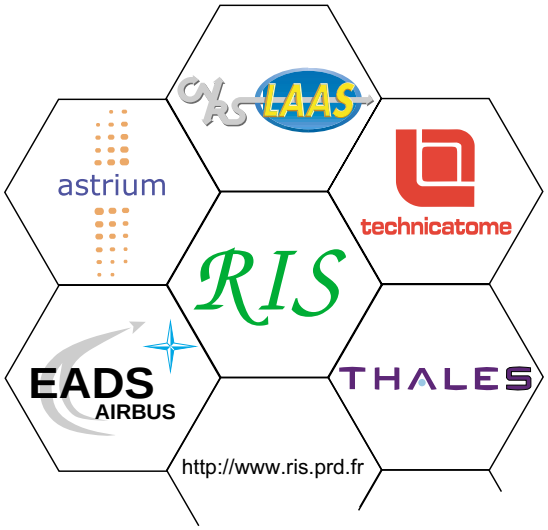

Réseau d'Ingénierie de la Sûreté de fonctionnement

Réunion n°4 du GT Logiciel Libre et Sûreté de Fonctionnement
14 janvier 2002 – LAAS-CNRS

Programme	
9 h 00-11 h 15	Introduction (approbation du compte-rendu précédent)
	Systèmes bord dans le domaine spatial, *Philippe David, Astrium*
	Wrapping pour le confinement d'erreurs, *Manuel Rodriguez-Moreno , LAAS-CNRS*
11 h 15-11 h 30	*Pause*
11 h 30-12 h30	Evolution des architectures embarquées dans l'avionique, *Serge Goiffon, Airbus*
12 h 30-13 h 30	*Déjeuner*
13 h 30-15 h 30	Mise à jour du planning des prochaines réunions et identification des contributions
	Méthodes formelles pour la vérification, *Béatrice Bérard, LSV*
	Synthèse du thème "Architecture et validation"

Participants :

Membres du GT :

B. Bérard (LSV), G. Bulsa (Astrium), P. Coupoux (Technicatome), Y. Crouzet (LAAS-CNRS), P. David (Astrium), Y. Garnier (SNCF), S. Goiffon (EADS Airbus), G. Mariano (INRETS), V. Nicomette (LAAS-CNRS), J-L. Terraillon (ESA), J-M. Tanneau (THALES), H. Waeselynck (LAAS-CNRS).

Absents ou excusés :

E. Conquet (ESA), Y. Paindaveine (Commission Européenne), I. Puaut (IRISA).

Invités :

J. Arlat (LAAS-CNRS), J-M. Astruc (SIEMENS Automotive), M. Rodriguez-Moreno (LAAS-CNRS).

1. Introduction

Le compte-rendu de la réunion du 20 septembre 2001 est approuvé avec quelques modifications. Il pourra être mis dans la partie publique du site web du *RIS*.

Jean-Marc Astruc, de Siemens Automotive, a été invité à participer à cette 4ème réunion du GT, pour un point de vue issu du domaine automobile.

2. Architecture et validation

Systèmes bord dans le domaine spatial, *Philippe David, Astrium*

Les systèmes bord sont constitués de trois types de sous-systèmes : liaison sol/bord, contrôle/commande de satellite, charge utile du satellite. Les logiciels libres y sont peu utilisés, sauf dans un cadre de R&D. Il y a cependant une dynamique forte vers les standards et l'interopérabilité qui pourrait jouer en faveur des logiciels libres.

Différents exemples d'architectures embarquées sont brièvement présentés :

- Architecture de type satellite. On a deux calculateurs SCU (spacecraft Controler Unit) qui peuvent fonctionner en redondance froide, chaude, ou tiède selon les phases de la mission. L'architecture est structurée autour de deux bus spécifiques (norme Mil-STD 1553B). Le trafic sur les bus est pré-déterminé, les différents sous-systèmes n'émettant que sur demande des calculateurs SCU.
- Architecture Columbus (station spatiale scientifique). Un hub Ethernet met en communication des calculateurs Sparc, et des portables baladeurs que l'équipage peut connecter. L'échange de données entre calculateurs est gérée par une couche intergicielle au dessus d'Ethernet – historiquement, d'abord un intergiciel spécifique, puis un intergiciel type VxWorks. Des MMU permettent le stockage de données en redondance. La partie critique contribuant à la survie de l'équipage (détection alarme feu, …) est réalisée selon une architecture classique de type satellite (voir plus haut), et physiquement ségréguée du reste de l'architecture. A noter que la charge utile de Columbus, c'est-à-dire les applications liées à la réalisation d'expériences scientifiques, est potentiellement riche en utilisation de logiciels libres.
- Architecture ATV (Automated Transfer Vehicle, chargé de ravitailler l'équipage de Columbus). Ce système est critique pour la survie de l'équipage. Il s'agit d'une architecture triplex en redondance chaude avec vote. Lorsque le rendez-vous véhicule/station est réussi, le système de l'ATV se connecte au hub Ethernet de Columbus. Les trois calculateurs étant identiques, il y a cependant un risque de manquer un rendez-vous suite à une défaillance de mode commun. Une unité spécifique (MSU = Monitoring and Safety unit) est donc chargée de la surveillance du déroulement du rendez-vous. Cette unité, conçue selon une architecture classique de type satellite, peut décider de déconnecter les calculateurs et gérer la retraite sûre du véhicule.

Dans ces architectures, le logiciel permet d'introduire de la souplesse : notamment, re-programmation en opération pour ajouter de nouvelles fonctions, ou récupérer des systèmes défaillants. Le logiciel embarqué se complexifie (ex : 48 KO de code embarqué dans les premiers satellites Spot, 4 MO dans l'ATV). Les exigences croissantes en termes d'autonomie des systèmes, qui devront être capables de s'adapter à des situation inconnues, font même envisager des solutions de type intelligence artificielle pour le futur.

Plusieurs points peuvent constituer des barrières à l'introduction de logiciels libres. Tout d'abord, les défaillances du logiciel ne sont actuellement pas prises en compte dans les analyses "Sûreté de Fonctionnement" : les AMDEC et arbres de fautes ne prennent en compte que le matériel, le logiciel étant supposé sans défaut. Il faudrait donc connaître les modes de défaillance des composants libres utilisés, et intégrer ces informations dans les analyses. Ensuite, l'existence d'un support technique est un critère crucial : vu les délais (24 mois jusqu'à livraison au client), l'effort nécessaire à l'appropriation d'un logiciel libre est prohibitif. De plus, la version embarquée doit être figée relativement tôt dans le cycle de vie, ce qui implique une exigence de stabilité du composant. On doit également mentionner des règles qualité (exigences de traçabilité, règles de programmation, …), et des exigences spécifiques au test (fonctions d'observation et de contrôle d'exécution) pouvant disqualifier des composants existants.

Plusieurs pistes sont cependant envisageables, quant à l'utilisation éventuelle de logiciels libres. L'utilisation de bus non spécifiques comme Ethernet, CAN, ouvre la porte à des protocoles libres. Au niveau système, on pourrait envisager de remplacer VxWorks par des logiciels libres compatibles POSIX, sous réserve qu'ils offrent les mêmes facilités du point de vue de la testabilité. Dans la mise en œuvre des aspects distribués, un principe général est d'uniformiser les communications distantes et locales : les intergiciels de type CORBA pourraient donc être de bons candidats. Au niveau applicatif, la nécessité d'effectuer des modifications en-ligne impose d'embarquer du code interprété : la technologie Java est ainsi une piste possible.

Les pistes ne se limitent pas au logiciel : la notion de "open hardware" est également intéressante. Un point dur est l'adaptation des processeurs aux besoins du test (notamment, nécessité d'observer l'état interne sans perturber le

fonctionnement du calculateur). Il est à noter que la frontière entre matériel et logiciel va devenir floue, avec l'évolution future vers les "system-on-chip". L'accès à des modèles VHDL faciliterait le co-design entre logiciel et matériel.

Cette discussion sur les architectures embarquées doit être replacée dans le contexte des préoccupations d'Astrium, qui est à la recherche de solutions pour améliorer son processus de développement. Les délais de livraison imposent de commencer le développement du logiciel trop tôt, en recouvrement avec les analyses système et le développement de bancs de test. Plusieurs versions d'un même système doivent éventuellement être gérées en parallèle, lorsqu'un nouveau projet démarre avant la fin du précédent. Actuellement, l'effort pour réduire la durée du processus de développement porte sur la validation de modèles amont (simulation) et la génération automatique de code.

Wrapping pour le confinement d'erreurs, *Manuel Rodriguez-Moreno, LAAS-CNRS*

Un *wrapper* est un composant logiciel qui entoure un composant cible. L'objectif peut être la mise en œuvre de politiques de sécurité (ex : Firewalls), la liaison de composants hétérogènes (ex : proxies de CORBA), ou encore la mise en œuvre de mécanismes de détection et de confinement d'erreurs. L'exposé porte sur cette dernière catégorie de wrappers. On propose un cadre générique d'empaquetage, visant à la fois les fautes de conception et les fautes matérielles : à partir de la documentation du composant cible (ex : interface POSIX), ou du résultat d'analyses de sûreté de fonctionnement, on élabore une modélisation formelle de propriétés à vérifier en-ligne ; ce modèle permet la génération automatique de wrappers de détection d'erreurs, constituant une version exécutable du modèle ; des mécanismes d'observation et de contrôle servent d'interface entre les wrappers et le composant cible. Dans cette approche, seuls les mécanismes d'observation et de contrôle dépendent de l'implémentation du composant cible. Les wrappers sont ré-utilisables pour tout exécutif offrant les fonctionnalités modélisées.

Le cadre générique a été instancié et expérimenté sur un exécutif temps-réel, une version antérieure du micro-noyau Chorus. Dans l'instanciation proposée, le *modèle formel* est constitué d'un ensemble de formules en logique temporelle (avec une notion explicite de temps discret en termes de nombre de ticks d'horloge). Les formules spécifient notamment des propriétés de sécurité-innocuité (une situation indésirable ne doit pas se produire) et de vivacité (une situation attendue doit se produire) se rattachant à un appel système particulier : la modélisation est donc guidée par le découpage fonctionnel des services offerts par l'exécutif temps-réel. Le *wrapper* correspondant à cette modélisation est constitué du code obtenu par compilation de chaque formule en langage C, et d'une bibliothèque de fonctions prédéfinies (le run-time checker). Le run-time checker peut être vu comme la machine virtuelle sur laquelle s'exécute l'ensemble des formules. Il inclut des fonctions correspondant aux opérateurs de la logique (ex : ALWAYS, NEXT) et des fonctions mettant en œuvre la détection d'erreur (ex : ASSERT, qui lève un signal d'erreur lorsqu'une condition passée en paramètre est évaluée à faux). Le wrapper (formules + run-time checker) interagit avec l'exécutif cible via des *mécanismes d'observation et de contrôle*. Ces mécanismes sont implémentés par instrumentation du source de l'exécutif selon une approche réflexive, définissant ainsi une "méta"-interface transparente au niveau applicatif. Trois catégories de mécanismes peuvent être distinguées : interception, introspection et intercession. Les mécanismes d'interception permettent la capture d'événements externes (ex : appel d'une fonction de l'API de l'exécutif) ou internes (ex : appel d'une fonction interne à l'exécutif, tick d'horloge) : ce sont eux qui vont activer de façon asynchrone l'exécution du code du wrapper. Lors de ses traitements, le wrapper effectue des appels aux mécanismes d'introspection, permettant l'observation de l'état interne de l'exécutif, et aux mécanismes d'intercession, permettant d'agir sur le comportement de l'exécutif. Le prototype réalisé a notamment permis de révéler une faute de conception affectant la primitive SetTimer de l'exécutif cible, et conduisant à violer plusieurs formules temporelles.

D'autres instanciations du cadre générique peuvent être envisagées. Par exemple, le modèle formel pourrait être une machine à états finis abstraite. L'observation n'est pas nécessairement conçue par instrumentation : pour un COTS sans visibilité du source, on peut envisager de négocier avec le constructeur une API étendue offrant des mécanismes d'observation. Cependant, ceci pose le problème de la vérification de ces mécanismes, dont la défaillance peut entraîner des déclenchements intempestifs du wrapper, ou son non-déclenchement. A noter que de telles défaillances peuvent également se produire suite à des problèmes affectant l'ensemble du système, par exemple lorsque l'horloge est défaillante.

Les travaux présentés dans cet exposé ont mis l'accent sur l'aspect détection d'erreur. Une poursuite prévue est d'étendre les traitements des wrappers à l'aspect recouvrement d'erreur. Par ailleurs, il est également envisagé d'appliquer cette approche à l'empaquetage de couches supérieures du système, en prenant notamment en compte les résultats de campagnes d'injection de fautes pour la caractérisation de supports exécutifs (cf. les travaux menés au LAAS sur les intergiciels CORBA, présentés lors de la réunion précédente).

Evolution des architectures embarquées dans l'avionique, *Serge Goiffon, Airbus*

Le logiciel prend une part croissante dans les avions modernes : par exemple, 12000 KO de code embarqué dans les A320-A340, à comparer avec les 4KO de code dans Concorde… Aux contraintes inhérentes à tout système embarqué (ressources physiques limitées, temps-réel, matériel dédié) viennent se superposer des contraintes propres à l'avionique, liées aux fonctions à réaliser, aux exigences de la normes DO178B, et à la durée de vie élevée des systèmes. Dans l'alternative "make or buy", la première stratégie a tout d'abord été privilégiée. Mais la tendance est maintenant d'utiliser des composants sur étagère, notamment pour des fonctions de criticité modérée (maintenance, communication), de complexité élevée rendant leur maîtrise difficile pour le constructeur avionique, ou encore pour des besoins d'interopérabilité. Ceci nécessite cependant une adaptation aux contraintes de certification et de pérennité.

L'intégration de composants matériels et logiciels du commerce s'effectue à différents niveaux, et suit l'évolution de la technologie : bus (ex : CAN, AFDX plutôt que bus spécifiques), mémoires (FLASH, RAM dynamique avec contrôle d'erreur), composants logiciels COTS (support exécutif, protocoles). Cette intégration croissante de composants accompagne une évolution des architectures vers une plus grande factorisation des ressources de calcul et de communication. Ainsi, le schéma LRU (Line Replaceable Unit), selon lequel les calculateurs étaient dédiés à des fonctions, évolue vers le nouveau schéma IMA (Integrated Modular Avionics), caractérisé par des calculateurs génériques dans lesquels on vient "plugger" des fonctions avioniques. Le réseau avion devient complexe, avec des calculateurs dédiés aux opérations d'entrée/sortie (IOM), des commutateurs pour la communication au sein de l'IMA, et des passerelles vers le monde extérieur (maintenance, mises à jour, télé-chargements).

Historiquement, la première intégration d'un support exécutif autre qu'un simple séquenceur "maison" date des A320-A340. Seuls les calculateurs dédiés aux fonctions de maintenance étaient concernés. L'utilisation du système VRTX répondait au besoin de gérer du multi-tâches, mais en restant dans un cadre mono-application. Un pas important a ensuite été franchi avec l'ATSU (Air Traffic Services Unit), un calculateur qui gère les communications sol/bord dans le FANS (Future Air Navigation System), et peut héberger des applications spécifiques aux compagnies aériennes. Dans ce cadre multi-applications, un COTS répondant à la norme POSIX a été choisi : LynxOS. Ceci a modifié la façon de concevoir et de valider une architecture logicielle. Une étroite coopération avec le fournisseur du COTS a été nécessaire, les rôles se partageant comme suit :

- Fournisseur COTS : support technique, incluant un support pour la certification, la maintenance, et des développements spécifiques pour ajouter des points d'observation dans le noyau.
- Intégrateur : choix et portage du COTS sur cible (compilation conditionnelle, édition de liens avec stubs, librairie C réduite), validation en vue de la certification (couverture structurelle du code par test, tests de robustesse "à la Ballista[1]" pour prendre en compte l'hébergement d'applications non sûres, analyse de mécanismes critiques tels que gestion mémoire et pile), intégration des applications.

A la suite de ce transfert mutuel d'expérience, le fournisseur du COTS a proposé un "kit de certification" (documentation issue de rétro-ingénierie, amélioration des tests, suivi d'anomalies) qui sera certainement utile pour des projets ultérieurs.

Côté Airbus, l'ATSU a également été l'occasion d'expérimenter l'utilisation de logiciels libres comme alternative aux COTS. Dans le cadre d'un projet de recherche, un portage de Linux sur l'ATSU a été réalisé. Ce portage n'a pas posé de problèmes techniques insurmontables, et les résultats sont suffisamment convaincants pour donner confiance dans la possibilité d'embarquer Linux pour des sous-systèmes de criticité modérée (niveau C), et ne présentant pas de contraintes temps-réel dures. L'infrastructure d'outils récupérables autour de Linux est appréciable. Cependant, un point dur reste le coût de la certification[2], et notamment le coût associé à la rétro-ingénierie du noyau.

L'ATSU s'inscrit encore dans le cadre d'un schéma d'architecture LRU mais, comme expliqué précédemment, l'avionique du futur sera basée sur un schéma IMA. La factorisation des ressources de calcul induit une architecture multi-partitions, avec une ségrégation de l'espace mémoire et du temps alloués à chaque partition. Le partage d'une ressource de calcul s'effectue selon un ordonnancement fixe pour les différentes partitions. Au sein d'une même partition, on a un ordonnancement multi-tâches. L'abandon du principe de déclenchement par interruption implique la délégation de certains traitements à des cartes d'entrée/sortie intelligentes. Le support exécutif doit répondre à la norme ARINC 653, qui définit les fonctions systèmes, ainsi que des mécanismes de communication intra et inter-partition. A titre d'exemple, sur l'A380, le support exécutif implémentant cette norme sera un COTS. Le partitionnement est destiné à permettre la certification séparée des fonctions (certification incrémentale), et à faciliter les évolutions technologiques. Il impose cependant des contraintes : la cohabitation de partitions de criticité différente entraîne une attraction du support exécutif vers la criticité la plus forte. L'utilisation de logiciels libres est alors difficilement envisageable – sauf à lancer un projet "logiciel libre" fédérant des acteurs du domaine avionique.

[1] Voir par exemple http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/edrc-ballista/www/
[2] Voir le rapport disponible sur le site privé du GT

Méthodes formelles pour la vérification, *Béatrice Bérard, LSV*

Ces méthodes permettent d'établir des propriétés sur des modèles formels d'un système étudié. Les modèles, et les propriétés à vérifier, sont spécifiés en se basant sur des documents/artefacts tels que : cahier des charges, spécification (abstraite/concrète), description d'algorithmes, code source, traces d'exécution (ex : analyse de fichiers log pour la détection d'intrusion). Comme exemple de formalisme de spécification, on peut mentionner les automates temporisés, intégrant une notion d'horloges évoluant de façon synchrone avec le temps, ou encore les automates hybrides linéaires, qui étendent les précédents en remplaçant les horloges par des variables linéaires (de pente constante). Des outils existent pour traiter ce type de modèles, parmi lesquels on peut citer les model-checkers Uppaal (universités d'Uppsala, Suède & d'Aalborg, Danemark) et HyTech (université de Berkeley, Californie). Il y a cependant des limitations liées à la combinatoire des modèles, voire des cas d'indécidabilité pour les automates hybrides linéaires. Après un bref panorama des projets menés au sein du LSV, l'exposé s'est focalisé sur cinq d'entre eux, fournissant ainsi des exemples concrets d'application de méthodes formelles.

Le protocole PGM (Pragmatic General Multicast) est un protocole multipoint de transmission de données. La source émet des données ainsi que des informations sur sa fenêtre de transmission (SPM, caractérisant la mémorisation de données émises en vue d'une éventuelle ré-émission). Lorsqu'un destinataire détecte une perte, il le signale par un message spécifique (NAK = negative acknowledgement), ce qui induit une ré-émission de la donnée. L'outil Uppaal a été utilisé pour vérifier un modèle simplifié de ce protocole par rapport à deux propriétés, liées respectivement à la détection et au recouvrement d'erreur. Dans ce modèle, on suppose que les messages NAK et SPM ne peuvent pas être perdus, et on se limite à des structures de données bornées (tableaux). La vérification a été conduite en instanciant le modèle avec des petites valeurs de paramètres (un ou deux destinataires, petites tailles de fenêtres temporelles), et cela a été suffisant pour mettre en évidence quelques problèmes pour les deux propriétés étudiées.

PNCSA (Protocole Normalisé de Connexion au Système d'Autorisation) est un protocole de carte bancaire, dont une modélisation par réseau de Petri a été proposée. L'analyse de ce protocole avec l'outil HyTech a permis de révéler un blocage qui n'avait pas été trouvé par l'analyse par graphe de couverture. L'utilisation de HyTech est rendue possible par une méthode de transformation du réseau de Petri en un automate hybride linéaire. Le calcul d'accessibilité repose sur une approximation des entiers par des réels, ce qui permet une résolution des contraintes plus efficace. Par contre, l'approximation impose de vérifier que le blocage trouvé correspond bien à un blocage du réseau de Pétri initial, ce qui s'est avéré être le cas pour le PNCSA.

L'ABR (Avalaible Bit Rate) est un module contrôlant la conformité du débit d'émission d'un utilisateur par rapport au débit autorisé par le réseau. Ce débit autorisé varie dynamiquement en fonction de la charge du réseau. L'algorithme "idéal" de calcul du débit autorisé met en jeu un échéancier de grande taille, et ne peut donc être implémenté. Un algorithme plus réaliste, permettant d'approximer ce calcul avec un échéancier borné, a été proposé par France Télécom. L'étude est partie d'une description de cet algorithme sous forme de pseudo-code. Une modélisation en automates temporisés a été réalisée, puis analysée avec HyTech. L'analyse n'a porté que sur une des propriétés requises : il s'agissait de montrer que l'approximation réalisée ne pénalise pas l'utilisateur, ce qui a pu être établi sur le modèle.

Le SRIC (Source Range Instrumentation Channel) est un logiciel critique pour la surveillance du flux neutronique d'un réacteur nucléaire, développé par Schneider Electric. En partant du cahier des charges, le LSV a réalisé une spécification algébrique en logique du 1er ordre, permettant d'utiliser l'assistant de preuve Larch prover. L'analyse manuelle en vue de la modélisation, et la vérification de propriétés de cohérence, ont mis en évidence des problèmes dans la rédaction du cahier des charges, dont une version révisée a été proposée par l'industriel. L'étude a ensuite porté sur la vérification formelle des 43 propriétés de sûreté énoncées dans le cahier des charges. Elle a permis de mettre en évidence des cas d'échec de preuve, dus à des hypothèses manquantes dans l'énoncé des propriétés.

Le projet VULCAIN a porté sur la vérification automatique d'automates programmables, en collaboration avec le LURPA (Laboratoire Universitaire de Recherche en Production Automatisée) et Alcatel. La norme IEC 61131-3 définit cinq notations pour spécifier de tels programmes : des langages graphiques et des pseudo-algorithmes type Pascal ou assembleur. A titre d'exemple, on montre un fragment de spécification d'une tourelle porte-outils, combinant des parties graphiques (notation SFC = Sequential Function Chart) et du pseudo-assembleur (notation IL = Instruction List). Cette étude de cas a été traitée avec le model-checker SMV. La spécification a été traduite sous la forme d'un automate, et les propriétés à vérifier ont été exprimées en logique temporelle linéaire. Les résultats ont mis en évidence des hypothèses manquantes dans l'énoncé des propriétés.

En conclusion, ces exemples illustrent la possibilité de vérifier formellement des propriétés de correction, généralement sur des portions critiques. Lorsque le modèle formel est construit à partir d'un document amont (ex :cahier des charges), ceci permet de consolider la spécification avant d'aller vers le code. Lorsque le modèle est une abstraction du code source (ex : analyse d'un logiciel libre), ceci permet une validation a posteriori. Pour un problème donné, une difficulté est le choix de l'outil de vérification le plus adéquat : des tâtonnements peuvent être nécessaires, même dans le cas de deux outils de même famille tels que Uppaal et HyTech.

3. Synthèse et discussion générale

Le thème “architecture et validation” a été l’occasion d’aborder différents points techniques, qui sont brièvement repris :

- Caractérisation des logiciels libres, et analyse des modes de défaillance → méthodes expérimentales basées sur l’injection de fautes. Comment capitaliser les résultats obtenus ? Structurer les retours d’expérience sous forme de base de données ?
- Rétro-ingénierie des logiciels libres→ éventuellement nécessaire pour les besoins de la certification, selon le niveau de criticité. Kit de certification ?
- Adaptation des LL au système cible par compilation conditionnelle et déconnexion de fonctionnalités indésirables par stubs → intérêt = déconnecter sans mutiler et sans créer une nouvelle souche du LL.
- Encapsulation → plusieurs exemples d’architectures à base de COTS ou LL ont été présentées. Le but de l’encapsulation peut être la pérennisation des systèmes, ou la mise en œuvre de la tolérance aux fautes.
- Validation des LL → cf. caractérisation + revues de code, AEEL et AMDEC, tests + calcul de WCET par analyse statique + vérification formelle de propriétés de correction sur des modèles.
- Architectures libres pour les applications présentant des caractéristiques récurrentes → quel “business model” ?

La discussion s’engage sur les processus de développement des logiciels libres. Y a-t-il un minimum requis pour les systèmes présentant des contraintes de sûreté de fonctionnement ? Y a-t-il une possibilité d’influer sur ce processus ? Il est souligné que, souvent, le développement suit une approche bottom-up. De plus, les contributeurs ne sont pas forcément prêts à accepter des recommandations/contraintes de développement (“celui qui code a raison”). Un point positif est cependant l’évolution des jeunes informaticiens qui, de par leur formation, sont sensibilisés au aspects Génie Logiciel. Il sera intéressant d’avoir le point de vue des contributeurs invités lors de la prochaine réunion.

4. Prochaines réunions

La réunion 5 portera sur les processus de création et des évolutions des LL. A cette occasion, plusieurs intervenants ayant contribué à la création – ou assurant du support sur – des logiciels libres seront invités : Franco Gasperoni (ACT Europe, pour GNAT Ada 95), Brian Bray (Minoru Development Corporation, logiciels libres dans le domaine de la santé), Matthieu Herrb (LAAS-CNRS, maintenance de OpenBSD). Georges Mariano (INRETS) rapportera sur une expérience de développement d’outils libres autour de la méthode B. J-M. Tanneau (THALES) présentera brièvement le rapport issu du Groupe RNTL sur les logiciels libres. Enfin, Jean-Marc Astruc, de Siemens Automotive, sera ré-invité pour se positionner par rapport à la thématique du GT RIS.

Le plan des deux réunions suivantes est :

- Réunion 6: aspects juridiques, certification, protection intellectuelle, sécurité informatique. Chacun devra contacter son service juridique, dans l’optique d’une table ronde avec des représentants des différents partenaires. Eventuellement, invitation de Bernard Lang de l’AFUL (Association Francophone des Utilisateurs de Linux et des Logiciels Libres). Pour les aspects certification, un contact sera repris avec le CEAT. Enfin, le thème “sécurité informatique” pourrait être introduit par Vincent Nicomette (LAAS-CNRS).
- Réunion 7: synthèse, préparation de la logique jusqu’à la fin du GT. Cette réunion étant prévue peu avant les vacances d’été, il serait souhaitable d’initier plus tôt la réflexion sur le document final. Philippe David et Hélène Waeselynck feront une première proposition lors de la réunion 5.