il existe donc une certaine désynchronisation entre les réveils des producteurs et des consommateurs (cf. figure 3.8). La borne maximale sur cette désynchronisation est notée O_{prod}. O_{prod} représente le pire délai entre un réveil de la tâche consommateur correspondant au début du délai d'attente étudié, et un réveil de la tâche producteur *prod* qui va émettre un message pendant ce délai. Durant cette désynchronisation, des messages peuvent être produits. Il est donc nécessaire de comptabiliser la production maximale de messages sur l'intervalle de temps $W_{max} + O_{prod}$.

3. Le calcul de L_{max} requiert d'évaluer l'équation 3.9 pour toute valeur de y. Par la loi de conservation du débit, y est borné. Notons que contrairement à la solution utilisée dans ATM, y messages sont retirés de la borne. En effet, la borne des tampons de la couche AAL1 est construite grâce au plus grand délai d'accumulation des messages dans le tampon qui est égal au plus grand délai sans consommation. Dans notre cas, ces deux délais ne sont plus équivalents. Pendant le délai d'attente, y messages sont consommés. Ils doivent donc être comptabilisés.

3.4.4.3 Exemple d'un test de faisabilité construit à partir de P/P/1

À partir d'une équation comme l'équation 3.9, il est possible d'élaborer différents tests de faisabilité. Il est alors nécessaire d'étudier y aux limites afin de calculer L_{max}. Par la suite, le calcul de W_{max} peut être obtenu par application de la loi de Little. Ainsi, le test suivant peut être proposé :

Théorème 6 *Pour un tampon partagé par n threads périodiques producteurs et un thread périodique consommateur tel que $D_i \leq P_i$, et tel que les threads périodiques sont harmoniques*[19]*, alors les bornes maximales sur le nombre de messages et sur le temps d'attente d'un message sont de :*

$$\begin{aligned} L_{max} &= 2 \cdot n \\ W_{max} &= 2 \cdot n \cdot P_{cons} \end{aligned} \tag{3.10}$$

Dans le cas d'un jeu de threads non harmoniques, les bornes maximales sur le nombre de messages et sur le temps d'attente d'un message sont de :

$$\begin{aligned} L_{max} &= 2 \cdot n + 1 \\ W_{max} &= (2 \cdot n + 1) \cdot P_{cons} \end{aligned} \tag{3.11}$$

Le suite de cette partie décrit comment ce test peut être élaboré. Le lecteur qui ne souhaite pas lire les détails concernant l'élaboration de ce test peut poursuivre sa lecture à la partie 3.4.5 page 70.

Nous commençons par étudier le cas où les *threads* ne sont pas harmoniques. Nous montrons grâce à l'équation 3.9, qu'au pire cas, un seul producteur peut accumuler 3 messages et les $n-1$ producteurs restants accumulent au plus 2 messages.

Nous montrons d'abord qu'un producteur peut produire au plus $y+3$ messages pendant le délai d'attente. Pour un jeu de *threads* composé de 1 consommateur et n producteurs, nous savons que $P_{prod} > P_{cons}$. En effet, la forme générale de la loi de conservation du débit s'exprime par :

$$\sum_{prod \in PROD} \frac{1}{P_{prod}} \leq \sum_{cons \in CONS} \frac{1}{P_{cons}} \tag{3.12}$$

Avec $PROD$, resp. $CONS$, l'ensemble des *threads* qui produisent, resp. consomment, des messages dans le tampon. Cette équation est une condition nécessaire d'existence d'une borne maximale de la taille des tampons. Dans le cas d'une file d'attente avec un consommateur, la loi de conservation du débit est alors :

$$\sum_{prod \in PROD} \frac{1}{P_{prod}} \leq \frac{1}{P_{cons}} \tag{3.13}$$

Le débit maximum d'un producteur i est obtenu lorsque $P_i \to P_{cons}$. Soit une production maximale de $\left\lceil \frac{(y+2) \cdot P_{cons} + O_i}{P_i} \right\rceil = \left\lceil \frac{(y+2) \cdot P_{cons} + P_{cons}}{P_{cons}} \right\rceil = y+3$ messages pendant W_{max}.

[19] Un ensemble de *threads* est harmonique si tous les *threads* de cet ensemble possèdent des périodes multiples entre eux.

Nous cherchons maintenant à déterminer l'intervalle I des valeurs de P_i pour lequel la production reste de $y+3$ messages. Lorsque $P_i \to P_{cons}$, le premier (ou le dernier) des $y+3$ messages est produit dans l'intervalle $]0, P_{cons}[$. Ce délai peut être uniformément réparti aux $y+1$ instances du producteur afin de maximiser sa période tout en conservant une production de $y+3$ messages. Ainsi, la plus grande période du producteur i, telle que $y+3$ messages soient produits, est donc $P_i = \frac{P_{cons}}{y+1} + P_{cons} = \frac{(y+2)\cdot P_{cons}}{y+1}$. La production est de $y+3$ messages pour un producteur lorsque sa période est comprise dans l'intervalle $I =]P_{cons}, \frac{(y+2)\cdot P_{cons}}{y+1}[$.

Nous montrons maintenant que si la période d'un producteur tend vers $\frac{(y+2)\cdot P_{cons}}{y+1}$, alors les $n-1$ producteurs restants ont une période qui tend vers $(y+2)\cdot P_{cons}^{+}$. Cette propriété peut être démontrée à l'aide de la loi de conservation de débit. La loi de conservation du débit peut être exprimée par :

$$\frac{1}{x} + \sum_{prod \in PROD^*} \frac{1}{P_{prod}} \leq \frac{1}{P_{cons}}$$

ou encore :

$$\sum_{prod \in PROD^*} \frac{1}{P_{prod}} \leq \frac{x - P_{cons}}{x \cdot P_{cons}}$$

Avec $PROD^*$, l'ensemble des producteurs $PROD$ auquel nous avons retiré le producteur dont la période est x. Ce producteur est celui qui produit $y+3$ messages.

Posons :

$$f(x) = \frac{x - P_{cons}}{x \cdot P_{cons}}$$

et étudions les limites de cette fonction lorsque $x \to \frac{(y+2)\cdot P_{cons}}{y+1}$ et lorsque $x \to P_{cons}$. On obtient :

1. $\lim_{x \to \frac{(y+2)\cdot P_{cons}}{y+1}} f(x) = \frac{1}{(y+2)\cdot P_{cons}}$
   Or $\sum_{prod \in PROD^*} \frac{1}{P_{prod}} \leq \frac{1}{(y+2)\cdot P_{cons}}$; ce qui implique, au pire cas : $\forall prod \in PROD^* : P_{prod} \to (y+2)\cdot P_{cons}^{+}$.
2. $\lim_{x \to P_{cons}} f(x) = 0$
   Or $\sum_{prod \in PROD^*} \frac{1}{P_{prod}} \leq 0$; ce qui implique, au pire cas : $\forall prod \in PROD^* : P_{prod} \to \infty$.

L'intervalle des périodes des producteurs appartenant à $PROD^*$ est donc $J =](y+2)\cdot P_{cons}, \infty[$. Nous obtenons donc une production maximale de :

$$\sum_{prod \in PROD^*} \left\lceil \frac{W_{max} + O_{prod}}{P_{prod}} \right\rceil$$

ou encore :

$$\sum_{prod \in PROD^*} \left\lceil \frac{(y+2)\cdot P_{cons} + (y+2)\cdot P_{cons}^{+}}{(y+2)\cdot P_{cons}^{+}} \right\rceil = 2\cdot n - 2$$

Finalement, d'après l'équation 3.9, la borne est de :

$$L_{max} = y + 3 + \sum_{prod \in PROD^*} \left\lceil \frac{(y+2) \cdot P_{cons} + (y+2) \cdot P^+_{cons}}{(y+2) \cdot P^+_{cons}} \right\rceil - y$$

et donc

$$L_{max} = y + 3 + 2 \cdot n - 2 - y = 2 \cdot n + 1$$

D'après Little, nous avons finalement :

$$W_{max} = (2 \cdot n + 1) \cdot P_{cons}$$

Dans le cas d'un jeu de tâches harmoniques, nous appliquons la méthode décrite ci-dessus. Cette fois-ci, un producteur émet au pire cas $y + 2$ messages pendant le délai d'attente. En effet, dans le cas harmonique, le décalage entre les activations de deux tâches ayant la même période est nul. Un producteur i émet donc au pire cas $\left\lceil \frac{(y+2) \cdot P_{cons}}{P_{cons}} \right\rceil = y + 2$ messages. L'intervalle I des valeurs possibles de P_i devient $[P_{cons}, \frac{(y+2) \cdot P_{cons}}{y+1}]$. Si l'on substitue à l'équation 3.9 ces informations comme pour le cas non harmonique, nous obtenons :

$$L_{max} = y + 2 + \sum_{prod \in PROD^*} \left\lceil \frac{(y+2) \cdot P_{cons} + (y+2) \cdot P_{cons}}{(y+2) \cdot P_{cons}} \right\rceil - y$$

Soit,

$$L_{max} = y + 2 + 2 \cdot n - 2 - y = 2 \cdot n$$

Et,

$$W_{max} = 2 \cdot n \cdot P_{cons}$$

3.4.5 Application à un modèle AADL

Dans les parties précédentes, par l'étude des files d'attente P/P/1 et M/P/1, nous avons montré comment élaborer des tests de faisabilité qui permettent le dimensionnement mémoire d'une architecture temps réel comportant des tampons. La figure 3.9 présente un modèle AADL sur lequel ces tests de faisabilité peuvent être appliqués. Le modèle contient trois *threads* : *producer1*, *producer2* et *consumer1*. Les *threads producer1* et *producer2* émettent des messages vers un port de type *event data*. Le *thread consumer1* lit les messages émis par *producer1* et *producer2* grâce à ce même port. Les *threads* sont périodiques et ordonnancés par un ordonnancement à priorité fixe. On suppose vérifié le respect des échéances des *threads* de cet exemple.

La figure 3.10 présente une analyse de l'empreinte mémoire pour ce modèle. Comme pour le patron «Ravenscar», la partie haute de la fenêtre affiche la simulation de cette architecture. La partie basse présente une analyse effectuée à l'aide de la file d'attente P/P/1. Dans la partie basse, la traçabilité

```
THREAD producer
   FEATURES
      data_source : OUT EVENT DATA PORT;
END producer;

THREAD IMPLEMENTATION producer.Impl
   PROPERTIES
      Dispatch_Protocol => Periodic;
      Period => 20 Ms;
END producer.Impl;

THREAD consumer
   FEATURES
      data_sink : IN EVENT DATA PORT;
END consumer;

THREAD IMPLEMENTATION consumer.Impl
   PROPERTIES
      Dispatch_Protocol => Periodic;
      Period => 10 Ms;
END consumer.Impl;

PROCESS with_buffer
END with_buffer;

PROCESS IMPLEMENTATION with_buffer.Impl
   SUBCOMPONENTS
      producer1 : THREAD producer.Impl;
      producer2 : THREAD producer.Impl;
      consumer1 : THREAD consumer.Impl;
   CONNECTIONS
      EVENT DATA PORT producer1.data_source -> consumer1.data_sink;
      EVENT DATA PORT producer2.data_source -> consumer1.data_sink;
END with_buffer.Impl;
```

FIG. 3.9: *Modèle AADL comportant des connexions de type event data*

des tests de faisabilité utilisés est assurée en affichant à l'utilisateur la publication décrivant le test de faisabilité appliqué, voire la page et le numéro d'équation.

Avec ce cas d'école, la période d'étude étant courte, la simulation permet d'énumérer exhaustivement l'ordonnancement du modèle. Le quatrième chronogramme montre les instants où un message est déposé dans le tampon du port (rectangle bleu) et les instants ou un message est extrait du tampon (rectangle rouge). On constate aisément qu'un tampon avec une capacité de mémorisation de deux messages est suffisant pour cette architecture. L'application des tests de faisabilité élaborés à partir de la file d'attente P/P/1 permet de déterminer le nombre maximum de message à mémoriser dans le port : la borne maximale est de quatre messages. La borne maximale de quatre messages est ici plus grande que la valeur obtenue par simulation. En effet, la file d'attente P/P/1 ne fait pas d'hypothèse sur l'ordonnancement des tâches, hormis le fait que chaque *thread* est supposé respecter son échéance. Le test de faisabilité employé ici est donc plus général mais conduit à une condition suffisante et non nécessaire.

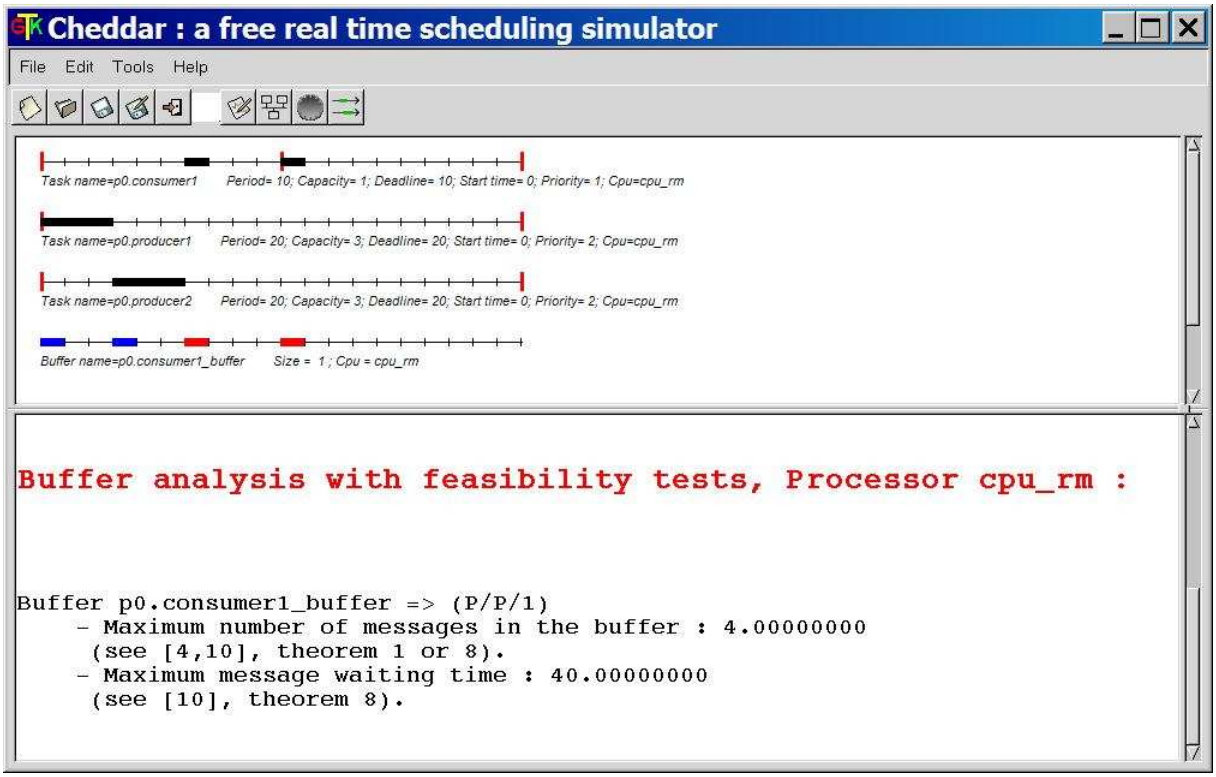

FIG. 3.10: *Analyse du patron «Queued Buffer»*

3.5 Conclusion

Dans ce chapitre, nous avons étudié comment faire une analyse des performances de modèles d'architecture qui soit la plus automatique possible. L'approche étudiée dans ce chapitre s'inspire de celle élaborée pour la méthode HRT-HOOD [BW94]. Elle consiste à définir des patrons de conception. Un patron de conception propose une solution à une famille de problèmes de concurrence. À chaque patron est associé des tests de faisabilité qui permettent de vérifier les performances d'un modèle d'architecture conforme au patron de conception. L'analyse de performances est simplifiée car le concepteur n'a pas à vérifier que son architecture respecte les hypothèses des tests de faisabilité : par construction, l'emploi d'un patron de conception pour la modélisation d'une architecture garantit que celle-ci est conforme aux hypothèses des tests de faisabilité.

Nous avons étudié quatre patrons de conception exprimés avec AADL. D'une part, avec le patron de conception «Ravenscar», nous avons vérifié que la version 1 du standard AADL définissait les propriétés nécessaires à l'application des méthodes d'analyse les plus couramment employées. Nous avons proposé des propriétés devant être ajoutées à un modèle AADL «Ravenscar» afin d'en assurer son analyse.

Télécom-Paris Tech a par ailleurs montré qu'il était possible d'intégrer ces mécanismes d'analyse dans un processus permettant, à partir d'un modèle AADL conforme au profil Ravenscar, de générer l'application avec PolyORB-HI et d'effectuer l'analyse du modèle avec CPN-AMI [HHK+06] et Cheddar

[HZPK07]. Ce travail montre que dans certains cas, il est possible de mettre en place un processus automatisant la vérification de l'ordonnancement.

D'autre part, avec le patron de conception «Queued buffer», nous avons abordé le problème de l'analyse de l'empreinte mémoire d'un modèle de conception. Nous avons proposé deux files d'attente : les files d'attente P/P/1 et M/P/1. Ces files d'attente modélisent le comportement des flots de contrôle généralement employés dans les architectures temps réel (exemple : tâches périodiques). Nous avons montré qu'il était possible d'en exhiber des tests de faisabilité, et que ces tests pouvaient être appliqués à des modèles AADL. L'utilisation de ces files d'attente n'est cependant pas limitée à AADL et de nombreux problèmes restent ouverts concernant ces deux files d'attente (généralisation de la file d'attente P/P/1, recherche d'une résolution exacte pour M/P/1, ...).

Les activités décrites dans ce chapitre ouvrent la porte à bien d'autres perspectives. D'abord, nous avons pour l'instant délaissé le patron de conception «Blackboard». Nous avons d'ailleurs indiqué qu'il existait de nombreux autres paradigmes de synchronisation employés par les architectes systèmes qu'il serait donc utile d'étudier. L'objectif final de ces premières perspectives reste l'intégration et l'expérimentation de l'usage de ces patrons AADL avec un outil d'ingénierie logicielle tel que Stood.

Certaines des propriétés AADL que nous avons proposées ont été intégrées dans la version 2 du standard AADL. L'annexe comportementale [SAE07] devrait, d'autre part, proposer une solution à certains des problèmes cités dans la partie 3.3, notamment la spécification des moments où un composant *data* est utilisé par un composant *thread*. Comme pour les propriétés standardisées par AADL, nous prévoyons de vérifier que l'annexe comportementale d'AADL permet effectivement de spécifier les comportements nécessaires en vue d'une analyse de performances automatisée.

Chapitre 4

Accroître l'applicabilité de la théorie de l'ordonnancement temps réel : quand aucun test de faisabilité ne peut être appliqué

Sommaire

Ce chapitre présente les contributions du projet Cheddar à l'analyse de performances lorsque aucun test de faisabilité ne peut être appliqué de façon automatique pour l'analyse d'une architecture. En effet, de nombreuses applications industrielles ne peuvent pas être analysées par les méthodes analytiques de la théorie de l'ordonnancement car elle ne fournit pas nécessairement de tests de faisabilité pour les ordonnanceurs et les modèles de tâches employés dans le monde industriel. En effet, de nombreux projets industriels font usage de modèles d'ordonnanceur et de tâche spécifiques et d'une complexité élevée. Par ailleurs, élaborer ces tests de faisabilité est une tâche coûteuse et généralement difficile. Les systèmes à analyser dans ce contexte sont généralement de grande taille, de sorte qu'il n'est pas toujours possible d'employer des méthodes de *model-checking* [BBL99].

Contrairement au chapitre précédent, le concepteur ne peut donc pas être assisté par le biais des patrons de conception. Le monde industriel est donc souvent réduit à vérifier certains de ces systèmes par la simulation. Dans ce contexte, une plafe-forme d'analyse de performance doit permettre de modéliser le comportement de ces ordonnanceurs spécifiques, puis de faciliter la mise en œuvre de simulations.

Même dans le cas où seule la simulation est possible, il est envisageable d'employer les fonctionnalités de simulation de certains environnements de *model-checking*. C'est notamment le cas des réseaux de Petri [GV03] et d'outils tels que CPN Tools [Wel06], CPN-AMI [HHK+06] ou Tina [BV06].

Une autre solution consiste à développer des outils de simulations ad-hoc. Cette solution est coûteuse et n'offre, en général, qu'un faible niveau de réutilisation du logiciel de simulation.

L'environnement Cheddar propose une alternative par la mise à disposition d'un langage spécifique ainsi que des outils associés (interpréteur, parser, générateur, ...). Ce langage spécifique permet de modéliser un ordonnanceur temps réel et d'en étudier le comportement par simulation.

Le langage Cheddar est défini par l'association de deux langages spécifiques : un sous ensemble d'Ada 95 et des automates temporisés.

Ils existent de nombreux travaux traitant de la modélisation d'ordonnanceurs temps réel. Ainsi, les projets Shark [GAGB01] ou BOSSA [BM02] proposent une architecture et un langage spécifique afin de faciliter le développement et le déployement de nouveaux ordonnanceurs au sein d'environnements d'exécution.

Ne cherchant pas à déployer nos modèles d'ordonnanceur, nous avons choisi d'employer un modèle plus adapté à l'analyse : le modèle des automates temporisés. D'abord, car il existe de nombreux outils d'analyse capables de manipuler ce type de modèle, que ce soit des simulateurs ou des *model-checkers* tels que TIMES [FMPY06, AFM+03] ou UPPAAL [HU01, AD90, BDL04]. Il existe, d'ailleurs, plusieurs approches proposant la vérification d'ordonnanceurs temps réel avec ce type de modèles [AGS02, SGSS06, IKL+99, Nas07]. Enfin, plusieurs langages de conception ou d'architecture adaptés aux systèmes temps réel proposent l'usage de modèles proches ; c'est le cas d'UML (cf. les *statecharts* UML [Dru06]) mais aussi d'AADL version 2 avec son annexe comportementale [SAE07].

```
entry := sections
sections := section {sections}
section := program_section | automaton_section
end_section := "end" "section" ";"
program_section := program_section_type [identifier] ":" statements end_section
automaton_section := "automaton_section" [identifier] ":" states transitions end_section
program_section_type := "start_section" | "priority_section"
  | "election_section" | "task_activation_section" | "check_resource_section" ...

states := state {states}
transitions := transition {transitions}
state := identifier  ":" "state" ";"
  | identifier  ":" "initial_state" ";"
transition  := "transition" identifier "==>" "[" [expression],[assignment_stat],
   [synchronization] "]" "==>" identifier ";"
synchronization := identifier '!'  | identifier "?"

statements := statement {statements}
statement := put_stat | assignment_stat | declare_stat | while_stat
           | for_stat | if_stat | return_stat | random_stat
put_stat := "put" "(" identifier {, expression} ")" ";"
declare_stat := identifier ":"  data_type [ ":=" expression ] ";"
assignment_stat := identifier ":=" expression  ";"
if_stat := "if" expression "then" statements [  "else" statements ] "end" "if" ";"
return_stat := "return" expression  ";"
for_stat := "for" identifier "in" ranges "loop" statements "end" "loop" ";"
while_stat := "while" expression "loop" statements "end" "loop" ";"
random_stat :=  law "(" identifier {"," expression} ")" ";"

data_type := scalar_data_type | vectorial_type
vectorial_type := "array" "(" ranges ")" "of"  scalar_data_type
ranges := "tasks_range"  | "buffers_range" |  "messages_range" ...
scalar_data_type := "boolean"  |  "integer"  |  "random" ...

law := "uniform" | "exponential" | "laplace_gauss" | ...
operator := "and"  | "or"  | "mod" | "<" | ">" | "<=" | ">=" ...
expression :=  expression operator expression
  | "max_to_index"  "("  expression  ")"
  | "lcm"  "("  expression  "," expression ")"
  | ...
...
```

FIG. 4.1: *Extrait de la syntaxe concrète BNF du langage Cheddar*

Nous proposons un processus d'utilisation associé à ce langage afin que l'utilisateur puisse aisément concevoir et tester son ordonnanceur. Par la suite, grâce à l'utilisation d'un méta-environnement, l'utilisateur peut générer son outil de simulation, qui après compilation, est capable de conduire des simulations sur des modèles de taille importante. Ce processus d'ingénierie dirigé par les modèles est mis en œuvre grâce à Platypus [PR98]. Platypus est un méta-environnement basé sur l'utilisation de la technologie STEP, notamment le langage de modélisation EXPRESS [ISO94, ISO04a].

Ce chapitre est organisé de la façon suivante. Dans la partie 4.1, nous présentons le langage de modélisation proposé dans le cadre du projet Cheddar. La partie 4.2 illustre ce langage par différents exemples d'ordonnanceur. Dans la suite de ce chapitre, nous parlerons indifféremment de modèle d'ordonnanceur ou de programme. Par la suite, nous décrivons dans la partie 4.3 le processus qui permet au concepteur d'architecture d'employer ce langage. La partie 4.4 décrit le processus dirigé par les modèles qui permet de construire les outils de simulation à partir d'un modèle d'ordonnanceur. Enfin, nous concluons dans la partie 4.5.

4.1 Un langage pour la modélisation d'ordonnanceurs temps réel

Le langage spécifique de Cheddar doit permettre de modéliser le fonctionnement d'ordonnanceurs temps réel, de tâches ainsi que les interactions avec leur environnement, c'est à dire les autres entités de l'architecture à analyser.

Dans ce contexte, décrire le comportement d'un ordonnanceur temps réel consiste à modéliser :

1. Les opérations arithmétiques et logiques de l'ordonnanceur, qui décrivent, par exemple, comment calculer la priorité des tâches, comment sélectionner une tâche parmi un ensemble de tâches prêtes, ... Il s'agit essentiellement de traitements à effectuer sur les attributs de certaines entités de l'architecture telles que les tâches, les processeurs ou les ressources partagées.

2. Les contraintes temporelles et les synchronisations entre des entités telles que les tâches et les ordonnanceurs de l'architecture à analyser. Il s'agit ici d'exprimer comment et quand ces entités doivent collaborer afin de partager les différentes ressources. On cherche ici à modéliser, par exemple, quand un ordonnanceur doit réveiller une tâche ou comment deux ordonnanceurs doivent coopérer afin de mettre en œuvre un ordonnancement hiérarchique [RH02, RH03, BHW03, ISO95, PUZ+06].

Le langage Cheddar est donc défini par l'association de deux langages spécifiques. La partie algorithmique est exprimée par un sous ensemble d'Ada 95 et les contraintes temporelles et de synchronisation par un modèle d'automates temporisés. Un extrait de la syntaxe concrète du langage est donné dans la figure 4.1. L'ensemble des attributs et entités du modèle de données de Cheddar (cf. figure 2.9) font aussi partie de ce langage car ils constituent des mots clés du langage. Sa syntaxe concrète est décrite en détail dans le guide d'utilisation de Cheddar [Sin07a].

4.1.1 Expression des opérations arithmétiques et logiques

Cette partie du langage permet l'expression des opérations arithmétiques et logiques sur les données de simulation. Les données de simulation sont associées aux entités du modèle à analyser (ex : priorité d'une tâche, quantum d'un processeur). Les instructions arithmétiques et logiques sont organisées en sous-programme appelés *sections* dans l'environnement Cheddar. Ces sous-programmes sont typés et peuvent être nommés. Il existe principalement trois types de sous-programmes :

- Des sous-programmes qui permettent la déclaration et l'initialisation de nouvelles données de simulation. Ces sous-programmes sont nommés **start_section**.
- Des sous-programmes qui permettent l'exécution d'opérations arithmétiques et logiques sur des données de simulation à chaque unité de temps simulé. Ces sous-programmes sont nommés **priority_section**.
- Enfin, des sous-programmes qui permettent, à chaque unité de temps simulé, de sélectionner une tâche parmi un ensemble de tâches prêtes en fonction d'une donnée de simulation. Ce sont les sous-programmes **election_section**.

Le langage définit un certain nombre de types, d'opérateurs et d'instructions spécifiques au domaine de l'ordonnancement temps réel.

La règle *entry* de la figure 4.1 spécifie qu'un programme Cheddar est un ensemble de sous-programmes.

Les règles *statement*, *operator* et *expression* décrivent l'ensemble des instructions et des opérateurs mis à disposition par ce langage. On y retrouve des instructions et opérateurs classiques tels que l'itération, la conditionnelle ou l'affectation.

D'autres instructions ou opérateurs plus spécifiques au domaine tels que *return*, *lcm*, *max_to_index* ou $uniform/exponential$ y sont aussi proposés.

Ainsi, l'instruction *return* permet d'élire une tâche conformément à une donnée de simulation.

Les instructions $uniform/exponential$ paramètrent les variables aléatoires nécessaires au simulateur.

L'opérateur *lcm* permet de calculer le plus petit commun multiple (opérateur usuel dans les tests de faisabilité). Enfin, l'opérateur *max_to_index* recherche la plus grande valeur dans un vecteur stockant une donnée de simulation pour chaque entité. Cette catégorie d'opérateur permet d'implanter, par exemple, la recherche de la tâche ayant la priorité maximale de l'architecture à analyser.

Comme Ada, le langage de Cheddar est typé. Les types usuels d'Ada tels que *integer*, *boolean*, *double* ou *string* sont disponibles. Là aussi, des types associés au domaine de la simulation pour l'ordonnancement temps réel sont mis à la disposition de l'utilisateur. C'est notamment le cas du type *random* qui permet de définir des données de simulation dont les valeurs sont conformes à des lois de probabilité telles que les lois de Poisson, Uniforme ou de Laplace-Gauss.

Cette première partie du langage est déjà suffisante pour exprimer de nombreux ordonnanceurs usuels, à condition de faire l'hypothèse d'une synchronisation fixe entre certaines des entités de l'architecture telles que les tâches et les ordonnanceurs. La figure 4.2 propose un modèle décrivant cette synchronisation fixe. Avec ce modèle, on suppose qu'un ordonnanceur exécute successivement trois traitements à chaque unité de temps simulé :

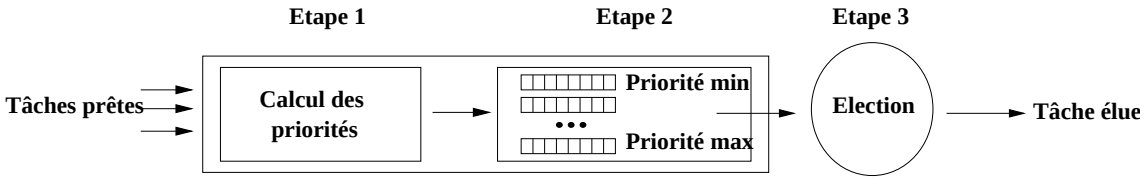

FIG. 4.2: *Fonctionnement d'un ordonnanceur*

1. Le premier traitement consiste à calculer la priorité des tâches. Si l'algorithme d'ordonnancement est à priorité fixe, cette première étape est ignorée. Dans le cas contraire, la priorité de toutes les tâches prêtes est mise à jour.
2. Par la suite, les tâches sont rangées dans un ensemble de files d'attente. Un modèle usuel consiste à considérer l'existence d'une file d'attente pour chaque niveau de priorité. C'est le modèle d'ordonnancement choisi dans certaines normes telles que POSIX 1003.1b [Gal95] ou Ada 2005 [BW07] et/ou implanté par différents systèmes d'exploitation tels que Solaris [Vah96]. La file d'attente associée au niveau de priorité n stocke donc toutes les tâches prêtes dont le niveau de priorité actuel est n. En général, la position de la tâche dans la file d'attente est relative à l'instant où cette tâche a basculé dans l'état prêt ou dans le niveau de priorité n.
3. Finalement, durant la phase d'élection, l'ordonnanceur sélectionne la file d'attente de plus forte priorité ayant au moins une tâche prête. Si la file d'attente sélectionnée contient plusieurs tâches, une politique de choix est appliquée. Selon le standard ou le système d'exploitation, une politique de choix peut être associée à chaque file d'attente ou à chaque tâche. La politique permet de choisir la tâche à exécuter parmi toutes les tâches prêtes de même niveau de priorité. De nombreuses politiques ont été proposées par le passé : FIFO, *Round-Robin*, EDF, préemptif ou non (cf. standards POSIX 1003.1b ou Ada 2005), ...

Ce modèle d'exécution permet de simuler facilement des règles de tri élémentaires [PI76] avec le langage spécifique de Cheddar. Ainsi, pour chaque unité de temps simulé, à partir d'un programme Cheddar donné, le simulateur exécute les trois étapes de la figure 4.2.

D'abord un sous-programme *priority_section* est invoqué pour chaque tâche de l'architecture à simuler. Ce premier traitement correspond à l'étape 1 de la figure 4.2.

L'étape 2 est directement implantée dans le simulateur. Le simulateur propose actuellement une implémentation des politiques $SCHED\_FIFO$, $SCHED\_RR$ et $SCHED\_OTHER$ du standard POSIX 1003.1b. Avec la politique $SCHED\_FIFO$, le processeur est alloué à la tâche en tête de file d'attente. Une tâche reste en tête de file d'attente tant qu'elle est prête, qu'elle n'est pas terminée ou qu'elle ne demande pas explicitement à libérer le processeur. Lorsqu'une tâche redevient prête ou qu'elle demande explicitement à libérer le processeur, elle est insérée en queue de file d'attente. La politique $SCHED\_RR$ fonctionne comme $SCHED\_FIFO$ mais exploite un quantum. Un quantum est une borne maximale sur le temps que la tâche en cours d'exécution peut conserver le processeur. À l'expiration du quantum, la tâche en cours d'exécution, qui est en tête de file d'attente est déplacée en queue. La politique $SCHED\_RR$ implante donc une politique de *Round-Robin*. Le standard

POSIX 1003.1b ne normalise pas le fonctionnement de la politique *SCHED_OTHER*. La politique *SCHED_OTHER* implantée dans le simulateur fournit une discipline d'ordonnancement adaptée aux applications temps partagées : la priorité d'une tâche est inversement proportionnelle au temps processeur qu'elle a consommé.

Finalement, l'étape 3 est assurée par l'exécution d'un sous-programme de type *election_section*.

4.1.2 Synchronisation et contraintes temporelles

La deuxième partie d'un modèle d'ordonnanceur temps réel dans l'environnement Cheddar est un ensemble d'automates temporisés modélisant les contraintes temporelles et les synchronisations entre les entités de l'architecture à analyser.

La première partie du langage décrite dans le paragraphe 4.1.1 est suffisante pour exprimer un nombre important d'ordonnanceurs pour lesquels les contraintes de synchronisation entre les entités sont fixes. Il existe toutefois des ordonnanceurs qui nécessitent l'expression de synchronisations complexes. C'est notamment le cas des ordonnanceurs hiérarchiques. Une architecture utilise un ordonnancement hiérarchique lorsque plusieurs entités se coordonnent pour le partage d'une ressource telle qu'un processeur.

Le modèle d'automate utilisé par Cheddar est un modèle simplifié de celui utilisé dans la plateforme UPPAAL [HU01, AD90, BDL04]. UPPAAL est un ensemble d'outils de *model-checking* et de simulation pour des automates temporisés étendus par des variables. Le modèle d'automate d'UPPAAL permet de définir des horloges, des gardes et des opérations de synchronisation entre les automates [LPY97, BDL04].

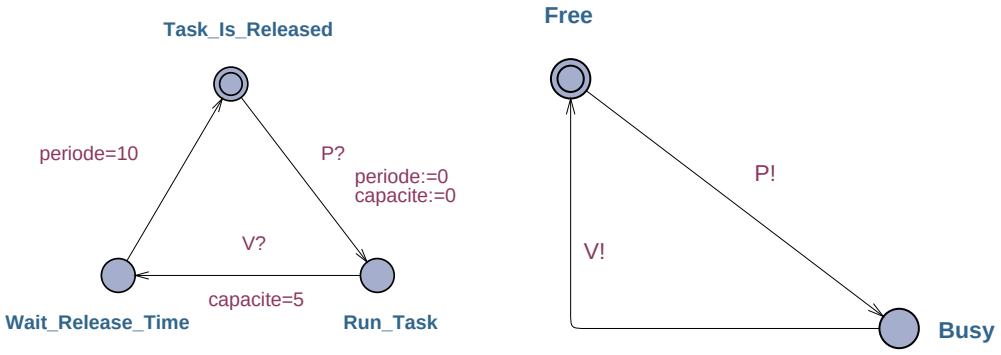

FIG. 4.3: *Exemple d'un modèle UPPAAL*

La figure 4.3 présente un exemple d'automate UPPAAL. L'automate à gauche de cette figure modélise une tâche périodique simplifiée. Cette tâche possède une période de 10 unités de temps. À chaque réveil, la tâche alloue une ressource partagée, exécute un traitement pendant 5 unités de temps, puis, finalement, libère la ressource partagée. La ressource partagée (automate de droite) est accédée via un sémaphore. L'allocation atomique de la ressource s'effectue par la primitive P. La libération atomique s'effectue par la primitive V. La modélisation de ces primitives P et V s'effectue par une synchronisa-

tion (ou rendez-vous) entre les deux automates : les opérateurs ! et ? permettent la spécification de ce rendez-vous. L'automate de gauche est temporisé par une horloge (horloge *periode*). Cette horloge est initialisée à zéro lors du réveil de la tâche (transition sortante de l'état $Task\_Is\_Released$), puis, est employée dans une garde (transition sortante de l'état $Wait\_Release\_Time$). Cette garde permet de conditionner le franchissement de la transition afin de modéliser le réveil périodique de la tâche.

Ainsi, dans Cheddar, un automate temporisé est une machine à états finis étendue par des variables permettant de modéliser le temps et les contraintes de synchronisation entre les différentes entités de l'architecture à analyser. Un ordonnanceur est ainsi modélisé par un réseau d'automates temporisés. Un sous-programme Cheddar peut être exécuté lors du franchissement d'une transition afin d'exprimer les opérations sur les données de simulation de l'algorithme d'ordonnancement. Cette exécution de sous-programme est représentée par une opération de synchronisation sur l'automate.

4.2 Quelques exemples de modèles d'ordonnanceur

Regardons maintenant quelques exemples de modèles d'ordonnanceur exprimés avec le langage spécifique de Cheddar. Nous proposons d'abord un modèle pour Rate Monotonic et Earliest Deadline First [CDKM02], les deux algorithmes classiquement étudiés dans la théorie de l'ordonnancement temps réel. Comme nous l'avons vu dans le chapitre 2, ces premiers modèles sont conformes à la synchronisation fixe décrite dans la figure 4.2. Puis, nous montrons comment modéliser un ordonnancement hiérarchique tel que celui proposé par le standard ARINC 653 [Ari97].

4.2.1 Ordonnanceurs temps réel classiques

```
election_section :
   return min_to_index(tasks.period);
end section;
```

FIG. 4.4: *Programme Cheddar modélisant Rate Monotonic*

Une modélisation possible pour Rate Monotonic est proposée par la figure 4.4. Rappelons qu'avec cet algorithme, les priorités sont fixées statiquement, proportionnellement aux périodes : seul un sous-programme *election_section* est donc nécessaire dans ce cas. Ce sous-programme spécifie simplement que la tâche à sélectionner à chaque unité de temps pendant la simulation de l'architecture doit être la tâche prête dont la période est la plus petite. Cette sélection s'exprime par l'usage de l'opérateur *min_to_index* qui examine une variable de simulation donnée (ici, la variable *tasks.period*) pour toutes les tâches éligibles et renvoie l'identificateur de la tâche qui possède la valeur la plus petite pour cette variable. *tasks.period* est un vecteur prédéfini dans le langage spécifique de Cheddar. En fait, chaque attribut des entités spécifiées dans le modèle de données de Cheddar est ainsi accessible par un programme Cheddar (cf. diagramme de classe de la figure 2.9). On peut donc considérer ces attributs comme des mots clés du langage. Cette association du modèle de données au langage de modélisation

d'ordonnanceur permet d'associer le modèle d'architecture au modèle qui décrit le comportement de l'ordonnanceur.

```
start_section :
        dynamic_priority : array (tasks_range) of integer;
end section;

priority_section :
        dynamic_priority := tasks.start_time
                + ((tasks.activation_number-1)*tasks.period)
                + tasks.deadline;
end section;

election_section :
        return min_to_index(dynamic_priority);
end section;
```

FIG. 4.5: *Programme Cheddar modélisant Earliest Deadline First*

Le second exemple propose un modèle pour Earliest Deadline First. Cette fois, l'algorithme est à priorité dynamique; il est donc nécessaire de calculer pendant la simulation la priorité des tâches. Ce traitement est modélisé à l'aide d'un sous-programme de type *priority_section*. Ce sous-programme (cf. figure 4.5) montre comment calculer, pour chaque unité de temps simulé, l'échéance conformément à la règle de tri Earliest Deadline First. Cette échéance est mémorisée dans la variable *dynamic_priority*. Cette fois, il s'agit d'un calcul vectoriel portant sur des données de simulation extraites du modèle de données de Cheddar ainsi que des variables de simulation déclarées par l'utilisateur au sein d'un sous-programme de type *start_section*. La phase d'élection, spécifiée par le sous-programme *election_section*, consiste à sélectionner la tâche de plus courte échéance; c'est à dire de plus petite priorité dynamique.

Nous avons montré que des algorithmes d'ordonnancement tels que Rate Monotonic ou Earliest Deadline First pouvaient être aisément exprimés avec le langage spécifique de Cheddar. La synchronisation des tâches et des ordonnanceurs décrite par la figure 4.2 est compatible avec le fonctionnement de ces algorithmes. Leur modélisation ne requiert donc que l'expression du calcul des priorités et de la règle de tri (ou d'élection). En pratique, l'environnement de simulation propose une synchronisation par défaut pour le programme 4.5. La figure 4.6 propose un modèle pour Earliest Deadline First qui explicite par un automate cette synchronisation par défaut. La figure 4.7 présente une représentation graphique de cet automate.

La définition d'un automate est effectuée par une section spécifique (*automaton_section*). Les opérations de synchronisation *elect*1! et *prio*1! sont des opérations de synchronisation avec le moteur de simulation. Ces synchronisations déclenchent une exécution séquentielle des sous-programmes *priority_section* et *election_section* dont les noms sont *prio*1 et *elect*1.

Enfin, l'automate de la figure 4.8 représente le fonctionnement du moteur de simulation pour ce modèle d'ordonnanceur. Cet automate n'est pas spécifié par l'utilisateur. Les synchronisations de ce dernier automate décrit finalement comment tous les automates d'un modèle d'ordonnanceur coopérent

```
start_section start1 :
        dynamic_priority : array (tasks_range) of integer;
end section;

priority_section prio1 :
        dynamic_priority := tasks.start_time
                + ((tasks.activation_number-1)*tasks.period)
                + tasks.deadline;
end section;

election_section elect1 :
        return min_to_index(dynamic_priority);
end section;

automaton_section automaton1 :
  Compute_Priority : state;
  Choose_Task : initial_state;

  transition Compute_Priority ⟹ [ , , prio1!] ⟹ Choose_Task;
  transition Choose_Task ⟹ [ , , elect1!] ⟹ Compute_Priority;
end section;
```

FIG. 4.6: *Programme Cheddar modélisant Earliest Deadline First et explicitant la synchronisation entre les sous-programmes*

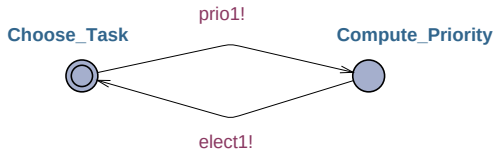

FIG. 4.7: *Automate modélisant la synchronisation fixe implémentée par le simulateur*

pour le calcul d'un ordonnancement.

En pratique, le moteur de simulation de Cheddar est un simulateur à événements discrets. L'interprétation des automates temporisés, dont le comportement est par nature continu, est discrétisée afin d'en assurer le couplage avec le moteur de simulation. Par ailleurs, le simulateur a aussi la charge d'assurer l'incrémentation cohérente des horloges des modèles.

Nous décrivons maintenant un exemple plus significatif qui montre comment un réseau d'automates temporisés permet au concepteur d'élaborer des modèles d'ordonnancement bien plus complexes.

4.2.2 Ordonnancement hiérarchique : exemple avec le standard ARINC 653

Les ordonnanceurs hiérarchiques ont été initialement proposés dans le contexte des systèmes temps partagés afin de permettre à l'utilisateur de définir des politiques d'ordonnancement adaptées aux besoins de ses applications [KL88, Vah96]. Un ordonnancement hiérarchique offre une solution séduisante pour l'assemblage d'applications ayant des besoins différents en ressource. Par exemple, les différentes formes de serveurs (serveurs différés, sporadiques, à scrutation, ...) qui rendent possible l'ordonnancement de tâches apériodiques par Rate Monotonic proposent différentes formes d'ordonnancement

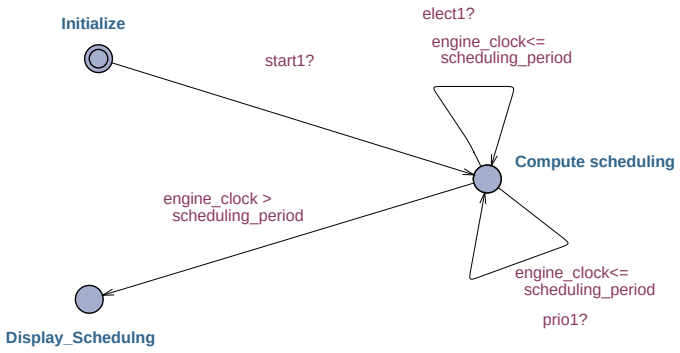

FIG. 4.8: *Automate modélisant le moteur de simulation pour l'exemple Earliest Deadline First*

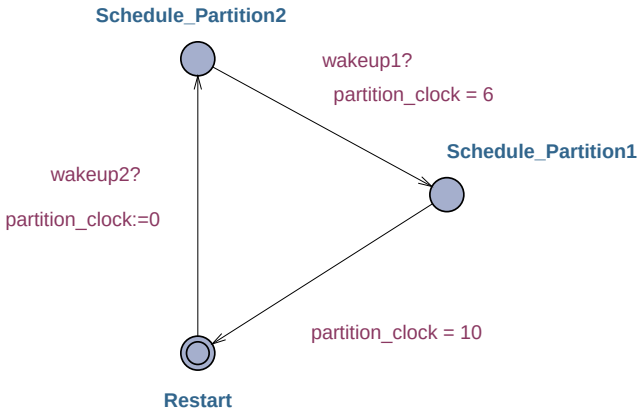

FIG. 4.9: *Automate modélisant l'ordonnanceur de partitions*

hiérarchique [SSL89]. Par ailleurs, plusieurs normes et standards offrent des services d'ordonnancement hiérarchique. Les modèles d'ordonnancement proposés par POSIX 1003.1b ou Ada 2005 en sont des exemples [RH02, RH03, BHW03, ISO95, PUZ+06].

Malheureusement, la mise en œuvre d'outils d'analyse de performances est difficile pour des architectures utilisant ce type d'ordonnancement. D'abord parce qu'il existe une très large variété d'algorithmes d'ordonnancements hiérarchiques. Ces algorithmes d'ordonnancement proposent des moyens parfois très différents pour synchroniser les entités de l'architecture [ABLL92] ou pour exprimer leurs besoins et leurs comportements [LMD04, RS01]. Il parait donc difficile de fournir un outil de simulation qui implante un ensemble significatif de ces algorithmes. C'est d'autant plus vrai que bien souvent, un algorithme d'ordonnancement hiérarchique propose une solution ad-hoc à un problème très spécifique. Par ailleurs, même si de nombreux travaux proposent des méthodes analytiques pour ces ordonnanceurs [AP04, HP03, SL03, DB05], dans la majorité des cas, un concepteur d'architecture qui emploie un ordonnanceur hiérarchique ne disposera pas de méthode analytique pour en étudier les performances. Comme il est peu probable que le concepteur élabore une méthode analytique pour une architecture donnée, compte tenu du coût et de la complexité d'un tel travail, la vérification par simulation de ce

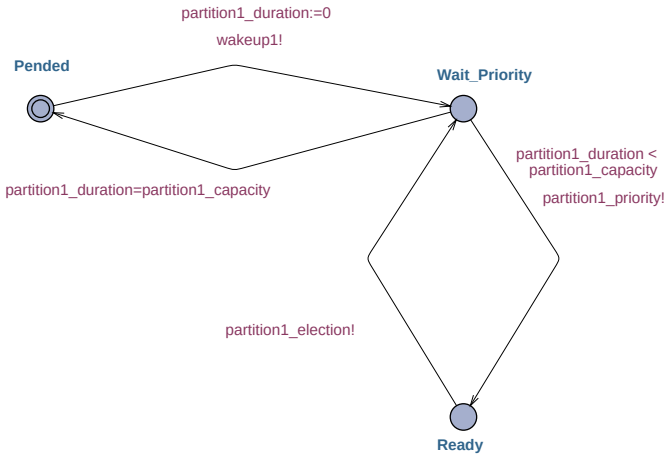

FIG. 4.10: *Automate modélisant l'ordonnanceur de la partition 1*

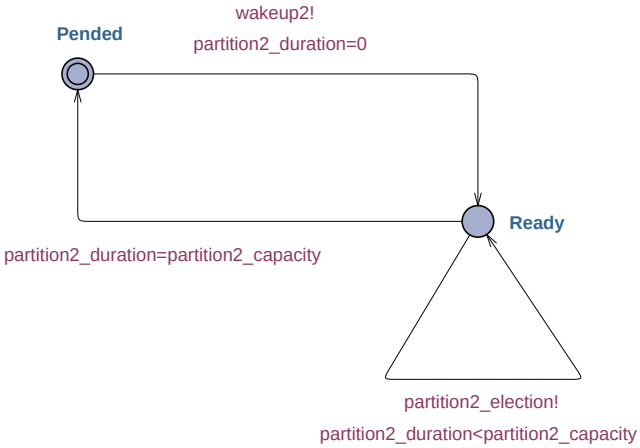

FIG. 4.11: *Automate modélisant l'ordonnanceur de la partition 2*

type d'architecture reste une solution aujourd'hui incontournable.

Regardons maintenant comment un ordonnancement hiérarchique tel que celui proposé par ARINC 653 peut être exprimé avec le langage spécifique de Cheddar, modélisation réalisée en vue d'une vérification par simulation.

Une architecture ARINC 653 est un ensemble d'applications éventuellement réparties sur plusieurs processeurs. Une application est appelée une partition. Dans la norme ARINC 653, la partition est l'abstraction qui permet la mise en œuvre des mécanismes d'isolation mémoire et temporelle. Chaque partition est composée d'un ensemble de tâches. Le partage du processeur est alors effectué grâce à un ordonnancement hiérarchique à deux niveaux :

1. Les partitions sont ordonnnancées cycliquement. Pour chaque cycle, l'ordonnancement de partitions stipule quand et pour combien de temps une partition est active, c'est-à-dire quand elle peut exécuter une de ces tâches. Ce premier niveau d'ordonnancement est statique : l'ordonnancement

```
start_section start1 :
  partition1_capacity : integer := 4;
  partition1_duration : clock := 0;
  quantum : integer :=2;
  my_prio : array (tasks_range) of integer;
  my_prio:=0;
end section;

priority_section partition1_priority :
   quantum:=quantum-1;
   if quantum = 0
      then quantum:=2;
           my_prio(previously_elected):=my_prio(previously_elected)+1;
   end if;
end section;

election_section partition1_election :
   return min_to_index(my_prio);
end section;

automaton_section partition1_scheduler :
  Pended : initial_state;
  Ready : state;
  Wait_Priority : state;

  transition Pended ⟹
     [ , partition1_duration:=0;,wakeup1!] ⟹ Wait_Priority;

  transition Wait_Priority ⟹
     [partition1_duration<partition1_capacity,
          , partition1_priority!] ⟹ Ready;

  transition Ready ⟹
     [ , , partition1_election!] ⟹ Wait_Priority;

  transition Wait_Priority ⟹
     [partition1_duration=partition1_capacity , , ] ⟹ Pended;
end section;
```

FIG. 4.12: *Programme Cheddar modélisant l'ordonnanceur Round-Robin des tâches de la partition 1*

est calculé lors de la phase de conception du système. C'est donc un ordonnancement hors-ligne.

2. Le deuxième niveau d'ordonnancement est celui des tâches. Chaque partition ordonnance ses tâches avec un ordonnanceur à priorités fixes. Les tâches d'une partition donnée peuvent être exécutées uniquement pendant les instants où leur partition est active. Ce deuxième ordonnancement est donc en-ligne.

Ce modèle d'ordonnancement d'ARINC 653 peut être représenté par un ensemble de programmes Cheddar. Une représentation graphique de la partie synchronisation de cet ordonnanceur hiérarchique est donnée par les figures 4.9, 4.10 et 4.11. La représentation textuelle d'un de ces programmes est donnée en figure 4.12. Le lecteur pourra se référer à [SP07a] pour consulter l'intégralité des programmes Cheddar de cet exemple. Enfin, l'automate de la figure 4.13 représente le fonctionnement du moteur de simulation pour l'ordonnanceur ARINC 653.

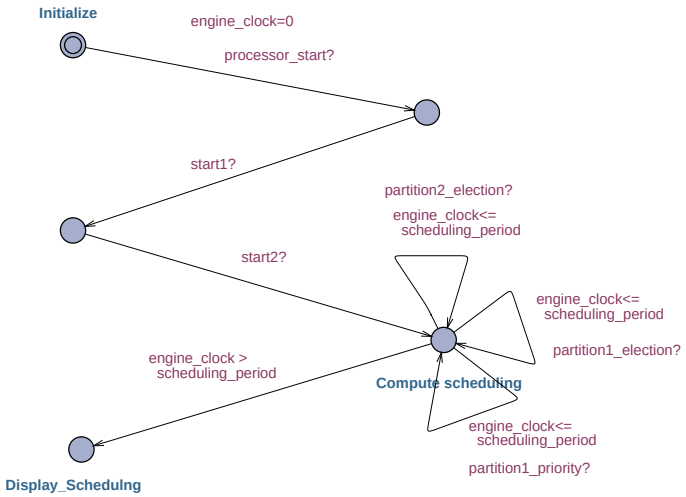

FIG. 4.13: *Automate UPPAAL du moteur de simulation pour l'ordonnanceur ARINC 653*

L'automate de la figure 4.9 modélise quand les partitions sont actives : c'est le premier niveau d'ordonnancement ARINC 653. Dans cet exemple, nous avons deux partitions. Le cycle d'ordonnancement des partitions est constitué de 4 unités de temps consacrées à la partition 1 suivies de 6 unités de temps pour la partition 2. Cet automate permet d'assurer l'isolation temporelle de ces deux partitions afin que les tâches d'une partition n'interfèrent pas sur l'exécution des tâches de la seconde partition.

Les automates des figures 4.10 et 4.11 modélisent l'ordonnancement des tâches des deux partitions : nous modélisons ici le deuxième niveau d'ordonnancement ARINC 653. La partition 2 ordonnance un jeu de tâches périodiques et critiques selon des priorités fixes alors que la partition 1 exécute un ensemble de tâches non critiques selon un algorithme de type *Round-Robin*. Les automates modélisant les ordonnanceurs de tâches des partitions 1 et 2 sont constitués des états suivants :

1. L'état *Pended* indique que la partition est inactive : elle ne peut donc pas utiliser le processeur afin d'exécuter une de ses tâches.
2. Les états $Wait\_Priority$ et *Ready* sont les états modélisant une partition comme active. Une partition dans l'un de ces états peut exécuter une de ses tâches conformément à son ordonnanceur. L'état $Wait\_Priority$ est un état intermédiaire pendant lequel l'ordonnanceur de tâches calcule une priorité pour chaque tâche. Ainsi, le sous-programme $partition1\_priority$ exécuté lors du franchissement de la transition sortante de $Wait\_Priority$ permet de calculer les priorités des tâches de la partition 1. L'état *Ready* permet de sélectionner la tâche à exécuter pendant la prochaine unité de temps.

En dehors des automates temporisés modélisant les synchronisations, un programme Cheddar peut aussi contenir des sous-programmes exprimant les opérations logiques et arithmétiques d'un ordonnanceur afin, par exemple, de calculer une priorité, d'implanter une règle de tri, d'effectuer l'initialisation de variables de simulation. Dans notre exemple de modèle ARINC 653, le sous-programme *start*1 du programme 4.12 montre comment effectuer l'initialisation de variables de simulation ; le

sous-programme *partition*1*_priority* spécifie comment calculer une priorité aux tâches ; enfin le sous-programme *partition*1*_election* montre comment sélectionner une tâche conformément à sa priorité.

4.3 Processus d'utilisation d'un programme Cheddar

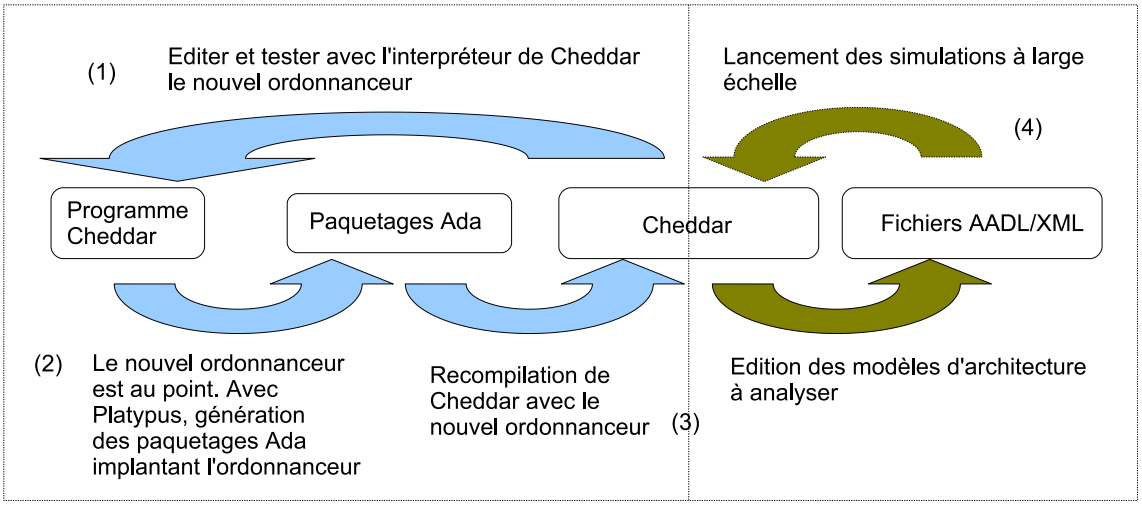

FIG. 4.14: *Processus de construction et d'utilisation d'un simulateur avec Cheddar*

Dans la partie précédente, nous avons expliqué comment modéliser un ordonnanceur avec le langage spécifique de Cheddar. Ces modèles d'ordonnanceur permettent d'effectuer des simulations d'architectures comportant des ordonnanceurs non encore implantés dans le canevas logiciel de Cheddar. Avec ce langage, l'utilisateur exploite le processus décrit par la figure 4.14 pour la mise en œuvre de ses simulations :

1. L'utilisateur doit d'abord décrire avec le langage de Cheddar l'ordonnanceur à implanter. Il lui est alors possible de mettre au point son modèle d'ordonnanceur grâce à un interpréteur implanté dans le canevas logiciel de Cheddar et qui lui permet de tester rapidement le comportement de son programme par simulation.
2. Une fois son ordonnanceur au point, grâce à Platypus, il peut transformer le programme Cheddar en un ensemble de paquetages Ada pouvant être automatiquement intégré au canevas logiciel. Dans la phase de transformation du modèle vers le code Ada, il est possible d'envisager la spécification de contraintes afin d'adapter le code généré aux futures simulations (exemple : adaptation de l'empreinte mémoire ou du temps de réponse des simulations selon les caractéristiques de l'architecture à analyser).
3. Les paquetages Ada générés sont compilés et intégrés dans Cheddar afin de produire une nouvelle version du canevas logiciel. Par la suite, l'utilisateur peut exécuter des simulations de grande taille comme si son ordonnanceur avait été implanté manuellement dans Cheddar.

4. La dernière étape consiste à employer le nouvel environnement Cheddar pour l'analyse de performance d'une architecture. Cette architecture peut être décrite soit avec AADL, soit avec le langage d'architecture spécifique de Cheddar.

   Cette phase d'analyse par simulation consiste :

   (a) À calculer sur un intervalle de temps donné l'ordonnancement des tâches.

   (b) Puis, à réaliser une analyse des temps de réponse des tâches à partir de l'ordonnancement préalablement calculé, ceci afin de finalement vérifier le respect des échéances des tâches.

   Bien que généralement une simulation de ce type ne conduit pas à l'établissement d'une preuve des propriétés recherchées, dans le cadre de la théorie de l'ordonnancement temps réel, il arrive que le calcul d'une telle simulation conduise à une simulation exhaustive de système. En effet, lorsque la simulation traite d'une architecture composée de tâches périodiques ordonnancées par un algorithme déterministe, il est alors possible de calculer l'ordonnancement sur la période d'étude.

   La période d'étude est une séquence d'ordonnancement qui se répète indéfiniment. La période d'étude peut être calculée comme suit [LM80, CDKM02] :

$$[0, max(\forall i : S_i) + 2 \cdot PPCM(\forall i : P_i)] \tag{4.1}$$

   où P_i est la période de la tâche i et $PPCM(\forall i : P_i)$ est le plus petit commun multiple des périodes de toutes les tâches. Si le concepteur peut démontrer qu'entre les instants 0 et $max(\forall i : S_i) + 2 \cdot PPCM(\forall i : P_i)$ toutes les échéances sont respectées, alors, cela signifie que les échéances des tâches seront respectées pendant toute la durée d'exécution de l'application.

   Pour vérifier exhaustivement une architecture avec un programme Cheddar, il est donc intéressant de pouvoir s'assurer du déterminisme de l'ordonnanceur modélisé. Dans [BPR07], le lecteur trouvera une approche permettant cette vérification.

4.4 Du modèle d'ordonnanceur au logiciel de simulation

Nous décrivons maintenant comment, à partir d'un programme Cheddar modélisant un ordonnanceur, nous générons le simulateur associé. Avant de décrire ce procédé de génération de code dans la partie 4.4.2, nous donnons quelques éléments concernant la conception de l'environnement de simulation Cheddar dans la partie 4.4.1. En particulier, nous décrivons quelles sont les entités générées à partir d'un modèle d'ordonnanceur.

4.4.1 Les composants architecturaux du canevas logiciel de Cheddar

Comme le montre la figure 4.15, le canevas logiciel de Cheddar est composé de plusieurs modules principaux répartis en deux couches logicielles :

- La couche basse, pour la gestion des données. Elle est directement dépendante des types de données manipulées et des contraintes associées en termes de règles structurelles et sémantiques. Cette couche de nature générique est fortement réutilisable et évolutive.
- La couche haute liée au domaine sous-jacent de la simulation de systèmes temps réel. Bien que cette couche soit de nature très spécifique, elle est cependant évolutive de part l'utilisation d'un procédé de mise en œuvre dirigé par les modèles qui se concrétise par l'utilisation d'un générateur de code spécifique.

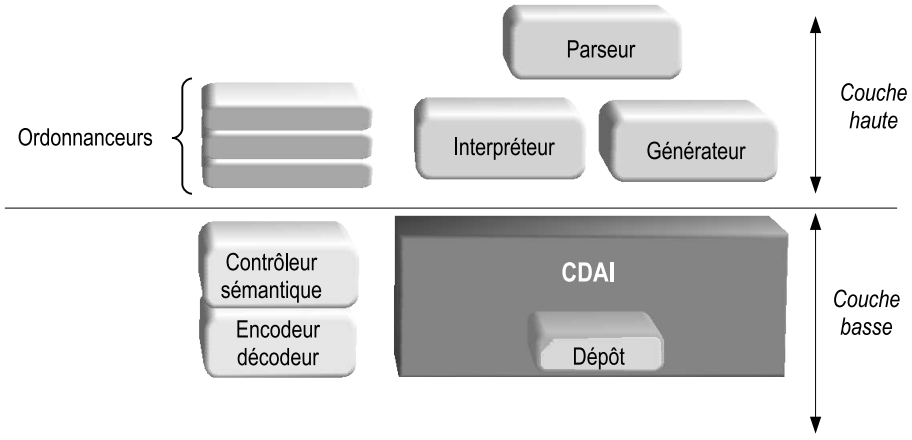

FIG. 4.15: *Architecture logicielle à deux couches du canevas logiciel de Cheddar*

4.4.1.1 La couche basse

Cheddar comprend un dépôt centralisé pour le stockage des données et des méta-données : les tâches, les processeurs ainsi que leurs paramètres sont des exemples de données. La définition d'une transition d'un automate ou une instruction de boucle sont elles des méta-données. Le dépôt est un constituant abstrait qui représente un composant physique de stockage de données. Concrètement, il peut s'agir de la mémoire ou d'une base de données.

Le dépôt est encapsulé dans un composant d'accès aux données : la CDAI[20]. 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 met en œuvre des composants additionnels, spécifiquement liés à la définition des données manipulées au travers de la CDAI :

- Le contrôleur sémantique permet la validation des données et des méta-données.
- Le composant d'encodage et de décodage assure à Cheddar une interopérabilité par échange de données. Ce composant met en œuvre un protocole standard pour l'encodage et le décodage des

[20]Cheddar Data Acces Interface.

données et des méta-données vers ou depuis des flux XML.

4.4.1.2 La couche haute

La couche haute permet la simulation de systèmes temps réel à deux niveaux :

1. Cheddar met en œuvre des algorithmes d'ordonnancement reconnus comme classiques par la communauté. Ces algorithmes d'ordonnancement peuvent être directement utilisés pour un modèle de tâches défini par l'utilisateur. Ces ordonnanceurs sont pré-intégrés sous la forme de paquetages Ada dédiés (un paquetage Ada par ordonnanceur).
2. Des ordonnanceurs spécifiques et non intégrés peuvent être programmés à l'aide du langage de Cheddar. Ces programmes sont analysés par le parseur qui, à partir d'un programme, produit les méta-données correspondantes. Ces méta-données sont des instances de la syntaxe abstraite du langage de Cheddar. Pour un modèle d'ordonnanceur, l'ensemble des méta-données produites constitue une représentation interne. L'interpréteur exploite cette représentation interne pour le calcul de la simulation.

4.4.2 Procédé dirigé par les modèles pour la mise en œuvre des couches basses et hautes

Pour la mise en œuvre des deux couches logicielles, un procédé dirigé par les modèles à deux niveaux est exploité. Comme le montre la figure 4.16, ce procédé assure une évolutivité incrémentale du système en six étapes : un ordonnanceur développé spécifiquement pour une application peut être analysé (1) ; le résultat de cette analyse est utilisé en entrée par un premier générateur de code (2) qui d'une part, produit le code du nouvel ordonnanceur (3) et qui, d'autre part, produit une nouvelle version du modèle de données de Cheddar (4). Le générateur de couche basse exploite alors ce nouveau modèle (5) pour produire une nouvelle version de la couche basse (6).

4.4.2.1 Mise en œuvre de la couche basse

Pour la couche basse, les objets manipulés, comprenant les données et les méta-données, sont spécifiés par le modèle de données de Cheddar (cf. figure 2.9) et par le méta-modèle du langage spécifique de Cheddar (cf. figure 4.1). Ces modèles intègrent la définition structurelle des hiérarchies et la définition des contraintes des entités du système. À une version particulière de ces modèles correspond une version de la couche basse puisque tous les constituants de cette couche (ensemble de paquetages Ada) sont automatiquement produits à partir de ces modèles par le générateur de couche basse. Le lecteur trouvera une présentation de cette génération de la couche basse pour le modèle de données de Cheddar dans [PS06].

4.4.2.2 Mise en œuvre de la couche haute

Pour un programme donné, le générateur d'ordonnanceur permet de traduire la représentation interne du programme en un ordonnanceur sous la forme d'un paquetage Ada spécifique intégré au

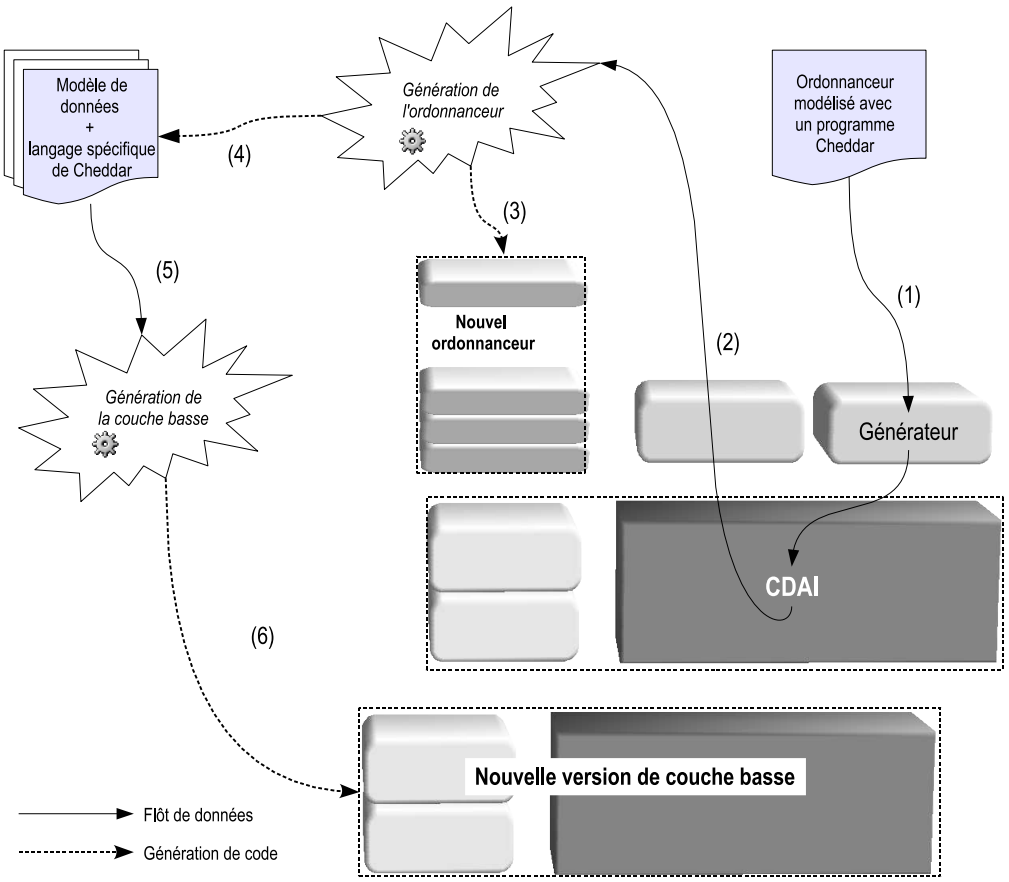

FIG. 4.16: *Évolution incrémentale de Cheddar*

canevas logiciel de Cheddar. Cette traduction met en jeu deux processus de génération. Tout d'abord, est généré le code de l'ordonnanceur comprenant 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'entités complémentaires (nouveau paquetage Ada pour le nouvel ordonnanceur) et la modification d'entités déjà existantes (nouveaux attributs pour des entités de type tâches, processeurs, ...) . 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 logiciel de 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 produite.

4.4.2.3 Modélisation avec EXPRESS du modèle de données de Cheddar ainsi que de son langage spécifique

Classiquement, un procédé dirigé par les modèles met en œuvre un langage de modélisation pour la spécification des modèles et des méta-modèles. Les règles de traduction peuvent être exprimées avec le langage de modélisation choisi ou à l'aide d'autres dialectes ou modèles. Pour la description et la mise en œuvre des modèles et méta-modèles, nous utilisons les outils apportés par la technologie STEP ou norme ISO-10303 [ISO94], dont essentiellement, le langage de modélisation de données EXPRESS [ISO04a].

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 manipulant des informations.

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 librairie de concepts réutilisables pour développer des modèles de données dans différents secteurs industriels [Bou95].

Ces travaux de normalisation sont toujours très actifs. Les développements récents de la norme visent à permettre l'interopérabilité sémantique avec les outils de l'OMG notamment en proposant un méta-modèle d'EXPRESS basé sur le MOF[21] [OMG02, MOF07] et deux passerelles, l'une vers UML version 2 et l'autre vers OWL.

```
SCHEMA Processors;
  USE FROM Objects;

  ENTITY Generic_Scheduler_Ptr;
  END_ENTITY;

  ENTITY Processor
    SUBTYPE OF ( Generic_Object );
    Scheduler : Generic_Scheduler_Ptr;
    Network : STRING;
  DERIVE
    SELF\Generic_Object.Object_Type :
      Objects_Type := Processor_Object_Type;
  END_ENTITY;
END_SCHEMA;
```

FIG. 4.17: *Un extrait du modèle de données de Cheddar : le schéma Processors*

Le langage EXPRESS est un langage de modélisation orienté données. Les données sont modélisées au sein de schémas. Un schéma comprend la définition de types, d'entités et de contraintes globales. Un schéma peut réutiliser les définitions d'autres schémas. Une entité peut hériter d'une ou de plusieurs

[21] Meta Object Facility.

autres entités. Une entité comprend un ensemble d'attributs et de contraintes locales. Un attribut peut être explicite (l'attribut est valorisé explicitement), dérivé (sa valeur est calculé par une expression) ou encore inverse (sa valeur est calculée et indique la cardinalité inverse d'une association entre deux entités).

La figure 4.17 montre le schéma *Processors* extrait du modèle de données de Cheddar (cf. figure 2.9 pour le modèle de données complet). Ce schéma comprend notamment l'entité *Processor* qui spécifie ce qu'est un processeur pour Cheddar. Un processeur est un objet (*Processor* hérite de *Generic_Object*) associé à un ordonnanceur (attribut explicite *Scheduler*) et à un réseau. Le réseau est représenté par son identifiant (attribut *Network*). L'entité *Processor* comprend la redéfinition de l'attribut explicite *Generic_Object.Object_Type*, en attribut dérivé de façon à en fixer la valeur pour une instance de *Processor*. Cet attribut facilite l'introspection d'une instance du modèle de données de Cheddar.

La mise en œuvre d'un schéma EXPRESS consiste en son intégration dans un dispositif d'échange de données (cf. figure 4.18). Ce dispositif met en œuvre un format d'échange de données standard, ou format pivot [ISO01, ISO04b]. Les composants « encodeur » et « décodeur » pour le format pivot sont produits automatiquement à partir du schéma EXPRESS qui décrit les données échangées. L'encodeur permet de sérialiser les données échangées vers le format pivot alors que le décodeur effectue le travail inverse de désérialisation. Dans le cas du canevas logiciel de Cheddar, le format pivot choisi est XML. Le flux XML peut être généré par un outil de modélisation, par l'éditeur graphique de Cheddar ou par tout autre outil logiciel. Les instances contenues dans le dépôt sont donc conformes à leur type décrit dans le schéma EXPRESS. Enfin, l'interface standard d'accès aux données, la SDAI[22] [ISO98], est utilisée pour construire les composants encodeurs et décodeurs.

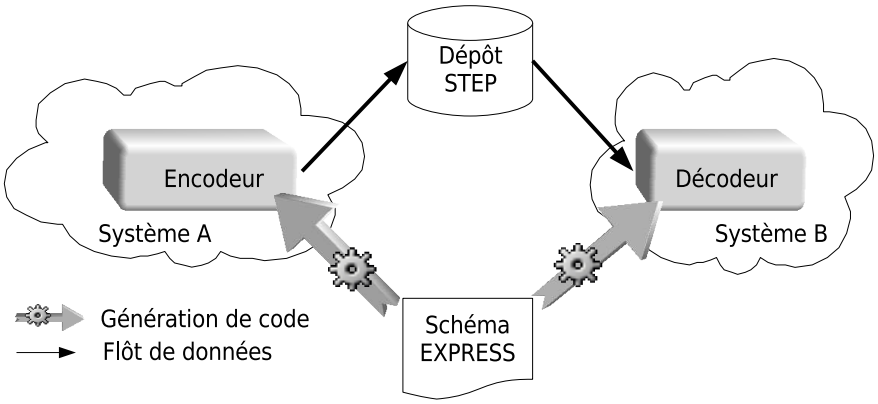

FIG. 4.18: *Processus standard d'échange de données dans STEP*

4.4.2.4 Génération de code pour les couches basses et hautes

Le processus dirigé par les modèles décrit par la figure 4.16 exploite deux générateurs complémentaires, un pour générer la couche basse et l'autre pour produire les ordonnanceurs de la couche haute.

[22]Standard Data Acces Interface.

Classiquement, dans le cadre de l'ingénierie dirigée par les modèles, la construction d'un générateur de code nécessite : 1) la déclaration d'une syntaxe abstraite associée à la déclaration des règles de traduction, 2) la génération de code proprement dite qui est mise en œuvre comme un processus de compilation. Regardons maintenant comment ces deux points sont traités avec STEP/EXPRESS.

4.4.2.4.1 Déclaration du générateur de code avec EXPRESS

```
SCHEMA Ada_For_Cheddar_Meta_Model;

 ENTITY class_in_package;
  name : STRING;
  attributes : LIST of attribute;
 DERIVE
  ads_code : STRING := class_in_package_ads_code(SELF);...
 END_ENTITY;

 ENTITY attribute;
  name : STRING;
  domain : attr_domain;
 END_ENTITY;
 ...
 FUNCTION class_in_package_ads_code(cip : class_in_package) : STRING;
 LOCAL
   code : STRING;
 END_LOCAL;
   code := 'type ' + cip.name + ' is new Generic_Object with '...
   RETURN(code);
 END_FUNCTION;

END_SCHEMA;
```

FIG. 4.19: *Extrait d'un méta-modèle simplifié pour Ada 95/Cheddar*

Pour chaque générateur de code, 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.

La figure 4.19 montre une partie de méta-modèle simplifié pour Ada 95 nommé *Ada_For_Cheddar-_Meta_Model* utilisé pour Cheddar. Les deux entités *class_in_package* et *attribute* y sont présentées :

- *class_in_package* spécifie ce qu'est une classe pour la mise en œuvre de Cheddar. Elle comprend deux attributs explicites, *name* pour le nom de la classe et *attributes*, pour la liste des variables de la classe.
- *attribute* spécifie très simplement ce qu'est un attribut avec son nom et son domaine.

La règle de génération de code *ads_code* est associée à *class_in_package*. Elle déclare comment produire le code d'une classe en Ada à partir des données accessibles depuis un *class_in_package*. Le code est en fait calculé par la fonction *class_in_package_ads_code*.

4.4.2.4.2 Mise en œuvre du générateur avec STEP

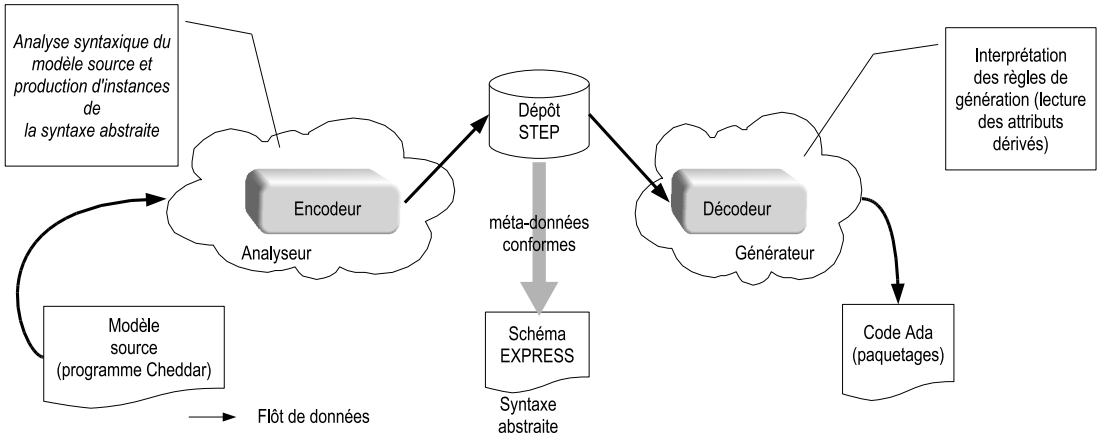

FIG. 4.20: *Génération de code mise en œuvre comme un processus d'échange STEP*

Comme le montre la figure 4.20, la mise en œuvre du générateur de code consiste en une chaîne de compilation intégrée au paradigme d'échange de données STEP [PR98]. Les données à échanger sont décrites par le modèle EXPRESS de la syntaxe abstraite. Ainsi, on dispose de l'encodeur et du décodeur de méta-données :

- L'analyseur de spécification source utilise l'encodeur pour produire les méta-données qui sont en fait des instances conformes au schéma qui spécifie la syntaxe abstraite.
- Au niveau du décodeur, pour une règle de génération de code spécifiée par un attribut dérivé, le code à produire est obtenu par lecture de cet attribut.

4.4.2.4.3 Génération de code avec Platypus

Platypus [PR98] est l'environnement STEP que nous utilisons pour construire des générateurs de code suivant la méthode Eugene [PR98]. Suivant cette méthode, le compilateur (typiquement un analyseur Lex/Yacc par exemple) est construit manuellement. Les actions sémantiques de la spécification du compilateur utilisent les points d'entrée de la SDAI pour instancier la syntaxe abstraite. Pour un langage de spécification source, la construction du compilateur peut être lourde et coûteuse. Cependant, avec Platypus, le développement d'un compilateur pour le langage source 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. Moyennant la spécification du lien de conformité entre les entités du modèle source et celle du méta-modèle (syntaxe abstraite), le processus de génération de code est intégralement automatisé.

4.5 Conclusion

Dans ce chapitre, nous avons proposé un langage spécifique adapté à la modélisation d'ordonnanceurs temps réel. Ce langage spécifique Cheddar est défini par l'association de deux parties : l'algorithmique est exprimée par un sous ensemble d'Ada 95 et les contraintes temporelles et de synchronisation par un modèle d'automates temporisés. L'association de ces deux langages est motivée par le souhait de respecter, tant que faire se peut, le principe de séparation des préoccupations [Par76, Rea89]. D'une part, nous pensons que l'analyse des propriétés d'un ordonnanceur en est simplifiée. D'autre part, ce langage offre ainsi plusieurs niveaux d'utilisation. En effet, bon nombre d'algorithmes d'ordonnancement ne requièrent pas l'usage d'automates temporisés pour l'expression de leur comportement. Dans ce cas, la description de l'ordonnanceur reste simple pour le concepteur car elle se résume à décrire comment calculer une priorité et comment cette priorité doit être utilisée pour l'élection de la prochaine tâche à exécuter. Elle permet en outre d'améliorer l'efficacité du moteur de simulation.

Associé à ce langage spécifique, nous proposons un processus dirigé par les modèles permettant, à partir d'un modèle d'ordonnanceur, d'effectuer des simulations d'une architecture à grande échelle. Ce processus traduit un modèle d'automate temporisé en un ensemble de paquetages Ada pouvant être intégré au canevas logiciel de Cheddar. L'utilisation d'un tel procédé facilite la mise en œuvre de simulateurs temps réel pour l'utilisateur. En effet, ce dernier est capable, sans comprendre les détails de conception et d'implémentation du canevas logiciel, de l'enrichir avec de nouveaux algorithmes.

Nous pensons que cette méthode est une alternative intéressante à l'utilisation de modèles et d'outils non spécifiquement conçus pour l'analyse de performances de systèmes temps réel. On pense ici, par exemple, aux réseaux de Petri [GV03] ou aux outils de simulation spécifiques à une application donnée, qui n'offrent, en général, qu'un faible niveau de réutilisation du logiciel.

Le générateur de code Ada à partir d'un modèle d'ordonnanceur est toujours en cours de développement. L'amélioration des performances obtenue par l'utilisation du générateur devra être évaluée. Nous espérons réaliser cette évaluation sur un modèle d'ordonnanceur Cheddar de taille industrielle tel que celui élaboré par la société Airbus [CCG07].

Le langage de modélisation d'ordonnanceur proposé par Cheddar soulève un certain nombre de questions concernant l'analyse statique et dynamique des ordonnanceurs ainsi spécifiés. Les automates temporisés autorisent la vérification de propriétés par *model-checking* [BBL99]. D'autre part, le sous-ensemble d'Ada utilisé dans le langage spécifique de Cheddar autorise l'application de méthodes d'analyse statique par déduction [Bar03]. Par l'application de ces techniques, nous espérons améliorer l'analyse de performances de systèmes temps réel dont les ordonnanceurs peuvent être spécifiés avec Cheddar. Cette piste de travail constitue l'une des perspectives du projet Cheddar que nous décrivons dans le chapitre suivant.

Chapitre 5

Synthèse des contributions et perspectives

Sommaire

Dans ce document, nous avons présenté les principales pistes de recherche explorées dans le cadre du projet Cheddar depuis septembre 2000. Ce projet a pour objectif d'accroître l'applicabilité de la théorie de l'ordonnancement temps réel.

Tout d'abord, nous avons réalisé des outils d'aide à la vérification. Ces outils fournissent au concepteur des méthodes analytiques et une plate-forme de simulation destinée à effectuer des analyses de performances d'une architecture temps réel. Ces outils sont basés sur la théorie de l'ordonnancement temps réel et des files d'attente.

Dans un deuxième temps, nous avons étudié comment un langage de conception d'architecture pouvait contribuer à automatiser la vérification des performances d'une architecture. L'approche considérée s'inspire de celle élaborée pour la méthode HRT-HOOD [BW94]. Elle consiste à utiliser des patrons de conception dédiés à la problématique des architectures temps réel. Ce type de patron de conception décrit et documente une solution architecturale à une famille de problèmes récurrents de concurrence. Chaque patron est associé à des méthodes analytiques appelées tests de faisabilité. Ces tests de faisabilité permettent de vérifier les performances d'un modèle conforme au patron de conception. Ils supposent qu'un ensemble d'hypothèses sont satisfaites. Le concepteur n'a pas à vérifier le respect de ces hypothèses s'il a employé un patron de conception pour élaborer son architecture. L'analyse de performance s'en trouve simplifiée. Nous avons expérimenté quatre patrons de conception avec le langage AADL : «Synchronous data-flows», «Ravenscar», «Blackboard» et «Queued Buffer».

Enfin, nous avons exploré une autre piste qui consiste à proposer un langage spécifique et des outils associés. Ceux-ci permettent de modéliser des ordonnanceurs et des modèles de tâches dont le comportement est trop complexe pour être analysé par des méthodes analytiques. Ce langage spécifique à Cheddar est défini par l'association de deux parties. D'une part, l'algorithmique nécessaire à modéliser un ordonnanceur est exprimée par un sous ensemble d'Ada 95. D'autre part, les contraintes temporelles et de synchronisation entre les entités de l'architecture sont exprimées par un modèle d'automates temporisés. Associé à ce langage spécifique, nous proposons un processus dirigé par les modèles permettant d'effectuer des simulations d'une architecture à grande échelle à partir d'un modèle d'ordonnanceur. Ce processus traduit un modèle orienté automate temporisé en un ensemble de paquetages Ada pouvant être intégré au canevas de simulation Cheddar.

Il existe de nombreux projets qui, comme Cheddar, ont étudié l'applicabilité de la théorie de l'ordonnancement temps réel. Ces projets ont parfois donné lieu au développement de divers logiciels.

La communauté des méthodes formelles est la première à avoir étudié l'analyse de l'ordonnançabilité. Une architecture composée de tâches qui partagent des ressources communes peut être modélisée grâce aux réseaux de Petri, aux automates temporisés ou avec une l'algèbre des processus. L'utilisation de ces différents langages formelles a conduit au développement de différents outils de simulation ou de *model-checking*. Par exemple, TIMES [FMPY06] est un outil basé sur les automates temporisés. VESTA [SLC06] exploite une algèbre de processus. Enfin, différents outils de réseaux de Petri peuvent être utilisés dans ce contexte : CPN Tools [Wel06], CPN-AMI [HHK+06], Tina [BV06] ou PeNSMARTS [GCG00].

Les outils décrits ci-dessus n'exploitent pas directement la théorie de l'ordonnancement temps réel. En fait, hormis Rapid-RMA [TP03], Timewiz [Tim02] et MAST [HGGM01], très peu d'outils implantent

les méthodes analytiques de cette théorie.

Rapid-RMA et Timewiz sont des outils commerciaux dont le spectre d'application est limité à des environnements d'exécution restreints. Ces outils ne sont donc pas librement disponibles pour la communauté. MAST est l'outil dont les services sont les plus proches de ceux fournis par Cheddar. MAST est un outil distribué sous GPL et qui implante certains des tests de faisabilité présentés dans ce mémoire. MAST propose très peu d'outils de simulation. Il existe, d'autre part, diverses expérimentations qui tentent d'employer MAST avec des langages de conception basés sur UML (ex : meta-modèle RCM [PV07] ou MARTE-UML [FGD06]).

Finalement, des projets tels que Shark [GAGB01], BOSSA [BM02] ou Giotto [HKSP03] ont étudié comment modéliser des ordonnanceurs temps réel. En général, ces projets définissent un langage de modélisation d'ordonnanceur temps réel et propose un modèle d'architecture. Dans le cas de BOSSA, un canevas logiciel est également proposé afin de faciliter l'intégration d'un modèle d'ordonnanceur au sein d'un système d'exploitation. Avec Shark et Giotto, à partir d'un modèle d'ordonnanceur temps réel, des outils de génération de code peuvent être employés afin d'implanter automatiquement le système temps réel. Dans le cadre du projet Cheddar, nous ne cherchons pas actuellement à générer une telle implantation : seule la génération d'un environnement de simulation est recherchée dans ce projet.

5.1 Synthèse des contributions du projet Cheddar

Il est difficile de dire si le projet Cheddar a effectivement amélioré l'applicabilité de la théorie de l'ordonnancement temps réel. Il est toutefois possible de dresser une liste de quelques contributions scientifiques et techniques du projet.

5.1.1 Contributions d'ordre scientifique

État de l'art de la théorie de l'ordonnancement temps réel

Au cours de la mise en œuvre des outils de vérification, nous avons sélectionné un ensemble de méthodes analytiques. Ce travail a permis de faire un état de l'art des méthodes analytiques proposées par la littérature. Des tests de faisabilité manquants ont été identifiés. Cet état de l'art a nécessité la conception d'un modèle de données permettant d'appliquer les méthodes classiques d'analyse de l'ordonnançabilité. Cette modélisation est essentielle pour faire interopérer différents outils de modélisation et d'analyse.

Propositions de files d'attente adaptées aux applications temps réel

Des contributions théoriques ont été proposées dans le domaine de la théorie des files d'attente pour les architectures temps réel concurrentes.

Dans le cadre de la thèse de Jérôme Legrand que j'ai encadré, nous avons proposés deux modèles de file d'attente : les files P/P/1 et M/P/1. Les lois de service et d'arrivée de ces files d'attente modélisent

le comportement temporel de tâches périodiques ordonnancées avec des algorithmes classiques tels que *Rate Monotonic* ou *Earliest Deadline First*.

Dans sa thèse, Jérôme Legrand a proposé une résolution exacte de la file P/P/1 et une résolution approchée de la file M/P/1. À partir de ces files d'attente, des tests de faisabilité ont été définis afin de conduire des analyses de l'empreinte mémoire d'une architecture temps réel.

Par la suite, nous avons montré comment appliquer ces files d'attente sur des modèles AADL.

Un langage spécifique pour la modélisation et la vérification d'ordonnanceurs temps réel

Nous avons proposé un langage spécifique qui permet de modéliser le comportement d'ordonnanceurs. Ce langage de modélisation a permis de concevoir et de vérifier de nouveaux ordonnanceurs temps réel. Dans [SP07a], nous avons proposé un modèle d'ordonnancement hiérarchique pour ARINC 653. [AKLL06] ont exprimé et vérifié avec Cheddar un ordonnanceur pour des applications multimédias. [MS08] ont étudié avec Cheddar un ordonnanceur temps réel non déterministe. Enfin, ce langage de modélisation est employé depuis plusieurs années par la société Airbus pour la modélisation d'une plate-forme de simulateur de vol [CCG07].

5.1.2 Contributions d'ordre technique

Une plate-forme de vérification libre

Les outils développés sont diffusés librement à la communauté. Ils contribuent à de nombreux projets de recherche concernant l'ingénierie dirigée par les modèles et l'analyse de performance [GH08, FRA+07, HZPK07, RTC07, RBP07, NCSA07]. À titre d'exemple, nous avons employé ces outils pendant la thèse de Loïc Plassart (UBO/LISyC) afin de vérifier le contrôleur d'une chaîne de production manufacturière mis en œuvre avec le système d'exploitation RTAI [PSPM05]. La plate-forme de vérification Cheddar a été déposée auprès de l'Agence de Protection des Programmes en 2008.

D'autre part, plusieurs enseignements sur le temps réel ont été élaborés avec les outils Cheddar dans différents établissements tels que Télécom Paris-Tech, University of Rhode Island, University of Monash, Universitat Politécnica de Catalunya, University of the West Indies, ... Ces enseignements constituent un moyen essentiel pour promouvoir la théorie de l'ordonnancement temps réel.

Contributions à la validation et à l'évolution du standard AADL

Plusieurs études et expérimentations conduites dans le cadre du projet Cheddar contribuent à l'évaluation et à l'évolution du standard AADL. Les activités sur le langage AADL que nous menons sont actuellement effectuées en collaboration avec la société Ellidiss Technologies et Télécom Paris-Tech. Ces expérimentations ont été présentées au comité de normalisation AADL et certaines de nos propositions ont été intégrées dans la version 2 du standard [SLN05, DSLN05, DLPS08].

Une de ces expérimentations consiste à décrire avec AADL un ensemble de patrons de conception afin de garantir que les performances de l'architecture pourront être vérifiées par des méthodes analytiques. Télécom-Paris Tech a par ailleurs montré qu'il était possible d'intégrer ces mécanismes d'analyse dans

un processus permettant, à partir d'un modèle AADL conforme au profil «Ravenscar», de générer l'exécutable avec PolyOrb, puis d'effectuer l'analyse du modèle avec Cheddar [HZPK08]. Ce travail montre que le langage AADL peut être employé comme langage pivot dans le cadre d'un processus automatisant la vérification de l'ordonnancement.

Par ailleurs, les outils de modélisation d'Ellidiss Technologies intègrent depuis peu les technologies d'analyse de performance développées dans le projet Cheddar. Cette activité conduit donc à un transfert technologique vers la société Ellidiss Technologies.

De façon plus générale : contribution à l'ingénierie dirigée par les modèles pour le logiciel temps réel

Les outils Cheddar ont été employés avec d'autres langages de conception logicielle ou d'architecture. La société Thalès a montré que les outils Cheddar pouvaient analyser des modèles exprimés grâce au profil UML Marte de l'OMG [Mae07]. Un travail similaire a été réalisé par l'Université de Madrid, dans le cadre du projet PPOOA. En effet, l'université de Madrid a étendu le méta-modèle d'UML pour les architectures temps réel orientées pipeline afin qu'elles puissent être analysées par Cheddar [Fer07, FM08]. Le projet Cheddar contribue donc indirectement aux activités de différentes équipes de recherche qui définissent ce que seront peut-être les langages et les plates-formes de demain pour l'ingénierie dirigée par les modèles des logiciels temps réel. Il existe plusieurs outils de modélisation qui peuvent être employés conjointement avec Cheddar : Stood [DS08], TOPCASED [FGC+06], IBM Rational Software Architect [Mae07] ou PPOOA-Visio [FM08].

5.2 Travaux en cours et perspectives scientifiques à moyen termes

Ces premiers résultats sont actuellement poursuivis et ouvrent de nombreuses perspectives scientifiques à moyen termes.

5.2.1 Travaux en cours

Les travaux en cours concernent principalement l'interopérabilité entre Stood et Cheddar grâce aux patrons de conception définis dans le chapitre 3 ainsi que l'évaluation de performances des outils proposés dans le chapitre 4.

AADL : un langage pivot pour l'ingénierie du logiciel temps réel

Dans le cadre de la collaboration avec Ellidiss Technologies, l'usage d'AADL comme langage pivot entre outils de modélisation et outils de vérification reste la perspective de travail prioritaire.

Dans le chapitre 3, nous avons décrit quatre patrons de conception avec AADL. Nous avons focalisé nos efforts sur les patrons de conception «Ravenscar» et «Queued Buffer». Nous avons pour l'instant peu étudié le patron de conception «Blackboard». Par ailleurs, il existe de nombreux autres paradigmes de synchronisation employés par les architectes systèmes qu'ils seraient utile d'étudier, toujours dans

l'optique d'aider à appliquer systématiquement et automatiquement les méthodes d'analyse proposées par la théorie de l'ordonnancement temps réel.

L'étude et l'expérimentation de ces quatre patrons de conception permettront d'évaluer la version 2 du standard AADL, et en particulier l'une de ses annexes en cours d'élaboration : l'annexe comportementale [SAE07]. Il s'agira ici de vérifier que cette annexe comportementale permet de spécifier les éléments nécessaires à la vérification de performances (exemple : informations sur l'accès aux ressources partagées par les *threads*, spécification de paradigmes de synchronisation spécifiques, ...). Enfin, toujours dans le contexte de la validation d'AADL version 2, il est envisagé d'employer le langage spécifique de Cheddar afin de modéliser les modèles de tâches proposés par AADL pour en vérifier la correction.

Évaluation de performance du générateur de simulateur

Il a été montré que le langage spécifique de Cheddar était pertinent pour modéliser des ordonnanceurs temps réel complexes. Nous avons proposé un procédé d'ingénierie dirigée par les modèles afin d'offrir des outils de simulation pour des architectures de grande taille spécifiées avec AADL et le langage spécifique de Cheddar. À ce jour, nous n'avons pas vérifié que ce procédé était effectivement capable de simuler efficacement des architectures de grande taille. En effet, le générateur de code Ada est toujours en cours de développement. L'amélioration des performances obtenue par l'utilisation de ce générateur devra être évaluée. Nous espérons réaliser cette évaluation sur un modèle d'ordonnanceur Cheddar de taille industrielle tel que celui élaboré par la société Airbus [CCG07].

5.2.2 Perspectives scientifiques à moyen terme

Nous décrivons maintenant quelques perspectives scientifiques à moyen terme qui peuvent donner lieu à plusieurs thèses.

Composition de patrons de conception AADL

Une perspective intéressante concerne la composition de modèles d'architecture. Actuellement, nous ne traitons pas le problème de la composition. En effet, une architecture est un assemblage de différentes entités telles que les patrons que nous avons décrits dans ce mémoire. Or, nous ne savons pas comment déterminer statiquement les performances globales d'une architecture à partir des propriétés et des performances de chacun de ses éléments. Pour espérer déduire d'une composition les performances d'une architecture, il est nécessaire de décrire formellement cette composition. Malheureusement, la description actuelle des patrons étudiés dans ce mémoire est en partie informelle.

Si l'on souhaite étudier la composition de patrons de conception, nous devrons aussi nous interroger sur la place des techniques de vérification proposées par la théorie de l'ordonnancement temps réel dans le cycle de développement. Ce deuxième volet est plutôt méthodologique et la collaboration avec la société Ellidiss technologies est essentielle dans ce contexte. En effet, il faudra s'assurer que les éventuelles propositions sont compatibles avec les pratiques industrielles. Par exemple, il faudra vérifier

que les processus de développement actuellement employés par les industriels peuvent être adaptés afin de guider le concepteur vers l'usage de patrons qui autorise l'analyse automatique de son architecture.

Vérification de modèles spécifiés par le langage spécifique de Cheddar

Une perspective plus ambitieuse consiste à évaluer les propriétés d'un modèle d'ordonnanceur temps réel exprimé avec le langage Cheddar autrement que par la simulation.

D'une part, les automates temporisés autorisent la vérification de propriétés par *model-checking* [BBL99]. Des plates-formes telles que TIMES [FMPY06, AFM+03] ont déjà exploré l'utilisation du *model-checking* avec les automates temporisés pour la vérification automatique de l'ordonnançabilité.

D'autre part, le sous-ensemble d'Ada utilisé dans le langage spécifique de Cheddar autorise l'application de méthodes d'analyse statique : nous pensons ici à des environnements tels que Spark [Bar03]. Par l'application de ces techniques, nous espérons pouvoir comparer des modèles d'ordonnanceur afin de simplifier l'évaluation de performances d'architectures temps réel. A titre d'illustration, un objectif possible consiste à utiliser les algorithmes d'ordonnancement classiques comme des étalons afin de pouvoir mesurer la performance des algorithmes d'ordonnancement spécifiés pour Cheddar.

Enfin, l'étude du déterminisme d'un modèle d'ordonnanceur Cheddar [BPR07] constitue également une piste de recherche intéressante. En effet, la démonstration qu'un modèle Cheddar est déterministe peut conduire à exhiber une preuve des performances de l'architecture. C'est notamment le cas pour certains modèles de tâches tels que le modèle des tâches périodiques. En effet, le calcul de l'ordonnancement avec un algorithme déterministe sur la période d'étude permet d'énumérer exhaustivement l'ordonnancement des tâches.

Élaboration des tests de faisabilité manquants

Par ailleurs, nous avons identifié de nombreux tests de faisabilité qui manquent aujourd'hui pour vérifier une plus large variété d'architectures temps réel.

Par exemple, de nombreux problèmes restent ouverts concernant les deux files d'attente P/P/1 et M/P/1 décrites dans le chapitre 3. Nous pensons que les perspectives associées à la file d'attente P/P/1 sont proches des préoccupations industrielles mais que la file d'attente M/P/1 offre des perspectives scientifiques intéressantes.

D'une part, la méthode d'approximation développée pour la file d'attente M/P/1 doit être améliorée. Dans le même temps, afin de valider ce modèle de file d'attente, les usages possibles de la file d'attente M/P/1 doit être explorés. Tous les systèmes où sont mélangés des événements, des traitements ou des tâches ayant des lois d'arrivée périodique et non déterministes sont susceptibles d'être modélisés avec une file d'attente telle que M/P/1. Ainsi, ce modèle d'analyse peut être employé avec de nombreuses applications multimédias. La modélisation de divers protocoles de communication adaptés aux systèmes temps réel, qui mélange échanges de messages périodiques et apérioques, peut aussi être envisagée avec ce modèle analytique. On pense ici à un protocole tel que FIP [CIA99]. Les architectures temps réel dont le comportement de l'environnement peut est variable ou indéterministe constituent un autre exemple de domaine applicatif possible : on pense ici, par exemple, aux réseaux de capteurs sans fil [Fol05].

Enfin, plus simplement, l'ordonnancement d'un jeu de tâches apériodiques par un serveur sporadique [SSL89] peut être étudié grâce à ce modèle de file d'attente.

D'autre part, la généralisation de la loi P pour la file d'attente P/P/1 doit être étudiée. En effet, la file d'attente P/P/1 proposée dans ce mémoire suppose que nous employons un modèle de tâches très limité. Ainsi, la possibilité de définir une file d'attente avec plusieurs serveurs, c'est à dire plusieurs consommateurs de messages, constitue un première extension nécessaire à l'applicabilité de cette file d'attente. Nous devrons aussi nous interroger sur la possibilité de spécifier plus précisément les instants de consommation ou de production des messages. Enfin, pour l'instant, nous avons supposé que les délais critiques sont inférieurs aux périodes. Or, la littérature propose des méthodes de calcul du temps de réponse pour des tâches dont les délais critiques sont supérieurs aux périodes. Il est dommage que la file d'attente P/P/1 ne puisse donc pas être appliquée dans ce contexte.

Application aux systèmes répartis et multiprocesseurs

Les différentes méthodes d'analyse que nous avons expérimentées font toutes l'hypothèse que les contraintes temporelles à vérifier concernent des entités localisées sur un seul processeur. Il se trouve que de nombreuses applications industrielles sont par nature réparties.

La problématique concernant la vérification des systèmes temps réel répartis n'est pas nouvelle. Bien que de nombreuses équipes aient exploré ce domaine, il existe peu de résultats et très peu de contributions font consensus dans la communauté. Par exemple, du point de vue des méthodes analytiques, l'approche la plus connue est celle du calcul holistique [TC94] que nous avons présentée dans le chapitre 2. Elle est malheureusement réputée trop pessimiste [MM05]. A ce jour, la simulation semble être la méthode la plus réaliste pour valider ce type d'architecture.

Les problèmes levés par l'analyse de performances des architectures réparties sont multiples et difficiles. D'abord, plusieurs ressources doivent être modélisées simultanément. Ensuite, dans les environnements industriels, il existe de nombreuses solutions différentes pour la gestion de ces ressources. Par exemple, le calcul du pire temps de réponse d'une fonction qui nécessite le transfert de messages implique que les temps de communication entre les processeurs soient analysés. Ces délais de communications dépendent de nombreux facteurs tels que les protocoles d'arbitrage du support physique [UK94]. Selon le domaine applicatif, de nombreux protocoles existent : protocoles unidirectionnelles (ARINC 429, DigiBus), protocoles à CSMA (bus CAN [Zel96]), protocoles à jeton (Token-Bus [Puj95], Profibus [Str96]), protocoles à base de polling (FIP [CIA99], MIL-STD-1553 [Hea96]) ou protocoles TDMA [23] (ARINC 629 [Koo95]), ... La réalisation d'un tel environnement de simulation devient alors un travail fastidieux.

Sur certains points heureusement, les méthodes d'analyse développées pour l'ordonnancement des tâches peuvent être adaptées à ces nouvelles ressources. Ainsi, le partage d'un bus CAN par un ensemble de messages périodiques peut être analysé grâce aux résultats proposés pour l'ordonnancement à priorité fixe non préemptif de tâches périodiques.

[23] Time Division Multiple Access.

Néanmoins, la conception et la mise en œuvre d'un environnement de vérification pour ces architectures reste un challenge très difficile.

5.3 Conclusion générale

Dans ce document, nous avons décrits les objectifs, les premiers résultats mais aussi les travaux en cours et à venir que nous prévoyons dans le cadre du projet Cheddar.

Une des difficultés majeures du travail entrepris ici, et son caractère multidisciplinaire. En effet, pour conduire un tel projet, il est nécessaire d'acquérir un savoir minimal dans de divers domaines associés aux systèmes temps réel. L'apport principal du projet Cheddar est la mise en commun de méthodes, d'algorithmes, de modèles et d'outils issues de la théorie de l'ordonnancemnt temps réel, de la théorie des files d'attente, des langages et méthodes de conception du logiciel mais aussi des systèmes, des méthodes et des outils de compilation/ingénierie logicielle, des méthodes de vérification, et enfin, des pratiques industrielles. Tous ces savoirs ne peuvent qu'être apportés dans le cadre de coopérations entre différents partenaires académiques, mais aussi industriels.

Il est difficile d'évaluer l'impact concret d'un projet comme Cheddar. Par ailleurs, cette multidisciplinarité rend parfois frustrant le fait de ne pas pouvoir développer et acquérir une expertise pointue sur l'un ou l'autre de ces domaines. Par la diversité des domaines abordés, ce projet est par contre une excellente opportunité de satisfaire sa curiosité. A titre personnel, c'est finalement l'une des plus intéressantes retombées de ce projet.

Bibliographie

[ABLL92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations : Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1) :53–79, February 1992.

[ABRT93] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[AD90] R. Alur and D. L. Dill. Automata for modeling real time systems. Proc. of Int. Colloquium on Algorithms, Languages and Programming, Vol 443, LNCS, pages 322–335, 1990.

[AFM+03] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES : a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems, FORMATS'03, Marseille, France, September 2003.

[AGS02] K. Altisen, G. Gossler, and J. Sifakis. Scheduler Modeling Based on the Controller Synthesis Paradigm. *Real Time Systems journal*, 23(1) :55–84, 2002.

[AKLL06] B. Ahn, J. Kim, D.H. Lee, and S.H. Lee. A Real Time Scheduling Method for Embedded Multimedia Applications. Proceedings of the 2006 International Conference on Pervasive Systems and Computing, Las Vegas, Nevada, USA, June 26-29, CSREA Press, Pages 104-107, 2006.

[AP04] L. Almeida and P. Pedreiras. Scheduling within Temporal Partitions : response-time analysis and server design. *Proceedings of the EMSOFT'04 conference. September 27-29, Pisa, Italy*, pages 95–103, 2004.

[Ari97] Arinc. *Avionics Application Software Standard Interface*. The Arinc Committee, January 1997.

[AVS02] K. E. Avrachenkov, N. O. Vilchensky, and G. L. Shevlyaokov. Priority queueing with finite buffer size and randomized push-out mechanism. Technical report, INRIA technical Report number 4434, March 2002.

[Bak91] T.P. Baker. Stack-based scheduling for realtime processes. *Journal of Real Time systems*, 3(1) :67–99, March 1991.

[Bar03] J. Barnes. *High integrity software : The Spark approach to safety and security.* Addison-Wesley Publishing Company, 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.

[BBL99] B. Bérard, M. Bidoit, and F. Laroussinie. *Vérification de logiciels, technique et outils du model-checking.* Editions Vuibert, 1999.

[BDF+06] J.P. Bodeveix, P. Dissaux, M. Filali, P. Gaufillet, and F. Vernadat. Behavioural descriptions in architecture description languages, application to AADL. Proceedings of ERTS conference, Toulouse, 2006.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. Technical Report Updated the 17th November 2004, Department of Computer Science, Aalbord University, Denmark, 2004.

[Ber05] G. Berry. Getting Started with Esterel Studio 5.3. Technical report, Esterel technologies SA. Available from http ://www.esterel-technologies.com/technology/getting-started/, April 2005.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Science Of Computer Programming*, 19(2) :87–152, 1992.

[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.

[BHW03] A. Burns, M. Harbour, and A. Wellings. A round robin scheduling policy for Ada. *In Reliable Software Technologies, Proceedings of the Ada Europe Conference, Toulouse, France vol. 2655, pp. 334-343*, 2003.

[BM02] J. Barreto and G. Muller. Bossa : a Language-based Approach for the Design of Real Time Schedulers . In RTS'2002, pp. 19-31, Paris, France, March 2002.

[BMR90] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard real-time sporadic tasks on one processor. pages 182–190. 11th Real time System Symposium, December 1990.

[Boe81] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[BoHT03] J. Blumenthal, olatowski, J. Hildebrandt, and D. Timmermann. *Framework for validation and Analysis of Real time Scheduling Algorithms and scheduler implementations* . University of Rostock, Technical report available from http ://yasa.e-technik.uni-rostock.de/, 2003.

[Bou95] M. Bouazza. *La norme STEP.* Hermès, Documentation multimédia, 1995.

[BPR07] F. Boniol, C. Pagetti, and F. Revest. Functionally Deterministic Scheduling. pages 33–40. ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation, December 2007.

[BRV+03] B. Berthomieu, P. O. Ribet, F. Vernadat, J.L. Bernartt, J.M. Farines, J.P. Bodeveix, P. Farail, M. Filali, G. Padiou, P. Michel, P. Farail, P. Gaufillet, P. Dissaux, and J.L. Lambert. Towards the verification of real time systems in avionics : the COTRE approach. Electronic notes in Theorical Computer Sciences ENTCS, volume 80, 2003.

[BV06] B. Berthomieu and F. Vernadat. Time Petri Nets Analysis with TINA. In Proceedings of 3rd Int. Conf. on The Quantitative Evaluation of Systems (QEST 2006), IEEE Computer Society, 2006.

[BW94] A. Burns and A.J. Wellings. HRT-HOOD : A Design Method for Hard Real-time Systems. *Real Time Systems journal*, 6(1) :73–114, 1994.

[BW07] A. Burns and A. Wellings. *Concurrent and Real Time programming in Ada. 2007.* Cambridge University Press, 2007.

[Car96] F. V. Carvahlo. Sur l'Intégration de Mécanismes d'Ordonnancement et de Communication dans la sous-Couche MAC de Réseaux Locaux Temps réel . Thèse de l'Université de Toulouse 3, 1996.

[CCG07] M. Castor, J. Casteres, and F. Gasmi. Modélisation et simulation de l'Architecture des simulateurs avion pour la mesure de performance. Rapport technique Airbus, September 2007.

[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.* John Wiley and Sons Ltd editors, 2002.

[CGL94] G. Ciardo, R. German, and C. Lindemann. A characterization of the stochastic process underlying a stochastic Petri net. *IEEE Trans. Softw. Eng.*, 7(20) :506–515, July 1994.

[CIA99] CIAME. *Réseaux de terrain.* Edition Hermès, 1999.

[CLX95] D. Clarke, I. Lee, and H. Xie. VERSA : A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems. *Journal of Computer and Software Engineering*, 3(2), April 1995.

[CPRS03] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Techniques et Sciences Informatiques (TSI)*, 22(5) :651–677, 2003.

[DB05] R. I. Davis and A. Burns. Hierarchical Fixed Priority Pre-Emptive Scheduling. *In the 26th IEEE International Real-Time Systems Symposium (RTSS'05). Miami, Florida, USA.*, pages 389–398, December 2005.

[DBF+06] P. Dissaux, J.P. Bodeveix, M. Filali, P. Gaufillet, and F. Vernadat. AADL behavioral annex. Proceedings of DASIA conference, Berlin, January 2006.

[Dis04] P. Dissaux. Using the AADL for mission critical software development. *2nd European Congress ERTS, EMBEDDED REAL TIME SOFTWARE Toulouse*, January 2004.

[DLPS08] P. Dissaux, J. Legrand, A. Plantec, and F. Singhoff. AADL performance analysis with Cheddar : a summary. SAE AADL Working Group meeting, Sevilla,, April 14-18th 2008.

[Dru06] D. Drusinsky. *Modeling and Verification using UML StateCharts*. Elsevier inc. editor, 2006.

[DS08] P. Dissaux and F. Singhoff. Stood and Cheddar : AADL as a Pivot Language for Analysing Performances of Real Time Architectures. Proceedings of the European Real Time System conference. Toulouse, France, January 2008.

[DSLN05] P. Dissaux, F. Singhoff, J. Legrand, and L. Nana. The stood-cheddar platform. SAE AADL Summer Working Group meeting, Grand Rapids,, July 11-14th 2005.

[Ell07] Ellidiss. ADELE : a versatile system architecture graphical editor based on AADL. http ://gforge.enseeiht.fr/projects/adele, 2007.

[FBFR07] R. B. Frana, J. P. Bodeveix, M. Filali, and J. F. Rolland. The AADL behaviour annex – experiments and roadmap. pages 377 – 382. 12th IEEE International Conference on Engineering Complex Computer Systems, July 2007.

[FD02] P. Farail and P. Dissaux. COTRE a new approach for modelling real-time software for avionics. Proceedings of DASIA conference, Dublin, January 2002.

[Fer07] J. L. Fernandez. PPOOA : Processes Pipelines in Object Oriented Architectures. http ://www.ppooa.com.es, 2007.

[FG05] P. Farail and P. Gaufillet. COTRE as an AADL profile. IFIP TC-2 Workshop on Architectural Description Languages, LNCS volume 175, pp 167-179, November 2005.

[FGC+06] P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. TOPCASED : An Open Source Development Environment for Embedded Systems. *Chapter 11, From MDD Concepts to Experiments and Illustrations, ISTE Editor*, pages 195–207, September 2006.

[FGD06] T. Frédéric, S. Gérard, and J. Delatour. Towards an UML 2.0 profile for real-time execution platform modeling. Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS 06), Work in progress session, July 2006.

[FM08] J. L. Fernandez and G. Marmol. An Effective Collaboration of a Modeling Tool and a Simulation and Evaluation Framework. 18 th Annual International Symposium, INCOSE 2008. Systems Engineering for the Planet. The Netherlands. 15-19 June 2008., 2008.

[FMPY06] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354(2) :301–317, March 2006.

[Fol05] B. Folliot. Les réseaux de capteurs : point de vue de vue du système d'exploitation. Workshop CNRS RECAP, Nice, November 2005.

[FRA+07] R. B. França, J.F. Rolland, M. Filali Amine, J.P. Bodeveix, and D. Chemouil. Assessment of the AADL Behavioral Annex. Journées FAC'2007, Formalisation des Activités Concurrentes, March 2007.

[GAGB01] JP. Gai, L. Abeni, M. Giogi, and G. Buttazzo. A New Kernel Approach for Modular Real-Time systems Development. Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.

[Gal95] B. O. Gallmeister. *POSIX 4 : Programming for the Real World* . O'Reilly and Associates, January 1995.

[GCG00] E. Grolleau and A. Choquet-Geniet. Off-Line Computation of Real-Time Schedules by Means of Petri nets. pages 309–316. Workshop On Discrete Event Systems, Analysis and Control, Ghent, Belgium, Kluwer Academic Publishers, June 21-25 2000.

[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.

[GH08] O. Gilles and J. Hugues. Applying WCET analysis at architectural level. pages 113–122. Worst-Case Execution Time (WCET'08). Prague, Czech Republic, July 2008.

[GK96] M. Gagnaire and D. Kofman. *Réseaux Haut Débit : réseaux ATM, réseaux locaux, réseaux tout-optiques.* Masson-Inter Editions, Collection IIA, 1996.

[GMSB96] M.C. Gaudel, B. Marre, F. Schlienger, and G. Bernot. *Précis de génie logiciel.* Collection Enseignement de l'Informatique. Masson, 1996.

[GN01] P. Gai and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip . volume 3, pages 67–99. In Proceedings of the 22nd IEEE Real-Time Systems Symposium, March 2001.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-time Uniprocessor Scheduling. INRIA Technical report number 2966, 1996.

[GV03] C. Girault and R. Valk. *Petri nets for systems engineering : a guide to modeling, verification, and applications.* Springer Verlag, 2003.

[Har87] D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.

[HCRP91a] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs : le langage Lustre. *Technique et Science Informatiques*, 10(2) :139–157, 1991.

[HCRP91b] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[Hea96] D. Head. MIL-STD-1553B. *Real Time Magazine*, (2) :25–28, April 1996.

[HGGM01] M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano. MAST : Modeling and Analysis Suite for Real Time Applications. pages 125–134. Proc. of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands,, June 2001.

[HHK+06] A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, and Y. Thierry-Mieg. New Features in CPN-AMI 3 : focusing on the analysis of complex distributed systems. pages 273–275. In 6th international Conference on Application of Concurrency to System Design (ACSD'06), Turku, Finland, IEEE Computer Society, June 2006.

[HKSP03] T. A. Henzinger, C. M. Kirsch, M. A.A. Sanvido, and W. Pree. From Control Models to Real-Time Code Using Giotto. *IEEE Control Systems Magazine*, 1(23) :50–64, 2003.

[HP03] M. G. Harbour and J.C. Palencia. Response Time Analysis for Tasks Scheduled under EDF within Fixed Priorities. In *Proceedings of the 24th IEEE Real-Time Systems Symposium, Cancun, Mexico*, December 2003.

[HU01] J. E. Hopcroft and J. D. Ullman. Introduction of Automata Theory, Languages and Computation. Addison-Wesley editor, 2001.

[HZPK07] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina. In 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07), Porto Allegre, Brazil, June 2007.

[HZPK08] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Transactions on Embedded Computing Systems (TECS) , ACM Press, New York, USA*, 7(4) :42 :2–42 :25, July 2008.

[IKL+99] T. K. Iversen, K. J. Kristoffersen, K. G. Larsen, R. G. Madsen, M. Laursen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen. Model-Checking Real Time Control Programs : Verifying LEGO Mindstorm Systems Using UPPAAL. Technical report, BRICS RS-99-53, December 1999.

[ISO94] ISO 10303-1. *Part 1 : Overview and fundamental principles*, 1994.

[ISO95] ISO. Ada Reference Manual ISO/IEC 8652 :1995(E) with Technical Corrigendum 1 and Amendment 1 (Draft 16). 1995.

[ISO98] ISO 10303-22. *Part 22 : Implementation method : Standard data access interface specification*, 1998.

[ISO01] ISO 10303-21. *Part 21 : edition 2, Implementation method : Clear text encoding of the exchange structure*, 2001.

[ISO04a] ISO 10303-11. *Part 11 : edition 2, EXPRESS Language Reference Manual*, 2004.

[ISO04b] ISO 10303-28. *Part 28 : Implementation method : XML Representation of EXPRESS Schemas and Data*, 2004.

[ISO07] ISO. *Ada 2005 International Standard ISO/IEC ISO/IEC 8652 :1995/Amd 1 :2007.* Ada Working Group (WG9), March 2007.

[JDM91] K. Jeffay, D.Stanat, and C. Martel. On non preemptive Scheduling of periodic and Sporadic Tasks. in Proc. Of the IEEE Real Time Systems Symposium (RTSS'91), San Antonio, Texas,, December 1991.

[J.M03] J.M. Rifflet. *Unix, Programmation et communication.* Dunod informatique, 2003.

[JP86] M. Joseph and P. Pandya. Finding Response Time in a Real-Time System. *Computer Journal*, 29(5) :390–395, 1986.

[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*, March 2005.

[KL88] J. Kay and P. Lauder. A Fair Share Scheduler. In *Communications of the ACM*, volume 31, pages 44–45, January 1988.

[Kle75a] L. Kleinrock. *Queueing Systems : Computer Application.* Wiley-interscience, 1975.

[Kle75b] L. Kleinrock. *Queueing Systems : theory.* Wiley-interscience, 1975.

[Koo95] P. J. Koopman. Time Division Multiple Access Without a Bus Master. Technical Report RR-9500470, United Techologies Research Center, June 1995.

[Kra85] S. Krakowiak. *Principe des systèmes d'exploitation des ordinateurs.* Dunod Informatique, 1985.

[KRP+94] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real Time Analysis.* Kluwer Academic Publishers, 1994.

[KWK02] J. Kwon, A. Wellings, and S. King. Ravenscar-Java : A High Integrity Profile for Real-Time Java. *University of York, Technical Report YCS 342*, May 2002.

[Leb98] L. Leboucher. *Algorithmique et Modélisation pour la Qualité de Service des Systèmes Répartis Temps Réel.* Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications de Paris, septembre 1998.

[Leg04] J. Legrand. Contribution à l'ordonnancement des systèmes temps réel comprenant des tampons. Doctorat de l'Université de Bretagne Occidentale, Brest, décembre 2004.

[LL73] 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.

[LM80] J.Y.T Leung and M.L. Merril. A note on preemptive scheduling of periodic real time tasks. *Information processing Letters*, 3(11) :115–118, 1980.

[LMD04] J.L. Lawall, G. Muller, and H. Duchesne. Language Design for implementing Process Scheduling Hierarchies. *Proceedings of the PEPM'04 conferences. August 24-26, Veronna Italy*, pages 80–90, 2004.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2) :134–152, 1997.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm : Exact Characterization and Average Case Behaviour. pages 166–171. in Proc. Of the IEEE Real Time System symposium (RTSS'89), December 1989.

[LSN+03] J. Legrand, F. Singhoff, L. Nana, L. Marcé, F. Dupont, and H. Hafidi. About Bounds of Buffers Shared by Periodic Tasks : the IRMA project. In the 15th Euromicro International Conference of Real Time Systems (WIP Session), Porto, July 2003.

[LSNM04] J. Legrand, F. Singhoff, L. Nana, and L. Marcé. Performance Analysis of Buffers Shared by Independent Periodic Tasks. LISyC Technical report, number legrand-02-2004, Available at http ://beru.univ-brest.fr/~singhoff/cheddar, January 2004.

[Mae07] E. Maes. Validation de systèmes temps-réel et embarqué à partir d'un modèle MARTE. Thales RT, Journée Ada-France 2007, Brest, décembre 2007.

[Mig99] J. Migge. *Real-time scheduling : a trajectory based model.* PhD Thesis, University of Nice Sophia Antipolis, November 1999.

[MM05] S. Martin and P. Minet. Worst case end-to-end response times for non-preemptive FP/DP* scheduling. INRIA Technical report number RR-5418, March 2005.

[MOF07] OMG's MetaObject Facility. http ://www.omg.org/mof/, 2007.

[MS08] P.K. Muhuri and K.K. Shukl. Real-time task scheduling with fuzzy uncertainty in proceeding times and deadlines. Applied Soft Computing review, Volume 8, Issue 1, Pages 1-13., January 2008.

[Nas07] O. Nasr. Spécification et Vérification des ordonnanceurs temps réel en B. *Thèse de doctorat, Université Paul Sabatier*, novembre 2007.

[NCSA07] F. Nemer, H. Cassé, P. Sainrat, and A. Awada. Improving the WCET accuracy by inter-task instruction cache analysis. IEEE International Symposium on Industrial Embedded Systems (SIES 2007), Lisbonne, July, p. 25-32, July 2007.

[NMB04] M. Nicholson, J.A. McDermid, and A. Burns. Analysis and Design Synthesis for Hard Real-Time Safety Critical Systems. YCS-237 technical report, Department of Computer Science, University of York, November 2004.

[OMG02] OMG. *OMG Meta-Object Facility (MOF) Specification, v1.4*, 2002.

[OMG07] OMG. *A UML Profile for MARTE, Beta 1.* OMG Document Number : ptc/07-08-04, August 2007.

[Par76] D.L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1) :1–9, 1976.

[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.

[PI76] S. S. Panwalkar and W. Iskander. A survey of Scheduling rules. *Operations Research*, 25(1) :45–1, January 1976.

[Pla08] L. Plassart. Contribution à l'optimisation de systèmes flow-shop coordonnés par un contrôleur central. Doctorat de l'Université de Bretagne Occidentale, Brest, mai 2008.

[PR98] A. Plantec and V. Ribaud. EUGENE : a STEP-based framework to build Application Generators. *AWCSET'98, CSIRO-Macquarie University*, 1998.

[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.

[PSPM05] L. Plassart, F. Singhoff, P. Le Parc, and L. Marcé. Impact de l'ordonnancement temps réel des tâches d'un superviseur de ligne d'assemblage . 1ères Rencontres des Jeunes Chercheurs en Informatique Temps Réel 2005 (RJCITR'05), conjointement à l'école d'été temps réel 2005 (ETR'05), Nancy, 2005.

[Puj95] G. Pujolle. *Les réseaux* . Editions Eyrolles, décembre 1995.

[PUZ+06] J. A. Pulido, S. Uruena, J. Zamorano, T. Vardanega, and J. A. De la Puente. Hierarchical Scheduling with Ada 2005. *Proceedings of the 11th International Conference on Reliable Software Technologies, Springer verlag, LNCS vol 4006, pp 1-12, Porto, Portugal*, June 2006.

[PV07] M. Panunzio and T. Vardanega. A Metamodel-Driven Process Featuring Advanced Model-Based Timing Analysis . Proceedings of the 12th International Conference on Reliable Software Technologies, Ada-Europe. Geneva, LNCS springer-Verlag, June 2007.

[RBP07] F. Revest, F. Boniol, and C. Pagetti. Aide à la conception multi-points de vue de systèmes embarqués. Journées Formalisation des Activités Concurrentes, 15-16 mars, CERT-ONERA, Toulouse, 2007.

[RCM96] J. Ripoll, A. Crespo, and A.K. Mok. Improvement in Feasibility testing for Real Time Tasks. *Real Time Systems journal*, 11 :19–39, 1996.

[Rea89] C. Reade. *Elements of Functional Programming.* Addison-Wesley Longman Publishing, 1989.

[RH02] M. Rivas and M. G. Harbour. POSIX-compatible application-defined scheduling in MaRTE OS. *In Proceedings of the 14th IEEE Euromicro Conference on Real-Time Systems, Wien, Austria*, June 2002.

[RH03] M. Rivas and M. G. Harbour. Application Defined Scheduling in Ada. *ACM SIGAda Ada Letters* , 23(4) :42–51, December 2003.

[Riv97] Wind River. *VxWorks : programmer's guide.* Wind River System, March 1997.

[Rob90] T. G. Robertazzi. *Computer Networks and Systems : queueing theory and performance evaluation.* Springer-Verlag, 1990.

[RPGC02] M. Richard, P.Richard, E. Grolleau, and F. Cottet. Contraintes de précédence et ordonnancement mono-processeur. pages 121–138, mars 2002.

[RS01] J. Regehr and J. A. Stankovic. HLS : a Framework for Composing Soft Real-Time Schedulers. *In the 22th IEEE International Real-Time Systems Symposium (RTSS'01). London, UK.*, pages 3–14, December 2001.

[RTC07] J.F. Rolland, D. Thomas, and D. Chemouil. Utilisation d'AADL pour la conception de logiciels de vol satellite. Revue Génie logiciel, Number 80, pages 41-44, mars 2007.

[SAE04] SAE. Architecture Analysis and Design Language (AADL) AS 5506. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 1.0, November 2004.

[SAE07] SAE. AADL Annex Behavior (draft V1.6), AS 5506. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, March 2007.

[SEI03] SEI. The Rate Monotonic Analysis. Technical report, In the Software Technology Roadmap. http ://www.sei.cmu.edu/str/descriptions/rma_body.html, September 2003.

[SEI04] SEI. OSATE : An extensible Source AADL Tool Environment. *SEI AADL Team technical Report*, December 2004.

[SFR+07] G. Sébastien, P. Feiler, J.F. Rolland, M. Filali, M.O. Reiser, D. Delanote, Y. Berbers, L. Pautet, and I. Perseil. UML and AADL 2007 Grand Challenges. ACM SIGBED Review, A Special Report on UML and AADL Grand Challenges, volume 4, number 4,, October 2007.

[SGSS06] V. Subraminian, C. Gill, C. Sanchez, and H. B. Sipma. Reusable Models for Timing and Liveness Analysis of Middleware for Distributed Real-Time and Embedded systems. Proceedings of the 6th ACM and IEEE International conference on Embedded software EMSOFT '06, October 2006.

[Sin99] F. Singhoff. *Spécification temporelle modulaire et support pour les applications multimédias réparties.* Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications de Paris, December 1999.

[Sin07a] F. Singhoff. Cheddar Release 2.x User's Guide. LISyC Technical report, number singhoff-01-2007, Available at http ://beru.univ-brest.fr/~singhoff/cheddar, February 2007.

[Sin07b] F. Singhoff. The Cheddar AADL property set (Release 2.x). LISyC Technical report, number singhoff-03-2007, Available at http ://beru.univ-brest.fr/~singhoff/cheddar, February 2007.

[SL03] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *4th IEEE International Real-Time Systems Symposium (RTSS'03)*, 2003.

[SLC06] O. Sokolsky, I. Lee, and D. Clark. Schedulability Analysis of AADL models . International Parallel and Distributed Processing Symposium, IPDPS 2006, Volume 2006,, April 2006.

[SLN05] F. Singhoff, J. Legrand, and L. Nana. AADL resource requirements analysis with Cheddar. SAE AADL Working Group meeting, Paris, October 19-21th 2005.

[SLNM04] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Extending Rate Monotonic Analysis when Tasks Share Buffers. In the DAta Systems in Aerospace conference (DASIA 2004), Nice, July 2004.

[SLNM05] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and Memory requirements analysis with AADL. *ACM SIGAda Ada Letters, ACM Press, New York, USA*, 25(4) :1–10, November 2005.

[SP07a] F. Singhoff and A. Plantec. AADL Modeling and Analysis of a hierarchical schedulers. *ACM SIGAda Ada Letters, ACM Press, New York, USA*, 27(3) :41–50, November 2007.

[SP07b] F. Singhoff and A. Plantec. Towards User-Level extensibility of an Ada library : an experiment with Cheddar. Proceedings of the 12th International Conference on Reliable Software Technologies, Ada-Europe. LNCS springer-Verlag, Volume 4498, pages 180-191, Geneva, June 2007.

[Spu96] Marco Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, 1996.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols : An Approach to real-time Synchronization. *IEEE Transactions on computers*, 39(9) :1175–1185, 1990.

[SSL89] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic Task Scheduling for Hard-real-time Systems. *The Journal of Real Time Systems*, 1 :27–60, 1989.

[Sta88] John Stankovic. Misconceptions about real-time computing. *IEEE Computer*, October 1988.

[Sta92] I. Stavrakakis. A Considerate Priority Queueing System with Guaranteed Policy Fairness. pages 2151–2159. In the proceedings of IEEE Infocom'92 Conference, Florence, May 1992.

[Str96] H. Strass. Factory Floor Networks PROFIBUS : the natural choice. *Real Time Magazine*, (2) :6–8, April 1996.

[Tan01] A. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2001.

[TC94] K. W. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3) :117–134, April 1994.

[TDB+06] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, and P. Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1.* LNCS Springer Verlag, number XXII, volume 4348., 2006.

[Til05] J. F. Tilman. Building Tool Suite for AADL. volume 176, pages 197–207. IFIP International Federation for Information Processing, Springer Verlag editor, 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.

[TP03] Tri-Pacific. *Rapid-RMA : The Art of Modeling Real-Time Systems.* http ://www.tripac.com/html/prod-fact-rrm.html, 2003.

[UK94] P. Upendar and P. J. Koopman. Communication Protocols for Embedded Systems. *Embedded Systems Programming*, 7(11) :46–58, November 1994.

[Vah96] U. Vahalia. *UNIX Internals : the new frontiers.* Prentice Hall, 1996.

[Ver06] T. Vergnaud. Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées. *Thèse de l'ENST-Paris*, décembre 2006.

[VPK05] T. Vergnaud, L. Pautet, and F. Kordon. Using the AADL to describe distributed applications for middleware to software components. *Ada Europe 2005, 20-24 June, York*, June 2005.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3) :36 :53, April 2008.

[Wel06] L. Wells. Performance Analysis using CPN Tools. ACM International Conference Proceeding Series ; Vol. 180, Proceedings of the 1st international conference on Performance evaluation methodolgies and tools., 2006.

[Zel96] H. Zeltwanger. CAN in industrial Applications. *Real Time Magazine*, (2) :20–24, April 1996.

[ZHR+01] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, and M. Hoeber. *The Java Tutorial : A Short Course on the Basics, 4th Edition (The Java Series)*. Addison Wesley, 2001.

Annexes

Annexe A

Référentiel de tests de faisabilité pour Cheddar

Sommaire

Cette annexe présente un extrait de la liste des tests de faisabilité que nous avons sélectionnés pour le canevas Cheddar dans le cadre des tâches périodiques. [GRS96] et [KRP+94] proposent une synthèse des tests existants dans la littérature.

A.1 Définitions supplémentaires

En complément des définitions données au chapitre 2, nous introduisons ici quelques nouvelles définitions relatives aux tâches périodiques afin de caractériser les conditions d'applicabilité de chaque test de faisabilité présenté dans cette annexe.

Définition 4 *Tâche non concrète*

Une tâche non concrète est définie par les paramètres P_i, D_i et C_i.

Définition 5 *Tâche concrète*

Une tâche concrète est une tâche non concrète pour laquelle S_i est définie.

Définition 6 *Jeu de tâches concrètes synchrones*

Un jeu de tâches concrètes synchrones est un ensemble de tâches périodiques concrètes tel que $\forall i : S_i = k$ où k est désigné comme l'instant critique du jeu de tâches.

Définition 7 *Jeu de tâches concrètes asynchrones*

Un jeu de tâches concrètes asynchrones est un ensemble de tâches concrètes pour lesquelles aucun instant critique n'existe.

A.2 Tests basés sur la charge processeur

A.2.1 Test C1 : Rate Monotonic préemptif

1. **Méthode de calcul :**

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq B$$

2. **Références :** [LL73].
3. **Hypothèses :**
   - Tâches indépendantes, concrètes et synchrones.
   - Tâches à échéances sur requêtes.
4. **Commentaires :**
   - Si les tâches ne sont pas harmoniques, alors $B = n(2^{\frac{1}{n}} - 1)$ et le test est une condition suffisante mais non nécessaire.
   - Si les tâches sont harmoniques, alors $B = 1$ et le test est une condition nécessaire et suffisante.

A.2.2 Test C2 : Earliest Deadline First et Least Laxity First préemptif

1. **Méthode de calcul :**

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

2. **Références :** [LL73, LM80, CDKM00].
3. **Hypothèses :**
   - Tâches indépendantes, concrètes et synchrones.
4. **Commentaires :**
   - Condition nécessaire et suffisante si $\forall i : D_i = P_i$.
   - Si $\exists i : D_i < P_i$, alors $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1$ est une condition suffisante seulement, et $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$ est une condition nécessaire seulement.

A.2.3 Test C3 : Rate Monotonic non préemptif

1. **Méthode de calcul :**

$$\forall\ i, 1 \leq i \leq n : \sum_{i=1}^{n} \frac{C_i}{P_i} + \max_{i<i\leq n} \left(\frac{B_i}{P_i} \right) \leq n(2^{\frac{1}{n}} - 1)$$

2. **Références :** [Car96].
3. **Hypothèses :**
   - Tâches indépendantes, concrètes et synchrones.
   - Tâches à échéances sur requêtes.
4. **Commentaires :**
   - Condition suffisante mais non nécessaire.
   - On suppose que les tâches sont ordonnées de façon décroissante selon leur priorité : la tâche $i-1$ est donc moins prioritaire que la tâche i.

A.2.4 Test C4 : Rate Monotonic préemptif et non préemptif

1. **Méthode de calcul :**

$$\forall\ i, 1 \leq i \leq n : \sum_{k=1}^{i-1} \frac{C_k}{P_k} + \frac{C_i + B_i}{P_i} \leq i(2^{\frac{1}{i}} - 1)$$

2. **Références :** [SRL90, Car96].
3. **Hypothèses :**
   - Pour Rate Monotonic non préemptif : tâches indépendantes, à échéances sur requête, concrètes et synchrones.

- Pour Rate Monotonic préemptif : tâches dépendantes, à échéances sur requête, concrètes et synchrones.

4. **Commentaires :**
   - Condition suffisante mais non nécessaire.
   - On suppose que les tâches sont ordonnées de façon décroissante selon leur priorité : la tâche $i-1$ est donc moins prioritaire que la tâche i.
   - Ce test peut être employé dans deux cas de figures :
     (a) Soit pour tester un jeu de tâches ordonnancées avec un algorithme Rate Monotonic préemptif quand les tâches accèdent à des ressources partagées. On suppose alors que B_i est calculé en fonction du protocole d'accès aux ressources partagées.
     (b) Soit pour tester un jeu de tâches indépendantes avec un algorithme Rate Monotonic non préemptif. Dans ce cas, on suppose que B_i est le temps d'attente de la tâche i provoqué par les tâches de plus faibles priorités.

A.2.5 Test C5 : Deadline Monotonic préemptif

1. **Méthode de calcul :**

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1)$$

2. **Références :** [LL73].
3. **Hypothèses :**
   - Tâches avec $\forall i : D_i \leq P_i$.
   - Tâches indépendantes, concrètes et synchrones.
4. **Commentaires :**
   - Condition suffisante mais non nécessaire.

A.2.6 Test C6 : Earliest Deadline First non préemptif

1. **Méthode de calcul :**

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

et

$$\forall\ 1 < i \leq n : C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{P_j} \right\rfloor \cdot C_j \leq L$$

avec $P_1 < L < P_i$

2. **Références :** [JDM91].
3. **Hypothèses :**
   - Tâches à échéances sur requêtes.
   - Tâches indépendantes, concrètes et synchrones.
4. **Commentaires :**
   - Condition nécessaire et suffisante.
   - On suppose que les tâches sont ordonnées de façon décroissante selon leur priorité : la tâche $i-1$ est donc moins prioritaire que la tâche i.

A.2.7 Test C7 : algorithmes préemptifs à priorité fixe

1. **Méthode de calcul :**

$$\forall\ 1 \leq i \leq n : min_{0 \leq t \leq D_i} \left(\sum_{j=1}^{i} \frac{C_j}{t} \cdot \left\lceil \frac{t}{P_j} \right\rceil \right) \leq 1$$

2. **Références :** [LSD89].
3. **Hypothèses :**
   - Algorithme à priorité fixe, quelque soit l'algorithme d'affectation des priorités.
   - Tâches à échéances sur requêtes.
   - Tâches indépendantes, concrètes et synchrones.
4. **Commentaires :**
   - Condition nécessaire et suffisante.
   - On suppose que les tâches sont ordonnées de façon décroissante selon leur priorité : la tâche $i-1$ est donc moins prioritaire que la tâche i.

A.2.8 Test C8 : Earliest Deadline First préemptif

1. **Méthode de calcul :**

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

et

$$\forall t \geq 0 : h(t) \leq t$$

avec $h(t)$, la demande processeur tel que :

$$h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \right) \cdot C_i$$

2. **Références :** [BHR90, GRS96].
3. **Hypothèses :**
   - Tâches avec $\forall i : D_i \geq P_i$.
   - Tâches indépendantes et non concrètes.
4. **Commentaires :**
   - Condition nécessaire et suffisante.

A.2.9 Test C9 : Earliest Deadline First préemptif

1. **Méthode de calcul :**

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

et

$$\forall t \in S : h(t) \leq t$$

avec

$$S = \left(\bigcup_{i=1}^{n} k \cdot T_i + D_i \right) \cap \left(0, min \left(L, \frac{\sum_{i=1}^{n} (1 - D_i/T_i) \cdot C_i}{1 - U} \right) \right)$$

$h(t)$, la demande processeur et L, la période occupée synchrone du processeur.

2. **Références :** [BHR90, RCM96, GRS96].
3. **Hypothèses :**
   - Tâches avec $\forall i : D_i \leq P_i$.
   - Tâches indépendantes et non concrètes.
4. **Commentaires :**
   - Condition nécessaire et suffisante.

A.2.10 Test C10 : Earliest Deadline First préemptif

1. **Méthode de calcul :**

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

et

$$\forall t \geq 0 : h(t) \leq t$$

avec t inclue dans l'intervalle $[0, \frac{U}{1-U} \cdot max_{i=1,\ldots,n}(T_i - D_i))$.

2. **Références :** [BMR90, BHR90, GRS96].
3. **Hypothèses :**
   - Tâches avec $\forall i : D_i \geq P_i$.
   - Tâches indépendantes et non concrètes.
4. **Commentaires :**
   - Condition nécessaire et suffisante.

A.3 Tests basés sur le temps de réponse

A.3.1 Test R1 : algorithmes préemptifs à priorité fixe

1. **Méthode de calcul :**

$$r_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil \cdot C_j$$

2. **Références :** [JP86].
3. **Hypothèses :**
   - Algorithme préemptif à priorité fixe, quelque soit l'algorithme d'affectation des priorités.
   - Tâches à échéances sur requêtes.
   - Tâches indépendantes, concrètes et synchrones.
4. **Commentaires :**
   - $hp(i)$ est l'ensemble des tâches de plus forte priorité que i.
   - Condition nécessaire et suffisante de validité des tâches.

A.3.2 Test R2 : algorithmes préemptifs à priorité fixe

1. **Méthode de calcul :**

$$r_i = \max_{q=0,1,2,\ldots} (J_i + B_i + w_i(q) - q \cdot P_i)$$

avec

$$w_i(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{P_j} \right\rceil \cdot C_j$$

et

$$\forall q : w_i(q) \geq (q+1) \cdot P_i$$

2. **Références :** [JP86, ABRT93, TC94].

3. **Hypothèses :**
   - Algorithme préemptif à priorité fixe, quelque soit l'algorithme d'affectation des priorités.
   - Tâches à échéances quelconques, concrètes et synchrones.
   - Tâches dépendantes (ressources partagées et/ou contraintes de précédence exprimée avec une gigue).
4. **Commentaires :**
   - $hp(i)$ est l'ensemble des tâches de plus forte priorité que i.
   - Condition nécessaire et suffisante si la gigue et le temps de blocage sur ressources partagées ne sont pas employés. Condition suffisante seulement dans le cas contraire.

A.3.3 Test R3 : Earliest Deadline First préemptif

1. **Méthode de calcul :**

$$r_i = max_{a \in S}(L_i(a) - a)$$

avec :

$$L_i(a) = W(a, L_i(a)) + \left(1 + \left\lfloor \frac{a}{T_i} \right\rfloor\right) \cdot C_i$$

$$S = \bigcup_{j=1}^{n} \left(k \cdot T_j + D_j - D_i, 0 \leq k \leq \left\lfloor \frac{min(\lambda, L_i)}{T_j} \right\rfloor\right)$$

$$\lambda = \sum_{j=1}^{n} \left\lceil \frac{\lambda}{T_j} \right\rceil \cdot C_j$$

2. **Références :** [Spu96, GRS96].
3. **Hypothèses :**
   - Tâches à échéances quelconques.
   - Tâches non concrètes.
4. **Commentaires :**
   - Condition nécessaire et suffisante.

A.3.4 Test R4 : algorithmes non préemptifs à priorité fixe

1. **Méthode de calcul :**

$$r_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil \cdot C_j + max(C_k, \forall k \in lp(i))$$

2. **Références :** [JP86, GRS96].

3. **Hypothèses :**
   - Algorithme préemptif à priorité fixe, quelque soit l'algorithme d'affectation des priorités.
   - Tâches à échéances sur requêtes.
   - Tâches indépendantes, concrètes et synchrones.
4. **Commentaires :**
   - $hp(i)$ est l'ensemble des tâches de plus forte priorité que i.
   - $lp(i)$, l'ensemble des tâches de plus faible priorité que i.
   - Condition suffisante seulement.

A.3.5 Test R5 : Earliest Deadline First non préemptif

1. **Méthode de calcul :**

$$r_i = max_{a \in S}(C_i, L_i(a) - a)$$

avec :

$$L_i(a) = max_{D_j > a + D_i}(C_j) + \sum_{j \neq i, D_j \leq a + D_i} min\left(1 + \left\lfloor \frac{L_i(a)}{T_j} \right\rfloor, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor\right) \cdot C_j + \left\lfloor \frac{a}{T_i} \right\rfloor \cdot C_i$$

$$S = \bigcup_{j=1}^{n} \left(k \cdot T_j + D_j - D_i, 0 \leq k \leq \left\lfloor \frac{min(\lambda, L_i)}{T_j} \right\rfloor\right)$$

$$\lambda = \sum_{j=1}^{n} \left\lceil \frac{\lambda}{T_j} \right\rceil \cdot C_j$$

2. **Références :** [Spu96, GRS96].
3. **Hypothèses :**
   - Tâches à échéances quelconques.
   - Tâches non concrètes.
4. **Commentaires :**
   - Condition nécessaire et suffisante.

A.4 Calcul du temps maximal d'attente sur ressource partagée

Cette dernière partie ne contient pas, à proprement dit, des tests de faisabilité. On décrit ici les méthodes de calcul du temps d'attente maximal sur ressources partagées pour différents protocoles.

A.4.1 Test P1 : Priority Inversion Protocol

1. **Méthode de calcul :**

   B_i = somme des sections critiques des tâches moins prioritaires que i.

2. **Références :** [SRL90].

3. **Commentaires :**
   - Calcul de temps de blocage d'une tâche i pour une ressource accèdée selon PIP.
   - Une seule ressource possible : le protocole conduit à un interblocage si plusieurs ressources sont accèdées simultanément.

A.4.2 Test P2 : Priority Ceiling Protocol

1. **Méthode de calcul :**

   B_i = plus grande section critique du jeu de tâches employant le même ensemble de ressources.

2. **Références :** [SRL90].

3. **Commentaires :**
   - Calcul de temps de blocage d'une tâche i pour une ressource accèdée selon les différentes versions de PCP telles que OCPP ou ICPP.
   - Plusieurs ressources possibles sans interblocage.

A.4.3 Test P3 : Stack Resource Protocol

1. **Méthode de calcul :**

   B_i = plus grande section critique des tâches dont la période est plus grande que i et dont le niveau de préemption est supérieur ou égal au niveau de préemption que i.

2. **Références :** [Bak91, GN01].

3. **Commentaires :**
   - Calcul de temps de blocage d'une tâche i pour une ressource accèdée selon SRP.
   - Plusieurs ressources possibles sans interblocage.

Annexe B

Extended abstract of this thesis

Sommaire

This chapter is an extended abstract of this thesis. The text of this chapter is based on an article titled "Increasing the usability of real time scheduling theory : the Cheddar project" and which was submitted for publication in the journal of Real time systems, published by Springer Verlag. Authors of this article are Frank Singhoff, Alain Plantec, Pierre Dissaux and Jérôme Legrand.

B.1 Introduction

The Cheddar project deals with real time critical systems modelling and verification. Real time systems have to meet hard timing constraints implied by the environment ; safety critical systems failure, including violation of timing constraints, could lead to life losses or environmental damages [NMB04].

Real time scheduling theory provides algebraic methods and algorithms in order to make timing constraints verifications. Real time scheduling theory foundations were proposed in 1970 [LL73] and has led to extensive researches. Since 1990, it makes it possible to analyse systems composed of periodic tasks sharing resources and running on a single processor [SRL90]. Numerous operating systems provide features allowing the implementation of such applications. Some standards and compilers also provide tools to enforce compliancy of applications with real time scheduling theory assumptions, such as the Ravenscar profile defined in the Ada 2005 standard [TDB+06].

Real time scheduling theory has been successfully used in many projects [SEI03]. However, it appears that in many practical cases no such analysis is performed although experience shows that it could be profitable.

Several reasons can explain why real time scheduling analysis is not applied as much as it could be. First of all, real time scheduling analysis may be difficult to apply on some kind of architectures. For example, few analytical methods were proposed for distributed systems analysis [TC94]. Another reason is that the scope of existing analytical methods is usually restricted to specific architectures composed of simple schedulers or task models. In these cases, practical real time scheduling analysis solutions should at least provide means to properly model the system and run simulations.

Furthermore, it seems that this theory is not so easy to understand and to apply for many engineers. Most of the known analytical methods and algorithms have been elaborated during the last 30 years. These analytical methods allow to compute different performance criteria. Each criterion requires the target system fulfills a set of specific assumptions. Thus, it may be difficult for a designer to choose the relevant analytical method. Unfortunately, there is currently a too limited support provided by design languages and CASE tools which can concretely help to automatically apply real time scheduling theory.

This chapter presents a summary of the Cheddar project contributions and ongoing works. The Cheddar project investigates why real time scheduling theory is not used and how its usability can be increased. The Cheddar project was launched at the University of Brest in 2002. This chapter presents three possible directions that we have explored in order to increase the usability of real time scheduling theory. Section A.2 presents a set of tools that have been developed. These tools aim at helping the

designer to apply real time scheduling theory on an architectural model. Section A.3 depicts how the use of an Architecture Description Language can help to apply the real time scheduling theory more automatically. Section A.4 presents a domain specific language and a set of tools that can be used on specific real time architectures when no existing analytical method can be directly applied. Finally, section A.5 is devoted to conclusions and presentation of Cheddar project future work.

B.2 Increasing the usability of real time scheduling theory : easing analysis with flexible tools

Real time scheduling theory provides scheduling algorithms and algebraic methods usually called feasibility tests. These methods help the system designer to analyze the timing behaviour of his architecture. With the Liu and Layland real time task model [LL73], each task periodically performs a treatment. This "periodic" task is defined by three parameters : its deadline (D_i), its period (P_i) and its capacity (C_i). P_i is a fixed delay between two release times of the task i. Each time the task i is released, it has to do a job whose execution time is bounded by C_i units of time. This job has to be ended before D_i units of time after the task release time. From this task model, some feasibility tests can provide a proof that an architecture will meet its periodic task performance requirements. Scheduling algorithms allow the designer to compute scheduling simulations of the architecture. Usually, simulations can not lead to a proof. However, with deterministic schedulers and periodic tasks, scheduling simulation may be considered as schedulability proof if the designer is able to compute the scheduling during the hyperperiod [LM80].

Different kinds of feasibility tests exist such as tests based on processor utilization factor or tests based on worst case task response time. The worst case response time feasibility test consists in comparing the worst case response time of each task with their deadline. Joseph, Pandia, Audsley et al. [GRS96] have proposed a way to compute the worst case response time of a task with pre-emptive fixed priority scheduler by :

$$r_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil \cdot C_j \tag{B.1}$$

Where r_i is the worst case response time of the task i and $hp(i)$ is the set of tasks which have a higher priority level than i. This feasibility test must be extended to take into account task waiting time on shared resources, jitter on task release time or task precedence relationships.

In order to apply a feasibility test, the designer must check that his design and his execution platform fulfill all feasibility test assumptions. For example, in the feasibility test based on equation B.1, D_i must be less or equal than P_i and all tasks must have the same first release time. Thus, for a designer who has not a deep knowledge of real time scheduling theory, verifying an architecture with feasibility tests becomes a difficult task because, for each part of the architecture to verify, he must (see figure B.1) :

1. Choose the performance criterion he would like to check.

2. Find the right model for each entity of his architecture. For example, should he model a function of his architecture as a set of periodic tasks or as a set of sporadic tasks? The designer must select the right abstraction level which decreases the model complexity but takes into account properties required for analysis.
3. Select a feasibility test able to compute the criterion chosen in (1) and compliant with the models chosen in (2). For such a purpose, model and feasibility test assumptions have to be compliant.

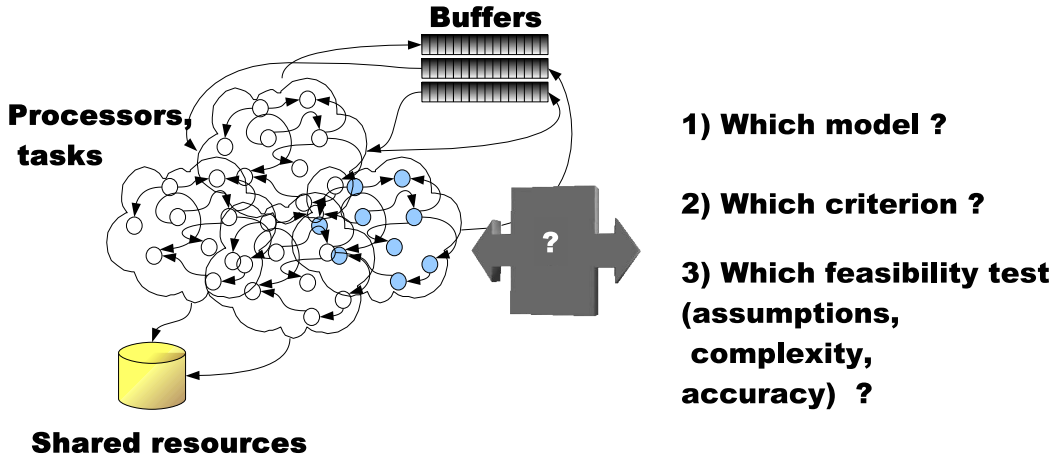

FIG. B.1: *From the architecture modelling to the analysis*

Of course, in some cases, this work can be quite simple since the studied architecture is simple too. In all the other cases, the toolset must provide advanced support to guide the user according to the level of complexity of the model to be processed. Several real time scheduling tools already exist such as MAST [HGGM01], Rapid-RMA [TP03] or Cheddar.

Cheddar is a GPL open-source toolset composed of a graphical editor and a library of processing modules (see figure B.2). The designer can design his real time architecture thanks to the Cheddar editor. However, it is also expected that designers perform the modelling activity with separate system or software engineering tools. The Cheddar library implements most of the current feasibility tests and the classical real time scheduling algorithms. This library also offers a domain specific language together with an interpreter and a compiler, for the design and the analysis of schedulers still not implemented into the library.

Cheddar offers different use levels depending on the architecture to analyze, on the separate modelling tool Cheddar is supposed to be connected to or on the designer knowledge. We have the following typical use cases :

– In a first one, an architecture model has just to be loaded into the Cheddar editor and a button has simply to be pushed to perform proper analysis. In this case, Cheddar chooses by itself the best feasibility test to apply, checks if the feasibility test assumptions are met and displays the

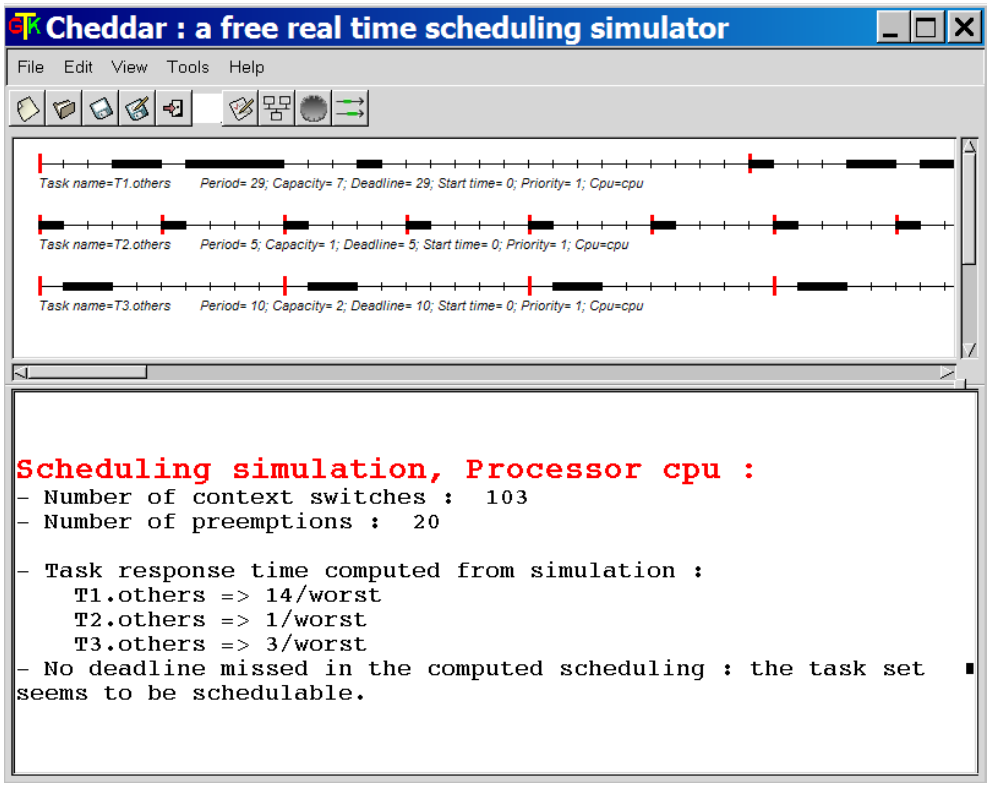

FIG. B.2: *Cheddar's GUI*

result. It is assumed that the designer makes use of a design pattern handled by Cheddar. This is the case for instance when the Ravenscar design pattern is used [TDB+06]. This first way to use Cheddar is also best suited for students who have to learn about real time scheduling foundations.

– A second way is to let the designer choose which performance criteria must be computed. This choice can be performed thanks to a customizable menu of the Cheddar user interface. Feasibility test assumptions are still automatically checked by Cheddar.
– Thirdly, if scheduling algorithms or feasibility tests implemented into Cheddar can not be applied, then the designer must extend the Cheddar library. Two ways exist for such a purpose. The library can be extended by the Cheddar domain specific language with the process explained in section B.4. Otherwise, the designer manually implements the performance analysis tools (in which case advanced knowledge about the Cheddar library is required).

```
DATA shared_resource_type
END shared_resource_type;
DATA IMPLEMENTATION shared_resource_type.Impl
   PROPERTIES
      Concurrency_Control_Protocol => Priority_Ceiling_Protocol;
END shared_resource_type.Impl;

THREAD task_type
   FEATURES
      get_access : REQUIRES DATA ACCESS shared_resource_type;
END task_type;
THREAD IMPLEMENTATION task_type.Impl
   PROPERTIES
      Dispatch_Protocol => Periodic;
      Period => 50;
      Compute_Execution_Time => 3 Ms .. 3 Ms;
      Cheddar_Properties::POSIX_Scheduling_Policy => SCHED_FIFO;
      Cheddar_Properties::Fixed_Priority => 5;
      Cheddar_Properties::Dispatch_Jitter => 10 Ms;
END task_type.Impl;

PROCESSOR a_cpu
END a_cpu;
PROCESSOR IMPLEMENTATION a_cpu.Impl
   PROPERTIES
      Scheduling_Protocol => Rate_Monotonic;
      Cheddar_Properties::Scheduler_Quantum => 1 Ms;
      Cheddar_Properties::Preemptive_Scheduler => True;
END a_cpu.Impl;

PROCESS a_proc
END a_proc;
PROCESS IMPLEMENTATION a_proc.Impl
   SUBCOMPONENTS
      th1 : THREAD task_type.Impl;
      th2 : THREAD task_type.Impl;
      r1  : DATA shared_resource_type.Impl;
   CONNECTIONS
      DATA ACCESS r1 -> th1.get_access;
      DATA ACCESS r1 -> th2.get_access;
END a_proc.Impl;

SYSTEM a_system
END a_system;
SYSTEM IMPLEMENTATION a_system.Impl
   SUBCOMPONENTS
      cpu0  : PROCESSOR a_cpu.Impl;
      proc0 : PROCESS   a_proc.Impl;
   PROPERTIES
      Actual_Processor_Binding => REFERENCE cpu0 APPLIES TO proc0;
END a_system.Impl;
```

FIG. B.3: *Example of an AADL model*

Many other ways to use a toolset such as Cheddar exist. For example, Cheddar can be associated with system or software engineering tools such as Stood [DS08] or Ocarina [HZPK07] in order to increase its usability. In this case, the designer does not use the Cheddar graphical editor anymore and the Cheddar library is directly called by connected tools. Cheddar exports analysis results as an XML data stream which can be processed back into the modelling tools display. The next section presents how the use of an architecture description language can facilitate modelling and analysis tool interoperability.

B.3 Increasing the usability of real time scheduling theory : from the engineering process to the performance analysis

A possible way to help the designer to apply real time scheduling theory, is to embed knowledge on this theory into the engineering process with the help of design languages and design patterns.

Panunzio and Vardanega have proposed a metamodel which permits the execution of timing analysis [PV07]. A UML profile called MARTE which allows such a timing analysis is also currently investigated by Frédéric et al. [FGD06].

The SAE Architecture Analysis and Design Language (AADL) is a textual and a graphical language support for model-based engineering of embedded real time systems. AADL has been approved and published as SAE Standard AS-5506 [SAE04]. AADL is used to design and analyze software and hardware architecture of embedded real time systems.

In the Cheddar project context, AADL was chosen to investigate how real time scheduling theory can be automatically applied. As Cheddar provides the most known real time scheduling feasibility tests and scheduling algorithms, it was primarily used to check first AADL standard real time scheduling theory compliance. Then, we have investigated how memory footprint analysis can be conducted with AADL [SLNM05]. Finally, some design patterns expressed in AADL were proposed. For each design pattern, we are investigating the performance criteria that Cheddar is able to automatically compute. The use of these design patterns enforces scheduling analysis compliance : an architecture which is composed of instances of these design patterns can then be analyzed by Cheddar. This approach also eases interoperability between AADL editor tools and AADL analysis tools [DS08].

In the sequel, we present the contributions of the Cheddar project to the performance analysis of AADL models.

B.3.1 Investigating AADL suitability for real time scheduling theory

An AADL model is a set of hardware and software components such as data, threads, processes (architecture software side), processors, devices and busses (architecture hardware side). A data component may represent a data structure in the program source. An AADL data component can be implemented by an Ada tagged record or by a C++ class. A thread is a sequential flow of control that executes a program. A thread can sequentially calls subprograms. An AADL thread can be implemented by an Ada task or by a POSIX thread. AADL threads can be released according to several policies : a

thread may be periodic, sporadic or aperiodic. An AADL process models an address space. An AADL operational system instantiates a set of process components encompassing thread and data components that are bound to an execution platform composed of processor, memory and bus components.

An AADL model can also contain properties and component connections. Component connections allow the components to share data or exchange messages. Properties are assigned to components. Information provided by component properties can be related to the component behavior, its state, the way it will be implemented or anything else that makes it possible to perform analysis. A property is defined by a name, a value and a type.

Figure B.3 shows an AADL model. This model contains a shared resource (called $R1$) accessed by two threads (threads $th1$ and $th2$). The threads and the shared resource are defined into an address space (process $proc0$). The process $proc0$ is bound to a processor called $cpu0$.

The first release of the AADL standard provides component properties required in order to apply the simplest real time scheduling analysis methods. Nevertheless, some properties were missing to apply several usual real time scheduling theory analysis methods. AADL provides a way to extend the AADL standard property sets. We have proposed a set of property extensions [Sin07b] to model :

- Usual properties of real time schedulers in order to define if the scheduler works preemptively or not ($Cheddar\_Properties$::$Preemptive\_Scheduler$ property), if the scheduler makes use of a quantum ($Cheddar\_Properties$::$Scheduler\_Quantum$ property) or runs a POSIX 1003.1b scheduler ($Cheddar\_Properties$::$POSIX\_Scheduling\_Policy$ property).
- Usual thread properties such as fixed priority, jitter, offset, first release time, shared resource blocking time, context switch overhead, criticality ($Cheddar\_Properties$::$Fixed\_Priority$, $Cheddar\_Properties$::$Dispatch\_Jitter$, $Cheddar\_Properties$::$Dispatch\_Offset$ properties) ...
- Properties to define when shared resources are accessed by threads ($Cheddar\_Properties$::$Critical\_Section$ property).
- And finally, AADL version 1 leading to some ambiguities, we have proposed some properties to express thread precedence relationships which can not be computed from standard AADL connections.

Some of the lacks presented above will be fixed in the next AADL standard version (AADL version 2) with the Behavioral Annex [SAE07, BDF+06] and with some of Cheddar properties which will be included in the standard AADL property set.

B.3.2 Memory footprint analysis with AADL

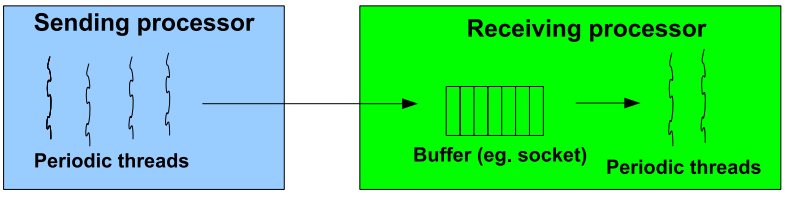

FIG. B.4: *Part of a distributed system*

One of the most interesting part of an architecture design language such as AADL is that it allows performance analysis on multiple resources. This is especially mandatory with distributed real time systems which may be composed of several processors, memory units and communication devices. The figure B.4 shows a distributed system composed of two processors exchanging messages through a TCP/IP socket. With such a system, performance analysis on processors and memory units can not be performed independently :

- In the one hand, if the periodic receiving/sending threads have a high priority level, then they have constant response times. With constant response times, the amount of memory required in the socket to store messages may be low.
- In the other hand, when sending or/and receiving threads have a low priority level, their response times may vary at execution time. With varying response times, the amount of memory required in the socket to store messages may be higher if no message have to be lost.

With AADL, thread synchronization and communication are expressed through port abstraction. Ports are logical connection points between components that can be used for control and data transfer between threads. Three kinds of ports exist : *event ports*, *data ports* and *event data ports*. Event data ports represent connection points for message transfer. With event data ports, messages may be queued before beeing read by consumer threads.

In the sequel, we present Cheddar memory footprint analysis tools which can be applied to AADL threads connected by event data ports. We assume that AADL consumer threads are periodic and AADL producer threads may be aperiodic or periodic. Consumers and producers are both scheduled with real time schedulers such as Rate Monotonic.

B.3.2.1 Queueing system theory

The queueing system theory makes it possible to study performances of a system composed of servers, customers and storage places [Kle75b] : people waiting in a room for a doctor, network switch routing data, ... A queueing system models a system composed of a server and a queue. When a customer arrives in the system and if the server is idle, the server immediately performs the customer request. When the server is busy, the customer requests are stored in the queue.

By defining average customer arrival rate and average request rate that the server can handle, queueing system theory allows the designer to compute many performance criteria.

Queue	L	W	P_n
M/M/1	$\frac{\lambda W_s}{1-\rho}$	$\frac{W_s}{1-\rho}$	$(1-\rho)\rho^n$
M/G/1	$\lambda W_s + \frac{\lambda^2(W_s^2+\sigma_s^2)}{2\cdot(1-\rho)}$	$\frac{\lambda(W_s^2+\sigma_s^2)}{2\cdot(1-\rho)}$	-
M/D/1	$\lambda W_s + \frac{\lambda^2 W_s^2}{2\cdot(1-\rho)}$	$W_s + \frac{\lambda W_s^2}{2\cdot(1-\rho)}$	-

TAB. B.1: *Main performance criteria*

Different customer inter-arrival time distributions and service time distributions exist. The most usuals are Deterministic, Markovian and General distributions :

- A Deterministic distribution means constant delay between two customer arrivals and constant customer service times.
- A Markovian describes a customer arrival rate or a service time with an exponential probability distribution.
- Finally, if no assumption on probability distribution is done, the distribution is called General. A General distribution is only defined by an average rate and its variance.

Following Kendall notation, a queueing system is described by at least 3 parameters : $a|b|c$. The a parameter is the customer arrival rate. b describes the service time rate. Finally, c is the number of servers. For instance, a system with one server, a constant service time and an exponential client arrival is a M/D/1 queueing system. M stands for a Markovian distribution and D stands for a Deterministic distribution. Furthermore, a General distribution is usually noted G.

Table B.1 gives main performance criteria for most usual queueing systems. In this table, λ represents the customer arrival rate, ρ the queueing system utilization factor, W_s and σ_s^2, respectively, the average customer service time and its variance. With these parameters, one can compute average customer number in the queueing system (L), average customer waiting time in the queueing system (W), and probability of having n customers in the queue (P_n).

B.3.2.2 Memory requirements with AADL threads connected by event data port

To study event data port memory requirements when connected AADL threads are scheduled according to a real time scheduler like Rate Monotonic, we propose a new service time distribution : the P distribution. Port buffers are modeled using queueing systems. The P distribution models the fact that periodic consumer/producer threads are scheduled with a real time scheduler. The P distribution assumes that thread release times are not synchronized with message arrivals : an AADL consumer thread is periodically released even if no message has arrived in the buffer.

Thanks to the P distribution, two new queueing systems are defined : P/P/1 and M/P/1. An exact resolution of P/P/1 is given and we provide an approximation of M/P/1 [LSNM04]. This approximation is based on a M/G/1 queueing model.

AADL models with event data ports can then be studied by both worst case and average case analysis. Worst case analysis can be performed if assumptions are made on message arrival rate. In this case, the system is checked with P/P/1 assuming that the highest message arrival rate is known. Otherwise, if no worst case assumption is made, an average analysis can be realized with M/P/1.

B.3.2.3 Average buffer performance analysis : producer threads are aperiodic

In this section, we assume that messages arrive in the event data port buffer at a random rate. This case occurs when AADL producer threads are aperiodic.

In the sequel, according to the Kendall notation, a buffer receiving random rate messages and accessed by an AADL periodic consumer thread will be modeled with a M/P/1 queueing system [Kle75b, Rob90]. Messages are served in a FIFO manner : the earlier a message arrives, the earlier it is served.

We propose an approximation of the M/P/1 queueing system. This M/P/1 approximation consists in evaluating its average service time W_s and its variance σ_s^2. With W_s and σ_s^2, a M/P/1 queueing system can be modeled with a M/G/1 queueing system. Then, M/P/1 message waiting time and buffer message number can be computed with the following M/G/1 equations [Kle75b] :

$$W = W_s + \frac{\lambda(W_s^2 + \sigma_s^2)}{2(1-\rho)}$$

$$L = \lambda W_s + \frac{\lambda^2(W_s^2 + \sigma_s^2)}{2(1-\rho)}$$

where ρ is the queueing system utilization factor and λ, the message arrival rate.

M/P/1 mean service time and its variance are [LSNM04] :

Théorème 1 *The M/P/1 average service time is equal to :*

$$W_s = \frac{P_{cons}}{2}(1+\rho) = \frac{P_{cons}}{2(1-\lambda\frac{P_{cons}}{2})}$$

And average service time variance is :

$$\sigma_s^2 = \rho \cdot \left(\frac{1}{n} \sum_{i=1}^{n} S_i^{'} - W_s^2 \right) + (1-\rho) \cdot \frac{P_{cons}^2}{12}$$

Where $\rho = \lambda W_s$, $S_i^{'}$ are the service time when ρ tends to 1 and P_{cons} is the consumer thread period. Due to the fact that the server models a periodic consumer thread, the number of service time $S_i^{'}$ is bounded by n.

B.3.2.4 Worst case buffer analysis : producer threads are periodic

We now study a system where event data port buffer production and consumption are assumed to be periodic. This case occurs when AADL producer threads are periodic.

According to Kendall notation, a buffer shared by n periodic producer AADL threads and 1 periodic consumer AADL thread scheduled with a real time scheduler can be modeled with a P/P/1 queueing system. Messages are served in a FIFO manner.

Some similarities exist between this system and voice transmission service provided by ATM networks AAL1 layer [GK96]. In order to solve our P/P/1 queueing system, we apply results from this ATM layer.

For a buffer shared by n periodic producer threads and one periodic consumer thread, buffer bound is given by [LSN+03] :

Théorème 2 *For a P/P/1 buffer shared by an harmonic threads set* [24] *and $\forall i : D_i \leq P_i$, maximum buffer size and maximum waiting time are respectively :*

[24] A thread set is said to be harmonic if and only if each thread period is a positive integer multiple of all smaller thread periods.

$$L = 2 \cdot n$$
$$and$$
$$W = 2 \cdot n \cdot P_{cons}$$

For a non harmonic threads set, maximum buffer size and maximum waiting time are respectively :

$$L = 2 \cdot n + 1$$
$$and$$
$$W = (2 \cdot n + 1) \cdot P_{cons}$$

B.3.2.5 Ada implementation of AADL memory footprint analysis tools

In order to implement AADL event data port memory requirement analysis tools, we wrote a set of object oriented Ada packages within Cheddar library. These packages allow to compute classical queueing system theory criteria (see table B.1) for usual queueing systems and also for M/P/1 and P/P/1 (see theorem 1 and 2). Besides of these analytical tools, Cheddar library also provides a set of Ada packages in order to design, write and run simulations when no analytical criterion can be computed for a given queueing system.

B.3.3 Towards interoperability between AADL tools

In the previous sections, we have presented several analytical methods which can be applied on AADL models. These analytical methods are not easy to apply alone and we believe that engineers must be helped by modelling tools. Coupling of modelling and analysis tools requires that both ends strictly comply with the same semantic definition of the exchanged model. This is particularly important for real time systems and software architectures. Such a guaranty can be brought by use of AADL standard all along the tool-chain. As an example, Cheddar has been well coupled with the modeling tool Stood [DS08] using AADL as a pivot language.

Stood was chosen because it provides an extended support for AADL in addition to its compliance with the HOOD methodology [BW94]. Stood makes it possible to manage a complete software project by building libraries of reusable components, reversing legacy code and specifying the real time application as well as its execution platform. Most of the modelling activities can be performed graphically and the corresponding AADL code is automatically generated by the tool.

B.3.3.1 A set of AADL design patterns

To ease interoperability between Stood and Cheddar, we have proposed a set of AADL design patterns. For each pattern, we are investigating the performance criteria that Cheddar is able to automatically compute. This set of design patterns models usual real time synchronization/threads-communication paradigms (e.g. ARINC 653 [Ari97]) :

1. **Synchronous data-flows design pattern :** this first design pattern is the simplest one. The data sharing is achieved by a clock synchronization of the threads as Meta-H [SAE04] proposed it.

In this synchronization schema, thread dispatch is not affected by the inter-thread communications that are expressed by pure data-flows. Each thread reads its input data ports at dispatch time and writes its output data ports at complete time. This design pattern does not require the use of a shared data component. In this simple case, the execution platform consists in one processor running a scheduler such as Rate Monotonic [LL73].

2. **Ravenscar design pattern :** main drawback of the previous pattern is its lack of flexibility at run time. Each thread will always execute, read and write data at pre-defined times, even if useless. In order to introduce more flexibility, asynchronous inter-thread communications can be proposed. An example of such a run-time environment is given by the Ravenscar profile. Ravenscar is a part of the Ada 2005 standard [TDB+06]. It is a set of Ada program restrictions usually enforced at compilation time, which guaranties that the software architecture is real time scheduling theory compliant. Ravenscar is an Ada subset with which real time applications are composed of a set of tasks and shared data. Ravenscar assumes tasks are scheduled with a fixed priority scheduler and shared data are accessed with ICPP (ICPP stands for Inheritance Ceiling Priority Protocol).
3. **Blackboard design pattern :** Ravenscar allows a thread to allocate/release several shared resources (e.g. AADL data component). Real time scheduling theory usually models such a shared resource as a semaphore, to represent, for example, a critical section. In classical operating systems, there exists many synchronization design patterns such as critical section, barrier, readers-writers, private semaphore, and various producers-consumers [Tan01]. The blackboard design pattern implements a readers-writers synchronization protocol. At a given time, only one writer can get the access to the blackboard in order to update the stored data, as opposed to the readers which are allowed to read the data simultaneously. The usual implementation of this protocol implies that readers and writers do not perform the same semaphore access, that requires extra analysis.
4. **Queued buffer design pattern :** in the blackboard design pattern, at any time, only the last written message is made available to the threads. Some real time execution platforms provide communication features which allow to store all written messages in a memory unit. AADL also proposes such a feature with event data ports or shared data components.

B.3.3.2 The AADL Behavior Annex

For each pattern, an applicative test case was described under the form of an AADL model which has been formatted in purpose to highlight some of the possible performance analysis that Cheddar is able to automatically compute (thread worst case response time, bound on shared resource blocking time, memory footprint analysis, ...) [DS08].

Indeed, performing proper analysis on real time architectures making use of communication paradigms such as those expressed in the previous section requires that enough details are given about the awaited scheduling conditions and internal behavior of each thread and subprogram.

```
SUBPROGRAM Switch
   FEATURES
      quick : IN PARAMETER Base_Types::Boolean;
END Switch;

SUBPROGRAM Short_Action
   PROPERTIES
      Compute_Execution_Time => 1 MS .. 2 MS;
END Short_Action;

SUBPROGRAM Long_Action
   PROPERTIES
      Compute_Execution_Time => 100 MS ..150 MS;
END Long_Action;

SUBPROGRAM IMPLEMENTATION Switch.NoBA
   CALLS {
      c1 : SUBPROGRAM Short_Action;
      c2 : SUBPROGRAM Long_Action;
   };
-- Without a B.A., an analysis tool would set the capacity of Switch to at
-- least 150 ms whereas it only occurs when the parameter value is False.
END Switch.NoBA;

SUBPROGRAM IMPLEMENTATION Switch.BA
ANNEX behaviour_specification {**
   STATES
      s0 : INITIAL STATE;
      S1 : RETURN STATE;
   TRANSITIONS
      s0 -[ ON quick ]-> s1 { Short_Action!; };
      s0 -[ ON NOT quick]-> s1 { Long_Action!; };
**};
-- With a B.A., an analysis tool could set the capacity to a more precise
-- value corresponding to each case.
END Switch.BA;
```

FIG. B.5: *Modelling precise thread capacity*

Such a behavior may be implicit. This is the case when the default run-time semantics of the core definition of the AADL is used. For instance, thread scheduling may be controlled at an architectural level by setting the appropriate value to the predefined AADL property *Scheduling_Protocol*. AADL version 1 run-time semantics handles periodic, sporadic and aperiodic threads. Similarly, interaction with shared data components can be specified thanks to data access features within thread or subprogram declarations. This is the case of the AADL model example of figure B.3. The AADL standard defines the effect of such declarations in terms of calls to a default run-time execution service (*Raise_Event*, *Await_Dispatch*, *Get_Resource*, *Release_Resource*, ...). Most of the time, the end user will thus be able to avoid such a low level description of the behaviour of the application and rely on the higher level architectural constructs provided by the core of the standard. However, a more explicit description of this behaviour may be sometimes required. AADL Behavior Annex is being defined to cover this requirement.

As mentioned previously, the AADL language may be extended by specific property sets (as opposed

to the predefined property set specified by the core of the language). This is the simplest extension mechanism that is proposed by the standard which enables the addition of new properties to all the existing constructs of the language. When this is not sufficient, the solution consists in defining an annex to the language.

Annexes to the AADL language can be seen as sublanguages that can be used to provide more detailed descriptions on specialized topics, or to experiment possible future extensions to be included in a later release of the standard. They may define their own syntax that can be embedded inside core AADL statements thanks to an appropriate escape sequence.

In order to cope with fine real time analysis of architectural models, definition of a Behavior Annex for the AADL is in progress and will be submitted to the standardization process. The origin of the AADL Behavior Annex is the COTRE project [FD02, BRV+03, FG05]. The AADL Behavior Annex provides a sublanguage extension to allow behavior specifications to be attached to AADL components that are expressed as state transition systems with guards and actions [BDF+06, DBF+06]. This can be seen as a refinement to call sequences defined by the core of the language. Typical improvements brought by the use of the Behavior Annex are :

- A more precise value for threads capacities.
- A more precise value for critical section durations.
- A more precise definition of thread dispatch conditions.

One first usage of the Behavior Annex extension is to enable expression of conditional actions depending on the actual value of data received in a thread port or a subprogram parameter. This feature is useful to provide a finer expression of the thread capacity, also called worst case execution time and which value is handled by the *Compute_Execution_Time* predefined AADL property (see figure B.3).

The example of figure B.5 shows such a refinement. If only the core language is used, the description of the functional behaviour of the subprogram is limited to a call sequence. For the purpose of real time analysis, only the worst case execution time will be considered. In this case, it will probably be the value of the *Compute_Execution_Time* property associated to the *Long_Action* subprogram. On the contrary, if a Behavior Annex subclause is added, an analysis tool may be able to allocate a smaller execution time to this subprogram when the parameter is known to be equal to True, thus resulting in a finer real time analysis.

A second usage of the AADL Behavior Annex deals with a more precise definition of critical sections (see example of figure B.6). When the action block of a transition contains an explicit access to a shared data component, then only the corresponding action block will be considered as a critical section. If no Behavior Annex subclause is defined, the risk is that an analysis tool evaluates the critical section duration to the complete execution time of the thread.

```
THREAD update
   FEATURES
      buffer : REQUIRES DATA ACCESS T_buffer;
END update;

THREAD IMPLEMENTATION update.NoBA
   CALLS {
      c1 : SUBPROGRAM Long_Computation;
   };
-- Without a B.A., an analysis tool could set the critical section duration
-- to the same value as the complete Thread duration.
END update.NoBA;

THREAD IMPLEMENTATION update.BA
ANNEX behaviour_specification {**
   STATE VARIABLES
      v1, v2 : T_buffer;
   STATES
      s0 : INITIAL STATE;
      s1, s2 : STATE;
      s3 : COMPLETE STATE;
   TRANSITIONS
      s0 -[]-> s1 { v1 := buffer; };
      s1 -[]-> s2 { Long_Computation!(v1,v2); };
      s2 -[]-> s3 { buffer := v2; };
**};
-- With a B.A., an analysis tool could isolate the actual
-- critical sections.
END update.BA;
```

FIG. B.6: *Modelling precise shared data access*

A third possible use of the Behavior Annex comes from the fact that the evaluation of the transition guards can be used to define a more precise thread dispatch condition. In the example of figure B.7, the thread dispatch condition can be specified either by the predefined AADL property *Dispatch_Protocol* or by a Behavior Annex subclause. In the first case, it may be understood that the thread can always be dispatched on the arrival of any of its input ports. In the second case, a more precise behavior can be expressed to explicitly show that the receptivity of each port can depend on the internal state of the serving thread, thus changing its actual dispatch conditions.

With these three examples, we saw how the Behavior Annex can be used to improve the expressiveness of AADL architectures to perform proper real time analysis. Additional improvements will be brought by AADL version 2 which release is planned at the end of year 2008. These enhancements include in particular new predefined scheduling protocols and a new category of components to manage software partitions in a distributed system.

```
THREAD FlipFlop
   FEATURES
      set : IN EVENT PORT;
      reset : IN EVENT PORT;
END FlipFlop;

THREAD IMPLEMENTATION FlipFlop.NoBA
   PROPERTIES
      Dispatch_Protocol => Sporadic;
-- Without a B.A., an analysis tool could consider that
-- the Thread can be dispatched on any event port.
END FlipFlop.NoBA;

THREAD IMPLEMENTATION FlipFlop.BA
ANNEX behavior_specification {**
   STATES
      On, off : INITIAL COMPLETE STATE;
   TRANSITIONS
      on -[set?]-> off { };
      off -[reset?]-> on { };
**};
-- With a B.A., dispatch conditions can vary according to the internal
-- state of the Thread.
END FlipFlop.BA;
```

FIG. B.7: *Modelling thread dispatching rules*

B.4 Increasing the usability of real time scheduling theory : when no feasibility test exists

In the previous section, we assumed that the architecture to analyze can be designed as a set of design-pattern instances. Since each design-pattern can be analyzed with a set of feasibility tests, the overall architecture can actually be analyzed with such analitycal tools.

But many practical cases can not be analyzed with design-patterns. Complex industrial real time architectures frequently make use of specific task models or schedulers. In this case, no feasibility tests exists and building new ones is a difficult and expensive work. Furthermore, industrial real time systems may be composed of a large number of entities (e.g. tasks, processors, memory units ...). For now, these large scale systems can not be efficiently analyzed with model-checking. The only way people can expect to verify performances of these real time systems is to perform analysis with extensive simulations.

Languages and models were proposed for such a purpose. CPN tools [Wel06] provides simulation features based on Petri Net for example. Unfortunately, the use of these general purpose simulation tools usually implies that the designer must model real time scheduling low level abstractions such as task preemption. A second way is to develop ad-hoc simulation programs, but this solution implies a very low level of reusability. The Cheddar library proposes a third way by the use of a domain specific language and a set of tools (compiler, interpreter, code generator ...). This domain specific language allows the designer to build models of his scheduler and task models.

We also propose an engineering process from which the designer can test his models and automatically generate a simulation program. This model driven engineering process is implemented with the

help of a software engineering tool called Platypus [PR98].

B.4.1 A language for the modelling of real time schedulers

Real time schedulers are composed of two different aspects :

1. Arithmetic and logical statements which allow to select a task among a set of ready tasks or to compute task priorities.
2. Temporal constraints and synchronization between entities (e.g. tasks and schedulers). These synchronizations describe how entities must work all together in order to share processors.

The Cheddar language is then defined by two parts : 1) an Ada like language for the modelling of arithmetic and logical statements of the schedulers and 2) a timed automaton language for the synchronizations modelling scheduler and task relationships. A detailed description of this language is given into the Cheddar users's guide [Sin07a].

B.4.1.1 Modelling arithmetic and logical statements

This part of the Cheddar language allows to express the arithmetic and logical statements on simulation data. Simulation data are associated to the entities composing the architecture to analyze (e.g. task release time, scheduler quantum, shared resource protocol, ...). This language allows the designer to express sort rules as Earliest Deadline for example. A Cheddar program is organized in subprograms called sections. These subprograms are typed :

- Some subprograms are devoted to data simulation declaration and initialization. They are called *start_section*.
- Some subprograms allow to select a task among a set of ready tasks according to simulation data (e.g. priority). These subprograms are called *election_section*.
- Finally, some subprograms contain statements which have to be ran at each unit of time before the task selection. They are called *priority_section*.

The language defines usual operators and statements. Schedulers can be modeled with loops, conditional tests or assignments. This domain specific language also provides statements and operators that are specific to real time scheduling theory. For example, the $uniform/exponential$ statements customize the way random values are generated during simulations; the *lcm* operator computes least common multiplier of simulation data; the *max_to_index* operator looks for the ready task which has the highest priority level ...

The language is typed and provides usual types as integer, boolean or string. Some types related to real time scheduling theory are also defined.

B.4.1.2 Modelling timing and synchronization relationships

The second part of a Cheddar program is a network of timed automata. A Cheddar scheduler model can contain timed automata similar to those proposed by UPPAAL [AD90, BDL04]. UPPAAL is a toolbox for the modelling and the verification of real time systems.

Timed automata are frequently used to express timing and synchronization requirements of real time systems. There are some experiments to model and verify real time schedulers with timed automata [AGS02, SGSS06, IKL+99]. Numerous tools exist (editors, simulators and model-checkers such as UPPAAL [BDL04] or Esterel Studio [Ber05]) and some standards are also based on such a formal model (e.g. UML Statecharts [Dru06]).

A network of timed automata models timing and synchronization between schedulers and tasks. The Ada like language described above is enough to model schedulers which have fixed synchronization relationships between tasks and schedulers. By the past, we have shown that this language allows the modelling of simple schedulers like Earliest Deadline First, Rate Monotonic or Maximum Urgency First. However, some real time schedulers require the modelling of complex synchronizations. This is the case, for example, of hierarchical schedulers[25].

In Cheddar language context, every automaton may fire a transition separately or synchronize with another automaton. Transitions may be guarded with time constraints. Delays can express time consumption at transition firing. Finally, at transition firing, automata may run Ada like sections in order to compute task priorities or to choose the next task to run.

The next section describes an example of Cheddar program that models an ARINC 653 hierarchical scheduling [SP07a].

B.4.2 Cheddar program examples : an illustration with the hierarchical ARINC 653 scheduler

Hierarchical scheduling has been initially proposed in the context of time sharing systems. In time sharing systems, hierarchical schedulers were proposed in order to define user-level scheduling policies (e.g. fair process scheduling [KL88] or user-level and kernel-level threads scheduling into Solaris [Vah96]). If user-level scheduling capability stays a motivation for the use of hierarchical schedulers, system designers mostly focus on hierarchical scheduling in order to reduce system design cost and to increase the sharing resource efficiency. Today, it is usual to share a processor between several applications. This allows old applications to run efficiently on newer processors without being re-designed (e.g. re-design of the scheduling). Applications sharing a processor can have different resource requirements. For example, in a real time multimedia application, a given scheduler may support critical tasks for audio and video presentation while uncritical tasks can be managed by a different scheduler which does not provide deterministic task response time. Today, hierarchical scheduling also exists in several real time system standards such as ARINC 653, POSIX 1003 or Ada 2005 [Gal95, TDB+06, Ari97].

[25] An architecture based on hierarchical scheduling is an architecture in which several entities work all together for the processor sharing.

```
start_section start1 :
  partition1_capacity : integer := 4;
  partition1_duration : clock := 0;
  quantum : integer :=2;
  my_prio : array (tasks_range) of integer;
  my_prio:=0;
end section;

priority_section partition1_priority :
   quantum:=quantum-1;
   if quantum = 0
      then quantum:=2;
           my_prio(previously_elected):=my_prio(previously_elected)+1;
   end if;
end section;

election_section partition1_election :
   return min_to_index(my_prio);
end section;

automaton_section partition1_scheduler :
  Pended : initial_state;
  Ready : state;
  Wait_Priority : state;

  transition Pended ⟹ [,partition1_duration:=0;,wakeup1!] ⟹ Wait_Priority;
  transition Wait_Priority ⟹ [partition1_duration<partition1_capacity,,partition1_priority!]
     ⟹ Ready;
  transition Ready ⟹ [,,partition1_election!] ⟹ Wait_Priority;
  transition Wait_Priority ⟹ [partition1_duration=partition1_capacity,,] ⟹ Pended;
end section;
```

FIG. B.8: *Cheddar program modelling the partition 1 scheduler*

From a performance point of view, real time hierarchical schedulers raise two difficult challenges.

The first challenge is related to the large number of hierarchical schedulers which were proposed. These hierarchical schedulers have complex and different ways to perform communication and synchronization relationships between the schedulers and the tasks [ABLL92]. It is also difficult to express scheduler requirements and behaviors in order to combine themself for example [LMD04, RS01]. Then, in contrary to the usual schedulers such as Earliest Deadline First or Rate Monotonic, it is difficult to implement into the Cheddar library a set of hierarchical schedulers satisfying most of system designer needs.

The second challenge is related to the availability of analytical methods for hierarchical schedulers : currently few feasibility tests exist in real time hierarchical scheduling context [AP04, HP03, SL03, DB05]. Building feasibility tests is usually a difficult work and it is more complex for hierarchical schedulers.

ARINC 653 is an avionic standard which provides such a hierarchical scheduling. An ARINC 653 system is a set of applications called partitions. Each partition is composed of tasks. The processor sharing is made according to a two-level hierarchical scheduling :

1. The partitions are cyclically activated. A task can be run only when its partition is activated.

```
start_section start2 :
   partition2_capacity : integer := 6;
   partition2_duration : clock := 0;
end section;

election_section partition2_election :
   return min_to_index(tasks.priority);
end section;

automaton_section partition2_scheduler :
  Ready : state;
  Pended : initial_state;

  transition Pended ⟹ [,partition2_duration:=0;,wakeup2!] ⟹ Ready;
  transition Ready ⟹ [partition2_duration<partition2_capacity,,partition2_election!]
      ⟹ Ready;
  transition Ready ⟹ [partition2_duration=partition2_capacity,,] ⟹ Pended;
end section;
```

FIG. B.9: *Cheddar program modelling the partition 2 scheduler*

```
start_section start_processor :
      partition_clock : clock:=0;
end section;

automaton_section processor_scheduler :
  Schedule_Partition2 : initial_state;
  Schedule_Partition1 : state;
  Restart : state;

  transition Schedule_Partition2 ⟹ [,,wakeup2?] ⟹ Schedule_Partition1;
  transition Schedule_Partition1 ⟹ [partition_clock=6,,wakeup1?] ⟹ Restart;
  transition Restart ⟹ [partition_clock=10,partition_clock:=0;,] ⟹ Schedule_Partition2;
end section;
```

FIG. B.10: *Cheddar program modelling the partition scheduler*

This first level of scheduling is fixed at design time : it is usually statically computed.

2. The second scheduling level is related to the task scheduling : tasks of a given partition are scheduled all together with a fixed priority scheduler. This task scheduling is an online scheduling.

With Cheddar, an ARINC 653 hierarchical scheduler is modeled as a set of Cheddar programs. A textual representation of those programs is given in the sections B.8, B.9 and B.10. A graphical representation is also given in figures B.11, B.12 and B.13.

The automaton of the figure B.13 specifies when each partition has to be active or not. This automaton models the first level of an ARINC 653 scheduling : the partition scheduling. The automata of the figures B.11 and B.12 model the schedulers of each partition. Such schedulers are responsible for the scheduling of the tasks of the modeled ARINC partition. These automata model the second level of an ARINC 653 scheduling : the task scheduling.

The automata modelling the task scheduling of each partition have two types of location :

1. The *Pended* locations. From these locations, the partitions can not get access to the processor in

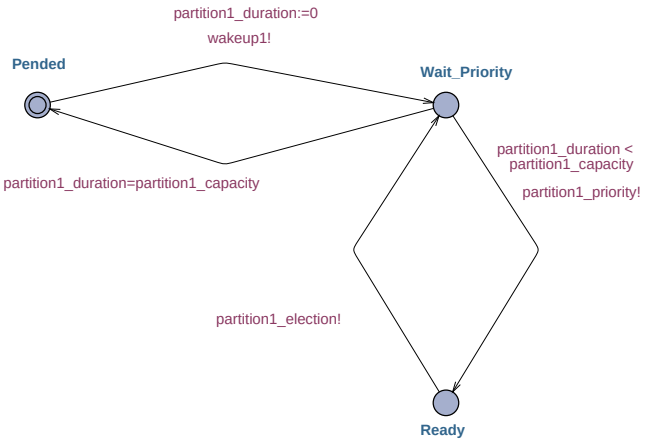

FIG. B.11: *Automaton modelling the task scheduler of the partition 1*

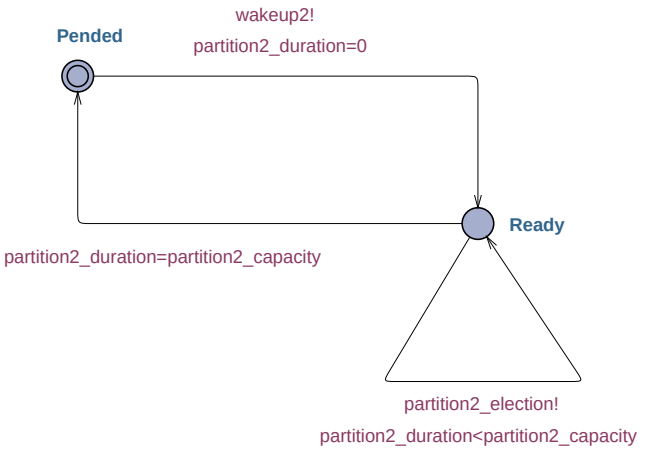

FIG. B.12: *Automaton modelling the task scheduler of the partition 2*

order to run one of their task.

2. The *Wait_Priority* and the *Ready* locations. If a partition is in one of these locations, it is allowed to run one of its task. *Wait_Priority* is an intermediate location from which the scheduler computes task priorities. Task priorities are computed by the *partition*1*_priority* section during the firing of the *Wait_Priority* outgoing transition (see the Cheddar program of section B.8). The *Ready* location chose the task to run during the next unit of time. To find the next task to run, the Cheddar program interpreter calls the *partition*1*_election* section during the firing of the *Ready* outgoing transition.

The partition scheduler automaton (see figure B.13) models the cyclic partition activation : in this example, the partition scheduling is made on a 10 units of time cycle. Each cycle, the partition 2 is activated during the 6th first units of time and the partition 1 is activated during the 4th last units of time. We assume that the partition 2 schedules critical periodic tasks according to Rate Monotonic

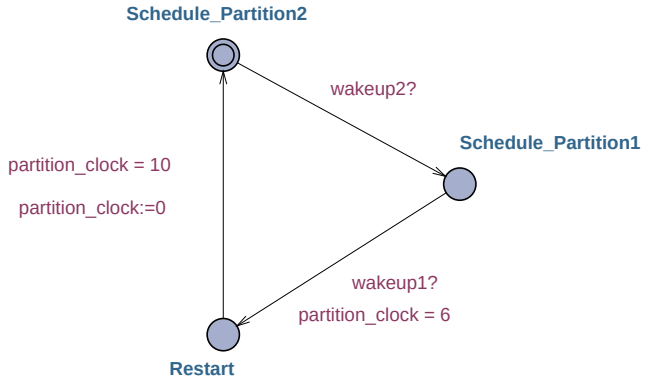

FIG. B.13: *Automaton modelling the ARINC 653 partition scheduler*

whereas the partition 1 schedules a set of uncritical tasks according to a round-robin scheduler. The partition scheduler enforces timing isolation between the two partitions which have different processor resource requirements.

B.4.3 From Cheddar programs to scheduling analysis

Scheduling simulation consists in predicting for each unit of time, the task to which the processor should be allocated. Checking if tasks meet their deadlines can then be performed by analysis of the computed scheduling. When an architecture only contains periodic tasks and for some real time schedulers such as Rate Monotonic, scheduling simulations leads to a proof whether tasks will meet their deadline. For such a proof, the designer has to run the scheduling simulation during a time interval called the hyperperiod (also called schedule length, base period, major cycle or scheduling period). If the architecture is composed of periodic tasks which have the same first release time, this hyperperiod can be computed by [LM80, CDKM02] :

$$[0, LCM(\forall i : P_i)] \tag{B.2}$$

where P_i is the period of the task i and LCM is the least common multiplier of all task periods of the system. If the system designer runs a scheduling simulation from the time 0 to the time $LCM(\forall i : P_i)$ and if no task deadline are missed during such a hyperperiod, then, no deadline will be missed during all the task scheduling.

From a Cheddar program which models a hierarchical scheduler, we generate Ada packages. These packages are part of the Cheddar library and allow the designer to run these scheduling simulations. In the following, we describe how such Ada packages are generated.

B.4.4 Increasing the usability of real time scheduling theory : Cheddar as an adaptable toolset

Real world problems often need very specific tools to be implemented. In the field of real time scheduling simulation tools, one very critical part is the scheduling simulation engine. This simulator can heavely change from one application to another and more, from one version to another version of the same system. A very important point for a toolset to be really useable is to allow such adaptation. With Cheddar, very specific schedulers can be implemented following a three steps process depicted by figure B.14 :

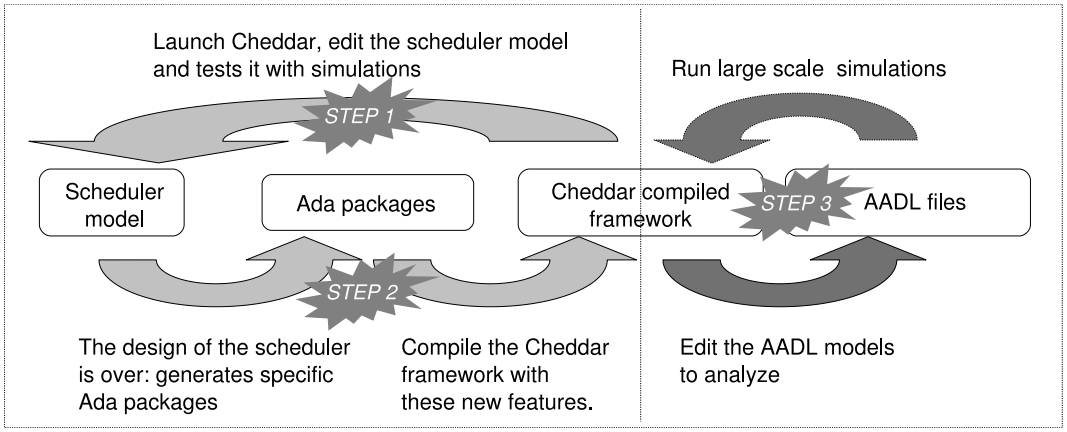

FIG. B.14: *A process to perform simulations from Cheddar scheduler models*

1. **Specification of the new scheduler** : the designer models a new high level scheduler with the Cheddar language. The resulting model can be directly interpreted using the Cheddar library. Cheddar toolset allows the designer to perform small scheduling simulations by running its high level specification using Cheddar language interpreter. This first running step gives the designer the opportunity to detect critical problem and to review its design very early in the life cycle.
2. **Scheduler integration** : the new scheduler can be integrated into Cheddar library as a specific Ada component that is automatically generated from its specification. The Cheddar library is then compiled in order to enrich it with this new scheduler.
3. **Scheduler using** : generated scheduler can be used as a builtin one : the designer makes use of his scheduler through this enriched Cheddar library in the same way he will make use of standard schedulers manually implemented into Cheddar (e.g. Rate Monotonic). Then more testings can take place because he can actually run large scale simulations.

This three steps process is implemented using three main tools : the compiler, the interpreter and the scheduler code generator. These tools use a central data management component calls the Cheddar Data Access Interface (CDAI). This architecture is shown in figure B.15 :

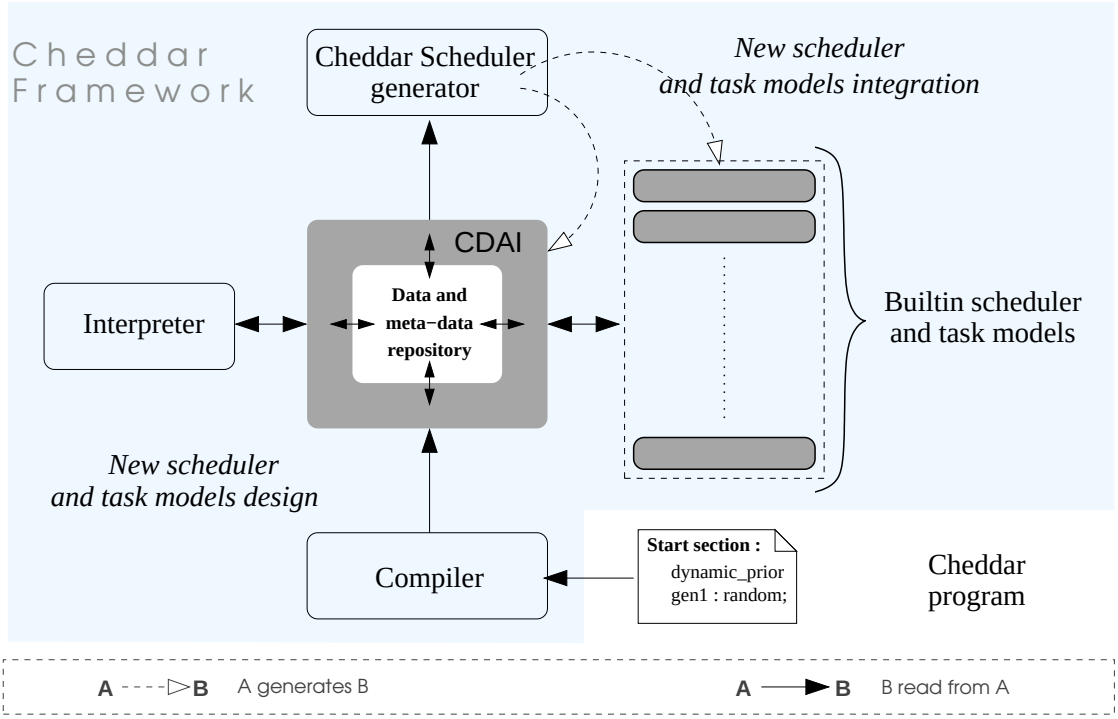

FIG. B.15: *The Cheddar library*

- The CDAI is used by every components of the Cheddar library in order to read or write data and meta-data (e.g. task priority, processor name, assignement, state definition, ...).
- A new scheduler is specified using the Cheddar programming language; Cheddar programs are read by the compiler which produces meta-data constituting an internal representation for them.
- The Cheddar interpreter is implemented in order to run Cheddar programs; an interpreting process run statements and expressions stored as meta-data and interacts with Cheddar library for data values reading and writing.
- The Cheddar scheduler generator is used to produce Ada packages from user-defined schedulers; for one user-defined scheduler, generated code consists in :
  - A package that implements the new scheduler.
  - A package that extends the CDAI for the new scheduler.

Data and meta-data stored into the repository are described by a set of models and meta-models. All these models are written with the ISO 10303 data modelling langage EXPRESS [ISO94, ISO04a]. The EXPRESS meta-models are specified with self contained code generators and are implemented using the software engineering tool Platypus [PR98].

As shown by figure B.16, code generation is used at two levels of abstraction :

1. The first level is the Cheddar library level. This level is related to a particular Cheddar library version for which all handled object types (processors, tasks, buffers, ...) are fixed and described by the Cheddar data model. The Cheddar data model is an EXPRESS model, it is parsed by

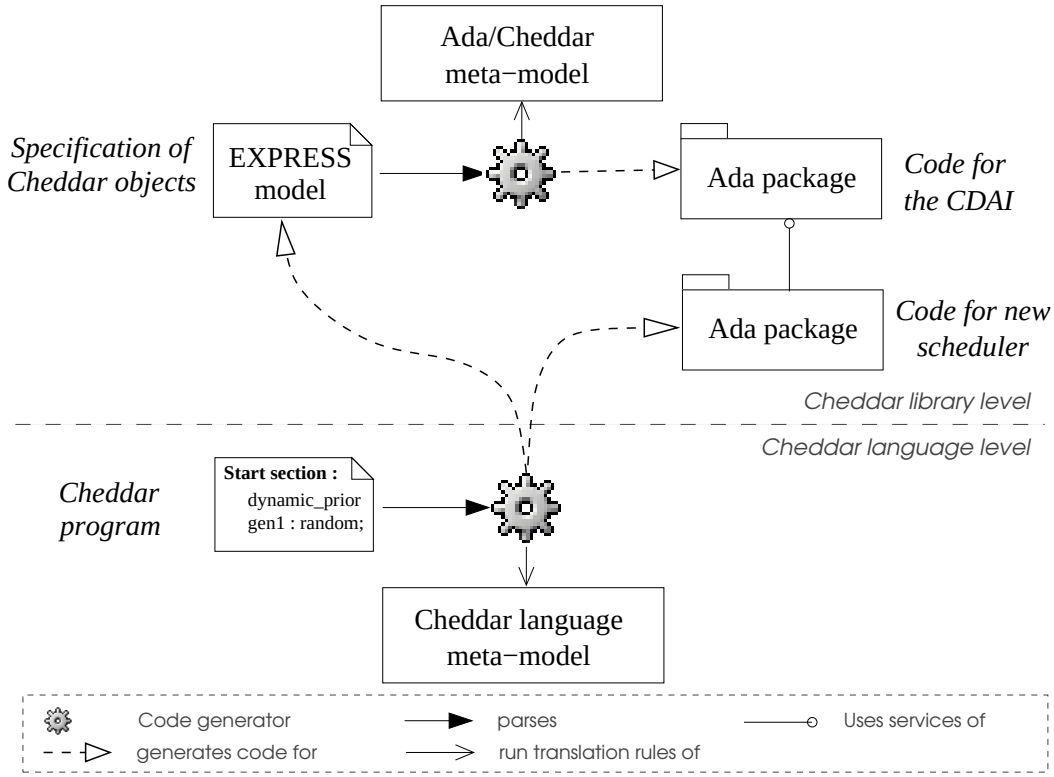

FIG. B.16: *Generation of Cheddar code from models and meta-models*

an Ada code generator that produces the CDAI set of packages. Translation rules applied by the generator are specified by an Ada for Cheddar meta-model [PS06].

2. The second level is the Cheddar language level. It corresponds to Cheddar specializations driven by the specification of new schedulers and task models. These new parts are specified using the Cheddar programming language. The dedicated code generator is able to parse a Cheddar program and first, to produce a new scheduler implementation and second, to enrich the Cheddar data model. Then, the CDAI packages are regenerated and a new Cheddar library version can be compiled [SP07b].

B.5 Conclusion and future work

This article presents three possible directions that are being explored by the Cheddar project in order to increase the usability of the real time scheduling theory. We have presented a set of open-source tools which help the designer to automatically apply this theory to concrete cases. This toolset allows several levels of use and is able to perform analysis of models written with standardized design languages. At this time, the Cheddar project has been focused on AADL, an architecture language which aims at modelling and analysing system and software real time architectures. We also presented

a domain specific language that can be used to investigate performances of architectures for which existing real time scheduling theory does not propose proper analytical method. The Cheddar toolset can be downloaded from http ://beru.univ-brest.fr/~singhoff/cheddar.

At the time this article is written, it is difficult to state if Cheddar has actually helped people to apply real time scheduling theory on practical cases. However, the project has already led to some significant contributions and returns of experience :

- The Cheddar project raised an interesting opportunity to make a survey about the feasibility tests that are proposed by the real time scheduling theory. Some missing feasibility tests were thus identified.
- New feasibility tests based on the queueing system theory have been proposed. Two queue types called P/P/1 and M/P/1 have been identified from which feasibility tests can be built. These feasibility tests enable memory footprint analysis of real time architectures and can be also applied on distributed systems.
- By experimenting automatic scheduling verification from AADL models, suitability of AADL v1 for such a purpose has been investigated. Some extensions to the default AADL properties set were proposed and will be integrated into the next release of the AADL standard. Some usual inter-task communication or synchronization paradigms have also been modelled in AADL in order to enable interoperability between modelling tools such as Stood and analysis tools such as Cheddar. In the same way, Hugues et al. have proposed a process which allows to performa analysis and generate code from AADL specifications which are compliant to Ravenscar [HZPK08]. In this project, the Ravenscar generated code is able to run on PolyORB-HI and the analysis were ran with CPN-AMI [HHK+06] and Cheddar. These two experiments can be seen as a proof that AADL can be actually used as an efficient pivot language to build model driven engineering tool chains.

  Cheddar has also been used for the same purpose with other modelling languages such as Marte/UML [OMG07] or PPOOA [Fer07]. Marte is a new UML profile designed by the OMG for modelling real time architectures. Thales RT has developed a set of Eclipse plugins which allows to perform scheduling analysis with Cheddar on UML/MARTE models [Mae07]. Similarly, PPOOA is a modelling framework based on UML with a customize profile for pipe-line architecture which has been proposed by the University of Madrid.
- Some other experiments have proved that the specific design language proposed to model new real time schedulers is well suited for this purpose. We have shown how to use it in order to model ARINC 653 schedulers [SP07a] and sporadic tasks [SP07b]. Several other real time schedulers have been verified with Cheddar. For example, Ahn et al. have designed and verified with Cheddar a real time multimedia scheduler [AKLL06]. In the same way, Muhuri and al. [MS08] have verified a fuzzy real time scheduler. Airbus industries have modeled and verified message scheduling for their flight simulator [CCG07]. Furthermore, the Cheddar project has contributed in many research projects related to the model driven engineering [RTC07, RBP07] or performance analysis [NCSA07].
- Finally, several universities and engineering schools have built real time scheduling courses with

the help of Cheddar : Telecom Paris-Tech, University of Rhodes Island, University of Monash, Universitat Politecnica de Catalunya, University of the West Indies, ... Including real time scheduling courses in engineering curriculum may help people to make use of such analysis method.

These first encouraging experiments may lead to numerous future works. Firstly, Ellidiss Technologies will distribute Cheddar with its modelling tool Stood. We expect to experiment the use of the real time scheduling tools in more industrial contexts. For such a purpose, more operational modelling design patterns will have to be studied and implemented consistently in Stood for modelling concerns and in Cheddar for analysis purposes [DS08].

Secondly, the Cheddar language that has been defined to model schedulers was experimented in several projects. We now know that this language is well suited for this purpose. The language is based on an Ada like language, which allows static analysis (e.g. SPARK [Bar03]) and on a timed automaton language which allows dynamic analysis (e.g. model-checking). We plan to investigate how Cheddar scheduler model analysis can help designers to compare their models.

Finally, the complexity of real time systems has been growing quickly for these 15 last years. In the past, the only kind of resource requiring deep and accurate analysis was the processor. But now, many real time systems are distributed over several processors and several resources have to be managed all together : processors, communication networks and memory units. The work we have done on memory footprint analysis with queueing system models must be extended to cope with distributed system analysis. Some new feasibility tests from P/P/1 or M/P/1 queueuing system must also be defined.