Développement de systèmes à l'aide d'AADL - Ocarina/Cheddar

Jérôme Hugues
Institut Télécom – Télécom-Paris-Tech – LTCI
46 rue Barrault
75634 Paris, France
hugues@telecom-paristech.fr

Frank Singhoff
LISyC – Université de Bretagne Occidentale – UEB
20, avenue le Gorgeu
29238 Brest Cedex 3, France
singhoff@univ-brest.fr

Abstract

La construction de systèmes embarqués critiques temps réel suppose de définir conjointement les aspects fonctionnels (algorithmes de calcul, logique de décision) et les aspects non fonctionnels (paramètres de qualité de service, échéances). Afin de valider que le système embarqué critique temps réel est correct, il est souvent nécessaire de disposer d'une vue complète de l'architecture logicielle, matérielle et de leur lien.

Le langage AADL (Architecture Analysis and Design Language) est un standard promu par la SAE afin d'aider à l'analyse et la conception de ces systèmes.

Cette article vise à fournir un panorama des techniques accessibles au concepteur de systèmes, basées sur AADL. Nous nous limiterons ici aux analyses d'ordonnancement, de pire temps d'exécution et la génération de code.

1 Introduction

Dans cet article, nous traitons de l'ingénierie des systèmes embarqués temps réel critiques [33, 6]. 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, ...). Lorsqu'un système embarqué est contraint par le temps, le système 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 [33]. 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 [16]. La partie descendante du V comprend les phases d'expression de besoins, 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. 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 [22].

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

La partie ascendante du V peut être constituée des phases de tests unitaires des composants, puis des tests d'intégration du système et d'une phase de recette.

La phase de tests unitaires a 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 récep-

tionne 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.

Récemment, 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, ces vérifications sont réalisées tardivement dans la vie du système : en phases de test voire de maintenance corrective. Or, le coût de ces phases est généralement très élevé [3] et dans un certain nombre de cas, les anomalies de fonctionnement peuvent être issues d'erreurs de conception ou d'expression des besoins.

Ainsi, aujourd'hui, les acteurs du domaine souhaitent appliquer ces 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.

Le langage AADL, associé à une démarche de MDE[1], propose une approche qui répond à cet objectif. Par la spécification de l'architecture en amont du cycle de développement, le langage AADL vise à fournir des moyens de vérification précosse. Par ailleurs, en offrant un langage standardisé, AADL autorise la collaboration de divers outils de vérification, de génération de code ou d'édition de modèles. Ainsi, après avoir vérifié une architecture AADL par différents outils et méthodes, un ingénieur peut envisager la production automatique d'une partie de son système.

Dans la suite de cet article, nous donnons tout d'abord une description succincte du langage d'architecture AADL. Cette description est illustrée par une étude de cas. Puis, dans la partie 3, nous montrons comment ce langage peut être employé pour automatiser différentes étapes du cycle de vie telles que la vérification de propriétés temporelles (ordonnancement et calcul des pires temps d'exécution des flux de contrôle) ou la mise en œuvre (génération de code). La partie 4 explique de façon concrête comment les outils AADL sont employés conjointement pour vérifier et implanter l'étude de cas. Enfin, nous concluons dans la partie 5.

2 Le langage d'architecture AADL

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 [37]. Différents langage d'architecture adaptés aux systèmes temps réel on été proposés : MARTE-UML [14], EAST ADL [5] ou AADL [27].

AADL est une norme internationale publiée par la SAE[2] sous le standard AS-5506 [27]. AADL offre la possibilité de décrire l'architecture complète, matérielle et logicielle d'un système embarqué. 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 [13, 2, 8].

De nombreux outils AADL ont été proposés ces dernières années : ADELE [10], Stood [7], OSATE [28], TOPCASED [11], ADes [35], Versa/Furness [32], ADAPT [26], Ocarina [38, 18] ou Cheddar [31]. Ces outils permettent :

- D'éditer des modèles AADL (outils Stood, ADELE, OSATE, TOPCASED, ...).
- De générer le logiciel du système à partir d'un modèle AADL (Stood, Ocarina, ...).
- Ou de conduire diverses analyses (OSATE, Versa/-Furness, ADAPT, Cheddar, ...).

Un langage d'architecture introduit généralement 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 [22]. 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.

AADL propose ces différents concepts : un modèle AADL est un ensemble hiérarchisé de composants, chacun disposant de son propre type ou interface. Le standard propose différents types de composants matériels et logiciels. Les composants $data$, $thread$, $process$ sont des composants logiciels. Les composants $processor$, $device$ et bus 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 tels que C ou Ada, ou peut être mise en relation avec des outils de modélisation utilisés dans l'industrie tels que SCADE, Esterel ou Simulink.

Voici la liste des composants logiciels disponibles :

- 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 sous-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.
- 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 dé-

[1] Model Driven Engineering.
[2] Society for Automotive Enginners.

ployés 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 de signaux ou évènements entre composants : par exemple pour indiquer l'occurrence d'anomalies dans le fonctionnement du système. Enfin les $event\ data\ ports$ sont dédiés à l'échange asynchrone de messages entre composants. Dans les deux cas, 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[3].

AADL dispose à la fois d'une représentation graphique, et d'une représentation textuelle. Le graphique à l'avantage de permettre une représentation à haut niveau du système. Le textuel est plus précis, et permet potentiellement de représenter plus d'informations.

Nous utiliserons l'exemple de la figure 1 tout au long de cet article. Il représente un modèle simplifié d'un radar. Il repose sur un processeur `leon` sur lequel s'exécutent plusieurs threads. Nous modélisons des communications avec le monde extérieur : une antenne émet une onde qui est ensuite réfléchie sur un obstacle, puis reçue à nouveau par l'antenne. On déduit la position de l'obstacle en analysant le temps mis pour recevoir l'onde réfléchie, et en analysant la position du moteur dirigeant l'antenne. Ces calculs sont pris en charge par plusieurs threads qui doivent se coordonner. Une fois le calcul réalisé, un affichage s'opère sur un troisième périphérique : un écran.

L'annexe de cet article présente un extrait de ce modèle AADL[4] en utilisant la représentation textuelle.

Ce modèle décrit un système radar constitué de plusieurs composants :

1. Un processeur (composant cpu), un bus (composant vme), une unité de mémoire (composant ram) et plusieurs $devices$ (composants $aerial$, $rotor$, et $monitor$).
2. Un processus (composant $main$) hébergeant plusieurs $threads$ (composants $receive$, $analyse$, $display$, $transmit$ et $control\_angle$).

Le composant $radar.impl$ modélise l'architecture globale et définit, entre autre, la projection des composants logiciels sur les composants matériels.

Ce modèle contient aussi des exemples de propriétés :

- Les propriétés $Deadline$, $Period$, $Dispatch\_Protocol$ ou $Compute\_Execution\_Time$ sont requises pour la vérification de l'ordonnancement des threads. Ces propriétés mémorisent respectivement : l'échéance d'un thread, sa période, comment un thread est activé (périodiquement, sporadiquement, ...) et son pire temps d'exécution.
- Les propriétés $Source\_Language$ et $Source\_Name$ indiquent comment un composant est implanté. Ainsi, dans le cas du composant $Controller\_Spg$ (qui est un sous programme), ces propriétés indiquent que le logiciel est écrit en Ada95 et que la procédure Ada correspondante à ce composant est la procédure $radar.controller$.
- La propriété $Scheduling\_Protocol$ du composant $leon2$ 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 aux propriétés $Actual\_Processor\_Binding$ et $Actual\_Memory\_Binding$ du composant $radar.impl$. Ainsi chaque processus est affecté à un processeur grâce à la propriété $Actual\_Processor\_Binding$.

3 Processus de développement basé sur AADL

À partir d'un modèle AADL comme celui de l'annexe, il est possible d'effectuer différentes analyses, voire de partiellement générer l'application. Nous détaillons dans les parties suivantes ces opérations.

3.1 Vérification de l'ordonnancement

AADL définit la notion de thread, d'ordonnanceur (attaché à un processeur) et un jeu de priorités pour décrire ces composants. Ainsi, il est possible d'effectuer une analyse de l'ordonnancement d'un modèle AADL avec de nombreux outils tels que OSATE [28], Versa/Furness [32] ou Cheddar [31].

[3] First In, First Out

[4] Le modèle complet est disponible dans l'archive d'Ocarina, disponible sur `http://aadl.telecom-paristech.fr`

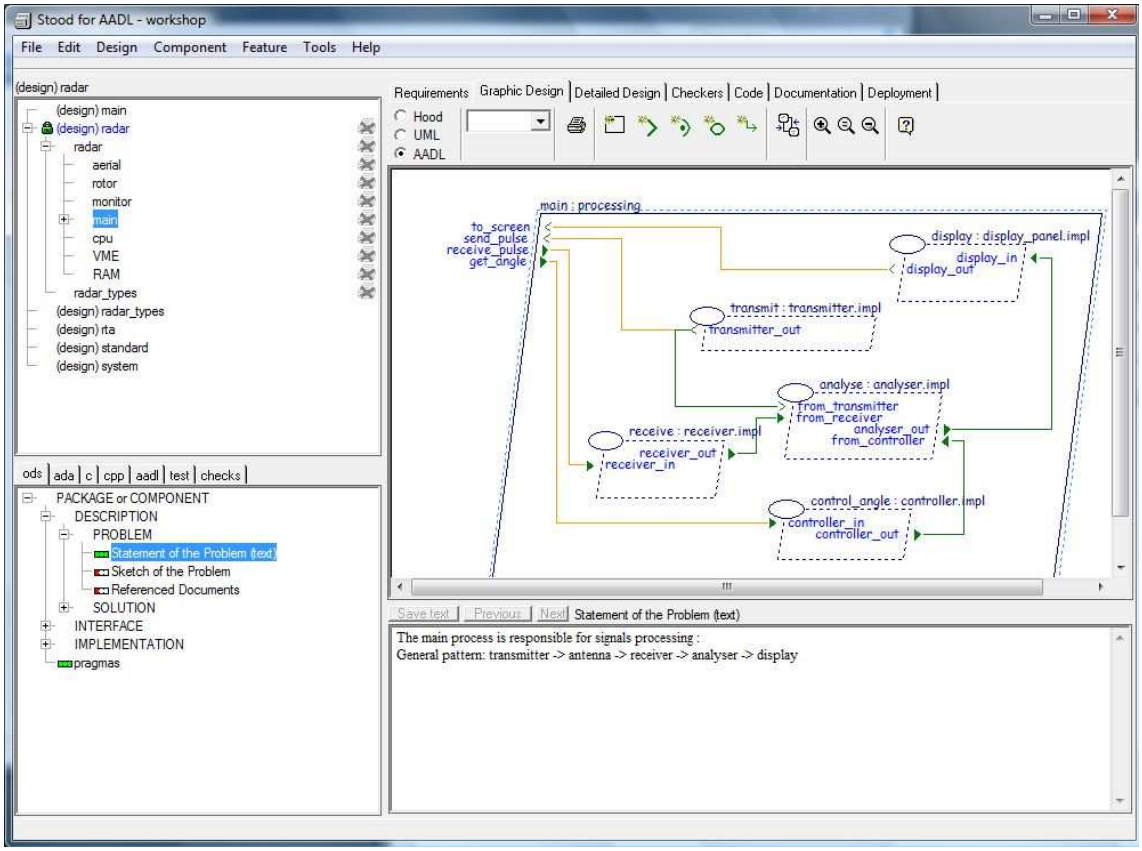

FIG. 1 – Modélisation à l'aide de Stood

Cheddar [5] 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.

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 [9], TOPCASED [11], IBM Rational Software Architect [25] ou PPOOA-Visio [12].

Le canevas logiciel est constitué d'un ensemble de paquetages Ada implantant les principales méthodes de vérification proposées par la théorie de l'ordonnancement temps réel. Ce canevas propose une bibliothèque de méthodes analytiques (ou tests de faisabilité) ainsi que les algorithmes d'ordonnancement temps réel les plus classiques. Cette bibliothèque permet de vérifier l'ordonnançabilité d'un système temps réel à partir de sa spécification architecturale.

Par ailleurs, Cheddar dispose d'un langage spécifique ainsi que d'outils associés (interpréteur, parseur, ...) permettant 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.

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.

3.1.1 Usage de patrons de conception

Une première façon d'utiliser un outil comme Cheddar consiste à supposer que le concepteur utilise un patron de conception pour décrire l'architecture de son système. Dans le contexte de cet article, un patron de conception propose une solution architecturale à une famille de problèmes de concurrence. Chaque patron de conception est associé à un ensemble de tests de faisabilité ou d'algorithmes d'ordonnancement pouvant être employés pour vérifier l'ordonnançabilité de l'architecture. On suppose que ces patrons de conception garantissent le respect des hypothèses associées aux tests de faisabilité.

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 de charger un modèle d'architecture et de réaliser l'analyse en pressant l'un des boutons de l'éditeur de Cheddar. En in-

[5] Logiciel disponible sur `http://beru.univ-brest.fr/~singhoff/cheddar`.

diquant le patron de conception utilisé, Cheddar choisit automatiquement les critères de performance à calculer et les tests de faisabilité à appliquer tout en contrôlant si les hypothèses des tests sont vérifiées.

À titre d'exemple, le concepteur peut employer le patron de conception $Ravenscar$. Ravenscar est défini dans le standard international Ada 2005 [20, 34]. Il s'agit d'un sous-ensemble du langage Ada 2005, et en particulier de ses fonctionnalités ayant trait à la concurrence. Ce sous-ensemble d'Ada assure que tout programme compilé est compatible avec certains tests de faisabilité.

Le profil Ravenscar impose un ensemble de restrictions qui sont vérifiées lors de la compilation et qui 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 Ada 2005 les restrictions qui sont pertinentes du point de vue de l'analyse de l'ordonnançabilité :

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 est 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 est la politique FIFO. Elle consiste à élire la tâche prête en tête de la file d'attente.
2. Utilisation du protocole ICPP[6] afin de borner les temps d'attente sur les ressources partagées. Le protocole ICPP est une variante du protocole PCP [29]. Cette restriction est imposée par le $pragma\ Locking\_Policy(Ceiling\_Locking)$.
3. Départ simultané 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 $No\_Select\_Statement$). Ces restrictions permettent de limiter la variabilité du pire temps d'exécution de chaque tâche.

[6] Immediate Ceiling Priority Protocol.

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 décrits dans [21, 1, 36, 29]. Par ailleurs, l'utilisation du patron Ravenscar n'est pas limitée aux applications écrites en Ada et il est envisageable d'employer ce patron pour une application écrite en langage C avec l'interface POSIX 1003.1b [15]. De même, une adaptation du profil Ravenscar existe aussi pour Java [23]. En fait, de nombreux environnements d'exécution sont potentiellement compatibles avec ces contraintes.

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. Toutefois, toutes les propriétés nécessaires à l'analyse du patron Ravenscar n'étaient pas incluses dans le standard. 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 adapté au patron Ravenscar.

Cet ensemble de propriétés spécifiques à Cheddar [30] contient principalement des propriétés associées aux composants $threads$, $processors$ ou $data$ (exemple : valeur du quantum d'un ordonnanceur, gigue sur le réveil d'un thread, ...). Certaines de ces propriétés ont été ajoutées à la version 2 du standard AADL. Dans le modèle en annexe, les propriétés préfixés par $Cheddar\_Properties$ sont des propriétés spécifiques à Cheddar.

Le modèle AADL en annexe est conforme au patron de conception Ravenscar, il est donc possible d'en effectuer une analyse de l'ordonnancement de façon automatique avec Cheddar. Les composants de ce modèle les plus significatifs vis-à-vis de ce type d'analyse sont les composants $thread$ et $processor$:

```
thread implementation receiver.impl
 properties
  Cheddar_Properties::Fixed_Priority => 63;
  Dispatch_Protocol => Periodic;
  Compute_Execution_Time => 1 ms .. 2 ms;
  Deadline => 15 ms;
  Period => 15 ms;
end receiver.impl;

thread implementation analyser ...
thread implementation display_panel ...
thread implementation transmitter ...
thread implementation controller ...

processor leon2
properties
 Scheduling_Protocol =>
  Posix_1003_Highest_Priority_First_Protocol;
end leon2;
```

Listing 1 – Principaux composants/propriétés pour l'ana-

lyse de l'ordonnancement

3.1.2 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 de faisabilité qu'il doit calculer. Toutefois, le concepteur doit s'assurer que le critère de performance est pertinent et que la méthode de calcul 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. 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. 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 s'avérer utile.

Les tests que nous avons 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 un modèle de tâches bien adapté pour les systèmes critiques.
2. L'architecture emploie un des 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 performance (optimalité). Ils constituent de bons candidats lorsque des ordonnancements hors-ligne doivent être calculés.
3. Les architectures sont préalablement partitionnées. Bien que des méthodes de partitionnement soient 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.
4. Les ressources sont partagées grâce aux protocoles PIP, PCP ou SRP. Ces protocoles sont les plus couramment utilisés ou référencés dans la littérature.

3.1.3 Vérification par simulation : usage du langage spécifique

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. Les tests de faisabilité implantés dans le canevas logiciel de Cheddar ne peuvent alors pas être appliqués. Dans ce cas, l'architecture est vérifiée par 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 tout d'abord 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.

3.2 Du modèle AADL au code

Comme nous l'avons vu à la section précédent, il est possible grâce à la sémantique clairement définie par le standard AADL de réaliser une analyse du système. Certaines de ces analyses tirent avantage du modèle comportemental décrivant les interactions entre composants.

Une fois le système validé, on peut donc passer à la phase d'implentation. Cependant, il serait dommage de ne pas tirer partie de la sémantique du modèle pour déduire automatiquement le code représentant fidèlement l'architecture. En effet, il est intéressant de procéder à une génération de code massive à partir d'un modèle : nous réduisons potentiellement de nombreux bogues issus de la traduction du modèle, et nous permettons un test réel du système dans une approche par prototypage et raffinement d'un modèle.

3.2.1 Annexes de description de données, inclusion de code source

AADL seul fournit de nombreux éléments de descriptions. Il définit par ailleurs différents services attachés à un exécutif AADL. Néanmoins, il ne fournit pas un niveau d'information suffisant permettant d'écrire directement le code métier d'un système. Pour ce faire, il faut adjoindre

à AADL une projection vers le langage de programmation retenu, et définir pour chaque construction AADL son équivalent dans le langage de programmation cible.

La runtime AADL fournit un support d'exécution pour des systèmes basés sur des composants et sur des communications par ports. Les fils d'exécution sont représentés par les threads AADL, regroupés par process. Afin de garantir une portabilité des exécutifs, et permettre différentes stratégies d'implantation, le comité AADL a décidé de définir deux annexes au standard décrivant comment précisément modéliser des types de données, et comment écrire la portion de code devant être exécutée par les threads. Le déploiement et la configuration des éléments de l'architecture (thread, process, bus, ...) ne sont pas standardisés afin de ne pas imposer un choix particulier, comme c'est le cas pour CORBA.

Les annexes de description de données et d'inclusion de code source sont deux documents séparés, décrivant :

1. Comment construire des composants $data$ AADL qui puissent être transposés en types du langage cible (tels que les $structs$ en C).
2. Comment interagir avec la runtime AADL pour échanger des messages au travers des files.
3. Et comment construire les sous programmes qui devront être exécutés.

Ces projections définissent des règles de nommages et de traduction strictes, afin de permettre une portabilité des programmes ainsi construites d'un exécutif AADL à un autre. Voici un exemple de modèle AADL, extrait de l'exemple du radar :

```
data target_distance
properties
   data_model::Data_Representation => integer;
end target_distance;

subprogram receiver_spg
features
  receiver_out : out parameter target_distance;
  receiver_in : in parameter target_distance;
properties
  Source_Language => Ada95;
  Source_Name => "radar.receiver";
end receiver_spg;
```

Listing 2 – Donnée et sous-programme AADL

$data\_model$ est un ensemble de propriétés définit dans l'annexe de modélisation de données, elle fournit les moyens de décrire des types, des plus simples aux plus évolués. Ici, $target\_distance$ est un entier.

AADL fournit les moyens de lier implantation et modèle. $receiver\_spg$ est un sous-programme AADL pour lequel on définit l'implantation associée. Cette implantation un sous-programme Ada défini dans le package $radar$.

Partant de ces descriptions, il est possible de déduire le code correspondant. Les règles pour C et Ada ont été définies. Les règles pour Java temps réel sont en cours d'élaboration et d'implantation dans Ocarina. Voici le code Ada correspondant à l'exemple précédent :

```
type target_distance is new standard.integer;

-- [..]

procedure receiver_spg
  (receiver_out : out target_distance;
   receiver_in : target_distance);
```

Listing 3 – Code Ada déduit

3.2.2 Ocarina, un générateur de code pour AADL

Ocarina [19] est un générateur de code pour AADL. Il s'agit d'un outil indépendant. Il fournit un ensemble de bibliothèques pour la manipulation de modèles AADL : analyse syntaxique, sémantique, exploration de modèle. Partant de ces constructions, il est possible de construire des outils évolués d'analyse tels que Cheddar lui même, ou des générateurs de code. Ainsi, notre équipe a développé deux générateurs de code, pour C et Ada. L'approche retenue par Ocarina est de proposer un compilateur AADL complet, avec tous les éléments canoniques : construction d'AST, visiteurs, expansion d'arbre puis parcours en vue de construire un second arbre dans le langage cible.

Cette approche de construction permet de concevoir un générateur de code optimisé, capable de gérer de gros modèles en un temps réduit, mais aussi de mieux contrôler la structure du code construit si on le compare à d'autres approches de programmation par template. Cette dernière fonctionnalité est importante pour la construction de systèmes temps réel critiques : elle permet de garantir a priori la qualité du code (structure et représentation) permettant ensuite une meilleur analyse.

Ocarina est couplé à un exécutif fournissant les constructions de base d'une runtime AADL : PolyORB-HI, disponible pour C et Ada. Contrairement à un framework classique, ces runtimes ne peuvent fonctionner sans une portion de code générée à partir du modèle AADL. Ce code définit un certain nombres de type centraux : tables de routage des messages, buffers nécessaires pour stocker les messages, instances des tâches, ...Une description complète de l'approche est disponible dans la thèse de B. Zalila [40].

Le code généré est conforme aux règles d'écriture de code les plus strictes, compatibles avec les contraintes de déterminisme des systèmes temps réel, mais aussi exempts des patrons de code pouvant compromettre une analyse statique de code (allocation dynamique de mémoire, d'aiguillage dynamique tels que l'orienté objet ou les pointeurs sur fonction, ...). Ces aspects font que le code généré est compact, et limité aux seuls aspects utiles.

Ocarina et la runtime associée permettent de garantir que l'empreinte mémoire du système construit est faible. Nous avons montré [24] que l'écart sur l'empreinte mémoire d'un exemple significatif écrit manuellement et généré à l'aide d'Ocarina était de l'ordre de 5%. Ce faible écart est peu significatif pour les plates-formes embarquées actuelles. Par ailleurs, les différentes analyses directement accessibles depuis un modèle AADL rendent

l'approche utilisable dans un contexte industriel.

Ocarina et les runtimes associées ont ainsi pu être mises en œuvre dans les projets IST-ASSERT[7], en partenariat avec l'ESA, et Flex-eWare[8] avec Thales.

3.3 Du modèle à l'évaluation du WCET

Comme nous l'avons montré, il est possible de combiner un modeleur tel que Stood pour construire son système, puis ensuite d'utiliser Cheddar et Ocarina pour valider l'ordonnancement de son système avant de procéder à sa génération. AADL ouvre de nombreuses autres perspectives de traitements automatisables.

Ainsi, un problème récurrent dans la construction de systèmes temps réel est l'obtention de bornes réalistes sur le pire temps d'exécution du système (ou WCET[9]). L'obtention d'un majorant du WCET reste un problème ouvert [39].

Le calcul de WCET nécessite d'évaluer à la fois le pire chemin d'exécution sur les parties séquentielles du programme, mais aussi d'intégrer certaines informations liées à l'exécutif (temps de prise d'un verrou, coût d'une commutation de contexte, ...). Ainsi, une analyse fine de WCET peut vite devenir complexe. Ceci requiert de disposer d'une description d'architecture fine telle que celle fournie par AADL, mais aussi d'un générateur de code fournissant un code de taille réduite, sans construction impossible à analyser.

Nous avons intégré l'outil Bound-T à Ocarina [17]. Nous tirons parti de la connaissance fine du code généré et de l'architecture pour indiquer à Bound-T les sous-programmes à analyser, correspondant aux sous-programmes de tâches et comment borner certaines boucles ou le temps d'exécution des fonctions du système d'exploitation sous-jacent.

Bound-T dispose de son propre langage de description pour représenter le programme à analyser. Nous avons défini un outil de transformation de modèles de AADL vers ce formalisme, tirant partie des informations architecturales. Nous complétons cette description dépendante de l'architecture par une seconde qui est fixe, dépendante de la runtime. Ainsi, Bound-T dispose de toutes les informations utiles pour fournir une borne fine du WCET.

Partant du résultat de l'analyse, nous pouvons injecter les nouvelles valeurs de WCET dans le modèle AADL, et procéder à une analyse plus fine de l'ordonnnancement du système. Ce rafinage du modèle initial est important pour valider a posteriori que le système est correct.

4 Utilisation des outils avec l'exemple radar

Dans la partie 3, nous avons décrit différents traitements automatisables à partir d'un modèle AADL. Dans cette partie, nous revenons sur l'exemple du radar présenté en annexe et montrons comment vérifier l'ordonnançabilité du modèle et générer le système. Nous supposons dans la suite que le modèle AADL de l'annexe est stocké dans les fichiers *radar.aadl* et *radar_types.aadl*.

4.1 Analyse de l'ordonnancement

Pour effectuer une analyse, Cheddar requiert un modèle AADL complet ainsi que le fichier contenant les propriétés spécifiques à Cheddar. On suppose que ce fichier de propriétés spécifiques est présent dans l'espace de travail de Cheddar. Ce fichier est généralement nommé `Cheddar_Properties.aadl`. L'analyse du modèle AADL radar est déclenchée par la commande suivante : `cheddar -a radar.aadl radar_types.aadl`. Cette commande valide le modèle AADL grâce à Ocarina, puis, lance l'éditeur de Cheddar. Le modèle radar étant conforme à Ravenscar, l'analyse à proprement dite est alors effectuée par simple pression d'un des boutons de l'éditeur de Cheddar.

Les résultats d'analyse sont alors affichés dans une fenêtre comportant deux parties. La figure 2 présente une sortie d'écran de cette analyse. La partie haute de la fenêtre contient un ensemble de chronogrammes décrivant l'ordonnancement des $threads$ attendu pendant l'exécution du système. D'autres évènements peuvent être présentés sur ces 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 associés au patron de conception. La sortie d'écran pour un modèle Ravenscar affiche principalement :

- Les temps de réponse des $threads$.
- Les délais maximaux d'attente pour l'accès aux composants $data$.

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.

4.2 Code généré

AADL permet de définir à la fois l'architecture matérielle et logicielle du système. Ocarina permet la génération de ce système, mais nécessite des informations supplémentaires pour sélectionner le langage cible, la plateforme, etc. Ainsi, Ocarina utilise certaines propriétés spécifiques afin de clarifier la plate-forme cible : compilateur, options, ...

```
processor leon2
features
  vme : requires bus access vme;
properties
  Deployment::Execution_Platform => LEON_ORK;
  Clock_Period => 200 ns;
  Scheduling_Protocol =>
     Posix_1003_Highest_Priority_First_Protocol;
end leon2;
```

Listing 4 – Information de déploiement

[7] `http://www.assert-project.net/`
[8] `http://www.flex-eware.org/`
[9] Worst Case Execution Time.

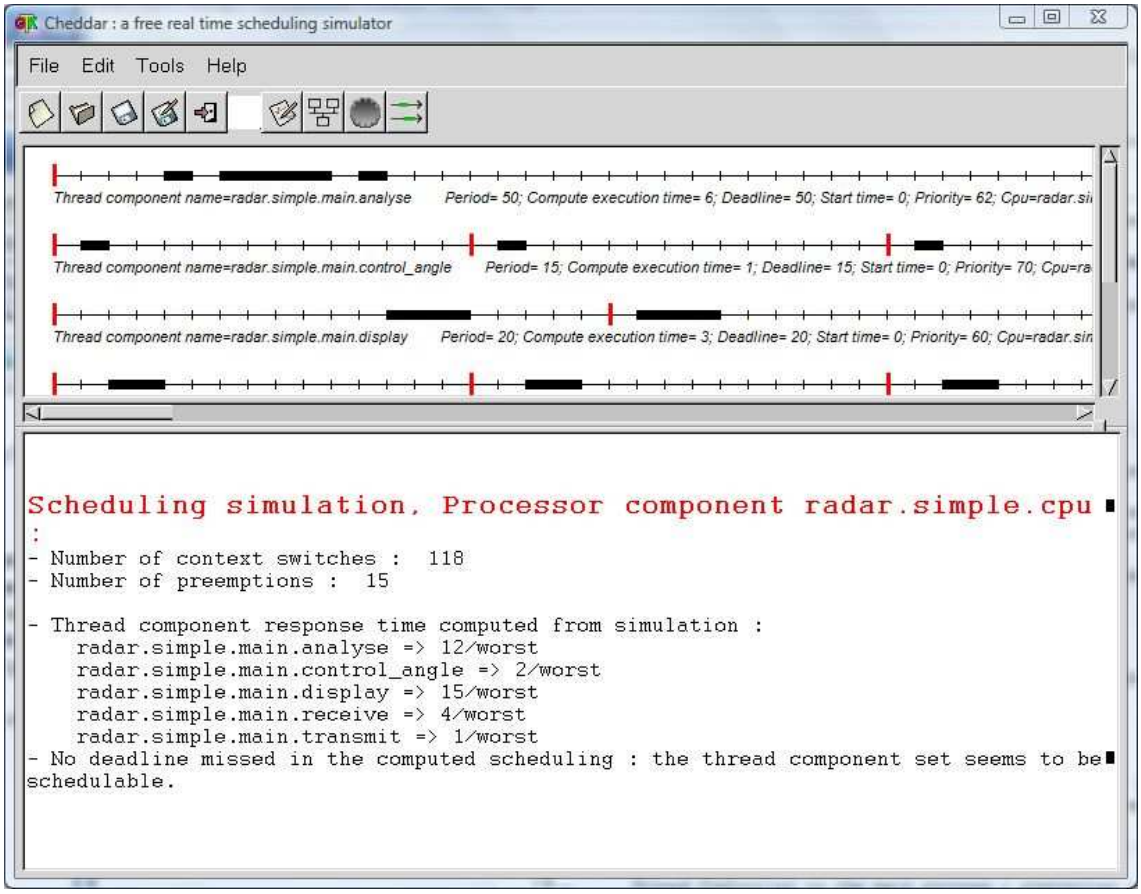

FIG. 2 – Analyse de l'ordonnancement

Ainsi, $Deployment :: Execution\_Platform$ précise le type de noyau à utiliser (noyau ORK+ [4] pour processeur LEON), $Scheduling\_Protocol$ la configuration de l'ordonnanceur. Les autres éléments sont déduits de l'architecture : types de données, tâches, connexions. À partir de ces informations, Ocarina est en mesure de générer le code du système, ainsi que les makefiles pour la compilation.

La génération et la compilation de code sont directement pris en charge par Ocarina à l'aide de la commande suivante : `ocarina -f -b -g polyorb_hi_ada radar.aadl radar_types.aadl`

L'exemple ci-dessus est prévu pour fonctionner avec le noyau ORK+. Il suffit alors d'exécuter le système, par exemple avec le simulateur `tsim-leon`[10].

```
tsim> go
resuming at 0x40000000
[ 0.00] Transmitter
[ 0.00] Controller, motor is at ..
[ 0.04] Receiver, target is at distance 1
[ 0.06] Analyser: target is at ..
[ 0.10] Display_Panel: target is at ( 1, 1)
[ 0.78] Transmitter
```

[10] Diffusé par Gaisler, `http://www.gaisler.com/`

L'affichage correspond au code utilisateur, fourni avec l'exemple. Ce code simpliste montre une exécution correcte du modèle, conforme à l'ordonnancement calculé par Cheddar.

Le modèle étant conforme au profil Ravenscar, on souhaite aussi garantir que le code final sera conforme aux mêmes restrictions. Le langage Ada permet de définir des restrictions au langage, à la fois syntaxique et sémantique. Ainsi, le profil Ravenscar est définit sous la forme d'un ensemble de directives de compilation qui permettent de valider que le code est conforme à ces restrictions. Ainsi, l'analyse du code réalisée lors de la compilation par le compilateur Ada GNAT montre qu'il est conforme à Ravenscar, préservant ainsi les hypothèses de modélisation.

Ce bref aperçu de Cheddar et d'Ocarina ne couvre qu'un sous-ensemble très restreint des possibilités de AADL et des outils associés. De nombreux autres exemples sont disponibles sur les sites respectifs.

5 Conclusion

Dans cet article, nous avons procédé à un tour d'horizon d'AADL, et nous avons présenté l'utilisation des outils Cheddar et Ocarina. Nous avons expliqué comment

construire un modèle AADL, l'analyser et générer le système correspondant. Ainsi, nous montrons comment prototyper simplement un système embarqué temps réel critique, en le simulant ou l'exécutant sur une cible. L'intégration de ces deux outils AADL permet de se concentrer sur les fonctionnalités du système à construire, et non sur la mise en œuvre d'une analyse particulière.

Ceci démontre que l'approche basée sur les modèles AADL peut simplifier les nombreuses phases d'analyse. D'autres outils existent pour valider l'architecture. Ces outils complétent la démarche par des analyses du comportement, des contraintes structurelles, du dimensionnement, . . .AADL est riche de nombreux outils qui ont tous un rôle clé dans la construction de l'application. Les combiner à bon escient permet de réduire grandement l'effort de conception en se concentrant sur l'algorithmique.

D'autre part, des efforts sont en cours pour aboutir à une approche permettant d'intégrer des modèles SCADE et Simulink à un système modélisé en AADL. Plusieurs exemples sont actuellement disponibles dans la distribution d'Ocarina. Dans ces exemples, toute phase de codage manuel est supprimé, ce qui montre que dans certains cas, l'approche dirigé par les modèles peut conduire à une génération complète du système. Il reste néanmoins à garantir que la sémantique de chaque bloc est préservée. Ceci reste un enjeu de recherche.

Remerciements

Nous tenons à remercier nos partenaires, les sociétés Ellidiss Technologies et Virtualys, pour leur travail commun à l'élaboration du modèle AADL présenté dans cet article. Par ailleurs, nous remercions l'ensemble de contributeurs aux projets Ocarina et Cheddar pour avoir permis la réalisation de ces chaînes d'outils

Références

[1] 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.

[2] J. 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.

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

[4] J. A. de la Puente, J. Zamorano, J. F. Ruiz, R. Fernandez, and R. García. The design and implementation of the Open Ravenscar Kernel. *Ada Letters*, 11(1), Mar. 2001.

[5] V. Debruyne, F. Simonot-Lion, and Y. Trinquet. EAST-ADL - An Architecture Description Language. pages 181–195. Book on Architecture Description Languages, IFIP International Federation for Information Processing, Springer Verlag, volume 176, 2005.

[6] I. Demeure and C. Bonnet. *Introduction aux systèmes temps réel*. Collection pédagogique de télécommunications, Hermès, septembre 1999.

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

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

[9] 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.

[10] Ellidiss. ADELE : a versatile system architecture graphical editor based on AADL. http ://g-forge.enseeiht.fr/projects/adele, 2007.

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] 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.

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

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

[22] 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*, Mar. 2005.

[23] 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.

[24] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. OCARINA : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. In *Reliable Software Technologies'09 - Ada Europe*, volume LNCS, pages 237–250, Brest, France, jun 2009.

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

[26] A. E. Rugina, K. Kanoun, and M. Kaaniche. The ADAPT Tool : From AADL Architectural Models to Stochastic Petri Nets through Model Transformation. 7th European Dependable Computing Conference (EDCC), Kaunas : Lituanie, 2008.

[27] 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.

[28] SEI. OSATE : An extensible Source AADL Tool Environment. *SEI AADL Team technical Report*, December 2004.

[29] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols : An Approach to real-time Synchronization. *IEEE Transactions on computers*, 39(9) :1175–1185, 1990.

[30] 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, Feb. 2007.

[31] F. Singhoff, A. Plantec, and P. Dissaux. Can we increase the usability of real time scheduling theory ? The Cheddar project. pages 240–253. 13th International Conference on Reliable Software Technologies, Ada-Europe, Lecture Notes on Computer Sciences, Springer-Verlag editor, volume 5026, Venice, June 2008.

[32] O. Sokolsky, I. Lee, and D. Clark. Schedulability Analysis of AADL models . International Parallel and Distributed Processing Symposium, IPDPS 2006, Volume 2006,, Apr. 2006.

[33] J. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, October 1988.

[34] 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.

[35] J. F. Tilman. Building Tool Suite for AADL. volume 176, pages 197–207. IFIP International Federation for Information Processing, Springer Verlag editor, 2005.

[36] 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.

[37] T. Vergnaud. Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées. *Thèse de l'ENST-Paris*, décembre 2006.

[38] 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.

[39] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[40] B. Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, École Nationale Supérieure des Télécommunications, nov 2008.

A Extrait d'un modèle AADL : modèle du système radar

```
system radar
end radar;

system implementation radar.impl
subcomponents
  aerial : device antenna;
  rotor : device motor;
  monitor : device screen;
  main : process processing.others;
  cpu : processor leon2;
  vme : bus vme;
  ram : memory ram;
connections
  data port aerial.antenna_out -> main.receive_pulse;
  data port rotor.motor_out -> main.get_angle;
  event port main.send_pulse -> aerial.antenna_in;
  event port main.to_screen -> monitor.screen_in;
  bus access vme -> aerial.vme;
  bus access vme -> rotor.vme;
  bus access vme -> monitor.vmpe;
  bus access vme -> cpu.vme;
  bus access vme -> ram.vme;
properties
  Actual_Memory_Binding => reference ram applies to main;
  Actual_Processor_Binding => reference cpu applies to main;
end radar.impl;

device antenna
features
  antenna_in : in event port;
  antenna_out : out data port radar_types::target_distance;
  vme : requires bus access vme;
properties
  Compute_Execution_Time => 1 ms .. 2 ms;
  Deadline => 5 ms;
  Period => 5 ms;
end antenna;

device motor ...
device screen ...

process processing
features
  to_screen : out event port;
  send_pulse : out event port;
  receive_pulse : in data port radar_types::target_distance;
  get_angle : in data port radar_types::motor_position;
end processing;

process implementation processing.others
subcomponents
  receive : thread receiver.impl;
  analyse : thread analyser.impl;
  display : thread display_panel.impl;
  transmit : thread transmitter.impl;
  control_angle : thread controller.impl;
connections
  data port receive_pulse -> receive.receiver_in;
  event port display.display_out -> to_screen;
  event port transmit.transmitter_out -> send_pulse;
  data port get_angle -> control_angle.controller_in;
  data port receive.receiver_out -> analyse.from_receiver;
  data port analyse.analyser_out -> display.display_in;
  event port transmit.transmitter_out -> analyse.from_transmitter;
  data port control_angle.controller_out -> analyse.from_controller;
end processing.others;

thread receiver
features
  receiver_out : out data port radar_types::target_distance;
  receiver_in : in data port radar_types::target_distance;
end receiver;

thread implementation receiver.impl
```

```
calls {
  rs : subprogram receiver_spg;
};
connections
  parameter rs.receiver_out -> receiver_out;
  parameter receiver_in -> rs.receiver_in;
properties
  Cheddar_Properties::Fixed_Priority => 63;
  Dispatch_Protocol => Periodic;
  Compute_Execution_Time => 1 ms .. 2 ms;
  Deadline => 15 ms;
  Period => 15 ms;
end receiver.impl;

subprogram receiver_spg
features
  receiver_out : out parameter radar_types::target_distance;
  receiver_in : in parameter radar_types::target_distance;
properties
  Source_Language => Ada95;
  Source_Name => "radar.receiver";
end receiver_spg;

thread analyser ...
thread display_panel ...
thread transmitter ...
thread controller ...

processor leon2
features
  vme : requires bus access vme;
properties
  Deployment::Execution_Platform => LEON_ORK;
  Clock_Period => 200 ns;
  Scheduling_Protocol => Posix_1003_Highest_Priority_First_Protocol;
end leon2;

bus VME ...
memory RAM ...

...
```

Listing 5 – Modèle AADL du système radar