Vérification formelle de spécifications AADL via FIACRE

B. Berthomieu[†‡] J.-P. Bodeveix[*‡]
S. Dal Zilio[†‡] M. Filali[*‡] F. Vernadat[†‡]
[*]IRIT-CNRS ; Université de Toulouse ; 118 route de Narbonne, F-31062 Toulouse, France
[†]LAAS-CNRS ; Université de Toulouse ; 7 avenue colonel Roche, F-31077 Toulouse, France
[‡]Université de Toulouse ; UPS, INSA, INP, ISAE ; UT1, UTM

I. Introduction

Les systèmes embarqués se caractérisent par des logiciels critiques pour lesquels la phase de conception est désormais reconnue comme cruciale. Dans ce contexte, au sein du pôle de compétitivité AESE, plusieurs acteurs de l'aéronautique et du spatial ont décidé de réunir leurs efforts pour développer une ensemble de méthodes et d'outils qui permettront de construire les logiciels sûrs de demain. Le projet TOPCASED [Top] fait partie de cette initiative. Un des langages retenus pour la modélisation des systèmes embarqués est le langage d'architecture AADL (Architecture and Analysis Design Language).

Les langages d'architecture ont été proposés pour permettre des analyses dès la phase de conception. Plus particulièrement, dans le domaine de l'aéronautique, la SAE (Society of Automotie Engineers) propose le langage AADL [SAE02] qui permet de modéliser des *modèles* de systèmes aussi biens logiciels que matériels.

L'objet de cet article est de présenter la chaîne de vérification du langage AADL, chaîne aujourd'hui intégrée dans l'environnement TOPCASED. Cette chaîne de vérification est construite autour du langage pivot FIACRE [FIA] dont les moteurs de vérification sont aujourd'hui les outils TINA [BRV04] et CADP [FGM+96]. Dans ce qui suit, tout d'abord, nous présentons le langage FIACRE ainsi que le langage AADL et illustrons la modélisation FIACRE de quelques protocoles de communication. Nous concluons sur les expérimentations menées à l'aide de la chaîne de vérification élaborée. Nous présentons aussi la poursuite de ces travaux dans le cadre de nouveaux projets.

Ce travail a été partiellement financé par le projet, du pôle de compétitivité AESE, Topcased ainsi que par la région Midi-Pyrénées.

II. Le langage Fiacre

Le langage FIACRE [FGP+08] offre un cadre formel pour représenter, dans la même syntaxe, les aspects comportementaux ainsi que les contraintes temporelles d'un système. L'objectif est de pouvoir utiliser la spécification d'un système en FIACRE pour mener des activités de vérification formelle ou de simulation.

La conception du langage FIACRE s'inspire des résultats issus de décennies de recherches sur les systèmes temps-réels et sur la théorie de la concurrence. Ainsi, les primitives de synchronisation fournies par le langage sont empruntées au modèle des réseaux de Petri temporels [MF76], [BD91], tandis qu'on peut comparer la possibilité d'intégrer contraintes de temps et contraintes de priorités dans le langage à ce qui existe dans le cadre de BIP [BBS06]. Finalement, le langage FIACRE supporte plusieurs mécanismes facilitant une modélisation compositionnelle des systèmes, tel qu'un opérateur de composition parallèle entre composants ainsi qu'un système de typage des ports de communications qui est proche de ce qu'on retrouve dans des langages tels que E-Lotos et Lotos-NT [Gar95], [Sig00]. protocoles de communication AADL en Bip [BBS06] et en FIACRE.

A. Programme FIACRE

La syntaxe du langage FIACRE est stratifiée autour de deux notions principales: les *processus*, qui décrivent les comportement de composants séquentiels; et les *composants*, qui décrivent un système comme une composition de processus, éventuellement de manière hiérarchique. Le listing 1 donne un exemple de programme FIACRE qui modélise l'exemple classique du "token ring". Il s'agit d'un réseau d'unités de calculs, logiquement organisés dans une topologie en anneau, qui synchronisent leurs communications par le biais d'un

Listing 1. Un exemple simple de programme FIACRE: l'anneau à jeton

```
process Start
  [start0: none, start1: none, start2: none] is
  states s0, s1
  from s0 select
      start0
  [] start1
  [] start2
  end ;
  to s1

process Node
  [prev : none, succ : none, start : in none] is
  states idle, wait, cs, st_1
  from idle select
      start ; to st_1
  [] prev ; to st_1
  end
  from st_1 succ ; select to idle [] to wait end
  from wait prev ; to cs
  from cs   succ ; to idle

component root is
        port s0 : none, s1 : none, s2 : none,
             p0 : none, p1 : none, p2 : none,
        par * in
            Start[s0,s1,s2]
        || Node[p0, p1, s0]
        || Node[p1, p2, s1]
        || Node[p2, p0, s2]
        end

root
```

jeton qui circule parmi eux et permet de contrôler l'accès aux canaux de communication.

Un programme FIACRE est une séquence de déclarations. Plus particulièrement: des déclarations de types, pour décrire des contraintes sur les valeurs échangées entre processus; et des déclarations de processus et de composants, pour décrire le comportement des éléments du système. Un programme FIACRE est statique: les instances de processus et leur architecture sont connues dès la compilation. FIACRE est un langage fortement typé, ce qui signifie que des annotations de type sont exploitées afin de garantir la compatibilité des données manipulées par les processus (par exemple, on ne pourra pas envoyer une entier sur un port là où une valeur booléenne est attendue).

1) Processus FIACRE: Un processus est défini par un ensemble d'états de contrôle, de paramètres et de macro-transitions, c'est-à-dire une expression complexe qui décrit l'ensemble des transitions pouvant être prises depuis cet état. La macro-transition décrit également de quelle manière les paramètres sont mis à jour après chaque transition. Par exemple, la déclaration suivante définit que le processus `T` peut interagir sur deux ports: `p`, qui transmet des valeurs booléennes, et `q`, qui ne peut être utilisé que pour la synchronisation. Le processus possède également deux paramètres: `v`, qui est un entier, et `u`, qui est une référence (pour une variable) partagée pour un tableau de booléens de taille `5`.

```
process T
  [p : bool, q : none]
  (v : int, &u : array 5 of bool) is
  ...
```

Les macro-transitions sont définies par un langage d'expressions bâti au dessus "d'opérateurs déterministes" classiques, qu'on retrouve dans la plupart des langages de programmation (affectations, conditionnelles, boucles `while` et composition séquentielle), et d'opérateurs non déterministe comme le choix et la communication par événements sur les ports de communication. Par exemple, la déclaration:

```
from s0    select
        (p!5 ; to s1)
   []   (v := v + 1 ; to s2)
end
```

exprime le fait que, dans l'état `s0`, le processus peut choisir (de manière non-déterministe) entre deux alternatives: soit envoyer la valeur `true` sur le port `p` puis changer vers l'état `s1`; ou bien incrémenter la valeur de la variable `v` et changer vers l'état `s2`. Une déclaration de processus peut définir plusieurs macro-transitions pour le même état, chacune pouvant être tirée de manière égale. Un processus, à la manière d'une classe dans un langage à objets, définit le comportement de base d'un élément du système. Les processus peuvent être instantiés et composés au sein de composants qui décrivent l'architecture du système que l'on cherche à modéliser.

2) Composant FIACRE: Un composant est défini comme la composition parallèle de processus et/ou d'autres composants. Cette opération est dénotée par la construction **par** ... || ... **end**. Les composants représentent à la fois l'unité de composition d'une spécification FIACRE, ainsi que l'unité d'instanciation des processus, et de création des ports et des variables partagées. La syntaxe des composants permet de restreindre le mode d'accès et la visibilité des variables partagées et des ports. Elle permet aussi d'associer des contraintes temporelles sur les communications et de définir des priorités entre les événements. Par exemple, dans un composant `C`, la déclaration **port** `p` : **none** définit un port appelé `p` qui est privé pour (ne peut pas être utilisé en dehors de) `C`. De même, la déclaration **port** `p` : **none in** [min,max] définit un port qui ne peut interagir que `min` unités de temps après qu'il ait été activé et doit être utilisé ou désactivé avant `max` unités de temps (`min` et `max` sont des constantes entières).

B. Chaîne de vérification FIACRE

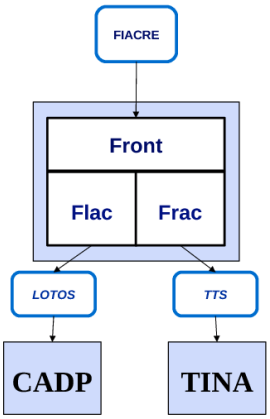

Fig. 1. Architecture de compilation FIACRE

La figure 1 représente la chaîne de vérification FIACRE développée dans le cadre du projet Topcased. L'architecture commune des compilateurs permet de factoriser au plus la chaîne de traduction vers les outils CADP et TINA. Une partie avant (FRONT) du compilateur, partagée, assure l'analyse lexicale, syntaxique, le typage, et la vérification d'un certain nombre de contraintes statiques. Toujours dans un souci de factoriser au plus la phase de traduction, l'outil FRONT embarque certaines procédures génériques d'élimination de constructions dérivées (expressions conditionnelles, par exemple) utilisées par l'un ou d'autre des générateurs. FRONT produit un arbre abstrait qui constitue le point d'entrée des générateurs FLAC (FIACRE to Lotos Adaptation Component) produisant du code LOTOS pour la connexion à CADP, et FRAC produisant des modèles au format "Time Transition Systems" (TTS) pour la connexion à TINA.

Des versions pré-compilées de FRONT, FLAC et FRAC pour diverses cibles sont disponibles sur la forge FIACRE [FIA]; leurs codes sources peuvent aussi y être librement téléchargés.

III. Modélisation de protocoles de communication AADL

A. Une brève introduction à AADL

Le langage AADL [SAE02] est le successeur de MetaH [Met97] développé par les laboratoires Honeywell et, à ce titre, il capitalise plus d'une dizaine d'années d'expérience dans le domaine des systèmes embarqués. AADL a de plus bénéficié de la connaissance autour des langages d'architecture (comme ACME [AG97] et Wright) développés à CMU.

B. Les éléments d'une architecture AADL

AADL définit trois catégories de composants, les composants logiciels (data, subprogram, thread, thread group, process), les composants matériels (memory bus, processor, device), et le composant system permettant de composer les deux catégories précédentes.

Chaque composant a un type qui spécifie son interface externe que ses implantations doivent satisfaire. Il contient l'interface du composant et une liste de propriétés. L'interface est décrite en termes de ports et d'accès proposés ou requis à un sous composant (data, bus, subprogram). De plus, on peut décrire les flots d'informations entre les entrées et les sorties du composant. Un type peut étendre un autre type, il hérite de toutes ses déclarations et propriétés. Il peut compléter et raffiner les déclarations du type dont il hérite. Les composants ainsi définis forment une hiérarchie d'extension.

Une ou plusieurs implantations sont associées à chaque type, elles représentent des variantes du composant se conformant au même type. Une instanciation est déterminée par un type et une implantation.

Enfin AADL possède un mécanisme d'annexe permettant d'étendre les mécanismes de description initiaux et donc d'introduire ainsi des langages dédiés. A titre d'exemple, l'annexe comportementale [FBF+07], aujourd'hui adoptée par le SAE, permet de raffiner le comportement des threads AADL.

C. Un exemple AADL

Dans cette section, nous exprimons en FIACRE les protocoles de base de communication par données du langage AADL.

Description de protocoles de communication en FIACRE. Bien qu'une sémantique précise d'exécution soit décrite dans le standard, il n'existe pas de spécification formelle de ces protocoles. L'expression FIACRE que nous avons étudiée [Pi10] peut donc être considérée comme une sémantique par traduction. Dans ce qui suit, nous considérons uniquement les protocoles de base AADL associées aux ports de communication de type data (sans file de communication) appelés immédiat ou différé.

Dans le cas du protocole de type immédiat, la communication a lieu effectivement lorsque le thread émetteur atteint la fin de son traitement (complete) et devient suspendu. On délivre alors au thread récepteur la valeur

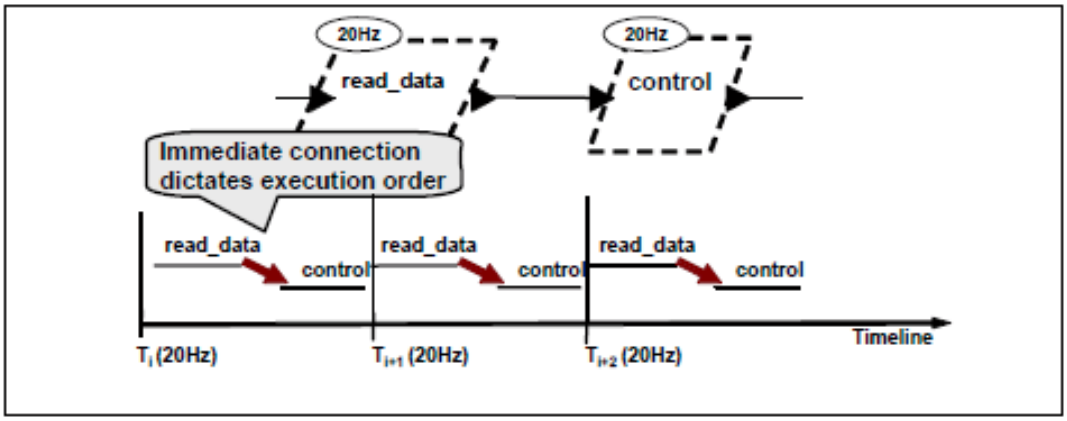

Fig. 2. Connexion immediate AADL

produite par le thread émetteur. Cette livraison est effective uniquement lorsque les threads émetteur et récepteur on été activés en même temps, ou encore lorsqu'une date égale à un multiple commun des périodes respetives des threads émetteur et récepteur a été atteint. Cet aspect est illustrée par la figure 2 issue de la documentation de AADL.

Dans le cas de protocole de type différé, la valeur est effectivement produite par le thread émetteur lorsque son échéance (deadline) a été atteinte. Cette valeur produite sera effectivement délivrée à la prochaine activation du thread récepteur. Cet aspect est illustrée par la figure 3 issue de la documentation de AADL.

a) Modélisation FIACRE.: Nous représentons un système AADL par un ensemble de processus FIACRE: un processus par thread AADL plus un processus FIACRE dédié à l'ordonnancement.

Un processus FIACRE associé à un thread AADL a un comportement illustré par la figure 4. Le processus FIACRE associé à l'ordonnancement a un comportement illustré par la figure 5. Pour les réveils périodiques ainsi que les échéances, nous utilisons des ports temporisés. Ainsi, pour contrôler un processus associé à un thread périodique, un port dont l'inervalle temporel est la période du thread est utilisée. A chaque période, l'ordonnanceur envoie un signal de dispatch au processus.

En ce qui concerne la communication, les automates suivants 6 et 7 illustrent le protocole immédiat et différé. Une description complète de ces protocoles est donnée dans [Pi10].

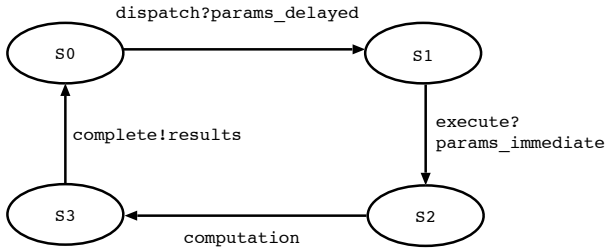

Fig. 4. Automate d'un processus FIACRE associé à un thread AADL

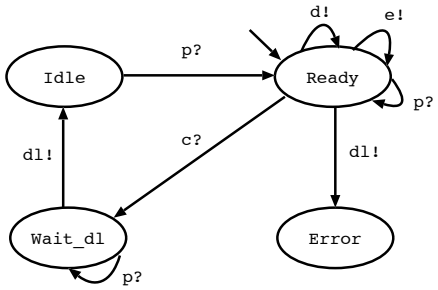

Fig. 5. Automate d'un processus FIACRE associé à l'ordonnancement

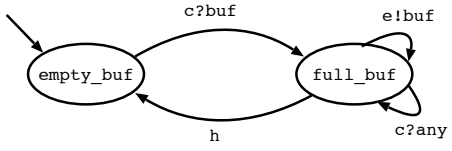

Fig. 6. Connexion immédate AADL en FIACRE

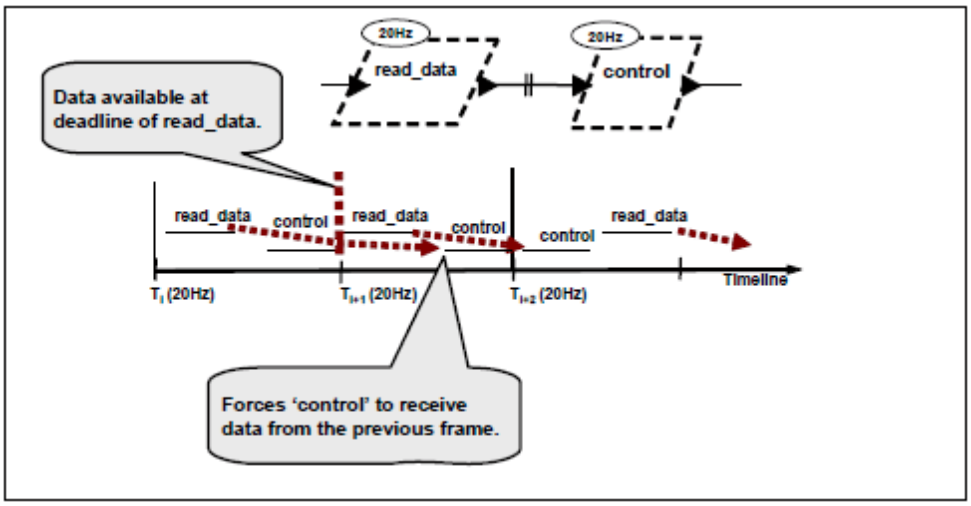

Fig. 3. Connexion différée AADL

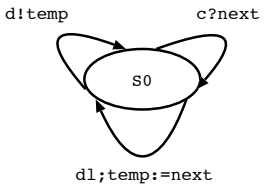

Fig. 7. Connexion différée AADL en FIACRE

D. Eléments de traduction AADL-FIACRE

La traduction AADL-FIACRE a été mise en oeuvre au sein de l'atelier TOPCASED en utilisant les techniques de l'IDM. La syntaxe abstraite du langage FIACRE est définie par son métamodèle. Nous avons utilisé le langage de transformation modèle vers modèle: Kermeta. Nous avons notamment utilisé la technique de programmation par aspets offerte par ce langage. A partir d'une instance de modèle FIACRE ainsi obtenue, nous générons le texte source FIACRE qui peut alors être analysé par le compilateur FIACRE présenté précédemment. Une description détaillée de cette chaîne de traduction est donnée dans [FGP+08].

IV. Vérification par TINA

TINA [BRV04], TIme Petri Net Analyzer, fournit un environnement logiciel permettant d'éditer et d'analyser par model-checking les réseaux de Petri temporels. Dans le cadre du projet TOPCASED, TINA a été étendue pour permettre la vérification de programmes Fiacre ou encore de programmes AADL en utilisant la traduction AADL2Fiacre.

A. La boîte à outils TINA

TINA (TIme Petri Net Analyzer)[TIN] est un environnement logiciel permettant l'édition et l'analyse de réseaux de Petri et réseaux temporels. Les différents outils constituant l'environnement peuvent être utilisés de facon combinée ou indépendante. Ces outils incluent:

`nd` (NetDraw) : `nd` est un outil d'édition de réseaux temporels et d'automates, sous forme graphique ou textuelle. Aussi, il intègre un simulateur "pas à pas" (graphique ou textuel) pour les réseaux temporels et permet d'invoquer les outils ci-dessous sans sortir de l'éditeur.

`tina`: cet outil construit des représentation de l'espace d'états d'un réseau de Petri, temporel ou non. Aux constructions classiques (graphe de marquages, arbre de couverture), `tina` ajoute la construction d'espaces d'états abstraits, basés sur les techniques d'ordre partiel, préservant certaines classes de propriétés, comme l'absence de blocage, les propriétés de certaines logiques, ou les équivalences de test ; ces facilités sont décrites dans [BRV04].
Les espaces d'états peuvent être produits dans divers formats: "en clair" (à but pédagogique), dans un format d'échange compact à destination des autres outils de l'environnement,

ou bien dans les formats acceptés par certains vérificateurs de modèles externes, comme MEC [ABC94] pour la vérification de formules du μ-calcul, ou les outils $CADP$ [FGM+96] pour notamment la vérification de préordres ou d'équivalences de comportements.
`tina` peut vérifier à la volée certaines propriétés "générales" telles que le caractère borné (condition souvent nécessaire pour une implantation ultérieure et souvent requise pour envisager d'autres vérifications), la présence de blocage - suivant le cas il sera attendu (toujours le cas pour un calcul qui doit en particulier terminer) ou non désiré (souvent le cas pour les systèmes réactifs) - la pseudo-vivacite qui permet de rechercher le "code mort" (transition jamais activée) ou encore la vivacité qui permet de s'assurer qu'une transition (ou une configuration du système) est atteignable à partir de tout état du comportement du système.
`plan`: cet outil permet le calcul de systèmes d'échéanciers. Il produit à la demande le système complet d'échéanciers, ou une solution de ce système, en délais ou dates, mis ou non sous forme canonique. `plan` est invoqué de facon transparente par `selt` pour obtenir des contre-exemple temporisés.
`struct`: il s'agit d'un outil d'analyse structurelle calculant des ensembles générateurs de flots ou semiflots, sur les places et/ou transitions.
`selt`: en plus des propriétés générales vérifiées par `tina`, il est le plus souvent indispensable de pouvoir garantir des propriétés spécifiques relatives au système modélisé. L'outil `selt` est un vérificateur de modèle (model-checker) pour les formules d'une extension de la logique temporelle $SE-LTL$ (State/Event LTL) de [CEC+04]. Il opère sur les abstractions d'espaces d'état produites par `tina`, ou, à travers un outil de conversion, sur des graphes produits par d'autres outils tels que les outils $CADP$ [FGM+96].
`ndrio`, `ktzio`: il s'agit d'outils de conversion de formats de réseaux (`ndrio`) et de systèmes de transitions (`ktzio`).

B. Vérification par `selt`

La boîte à outils TINA fournit le model-checker `selt` qui permet de vérifier des propriétés spécifiques - par opposition aux propriétés générales d'accessibilité telles que le caractère borné, l'absence de blocage, la vivacité, ... - exprimées en logique temporelle linéaire.

1) Vérification via `selt`: La vérification via `selt` comporte deux phases :

- construire un automate de Büchi acceptant les mots qui ne satisfont pas la formule $SE-LTL$ à vérifier; cette phase est réalisée de facon transparente pour l'utilisateur en invoquant le logiciel `ltl2ba` développé au LIAFA par Paul Gastin et Denis Oddoux[GO01];
- construire la composition de la structure de Kripke obtenue depuis le graphe des classes et de l'automate de Büchi, et rechercher à la volée une composante fortement connexe contenant un état acceptant de l'automate de Büchi. Si aucune telle composante n'est trouvée, alors la formule est satisfaite, sinon elle définit un contre-exemple.

En cas de non-satisfaction d'une formule, `selt` peut fournir une séquence contre-exemple en clair ou sous un format exploitable par le simulateur de TINA, afin de pouvoir l'explorer pas à pas. Notons que, dans le cas d'un modèle temporisé, il faut au préalable associer à cette exécution un échéancier temporel. Celui-ci sera calculé de facon transparente par le module `plan`.

Dans certains cas - notamment pour les systèmes temporisés - la séquence contre-exemple d'une formule peut être très longue et donc difficilement exploitable par l'utilisateur. Afin d'en faciliter l'exploitation, `selt` peut aussi produire des contre-exemples sous forme compactée (symbolique). Le contre-exemple est alors présenté comme une suite de séquences, chacune imprimée comme une seule transition, ne matérialisant que les changements d'états "essentiels" pour la compréhension du problème (les changements d'états de l'automate de Büchi).

2) Les logiques LTL et SE–LTL : La logique LTL étend le calcul propositionnel en permettant l'expression de propriétés spécifiques sur les séquences d'exécution d'un système. $SE-LTL$ est une variante de LTL [CEC+04], qui permet de traiter de facon homogène des propositions d'états et des propositions de transitions. Les modèles pour la logique $SE-LTL$ sont des structures de Kripke étiquetées (ou SKE), aussi appelées systèmes de transitions de Kripke (ou KTS).

$SE-LTL$ est définie sur les ensembles AP *et* Σ d'une $\mathcal{SKE}$. p dénote une variable de AP et a une variable de Σ.

Les formules Φ de $SE-LTL$ sont définies par la grammaire suivante:

$$\Phi ::= p \mid a \mid \neg\Phi \mid \Phi \vee \Phi \mid \bigcirc \Phi \mid \Box \Phi \mid \Diamond \Phi \mid \Phi \, U \, \Phi$$

Leur sémantique est définie inductivement comme suit:

$\mathcal{SKE} \models \Phi$ ssi $\pi \models \Phi$ pour tout chemin $\pi \in \Pi(\mathcal{SKE})$
$\pi \models p$ ssi $p \in \nu(s_1)$ (où s_1 est le premier état de π)
$\pi \models a$ ssi $a \in \epsilon(a_1)$ (où a_1 est la première transition de π)
$\pi \models \neg\Phi$ ssi $non\pi \models \Phi$
$\pi \models \bigcirc \Phi$ ssi $\pi^2 \models \Phi$
$\pi \models \Box \Phi$ ssi $\forall i \geq 1, \pi^i \models \Phi$
$\pi \models \Diamond \Phi$ ssi $\exists i \geq 1, \pi^i \models \Phi$
$\pi \models \Phi_1 \ U \ \Phi_2$ ssi $\exists i \geq 1, \pi^i \models \Phi_2$ et $\forall 1 \geq j \geq i-1, \pi^j \models \Phi_1$

a) Quelques formules de $SE-LTL$: :

	(Pour tout chemin)
P	P vraie au départ du chemin (pour l'étape initiale),
$\bigcirc P$	P vraie dans l'étape suivante,
$\Box P$	P vraie tout le long du chemin,
$\Diamond P$	P vraie une fois au moins le long du chemin,
$P \ U \ Q$	Q sera vraie dans le futur et P est vraie jusqu'à cet instant,
$\Box \Diamond P$	P vraie infiniment souvent,
$\Box (P \Rightarrow \Diamond Q)$	Q "répond" à P.

3) Exemple d'application: Appliqué sur notre exemple, nous pouvons ainsi considérer par exemple les propriétés suivantes :

La formule (1) exprime une propriété de sûreté - ici la propriété d'exclusion mutuelle - sous la forme d'un invariant en assurant qu'au plus un `Node` processus peut-être en exclusion mutuelle.

```
[](cs_Node_1+cs_Node_2+cs_Node_3<=1)
```

La formule (2) exprime une propriété de vivacité conditionnelle - ici l'absence de famine - en assurant que tout processus en attente d'accéder à la section critique finira par l'atteindre.

```
[](wait_Node_1 => <> cs_Node_1)
```

Les formules (1) et (2) ont été évaluées positivement sur l'exemple du token ring décrit par le listing Fiacre (1).

Les propriétés temps réel exprimées dans des logiques temporelles temporisées et notamment celles d'accessibilité en temps borné sont vérifiées en utilisant la technique des observateurs permettant ainsi de se ramener à un problème d'accessibilité. Les travaux en cours dans le cadre du projet Quarteft visent à automatiser cette approche et de rendre transparente le recours aux observateurs.

Notons enfin que `selt` permet la déclaration d'opérateurs dérivés ainsi que la redéclaration des opérateurs existants; un petit nombre de commandes sont aussi fournies pour contrôler l'impression des résultats ou encore permettre l'utilisation de bibliothèques de formules. Il est ainsi particulièremet aisé de povoir définir ou réutiliser les bibliothèques de patterns de propriétés telles que celles définie dans [Pat].

V. Conclusion et perspectives

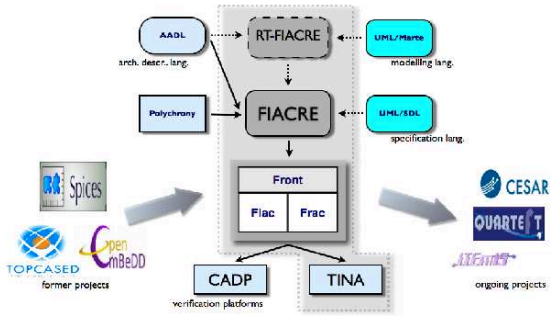

Fig. 8. Projets Fiacre

Cette présentation, nous a permis de présenter le langage Fiacre "de base" et sa première utilisation; en tant que langage pivot, pour la traduction du langage AADL. De nombreuses études de cas dans des projets tels que TOPCASED [Top], [FGP+08], [BBC+09], OpenEmbedd [CMB+09], [Ope] et SPICES [Spi] ont permis une première validation de la chaîne de traduction AADL-FIACRE. Parmi les études de cas considérées, citons:

- Dans le cadre des projets TOPCASED et CESAR, nous avons traîté des exemples de contrôle de portes d'avion. Nous avons en particulier modélisé les plateformes logicielle (threads, connexions) et matérielle (processeur, bus, périphériques). Toujours dans le cadre du projet TOPCASED, nous avons aussi traîté l'exemple de la commande automatique de stationnement de véhicule dans un parking [CBB+10].
- Dans le cadre du projet SPICES, nous avons étudié un protocole réseau dans le domaine de l'avionique [BBDZ+10]. Ce protocole permet au pilote de communiquer avec le sol des informations relatives à la vitesse, la destination, ... Du côté avionique, ce protocole s'exécute sur un calculateur IMA au sein d'une partition ARINC 653 [Air97]. La modélisation AADL décrit les composants logiciels et matériels.

Dans [PBF09], nous comparons la modélisation de protocoles de communication AADL en Bip [BBS06] et en FIACRE.

Le travail réalisé autour du langage Fiacre et de la chaîne de vérification asynchrone continuent au delà du projet Topcased. La figure 8 montre quelques projets dans lesquels le langage Fiacre est soit utilisé CESAR[CES] et ITEMIS[ITE] ou étendu Quarteft [QUA].

Les retours d'expérience autour du langage FIACRE ont montré qu'il était possible d'envisager des mécanismes de plus haut niveau que ceux initialement envisagés et faisant partie de la plupart des langages de modélisation ou des concepts temps-réel aujourd'hui utilisés. Un second retour a montré la difficulté d'exploiter les messages de vérification au niveau utilisateur. Ces deux aspects sont traités dans le cadre du projet FRAE Quarteft.

Quarteft a pour objectif de définir une chaîne de transformation qualifiable entre les langages métier de description de systèmes temps réel et les outils de model-checking. Quarteft prévoit la définition d'une extension spécifiquement temps réel du langage Fiacre permettant de faciliter le développement des chaînes de traduction de les optimiser et d'en faciliter la preuve.

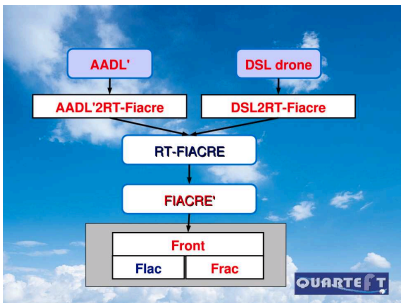

Fig. 9. Quarteft : RT-Fiacre

La figure 9 représente les extensions et les évolutions en cours pour Fiacre dans le cadre du projet Quarteft. La couche Fiacre-Etoile permet d'offrir des constructions dérivées – c'est à dire des constructions syntaxiques se ramenant par simple traduction à du Fiacre-Topcased - permettant d'écrire de facon directe des concepts tels que la généricité, ... La couche RT-Fiacre est spécifique temps réel et permet par exemple de décrire de facon directe des processus temps réel (périodiques, apériodiques, ...), automates de mode, ... Deux cas d'étude sont considérés dans la cadre de Quarteft, le langage AADL et le DSL Mauve dédié à la modélisation de systèmes de commande de drônes.

REFERENCES

[ABC94] A. Arnold, D. Begay, and P. Crubille. Construction and analysis of transition systems with MEC. Word Scientific, 1994.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology, July 1997.

[Air97] Airlines electronic engineering committee, 2551, Riva road, Annapolis, Maryland 21401. Avionics Application Software Standard Interface, ARINC Specification 653, january 1997.

[BBC+09] B. Berthomieu, Jean-Paul Bodeveix, Christelle Chaudet, Silvano Dal Zilio, Mamoun Filali, and F. Vernadat. Formal Verification of AADL Specifications in the Topcased Environment. In International Conference on Reliable Software Technologies - Ada-Europe, Brest, France, 08/06/09-12/06/09, number 5570 in lncs, pages 207–221, http://www.springerlink.com/, 2009. Springer-Verlag.

[BBDZ+10] B. Berthomieu, Jean-Paul Bodeveix, Silvano Dal Zilio, Pierre Dissaux, Mamoun Filali, Pierre Gaufillet, Sebastien Heim, and F. Vernadat. Formal Verification of AADL models with Fiacre and Tina (regular paper). In European Congress on Embedded Real-Time Software (ERTS), Toulouse, 19/05/2010-21/05/2010, page (electronic medium). SIA/3AF/SEE, 2010.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In SEFM, pages 3–12, 2006.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. on Software Engineering, 17(3):259–273, 1991.

[BRV04] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. International Journal of Production Research, 42-No 14, 2004.

[CBB+10] Tiago Correa, Leandro Becker, Jean-Paul Bodeveix, J.-M. Farines, Mamoun Filali, and F. Vernadat. Supporting the Design of Safety Critical Systems Using AADL (regular paper). In Oxford, 22/03/2010-26/03/2010, pages 331–336, http://www.computer.org, mars 2010. IEEE Computer Society.

[CEC+04] S. Chaki, M E, Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In 4th International Conference on Integrated Formal Methods (IFM'04) , Springer LNCS 2999, pages 128–147, apr 2004.

[CES] CESAR. Cost-efficient methods and processes for safety relevant embedded systems. http://www.cesarproject.eu/. Artemis Joint Undertaking.

[CMB+09] C.Andre, M.Belaunde, B.Berthomieu, C.Brunette, A.Canals, H.Garavel, S.Graf, F.Lang, V.Mahe, M.Nakhle, R.Schnekenburger, R.De Simone, J.P.Talpin, and F.Vernadat. Présentation des résultats du projet openembedd. In Journées NEPTUNE, pages 53–65, may 2009.

[FBF+07] Ricardo Bedin Franca, Jean-Paul Bodeveix, Mamoun Filali, Jean-Francois Rolland, David Chemouil, and Dave Thomas. The aadl behaviour annex – experiments and roadmap. In Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pages 377–382, Washington, DC, USA, 2007. IEEE Computer Society.

[FGM+96] J-C. Fernandez, H. Garavel, R. Mateescu, L. Mounier, and M. S ighireanu. Cadp, a protocol validation and verification toolbox. In 8th Conference Computer-Aided Verification, CAV'2001, Springer LNC S 1102, pages 437–440, jul 1996.