ACI sécurité ALIDECS 2004 – 2007
Rapport à mi-parcours

16 novembre 2006

Buts du projet : Proposer un atelier intégré pour le développement de composants embarqués sûrs. Cet atelier est fondé sur un ensemble de langages permettant de programmer à la fois le système et son environnement à l'exécution. Il est associé à des outils de génération de code, de vérification et de simulation.

Trois axes ont été développés dans le projet : un axe autour de langages ; un axe sur la comparaison avec les outils et modèles du domaine appuyée par des études de cas ; un axe sur la conception d'un modèle formel de *composants pour l'embarqué*.

Page web : `http://www-verimag.imag.fr/SYNCHRONE/alidecs`

1 Liste des participants et des équipes impliquées

LRI (Orsay) Marc Pouzet, (coordinateur) Professeur (MdC au LIP6 jusqu'à septembre 2005) ; Louis Mandel, doctorant (LIP6, jusqu'à octobre 2005) ; Florence Plateau, doctorante depuis septembre 2005, support MENRT.

VERIMAG (Grenoble) Paul Caspi (DR CNRS) ; Florence Maraninchi (Professeur ENSIMAG) ; Pascal Raymond (CR CNRS) ; Erwan Jahier (IR CNRS) ; Louis Mandel, post-doctorant jusqu'à octobre 2006 ; David Stauch, doctorant.

INRIA POPART (Grenoble) Pascal Fradet (CR INRIA) ; Alain Girault (CR INRIA) ; Gregor Goessler (CR INRIA) ; Massimo Tivoli (Post-doctorant depuis le 15 octobre 2005, financement INRIA) ; Gwenaël Delaval (Doctorant INPG depuis le 1 octobre 2004, bourse MENRT) ;

INRIA Mimosa (Sophia-Antipolis) Frédéric Boussinot (CR Ecole des Mines).

LAMI (Evry) Jean-Marc Delosme (Professeur) ; Bachir Djafri (Maitre de Conférences).

2 Langages

2.1 Lutin, un langage de modélisation de l'environnement

Les systèmes embarqués sont généralement destinés à entretenir une interaction continuelle avec leur environnement. Dans le cadre d'un atelier intégré de développement et de simulation, il est donc nécessaire de pouvoir décrire et simuler ces environnements. Les environnements sont généralement indéterministes, mais pas quelconques : ils sont connus par des propriétés de haut niveau que toute simulation réaliste doit satisfaire.

Une solution pour décrire les environnements serait d'utiliser des langages de programmation réactive (comme Lustre) en leur rajoutant des primitives pseudo-aléatoires pour simuler l'indéterminisme. Cette démarche n'est cependant pas très satisfaisante, car la distance est souvent

très grande entre les spécifications de haut niveau de l'environnement et un programme concret sensé les simuler.

C'est pourquoi Verimag suit la démarche inverse qui consiste à proposer des formalismes de description intrinsèquement indéterministes [19]. Plus précisément, Verimag érudit :

- un modèle exécutable simple, basé sur des automates finis et des relations d'entrées/sorties,
- un langage de haut niveau, « user-friendly », proposant des structures de contrôle sophistiquées et pouvant être compilé dans le modèle exécutable.

Dans le modèle de base, Lucky, le comportement est décrit par un automate fini indéterministe dont les transitions représentent les réactions atomiques. Chaque transition est étiquetée par :

- une relation entre les variables d'entrée/sortie qui doit être satisfaite au cours de la réaction ; par exemple, soit `x` une entrée et `a` une sortie, `x and (0.0 < a) and (a > 10.0)` stipule que `x` doit être vraie et `a` comprise entre 0 et 10. Dans une telle relation, les entrées, qui sont fournie par le système embarqué, sont incontrôlables. Il peut donc arriver que la relation soit insatisfiable, auquel cas la transition est *irréalisable* (c'est le cas dans l'exemple quand `x` est fausse) ;
- un poids, c'est-à-dire une indication numérique représentant la probabilité relative de prendre cette transition plutôt qu'un transition concurrente (une transition réalisable de poids 3 a $3/2 = 1,5$ fois plus de chance d'être choisie qu'une transition réalisable de poids 2).

L'algorithme de simulation est assez simple : dans un état courant, une transition est élue parmi toutes celle qui sont réalisables, en tenant compte de leurs poids relatifs. La contrainte associée est alors résolue et une solution particulière est choisie pour être envoyée au système embarqué. Si aucune transition n'est réalisable, l'exécution stoppe, sinon, elle continue à partir de l'état but de la transition.

Le modèle d'automate de Lucky est suffisamment riche pour décrire des comportements complexes. Cependant, programmer directement en automate explicite devient vite rédhibitoire. C'est pourquoi un langage de plus niveau est proposé. Les bases de ce langage, Lutin, sont présentées dans [20]. Comme dans Lucky, les atomes sont des réactions atomiques indéterministes, mais la structure de contrôle est construite à partir d'instructions inspirées par les expressions régulières : séquence, choix indéterministe et boucle indéterminée.

La nouvelle version du langage, développée dans le cadre d'Alidecs, est enrichie de nouvelles structures de contrôle : exécution concurrente, levée et interception d'exceptions. Le langage est aussi enrichi d'un couche fonctionnelle pour permettre une programmation modulaire. L'utilisateur peut par exemple construire et réutiliser ses propres structures de contrôle adaptées.

2.2 Lucid Synchrone

Le langage Lucid Synchrone combine le modèle synchrone de Lustre et des traits propres aux langages fonctionnels de la famille ML pour en augmenter l'expressivité et la modularité. Il est associé à une famille de systèmes de types (typage classique, calcul d'horloge, initialisation, etc.) donnant des garanties de sûreté sur le comportement du système (e.g., absence de blocage, exécution en temps et mémoire bornée).

La programmation par composant nécessite de pouvoir exprimer, dans un cadre unifié, le mélange des deux principaux styles de description utilisés dans les outils de l'embarqué : les descriptions flot-de-données *à la SCADE ou Simulink* et les automates *à la SyncCharts ou StateFlow*. Nous avons ainsi proposé une extension d'un noyau synchrone flot-de-données avec des machines à état dans l'esprit des *automates de modes* de Maraninchi & Rémond. L'extension proposée est pleinement compatible avec SCADE est sera intégrée à la prochaine version de l'outil.

Dans le cadre d'Alidecs, un nouveau compilateur de Lucid Synchrone a été développé au LRI (version 3). Ce compilateur intègre des traits nouveaux et les derniers résultats obtenus : descriptions de systèmes mixtes combinant automates et systèmes d'équations flot-de-données, mélange de flots et de signaux *à la Esterel* permettant de décrire dans un cadre uniforme des systèmes à contrôle prépondérant, traits d'ordre supérieur permettant de décrire des systèmes

reconfigurables dynamiquement, etc. Cette nouvelle version est distribuée depuis avril 2006 [1].

Ces travaux sont décrits dans [10, 13, 12, 11, 7].

2.3 Un système de type avec effets pour la distribution modulaire de programmes synchrones

Les langages synchrones permettent de décrire des systèmes fonctionnellement centralisés, où toutes les valeurs, entrées, sorties, ou fonctions intermédiaires, sont accessibles directement. Cependant, la plupart des systèmes embarqués sont de nos jours composés de plusieurs unités de calculs (et ne sont donc *pas* centralisés). Nous avons proposé une approche orientée langage permettant la description de systèmes réactifs fonctionnellement distribués. Nous avons étendu un noyau synchrone fonctionnel (à la *Lustre ou Lucid Synchrone*) avec des primitives pour la distribution de programmes. Ces primitives permettent au programmeur de décrire l'architecture du système en terme de sites symboliques représentant les diverses unités de calcul, ainsi que les liens de communications possibles entre ces sites, puis d'exprimer sur quel site les données doivent être calculées. La compilation de ce langage produit ensuite un programme réactif par site symbolique, où des instructions de communication ont été insérées aux bons endroits. Le système réactif distribué résulte de l'exécution simultanée de ces programme sur chacun des sites physiques. Un système de types spaciaux, basé sur les types avec effets, permet de vérifier la cohérence des annotations de distribution.

Ce travail est fait dans le cadre de la thèse de Gwenael Delaval dirigée par Alain Girault (INRIA PopArt) et Marc Pouzet (LRI).

2.4 FunLoft

Dans un cadre concurrent, il est naturel de demander que l'exécution d'un fragment de code séquentiel reste atomique, c'est-à-dire que la signification d'une séquence d'instructions exécutée par un composant parallèle reste invariante en présence d'autres composants parallèles. La séquence `x :=1 ; x :=x+1` est d'un point de vue séquentiel strictement équivalente à l'affectation `x :=2`. Ce n'est plus vrai dans un cadre concurrent, si une autre affectation peut s'intercaler entre les deux instructions de la première séquence. La préservation de la signification d'une séquence d'instructions dans un contexte concurrent est fondamentalement un problème d'atomicité ; c'est à ce problème que l'on s'est attaqué, dans le cadre des FairThreads [6], un modèle qui mélange *threads* coopératifs et *threads* préemptifs.

Dans un premier temps, nous avons considéré le langage Cyclone et réalisé une implémentation de Loft en Cyclone. Ce travail a fait l'objet d'un rapport de recherche INRIA [5]. La conséquence de cette étude a été d'abandonner l'idée de partir de Cyclone et de concevoir un langage permettant d'analyser les atomes, pour être capable de prouver leur atomicité.

Dans une deuxième phase, on a conçu, en collaboration avec F. Dabrowski (participant à l'ACI CRISS), un langage réactif implémentant une version simplifiée du modèle des FairThreads, avec un seul scheduler et la possibilité pour les threads de se délier temporairement. Nous avons défini un système de types et d'effets avec contraintes pour assurer l'étanchéité mémoire : un thread délié n'a accès qu'à sa mémoire privée et ne risque dont pas d'interférer avec les autres threads liés. Les threads déliés ne peuvent pas non plus se gêner entre-eux ce qui garantit la proriété d'atomicité recherchée. Ce travail sera présenté à un colloque sur les threads[14].

On a également commencé à implémenter une variante du langage, appelée FunLoft. Les points principaux de cette implémentation sont : (1) Types inférés. En particulier, l'utilisateur n'a pas à indiquer le statut des références créées. (2) Traduction en Loft/FairThreads en C.

Pour tester l'expressivité du modèle de programmation, on a principalement implémenté trois exemples :

– Automates cellulaires. Ces exemples sont interfacés avec la librairie SDL, pour l'affichage graphique. Ils reprennent des programmes Loft/C[2]. Dans ces exemples, chaque cellule est un

[1] `http://www.lri.fr/~pouzet/lucid-synchrone`

[2] `http ://www-sop.inria.fr/mimosa/rp/CellularAutomata`

thread. L'aspect mis en avant est principalement la possibilité de traiter des grands nombres de threads (dans le cas présent, 40000).
- Simulation de proies/prédateurs. L'aspect qui est mis en avant par cet exemple est la création dynamique : lorsque les prédateurs ont détruit les proies, de nouvelles proies sont automatiquement recréées.
- Exemple de producteurs/consommateurs.

2.5 ReactiveML

ReactiveML [3] est un langage de programmation généraliste fondé sur le modèle synchrone réactif de Boussinot et construit au dessus de OCaml. Il s'agit de proposer un langage de programmation permettant de décrire des systèmes où des composants réactifs peuvent être ajoutés ou retirés dynamiquement. Le langage mélange des traits d'ordre supérieur de ML, un modèle de la concurrence synchrone, et des aspects de création dynamique tels qu'on peut les trouver dans les langages asynchrones.

Le langage bénéficie d'une part des avantages des langages ML (ici OCaml) : inférence des types, types de donnée définissables par l'utilisateur et filtrage, gestion automatique de la mémoire et compilation efficace vers du code natif. D'autre part, les nouvelles constructions introduites en ReactiveML sont des constructions de haut niveau pour la description des comportements réactifs. Ainsi, au niveau du langage, il y a des constructions de composition parallèle et de communication entre processus par diffusion instantanée d'événements.

La sémantique formelle de ReactiveML (typage, sémantiques opérationnelles) est décrite dans [15, 18, 16]. Celle-ci prend en compte l'ensemble du langage et exprime, en particulier, les interactions entre les expressions ML et les constructions réactives.

Une implantation efficace a été réalisée. Les programmes ReactiveML sont compilés vers du code OCaml sans mécanisme de concurrence à l'exécution. Cette implantation repose sur des techniques d'ordonnancement décrite dans [16].

2.6 Larissa, aspects et composants pour les systèmes embarqués

Dans le cadre de la thèse de D. Stauch à Verimag, nous étudions la notion de programmation par aspects dans la famille des langages synchrones. Pour ne pas être trop dépendant de la forme d'un langage particulier, nous travaillons sur un noyau de base constitué d'automates de Mealy et d'opérations de composition simples (mise en parallèle, hiérarchie, diffusion synchrone).

La proposition est publiée dans [1, 23].

Une utilisation particulière de ce langage d'aspects pour la programmation par composants des interfaces de systèmes embarqués comme les équipements de l'électronique grand public est publiée dans [2]. L'idée est que les interfaces des différents modèles d'un tel équipement (une montre multi-fonctions à affichage numérique par exemple) sont toujours constitués des mêmes ingrédients. Elles varient très peu pour inclure plus ou moins de fonctions. Nous décrivons les interfaces de 3 modèles de la même marque à base de composants comportementaux et de composants qui sont en fait des aspects à tisser. Cette utilisation des aspects est particulièrement utile pour décrire les mécanismes de "raccourcis" dans les interfaces homme-machine qui disposent de peu de boutons : dans un certain mode, une fonction normalement accessible par une suite de pressions sur des boutons devient accessible avec un seul bouton.

Le fait de considérer les aspects comme des composants particuliers est rendu possible par les qualités du langage d'aspects que nous utilisons : le tissage est défini comme une opération à part entière, qui possède les mêmes bonnes propriétés que les autres opérations. En particulier, si l'on tisse le même aspect dans deux programmes équivalents, on obtient deux nouveaux programmes équivalents.

[3] Le langage est accessible à l'adresse `http://www-spi.lip6.fr/~mandel/rml`.

3 Outils/étude de cas

3.1 Simulation avec ReactiveML et Lucky

Plusieurs expériences de simulation de réseaux ont été réalisées en utilisant ReactiveML et son lien avec le langage Lucky.

Le premier simulateur (ELIP) a été fait en collaboration avec Farid Benbadis de l'équipe réseau du LIP6. C'est un simulateur de protocole de routage dans les réseaux mobiles ad-hoc avec positionnement géographique. L'implantation du simulateur est décrite dans [17]. Ce simulateur a permis de mettre en avant l'utilité des aspects dynamiques du langage pour simuler l'ajout et le retrait de nœuds du réseau.

Le second simulateur (Glonemo [22]) est un simulateur de réseaux de capteurs. Il a été réalisé pour évaluer l'influence des protocoles MAC sur la consommation d'énergie. Ainsi, contrairement au simulateur précédent qui faisait une simulation au niveau paquet, Glonemo est un simulateur à grain fin.

La particularité de Glonemo est d'utiliser la connexion entre ReactiveML et Lucky. Comme il a été montré dans [21], le prise en compte de l'environnement pour la simulation de réseaux de capteurs est nécessaire. Ainsi, dans le simulateur le réseau est modélisé en ReactiveML mais l'environnement qui active les capteurs est modélisé en Lucky.

Il est ressorti de ces expériences que le langage est bien adapté pour la simulation. Les structures de données utilisées dans les simulateurs (nœuds et paquets) ont pu être définies aisément. La composition parallèle synchrone s'est révélée bien adaptée pour simuler des réseaux asynchrones.

Par ailleurs, ces simulateurs ont permis l'évaluation des performances du langage. Par exemple ELIP a été comparé à un simulateur analogue écrit en OCaml (NAB [4]). Nous nous sommes également intéressé a la simulation de grands réseaux : le simulateur Glonemo peut ainsi simuler des systèmes de grande traille (140000 nœuds dans moins de 700 Megabytes). Enfin, pour vérifier la robustesse, nous avons exécuté des simulations pendant plus de 20 jours sans augmentation de la mémoire consommée.

3.2 Comparaison avec Fractal

Dans le cadre des travaux de Verimag sur un modèle de composants pour l'embarqué (voir paragraphe 4.2 ci-dessous), nous avons fait une expérience de codage dans le modèle Fractal. L'idée est de se plier aux contraintes d'un modèle à composants qui a fait ses preuves, pour valider notre notion de composant.

Fractal n'étant pas prévu pour la description de comportements parallèles, cet aspect doit être *codé*. Pour coder les différents modes de comportement parallèle des composants, nous adoptons le principe de Ptolemy : un ensemble de composants qui doivent fonctionner en parallèle sont rassemblés dans un super-composant en compagnie d'un composant particulier qui correspond au "directeur" de Ptolemy et exprime la sémantique du parallélisme à cet endroit-là.

Contrairement à Ptolemy, toutefois, nous ne donnons pas un catalogue de directeurs non liés les uns aux autres. Nous préférons exprimer plusieurs directeurs (synchrone, asynchrone, GALS, etc.) en termes d'opérations de base plus fines. Un composant élémentaire a des entrées et des sorties de données, et des entrées et des sorties de contrôle. Les composants sont liés par des "fils" qui n'expriment a priori ni synchronisation, ni mémorisation. C'est le composant directeur qui décide ce qui se passe sur les fils, et comment les différents composants sont sollicités lors d'une exécution parallèle. Les composants ont toujours la même forme, quel que soit le directeur à appliquer, et les directeurs s'expriment sous forme de petits programmes dont les opérations de base sont : activation des différents composants par leurs entrées de contrôle, gestion de la mémoire éventuellement correspondant aux fils, etc.

Les modèles Fractal pour le synchrone (sémantique de Lustre ou des circuits synchrones), l'asynchrone (processus multiples sur mono-processeur avec mémoire partagée) et les GALS ont été développés.

[4] http ://nab.epfl.ch

Les détails peuvent être trouvés dans [4].

3.3 Mise en œuvre de la tolérance aux fautes par des transformations automatiques de programmes

Nous avons proposé une approche formelle pour rendre automatiquement tolérant aux fautes un système initialement non tolérant aux fautes. Le système initial est constitué d'un ensemble de tâches périodiques indépendantes, ordonnancées sur un ensemble de processeurs à silence sur défaillance connectés par un réseau de communication parfaitement fiable. Notre but est que, en supposant la présence d'un processeur supplémentaire, le système final tolère une faute à la fois d'un processeur (permanente ou non). La détection des fautes est réalisée grâce à du « heart-beating » périodique, et le masquage par des points de reprise périodiques (« checkpointing & rollback »). Le principe de notre méthode est de transformer le code exécuté par chaque tâche pour qu'il mette en œuvre ces techniques. Chaque transformation est formellement représentée par des règles de transformation. Cette approche formelle illustre l'intérêt de la séparation des préoccupations et nous a permis de démontrer formellement que le système final tolère une faute tout en continuant à satisfaire des contraintes de temps d'exécution maximal [3].

En prenant cette approche comme point de départ, nous travaillons actuellement à la définition d'un langage d'aspects permettant de tisser différentes techniques de tolérance aux fautes dans des programmes temps-réel.

4 Modèles de composants pour l'embarqué

4.1 Synthèse d'adaptateurs pour composants temps-réels

Aujourd'hui, un nombre croissant de systèmes logiciels sont construits à partir de composants réutilisables ou sur étagère (« off-the-shelf »). La construction d'un système temps-réel à partir de tels composants soulève plusieurs problèmes, souvent liés à des questions de compatibilité, de communication et de qualité de service.

Afin de répondre à ces problèmes, nous avons développé un modèle à composants légers qui interagissent selon un modèle de flot de données. Chaque composant déclare des ports d'entrée et de sortie. Les ports des composants sont connectés par des canaux de communication synchrones. Dans ce cadre, une *interface* de composant est une description formelle du protocole d'interaction du composant avec son environnement attendu, en termes de séquences d'action d'écriture et de lecture. Le développeur d'un composant peut spécifier des contraintes de qualité de service comme la latence et les durées des écritures et lectures, et la fréquence d'activation du composant (son horloge). Nous avons formalisé le langage de spécification des composants, appelé DLiPA ; sa sémantique et celle du modèle à composants, basé sur une approche d'algèbre de processus.

Afin de combiner des composants incompatibles (du fait, par ex., d'horloges, de latences, de durées ou de protocoles incompatibles), nous synthétisons des *adaptateurs*. Un adaptateur est placé entre (au moins) deux composants communicants et incompatibles. Un adaptateur peut être vu comme un composant pourvu d'une mémoire tampon permettant d'harmoniser (par ex. retarder) les communications entre les autres composants. Chaque adaptateur est dérivé automatiquement de la spécification des interfaces des composants à harmoniser. La synthèse d'adaptateur facilite la réutilisation de composants et la construction incrémentale de systèmes.

Ce processus a été formalisé dans la théorie des automates et des réseaux de Pétri. Nous avons appliqué cette approche à deux études de cas : un système de contrôle d'échange de chaleur et un contrôleur adaptatif de vitesse de croisière.

4.2 Un modèle de composants pour le prototypage virtuel des systèmes embarqués

4.2.1 Contexte des travaux

Dans l'industrie des systèmes embarqués, on trouve plusieurs approches qui relèvent de la conception par composants, au sens large. La plus ancienne est à chercher dans le domaine du matériel : les industriels développement des "IP" (pour "intellectual property") qui constituent des briques de base pour la construction de systèmes plus complexes. L'évolution de la conception des circuits a conduit aux outils de prototypage virtuel comme SystemC, qui permet de décrire un système sur puce comme un assemblage d'IPs et de moyens de communication, à divers niveaux d'abstraction. SystemC est clairement dédié à une approche par composants, et permet d'assembler des composants matériels et des composants logiciels.

Dans le domaine des logiciels embarqués, la situation est moins claire, mais il existe tout de même des approches que l'on peut ranger dans les approches composants. Par exemple, les systèmes de contrôle embarqué critique, qui sont des solutions informatiques à des problèmes d'automatique, sont souvent conçus en Simulink, et en assemblant des "boîtes" de diverses bibliothèques.

Il est intéressant de remarquer que ces approches industrielles reposent sur des environnements de *prototypage virtuel*, dans lesquels on assemble des composants plus ou moins détaillés pour construire un prototype exécutable très tôt.

De manière générale, ces approches industrielles manquent de bases solides qui pourraient permettre d'analyser les systèmes construits, de valider les composants individuellement, etc.

4.2.2 État de l'art en recherche

Dans le monde académique, de nombreux travaux de recherche portent sur la notion de composant pour systèmes embarqués. Il est en général reconnu que la difficulté vient de l'hétérogénéité de ces systèmes. D'autre part il faut être capable de décrire les composants à la fois par leur propriétés *fonctionnelles*, et par leur propriétés dites non fonctionnelles (ou parfois "extra-fonctionnelles") comme la consommation d'énergie.

Les travaux existants sont nombreux. Citons l'outil de modélisation hétérogène Ptolemy (de E. Lee à Berkeley), qui permet de décrire des systèmes en utilisant plusieurs modèles de calcul, ou "MoCs" (models of computation). C'est un outil assez abouti, mais dans lequel il est difficile de comprendre la sémantique des différents MoCs et surtout les liens entre ces MoCs.

Il existe également des travaux plus théoriques, et donc plus loin des besoins d'une approche de prototypage virtuel. C'est le cas du modèle BIP (Behavior-Interaction-Priorities) de J. Sifakis.

Enfin il existe des approches par composants formalisées, comme Fractal et le Kell calculus, mais ils ne sont pas dédiés à la modélisation fine du temps et des mécanismes de concurrence, ce qui rend leur usage difficile pour les systèmes embarqués très hétérogènes.

4.2.3 Les travaux dans le cadre d'Alidecs

Le master recherche de Tayeb Bouhadiba en 2005-2006 à Verimag [4] porte sur ce thème. Tayeb a réalisé une étude de plusieurs modes de comportement parallèle (synchrone, asynchrone et GALS), en s'attachant à identifier une structure en composant générique, et en décrivant ces trois modes en termes d'opérations plus élémentaires. Par ailleurs il a implanté ses modèles en Fractal, pour valider l'approche composant.

Le point central de ses travaux est d'essayer de trouver un dénominateur commun simple à tous les modes de composition nécessaires entre composants embarqués.

4.3 Modèle N-synchrone

Dans le domaine de l'embarqué, on rencontre de nombreuses applications mêlant à la fois calcul intensif, contraintes de temps-réel dur et besoin de sûreté. Le cas le plus représentatif est celui du traitement vidéo (TV grand public, vidéo médicale). Ces applications demandent une interaction

forte entre des spécialistes de domaines différents (analyse numérique, automatique, architecture, parallélisme, compilation et optimisation). La caractéristique commune de ces systèmes est de s'exécuter sur une architecture mêlant logiciel/matériel et où les composants communiquent par des *buffers*. Le problème de la spécification des différents composants du systèmes et la garantie par construction de propriétés sur le comportement de l'ensemble (e.g., taille bornée des *buffers* et calcul de leur taille, déphasage ou gigue) reste largement ouvert.

Le modèle de description couramment utilisé dans ce domaine est celui des réseaux de Kahn. En étudiant les modèles de description du domaine nous avons introduit un modèle de synchronisme dit N-synchrone. Ce modèle est fondé sur un assouplissement du modèle synchrone traditionnel *à la Lustre*. Dans ce modèle, il est possible de composer des flots non strictement synchrones dès lors que ceux-ci peuvent être synchronisés par l'introduction d'un *buffer* dont la taille peut être calculée statiquement. Ainsi, les programmes synchrone Lustre correspondent à des réseaux de Kahn 0-synchrones. Ce modèle est également à rapprocher du modèle des *Synchronous Data-Flow* d'Edward Lee (et intégré à l'outil Ptolemy) mais est strictement plus expressif car permettant de décrire à la fois des systèmes périodiques et non périodiques. L'objectif est donc ici d'obtenir un modèle de composants bien adapté au domaine du traitement vidéo et de l'intégrer dans un langage de programmation. Ce modèle doit permettre une composition souple de systèmes faiblement synchronisés, de calculer automatiquement les tailles de *buffers* et de générer automatiquement du code directement exécutable.

Ce travail est fait dans le cadre de la thèse de Florence Plateau au LRI et est publié dans [8, 9]

Références

[1] K. Altisen, F. Maraninchi, and D. Stauch. Aspect-oriented programming for reactive systems : a proposal in the synchronous framework. *Science of Computer Programming*, 2006.

[2] K. Altisen, F. Maraninchi, and D. Stauch. Larissa : Modular design of man-machine interfaces with aspects. In *Fifth International Symposium on Software Composition (an ETAPS satellite event)*, March 2006.

[3] T. Ayav, P. Fradet, and A. Girault. Implementing fault-tolerance in real-time systems by program transformations. Research report 5919, Inria, May 2006.

[4] Tayeb Sofiane Bouhadiba. Modèle à composants pour systèmes embarqués, une expérience en fractal. Master report, Université Joseph Fourier, Grenoble, June 2006.

[5] F. Boussinot. *Loft+Cyclone*. Inria research report, RR-5680, 2005.

[6] F. Boussinot. FairThreads : mixing cooperative and preemptive threads in C. *Concurrency and Computation : Practice and Experience*, vol 18 pp 445-469, 2006.

[7] Paul Caspi, Grégoire Hamon, and Marc Pouzet. *Systèmes Temps-réel : Techniques de Description et de Vérification – Théorie et Outils*, chapter Lucid Synchrone, un langage de programmation des systèmes réactifs, pages 217–260. Hermes International Publishing, 2006. A paraître.

[8] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. Synchroning Periodic Clocks. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[9] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-Synchronous Kahn Networks : a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[10] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[11] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. Submitted to publication, May 2006.

[12] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[13] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :245–255, August 2004.

[14] F. Dabrowski and F. Boussinot. Cooperative threads and preemptive computation. In *Workshop on Multithreading in Hardware and Software : Formal Approaches to Design and Verification*. TV'06, a FLoC'06/CAV Affiliated Workshop, Seatle, August 2006.

[15] Louis Mandel et Marc Pouzet. ReactiveML, un langage pour la programmation réactive en ml. In *Journées Francophones des Langages Applicatifs (JFLA)*, Obernai, France, Mars 2005. INRIA.

[16] Louis Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006.

[17] Louis Mandel and Farid Benbadis. Simulation of Mobile Ad-Hoc Networks in ReactiveML. In Electronic Notes in Theoretical Computer Science, editor, *Synchronous Languages, Applications, and Programming (SLAP)*, 2005. available at `www-spi.lip6.fr/~mandel/rml`.

[18] Louis Mandel and Marc Pouzet. ReactiveML, a Reactive Extension to ML. In *ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*, Lisboa, July 2005.

[19] P. Raymond, E. Jahier, and Y. Roux. Describing and executing random reactive systems. In *SEFM 2006, 4th IEEE International Conference on Software Engineering and Formal Methods*, Pune, India, September 2006.

[20] Pascal Raymond and Yvan Roux. Describing non-deterministic reactive systems by means of regular expressions. *Electr. Notes Theor. Comput. Sci.*, 65(5), April 2002.

[21] Ludovic Samper, Florence Maraninchi, Laurent Mounier, Erwan Jahier, and Pascal Raymond. On the importance of modeling the environment when analyzing sensor networks. In *Proceedings of International Workshop on Wireless Ad-Hoc Networks 2006 (IWWAN 2006)*, page 7, New York, United States, June 2006.

[22] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. GLONEMO : Global and accurate formal models for the analysis of ad-hoc sensor networks. In *Proceedings of the First International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06)*, Nice, France, May 2006.

[23] D. Stauch, K. Altisen, and F. Maraninchi. Interference of Larissa aspects. In *FOAL : Foundations of Aspect-Oriented Languages workshop (an AOSD'06 satellite event)*, March 2006.