Définition d'une famille de patrons de transformation pour l'analyse de modèles AADL

Xavier Renault et Jérôme Hugues

Résumé : AADL (Architecture Analysis and Design Language) est un langage de description d'architecture permettant une multitude d'analyses formelles ou semi-formelles, en utilisant par exemple des analyses statiques ou, encore, des techniques de model-checking. AADL fournit un niveau d'abstraction intéressant pour exprimer de nombreuses constructions utiles à la réalisation de systèmes embarqués temps réel. Parallèlement, nous remarquons que les propriétés à vérifier sur ces systèmes ne couvrent bien souvent qu'un sous-ensemble des éléments du modèle (composants ou propriétés non-fonctionnelles). Dans le présent article il est montré comment tirer parti de ces informations pour définir plusieurs patrons de transformation d'AADL vers les réseaux de Petri adaptés à la propriété à vérifier. Les propriétés qualitatives du système (comme la détection d'interblocage ou la traçabilité des messages échangés) peuvent être analysées à l'aide des réseaux de Petri colorés, en utilisant l'environnement CPN-AMI. Les propriétés quantitatives (comme l'ordonnancement du système ou la vérification du dimensionnement des tampons de communication) peuvent être traitées à l'aide des réseaux de Petri temporels, en utilisant l'environnement Tina. Les auteurs se sont intéressés à l'élaboration de patrons génériques pouvant être annotés en accord avec le type de propriété à traiter ; ils montrent comment ces patrons permettent de limiter l'explosion combinatoire en restreignant le modèle à analyser aux seuls blocs utiles.

Mots clés : AADL, architecture, réseaux de Petri, patron de transformation, temps réel

1. Introduction

AADL (Architecture Analysis and Design Language) est un langage dédié à la description d'architecture de systèmes temps réel, répartis et embarqués. Il est standardisé par la SAE [6], la seconde révision du standard ayant été publiée en janvier 2009. Basé sur la description de composants, il permet de modéliser à la fois les aspects logiciels et matériels d'un système : les composants logiciels permettent de décrire les éléments applicatifs d'une architecture, tandis que les composants matériels décrivent les éléments servant de support à l'exécution du système. Ces composants sont organisés de façon hiérarchique, et assemblés par des interfaces bien définies, de façon à permettre à l'architecte de séparer les préoccupations fonctionnelles de leurs implantations.

Une large communauté académique s'est regroupée autour de ce langage avec pour objectif de lier descriptions AADL et outils d'analyse formelle. On peut ainsi citer des passerelles vers des outils d'analyse d'ordonnançabilité, de *model-checking* (basés sur des réseaux de Petri colorés, stochastiques ou temporisés ; les outils FIACRE ou BIP, LOTOS, les automates temporisés, UPPALL), Alloy, ou encore Lustre et Signal. La liste complète des ces projets est maintenue sur le site du comité AADL [9]. Ces liaisons définissent une projection exhaustive du modèle AADL, puis l'analyse. Cela a un coût non négligeable lors de l'analyse : le modèle peut inclure des éléments superflus qui vont impacter l'analyse : allongement du temps d'analyse voire explosion combinatoire qui empêche le processus de converger.

Dans cet article, nous montrons une famille de patrons de modélisation basée sur des réseaux de Petri adaptable en fonction des propriétés à vérifier. Ces travaux font suite à une série de travaux

menés dans le cadre du projet ANR Flex-eWare ainsi qu'aux travaux de thèse de X. Renault.

2. Les réseaux de Petri

Les réseaux de Petri sont un formalisme permettant de spécifier formellement le comportement de systèmes à parallélisme et distribués. Il en existe différentes variantes, permettant de prendre en compte des éléments particuliers d'un système (types de données, temps, erreurs, etc.). Nous allons présenter les variantes que nous avons utilisées pour mettre en œuvre notre approche : les réseaux de Petri colorés et les réseaux de Petri temporels à priorité.

Les réseaux de Petri sont des graphes bipartites, avec des nœuds (places) modélisant des ressources et pouvant contenir des jetons, symbolisant la quantité disponible de celles-ci, et des nœuds (transitions) permettant de consommer ou de produire des jetons. [7] fournit une référence complète sur les réseaux de Petri.

2.1 Les réseaux de Petri colorés

Un réseau de Petri coloré est un 5-uplet <P, T , C, Pre, Post> où :

- P est un ensemble de places ;
- T est un ensemble de transitions ;
- C est une famille indexée par $P \cup T$ d'ensembles C(x) ; C(x) est appelé domaine de couleurs associé à la place ou à la transition x ;
- Pré et Post sont deux familles indexées par $P \cup T$ d'applications Pre(p,t) et Post(p,t) telles que pour tout (p,t) de $P \cup T$, Pré(p,t) et Post(p,t) aient pour domaine C(t) et co-domaine Bag(C(p)). Pré et Post sont respectivement appelées fonction d'incidence avant et fonction d'incidence arrière.

On utilise des prédicats (gardes) pour imposer des contraintes sur le type de jetons autorisés à franchir une transition. Ce sont des expressions booléennes qui ne permettent de franchir une transition que si elles sont vraies.

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

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

2.2 Les réseaux de Petri temporels à priorité

Les réseaux de Petri temporels sont une extension des réseaux de Petri classiques, où une horloge et un intervalle de temps sont associés à chaque transition du réseau. L'horloge mesure le temps écoulé depuis l'instant où la transition est franchissable. L'intervalle de temps intervient comme une condition de franchissement.

Une transition est dite franchissable si :

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

La notion de priorité permet de considérer une information supplémentaire lorsque deux transitions sont franchissables (puisqu'une seule transition peut être franchie à la fois). Dans ce cas, la transition avec la plus haute priorité est choisie.

3. Patrons génériques de transformation d'AADL vers les réseaux de Petri

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

Nous proposons une méthode permettant de procéder à l'analyse comportementale d'un système spécifié en AADL. Nous avons pour cela :

- analysé le standard AADL afin d'en extraire les éléments relatifs au comportement des systèmes,
- proposé des patrons génériques de transformation depuis AADL vers les réseaux de Petri place-transitions,
- établi des règles permettant de raffiner les modèles formels génériques produits en exploitant les informations de la spécification AADL.

3.1 Patrons génériques

Les composants d'une spécification AADL qui concentrent le comportement de l'application sont les flots d'exécution (*ou threads*). Leur cycle de vie est le suivant :

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

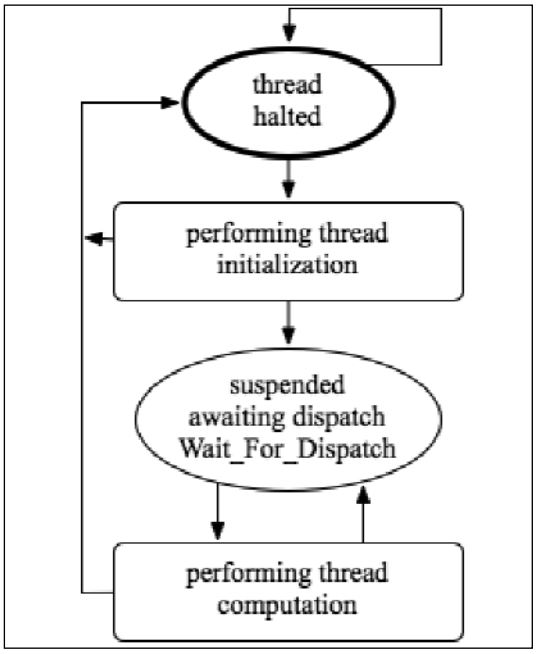

Figure 1 : Cycle de vie d'un flot AADL

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

Nous nous sommes intéressés aux attributs des flots d'exécution contenant de l'information sur leur comportement et sur les différentes échéances temporelles qu'ils doivent respecter :

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

Nous avons identifié les patrons élémentaires permettant de transformer une spécification AADL en un modèle de réseaux de Petri :

- le squelette du flot d'exécution, servant de base à l'intégration des autres patrons de transformation,
- les éléments liés au déclenchement du flot d'exécution,
- les éléments liés à la réception, au traitement et à l'envoi de données,
- les éléments liés à l'échange de données entre les acteurs du système.

Nous présentons la démarche mise en œuvre pour produire les patrons liés à la génération du squelette d'un flot d'exécution. Un flot d'exécution démarre dans un état stable, passe par une phase d'initialisation, puis attend le déclenchement de son exécution. Une fois déclenché, il effectue un certain nombre d'actions, puis revient dans cet état d'attente de déclenchement. Il y a donc trois principaux états dans ce patron : l'état initial, l'état d'attente et l'état de calcul. Pour passer d'un état à l'autre, différentes transitions sont mises en place, comme le montre la partie gauche de la figure 2.

La partie droite montre la traduction qui est faite du patron en réseaux de Petri Place-Transition, où aucune valeur d'attributs AADL n'est exploitée. La figure 2 montre la spécialisation de ce patron selon les informations disponibles dans la spécification AADL.

Spécialisation en réseaux de Petri colorés : comme il s'agit du patron servant à en intégrer d'autres, l'information primordiale qui lui est ajoutée concerne les mécanismes de discrimination et d'identification des flots d'exécution du système. Cela est réalisé en introduisant une classe de couleur modélisant les identifiants de flots d'exécution (Figure 3).

Spécialisation en réseaux de Petri temporels : ici, les informations pouvant être ajoutées sont

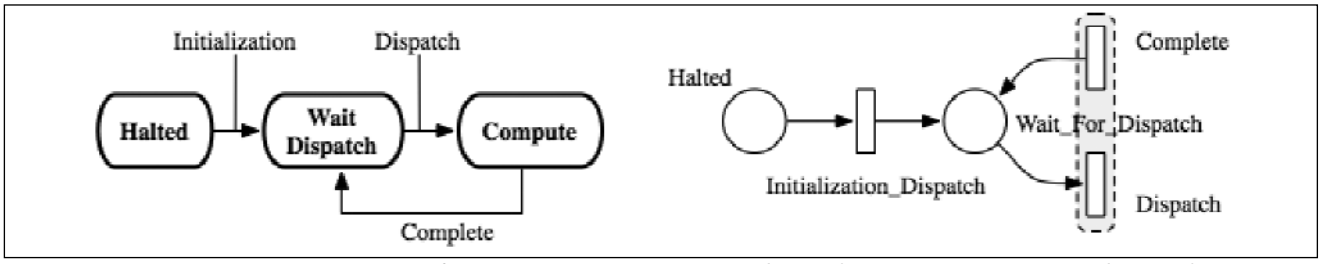

Figure 2 : Patron de squelette de flot d'exécution en réseaux de Petri (à droite), à partir de son cycle de vie (à gauche)

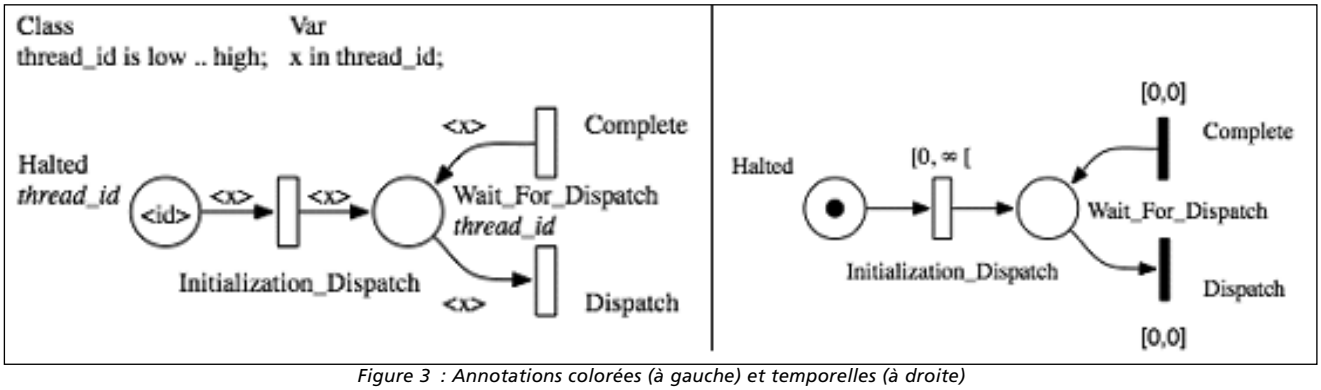

Figure 3 : Annotations colorées (à gauche) et temporelles (à droite)

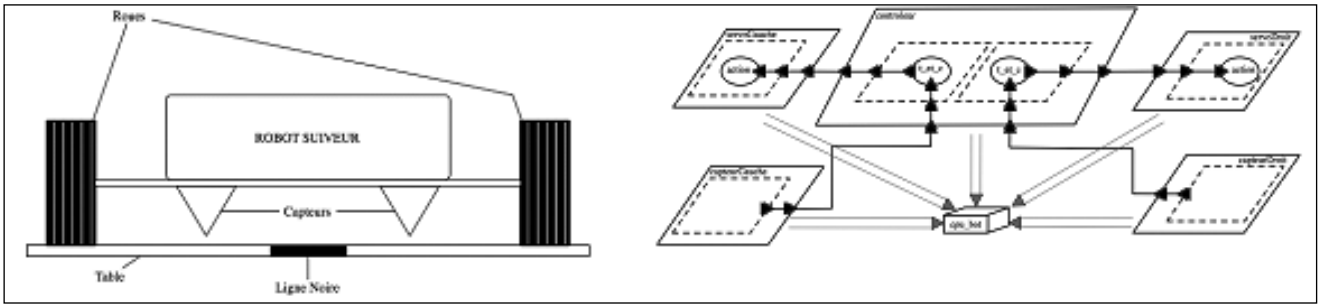

Figure 4 : Modèle AADL (à droite) d'un robot suiveur de ligne (à gauche)

celles relatives, par exemple, à l'intervalle de déclenchement du flot s'il est périodique. Dans ce cas, un intervalle de temps sera associé à la transition Dispatch (elle même pouvant être combinée à des patrons reflétant d'autres contraintes exprimées dans la spécification AADL).

Les règles de transformation ont donc été systématiquement établies pour chacun des attributs considérés : nous avons identifié l'impact de leurs valeurs sur les patrons, de façon à modéliser formellement les différentes contraintes du système. Davantage de détails quant à ce processus sont décrits dans [8].

4. Application à un cas d'étude

Nous présentons dans cette section un cas d'étude permettant de montrer l'adéquation de notre approche pour répondre aux besoins d'analyse des systèmes temps réel embarqués. Nous considérons un robot « suiveur de ligne » : il est constitué de deux capteurs, d'un module de traitement des données (le contrôleur) et de servomoteurs (asservis par le contrôleur) permettant d'actionner les roues (droites ou gauche) du robot. La figure 4 présente le modèle AADL correspondant à ce cas d'étude.

Le comportement du robot est le suivant :

- À l'initialisation, les servomoteurs sont actifs, faisant avancer le robot en ligne droite ;
- Les servomoteurs restent actifs tant que la ligne est droite ;
- Si la ligne bifurque sur la gauche ou la droite, alors :
  1. Un des capteurs passera au-dessus de la ligne noire ;
  2. L'information est capturée et transmise au contrôleur ;
  3. Le servomoteur concerné est alors éteint, faisant pivoter de ce fait le robot dans la bonne direction ;
  4. Dès que le capteur repasse en zone « blanche », alors le servomoteur concerné est réactivé.

Traditionnellement, nous cherchons à vérifier que le comportement de ce système robotique est correct : nous souhaitons valider l'ordonnançabilité et le bon dimensionnement des ?les de messages et des autres ressources, ainsi que le bon fonctionnement des moteurs et du lien qui les lie aux capteurs.

Les patrons que nous proposons permettent de vérifier ces propriétés en tirant parti de chacun des formalismes que nous avons présenté : les propriétés sont analysées à l'aide de techniques de *model-checking* mises en œuvre à partir des réseaux de Petri colorés (resp. temporels) pour les propriétés qualitatives (resp. quantitatives).

Ce processus de transformation a été automatisé dans une chaîne d'outils [1, 2] spécialisée : nous avons implanté ces règles dans le compilateur AADL Ocarina [5]. Les modèles générés sont ensuite exploités dans des environnements dédiés aux réseaux de Petri, comme CPN-AMI [4] (réseaux colorés) ou Tina [3] (réseaux temporels).

Nous présentons les résultats obtenus lors de l'analyse des modèles en réseaux de Petri colorés. Les propriétés suivantes ont été exprimées en LTL (Linear Tree Logic), et vérifiée à l'aide de méthodes de *model-checking* :

- Les capteurs échantillonneront les données régulièrement : ceci permet de s'assurer que le robot sera toujours en mesure de corriger sa position ;
- Lorsqu'un capteur produit une donnée, elle est analysée par le contrôleur : cette propriété permet de vérifier qu'aucune donnée n'est perdue ou non traitée ;
- L'activation d'un contrôleur implique la génération d'une commande pour un servomoteur : cette propriété complète la propriété n°1, en montrant que les mouvements des roues sont conformes aux données reçues.

Le tableau suivant présente des statistiques sur le nombre d'éléments du réseau de Petri coloré généré par Ocarina. Il présente également des statistiques relatives au graphe des marquages accessible associé, généré soit à l'aide de l'outil Prod, soit à l'aide de l'outil GreatSPN. Prod calcule de façon exhaustive l'espace d'état du système, à l'inverse de GreatSPN qui travaille sur des états symboliques (graphe quotient, ou GQ).

	Réseau de Petri coloré	GMA (Prod)	GQ (gSPN)
Nœuds	*38 places 29 transitions*	*65537 états*	*1025 états symboliques*
Arcs	*92*	*425985*	*NC*

Le réseau de Petri généré reste assez petit, mais le nombre d'état engendré est grand, considérant le fait que l'on cible un système où s'exécutent six flots. Cela reste cependant très raisonnable pour les model-checkers explicites qui sont capables de gérer jusqu'à 10^7 états. Si le nombre d'états à traiter est plus important nous utilisons les *model-checkers* symboliques comme GreatSPN, permettant de stocker davantage d'états en mémoire.

Il est important de relever que l'utilisation conjointe de différents formalismes pour analyser un système permet d'affiner le processus de vérification : en effet, les notions de temps ou de priorité n'intervenant pas en réseaux de Petri colorés, elles ne sont pas exprimées dans les patrons : le GMA contient alors des chemins *a priori* irréalisables qui peuvent donner lieu à des faux positifs lors de l'analyse. L'exploitation des réseaux de Petri temporels de façon complémentaire permet d'éliminer ces faux positifs du processus de vérification en ne conversant que les chemins d'exécution calculés par l'ordonnanceur du système.

Dans notre cas d'étude, si nous procédons à des analyses uniquement depuis les réseaux de Petri colorés, nous déduisons qu'il est possible que des données ne soient pas traitées par le contrôleur : en effet, puisque la notion de temps n'est pas exprimée, il existe un chemin d'exécution du système où les capteurs (qui sont des tâches périodiques) vont s'exécuter à l'infini, empêchant les autres éléments du système de le faire. Ce chemin d'exécution est irréalisable dès l'instant où la notion de période, voire de priorité entre composants, intervient. Ces éléments étant pris en compte par le formalisme des réseaux de Petri temporels à priorité, nous pouvons les exploiter pour éliminer ces traces fautives. Néanmoins, les réseaux de Petri colorés sont intéressants pour analyser des systèmes qui ne peuvent être vérifiés par un *model-checker* temporisé, voire valider des propriétés de sûreté quel que soit l'ordonnancement du système. De ce fait, l'utilisation conjointe des deux formalismes se justifie suivant le type de propriété à établir.

5. Conclusion

L'utilisation de méthodes formelles de manière transparente au travers d'un cadre de modélisation tel que AADL s'est largement développé. Nous notons cependant que leur utilisation requiert une certaine technicité.

Dans cet article, nous avons présenté brièvement comment combiner deux approches de *model-checking* pour évaluer le comportement de systèmes temps réel, en montrant comment combiner AADL et réseaux de Petri colorés ou temporels. Cette approche permet de choisir le formalisme le mieux adapté à la propriété à prouver, suivant qu'elle fasse apparaître la notion de temps ou non, qu'elle soit compatible avec les limites des *model-checkers*, ou son domaine de validité.

Les travaux en cours visent à mieux guider le concepteur dans le choix de l'une ou l'autre de ces méthodes, mais aussi dans l'expression des propriétés à vérifier à l'aide de notations plus simples à appréhender que LTL.

6. Références

[1] Xavier Renault, Fabrice Kordon et Jérôme Hugues : *Adapting models to model checkers, a case study: Analysing AADL using time or colored Petri Nets* ; IEEE/IFIP 20th International Symposium on Rapid System Prototyping, Paris, juin 2009.

[2] Xavier Renault, Fabrice Kordon et Jérôme Hugues : *From AADL architectural models to Petri Nets: Checking model viability* ; 12th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'09), Tokyo, Japon, mars 2009 pp. 313-320.

[3] *The TINA Home page*, LAAS, URL: http://www.laas.fr/tina/description.php.

[4] *The CPN-AMI home page*, MoVe-Team, URL : http://www.lip6.fr/cpn- ami.html.

[5] Jérôme Hugues, Bechir Zalila et Laurent Pautet : *From the prototype to the final embedded system using the Ocarina AADL tool suite* ; ACM Transactions on Embedded Computing Systems (TECS). 7 (4). juillet 2008, pp. 1-25.

Model) visant les architectures embarquées. Comme Fractal, LwCCM n'est pas associé à un langage de programmation particulier. Le modèle de composant lui-même est légèrement différent de Fractal dans ses objectifs : LwCCM définit en standard des mécanismes d'interaction (envoi/réception d'événements et appels d'opérations). L'approche de conception induite par le standard implique que le code utilisateur se repose sur ces mécanismes de communication, et n'a donc pas de contrôle sur l'exécution et les interactions. Cela permet la programmation de ce code technique de façon séparée de la programmation du code fonctionnel et de confier à la chaîne de compilation la combinaison de ces deux parties pour aboutir au code exécutable complet. L'une des conséquences est que, contrairement à ce que son nom peut suggérer, une implémentation de LwCCM ne repose pas nécessairement sur un ORB CORBA ; il est possible d'utiliser tout autre exécutif. Ainsi, Thales a implémenté LwCCM au sein d'un framework, MyCCM. Dans le cadre du projet Flex-eWare, une version particulière a été développée, MyCCM-HI, utilisant l'exécutif AADL PolyORB-HI développé par Télécom ParisTech. La situation étudiée par Thales consiste à spécifier, à concevoir, à déployer et à exécuter de façon correcte la partie de code dévolue à la reconfiguration. Cela a été fait en étendant les modèles LwCCM. Des informations de déploiement supplémentaires ont été introduites, regroupées dans un langage appelé COAL (Component-Oriented Architecture Language). Ce dernier a été en grande partie défini par la transposition des concepts de déploiement définis dans des standards tels qu'AADL ou MARTE pour compléter le langage IDL3 de LwCCM. La problématique des travaux a été de préserver l'intégrité des données manipulées par l'application en cas de changement de mode. Le résultat des travaux est un framework capable de prendre en charge des modes de configuration et les mécanismes de changement de mode associés. Différentes stratégies de changement de mode ont été définies afin de répondre aux différentes configurations architecturales possibles.

5. Conclusion

Notre tentative de rapprochement de technologies différentes en se focalisant sur les besoins de flexibilité du logiciel s'est au final montrée féconde. Par exemple, plusieurs besoins existant à des étapes différentes du cycle du logiciel ont pu être résolus par les mêmes approches. Ces approches se sont révélées indépendantes des technologies d'implémentation présentes chez les partenaires et représentatives de la diversité des pratiques dans le domaine de l'embarqué.

Nous avons proposé un certain nombre de solutions. Leur efficacité a été démontrée en parcourant complètement des préfigurations de chaînes d'outils couvrant la modélisation, le développement et la génération d'une image mémoire du code exécutable. Nous avons ciblé différentes cartes du commerce (BeagleBoard/ARM+ DaVinci [6], Gumstick/ARM [7], Raven/ATMega [8], Sam7/ARM [9]…)

Pour autant, un certain nombre de besoins en flexibilité logicielle tels que nous les avons identifiés ne sont pas couverts. Citons, par exemple, la plupart des réponses aux exigences liées au déploiement qui ne sont pas résolues statiquement par les chaînes de compilations actuelles. De quoi nourrir des investigations et des développements futurs. Le principal défi pour ces solutions restera d'être intégrées et réunies dans un outillage de niveau industriel à disposition des divers « joueurs » dans le cercle du logiciel embarqué.

6. Références

[1] *http://www.flex-eware.org*

[2] *http://fractal.ow2.org* le site donne des url vers les principales implémentations du modèle Fractal à l'exception de think *http://think.ow2.org* et de mind *http://mind.ow2.org* qui constituent des projets indépendants sur le même site d'objectWeb2

[3] *http://myccm-hi.wiki.sourceforge.net*

[4] *http://www-list.cea.fr/labos/gb/LLSP/oasis/OASIS_presentation_gb.htm*

[5] *http://www.ec3m.net*

[6] *http://beagleboard.org*

[7] *http://www.gumstix.com*

[8] *http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4395*

[9] *http://www.atmel.com/dyn/products/devices.asp?family_id=605*

Remerciements

L'auteur remercie Fabrice Kordon du LIP6, Lionel Seinturier et Frédéric Loiret de l'INRIA, Laurent Pautet de Télécom Paris Tech, Juan Navas d'Orange Lab, Ansgar Radermacher et Arnaud Cuccuru du CEA/List, Antonio Kung et Christophe Jouvray de Trialog, ainsi que tous les autres partenaires du projet Flex-eWare, en particulier Frédéric Gilliers et Grégory Haïk de Thales sans oublier Franck Bernier de Schneider Electric, ainsi que les thésards aujourd'hui docteurs, dont les idées nourrissent cette contribution : Étienne Borde, Juraj Polakovic et Marc Poulhiès.

[6] *Architecture Analysis & Design Language V2* (AS5506A), SAE, janvier 2009.

[7] C. Girault et R. Valk : *Petri Nets for Systems Engineering* ; Springer Verlag, 2003.

[8] Xavier Renault : *Mise en œuvre de notations standardisées, formelles et semi-formelles dans un processus de développement de systèmes embarqués temps réel répartis* ; Thèse de Doctorat, décembre 2009.

[9] *Wiki du comité AADL*, URL : https://wiki.sei.cmu.edu/aadl/index.php/Main_Page.

[10] *UML et/ou AADL* : Actes du Premier Atelier International « UML & AADL » ; Génie Logiciel, n° 80, pages 1-44, mars 2007.