INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Projet EP-ATR

Environnement de programmation d'applications temps réel

Rennes

THÈME 1C

Table des matières

1 Composition de l'équipe

Responsable scientifique

Paul Le Guernic [DR Inria]

Assistante de projet

Huguette Béchu [TR Inria]

Personnel Inria

Albert Benveniste [DR, projet Sigma 2]
Patricia Bournai [IR, Atelier, à mi-temps dans le projet]
Thierry Gautier [CR]
Hervé Marchand [CR]
Luc-Michel Sévère [ingénieur associé, à partir du 1er octobre 2001]
Jean-Pierre Talpin [CR]

Personnel CNRS

Loïc Besnard [IR, Atelier]

Personnel Université de Rennes 1

Bernard Houssais [maître de conférences]
Sophie Pinchinat [maître de conférences, détachée en tant que CR Inria]
Christophe Wolinski [maître de conférences, jusqu'au 31 août 2001]

Chercheurs doctorants

Abdoulaye Elhadji Gamatié [bourse Inria]
Fernando Jiménez Fraustro [bourse du gouvernement mexicain, jusqu'au 31 mars 2001]
Mickaël Kerbœuf [bourse MENRT]
Sylvain Kerjean [bourse MENRT]
Pierre Le Maigat [bourse MENRT]
Mirabelle Nebut [bourse MENRT jusqu'au 31 août 2001, demi poste Ater à l'Université de Rennes 1 à partir du 1er septembre 2001]
Stéphane Riedweg [bourse Inria-région]
Laurent Vibert [normalien]
Yunming Wang [bourse CIES, jusqu'au 31 mars 2001]

2 Présentation et objectifs généraux

Mots clés : EP-ATR, système enfoui, temps réel, conception synchrone.

Le projet EP-ATR conduit des travaux à la fois théoriques, méthodologiques, algorithmiques et d'expérimentation sur le thème de la conception de systèmes (ou d'applications) enfouis temps réel ; ces travaux sont basés sur une approche synchrone de la description de processus.

2.1 Contexte des études

Mots clés : système enfoui, système temps réel, système critique, certification, cycle en V, transformation de programme, conception conjointe, composants, vérification, méthode

formelle.

Les produits informatiques *enfouis,* le plus souvent avec des contraintes *temps réel* fortes, dans des systèmes industriels de grande envergure comme les centrales nucléaires, les systèmes de contrôle aérien, les télécommunications, ou dans des systèmes de taille plus modeste (avions, automobiles...), mais aussi de petite taille comme les processeurs ou contrôleurs divers utilisés dans des produits de grande diffusion, ont pour caractéristique commune de devoir fonctionner selon un mode de coopération permanente avec un environnement. Un système enfoui peut ainsi influencer sérieusement le comportement de cet environnement, non seulement par son activité mais aussi par son inactivité : ainsi le dysfonctionnement d'un système par exemple de commande de vol peut entraîner un accident d'avion ; celui d'un système de régulation thermique, la perte d'une entreprise avicole, par exemple. En fonction de la gravité des conséquences prévisibles d'un dysfonctionnement du composant informatique sur les plans économique, humain, ou social, le développement des applications considérées est l'objet de procédures plus ou moins lourdes, telles que *certification* pour les *systèmes critiques,* visant à la réduction des risques d'erreurs ou de défaillances du système.

Traditionnellement le développement de ces applications s'appuie sur un *cycle en V* qui de l'amont vers l'aval, en partant d'un cahier des charges, passe par l'établissement de spécifications aussi complètes et consistantes que possible, sur lesquelles s'appuie la conception globale puis détaillée à partir de laquelle est produit le codage de l'application. Ce processus de conception s'appuye à des niveaux et à des degrés divers sur des techniques de *transformation de programmes*, sur des techniques de *conception conjointe*, sur l'utilisation de *composants* matériels ou logiciels. La confiance dans le produit réalisé repose sur le respect de procédures codifiées incluant un chemin de *vérification,* inverse du chemin de conception, allant du test unitaire au test système. Les *méthodes formelles,* s'appuyant sur des modèles mathématiques correctement définis, trouvent aujourd'hui une place grandissante dans ces méthodologies et sont d'ores et déjà admises, voire recommandées, comme complément dans les méthodes traditionnelles, y compris dans des documents normatifs.

2.2 Objectif général du projet

Mots clés : modèles théoriques, méthodologie de conception, prototypes logiciels, applications, traitement temps réel du signal, télécommunication, avionique, automobile, temps réel, programmation synchrone, génération de code enfoui, génération de code distribué, architecture hétérogène, circuits.

C'est dans ce cadre que s'inscrivent les activités conduites par le projet EP-ATR : elles visent à proposer des *modèles théoriques*, des enrichissements du contexte *méthodologique* traditionnel (cycle en V) prenant en compte ces modèles, et enfin des logiciels (à l'état de *prototype avancé*) permettant de mettre en œuvre ces méthodes de conception nouvelles et de conduire des expérimentations des solutions proposées. Ayant débuté dans le domaine du traitement temps réel du signal en télécommunication (avec le soutien du Cnet et de la DGA, nos travaux ont trouvé de nouveaux contextes d'applications, en particulier par des collaborations suivies avec différents projets de l'Inria (robotique, image), avec EDF, puis dans le domaine de l'avionique avec nos partenaires des projets Esprit Sacres, Syrf, puis IST Safeair, dans le domaine des

télécommunications, là encore en collaboration avec des projets de l'Inria, avec Alcatel et le Cnet, et dans le domaine de l'automobile, avec les industriels impliqués dans le projet AEE, et là aussi différents projets de l'Inria et d'autres partenaires universitaires.

En même temps que se sont élargis les domaines d'applications, les problèmes que nous considérons se sont étendus d'une part en amont vers la spécification de systèmes temps réel, d'autre part en aval vers la génération de code enfoui, éventuellement sous la forme de *code distribué* sur des architectures hétérogènes et au delà, vers la prise en compte de composants matériels, en synthèse et en modélisation.

Ces travaux, qui concernent ainsi aujourd'hui le développement des applications temps réel depuis leur spécification jusqu'à leur mise en œuvre matérielle, reposent sur une modélisation homogène des niveaux de description rencontrés au cours de la conception : cette approche, fondée sur un modèle mathématique de la *programmation synchrone,* permet de mettre formellement en relation différentes versions de l'application, réduisant en cela les risques liés à des ruptures dans le cycle de leur conception.

2.3 Conception synchrone

Mots clés : conception synchrone, Signal, contraintes, non-déterminisme, système réactif.

L'idée assez simple du modèle de *conception synchrone* est de considérer qu'un programme plongé dans un environnement (constitué de procédés, d'opérateurs, d'autres programmes), interagit significativement avec cet environnement au travers d'un ensemble fini de supports de communication à des instants formant un ensemble dénombrable partiellement ordonné. À chaque instant, plusieurs actions (messages reçus, émis, calculs...) peuvent être effectuées ; elles sont alors simultanées et possèdent un même indice temporel. L'écoulement du temps résulte des successions de ces communications, mais aussi de changements explicites décrits dans l'algorithme que le programme met en œuvre (par exemple pour une équation $y_t = f(x_{t-1})$, la sortie y sera simultanée à chaque occurrence suivante de l'entrée x). Selon les formalismes, les sorties calculées sont, sauf changement explicite dans le programme, soit simultanées aux entrées qui ont provoqué leur calcul comme dans les langages Esterel, Lustre ou Signal, soit produites à l'instant suivant comme dans Statemate ou VHDL par exemple.

Le langage Signal [8], conçu et mis en œuvre dans le projet, est de plus, comme Lustre, construit selon une approche flot de données faisant d'un programme un système d'équations. À la différence de ce qui se passe en Lustre, le système d'équations d'un programme Signal décrit par une *relation* des *contraintes* entre les signaux d'entrée et de sortie du système ; ceci permet d'utiliser Signal pour donner des spécifications partielles ou décrire des comportements non déterministes. Cette banalisation (néanmoins partielle) des entrées et des sorties permet en outre la spécification et la programmation de systèmes *réactifs* [HP85] mais aussi *«pro-actifs»* (pouvant par exemple contraindre leurs entrées).

[HP85] D. Harel, A. Pnueli, « On the development of reactive systems », *in: Logics and models of concurrent systems*, K. Apt (éditeur), NATO Advanced Study Institute on Logics and Models for Verification of Concurrent Systems, Springer Verlag, p. 477–498, New-York, 1985.

2.4 Thèmes de recherche

Mots clés : Signal, conception synchrone, vérification, transformation de programme, communication synchrone/asynchrone, architecture hétérogène, composant matériel, format commun, programmation synchrone, DC+, Statecharts, automatismes industriels, système dynamique polynomial, synthèse de contrôleur, BDD.

S'appuyant sur une expression en Signal des différentes versions d'un système en cours de conception, la méthodologie que nous développons consiste en l'application d'une série de transformations de descriptions respectant le schéma suivant :

- spécification, conception et vérification de l'application indépendamment de l'architecture cible, grâce à l'hypothèse de synchronisme et au non-déterminisme qui permet d'une part de fournir un modèle du comportement de l'environnement, d'autre part de donner des spécifications partielles ;
- affinement progressif vers l'implémentation (conception détaillée incluant le partitionnement) guidé par des simulations/vérifications à différents niveaux ; à cette étape, la cible est une architecture logicielle pouvant être vérifiée et évaluée ; cette cible peut comporter des spécifications de composants prédéfinis ;
- implantation effective du schéma d'exécution obtenu sur des architectures y compris asynchrones et distribuées, en relâchant au besoin l'hypothèse de synchronisme (tout en restant dans le cadre du modèle synchrone), et en garantissant une implémentation correcte ;
- génération de code exécutable, de descriptions de composants matériels ou de composants hybrides matériels/logiciels.

Nos activités de recherche concernent les différentes étapes de ce schéma de conception. Comme certaines de ces activités concernent plusieurs de ces étapes, nous distinguons ici, pour une meilleure lisibilité, les thèmes suivants : description d'applications, étude des propriétés des processus synchrones, méthodes et outils pour la conception d'architectures de mise en œuvre, développements et expérimentations.

2.4.1 Description d'applications

La description d'une application, tant au niveau de la spécification que de la conception, suppose l'existence d'un langage suffisamment expressif pour les classes d'applications visées ; la réduction des ruptures dans le cycle de conception est favorisée par le support logiciel des traitements dans une sémantique homogène. Les études portant sur l'augmentation du pouvoir d'expression du langage Signal ont pour but d'étendre le domaine d'application des techniques synchrones, soit par la définition de relations sémantiques entre les formalismes synchrones et d'autres formalismes, soit par l'adjonction de nouvelles primitives ou de nouveaux concepts dans le langage lui-même.

- Avec la société TNI (voir section 7.5) qui commercialise Sildex, un environnement de programmation fondé sur Signal, nous avons travaillé à la définition d'une *nouvelle version (V4)* dont l'une des principales extensions porte sur la modularité ; d'autres extensions ont également été définies et de nouvelles techniques de compilation sont implémentées (voir section 6.1) ;

– Entreprises initialement dans le cadre d'un ancien projet avec les Laboratoires de Marcoussis, les études sur les liens entre le paradigme objet et le modèle synchrone trouvent un nouvel élan dans les travaux auxquels nous participons sur la conception de BDL et le lien avec UML ; BDL est un formalisme reposant sur le modèle synchrone pour la spécification de comportements d'objets (voir section 6.2). Nous travaillons d'autre part à l'unification des paradigmes synchrone et objet dans un même cadre formel (voir section 6.3).

Le *format commun* de la programmation synchrone (DC+) résulte de travaux menés depuis plusieurs années, tout d'abord par les équipes françaises dans lesquelles ont été conçus les langages synchrones Esterel, Lustre et Signal au sein du groupement C2A, puis dans le cadre d'un projet européen Eureka (Synchron), et enfin dans le cadre des projets européens LTR Syrf et R&D Sacres [Sac97]. La définition de ce format a été décidée en vue de favoriser le partage de logiciels issus de la communauté synchrone, mais aussi de permettre une large ouverture vers d'autres formalismes en amont, et d'autres outils, latéralement et en aval. Il est aujourd'hui intégré à l'environnement Signal (voir section 5.1).

Le projet européen Sacres a eu précisément pour objectif de développer un environnement de conception multi-formalisme synchrone à destination des applications critiques enfouies. Il se poursuit aujourd'hui sous une autre forme dans le cadre du projet européen IST SafeAir (voir section 7.3). Une autre étude basée sur le multi-formalisme concerne la modélisation et traduction des langages de programmation d'automatismes industriels de la norme IEC 1131 (voir section 6.4).

2.4.2 Étude des propriétés des processus synchrones

Un programme Signal décrit le comportement d'un système de transitions dont divers formalismes permettent d'étudier les propriétés.

Jusqu'à une période récente, nos travaux sur ce thème ont porté essentiellement sur une modélisation des comportements par des *systèmes dynamiques polynomiaux* sur les entiers modulo 3 (permettant le codage des booléens et de l'absence d'occurrence d'un signal à un instant donné). Un système de calcul symbolique (Sigali) a été développé dans ce cadre (voir section 5.2). C'est dans ce système que sont maintenant étudiées des techniques de *synthèse de contrôleurs,* techniques utilisables pour affiner des spécifications, pour aider à la conduite de postes de commandes, voire à la génération automatique de tests (voir section 6.6).

Les systèmes dynamiques permettent d'analyser (partiellement) les trajectoires des processus dans un ensemble d'états résultant de la combinaison de variables du programme Signal. Cet ensemble lui-même est l'objet, dès la compilation, de calculs qui reposent sur une représentation par hiérarchie de BDD (*Binary Decision Diagrams* — représentation de formules booléennes) : ceci est appelé *calcul d'horloges* [1]. Ces calculs sont utilisés en particulier pour effectuer des *transformations de programmes* en vue de vérification, de distribution et de génération de code.

Des approches complémentaires pour la vérification de propriétés sont fournies par les techniques d'*interprétation abstraite,* permettant la prise en compte de signaux numériques,

[Sac97] SACRES CONSORTIUM, « The Declarative Code DC+, version 1.4 », novembre 1997, Esprit project EP 20897: Sacres, `ftp://ftp.irisa.fr/local/signal/publis/research_reports/dc+.ps.gz`.

par exemple (voir section 6.5).

2.4.3 Méthodes et outils pour la conception d'architectures de mise en œuvre

La définition de l'architecture supportant l'application temps réel peut comporter différentes classes de composants comme des processeurs standards, des DSP, des Asics, etc., et c'est donc dans la perspective de mise en œuvre sur de telles cibles hétérogènes que nous inscrivons aussi bien nos études sur la conception détaillée et le codage que des activités relevant de la conception conjointe matériel/logiciel.

Le problème du partitionnement d'un programme flot de données synchrone est abordé en s'attachant aux propriétés structurelles du graphe de l'application ; supposant donnée une répartition du code sur une architecture cible (cette répartition peut résulter d'algorithmes d'optimisation ou être explicitée comme spécification non fonctionnelle, par exemple pour des raisons de traçabilité), nous nous proposons de produire, par des transformations locales à chaque composant de cette architecture, à la fois le code pour ces composants et les protocoles de communication à mettre en œuvre (voir section 6.7). Cette mise en œuvre peut être «Globalement Asynchrone Localement Synchrone» (Gals). Le cadre théorique de nos recherches permet de caractériser des mises en œuvre synchrones ou partiellement désynchronisées en terme de préservations de propriétés de flots, d'ordres partiels, de taille mémoire, par rapport au programme initial.

Pour la génération de composants matériels ou de code exécutable sur un processeur programmable, l'approche que nous adoptons est là encore celle de la transformation de programmes conduisant à une expression en Signal aussi proche que possible de la structure du code cible (C ou VHDL actuellement). Cette approche diminue les risques d'incorrection liés au passage à un autre langage.

Elle est accompagnée d'études sur la modélisation synchrone de composants matériels en vue du prototypage rapide de circuits spécifiques, qui participent à l'activité grandissante du projet sur le thème de la conception conjointe matériel/logiciel (voir section 6.8).

Dans l'approche synchrone, les durées d'exécution ne sont pas directement prises en compte. La prise en compte de ces durées, nécessaire à la vérification de la correction de la mise en œuvre en regard des contraintes temps réel, est effectuée en considérant une interprétation qui, à un programme synchrone et une architecture donnée, associe un programme synchrone sur nombres entiers ; l'image du programme initial modélise les durées d'exécution de la mise en œuvre. On peut alors simuler le programme image (voir section 6.1) ou chercher à utiliser des techniques analytiques, comme par exemple celles qu'offre l'algèbre MaxPlus (voir section 6.9).

2.4.4 Développements et expérimentations

Les travaux théoriques conduits dans le projet aboutissent à des prototypes mettant en œuvre les algorithmes conçus au cours de ces études. Nous nous efforçons de conserver et diffuser ce savoir-faire du projet par une intégration dans un environnement de conception construit autour du langage Signal. Une section est consacrée à la présentation de ce logiciel (section 5.1).

Des expérimentations sont conduites pour valider les algorithmes en relation avec des industriels. Elles concernent également la diffusion des principes synchrones par des développements donnant à des formalismes divers une traduction en Signal. Enfin l'étude d'applications en coopération avec des industriels ou des projets académiques est une source importante des problèmes nouveaux que traite le projet EP-ATR.

2.5 Problèmes ouverts et perspectives

Mots clés : système hybride, génération de test.

Le développement d'un système enfoui temps réel, tel que décrit dans ces pages, prend des processus discrets pour modèles des procédés physiques de l'environnement et suppose donc l'étude algorithmique (par exemple de lois de commande) effectuée. Or la validation d'un système complet doit prendre en compte, soit par modèle, soit dans des maquettes matérielles (et souvent les deux), les fonctions continues dirigeant le comportement des procédés. Si des travaux d'interfaçage entre des langages synchrones et des outils dédiés au continu tels que Matlab et Simulink ont été entrepris, il reste beaucoup à faire pour obtenir une réelle *intégration des formalismes discrets et continus* dans des systèmes hybrides.

Un obstacle sérieux à la mise en œuvre correcte par construction des systèmes temps réel reste le lien entre temps physique et temps logique et entre différents grains de temps discret, en particulier pour les points suivants :

- Prise en compte des événements fournis par l'environnement le plus souvent asynchrone au programme synchrone : les études sur les modèles de communication *synchrone/asynchrone* sont à compléter.
- Si nous savons définir correctement des procédures de raffinement efficaces portant sur l'espace (adjonction/suppression de variables), en ce qui concerne le temps, les études que nous avons entreprises tant par le biais des transformations affines, que dans les cadres du (re)timing monocadencé ou de l'utilisation de l'algèbre MaxPlus, restent encore de portée trop limitée pour satisfaire les besoins de transformation visant à éliminer les systèmes d'exploitation dans les systèmes critiques enfouis.

Pour aller vers la vérification complète des propriétés d'un système, il est nécessaire de prendre en compte des domaines plus riches que les booléens. Nous menons des études en ce sens visant à compléter dans un premier temps le calcul d'horloges par des techniques provenant des résolutions de contraintes sur domaines finis ; pour aller plus loin, il deviendra nécessaire d'adapter des techniques de démonstration automatique.

Même si les techniques formelles progressent, tant dans les esprits de leurs utilisateurs potentiels que dans les performances des algorithmes mis en œuvre, la simulation et le test resteront des activités nécessaires à la validation d'un système. C'est pourquoi nous envisageons de participer à des études sur la génération automatique de tests, principalement d'intégration, qui posent des problèmes non résolus. Ces études pourront se faire en corrélation étroite avec les méthodes de synthèse automatique de contrôleurs actuellement développées dans le projet.

En supposant un système correct vis-à-vis de spécifications de comportements nominaux, l'étude du comportement du système en présence de défaillances reste un sujet majeur pour les entreprises mettant en œuvre des systèmes critiques. Les études théoriques (qui bien sûr ne

peuvent concerner les seuls informaticiens) doivent être développées.

Ces activités et d'autres thèmes de recherche connexes seront poursuivis dans les nouveaux projets de recherche issus du renouvellement des projets du thème 1c à l'Irisa. Si le projet EP-ATR s'est arrêté en effet courant 2001, la plupart des membres du projet se retrouvent dans les nouveaux projets suivants :

- projet Espresso : P. Le Guernic, Th. Gautier, L.-M. Sévère, J.-P. Talpin, L. Besnard, B. Houssais, A. Gamatié, M. Kerbœuf, S. Kerjean, M. Nebut, L. Vibert ;
- projet S4 : A. Benveniste, S. Pinchinat, S. Riedweg ;
- projet VerTeCs : H. Marchand.

Le projet Espresso notamment, sous la responsabilité de J.-P. Talpin, assurera le développement et la diffusion de la plateforme Polychrony développée autour de Signal.

3 Fondements scientifiques

3.1 Spécification et programmation synchrone

Mots clés : conception synchrone, sémantique synchrone, programmation synchrone, Signal, transformation de programme.

Résumé : *Nous définissons la sémantique fonctionnelle d'un programme synchrone comme un ensemble de suites de valuations des variables de ce programme dans un domaine de valeurs complété par une représentation de l'absence d'occurrence d'une variable. Les opérateurs du langage Signal décrivent des relations sur de telles suites. Le compilateur de Signal est un outil formel capable de synthétiser la synchronisation globale d'un programme et l'ordonnancement de ses calculs. De nombreuses phases de compilation s'expriment par des transformations de programmes définies par des homomorphismes de programmes Signal.*

Les usages différents des termes *synchrone* et *asynchrone* selon les contextes dans lesquels ils apparaissent font qu'il nous semble nécessaire de préciser, autant que possible, ce qui constitue l'essence même du paradigme synchrone [BB91,BCLH93,Ber89,Hal93]. Les points suivants apparaissent comme caractéristiques de l'approche synchrone :

- Le comportement des programmes synchrones progresse via une suite infinie d'actions composées.
- Chaque a2ction composée est constituée d'un nombre borné d'actions élémentaires, pour les langages Esterel, Lustre et Signal.

[BB91] A. Benveniste, G. Berry, « The Synchronous Approach to Reactive and Real-Time Systems », *Proceedings of the IEEE 79*, 9, Septembre 1991, p. 1270–1282.

[BCLH93] A. Benveniste, P. Caspi, P. Le Guernic, N. Halbwachs, « Data-Flow Synchronous Languages », *in : Lecture Notes in Computer Science 803, Proc. of the REX School/Symposium, Noordwijkerhout, Netherlands*, J. W. de Bakker, W. de Roever, G. Rozenberg (éditeurs), 803, Springer–Verlag, p. 1–45, Juin 1993.

[Ber89] G. Berry, « Real-Time Programming: special purpose or general purpose languages », *in : Information processing 89*, G. X. Ritter (éditeur), Elsevier Science Publisher B.V., 1989.

[Hal93] N. Halbwachs, *Synchronous programming of reactive systems*, Kluwer, 1993.

– À l'intérieur d'une action composée, les décisions peuvent être prises sur la base de l'*absence* de certains événements, comme il apparaît sur l'exemple des trois instructions suivantes, issues respectivement de Esterel, Lustre, et Signal :

`present x else 'stat'`	l'action `stat` est exécutée en l'absence du signal `x`.
`y = current x`	en l'absence d'occurrence du signal `x`, `y` prend la valeur de la dernière occurrence de `x`.
`y := x default z`	en l'absence d'occurrence des signaux `x` et `z`, `y` est absent, sinon en l'absence d'occurrence du signal `x`, `y` prend la valeur de `z`.

– *Lorsqu'elle est définie,* la composition parallèle de deux programmes s'exprime toujours par la composition des couples d'actions composées qui leur sont respectivement associées, elle-même obtenue par la conjonction de leurs actions élémentaires respectives.

Pour ce qui concerne la spécification de programmes (ou de propriétés), la règle ci-dessus est clairement la bonne définition de la composition parallèle.

S'il s'agit également de programmation, la nécessité que cette définition soit compatible avec une sémantique opérationnelle complique largement la condition «lorsqu'elle est définie».

3.1.1 Sémantique synchrone

La sémantique fonctionnelle d'un programme Signal est décrite comme l'ensemble des suites admissibles de valuations des variables de ce programme dans un domaine de valeurs complété par la notation d'absence d'occurrence [4, 9].

Considérant :

– A, un ensemble de variables,

– D, un domaine de valeurs incluant les booléens,

– $\perp$, n'appartenant pas à D,

une *trace* T sur $A_1 \subset A$ est une fonction $T : \mathbf{N} \to A_1 \to (D \cup \{\perp\})$.

Pour tout $k \in \mathbf{N}$, un événement sur A_1 est une valuation $T(k)$: une trace est une suite d'événements. On appelle événement nul l'événement dans lequel chaque valeur est égale à $\perp$.

Pour toute trace T, il existe une trace F unique appelée *flot*, notée $\mathit{flot}(T)$, dont la sous-suite des événements non nuls est égale à celle de T et initiale dans F. La projection $\pi_{A_2}(F)$ sur un sous-ensemble $A_2 \subset A_1$ d'un flot F, défini sur A_1, est le flot $\mathit{flot}(T)$ pour T trace des restrictions des événements de F à A_2.

Un *processus* P sur A_1 est alors défini comme un ensemble de flots sur A_1. L'union de l'ensemble des processus sur $A_i \subset A$ est noté $\mathcal{P}_A$.

Étant donné P_1 et P_2 deux processus définis respectivement sur des ensembles de variables A_1 et A_2, leur composition, notée $P_1|P_2$, est l'ensemble des flots F définis sur $A_1 \cup A_2$ tels que $\pi_{A_1}(F) \in P_1$ et $\pi_{A_2}(F) \in P_2$. La composition de deux processus P_1 et P_2 est ainsi définie par l'ensemble de tous les flots respectant, en particulier sur les variables communes, l'ensemble des contraintes imposées respectivement par P_1 et P_2.

Soit $\mathbf{1}_{\mathcal{P}}$ le processus ayant comme seul élément la trace (unique) sur l'ensemble vide de

variables. On montre alors que $(\mathcal{P}_A, \mathbf{1}_\mathcal{P}, |)$ est un monoïde commutatif (cette propriété rend possibles les transformations de programmes mentionnées en 3.1.2) :

$$(P_1|P_2)|P_3 = P_1|(P_2|P_3)$$
$$P_1|P_2 = P_2|P_1$$
$$P|\mathbf{1}_\mathcal{P} = P$$

De plus, pour tout $A_1 \subset A$, les processus $\mathbf{0}_\mathcal{P}$ définis par l'ensemble vide de flots sur A_1 sont absorbants :

$$P|\mathbf{0}_\mathcal{P} = \mathbf{0}_\mathcal{P}$$

Enfin, l'opérateur de composition est idempotent (ceci autorise la réplication de processus) :

$$P|P = P$$

Par ailleurs, si P est un processus sur A_1 et Q un processus sur A_2 inclus dans A_1, on a :

$$P|Q = P$$

si et seulement si tout flot de la projection de P sur A_2 est un flot de Q (Q est moins contraint que P).

3.1.2 Langage Signal

Un programme Signal [8] spécifie un système temps réel au moyen d'un système d'équations dynamiques sur des *signaux*. Les systèmes d'équations peuvent être organisés de manière hiérarchique en sous-systèmes (ou *processus*). Un signal est une suite de valeurs à laquelle est associée une *horloge,* qui définit l'ensemble discret des instants auxquels ces valeurs sont présentes (différentes de $\perp$). Les horloges ne sont pas nécessairement reliées entre elles par des fréquences d'échantillonnage fixes : elles peuvent avoir des occurrences dépendant de données locales ou d'événements externes (comme des interruptions, par exemple).

Le langage Signal est construit sur un petit nombre de primitives, dont la sémantique est donnée en termes de processus tels que décrits ci-dessus. Les autres opérateurs de Signal sont définis en terme de ces primitives, et le langage complet fournit les constructions adéquates pour une programmation modulaire.

Pour un flot F, une variable X et un entier t on note, lorsqu'il n'y a pas de confusion possible, X_t la valeur $F(t)(X)$ portée par X en t dans le flot F. On note par le même symbole une variable ou une fonction dans les domaines syntaxique et sémantique. Dans le tableau ci-dessous, on omet les événements nuls (qui, rappelons-le, terminent les flots ayant un nombre fini d'événements non nuls).

Le noyau de Signal se compose des primitives suivantes :

– Fonctions ou relations étendues aux suites :

$$\texttt{Y := f(X1,...,Xn)} : \begin{cases} \forall k, \texttt{Y}_k = \perp \Rightarrow \texttt{X1}_k = ... = \texttt{Xn}_k = \perp \\ \forall k, \texttt{Y}_k \neq \perp \Rightarrow \texttt{Y}_k = \texttt{f}(\texttt{X1}_k, ..., \texttt{Xn}_k) \end{cases}$$

– Retard (registre à décalage) :

$$\texttt{Y := X \$1 init v0} : \begin{cases} \forall k, \mathtt{Y}_k = \ \bot \ \Rightarrow \ \mathtt{X}_k = \bot \\ \mathtt{Y}_0 \neq \ \bot \ \Rightarrow \ \mathtt{Y}_0 = \mathtt{v0} \\ \forall k > 0, \mathtt{Y}_k \neq \ \bot \ \Rightarrow \ \mathtt{Y}_k = \mathtt{X}_{k-1} \end{cases}$$

– Extraction sur condition booléenne :

$$\texttt{Y := X when B} : \forall k, \begin{cases} \mathtt{B}_k \ \neq \ true \ \Rightarrow \ \mathtt{Y}_k = \bot \\ \mathtt{B}_k \ = true \ \Rightarrow \ \mathtt{Y}_k = \mathtt{X}_k \end{cases}$$

– Mélange avec priorité :

$$\texttt{Y := U default V} : \forall k, \begin{cases} \mathtt{U}_k \ \neq \ \bot \ \Rightarrow \ \mathtt{Y}_k = \mathtt{U}_k \\ \mathtt{U}_k \ = \bot \ \Rightarrow \ \mathtt{Y}_k = \mathtt{V}_k \end{cases}$$

La composition de deux processus $P|Q$ se traduit directement en la composition des ensembles de flots associés à chacun d'eux.

La restriction de visibilité de `X`, P `/ X` est la projection de l'ensemble des flots associés à P sur l'ensemble des variables obtenu en enlevant `X` à celles de P.

Comme on peut le voir pour les primitives, chaque signal a sa propre référence temporelle (son «horloge», ou ensemble des instants où il est différent de $\bot$). Par exemple, les deux premières primitives sont monocadencées : elles imposent que tous les signaux impliqués aient la même horloge. En revanche, dans la troisième et la quatrième primitives, les différents signaux peuvent avoir des horloges différentes. L'horloge d'un programme Signal est alors la *borne supérieure* de toutes les horloges des différents signaux du programme (les instants du programme sont les instants de l'un au moins de ces signaux).

Le compilateur de Signal consiste principalement en un système formel capable de raisonner sur les horloges des signaux, la logique, et les graphes de dépendance. En particulier, le *calcul d'horloges* [1] et le calcul de dépendances fournissent une synthèse de la synchronisation globale du programme à partir de la spécification des synchronisations locales (qui sont données par les équations Signal), ainsi qu'une synthèse de l'ordonnancement global des calculs spécifiés. Des contradictions et des inconsistances peuvent être détectées au cours de ces calculs.

On peut toujours ramener un programme P comportant des variables locales à un programme, égal à P, de la forme Q`/A1/.../An` où Q est une composition de processus élémentaires sans restriction de visibilité (i.e., sans R`/A`). Un principe général de transformation de programmes que nous appliquons (dans un but de vérification, pour aller vers la mise en œuvre, pour calculer des abstractions de comportement) est alors de définir des homomorphismes $\mathcal{T}_i$ sur les programmes Signal, tels que Q est égal à la composition de ses transformés par $\mathcal{T}_i$. Grâce aux propriétés du monoïde commutatif, la transformation qui à Q associe cette composition est elle-même un homorphisme. On sépare ainsi un programme en différentes parties sur lesquelles seront alors appliqués des traitements spécifiques.

3.2 Vérification et synthèse

Mots clés : Signal, transformation de programme, système dynamique polynomial,

vérification, synthèse de contrôleur, BDD.

Résumé : *Le principe de transformation des programmes Signal permet de décomposer un programme en une partie décrivant le contrôle booléen et une partie contenant les calculs. Le contrôle lui-même définit un système dynamique qui peut être étudié sous plusieurs aspects à des fins de vérification et de synthèse : étude de l'ensemble des états admissibles (partie statique), pour laquelle une forme canonique arborescente utilisant des* BDD *a été définie ; calcul dynamique s'appuyant sur la représentation équationnelle d'un automate.*

En appliquant le principe de transformation, on décompose un programme P en une partie $Q(P)$ contenant le contrôle booléen et une partie $C(P)$ contenant les calculs non booléens, telles que $P = Q(P)|C(P)$.

Toute propriété de sûreté, qui peut s'exprimer sous la forme d'un programme Signal R, satisfaite par $Q(P)$, ce qui s'exprime sous la forme $R|Q(P) = Q(P)$, est également satisfaite par P, puisqu'il résulte de $P = Q(P)|C(P)$ que $P = Q(P)|P$ (voir 3.1.1).

À une équation purement booléenne I correspond $Q(I) = I$; $C(I)$ est alors l'élément neutre du monoïde.
D'une équation monocadencée, est extraite par Q la partie synchronisation des signaux ; on obtient par exemple pour `x := y+z` l'expression :
`x ^= y ^= z | x := y+z`
(`x ^= y ^= z` spécifie l'égalité des horloges de `x`, `y` et `z`).
Une équation de la forme `x := y when b`, dans laquelle `x` est non booléen, est décomposée en : `x ^= (y when b) | x := y when b`.
Une équation de la forme `x := y default z`, dans laquelle `x` est non booléen, est décomposée en : `x ^= (y default z) | x := y default z`.

Cette interprétation permet donc d'extraire, de façon automatique, par $Q(P)$, l'aspect système à événements discrets, du système hybride spécifié par le programme. En raison de l'opérateur de retard qui introduit des indices temporels différents, le système est dynamique.

L'étude de ces systèmes dynamiques repose sur l'utilisation de techniques algébriques sur les corps de Galois. Elle vise à exprimer les propriétés des systèmes dynamiques et à donner une solution algorithmique pour leur vérification et pour la synthèse de systèmes satisfaisant certaines spécifications.

$Q(P)$ est défini sur trois valeurs : `{ vrai, faux, absent }`. La sémantique des opérateurs de Signal et l'approche flot de données équationnelle conduisent naturellement à un codage de $Q(P)$ en équations polynomiales sur le corps $\mathbf{Z}/3\mathbf{Z}$ (ou $\mathcal{F}_3$), `vrai`, `faux`, `absent` étant représentés respectivement par 1, -1, 0 ($+$ est l'addition modulo 3, $\times$ est la multiplication usuelle).

L'étude de la sémantique abstraite d'un programme Signal se ramène alors à l'étude des systèmes dynamiques de la forme :

$$\left\{ \begin{array}{lcl} X_{n+1} & = & P(X_n, Y_n) \\ Q(X_n, Y_n) & = & 0 \\ Q_0(X_0) & = & 0 \end{array} \right.$$

où X est un vecteur d'état dans $(\mathbf{Z}/3\mathbf{Z})^n$ et Y un vecteur d'événements (interprétations abstraites de signaux) qui font évoluer le système.

Un tel système dynamique n'est qu'une forme particulière de système de transitions à espace d'états finis. C'est donc un modèle de système à événements discrets sur lequel il est possible de vérifier des propriétés [2] ou bien de faire du contrôle.

L'étude d'un programme consiste alors en :

– l'étude de sa partie *statique,* c'est-à-dire l'ensemble de contraintes

$$Q(X_n, Y_n) \quad = \quad 0$$

– l'étude de sa partie *dynamique,* c'est-à-dire le système de transitions

$$\begin{aligned} X_{n+1} &= P(X_n, Y_n) \\ Q_0(X_0) &= 0 \end{aligned}$$

et l'ensemble de ses états atteignables, etc.

Différentes techniques ont été développées pour ces deux problèmes. Les contraintes statiques sont essentielles pour la *compilation* des programmes Signal, et des techniques très efficaces ont été développées pour cela. La partie dynamique demande plus de calculs, et est utilisée principalement pour la vérification de propriétés ; une technique plus générale — mais en retour moins efficace — a été développée pour la prendre en compte.

Le calcul d'horloges statique est au cœur du compilateur Signal ; il en détermine largement ses performances. Ce calcul s'appuie sur l'ordre partiel des horloges, qui correspond à l'inclusion des ensembles d'instants (une horloge pouvant être plus fréquente qu'une autre).

La situation suivante doit être considérée : `H` est l'horloge d'un signal, par exemple un signal à valeurs réelles `X`, et `K` est l'ensemble des instants où le signal `X` dépasse un seuil : `K := when (X > X_MAX)`. Alors 1/ chaque instant de `K` est un instant de `H`, et 2/ pour calculer le statut de `K`, il faut d'abord connaître le statut de `H`. Il y a donc à la fois le fait que `K` est moins fréquent que `H` et qu'il existe une contrainte de causalité de `H` vers `K`. Ceci est dénoté par `H` $\rightarrow$ `K`. De tels *sous-échantillonnages* successifs organisent les horloges en plusieurs *arbres,* l'ensemble de ces arbres constituent la *forêt* d'horloges du programme considéré. Si un seul arbre est obtenu, la synchronisation du programme et son exécution s'en déduisent aisément.

La forêt associée à un programme donné n'est pas unique, la question de l'équivalence de forêts d'horloges se pose donc. Une *forme canonique* de forêt a été définie [1]. Un algorithme efficace pour trouver cette forme canonique a été développé. Il repose sur des manipulations préservant l'équivalence, prenant en compte l'ordre des variables résultant de la causalité, et combinées à des techniques BDD (*Binary Decision Diagrams* introduits par Bryant en 1986 [Bry86]).

Le calcul dynamique s'appuie sur la représentation équationnelle d'un automate : les automates, leurs états, événements et trajectoires sont manipulés au travers des *équations* qui les représentent. Calculer des trajectoires d'états ou d'événements, des états atteignables, des projections de trajectoires, des états de *deadlock,* etc., s'effectue alors sur les coefficients des équations polynomiales. De manière similaire, des techniques de *synthèse de contrôle* ont été

[Bry86] R. Bryant, « Graph-Based Algorithms for Boolean Function Manipulations », *IEEE Transaction on Computers C-45,* 8, Août 1986, p. 677–691.

développées pour un système dynamique donné pour différents types d'objectifs de contrôle proposés par Manna et Pnueli [MP92,MP95], mais aussi pour des objectifs de contrôle portant sur la qualité de service.

La manipulation d'équations dans $\mathcal{F}_3$ est tout à fait similaire à la manipulation d'équations booléennes. Une variante de la technique BDD, appelée TDD (Ternary Decision Diagrams — les nœuds ont trois valeurs possibles), a été développée pour réaliser ces calculs. Des expériences ont montré que le système formel Sigali qui en résulte peut effectuer en un temps raisonnable des preuves (ou de la synthèse de contrôleurs) sur des automates comportant plusieurs millions d'états atteignables.

Synthèse d'automatismes discrets

Partant d'un modèle global du système, contrôler un système dynamique polynomial consiste à se donner un objectif de commande (propriétés des trajectoires) et à synthétiser un contrôleur répondant à cet objectif [10]. Dans notre approche, le contrôleur synthétisé est une équation

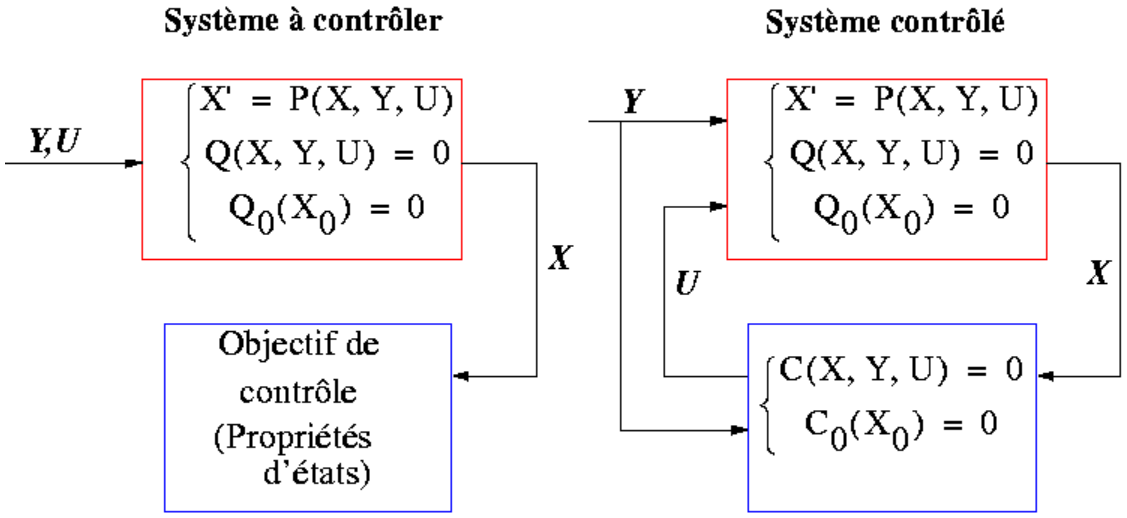

FIG. 1 – Principe du contrôle

polynomiale, C(X, Y, U), dépendant de l'état courant du système, X, des événements incontrôlables, Y, et des commandes, U (figure 1). Le rôle de cette équation, ajoutée au système initial, consiste à forcer la valeur de ceux-ci en restreignant, pour un état donné, le choix possible des valeurs des commandes admissibles. Les événements contrôlables peuvent alors être vus comme des événements de sortie du contrôleur (respectivement des événements d'entrée du système initial).

[MP92] Z. MANNA, A. PNUELI, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992.

[MP95] Z. MANNA, A. PNUELI, *The Temporal Logic of Reactive and Concurrent Systems: Safety*, Springer-Verlag, 1995.

Différents types d'objectifs de contrôle peuvent être considérés : assurer l'invariance, l'atteignabilité ou l'attractivité d'un ensemble d'états, etc. Les objectifs de contrôle peuvent également traduire un critère qualitatif et non plus logique. Ils s'expriment alors comme des relations d'ordre ou comme un critère de minimisation sur une trajectoire bornée du système.

Les différentes mises en œuvre réalisées sont surtout axées sur l'intégration des différents algorithmes induits par les objectifs de contrôle dans le système de calcul formel Sigali, mais sont également axés sur l'intégration de Sigali dans l'environnement Signal de manière à faciliter les preuves de programmes et la synthèse d'objectifs de commande.

4 Domaines d'applications

4.1 Panorama

Mots clés : conception synchrone, télécommunication, traitement temps réel du signal, énergie, transport, avionique, automobile.

Résumé : *Les recherches sur les méthodes de développement d'applications ne sauraient se concevoir sans une confrontation avec des applications, pour identifier en amont les problèmes rencontrés par les concepteurs (utilisateurs potentiels de nos techniques) et pour valider en aval les solutions proposées. C'est ainsi que le projet* EP-ATR *s'est impliqué très tôt dans des travaux liés aux télécommunications. En outre, il a été longtemps engagé dans une coopération avec* EDF *sur l'utilisation des techniques synchrones dans le domaine de l'énergie. D'autres domaines ont été abordés, en particulier une application en traitement radar infra-rouge a été traitée.*

Dans le cadre du projet Sacres puis du projet SafeAir, les applications considérées relèvent du domaine de l'avionique. Le domaine des applications automobiles a également été abordé à travers le projet AEE.

4.2 Télécommunications

Participants : Albert Benveniste, Paul Le Guernic, Jean-Pierre Talpin, Yunming Wang.

Mots clés : Signal, télécommunication, communication synchrone/asynchrone, conception objet.

Résumé : *Nos activités dans le domaine des télécommunications sont issues d'une longue collaboration avec le Cnet, d'où provient le développement initial du langage Signal. Elles se sont poursuivies dans le cadre du projet Cairn autour des langages Signal et Alpha, ainsi que dans de nouvelles collaborations dans lesquelles l'utilisation mixte des modèles synchrone et asynchrone est étudiée.*

L'industrie des télécommunications est, de plus en plus, soumise à de fortes contraintes qui demandent un effort important visant à maximiser la généricité des solutions proposées et à raccourcir les délais de mise sur le marché des produits. La diversité que l'on rencontre dans les applications développées nécessite la mise en œuvre de techniques variées pour répondre

aux problèmes rencontrés. Les techniques synchrones peuvent fournir des solutions partielles qu'il convient d'intégrer dans des méthodes de conception plus globales.

Issu d'une longue collaboration avec le Cnet, le langage Signal a d'abord été développé dans le cadre d'applications en traitement du signal. Le projet Cairn («Codesign d'Applications Irrégulières et Régulières par Niveaux») nous a permis, en collaboration avec le projet Api de l'Irisa, d'étendre les thèmes abordés à la conception de composants supportant des algorithmes qui comportent du calcul numérique intensif (image ou signal) et du contrôle complexe.

La taille et la complexité des applications mises en œuvre, la nécessité d'obtenir des spécifications et des programmes génériques, destinés à des configurations diverses fonctionnant dans des contextes hétérogènes, conduisent à mettre en œuvre des outils de conception fondés sur l'approche objet. Dans le cadre de cette approche, la complexité des interactions et les contraintes temps réel sont traitées à l'aide de descriptions de comportements faisant appel à des modèles d'automates. L'étude de l'utilisation de l'approche synchrone dans ce contexte a d'abord été entreprise en collaboration avec les Laboratoires de Marcoussis par la définition d'un modèle d'interaction entre des objets (décrits dans le langage Spoke) et des processus (décrits dans le langage Signal). Elle s'est poursuivie ensuite avec Alcatel, dans le cadre de l'action Reutel-2000 qui visait notamment à maîtriser la conception objet d'applications mises en œuvre selon des techniques mêlant les modèles synchrone et asynchrone.

4.3 Énergie

Participants : Fernando Jiménez Fraustro, Hervé Marchand, Sophie Pinchinat.

Mots clés : énergie, méthode formelle, programmation synchrone, automatismes industriels.

Résumé : *Nos activités dans le domaine de l'énergie se sont situées depuis plusieurs années dans le cadre de coopérations avec* EDF*, prolongées à travers le projet européen Syrf. Elles concernent notamment la vérification et la synthèse de contrôleurs, mais aussi la simulation de systèmes hybrides.*

Dans le domaine de la production et de la distribution d'énergie, en particulier électrique, on est en présence de systèmes dont :

- La sécurité est un caractère essentiel ; elle concerne divers aspects des systèmes de contrôle :
  - le service fourni peut avoir des aspects critiques dans la nature de ceux à qui il est destiné (par exemple les hôpitaux),
  - le matériel lui-même est soumis à des conditions de fonctionnement dont le dérèglement peut entraîner des dommages fatals,
  - enfin les techniques utilisées peuvent comporter un risque pour l'environnement dans lequel s'effectue l'activité.

  Dans ces conditions, les systèmes qui contrôlent la production et la distribution d'énergie posent des problèmes qui sont du domaine d'application de nos techniques d'analyse et de vérification de comportement.
- La complexité est grande :

- que ce soit dans le cas de réseaux de distribution, constitués de contrôleurs de transformateurs interconnectés,
- ou bien de contrôleurs de centrale électrique, où les capteurs à prendre en compte et les actionneurs auxquels fournir une commande se comptent par centaines, voire milliers, et où les architectures sur lesquelles le contrôle s'exécute comprennent des réseaux d'automates programmables en parallèle.

La conception et l'analyse de ces systèmes requièrent le support d'outils automatisés pour la construction de modèles et le calcul de mises en œuvre correctes vis-à-vis de la spécification. Elles doivent exploiter la particularité d'architectures à base d'automates programmables, comme il en est construit par Siemens, avec qui nous coopérons dans le projet européen SafeAir.

- L'héritage des spécifications de contrôleurs obéit à une culture particulière : les langages de spécification des contrôleurs ont été utilisés pour la conception de systèmes de taille importante, et récrire ces systèmes dans un langage nouveau n'est guère envisageable. Il s'agit de langages à base d'automates communicants, ou d'autres de type *blocs-diagrammes* et circuits d'opérateurs. Dans un but de meilleure acceptation des méthodes formelles par les utilisateurs, et de réutilisation du fonds des développements antérieurs, il est intéressant de travailler à l'intégration de ces langages aux techniques que nous proposons, sous la forme de leur encodage, et d'une traduction automatique dans un format synchrone.

4.4 Avionique

Participants : Albert Benveniste, Loïc Besnard, Abdoulaye Elhadji Gamatié, Thierry Gautier, Paul Le Guernic.

Mots clés : conception synchrone, Statecharts, Scade, Arinc.

Résumé : *Nos activités dans le domaine de l'avionique se sont développées depuis plusieurs années dans le cadre de projets européens, notamment Sacres et SafeAir. Elles concernent tous les aspects du processus de développement, et en particulier la génération de code sur une architecture temps réel.*

Le domaine de l'avionique est clairement un domaine critique pour les systèmes enfouis, où les industries de pointe sont confrontées à une exigence de sécurité maximale. L'industrie européenne est particulièrement bien placée dans ce secteur et doit conserver et améliorer ses atouts pour répondre à la forte augmentation actuelle en fonctionnalité et en complexité. L'avionique est donc un domaine d'application privilégié pour les méthodes formelles en général et spécialement la technologie synchrone.

Dans le cadre de plusieurs projets européens successifs, nous avons été amenés à collaborer étroitement avec des industriels clés du domaine. Il s'agit des projets Synchron, Sacres (voir `http://www.tni.fr/sacres`), Syrf, et aujourd'hui SafeAir (voir section 7.3). La conception de systèmes avioniques s'appuie d'ores et déjà sur un processus de développement suivant le «cycle en V», sur l'emploi de technologies innovantes (utilisation des outils Statemate, Scade, Simulink, par exemple), sur l'utilisation de librairies temps réel standards (Arinc), sur le res-

pect de normes de certification (DO-178B). L'objectif d'un projet comme SafeAir, qui se base sur l'utilisation de ces technologies, est de permettre de réduire de 35 à 40% l'effort de développement des systèmes logiciels aéronautiques, tout en augmentant leur fiabilité : passage du «cycle en V» vers un «cycle en Y», en utilisant des méthodes de conception de haut niveau et des outils de génération automatique de code et de vérification formelle.

5 Logiciels

Polychrony pour Signal

5.1 Environnement de programmation Polychrony pour Signal

Mots clés : Signal, programmation synchrone, format commun, DC+, transformation de programme, Sildex.

Contact : L. Besnard.

Résumé : *Le développement d'un environnement de programmation Signal, construit à l'Irisa selon des techniques de conception modulaires, répond à trois objectifs :*

a) il nous permet d'étudier des extensions sémantiques ou algorithmiques du modèle synchrone ;

b) il nous permet de mieux comprendre les applications et de dégager ainsi des problématiques nouvelles ;

c) il est diffusé à des fins d'expérimentation et d'enseignement dans des laboratoires pour lesquels la version commerciale Sildex ne convient pas.

L'environnement de programmation Signal, appelé Polychrony, se compose d'un ensemble de fonctionnalités de compilation et d'un éditeur graphique orienté blocs-diagrammes. *Il doit être vu comme une «boîte à outils» synchrone. L'éditeur de Signal permet à l'utilisateur de construire ses programmes sous forme à la fois textuelle et graphique. L'existence de cet éditeur est un vecteur majeur pour la diffusion.*

Les fonctionnalités de compilation sont écrites en C++ et C Ansi. L'éditeur graphique est écrit en Java (utilisation de la librairie Swing), C++ et Signal. Ceci permet d'avoir une version commune de l'environnement Signal dans les mondes Windows et Unix.

Environnement de compilation

L'environnement de compilation Signal est un outil interactif de conception d'applications. L'architecture de l'environnement peut être vue comme un ensemble de services. Un service peut être ou non appliqué selon l'objectif : simulation, vérification formelle, compilation séparée, génération de code distribué, etc. Ces fonctionnalités sont accessibles au moyen d'options pour le compilateur «batch» et de manière interactive sous l'éditeur graphique. Le superviseur de l'environnement est lui-même un programme Signal.

Ces fonctionnalités s'appliquent également au *format commun* de la programmation synchrone (DC+). Ce format DC+, issu des travaux des projets européens Synchron et Sacres, permet de représenter, par delà un langage particulier, la paradigme «flots de données synchronisés». Il constitue aussi un format concret, servant de vecteur commun de représentation, pour des programmes ou des propriétés sur lesquels on souhaite appliquer des transformations définies dans le cadre du modèle synchrone.

Un programme Signal/DC+ est représenté de façon interne par un GHDC (graphe hiérarchisé aux dépendances conditionnées) qui constitue donc la structure principale de l'environnement.

On peut distinguer :

- un ensemble de traitements qui produisent un graphe hiérarchisé à partir d'un source Signal ou DC+ ;
- un ensemble de transformations du graphe hiérarchisé, transformations qui restituent un graphe hiérarchisé ; ceci constitue le cœur du compilateur ;
- un ensemble de traitements qui produisent les sources d'autres outils.

Production du graphe. L'ensemble des fonctionnalités pour la production du GHDC est constitué :

- de l'analyse syntaxique et contextuelle, qui fournit la représentation interne d'un programme source, Signal ou DC+, sous forme d'un arbre de syntaxe abstraite ;
- de la production de graphe, qui associe à tout programme un graphe caractérisé par un système d'équations d'horloges. À ce stade, aucune vérification sur le système d'équations ni sur le graphe (cycles...) n'est effectuée.

Transformations du graphe. L'ensemble de traitements qui transforment le graphe hiérarchisé est constitué :

- de la compilation, dont le rôle principal est de trianguláriser le système d'équations d'horloges et de détecter la présence de cycles de dépendance de données. Cette transformation permet la vérification partielle de la correction du programme vis-à-vis de ses synchronisations. La synthèse partielle d'expressions explicites du contrôle se traduit en une forêt d'arbres d'horloges, dont les racines sont éventuellement arguments de contraintes non résolues, et les nœuds internes des expressions explicites. Cette représentation sous forme de forêt d'arbres d'horloges (événements) correspond au format DC+. Pour ce *calcul d'horloges,* nous avons développé une structure hiérarchique de BDD qui s'avère très performante ; nous utilisons actuellement le package BDD de Berkeley.
- des transformations inter-formats définies dans le cadre du projet Sacres. Ces transformations, décrites d'abord sur le format DC+, sur les différents niveaux de sous-formats ayant été identifiés, s'appliquent sur la forme interne d'un programme Signal (GHDC). Les différents sous-formats caractérisent une forme particulière du GHDC.
  Ainsi, le sous-format bDC+ (pour «boolean DC+»), dans lequel les horloges, représentées comme des flots booléens, sont organisées en une hiérarchie pour laquelle il existe une horloge maîtresse, est le point d'entrée adéquat pour des outils s'appuyant sur la hiérarchie des horloges, comme par exemple des générateurs de code.
  Le sous-format sbDC+ (pour «sequentialized boolean DC+») est le format d'entrée effectif des générateurs de code. Il est produit pour les programmes (bDC+) sans cycles et sans

contraintes. Un code sbDC+ est une liste de nœuds ordonnés selon les dépendances implicites et explicites du programme Signal–DC+. Ceci permet d'écrire de nouveaux générateurs de code sans avoir à parcourir le graphe du programme.
Le sous-format STS (pour «Symbolic Transition Systems») de bDC+, dans lequel la hiérarchie des horloges est plate (le statut présent/absent d'un signal est défini à tout instant par un booléen), est utilisé en entrée d'outils de vérification.
Le sous-format DC est un sous-format mono-horloge de STS.
Les transformations inter-formats, DC+ → bDC+, bDC+ → SBDC+, bDC+ → STS et STS → DC sont des fonctionnalités de l'environnement. Elles sont appliquées selon l'objectif de la compilation.

- de l'application du principe de substitution du langage. On peut citer :
  - la transformation des booléens (opérateur logiques et relationnels) en événements ; cette opération peut être utile au calcul d'horloges afin de prouver des équivalences ;
  - la suppression des renommages (équations de définition triviales) ;
  - l'unification des signaux définis par la même expression ;
  - la substitution des signaux référencés au plus N fois dans le programme (N étant un paramètre de compilation), par leur expression de définition.
- des opérations de partitionnement de graphe, qui produisent un graphe constitué de nœuds représentant eux-mêmes des graphes. On peut citer :
  - la séparation contrôle/calculs, qui consiste à séparer la partie contrôle de l'application de la partie calcul.
  - la séparation état/reste, qui consiste à séparer la partie état du programme du reste de l'application.
  - le calcul de lignées sur entrées, qui consiste à partitionner un graphe selon le critère qualitatif suivant : deux nœuds sont éléments de la même lignée sur entrée si et seulement s'ils sont précédés du même sous-ensemble d'entrées. Ce partitionnement est à la base d'un nouveau schéma de génération de code séquentiel : une lignée peut être exécutée de manière atomique dès que ses entrées sont disponibles.
  - la répartition de programmes, qui se base sur l'utilisation de *pragmas* pour l'affectation des nœuds à des unités de calcul.
- des calculs systèmes suivants :
  - la synthèse d'interface, dont le but est l'extraction d'éléments de la représentation interne en vue de la compilation séparée de programmes Signal. Cette opération consiste en le calcul de la fermeture transitive du graphe réduite aux entrées/sorties du processus compilé.
  - le *retiming,* qui consiste en la réécriture de toute fonction synchrone construite sur les expressions de retard afin d'une part, de faire apparaître des variables d'état booléennes et d'autre part, de réduire le nombre des variables d'état.

Production de code Cet ensemble est constitué :

- de la génération de code séquentiel, qui passe par un tri topologique du graphe et qui produit du code C ou c++ ;
- de la restitution de source Signal, qui fournit à l'utilisateur le GHDC sous la forme d'un nouveau programme Signal faisant apparaître la hiérarchie obtenue et les synchronisa-

tions calculées. La restitution du source peut également être effectuée en partant d'une représentation sous forme d'arbre de syntaxe abstraite.
- de la restitution de source DC+, ceci afin de pouvoir se connecter aux outils disponibles autour du format.
- de l'interfaçage avec des systèmes de preuves ; actuellement la connexion avec l'outil Sigali est réalisée dans le but d'étudier les propriétés dynamiques des programmes (décompilateur $\mathbf{Z}/3\mathbf{Z}$).
- les calculs d'architectures ; actuellement la connexion avec l'outil Syndex est réalisée. L'outil Syndex (Y. Sorel, projet Sosso à Rocquencourt) permet d'effectuer une implantation optimisée sous contraintes temps réel sur une architecture multi-processeur.

Diffusion du logiciel

La version commerciale de Signal est vendue par TNI sous la forme de l'environnement Sildex. La version Inria de Signal, qui jusqu'à cette année pouvait être obtenue dans le cadre d'une convention de mise à disposition gratuite signée pour un an renouvelable sera bientôt accessible, en accès libre, sur le site Web du projet Espresso, successeur du projet EP-ATR, dans le cadre du nouvel environnement Polychrony.

Signal est actuellement mis à disposition dans des écoles ou universités (Ubo, IUP de Lorient, université de Nantes Irin, Supelec, Oil & Gas University of Ploiesti — Pologne, University of Victoria — Canada, University of Michigan — USA, Mecaprom — Mexique), et chez certains industriels pour des études ou évaluations particulières (EDF). Notre objectif est de fournir à court terme une distribution sous une licence de type logiciel libre (voir section 6.1).

5.2 Sigali

Mots clés : Signal, DC+, Sigali, système dynamique polynomial, vérification, synthèse de contrôleur.

Contacts : L. Besnard, H. Marchand.

Résumé : *Sigali est un système de calcul formel permettant la vérification de propriétés de programmes Signal ou* DC+.

Sigali est un système de calcul formel interactif spécialisé dans les calculs algébriques sur l'anneau $\mathbf{Z}/3\mathbf{Z}[X]$. Il est destiné à la vérification des propriétés statiques et dynamiques de programmes Signal ou DC+ [1, 2] et plus généralement de tout système dynamique polynomial dans $\mathbf{Z}/3\mathbf{Z}[X]$. Il permet également la synthèse de contrôleurs de systèmes à événements discrets. L'adjonction de primitives de création et de manipulation de fonctions à valeurs entières autorise les calculs de commande optimale.

Sigali est un logiciel déposé à l'APP sous le numéro IDDN.FR.001.370006.S.P.1999.000.10600. Comme pour l'environnement Polychrony, Sigali peut être obtenu actuellement après signature d'une convention de mise à disposition.

6 Résultats nouveaux

Nos thèmes de recherche introduits en 2.4 ont donné lieu aux résultats décrits dans les sections suivantes :

- les sections 6.1 à 6.4 relèvent de la «Description d'applications» (2.4.1) ;
- les sections 6.5 et 6.6 concernent l'«Étude des propriétés des processus synchrones» (2.4.2) ;
- les sections 6.7 à 6.9 portent sur des «Méthodes et outils pour la conception d'architectures de mise en œuvre» (2.4.3).

6.1 Évolutions de Signal et de son environnement

Mots clés : programmation synchrone, Signal, transformation de programme, compilation séparée, évaluation de performance.

Participants : Loïc Besnard, Patricia Bournai, Thierry Gautier, Paul Le Guernic, Laurent Vibert.

Résumé : *La version actuellement diffusée de Signal, Signal V4, a été définie en coopération avec la société* TNI *(François Dupont), qui développe et commercialise l'environnement Sildex issu des travaux sur Signal. Elle est progressivement enrichie dans le sens d'une meilleure expressivité, et son environnement est amélioré de manière à en assurer une diffusion plus large.*

Évolutions du langage

Nous avons précisé la notion d'«affectation multiple» (syntaxique) de signaux qui avait été introduite l'an passé dans le langage Signal. Une équation de définition partielle `X :=` $\mathtt{E}_i$ se récrit en l'équation flot de données `X :=` $\mathtt{E}_i$ `default X`. Différentes expressions $\mathtt{E}_i$ définissant `X` dans des équations de définition partielle doivent avoir la même valeur à leurs instants communs lorsqu'elles ont des instants communs. L'horloge de `X` est plus grande que la borne supérieure des horloges des expressions $\mathtt{E}_i$ définissant `X`. Il est possible de compléter la définition de `X` par une valeur par défaut (qui peut être une constante) lorque les $\mathtt{E}_i$ sont absents.

Nous avons également introduit une notion de variables d'état (*statevar*), représentées par des signaux dont l'horloge est plus grande que la borne supérieure des horloges de tous les signaux de l'unité de compilation dans laquelle la variable d'état est déclarée. Les variables d'état sont définies exclusivement par des équations de définition partielle, qui définissent, à une horloge donnée, les valeurs *suivantes* de la variable d'état. L'occurrence d'une variable d'état dans une expression permet d'accéder à sa valeur au début du *step* courant.

Combinées avec l'extension de la visibilité des signaux et avec l'intégration de la structure *case,* ces extensions permettent notamment une spécification plus souple des automates en Signal. L'introduction de ces notions nous a permis par exemple de simplifier l'écriture du programme Signal qui supervise l'environnement Signal.

Le symbole de définition flot de données, `:=`, induit une relation de dépendance entre l'expression de partie droite et le signal défini. Nous avons introduit un nouveau symbole, noté `:= :`, qui permet de spécifier des définitions non orientées. Une équation `X := E` se traduit

alors par la composition `X := : E | E --> X`. La sémantique de traces de Signal s'exprime sur des équations non orientées (en cours d'écriture dans la nouvelle version du manuel de référence du langage).

De manière proche de ce que permet le langage Esterel, nous considérons qu'il existe un *tick* pour tout processus Signal, qui est un signal dont l'horloge est plus grande que la borne supérieure des horloges de tous les signaux du processus. Ce *tick* peut être désigné en Signal à travers l'étiquette pouvant être associée à un processus.

Nous distinguons alors différentes classes syntaxiques de processus :

- la *fonction* : sans état, synchrone sur ses entrées-sorties, toute entrée précède toute sortie ;
- le *nœud* (à la Lustre) : endochrone, le *tick* appartient aux entrées-sorties, toute entrée précède toute sortie ;
- l'*action* : le *tick* est externe, toute entrée précède toute sortie ;
- le *processus* : le plus général, exochrone.

Compilation séparée

La compilation séparée de différents modules agissant entre eux est un besoin de plus en plus important pour une utilisation dans de larges applications. À la différence de [NTGL97], les modules que nous lions dynamiquement ne dépendent pas l'un de l'autre via une interface de foncteur, mais partagent juste un ensemble de signaux. Pour que ce partage puisse avoir lieu sans gêner la compilation séparée, il faut trouver des contraintes pertinentes sur les horloges de ces signaux.

Partant d'une algèbre de processus minimale, on élabore un ensemble de contraintes d'horloges pour chacun des modules. Pour que deux processus P et Q partageant un signal $\hat{}x$ puissent être liés, il faut que l'horloge de ce signal x ne soit pas gardée par des contraintes contradictoires, c'est-à-dire que l'on puisse prouver que x ne peut pas être à la fois présent dans P et absent dans Q.

À plus long terme, on essaiera d'étendre les techniques de compilation séparée à la génération dynamique de processus synchrone, et de l'appliquer à un calcul objet tel que celui développé par M. Kerbœuf et J.-P. Talpin (voir section 6.3).

Environnement Signal

Outre l'introduction des notions nouvelles décrites ci-dessus, nous avons également défini et mis en œuvre cette année des algorithmes de morphismes de programmes Signal : pour un programme P et une description de la transformation (module Signal qui décrit pour tout opérateur la transformation qui lui est associée), un nouveau programme (image de P) est automatiquement engendré. Ce principe a été appliqué pour l'extraction de propriétés temporelles [6, 27] dans le cadre du projet SafeAir. Pour cela, à partir du programme source, correspondant à la spécification fonctionnelle d'un système, on engendre automatiquement son image «temporelle» par une interprétation ; cette image est un processus Signal qui consiste à

[NTGL97] D. Nowak, J. Talpin, T. Gautier, P. Le Guernic, « An ML-like module system for the synchronous language Signal », *in : European Conference on Parallel Processing (Euro-Par'97)*, Springer-Verlag, LNCS 1300, p. 1244–1253, 1997, `ftp://ftp.irisa.fr/local/signal/publis/articles/EuroPar-97:modul.ps.gz`.

mettre à jour les dates de disponibilité des entrées afin de produire les dates de disponibilité des sorties. La mise à jour des dates est faite de manière à prendre en compte les facteurs tels que les délais des opérations individuelles (dépendants de l'architecture cible) et le flot d'exécution (dépendant du contrôle de l'application).

Cette image, couplée avec le programme d'origine, permet alors d'engendrer des simulateurs capables de simuler simultanément les comportements fonctionnel et temporel d'un programme.

D'autre part, un travail de grande ampleur a été entrepris pour redéfinir complètement l'environnement de programmation Signal, désormais appelé Polychrony. Le nouvel environnement d'édition des programmes Signal est écrit en Java. Cette refonte a retardé la diffusion prévue de la «boîte à outils» Polychrony comme logiciel libre disponible sur le Web, mais elle devrait améliorer largement tant les fonctionnalités que l'ergonomie de Polychrony.

6.2 Sémantique des diagrammes d'état de UML

Participants : Albert Benveniste, Paul Le Guernic, Jean-Pierre Talpin, Yunming Wang.

Mots clés : système de transition synchrone, Statecharts, UML, BDL.

Résumé : *En collaboration avec le projet Pampa, et dans le cadre du contrat Reutel avec Alcatel (maintenant achevé), nous avons élaboré un formalisme synchrone de préordres étiquetés, appelé* BDL *(*Behavioral Description Language*).* BDL *est destiné à permettre la manipulation abstraite d'objets, de composants, et d'architectures, avec une sémantique double synchrone/asynchrone.*

Nous avons introduit un modèle permettant de représenter indifféremment les systèmes réactifs synchrones et les systèmes distribués asynchrones. Par rapport aux langages synchrones existants, ce modèle offre une nouvelle loi de composition : le choix non déterministe. Avec la composition parallèle synchrone et le choix non déterministe, on dispose des opérations adéquates pour faire de l'héritage sur le plan comportemental (par addition de contraintes, ou par enrichissement de comportements). BDL est une syntaxe concrète qui met en œuvre ce modèle. BDL bénéficie complètement de l'ensemble des résultats sur la désynchronisation (endo- et isochronie). BDL est destiné à être connecté à la plateforme Umlaut dévloppée dans le projet Pampa, ainsi qu'à l'environnement Signal pour bénéficier des outils qui y sont associés. Nous avons défini une syntaxe graphique pour BDL, par adoption de traits graphiques du formalisme des Statecharts et des formalismes à scénarios. Notre objectif est de pouvoir transcoder en BDL à la fois des Statecharts UML, et des diagrammes de séquence UML. Une version révisée du rapport Inria N^o 4003 est en cours d'achèvement. Nous y traitons d'un exemple d'adaptation de service à l'aide de BDL.

Traduction des Statecharts UML en BDL. Nous avons proposé une structure formelle, complète et récursive des Statecharts et une méthode de traduction vers BDL. Cela donne une sémantique formelle et complète des Statecharts, à la différence des travaux existants, où les sémantiques formelles sont toujours incomplètes, et les sémantiques complètes sont toujours informelles. Cette sémantique se conforme à la sémantique définie par UML. Le prototype de

cette traduction qui avait été entamé l'an dernier, a été achevé cette année, et fait l'objet de la thèse de Yunming Wang [13].

Traduction de BDL vers Signal. Une première maquette de traducteur de BDL vers Signal a été écrite en oCaml. Grâce aux récentes évolutions de Signal, ce traducteur est particulièrement compact.

6.3 Objets synchrones

Participants : Michaël Kerbœuf, Jean-Pierre Talpin.

Mots clés : programmation objet.

Résumé : *Le modèle de concurrence synchrone des langages comme Signal est parfaitement adapté au mécanisme d'*héritage *du monde objet. En effet, il n'est pas confronté à l'«*anomalie de l'héritage*» induite par le modèle plus classique de concurrence asynchrone. Objective-*SIGMA *est un calcul unifiant les paradigmes synchrone et objet dans un même cadre formel. Il autorise la conception d'applications concurrentes en bénéficiant pleinement de la souplesse de développement des langages objets.*

Les principes de la programmation objet sont maintenant des concepts largement adoptés dans l'industrie. Parce qu'elle met en œuvre une forme de programmation incrémentale satisfaisant la propriété d'encapsulation en faisant abstraction des détails d'implémentation grâce au mécanisme d'héritage, l'approche objet autorise la réutilisation fiable et efficace de composants logiciels et favorise de ce fait la conception et la maintenance à grande échelle.
Ce modèle n'est cependant pas naturellement adapté à la conception d'applications concurrentes. En effet, si le concept d'objet sied bien à l'idée de *processus* en tant qu'*unité de concurrence*, le concept d'héritage semble en revanche souvent inapproprié. Cette inadéquation entre héritage et concurrence identifiée comme étant l'«*anomalie de l'héritage*» provient notamment de l'absence d'information quant au comportement de synchronisation du composant hérité.
Un moyen simple d'éviter cet écueil est de donner un cadre *synchrone* à la concurrence. Dans cette approche, le schéma de synchronisation d'un composant dont on souhaiterait hériter est naturellement exprimé dans son interface par des *relations d'horloges* sur les signaux présents en entrée et en sortie.
En Signal, la composition de processus est une forme de programmation incrémentale qui respecte la propriété d'encapsulation. Pour qualifier ce mécanisme d'«héritage», il manque cependant la possibilité de *récrire* une partie du composant hérité (i.e., redéfinir des signaux du système initial) ainsi que la possibilité d'accéder aux versions précédentes des signaux redéfinis (comme on le ferait via la référence *self* dans les langages impératifs classiques).
Ce mécanisme d'*héritage synchrone* est défini dans objective-SIGMA : une extension objet d'un calcul de processus synchrones : core-SIGMA. Ce formalisme constitue un dénominateur commun entre les langages Lustre et Signal et résume à l'aide d'un opérateur de choix non-déterministe et d'un opérateur de composition synchrone l'essence du paradigme synchrone flot de données. L'extension objet de ce calcul donne une orientation à ces opérateurs et leur

confère respectivement un pouvoir d'extension et de raffinement avec prise en compte des redéfinitions.
Un rapport de recherche en cours de rédaction rassemble ces résultats et un prototype de traducteur de objective-SIGMA vers Signal est développé parallèlement.

synchrone de la norme IEC 1131 de programmation des systèmes de contrôle

6.4 Multi-formalisme et modélisation synchrone de la norme IEC 1131 de programmation des systèmes de contrôle

Participants : Fernando Jiménez Fraustro, Paul Le Guernic.

Mots clés : programmation synchrone, Signal, automatismes industriels, norme IEC 1131, Grafcet.

Résumé : *Dans le prolongement des approches multi-formalisme que nous avions adoptées dans les projets européens Synchron, Sacres et Syrf avec le format commun* DC+*, nous considérons une telle approche dans le contexte des langages de la norme* IEC 1131 *concernant les automatismes industriels (ces langages comportent le formalisme du Grafcet ainsi que différents langages graphiques et textuels).*

Ce travail a été conduit en coopération avec Éric Rutten (projet Bip, Inria Rhône-Alpes).

Nous avons finalisé un modèle [11] de langages de la norme IEC 61131 [IEC93] en termes du langage synchrone Signal. Ce modèle repose sur une traduction structurelle, qui fait usage des particularités de Signal, notamment le sur-échantillonnage : ce dernier permet de construire un modèle dont la granularité est assez fine pour distinguer les calculs internes des interactions avec l'environnement (lecture des entrées, écritures des sorties), et de prendre en compte des programmes comportant des itérations non-bornées [19] ; Ce faisant nous réutilisons des concepts mis au point en relation avec la modélisation synchrone des Statecharts [14].

6.5 Vérification des programmes

Participants : Abdoulaye Elhadji Gamatié, Sylvain Kerjean, Paul Le Guernic, Mirabelle Nebut, Sophie Pinchinat.

Mots clés : vérification, Signal, calcul d'horloges, système logique, sémantique synchrone, calcul d'intervalles.

Résumé : *Nous avons défini un langage appelé $\mathcal{CL}$ pour la description des propriétés «instantanées» des processus flot de données, c'est-à-dire des propriétés de sécurité ne faisant référence qu'à un seul instant quantifié universellement. Ces propriétés comportementales sont vérifiées sur une abstraction statique des programmes. Nous donnons une procédure de décision pour $\mathcal{CL}$ ainsi qu'une traduction*

[IEC93] IEC, « International Standard for Programmable Controllers », *rapport de recherche n° IEC 1131 parts 1–5*, IEC (International Electrotechnical Commission), 1993.

de Signal en $\mathcal{CL}$, permettant de ramener le model-checking *des programmes à la procédure de décision.*

Dans le cadre de l'élargissement des domaines d'abstraction à des domaines finis, nous expérimentons actuellement un calcul d'intervalles.

Modèle pour la vérification des programmes statiques

Nous utilisons les modèles flot de données classiques : les modèles d'exécution sont des flots (des séquences de valuations) et les modèles des programmes sont des processus (des ensembles de flots). Nous avons défini un langage appelé $\mathcal{CL}$ (pour *Clock Language*) qui décrit les propriétés instantanées des processus. Une procédure de décision a été établie : étant donné une formule ϕ de $\mathcal{CL}$ elle répond OUI s'il existe un processus qui satisfait cette formule, NON sinon. Elle repose sur une abstraction booléenne couplée à un test de satisfiabilité numérique. Il est en fait suffisant de trouver une valuation qui safisfait ϕ : le flot qui répète infiniment cette valuation et donc le processus composé de ce flot satisfait aussi ϕ. Le langage $\mathcal{CL}$ est appliqué à la vérification des processus décrits en Signal. Il peut être vu comme une extension du langage purement booléen des horloges couramment utilisé dans l'environnement Polychrony sous le nom d'«algèbre des horloges», et dont l'isomorphisme avec le calcul propositionel sous-tend le processus de compilation. Comme la traduction de l'abstraction statique des programmes flot de données (en particulier de programmes Signal) en $\mathcal{CL}$ est directe, leur *model-checking* (le programme satisfait-il une propriété spécifiée en $\mathcal{CL}$?) est ramené à la procédure de décision.

Ce travail est présenté dans un rapport de recherche [24]. Dans ce rapport on revient sur les abstractions, analyses et formalismes utilisés dans l'environnement Polychrony, qui sont très variés. Par exemple une analyse d'atteignabilité utilisant une abstraction dans le domaine des polyèdres [BJT99] se réclame de l'interprétation abstraite, tandis que d'autres abstractions sont décrites par des récritures syntaxiques de programmes Signal. De plus, l'abstraction par les synchronisations est originale dans son principe, tout en n'étant qu'un cas particulier d'abstraction booléenne. Nous avons clarifié les choses en opposant les abstractions sémantiques dans des domaines abstraits aux abstractions que nous appelons «structurelles» : un programme est décomposé en deux sous-programmes sont l'un est utilisé comme abstraction du programme d'origine. Nous replaçons ces abstractions dans le cadre de l'interprétation abstraite.

Nous nous intéressons maintenant aux aspects algorithmiques. La procédure de décision pour $\mathcal{CL}$ traite d'abord les aspects booléens puis applique une procédure de décision numérique classique. Elle prouve que le mélange des horloges «booléennes» et des valeurs de variables est effective, mais ne fournit pas d'interaction entre les deux. Or un algorithme pratique devrait faire interagir le contrôle et les données. Nous travaillons sur la représentation d'une formule $\mathcal{CL}$ par un graphe dont les nœuds sont des horloges et les arcs dénotent une inclusion d'horloges, décorés avec des ensembles de formules numériques. Ce graphe, construit dans une approche «bottom-up» avec un degré de précision à définir, devrait permettre la résolution de problèmes spécifiques. Deux applications immédiates devraient être un enrichissement du calcul d'horloges

[BJT99] F. BESSON, T. JENSEN, J.-P. TALPIN, « Polyhedral Analysis for Synchronous Languages », *in: Static Analysis*, A. Cortesi, G. Filé (éditeurs), *Lecture Notes in Computer Science, 1694*, Springer, p. 51–68, 1999.

et un remaniement de l'algorithme de normalisation de [BJT99].

Calcul d'intervalles
La vérification de spécifications temps réel synchrones en Signal s'effectue actuellement par l'intermédiaire du calcul d'horloges qui, on l'a vu, prend en compte uniquement les aspects de contrôle booléen de la spécification. Dans le cadre de l'extension de la vérification à des propriétés non booléennes, nous avons testé et modifié en vue d'integration dans le compilateur Signal un *package* dit IBDD, défini par A. Gamatié, permettant de représenter des formules logiques portant sur des variables booléennes ou à valeurs dans des domaines d'intervalles. Ceci a permis sur des exemples d'affiner le calcul d'horloges booléen. À terme nous espérons élaborer des domaines abstraits (au sens des connexions de Galois) performants en vue de l'analyse statique ou dynamique de spécifications synchrones. Les intervalles représentent pour cela une base de départ intéressante car elle présente un bon rapport coût/performance. Cela va nous amener naturellement à combiner des treillis relationnels et non relationnels en vue d'analyser des spécifications synchrones.

6.6 Synthèse automatique de contrôleurs

Participants : Hervé Marchand, Sophie Pinchinat, Stéphane Riedweg.

Mots clés : vérification, Sigali, système dynamique polynomial, Signal, génération d'automate, bisimulation, méthodes symboliques, synthèse de contrôleurs, commande optimale, modèle hiérarchique, génération de test.

Résumé : *Sur la base des modèles d'automates symboliques* [MBLL00]*, nous nous sommes intéressés à l'application de techniques de synthèse de contrôleurs discrets, eux-mêmes intégrés dans le cadre d'un environnement de programmation au niveau tâche en robotique dans l'environnement Signal/Sigali, ainsi qu'au calcul de diagnostiqueur symbolique. Pour finir, une nouvelle maquette de Sigali a été réalisée. Dans un cadre explicite, nous nous sommes intéressés à des méthodes de synthèse de contrôleurs optimaux pour des systèmes partiellement observés. En parallèle, de manière à réduire la complexité des algorithmes de synthèse de contrôleurs, nous regardons actuellement comment conserver la hiérarchie implicite des systèmes lors du calcul d'un contrôleur. Une étude visant à «unifier» la génération de test et la synthèse de contrôleurs a également débuté cette année.*

Sigali : plate-forme de vérification/synthèse basée sur les automates. Nous avons poursuivi le développement des méthodes de vérification et de synthèse basées sur les automates. L'approche est basée sur des modèles comportementaux *intensionnels* (encore appelés

[BJT99] F. Besson, T. Jensen, J.-P. Talpin, « Polyhedral Analysis for Synchronous Languages », *in: Static Analysis*, A. Cortesi, G. Filé (éditeurs), *Lecture Notes in Computer Science, 1694*, Springer, p. 51–68, 1999.

[MBLL00] H. Marchand, P. Bournai, M. Le Borgne, P. Le Guernic, « Synthesis of Discrete-Event Controllers based on the Signal Environment », *Discrete Event Dynamic System: Theory and Applications 10*, 4, octobre 2000, p. 347–368.

symboliques ou implicites), obtenus par abstraction booléenne à partir des spécifications en langage Signal. Les différents algorithmes tant pour la vérification que pour la synthèse ont été intégrés dans une nouvelle plateforme Sigali, écrite en Java, facilitant ainsi l'utilisation de cet outil. Cette API permettra à terme de diffuser plus largement nos techniques de vérification et de synthèse de contrôleur dans une philosophie «open», c'est-à-dire ouverte à d'autres langages de spécification. Notons pour finir sur ce point, que nous avons regardé, en collaboration avec le projet Aida (Laurence Rozé), des techniques symboliques pour le calcul d'un diagnostiqueur [22]. Ce calcul s'opère notamment en utilisant des techniques de réduction symboliques modulo une relation d'équivalence. Cette étude se poursuit à l'heure actuelle pour voir comment intégrer les techniques de *model-checking* pour caractériser les pannes dans un système.

Application de la synthèse de contrôleurs à la robotique. La conception des systèmes robotiques et de contrôle-commande est de plus en plus difficile, ainsi que leur programmation et leur opération. Ceci est dû à leur taille et complexité grandissante. C'est pour cela que leurs architectures de programmation et exécution temps-réel requièrent plus d'assistance à l'utilisateur et au concepteur, fondée sur des abstractions utilisables et le support d'outils. Nous avons regardé la programmation au niveau tâche en robotique et contrôle-commande, où les tâches encapsulent les lois de commande et nous avons considéré l'application de techniques de synthèse de contrôleurs discrets, intégrées dans le cadre d'un environnement de programmation au niveau tâche en robotique [25]. Nous nous intéressons actuellement au problème du contrôle de tels systèmes en modes dégradés par l'utilisation notamment de fonctions coûts. Cette étude est réalisée en collaboration avec Éric Rutten de l'équipe Bip (Inria Rhône-Alpes).

Commande optimale sous observation partielle. En considérant une modélisation du système sous forme d'un automate explicite, nous avons regardé le problème de la synthèse de contrôleur optimal appliquée à un système partiellement observé [21]. L'idée de la commande optimale est de calculer un contrôleur qui pilote un système de manière à forcer celui-ci à achever une tâche en minimisant un certain critère de performance. Des coûts sont ainsi affectés à chaque événement, induisant alors un coût sur les trajectoires. Le but du contrôleur est alors de restreindre le comportement du système de manière à ce que celui-ci n'emprunte que des trajectoires de coûts minimaux. Notre solution se fait en deux étapes : premièrement, nous dérivons du système partiellement observé un observateur, appelé *C-Observer,* qui «mémorise» une approximation du coût des trajectoires non-observables entre deux états observables du système. Une notion de coûts observables sur les trajectoires est définie à partir de cet observateur. Durant la seconde étape, nous utilisons l'algorithme de [SL98] pour synthétiser une sous-machine optimale du *C-Observer.* Ces travaux se sont conduits en collaboration avec Stéphane Lafortune de l'université du Michigan, Ann Arbor, MI, USA.

Synthèse de contrôleurs pour des systèmes à événements discrets hiérarchiques. Que ce soit en vérification ou en synthèse de contrôleurs, la méthodologie appliquée est le plus souvent la suivante : les systèmes à contrôler et/ou à vérifier sont à l'origine spécifiés de manière hiérarchique (analyse descendante en génie logiciel). Puis la synthèse et/ou la vérification s'appliquent sur le système mis à plat (toute la modularité verticale a été oubliée). Sachant que la complexité des algorithmes de calcul croît exponentiellement avec le nombre d'états des

[SL98] R. Sengupta, S. Lafortune, « An Optimal Control Theory for Discrete Event Systems », *SIAM Journal on Control and Optimization 36*, 2, Mars 1998.

systèmes mis en parallèle et imbriqués, il semble très important de chercher à garder cette hiérarchie lors du calcul des contrôleurs. Un premier modèle hiérarchique a été décrit. Il s'agit d'une généralisation des structures de Kripke Hiérarchiques développées par [AY98]. Le modèle hiérarchique (HFSM, pour *Hierarchical Finite State machine*) que nous considérons peut être caractérisé par une collection de structures imbriquées $\langle K_1, \ldots K_n \rangle$, où K_1 représente le plus haut niveau de la HFSM. À un niveau intermédiaire, la structure K_i est une HFSM, pour laquelle les états sont soit des «états ordinaires» soit des «macro-états b» qui sont constitués d'un ensemble de structures $(K_j)_{j \in J_b} \subseteq 2^{\langle K_{i+1}, \ldots, K_n \rangle}$, évoluant en parallèle. Chaque structure peut avoir plusieurs états initiaux (resp. finaux). Le comportement d'une structure est le suivant : lorsque le système transite dans un macro-état b, toutes les structures associées à b sont activées et initialisées dans un de leurs états initiaux. *A contrario*, la sortie d'un macro-état est synchronisée avec la fin des tâches associées aux différentes structures de b (i.e., chaque structure est dans un état final). Entre deux états (ordinaires ou macro), le comportement de la structure est similaire à celui d'un automate plat. Sur ce nouveau modèle, nous avons montré qu'il était possible d'utiliser les algorithmes classiques de synthèse sur chaque structure en partant du niveau le plus bas et en remontant au fur et à mesure dans la hiérarchie du modèle. Des algorithmes de synthèse de contrôleur pour des objectifs de contrôle traitant de l'invariance et l'atteignabilité ont ainsi été développés. De manière similaire, nous avons montré comment généraliser le problème de la commande optimale [SL98] dans ce nouveau cadre hiérarchique. Les études en cours visent d'une part à étendre le modèle sur lequel nous travaillons (synchronisation des structures sur des événements communs, multiples états finaux, possibilité de préemption, communication entre contrôleurs, etc.) et d'autre part à étendre l'expressivité des objectifs de contrôles. Dans ce cadre, nous recherchons comment spécifier des objectifs de contrôle à différents niveaux selon la «vue» que l'on a du système (e.g., objectif d'ordonnancement de tâches à haut niveau, objectifs de sécurité propres, plus bas dans la hiérarchie). L'utilisation de méthodes probabilistes sur ces systèmes est également envisageable.

Test et contrôle. Malgré l'effort de vérification formelle qui peut être fait sur la spécification (avant ou après synthèse), les implémentations du système peuvent contenir des erreurs. Les tests qui sont constitués de séquences de contrôle et d'observation entre l'implémentation et le testeur fournissent alors le moyen de valider ou d'invalider l'implémentation par rapport à une relation de conformité donnée. La génération automatique de tests consiste alors à générer automatiquement des tests à partir de la spécification et d'un objectif de test (qui peut être décrit par une propriété logique ou un automate). On peut voir que les notions de testeur et de contrôleur sont proches. Il y a cependant quelques différences (notion de contrôle, correction, expressivité des objectifs, existence de solutions, etc.). Malgré des différences, les problématiques sont proches. Dans les deux cas, il s'agit d'extraire des comportements d'une spécification pour les faire interagir avec un système sous contrôle et observation partielle. Une

[AY98] R. Alur, M. Yannakakis, « Model checking of hierarchical state machines », *in : Sixth ACM Symposium on the Foundations of Software Engineering, Software Engineering Notes, 23, 6*, ACM Press, p. 175–188, New York, 1998.

[SL98] R. Sengupta, S. Lafortune, « An Optimal Control Theory for Discrete Event Systems », *SIAM Journal on Control and Optimization 36*, 2, Mars 1998.

première étude[1] sur la corrélation test/contrôleur, nous a permis de décrire des algorithmes, basés sur ceux développés dans TGV (*Test Generation with Verification Technology*)[2], nous permettant de calculer des contrôleurs pour des systèmes partiellement observés (i.e., avec des événements inobservables et/ou internes). Ces algorithmes de synthèse de contrôleurs pourront par la suite être utilisés pour générer des cas de tests dans un cadre où le critère de contrôlabilité s'apparente à celui de la synthèse. On s'intéresse actuellement à trouver des relations reliant test et contrôle sur des sujets qui tournent autour des modèles, des relations entre spécification et implémentation et l'extension de l'expressivité des objectifs de tests.

modélisation d'architectures de communication

6.7 Mises en œuvre distribuées et modélisation d'architectures de communication

Participants : Albert Benveniste, Loïc Besnard, Abdoulaye Elhadji Gamatié, Thierry Gautier, Paul Le Guernic.

Mots clés : Signal, DC+, programmation synchrone, génération de code enfoui, code distribué, architecture hétérogène, compilation séparée, causalité, communication synchrone/asynchrone, description d'architecture, multi-tâche préemptif, OS temps réel.

Résumé : *La génération de code distribué pour les programmes synchrones pose deux grands types de difficultés. 1/ La compilation directe de code C (ou, plus généralement, de code séquentiel) n'est pas compatible avec une recomposition ultérieure avec d'autres modules. En d'autres termes, on ne peut pas compiler séparément (du moins de façon brutale) des programmes synchrones. 2/ Une architecture distribuée ne s'accomode pas, en général, de l'hypothèse de synchronisme parfait qui correspond au modèle de la programmation synchrone. Pour le premier problème, nous avons proposé une notion de «*tâche atomique*» qui constitue le grain maximal autorisant une compilation séparée avec réutilisation possible dans tout contexte. Pour le second problème, nous avons proposé les propriétés* d'endochronie *pour un programme synchrone, et* d'isochronie *pour un réseau de programmes synchrones. Ces deux propriétés garantissent une distribution correcte par construction, en s'appuyant uniquement sur les services d'une communication de type «send/receive» fiable.*

Sur ces bases, une méthodologie a été développée pour la génération de code distribué, qui implique : 1/ la spécification par l'utilisateur, au niveau du programme source, de la répartition du programme, 2/ l'application de transformations automatisées du code intermédiaire, qui contrôle la correction du partitionnement, 3/ la génération du code distribué correspondant aux différents processus et à leurs communications.

[1] Dans le cadre d'un stage de DEA.

[2] TGV est un outil de génération automatique de séquences de tests de conformité pour les protocoles développé dans l'équipe Pampa.

Nous avons pour objectif d'enrichir notre méthodologie actuelle de génération de code distribué sur une architecture donnée en utilisant le langage Signal pour modéliser et évaluer a priori l'architecture de communication supportant la mise en œuvre d'une application temps réel. Le développement d'applications critiques dans un langage temps réel synchrone tel que Signal permet en effet de disposer à la fois d'outils de vérification de propriétés de programmes et d'outils de génération automatique de code.

La méthodologie que nous proposons consiste à modéliser des architectures de communication en définissant en Signal un ensemble de primitives génériques sur la base de standards (ARINC dans le cadre du projet SafeAir), et à utiliser cette modélisation pour permettre l'évaluation temporelle de l'application décrite sur l'architecture d'implémentation. Les modèles considérés devront permettre d'étudier des ordonnancements préemptifs et, lorsque c'est possible, utiliser les résultats de méthodes d'analyse existantes (RMA...).

L'approche ARINC [Air97] peut être vue de la manière suivante : étant données une ou plusieurs applications, il est procédé à un découpage fonctionnel qui tient compte de l'espace mémoire et du temps disponibles. L'unité de ce partitionnement est appelée «partition» ARINC (c'est l'équivalent d'un programme dans le cas d'un environnement contenant une seule application). La «partition» quant à elle est découpée en d'autres entités élémentaires de travail dites «process».

Nous avons donc défini une bibliothèque Signal de modèles de composants de communication et synchronisation entre les «process» [28]. Ce sont d'une part, les mécanismes d'échange de messages (buffer, blackboard...) et ceux de synchronisation (sémaphores) et d'autre part, les services associés à chacun de ces composants. Actuellement, cette bibliothèque est en train d'être étendue avec d'autres modèles de composants (tâches, scheduler...) nécessaires à la description d'une application entière. Parallèlement, nous travaillons sur la définition d'une sémantique temps réel du langage Signal qui offrira un moyen d'évaluer les architectutres modélisées.

D'autre part, dans le cadre d'une collaboration avec Paul Caspi et Stavros Tripakis, de Verimag, nous avons examiné comment adapter notre méthodologie de génération de code distribué pour des architectures distribuées de type «mollement synchrone». Ces architectures sont constituées de modules capteurs/calculateur/actionneurs autonomes et répartis, reliés par un bus de terrain également autonome. Autonome veut dire ici que chaque module possède sa propore horloge et communique avec les autres selon un schéma non bloquant, les diverses horloges n'étant pas synchronisées. Ce type d'architecture est, par exemple, utilisé par EADS-Toulouse. Il s'agit d'une architecture dite «déclenchée par le temps» *(time-triggered)*, mais pas strictement au sens de Hermann Kopetz, car les horloges ne sont pas, dans notre cas, strictement synchronisées. Ce type d'architecture ne satisfait pas aux hypothèses requises pour une application directe de la méthodologie indiquée plus haut, ces hypothèses requises revenant à disposer d'un réseau de communication par FIFOs fiables, avec un mode de communication bloquant en lecture. Néanmoins nous avons pu montrer dans [23] qu'on pouvait émuler un réseau de FIFOs fiables au-dessus d'une architecture mollement synchrone, rendant ainsi possible l'application de nos méthodes à ce genre d'architecture. Enfin, suite à une étude plus appro-

[Air97] AIRLINES ELECTRONIC ENGINEERING COMMITTEE, « Avionics Application Software Standard Interface – ARINC Specification 653 », janvier 1997, Aeronautical Radio, INC. Maryland (USA).

fondie des travaux de Paul Caspi à Verimag dans le cadre du projet Esprit Crisys, ainsi qu'à la lecture de travaux sur la conception dite *latency insensitive* d'IP *(Intellectual Properties)* en hardware synchrone, nous avons pu réaliser une étude comparée des problèmes de répartition désynchronisée en hardware, en software avec une architecture de type «objets répartis», et en contrôle réparti temps-réel. Ceci fait l'objet d'une communication invitée au colloque EMSOFT'01, premier colloque sur les systèmes embarqués [18].

6.8 Modélisation de circuits et synthèse de matériel

Participants : Paul Le Guernic, Christophe Wolinski.

Mots clés : Signal, programmation synchrone, conception conjointe matériel/logiciel, composant matériel, synthèse de circuits, consommation des circuits.

Résumé : *Nous poursuivons notre étude dans le domaine de la synthèse comportementale de circuits digitaux à partir de graphe* GHDC *(format interne du compilateur Signal). Un graphe* GHDC *peut être obtenu soit à partir d'applications décrites en langage Signal à la suite d'un processus de compilation, soit à partir de langages impératifs tels que C, HardwareC,* VHDL *à la suite d'un processus de construction à deux niveaux. Dans ce graphe, le traitement et le contrôle sont représentés de façon uniforme, ce qui simplifie l'analyse globale. La synthèse est basée sur la transformation formelle du graphe* GHDC.

Les travaux poursuivis au cours de cette année, notamment dans le cadre de l'achèvement du travail de thèse de Jean-Christophe Le Lann [12], ont visé à mettre en avant les possibilités offertes par l'approche synchrone – et le langage Signal en particulier – dans la modélisation de circuits numériques, l'élaboration de simulateurs et la synthèse de matériel.

Nous avons souligné et exploité l'affinité existant entre les langages synchrones et les simulateurs cycles (*«cycle-based simulators»*). Ces simulateurs s'opposent aux simulateurs *event-diven* par interprétation, sur lesquels les langages comme VHDL et Verilog reposent traditionnellement : au lieu de maintenir la causalité des événements par un mécanisme d'échéancier, une analyse des dépendances est effectuée à la compilation, et un code approprié est généré (*«levelized compiled code»*). Ainsi, chaque élément du circuit n'est évalué qu'une seule fois par cycle d'horloge. Ceci se révèle extrêmement efficace pour des circuits présentant une forte activité. Lorsque l'activité diminue, la non-rémanence des simulateurs cycle se fait sentir : la simulation passe par des phases de réévaluation de portions de circuits inchangées par rapport au cycle précédent. Afin de profiter au mieux du calcul symbolique réalisé par le compilateur Signal, nous sommes passés par une série d'expériences, autour d'une chaîne de compilation nommée CycloTimic, qui ont montré que deux styles de modélisation au niveau portes étaient possibles : une première modélisation ne tient pas compte du fait précédent, mais permet d'accéder rapidement à un code exécutable (l'arbre d'horloges ne présente alors aucune hiérarchie). La seconde modélisation, plus souple, permet à l'inverse de hiérarchiser le contrôle du simulateur, en prenant en compte l'instant précédent. Ces travaux et une étude de la sémantique de VHDL ont permis d'appliquer les mêmes techniques à ce langage. Ainsi, une modélisation des listes de sensibilité des processus permet de travailler dans l'espace des événements de changements de

valeurs des signaux. Un prototype est toujours à l'étude. Nous avons également cherché à travers ces travaux à étudier les limites d'applicabilité de telles méthodes dont la cible idéale reste les circuits synchrones monocadencés. Nous pensons que la souplesse des modélisations présentées doit à terme pouvoir nous permettre d'accéder à des types de relations qui se rencontrent typiquement dans les circuits asynchrones (ou assimilés : multi-horloges, horloges générées in situ, *gated-clocks*, etc.).

Concernant le domaine de la synthèse de matériel, la modélisation précédente du langage VHDL se révèle par exemple très proche des travaux de Vemuri et Roy à l'Université de Cincinatti autour de l'outil de synthèse comportementale DSS ou de ceux de Kuchinski à Linköping sur Camad.

Cette année, notre attention a porté sur l'amélioration des outils de synthèse de haut niveau Codesis [15] que nous avons développés autour de Signal. Plus précisément nous avons modifié le cœur du système réalisant l'ordonnancement sous contraintes. Le nouveau module utilise la technique CLP (*Constraint Logic Programming*) pour optimiser l'utilisation des ressources physiques pendant la phase d'ordonnancement du graphe GHDC. Durant l'optimisation on utilise l'information telle que la hiérarchie des gardes et l'exclusivité des gardes pour pouvoir effectuer l'exécution spéculative et le partage des ressources. Le nouveau module donne une solution optimale (les outils prouvent que la solution est optimale) [20]. Le prototype du module a été réalisé et testé. Ce travail a été possible grâce à une coopération avec l'Université de Lund (Suède).

6.9 Techniques d'algèbre MaxPlus pour l'évaluation de performances

Participants : Albert Benveniste, Pierre Le Maigat.

Mots clés : évaluation de performance, algèbre (max, +), système synchrone/asynchrone.

Résumé : *Le but de ce travail est de développer les outils nécessaires à l'étude quantitative des processus synchrones et asynchrones. Il s'agit, grâce à un cadre formel relativement récent, les algèbres Max-Plus, d'étudier dans quelles mesures les systèmes temporisés possèdent des régimes stationnaires. Le calcul du comportement asymptotique d'un système temporisé nécessite une bonne compréhension des mécanismes mathématiques de ces algèbres qualifiées de «tropicales».*

Dans une collaboration avec Loïc Hélouët, du projet Pampa, il a été défini la notion de *«High-Level-Message-Sequence-Chart temporisé»*. Les HMSC forment un langage de description de scénarios de haut niveau, ces scénarios pouvant être définis de manière incomplète. Ils permettent, entre autres, de spécifier des protocoles de communication ; c'est pourquoi afin de pouvoir faire des estimations sur le débit de ces protocoles, nous avons étendu les définitions afin de prendre en compte les durées. C'est grâce à la notion d'automate d'ordres, définie par A. Benveniste, S. Gaubert et C. Jard, que les résultats de l'algèbre Max-Plus ont pu être appliqués avec succès. Ceci a permis l'évaluation quantitative des notions de trafic, débit ou charge d'un réseau, et d'appliquer notre étude au cas du protocole du bit alterné. Une analyse plus poussée sur les matrices à coefficient Max-Plus permet d'affiner les différentes notions de trafic en les

spécialisant à un canal, à un message particulier, à un ensemble de messages pertinent pour le protocole défini (ce qui a conduit à la décomposition des HMSC en éléments irréductibles)... On peut également, dans le cas d'une divergence de processus, faire des estimations sur la vitesse d'accroissement de la taille des buffers. Éliminer la divergence quand elle existe permet dans certains cas d'obtenir de manière effective un contrôleur qui optimise le trafic. Le problème dans le cas général est ouvert.

Nous développons maintenant cette approche algébrique pour les formalismes synchrones/ asynchrones, ce qui nous conduit à développer un calcul «polynomial» $(\max, \min, +)$ qui est une extension du calcul linéaire $(\max, +)$. Cette approche permet la modélisation du comportement temporel des réseaux de Petri. On montre que celui-ci est donné par le grand point fixe d'un système «polynomial». Nous avons mis en évidence la dualité qu'il existait entre la modélisation par des équations sur les compteurs et celle par des équations sur les dateurs.

7 Contrats industriels (nationaux, européens et internationaux)

7.1 Projet RNTL Expresso, convention n°2 01 C 0299 00 31307 01 1 (06/2001–05/2003)

Participants : Paul Le Guernic, Jean-Pierre Talpin.

Mots clés : Java temps réel, systèmes enfouis, approche synchrone, inférence de régions.

Résumé : *L'objectif du projet Expresso est de réaliser l'atelier mettant en jeu les composants nécessaires à l'environnement de développement d'applications temps réel Java pour des systèmes critiques enfouis ou embarqués et susceptibles d'être certifiés au sens des standards tels que le DO178-B.*

La communauté temps réel qui souhaite utiliser le langage Java pour ses qualités intrinsèques (orientation objet, sécurité, etc.) ne peut le faire aujourd'hui sans payer le prix du manque de performance du code interprété ou du non déterminisme de la machine virtuelle sous-jacente.

L'objectif du projet Expresso est de réaliser l'atelier mettant en jeu les composants nécessaires à l'environnement de développement d'applications temps réel Java pour des systèmes critiques enfouis ou embarqués et susceptibles d'être certifiés au sens des standards tels que le DO178-B.

Les composants retenus dans ce cadre sont la compilation et l'optimisation du *Java Byte code* vers le code binaire natif de la cible, une machine virtuelle Java décomposable de façon graduelle en un exécutif Java et des outils de vérification des propriétés temps réel et de l'ordonnancement des applications Java.

Les extensions temps réel du langage Java qui seront mises en oeuvre s'appuieront sur les standards émergeants du «Sun Community Process» et du J Consortium. Le développement de ces extensions sera rendu accessible en mode «logiciel libre».

Les membres partenaires de ce projet comprennent les industriels Thomson-CSF Detexis et EDF qui définissent les besoins et évaluent les résultats du produit, les sociétés de produits et services informatiques Aonix et Silicomp qui s'associent pour fabriquer la chaîne de compilation

et l'exécutif Java et en feront la commercialisation, les laboratoires Inria/Irisa et Verimag experts en techniques formelles qui complètent l'atelier logiciel avec les outils d'analyse temps réel.

L'environnement Java intégré (IDE) final sera disponible dans deux ans. Les applications ainsi produites s'exécuteront soit sur un système d'exploitation Posix, soit sur le noyau Raven pour machine nue dans le cas des systèmes certifiés ou fortement contraints.

7.2 Projet RNTL Acotris, convention n°2 00 C 0527 00 31307 01 1 (02/2001–07/2003)

Participants : Thierry Gautier, Paul Le Guernic.

Mots clés : temps réel, UML, Signal, SynDEx.

Résumé : *Le projet Acotris a pour objet d'aider à la conception d'applications temps réel embarquées par l'intégration de la méthodologie et des concepts issus des approches synchrones au formalisme standard* UML.

`http://www.acotris.c-s.fr`

Les partenaires du projet Acotris sont CS-SI, le CEA, EADS (Aérospatiale Matra Missiles), Sitia et l'Irisa (projets EP-ATR et Sosso).

L'objectif du projet est d'aider à la conception d'applications temps réel embarquées imposant un niveau de parallélisme important, en se basant sur un support de conception UML et les outils de génération, dimensionnement et placement des applications issus des approches synchrones, ou plus largement multi-horloges.

Pour atteindre cet objectif, le projet propose :

- d'annoter les descriptions UML par des propriétés complémentaires (ex. : annotations multi-horloges). La démarche envisagée repose sur l'utilisation des mécanismes d'extension normalisés pour obtenir un nouveau profil UML dédié à ce domaine applicatif. Nous pourrons ainsi contribuer à l'évolution de la norme UML ;
- de mettre en place une passerelle UML – Signal – SynDEx qui permettra à l'utilisateur de concevoir progressivement son application décrite en UML en s'appuyant sur les techniques formelles de validation ;
- de poursuivre cette conception jusqu'à la réalisation en utilisant des techniques de conception conjointe matériel/logiciel (*co-design*) avec une chaîne de développement et de simulation permettant d'évaluer différentes architectures multi-composant (processeurs et/ou circuits dédiés) et d'exécuter les applications en temps réel (SynDEx).

7.3 Projet IST SafeAir, convention n°1 00 C 0149 00 31307 00 5 (01/2000–06/2002)

Participants : Albert Benveniste, Loïc Besnard, Abdoulaye Elhadji Gamatié, Thierry Gautier, Paul Le Guernic.

Mots clés : système enfoui, méthode formelle, Statecharts, Scade, Sildex, Lustre, Signal, avionique, description d'architecture, multi-tâche préemptif, OS temps réel, vérification, validation de code.

Résumé : *Le projet IST SafeAir («Advanced Design Tools for Aircraft Systems and Airborne Software») a pour objectif de réaliser un environnement de développement pour les systèmes avioniques (ASDE), qui réponde aux besoins cruciaux en sûreté de fonctionnement pour les systèmes enfouis. Le projet SafeAir, qui prend la suite du projet Esprit Sacres, doit permettre aux concepteurs de systèmes critiques embarqués de réduire significativement le risque d'erreurs et le temps de conception. L'approche proposée s'appuie sur des outils industriels existants, notamment, Statemate, Scade, Simulink, et offrira de nouveaux outils pour la modélisation d'architectures, la vérification formelle, et la validation de code. Le langage Scade, basé sur Lustre, est utilisé comme langage pivot.*

`http://www.safeair.org`

Présentation générale

Le projet IST-1999-10913 SafeAir a débuté en janvier 2000. Il regroupe les partenaires suivants : Aérospatiale Matra Airbus, devenu Airbus France, DaimlerChrysler Aerospace Airbus, devenu Airbus Deutschland GmbH, IAI (Israël), Snecma Control Systems (France), Telelogic Technologies Toulouse (France), TNI (France), I-Logix (USA), Siemens (RFA), Offis (RFA), Inria (France), Weizmann Institute of Science (Israël).

L'environnement ASDE («Avionics Systems Development Environment») développé dans le projet SafeAir devra permettre :

- de réduire significativement l'effort de validation à l'intégration grâce à l'emploi de techniques de vérification formelle,
- de fournir une intégration des étapes de conception, depuis les outils de modélisation de niveau système jusqu'à une génération de code automatique respectant les standards DO-178B qui s'appliquent dans les systèmes embarqués critiques en avionique,
- d'offrir une approche permettant de prouver automatiquement la consistance du source et du code généré, ce qui permettrait de réduire considérablement les tests unitaires.

Sur le plan technique, l'architecture ASDE [29] est illustrée par la figure 2.

L'utilisateur de ASDE pourra construire ses modèles à partir de plusieurs composants, chaque composant étant conçu avec l'un des outils Sildex, Simulink, Statemate et Scade. Les

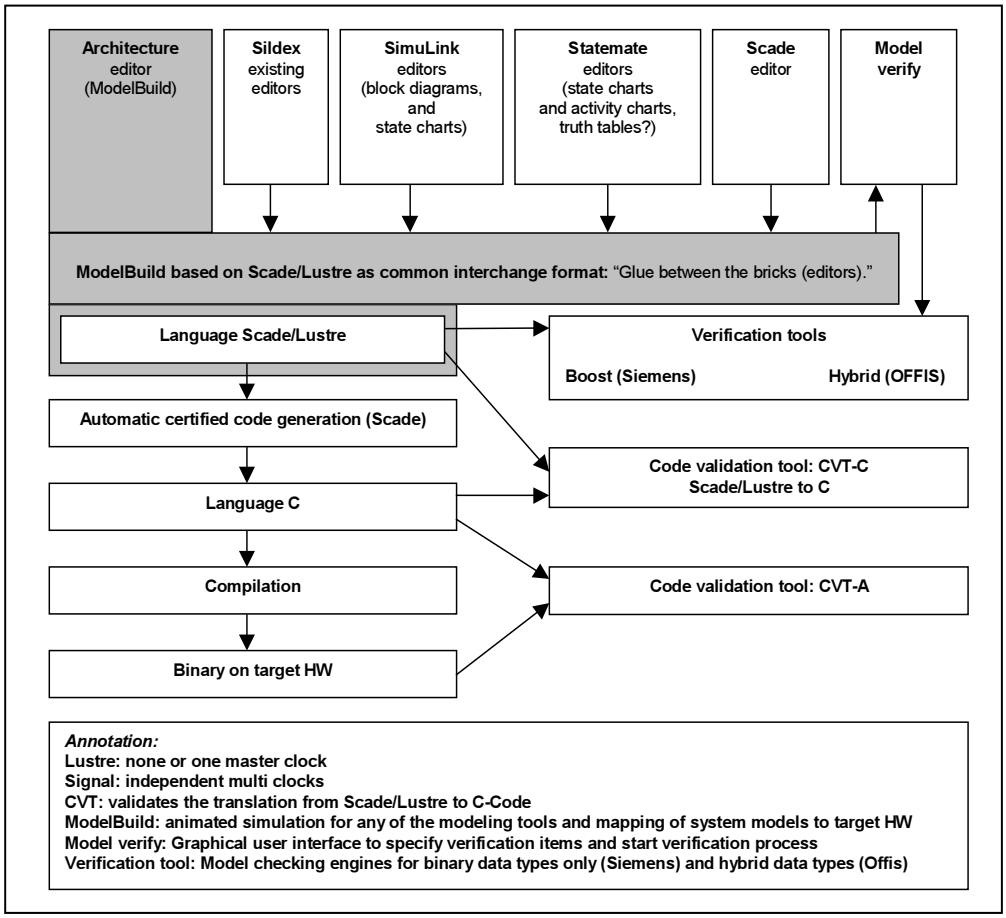

FIG. 2 – Architecture fonctionnelle de ASDE

communications entre ces composants peuvent être synchrones ou asynchrones. L'outil ModelBuild est un nouvel outil, basé sur Sildex, qui sera développé au cours du projet. Il permettra de décrire et de simuler l'intégration des différents composants d'un modèle sur une architecture. L'outil ModelVerify permettra de vérifier des propriétés du modèle global et de ses composants. L'intégration se fait par échange de fichiers Scade.

Activités de EP-ATR dans le cadre du projet SafeAir

Le projet EP-ATR est impliqué dans SafeAir principalement sur les aspects suivants :
- définition de l'architecture ASDE ;
- «convergence» Lustre–Signal ;
- définition et mise à disposition d'un ensemble de fonctionnalités de transformations correctes de programmes en vue de la vérification et de la génération de code ;
- études d'abstractions de programmes ;
- définition en Signal d'une bibliothèque de composants de communication pour une mise

en œuvre temps réel ;
- étude des caractéristiques temporelles d'une application pour une mise en œuvre temps réel.

SafeAir est utilisé comme cadre de développement et d'expérimentation pour la version «domaine public» de l'environnement Signal.

7.4 Projet Castor, convention n°1 00 C 0156 00 31399 01 2 (10/1999–04/2002)

Participants : Paul Le Guernic, Hervé Marchand.

Mots clés : sécurité des systèmes informatiques, modèles hiérarchiques, synthèse de contrôleurs.

Résumé : *Le projet Castor a pour but la réalisation d'outils pour l'aide à la conception d'architectures sécurisées d'un système d'information. L'autre partenaire Irisa de cette action est le projet Inria Lande.*

Les partenaires du projet Castor sont le Celar, Matra, AQL, TNI et l'Irisa. L'objectif général du projet Castor est de montrer la faisabilité de la modélisation de la sécurité d'un système informatique. En effet la complexité des systèmes d'information fait que la notion de sécurité répartie est difficile à mettre en œuvre (de même que son maintien au cours du temps). L'objectif des travaux consiste à étudier et à prototyper la modélisation d'architectures bâties par assemblage de composants suivant des règles d'intégration de la sécurité. Dans cette optique, nous nous occupons du modèle théorique, et plus précisément de l'aspect hiérarchique de ce modèle. L'application des techniques de synthèse de contrôleur à la génération automatique de chemins d'attaque d'un système dans un but de simulation est également à l'étude [30].

7.5 TNI

Mots clés : Signal, Sildex.

Résumé : *La société TNI, qui développe et commercialise l'environnement Sildex pour Signal, est un partenaire associé à nombre de nos activités.*

`http://www.tni.fr/tni/frame-sommaire.fra.html`

Nous collaborons étroitement avec la société TNI, qui assure l'industrialisation de Signal à travers l'environnement Sildex. Un axe essentiel de cette collaboration concerne la diffusion du synchrone en général, et en particulier des outils développés d'un côté et de l'autre autour de Signal.

Notre collaboration avec TNI s'effectue également au sein du projet Castor et du projet européen SafeAir.

8 Actions régionales, nationales et internationales

8.1 Actions internationales

8.1.1 Accueil de chercheurs étrangers

Le professeur B. Wadge, de l'University of Victoria, BC, Canada, a effectué un séjour d'une semaine en juillet 2001, dans le cadre d'une collaboration informelle sur Signal et la programmation intensionnelle.

9 Diffusion de résultats

9.1 Animation de la communauté scientifique

P. Le Guernic est membre du bureau exécutif du RNTL, «Réseau National de recherche et d'innovation en Technologies Logicielles».

A. Benveniste a été «co-chair» de «FEmSys'01 : Third Workshop on Formal Design of Safety Critical Embedded Systems» (Munich, RFA, mars 2001).

Ch. Wolinski a co-organisé une session spéciale «Modern Digital System Synthesis» à «SCI'2001» (Orlando, USA, juillet 2001). Il a fait partie du comité de programme de «EUROMICRO'01 : 27th EUROMICRO Conference» (Varsovie, Pologne, septembre 2001). Il a donné un séminaire invité au «Los Alamos National Laboratory» (USA).

H. Marchand a fait partie du comité de programme de «SCODES'2001 :Symposium on the Supervisory Control of Discrete Event Systems» (Paris, juillet 2001).

9.2 Enseignement universitaire

Les membres de l'équipe participent à divers titres à la formation d'étudiants à l'université et à l'Insa.

P. Le Guernic, S. Pinchinat, H. Marchand, Th. Gautier, B. Houssais et L. Besnard ont donné des cours de DEA, 5^{e} année Insa, DESS-Isa, Diic 2^{e} année consacrés à la programmation temps réel.

Dans le cadre des formations de troisième cycle, nous avons assuré l'encadrement d'étudiants stagiaires de DEA informatique : Benoît Gaudin, Valéry Tschaen (en collaboration avec le projet Pampa), ainsi que d'un étudiant en stage de fin d'études de *Information Technology Institute of Cairo* (Égypte) : Hany Louis Malika.

9.3 Participation à des colloques, séminaires, invitations

On pourra se reporter à la bibliographie pour la liste des colloques et congrès auxquels les membres de l'équipe ont participé.

10 Bibliographie

Ouvrages et articles de référence de l'équipe

[1] T. P. Amagbegnon, L. Besnard, P. Le Guernic, « Implementation of the Data-flow Synchronous Language Signal », *in : Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'95)*, ACM, p. 163–173, 1995, `ftp://ftp.irisa.fr/local/signal/publis/articles/PLDI-95:compil.ps.gz`.

[2] T. P. Amagbegnon, P. Le Guernic, H. Marchand, E. Rutten, « Signal- the specification of a generic, verified production cell controller », *in : Formal Development of Reactive Systems - Case Study Production Cell*, C. Lewerentz, T. Lindner (éditeurs), *Lecture Notes in Computer Science*, 891, Springer Verlag, p. 115–129, janvier 1995.

[3] A. Benveniste, B. Caillaud, P. Le Guernic, « From synchrony to asynchrony », *in : CONCUR'99, Concurrency Theory, 10th International Conference*, J. C. M. Baeten, S. Mauw (éditeurs), *Lecture Notes in Computer Science, 1664*, Springer, p. 162–177, août 1999.

[4] A. Benveniste, P. Le Guernic, C. Jacquemot, « Synchronous programming with events and relations : the Signal language and its semantics », *Science of Computer Programming 16*, 1991, p. 103–149.

[5] T. Gautier, P. Le Guernic, « Code generation in the SACRES project », *in : Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99*, F. Redmill, T. Anderson (éditeurs), Springer, p. 127–149, Huntingdon, UK, février 1999, `ftp://ftp.irisa.fr/local/signal/publis/articles/SSS-99:format_dist.ps.g%z`.

[6] A. Kountouris, P. Le Guernic, « Profiling of Signal Programs and its Application in the Timing Evaluation of Design Implementations », *in : Proc. of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, IEE, p. 6/1–6/9, Février 1996, `ftp://ftp.irisa.fr/local/signal/publis/articles/HWSWCRS-96:profiling.ps%.gz`.

[7] A. Kountouris, C. Wolinski, « High-level Pre-synthesis Optimization Steps using Hierarchical Conditional Dependency Graphs », *in : Proceedings of the EUROMICRO'99*, IEEE Computer Society Press, Milan, Italie, août 1999.

[8] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, « Programming Real-Time Applications with Signal », *in : Proceedings of the IEEE, 79*, 9, p. 1321–1336, Septembre 1991, `ftp://ftp.irisa.fr/local/signal/publis/articles/ProcIEEE-91:gen_lang.ps%.gz`.

[9] P. Le Guernic, T. Gautier, « Data-Flow to von Neumann : the Signal approach », *in : Advanced Topics in Data-Flow Computing*, J. L. Gaudiot, L. Bic (éditeurs), p. 413–438, 1991, `ftp://ftp.irisa.fr/local/signal/publis/articles/ATDFC-91:sem_distr.ps.g%z`.

[10] H. Marchand, M. Samaan, « On the Incremental Design of a Power Transformer Station Controller using Controller Synthesis Methodology », *in : FM'99 — Formal Methods*, J. M. Wing, J. Woodcock, J. Davies (éditeurs), *Lecture Notes in Computer Science, 1709*, Springer, p. 1605–1624, Toulouse, France, septembre 1999, `ftp://ftp.irisa.fr/local/signal/publis/articles/FM99:control_synth_appl%i.ps.gz`.

Thèses et habilitations à diriger des recherches

[11] F. F. Jiménez Fraustro, *Conception sûre des automatismes industriels : modélisation synchrone de langages d'automates programmables de la norme CEI-61131-3*, thèse de doctorat, Université de Rennes 1, IFSIC, mars 2001.

[12] J.-C. Le Lann, *Simulation et synthèse de circuits s'appuyant sur le Modèle Synchrone*, thèse de doctorat, Université de Rennes 1, IFSIC, mars 2002, à paraître.

[13] Y. Wang, *UML et technologie synchrone pour les systèmes réactifs distribués*, thèse de doctorat, Ifsic, Université de Rennes 1, décembre 2001.

Articles et chapitres de livre

[14] J.-R. Beauvais, E. Rutten, T. Gautier, P. Le Guernic, Y.-M. Tang, « Modelling Statecharts and Activitycharts as Signal equations », *ACM Transactions on Software Engineering and Methodology 10*, 4, 2001, à paraître.

[15] A. Kountouris, C. Wolinski, J.-C. Le Lann, « High-Level Synthesis Using Hierarchical Conditional Dependency Graphs in the CODESIS System », *EUROMICRO Journal of Systems Architecture on Modern Methods and Tools in Digital System Design*, 2001.

[16] H. Marchand, E. Rutten, M. Le Borgne, M. Samaan, « Formal Verification of Programs specified with Signal : Application to a power transformer Station Controller », *Science of Computer Programming 41*, 1, 2001, p. 85–104.

Communications à des congrès, colloques, etc.

[17] A. Benveniste, P. Bournai, T. Gautier, M. Le Borgne, P. Le Guernic, H. Marchand, « The Signal declarative synchronous language : controller synthesis & systems/architecture design », *in : 40th IEEE Conference on Decision and Control*, Décembre 2001.

[18] A. Benveniste, « Some synchronization issues when designing embedded systems from components », *in : First International Workshop on Embedded Software (EMSOFT'2001)*, *Lecture Notes in Computer Science, vol 2211*, Springer, août 2001.

[19] F. Jiménez-Fraustro, E. Rutten, « A synchronous model of IEC 61131 PLC languages in Signal », *in : Proceedings of the 13th Euromicro Conference on Real-Time Systems, ECRTS'01,* June 13th-15th, 2001, Delft, The Netherlands, p. 135–142, 2001.

[20] K. Kuchcinski, C. Wolinski, « Synthesis of Conditional Behaviors Using Hierarchical Conditional Dependency Graphs and Constraint Logic Programming », *in : Proceedings of EUROMICRO'01*, Varsovie, Pologne, septembre 2001.

[21] H. Marchand, O. Boivineau, S. Lafortune, « Optimal control of discrete event systems under partial observation », *in : 40th IEEE Conference on Decision and Control*, Décembre 2001.

[22] H. Marchand, L. Rozé, « Diagnostic de pannes sur des systèmes à événements discrets : une approche à base de modèles symboliques », *in : 13ème Congrès Francophone AFRIF-AFIA de Reconnaissance des Formes et Intelligence Artificielle*, Janvier 2002.

Rapports de recherche et publications internes

[23] A. Benveniste, P. Caspi, S. Tripakis, « Distributing synchronous programs on a loosely synchronous, distributed architecture », *Rapport de recherche n° 1289*, Irisa, décembre 1999, révisé 2001, `http://www.irisa.fr/sigma2/benveniste/pub/caspi_2001.ps.gz`.

[24] M. Nebut, S. Pinchinat, P. Le Guernic, « A Model for the Verification of Static Synchronous Data-flow Specifications », *rapport de recherche n° 1402*, Irisa, 2001, `http://www.irisa.fr/bibli/publi/pi/2001/1402/1402.html`.

[25] E. Rutten, H. Marchand, « Using discrete control synthesis for safe robot programming », *rapport de recherche*, INRIA, 2002, à paraître.

[26] J.-P. Talpin, « A simplified account of region inference », *rapport de recherche n° 4104*, Inria, septembre 2001, `http://www.inria.fr/rrrt/rr-4104.html`.

Divers

[27] « Component performance evaluation », février 2001, Advanced Design Tools for Aircraft Systems and Airborne Software (SafeAir).

[28] « Design and Implementation of Architecture Components », février 2001, Advanced Design Tools for Aircraft Systems and Airborne Software (SafeAir).

[29] T. Le Sergent, J.-L. Camus, F. Dupont, T. Gautier, P. Le Guernic, H. Hungar, K. Winkelmann, O. Shtrichman, M. Cohen, « ASDE V1.0 specification », mars 2001, Advanced Design Tools for Aircraft Systems and Airborne Software (SafeAir).

[30] H. Marchand, « Utilisation de Signal/Sigali pour la synthèse de séquences d'attaque et la simulation + Synthèse de systèmes hérarchiques », Rapport de convention CASTOR, 2001.