Quentin QUADRAT
quadra_q, UID 17115, promo 2007

EPITA
École Pour l'Informatique et les Techniques Avancées

RAPPORT DE STAGE

Aide à la conception de noyau applicatif pour le logiciel SynDEx

1er Septembre, 31 Décembre 2005

Supervisé par

Yves SOREL
INRIA - Domaine de Voluceau - Rocquencourt B.P.105
78153 Le Chesnay Cedex - France
Tél : (1) 39 63 52 60 - email : Yves.Sorel@inria.fr

INRIA
Institut National de Recherche en Informatique et en Automatique

---

Remerciements

Je tiens à remercier tout d'abord Yves Sorel, Directeur de Recherche à l'INRIA, pour m'avoir accueilli dans son équipe durant ce stage et m'avoir confié ce sujet très intéressant et particulièrement riche. J'ai apprécié son encadrement rigoureux et souple.

Je souhaite aussi remercier des membres de l'équipe AOSTE. Plus précisément :

- Christophe Gensoul et Patrick Meumeu pour avoir pris, sans faute, le café (ou le thé) matin, midi et soir tous les jours ;
- Daniel de Rauglaudre pour son humour (parfois douteux) ;
- Nicolas Pernet pour être resté calme même après le passage de notre (bien aimé ?) Sebastien Cornuejols ;
- Cyril Faure pour son professionnalisme "hard-core gamer" ;
- les autres personnes que, malheureusement, je ne connais pas bien.

Je remercie également notre assistante de projet, Nelly Maloisel, pour sa bonne humeur et l'aide ponctuelle qu'elle a pu me donner. Plus généralement, je remercie toute l'équipe pour la bonne humeur qui y régnait, ce qui a permit de travailler dans des conditions agréables. Je souhaite un grand courage à Patrick pour le commencement de sa thèse sur l'ordonnancement et un grand courage à Christophe, puisque, maintenant, il se retrouve seul contre le terrible Daniel (et par lequel je laisse à ce dernier le message suivant : "Non, le fonctionnement de SynDEx n'est pas fourni en option !").

Table des matières

Introduction

Ce rapport présente le travail réalisé dans le cadre du stage de fin de tronc commun de l'EPITA. Ce stage de 4 mois intitulé "Aide à la conception de noyau applicatif pour le logiciel SynDEx" a été réalisé à l'INRIA sous la direction de Yves Sorel, responsable du projet AOSTE dans lequel le logiciel SynDEx [3] a été développé.

Ce logiciel de CAO met en oeuvre la méthodologie AAA pour le prototypage rapide et l'optimisation de la mise en oeuvre d'applications distribuées temps réel embarquées. A partir d'un algorithme et d'une architecture donnés sous forme de graphe SynDEx génère une implémentation distribuée de l'algorithme en macro-code m4.

Le but de ce stage a été d'étudier et de développer un éditeur de noyaux applicatif. Cet éditeur intégré dans l'interface graphique de SynDEx, facilite en automatisant certaines tâches fastidieuses, la création de noyaux d'exécutif. Avant ce travail, l'utilisateur devait écrire pour chaque opération un code source dans le langage choisi et écrire aussi le squelette en langage m4 qui entoure ce code source. Maintenant, seul le code source reste à écrire, le squelette étant généré automatiquement par l'Éditeur de Code.

Ce rapport se compose de deux parties, chacune divisée principalement en deux chapitres. Dans la première partie, on présente d'abord l'INRIA et le projet AOSTE, puis on rappelle le contexte du temps réel embarqué dans lequel s'inscrit le stage et présente le logiciel SynDEx. Dans la deuxième partie, d'écrivant le travail réalisé pendant ce stage, le premier chapitre est consacré au développement de l'éditeur de noyau applicatif pour SynDE alors que le deuxième présente une application complète, illustrant le développement précédent et montrant le potentiel de SynDEx.

Première partie

Accueil et contexte

Chapitre 1

Structure d'accueil

1.1 L'INRIA Rocquencourt

Faisant suite à l'IRIA créé en 1967, l'INRIA est un établissement public à caractère scientifique et technologique (EPST) placé sous la double tutelle du ministre chargé de la Recherche et de l'Industrie.

Les missions qui lui ont été confiées sont :
- entreprendre des recherches fondamentales et appliquées,
- réaliser des systèmes expérimentaux,
- organiser des échanges scientifiques internationaux,
- assurer le transfert et la diffusion des connaissances et du savoir-faire,
- contribuer à la valorisation des résultats de recherches,
- contribuer, notamment par la formation, à des programmes de coopération avec des pays en voie de développement,
- effectuer des expertises scientifiques.

L'objectif est donc d'effectuer une recherche de haut niveau, et d'en transmettre les résultats aux étudiants, au monde économique et aux partenaires scientifiques et industriels.

L'INRIA accueille dans ses 6 unités de recherche situées à Rocquencourt, Rennes, Sophia Antipolis, Grenoble, Nancy et Bordeaux, Lille, Saclay et sur d'autres sites à Paris, Marseille, Lyon et Metz, 3 500 personnes dont 2 700 scientifiques, issus d'organismes partenaires de l'INRIA (CNRS, universités, grandes écoles) qui travaillent dans plus de 120 "projets" (ou équipes) de recherche communs. Un grand nombre de chercheurs de l'INRIA sont également enseignants et leurs étudiants (environ 950 préparent leur thèse dans le cadre des projets de recherche de l'INRIA).

Les chercheurs en mathématiques, automatique et informatique de l'INRIA collaborent dans les cinq thèmes suivants :

1. systèmes communicants,
2. systèmes cognitifs,
3. systèmes symboliques,
4. systèmes numériques,
5. systèmes biologiques.

Un projet de recherche de l'INRIA est une équipe de taille limitée, avec des objectifs scientifiques et une thématique relativement focalisés, et un chef de projet qui a la responsabilité mener et coordonner les travaux de l'équipe. Toutes ces équipes sont très souvent communes avec des établissements partenaires.

Le sujet de ce stage s'inscrit dans les activités du projet AOSTE : Modèles et Méthodes pour l'Analyse et l'Optimisation des Systèmes Temps-Réel Embarqués. Ce projet est bilocalisé à Rocquencourt et Sophia Antipolis. La partie située à Rocquencourt s'intéresse plus particulièrement à l'optimisation des systèmes distribués temps réel embarqués.

1.2 Projet AOSTE Rocquencourt

AOSTE est l'acronyme pour modeling Analysis Optimisation of Systems with real-Time and Embedded constraints.

Les travaux de l'équipe concernent quatre axes de recherche :

- la modélisation de tels systèmes à l'aide de la théorie des graphes et des ordres partiels,
- l'optimisation d'implantation à l'aide :
  - d'algorithmes d'ordonnancement temps réel dans le cas monoprocesseur,
  - d'heuristiques de distribution et ordonnancement temps réel dans le cas multicomposant (réseau de processeurs et de circuits intégrés),
- les techniques de génération automatique de code pour processeur et pour circuit intégré, en vue d'effectuer du co-développement logiciel-matériel,
- la tolérance aux pannes.

Tous ces travaux sont réalisés avec l'objectif de faire le lien entre l'automatique et l'informatique en cherchant à supprimer la rupture entre la phase de spécification/simulation et celle d'implantation temps réel, ceci afin de réduire le cycle de développement des applications distribuées temps réel embarquées.

Ils ont conduit d'une part à une méthodologie appelée AAA (Adéquation Algorithme Architecture) et d'autre part à un logiciel de CAO niveau système pour l'aide à l'implantation de systèmes distribués temps réel embarqués, appelé *SynDEx*.

Chapitre 2

Système temps réel

2.1 Systèmes réactifs temps réel embarqués

On s'intéresse dans ce document à la programmation de systèmes informatiques pour des applications de commande et de traitement du signal et des images, soumises à des contraintes temps réel et d'embarquabilité [1]. Dans ces applications, le système commande son environnement en produisant, par l'intermédiaire d'actionneurs, une commande qu'il calcule à partir de son état interne et de l'état de l'environnement, acquis par l'intermédiaire de capteurs.

Les systèmes informatiques étant numériques, les signaux d'entrée acquis par les capteurs, ainsi que ceux de sortie produits par les actionneurs, sont discrétisés (échantillonnage-blocage-quantification), aussi bien dans l'espace des valeurs que dans le temps. La précision de la commande dépend de la résolution de cette discrétisation.

Réagir trop tard peut conduire à des conséquences catastrophiques pour le système lui-même ou son environnement. Une analyse mathématique utilisant la théorie de la commande permet de déterminer d'une part une borne supérieure sur le délai qui s'écoule entre deux échantillons (cadence), et d'autre part une borne supérieure sur la durée du calcul (latence) entre une détection de variation d'état de l'environnement (stimulus) et la variation induite de la commande (réaction).

En plus de ces contraintes temps réel, l'application est soumise à des contraintes technologiques d'embarquabilité et de coût, qui incitent à minimiser les ressources matérielles (architecture) nécessaires à sa réalisation (l'architecture peut être composée de plusieurs processeurs, et de circuits spécialisés, pour satisfaire les contraintes temps réel).

Dans ce document, comme nous nous intéressons uniquement aux processeurs, plutôt qu'aux circuits intégrés spécialisés, l'implantation de l'algorithme sur l'architecture consiste donc à traduire (coder) l'algorithme de commande en programmes à charger dans les mémoires des processeurs pour que ceux-ci les exécutent.

2.2 Parallélisation

Pour une architecture monoprocesseur, l'algorithme serait traduit en un seul programme, c'est-à-dire codé en un ensemble d'instructions exécutées séquentiellement par le séquenceur d'instructions du processeur. Pour une architecture multiprocesseur, composée de plusieurs séquenceurs d'instructions opérant en parallèle, ainsi que de médias de communication leur permettant d'échanger des données, il faut partitionner l'ensemble des instructions en fonction du nombre de séquenceurs d'instructions, allouer un séquenceur d'instructions à chacun des sous-ensembles d'instructions et enfin ajouter des instructions de synchronisation et de trans-

fert de données, et leur allouer des médias de communication, pour supporter les dépendances de données entre les instructions de l'algorithme qui sont exécutées par des séquenceurs d'instructions différents.

Un partitionnement simple, par découpage linéaire du programme séquentiel en étapes successives exécutées chacune par un séquenceur d'instructions différent, permet rarement une exploitation efficace du parallélisme disponible de l'architecture, car les dépendances de données entre étapes sont alors souvent telles que les séquenceurs d'instructions passent une partie importante de leur temps à exécuter les instructions de synchronisation ajoutées pour supporter les dépendances de données entre étapes. Pour permettre une exploitation plus efficace du *parallélisme disponible* de l'architecture, il faut étendre l'*ordre total*, d'exécution des instructions du programme séquentiel monoprocesseur, à un emphordre partiel extrait par une analyse de dépendances de données entre instructions, exhibant le emphparallélisme potentiel de l'algorithme, et permettant une distribution (partitionnement ou "allocation spatiale") et un réordonnancement ("allocation temporelle", limitée au respect de l'ordre partiel) individuel des instructions.

La parallélisation est donc un problème d'allocation de ressources, que l'on désire réaliser de manière efficace, où les ressources sont les séquenceurs d'instructions et les médias de communication inter-séquenceurs, et où les tâches à allouer à ces ressources sont les instructions de l'algorithme ainsi que celles de synchronisation et de communication.

2.3 Optimisation

Pour un codage monoprocesseur d'un algorithme donné, on peut choisir n'importe quelle architecture cible multiprocesseur, et pour chaque architecture choisie il existe un grand nombre d'implantations possibles (c'est-à-dire de distributions et d'ordonnancements des instructions, qui respectent l'ordre partiel). Parmi toutes ces implantations possibles, seules sont éligibles celles dont les performances temps réel (calculées à partir des durées connues d'exécution des instructions et des transferts de données interprocesseurs) respectent les contraintes temps réel. Parmi les implantations éligibles, pour satisfaire les contraintes technologiques d'embarquabilité et de coût, il faut choisir celle qui minimise les ressources matérielles (nombre de processeurs, de médias de communication interprocesseur, et de cellules mémoire).

Le plus difficile n'est pas de comparer les coûts de deux architectures (il suffit d'en sommer les coûts des composants), ni même de vérifier si une implantation respecte les contraintes temps-réel (ce qui nécessite un modèle prédictif des performances temps-réel de n'importe quelle implantation possible), c'est d'éliminer rapidement les solutions inadéquates, afin d'aboutir dans un temps raisonnable au choix d'une bonne implantation. Or ce problème s'apparente aux problèmes d'allocation de ressources, reconnus NP-complets, et est en général de taille gigantesque (variant exponentiellement avec le nombre de processeurs et d'instructions de l'algorithme), ce qui justifie l'utilisation d'heuristiques pour le résoudre.

2.4 Minimisation de l'exécutif

Tout d'abord, pour minimiser les ressources, on commence par minimiser leur gaspillage Pour cela, il faut :

- d'une part équilibrer la charge des ressources (mémoires, séquenceurs d'instructions et médias de communication), c'est-à-dire paralléliser au maximum calculs et communications afin de minimiser les durées d'inactivité (attentes) nécessaires aux synchronisations entre calculs et communications,

– d'autre part minimiser les ajouts d'instructions qui prennent les décisions d'allocation de ressources (mémoire, séquenceurs d'instructions et séquenceurs de communications) afin d'équilibrer leur charge.

Pour que le gain apporté par l'optimisation de l'allocation des ressources (distribution et ordonnancement des calculs et des communications, dont découle la distribution des données dans les mémoires) ne soit pas pénalisé par le coût de l'optimisation elle-même, il faut que celle-ci soit faite avant l'exécution. Comme on dispose alors d'énormément plus de temps que pendant l'exécution, on peut faire des optimisations plus complexes, plus globales et donc probablement plus efficaces. Aux méthodes dites "dynamiques" d'allocation de ressources, qui nécessitent un exécutif représentant un volume de code et un temps d'exécution non négligeables pour prendre à l'exécution des décisions d'allocation, on préférera donc des méthodes plus "statiques" qui consistent à *synthétiser* un exécutif sur mesure, d'un surcoût bien moindre, à partir des décisions de distribution et d'ordonnancement prises avant l'exécution par l'heuristique d'optimisation.

2.5 Choix de la granularité

Ensuite, comme ce problème d'optimisation d'allocation de ressources a un espace des solutions à explorer variant exponentiellement avec le nombre de processeurs et d'instructions de l'algorithme, il faut en réduire la taille afin que la durée d'exécution de l'heuristique d'optimisation reste acceptable.

Aussi, comme atomes ou "grains" indivisibles de distribution et d'ordonnancement, plutôt que de considérer des instructions individuelles, on considérera des agrégats d'instructions préordonnancées (correspondant par exemple à des séquences d'instructions issues de la compilation séparée de sous-programmes FORTRAN ou de fonctions C) qu'on appellera par la suite indifféremment soit *macro-instructions*, soit plus généralement *opérations*, et des agrégats de cellules mémoire contiguës (correspondant par exemple à des tableaux ou à des structures C) qu'on appellera par la suite soit *macro-registres* soit plus généralement *dépendances (de données)*.

2.6 Synchronisation dans les architectures

La synchronisation n'est pas simple entre opérateurs (séquenceurs d'instructions et/ou de communications) car chacun peut séquencer ses opérations indépendamment des autres, sauf dans les deux cas suivants :

1. Lorsque deux opérateurs requièrent simultanément, pour leur micro-opération en cours, un accès à une même ressource (partagée), comme par exemple un bus mémoire, les deux accès doivent être séquentialisés, donc l'un des deux opérateurs doit, avant de commencer son accès, attendre que l'autre opérateur ait terminé le sien, ce qui rallonge d'autant la durée d'exécution de la micro-opération mise en attente ;
2. Lorsqu'une macro-opération consomme en donnée le résultat produit par une autre macro-opération exécutée par un autre opérateur, il faut que ce dernier termine l'exécution de la macro-opération productrice avant que l'autre opérateur ne commence l'exécution de la macro-opération consommatrice, et de plus, comme les algorithmes réactifs sont par nature répétitifs, il faut que la macro-opération consommatrice soit terminée avant que la macro-opération productrice soit à nouveau exécutée lors de l'itération suivante de la séquence répétitive, tout ceci afin que les données ne soient pas modifiées pendant leur utilisation.

Dans les deux cas, des opérations qui auraient pu être exécutées concurremment doivent être exécutées séqueniellement, mais leur ordre d'exécution est sans influence sur le résultat fonctionnel des opérations dans le premier cas, alors qu'il doit être imposé dans le second cas.

C'est la raison pour laquelle dans le premier cas il n'est pas nécessaire de spécifier les synchronisations, d'autant plus qu'elles doivent être faites au niveau de la micro-opération, "invisible" au niveau macroscopique (et même au niveau de l'instruction), et que leurs dates d'occurrence dépendent des durées relatives d'exécution des micro-opérations exécutées concurremment.

Par contre dans le second cas, les synchronisations doivent être spécifiées au niveau macroscopique, insérées entre les macro-opérations.

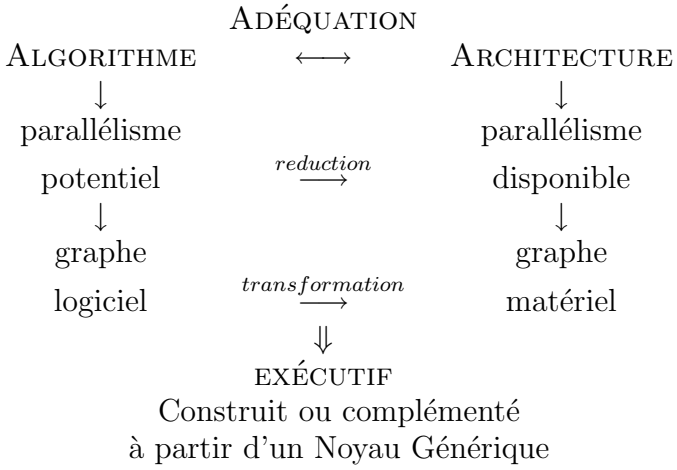
Ce niveau "opérateur" de décomposition de l'architecture correspond à un grain adéquat pour le problème d'optimisation d'allocation de ressources : chaque opérateur séquence des macro-opérations de calcul et/ou de communication, et de synchronisation.

Chapitre 3

Méthodologie AAA et le logiciel SynDEx

3.1 Méthodologie AAA

La méthodologie d'Adéquation Algorithme Architecture est basée sur des modèles de graphes pour spécifier d'une part l'algorithme et d'autre part l'architecture matérielle. La description de l'algorithme permet de mettre en évidence le parallélisme potentiel tandis que celle de l'architecture met en évidence le parallélisme disponible. Cette méthode consiste en fait à décrire l'implantation en terme de transformations de graphes. En effet, le graphe modélisant l'algorithme est transformé jusqu'à ce qu'il corresponde au graphe matériel modélisant l'architecture. L'implantation de l'algorithme sur l'architecture consiste donc à réduire le parallélisme potentiel au parallélisme disponible tout en cherchant à respecter les contraintes temps réel. Toutes ces transformations effectuées avant l'exécution en temps réel de l'application, correspondent à une distribution et à un ordonnancement des différents calculs sur les processeurs et des communications sur les liaisons physiques inter-processeurs. C'est à partir de ces allocations spatiales et temporelles qu'un exécutif va pouvoir être généré et permettre l'exécution de l'algorithme sur l'architecture construite avec des processeurs. Cependant, pour que cette implantation soit vraiment efficace, il est nécessaire de réaliser une adéquation entre l'algorithme et l'architecture. Celle-ci consiste à choisir parmi toutes les transformations proposées celle qui optimise les performances temps réel. Cette méthodologie a été concrétisée dans un logiciel appelé SynDEx.



3.2 Le logiciel SynDEx

Comme il a été dit en introduction, SynDEx est un outil de développement pour l'implantation optimisée d'algorithmes respectant des contraintes temps réel sur des architecture distribuées. A partir de graphes flot de données (la description hiérarchique d'opérations est possible) et d'un graphe d'architecture matérielle, des heuristiques sont mises en œuvre afin d'en déduire une distribution et un ordonnancement optimisé des opérations satisfaisant les contraintes. L'adéquation est réalisée en fonction des paramètres des opérations tels que temps estimé de calcul ou un impératif sur le type de ressources (processeurs, media de communication) où l'opération doit être exécutée. L'approche y est formelle et fondée sur la sémantique des langages synchrones. Le code issu de l'adéquation est un macro-code (m4) qui est ensuite traduit par le macroprocesseur standard M4 utilisant des noyaux d'exécutif dépendant des processeur spécifiés sur le graphe d'architecture (figure 3.1).

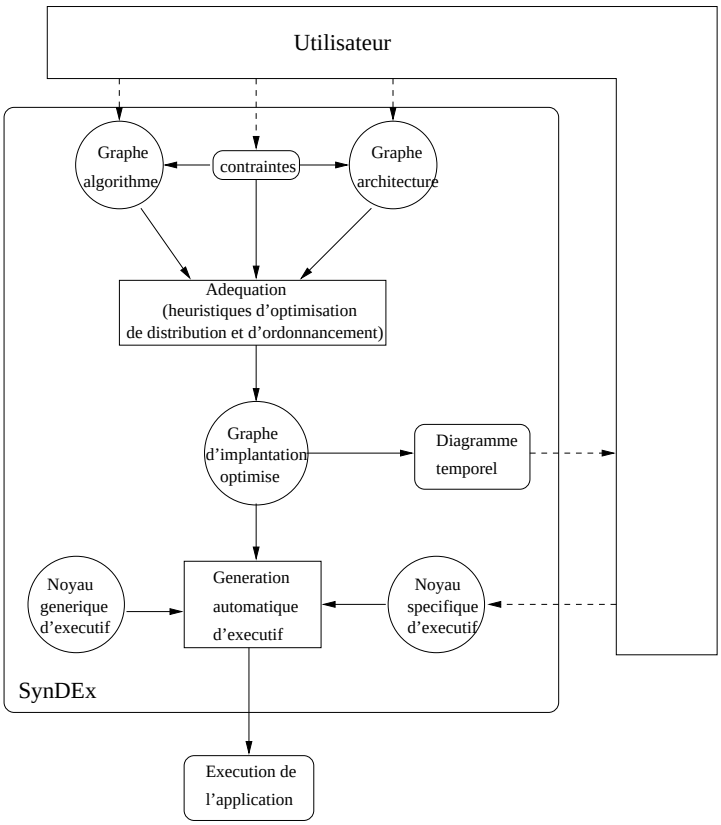


FIG. 3.1 – Principes de SynDEx

3.2.1 Modèles d'algorithmes et d'architectures

Algorithmes

Un algorithme est modélisé par un graphe flot de données éventuellement conditionné (il s'agit d'un hypergraphe orienté), qui se compose de sommets et d'arcs. Un sommet est une opération et un arc un flot de données, c'est-à-dire un transfert de données entre deux opérations.

Une opération peut-être soit un calcul, soit une mémoire d'état (retard), soit un conditionnement ou encore une entrée-sortie. Les sommets qui ne possèdent pas de prédécesseur sont des interfaces d'entrée (capteurs recevant les stimuli de l'environnement) et ceux qui ne possèdent

pas de successeur représentent des interfaces de sortie (actionneurs produisant les réactions vers l'environnement). Dans le cas d'une opération de calcul, la consommation des entrées précède la production des sorties. La figure 7.5 donne un exemple de la description d'un algorithme via SynDEx.

Architectures

Une architecture est modélisée par un graphe dont chaque sommet représente un processeur ou un média de communication, et chaque arc représente une connexion entre un processeur et un média de communications (SAM ou RAM). On ne peut connecter directement deux processeurs ou deux médias. Chaque sommet est une machine séquentielle qui séquence soit des opérations de calcul pour les processeurs, soit des opérations de communications pour les médias de communications.

3.2.2 Heuristique de distribution et d'ordonnancement

Une fois les spécifications de l'algorithme et de l'architecture effectuées, il est nécessaire de réaliser l'adéquation. Celle-ci est chargée de respecter d'une part l'ordre des événements vérifiés lors de la spécification de l'algorithme et d'autre part les contraintes temps réel. Pour cela, est choisie parmi toutes les transformations de graphes possibles, celle qui optimise les performances temps réel de l'implantation en terme de latence. La latence ou temps de réponse R est la longueur du chemin critique du graphe logiciel, dont les sommets sont valués par les durées d'exécution des opérations correspondantes y compris celles des communications inter-processeurs.

Afin de résoudre ce problème d'optimisation du temps de réponse, une heuristique a été développée. Il s'agit d'un algorithme glouton dont chaque étape alloue une opération à un processeur, route les éventuelles communications inter-processeurs c'est-à-dire crée des opérations de communication et alloue chacune d'elles à une liaison physique. L'ordonnancement des opérations de calculs ou de communication est directement déduit de l'ordre dans lequel elles sont allouées.

Cette méthode consiste donc à faire progresser au long du graphe une coupe séparant les opérations déjà placées sur des processeurs de celles qui ne le sont pas encore. La progression se fait en respectant les précédences du graphe logiciel. De toutes les opérations à distribuer sur la coupe et de tous les processeurs, on choisit la paire qui optimise une fonction locale de coût prenant en compte :

- les différences entre dates locales d'exécution au plus tôt et au plus tard (schedule flexibility),
- l'allongement du temps global d'exécution : le temps de réponse (latence),
- le rythme d'entrée (cadence),
- la capacité mémoire.

Afin d'illustrer l'adéquation, la figure 3.2 montre le graphe temporel d'exécution de l'algorithme de la figure 7.5 sur l'architecture de la figure 7.15. Le temps se déroule de haut en bas avec une colonne par processeur (`root`, ...) ainsi qu'une colonne par média de communication (`bus`). Chaque opération de calcul est représentée par une boîte dont la hauteur est proportionnelle à la durée d'exécution de l'opération. Chaque communication inter-processeurs est représentée par une boîte dont la taille est proportionnelle à la durée de la communication. La communication commence dès que l'opération qui a fournit la donnée à transmettre est terminée, l'opération qui a besoin de la donnée transférée commence dès que la communication est terminée. La valeur de la durée d'une itération du graphe est, quant à elle, donnée dans la fenêtre principale de SynDEx.

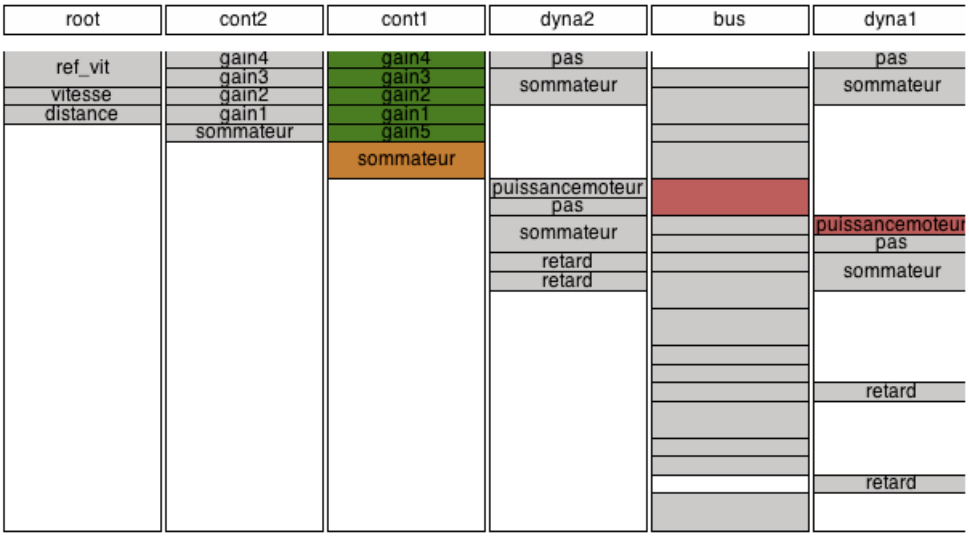


FIG. 3.2 – Graphe temporel pour l'application du chapitre 7

3.3 Les noyaux d'exécutif

3.3.1 Génération d'exécutif

Les transformations de graphes modélisant le processus d'implantation de l'algorithme sur l'architecture, permettent de produire automatiquement des exécutifs temps réel optimisés, déchargeant ainsi l'utilisateur des tâches fastidieuses de programmation bas niveau et autorisant du même coup une meilleur concentration sur les problèmes directement liés au programme applicatif.

Un exécutif a pour rôle d'allouer les ressources de l'architecture matérielle (unités de calcul, de mémoire et communication) au programme d'application. Les exécutifs peuvent être classés en fonction de leur manière d'arbitrer l'allocation des ressources. Nous avons vu précédemment que cette allocation est à la fois spatiale (distribution) et temporelle (ordonnancement). Si les optimisations et les décisions que doit prendre l'exécutif sont effectuées à l'exécution, on dit que l'allocation est dynamique. Par contre, si cela est fait avant l'exécution, on dit que l'allocation est statique (il faut connaître les durées d'exécution). Dans le cas d'une allocation statique, les exécutifs sont les moins pénalisants car dans le cas d'une allocation dynamique, l'exécutif consomme une partie des ressources pour effectuer ses décisions d'arbitrage et d'optimisation. C'est pourquoi SynDEx génère des exécutifs statiques. Il y en a autant que de processeurs dans l'architecture.

Comme les exécutifs sont générés automatiquement, l'utilisateur n'a pas d'autre code à écrire que celui de ses opérations de calculs et d'entrée/sortie. Tout le reste, c'est-à-dire la distribution, l'ordonnancement, les appels des opérations de calcul et d'entrée/sortie, les allocations de la mémoire nécessaires aux communications inter-opérations est généré automatiquement à partir des graphes logiciel et matériel, des résultats de l'optimisation et d'un noyau générique. En effet, pour chaque processeur, l'exécutif est constitué par un assemblage d'éléments d'un noyau générique d'exécutifs (tirés d'une bibliothèque système) qui gère les communications inter-processeurs. Chaque exécutif généré par SynDEx est un macro-code indépendant de l'architecture. Ce macro-code est ensuite macro-processé avec des noyaux d'exécutifs dépendant de l'architecture et de l'application, afin d'obtenir du code source, qui après compilation sera exécutable.

3.3.2 Arborescence des noyaux d'exécutifs

Les noyaux d'exécutifs sous SynDEx sont divisés en différents groupes comme le montre la figure 3.3. Le travail de ce stage qui a consisté à aider à la conception des noyaux applicatifs (dépendants de l'application) sera expliqué plus en détail dans la deuxième partie de ce rapport.

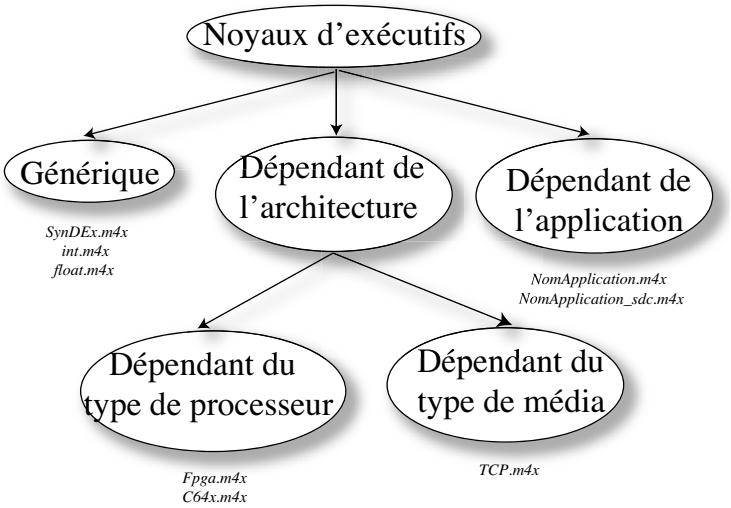


Fig. 3.3 – Arborescence des exécutifs SynDEx

Deuxième partie

Travail effectué

Chapitre 4

Présentation

4.1 Contexte et objectif

La tâche ultime de SynDEx consiste à générer des fichiers (un par processeur) contenant un macrocode. L'utilisateur de SynDEx doit, ensuite, utiliser le macro processeur GNU m4 pour transformer ces macros en code source (langage C ou assembleur, ...) qui seront compilés afin d'obtenir les exécutable pour les architectures cibles.

Pour obtenir un fichier exécutable, l'utilisateur doit suivre les trois étapes suivantes :

1. Premièrement, il doit donner dans un même fichier (noyau applicatif nommé par le nom de l'application plus l'extension m4x) tous les codes sources avec leur squelette en langage m4, associés aux définitions des opérations contenues dans une application SynDEx (opérations du graphe d'algorithme).
2. Deuxièmement, il doit créer un makefile permettant de lancer le macro processeur m4 qui va macro-expanser tous les fichiers produits par la génération de code de SynDEx et créer à l'aide des noyaux d'exécutifs des fichiers source (C, assembleur).
3. Troisièmement, une fois ces fichiers obtenus via le makefile, il reste à les compiler sur les différents processeurs.

De plus, des fichiers génériques viennent avec le logiciel SynDEx, et sont utilisés pour réaliser les deux dernières étapes. Ce sont des fichiers contenant des macros génériques en langage m4 de routines usuelles (calcul en entier, calcul en flottant, communication TCP/IP, ...), un script rsh et un fichier nommé **syndex.m4x** permettant de créer des makefiles automatiquement.

Le travail de ce stage a consisté à s'occuper de la première partie (comme le montre les figures 4.2 et 4.3).

4.2 Problèmes rencontrés lors de la conception manuelle de noyaux d'exécutif

Nous décrivons ici les problèmes rencontrés par l'utilisateur de SynDEx qui ont conduit à ce stage. Supposons que dans une application appelée **MonApplication** il existe une fonction **foo** incrémentant de un la valeur de son port d'entrée **in** et renvoyant le résultat vers son port de sortie **out**.

Nous devons écrire le code m4 de la figure 4.1, dans le fichier du noyau applicatif correspondant (ici **MonApplication.m4x**), où, les macros m4 **$1** et **$2** (que l'on appellera par la suite des arguments) correspondent respectivement aux ports **in** et **out** de la fonction **foo** dans le

```
define(‘foo’,‘ifelse(
        MGC,‘INIT’,‘‘’’,
        MGC,‘LOOP’,‘‘$2[0] = $1[0] + 1’’,
        MGC,‘END’,‘‘’’),)
```

FIG. 4.1 – Un exemple simple

langage m4. La numérotation des arguments est précise : elle doit suivre le type (port d'entrée, port de sortie) et l'ordre dans lequel sont déclarés les ports.

Ecrire manuellement ce type de code n'est pas aisé, pour plusieurs raisons :

- La numérotation des arguments change lors d'ajoût ou de suppression de ports ou des paramètres dans la fonction. Par exemple, après avoir ajouté un paramètre **P** dans la fonction **foo**, **$1** ne désignera plus l'entrée **in** mais le paramètre **P** ajouté. Toutes les numérotations des arguments m4 sont alors décalées de 1. Le code doit être corrigé : il faut remplacer le argument **$1** par **$2** et le argument **$2** par **$3**.
- Cette tâche devient répétitive pour une application complexe contenant beaucoup d'opérations et par conséquent beaucoup de ports.
- Il est facile de se tromper dans la syntaxe m4 (en plus des erreurs de numérotation des arguments). De bonnes connaissances en m4 (connaître les différents types de guillemets) et les macros SynDEx sont nécessaires (`MGC`).

Il serait plus simple de pouvoir écrire directement son code source (par exemple `@OUT(out)[0] = @IN(in)[0] + @PARAM(P)`) et laisser SynDEx s'occuper de la transformation du code en directives m4. SynDEx (version $\geq$ 6.8.4) est capable de réaliser cette tâche grâce à l'outil, intégré dans l'IHM de SynDEx, qui a été réalisé pendant ce stage que l'on appellera *Éditeur de Code*.

4.3 Contribution de ce stage à SynDEx

Les figures 4.2 et 4.3 montrent l'état de SynDEx avant et après le stage. La figure 4.2 permet de mieux comprendre le traitement effectué par SynDEx et celui effectué par l'utilisateur. Elle explique le fonctionnement interne de SynDEx (déjà vu dans la figure 3.1 de la partie temps réel de ce rapport) et montre les différents fichiers manipulés par SynDEx et/ou par l'utilisateur. La figure 4.3 montre comment l'Éditeur de Code s'intègre dans la structure interne de SynDEx. On voit également que grâce à lui, le nombre de fichiers à éditer manuellement diminue.

Dans le chapitre 5 on explique le fonctionnement global de l'Éditeur. Dans le chapitre 7 une application complète réalisée pendant ce stage permet de comprendre l'utilité de l'Éditeur de Code.

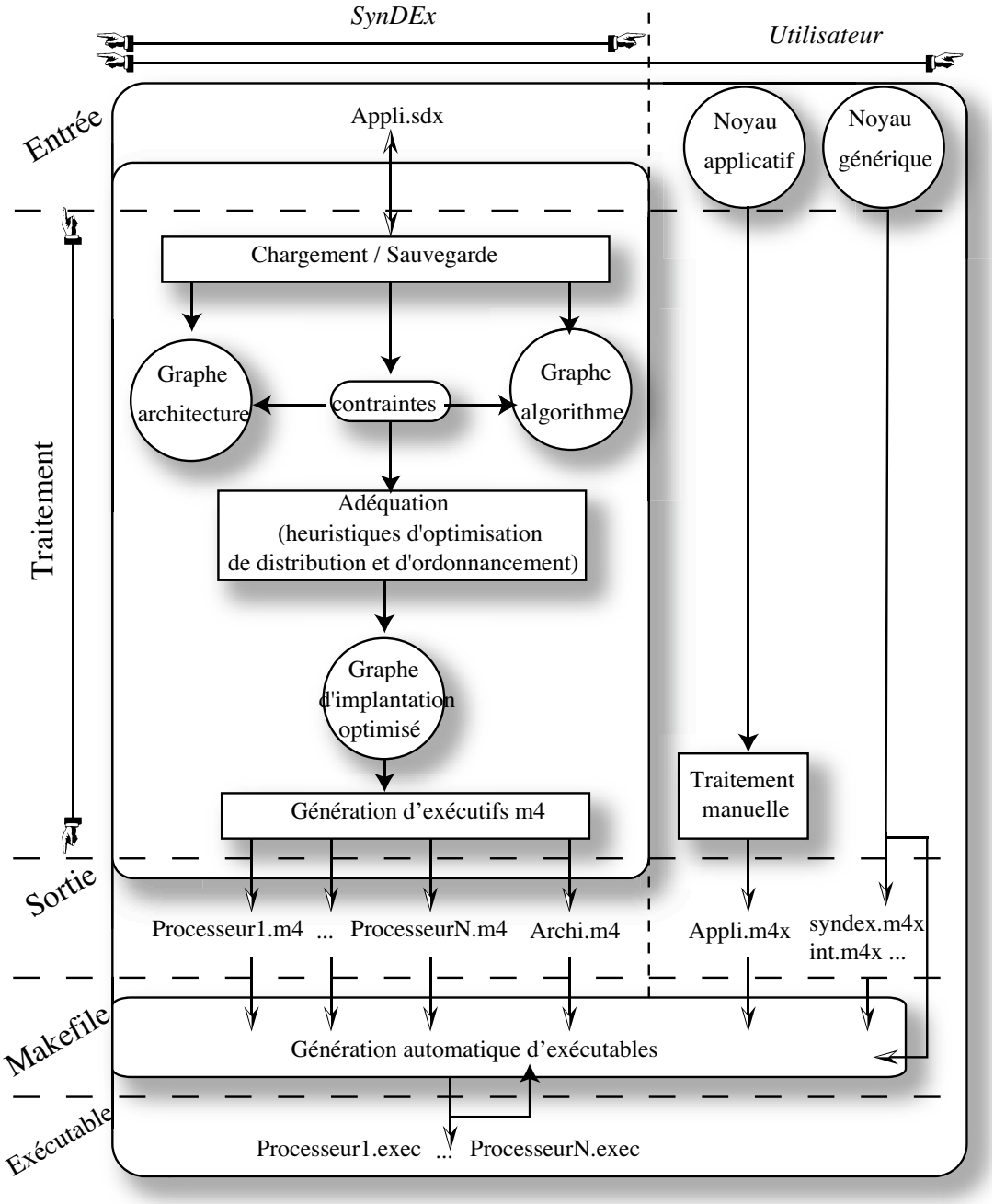


FIG. 4.2 – Le fonctionnement de SynDEx avant le stage.

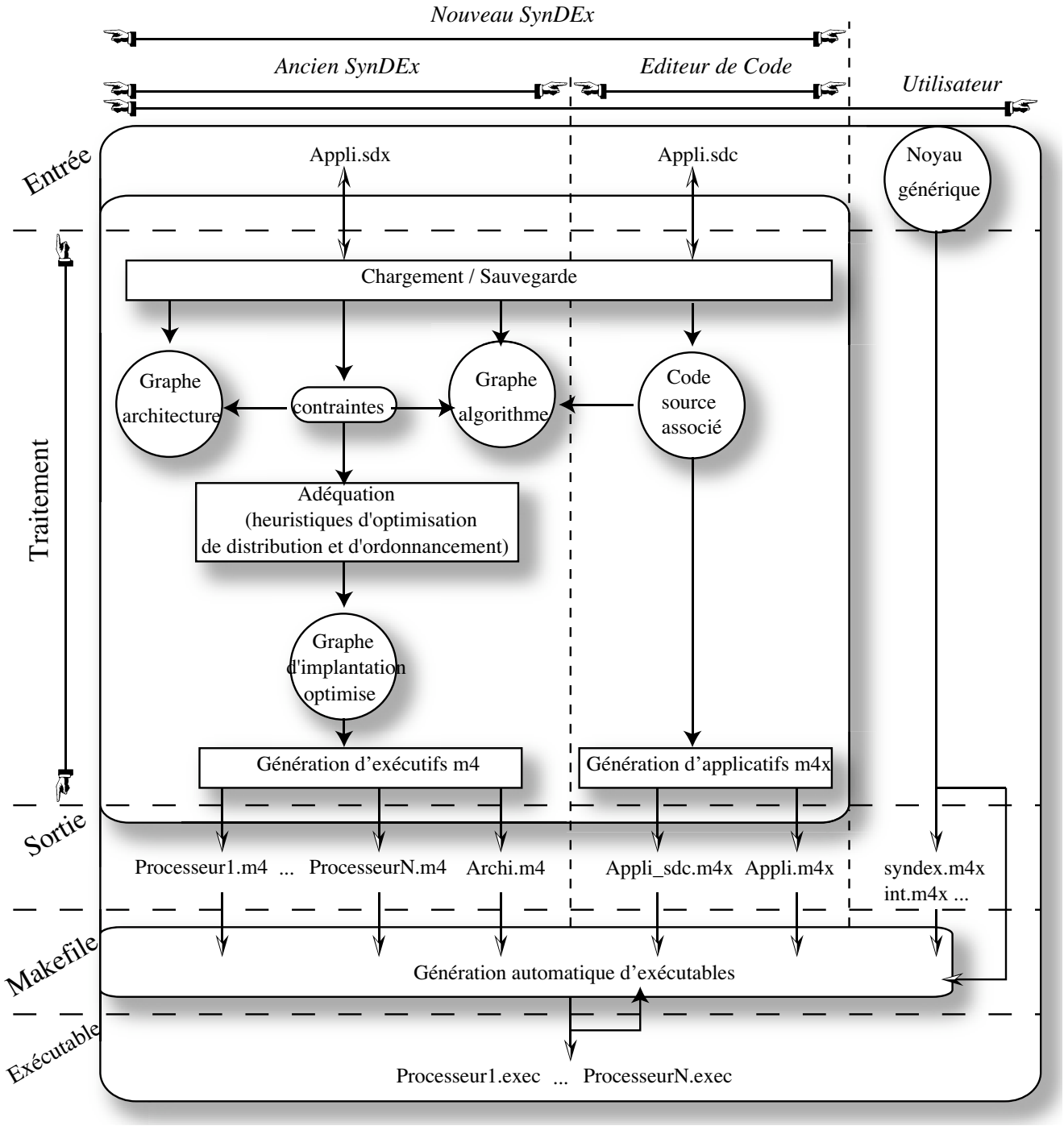


FIG. 4.3 – Le fonctionnement de SynDEx après le stage.

Chapitre 5

L'Éditeur de Code

Le premier travail consistait à construire la fenêtre de l'Éditeur de Code, comme présentée figure 5.1. Elle contient trois boutons poussoirs et une zone de texte éditable. Chaque bouton correspond à l'une des trois phases d'un opérateur SynDEx, ce qui permet de diviser le code source en trois sous codes (code pour la phase d'initialisation, code pour la phase de boucle et code pour la phase de terminaison). Lorsqu'un bouton est enfoncé le code source correspondant à la phase est affiché dans la zone de texte pour y être éditée. L'utilisateur entre alors son code source dans le langage choisit dans la zone de texte.

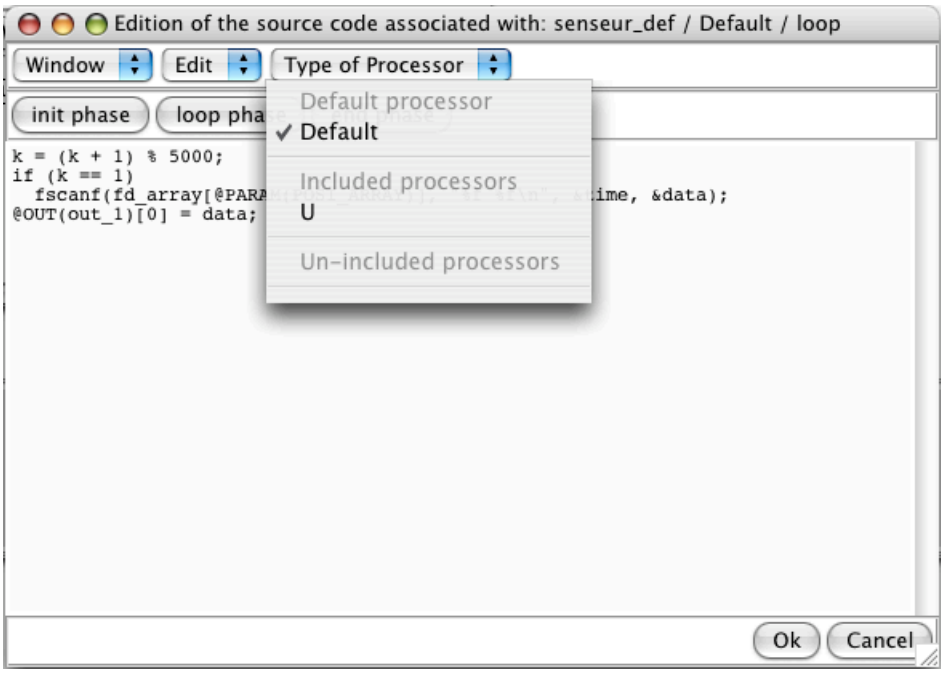


Fig. 5.1 – Capture d'écran de l'Éditeur de Code sous Mac OS X

Cet outil est utile car il permet de cacher le squelette en langage m4 entourant le code source. Seul le code source est éditable. Des macros spécifiques au logiciel SynDEx (comme `MGC`) ou de syntaxe m4 (comme `ifelse`) ne devient plus à être connues de l'utilisateur et le problème des guillemets disparaît également. Un jeu de macros plus simples et spécifique à SynDEx permettent de transformer les noms des paramètres ou de ports en arguments m4 (`$0` ...) et permet d'éviter de connaître la règle de numérotation.

5.1 Edition du code source associé à une opération

5.1.1 Le jeu de macros

SynDEx gérant les noms des ports et des paramètres, aussi utilisés dans le code utilisateur dont il ne connaît pas la syntaxe, un jeu de macros spécifique à l'Éditeur de Code a été créé pour faciliter l'écriture du code spécifique de l'utilisateur que SynDEx ne peut pas automatiser.

La première macro `@NAME(nom_port_ou_param)` prend en paramètre un nom de port ou de paramètre associé à l'opération et le transforme en macro m4 (le caractère `$` suivi d'un entier positif ou nul. La numérotation, comme nous l'avons dit en introduction dans le chapitre précédent, dépend du type de port (entrée, sortie, entrée–sortie) ainsi que l'ordre où ils sont déclarés. Toute erreur avec des noms inconnus est collectée puis affichée lors de la génération de code.

Comme SynDEx accepte que des ports de types différents aient des noms identiques alors l'Éditeur de Code peut se tromper. Des macros comme `@IN()` ou `@OUT()` ou `@INOUT()` ou `@PARAM()` permettent de spécifier le type. Toute erreur sur un nom connu mais de mauvais type produit une erreur lors de la génération de code.

Par défaut, le code associé est entouré par deux niveaux de guillemets (confère l'exemple 4.1 de l'introduction). Un utilisateur peut vouloir retirer ou ajouter un niveau de guillemets. Cela est possible grâce aux macros `@TEXT(texte)` et `@QUOTE(texte)`, où `texte` est du texte de taille arbitrairement grande et qui peut inclure d'autres macros.

Ce jeu de macro a été créé en utilisant les lexer et parser de OCaml : Camllex et Camlyacc.

5.1.2 Edition du code pour un processeur unique et générique

Pour obtenir l'exemple 4.1 de l'introduction il suffit d'ouvrir la fenêtre de l'Éditeur de code de l'opération `foo` de sélectionner la phase de boucle et d'entrer le code suivant :

```
@OUT(out)[0] = @IN(in)[0] + @PARAM(P);
```

Par défaut, ce code est considéré comme un code générique, à savoir fonctionnant sur n'importe quel type de processeur. Mais on peut vouloir un code spécifique à un processeur.

5.1.3 Edition du code pour des processeurs de types différents

Il est intéressant pour une opération d'avoir des codes sources différents selon le type des processeurs. En effet on peut désirer une architecture hétérogène, où chaque processeur possède son propre langage, mais alors une difficulté apparaît. Lors de l'adéquation de SynDEx, une opération peut s'exécuter sur un premier type de processeur mais après une modification de sa durée d'exécution, elle pourrait devoir être exécutée sur un autre processeur. On aurait un problème puisque le langage ne serait pas reconnu par ce dernier.

Pour résoudre ce problème, l'utilisateur est obligé de donner le code spécifique dans différentes syntaxes. Pour faciliter ce travail l'éditeur associe une fenêtre pour chaque triplet (phase, processeur, opération). Par exemple, pour compléter l'exemple 4.1 de l'introduction il suffit d'ouvrir la fenêtre de l'Éditeur de Code de l'opération `foo` et de sélectionner un type de processeur connu de SynDEx (grâce à des bibliothèques). Par exemple écrivons le code (en pseudo assembleur) suivant pour le processeur `X` dans la phase de boucle :

```
add #@IN(in), @PARAM(P)
mov #@OUT(out), #@IN(in)
```

Lors de la génération de code (que nous expliquerons dans la section suivante), SynDEx génére un macro-code m4 suivant :

```
define(‘foo’,‘ifelse(
    processorType_,‘X’,‘ifelse(
        MGC,‘INIT’,‘‘’’,
        MGC,‘LOOP’,‘‘add #$2, $1
                     mov #$3, #$2’’,
        MGC,‘END’,‘‘’’)’,
    processorType_,processorType_,‘ifelse(
        MGC,‘INIT’,‘‘’’,
        MGC,‘LOOP’,‘‘$3[0] = $2[0] + $1;’’,
        MGC,‘END’,‘‘’’)’)’)
```

Les macros `processorType_` et `MGC` contiennent respectivement le nom du processeur et le nom de la phase.

La macro m4 `ifelse(string-1, string-2, equal, ...)`, si elle est appelée avec trois ou quatre arguments, se substitue en `equal` si `string-1` et `string-2` sont égaux (caractère par caractère), sinon se substitue en `not-equal`. Cependant, `ifelse` peut prendre plus de quatre arguments comme c'est le cas dans notre exemple, alors cette macro fonctionne comme un `switch` ou un `case` des langages traditionnels. Si `string-1` et `string-2` sont égaux alors `ifelse` se transforme en `equal`, autrement la procédure est répété sans les trois premiers arguments. Le code m4 `ifelse(processorType_, processorType_, equal` sera toujours vrai. Cette astuce est utilisée pour stocker le code source générique.

5.2 Les nouveaux types de fichiers

5.2.1 Les fichiers du noyau applicatif

Avant ce stage, le code généré pour l'exécutif distribué d'une application était constitué :

- d'un fichier `processeur.m4` pour chaque processeur, macro-codant l'exécutif dédié à ce processeur, qui sera traduit en source compilable pour ce processeur ;
- et d'un fichier unique `MonApplication.m4x` macro-codant la topologie de l'architecture, qui sera traduit en makefile pour automatiser les opérations de compilation.

Désormais, il génère en plus deux fichiers appelés respectivement (**MonApplication.m4x** et **MonApplication_sdc.m4x**) définissant le noyau applicatif (voir section 3.3.1 et figure 3.3) :

- Le fichier **MonApplication_sdc.m4x** est le fichier le plus important puisque il contient tous les macro-codes m4 des codes sources associés à toutes les opérations utilisées dans une application, comme l'exemple de la section 5.1.3. A chaque génération, le nouveau fichier écrase l'ancien.
- Le fichier **MonApplication.m4x** est un fichier éditable par l'utilisateur afin de lui permettre de compléter son noyau applicatif en cas de besoin. Lorsque SynDEx génère les exécutifs et que ce fichier n'est pas physiquement présent sur le disque (et uniquement dans ce cas) alors il crée un nouveau fichier `MonApplication.m4x` (contenant incluant le fichier **MonApplication_sdc.m4x**), sinon il n'y touche pas.

5.2.2 Le fichier de sauvegarde de SynDEx

Une application SynDEx est sauvegardée dans un fichier nommé par le nom de l'application suivi de l'extension `sdx`. Il contient à la fois le graphe de l'algorithme, le graphe de l'architecture et les contraintes (comme montré dans la figure 3.1). Le code associé aux opérations de l'application doit lui aussi être sauvegardé mais dans un deuxième type de fichier (nommé par le nom de l'application mais suivi de l'extension `sdc`).

La décision d'utiliser ce deuxième type de fichier a été prise parce qu'il était trop dangereux et trop long de créer un parser dans le même langage que celui du noyau applicatif généré (à savoir m4). En effet, il est impossible d'empêcher l'utilisateur de les modifier ce qui implique l'utilisation d'un parser m4 complet.

Donc l'utilisation d'une syntaxe plus simple et spécifique à SynDEx a été adoptée. Il a été ensuite question de savoir si elle devait être présente dans le fichier `sdx` ou être détachée de celui-ci dans un deuxième fichier. La deuxième solution fut acceptée car : – d'une part, mon travail ne devait pas mettre en péril tout le fonctionnement du programme en cas d'erreur de ma part, – d'autre part, il était préférable de mieux distinguer le code source associé aux opérations du squelette de l'application (la structure des graphes) puisque cette dernière est l'élément primordial du logiciel.

La sauvegarde dans le fichier `sdc` ne fut pas une étape difficile contrairement au chargement qui a nécessité les outils Camllex (lexer) et de Camlyacc (parser).

5.3 Autres utilitaires attachés à l'Éditeur de Code

Pour finaliser le fonctionnement de l'Éditeur de Code, deux fonctionnalités importantes ont été rajoutées :

- la possibilité d'éditer son buffer grâce à un éditeur de texte extérieur à l'IHM de SynDEx (comme Emacs, Vi, ou autre), puisqu'il se peut que l'on veuille traiter le code source (re-indenter les lignes, enlever les espaces doublons, avoir la coloration de son code ...), chose que l'Éditeur de Code n'est pas capable de faire.
- la complétion des noms des ports et des paramètres de l'opération dans la zone de texte permet d'écrire les premiers caractères du nom et la touche `TAB` complète automatiquement les noms possibles.

Chapitre 6

Introduction à l'automatique

Dans ce chapitre, des éléments d'automatique sont présentés rapidement afin de faciliter la compréhension du chapitre suivant où une application SynDEx complète est donnée.

6.1 Logiciels utilisés : Scilab et Scicos

Il existe deux types de programmes scientifiques : – les logiciels algébriques faisant essentiellement du calcul symbolique (Maple, Mathematica, Maxima, Axiom, et MuPad), – les logiciels de calcul scientifique faisant essentiellement de l'analyse numérique (Scilab, MATLAB).

Scilab [4, 5] est un logiciel libre pour le calcul scientifique. Scilab est un interpréteur de langage manipulant des objets typés dynamiquement. Il inclut de nombreuses fonctions spécialisées pour le calcul numérique organisées sous forme de librairies ou de boites à outils qui couvrent des domaines tels que la simulation, l'optimisation, et le traitement du signal et du contrôle.

Une des boites à outils les plus importantes de Scilab est Scicos [4, 6]. Scicos est un éditeur graphique de bloc diagrammes permettant de modéliser et de simuler des systèmes dynamiques. Il est particulièrement utilisé pour modéliser des systèmes où des composants temps-continu et temps-discret sont inter-connectés.

Un programme Scicos peut être traduit en un programme SynDEx grâce à un traducteur intégré dans l'IHM de Scilab. Scicos a été utilisé pour simuler et de vérifier le bon fonctionnement de application de SynDEx à l'automatique décrite ci dessous.

6.2 Rappel de quelques éléments de l'automatique

Considérons un bateau [7] ayant un pilote automatique recevant en permanence le cap actuel α du bateau et le cap désiré α_c. En utilisant ces informations le pilote automatique génère au cours du temps des ordres de positionnement ϵ du gouvernail de façon à ce que l'erreur de cap $e = \alpha_c - \alpha$ soit maintenue aussi faible que possible sachant que le bateau reçoit des perturbations extérieures (vent, ...) (figure 6.1).

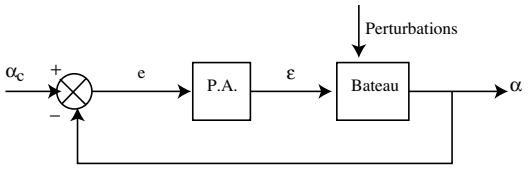


Fig. 6.1 – Pilote automatique de bateau (P.A.)

Pour ce faire, une loi de commande, calculant $\epsilon = f(e)$ en fonction des informations disponibles, pourrait être :

- par à-coups :

$$\epsilon = \begin{cases} +\epsilon_m & \text{si } e > 0 \ , \\ -\epsilon_m & \text{si } e < 0 \ , \end{cases}$$

- proportionnelle :

$$\epsilon = Ke \ ,$$

- proportionnelle et dérivée :

$$\epsilon = Ke + K' de/dt \ ,$$

- proportionnelle et intégrale :

$$\epsilon = Ke + 1/\rho \int^t e(\alpha) \, d\alpha \ ,$$

- proportionnelle et intégrale et dérivé etc.

C'est la théorie des asservissements qui permettra, dans un cas particulier de choisir la loi de commande la mieux adaptée. Dans les paragraphes suivant on va introduire les outils permettant d'analyser la classe des systèmes linéaires temps invariant.

6.3 Transformée de Laplace et transformée en z

Deux transformations permettent de ramener l'analyse des systèmes dynamiques linéaires à du calcul algébrique. Ce sont la transformée de Laplace et la transformée en z. Elles transforment des fonctions du temps en une fonction d'une variable dont la partie imaginaire s'interprète en terme de fréquence. Ces transformations permettent de résoudre les équations différentielles linéaire à coefficients constants.

6.3.1 Transformée de Laplace

Définition

La transformée de Laplace est définie de la manière suivante : soit $f(t)$ une fonction du temps définie pour $t > 0$. Alors :

$$\mathcal{L}[f(t)] \equiv F(s) \equiv \lim_{\substack{T \to \infty \\ \epsilon \to 0}} \int_\epsilon^T f(t) \, e^{-st} dt = \int_{0^+}^\infty f(t) \, e^{-st} dt \quad 0 < \epsilon < T$$

où s est un variable complexe défini par $s \equiv \sigma + j\omega$.

Transformée de Laplace de dérivées

Montrons que la transformée de Laplace de la dérivée df/dt d'une fonction $f(t)$ vaut :

$$\mathcal{L}\left[\frac{df}{dt}\right] = s\mathcal{L}f - f(0^+)$$

En intégrant par partie, on obtient :

$$\left[e^{-st} f(t)\right]_{0^+}^\infty + s \int_{0^+}^\infty f e^{-st} dt$$

Finalement :

$$\mathcal{L}\left[\frac{df}{dt}\right] = s\mathcal{L}f - f(0^+)$$

Autre exemple

Une autre formule utilisée dans l'application du chapitre suivant est la transformée de Laplace de la double dérivée d^2f/dt^2 d'une fonction $f(t)$:

$$\mathcal{L}\left[\frac{d^2f}{dt^2}\right] = s^2F(s) - sf(0^+) - \frac{df}{dt}\Big|_{t=0^+}$$

6.3.2 Transformée en z

La transformée en z est utilisée pour décrire des signaux en temps discret. Soit $\{f(k)\}$ dénote une séquence de valeur réelle $f(0), f(1), f(2), \ldots$ ou bien $f(k)$ pour $k = 0, 1, 2, \ldots$. Alors on définit la transformée en z par :

$$\mathcal{Z}\{f(k)\} \equiv F(z) = \sum_{k=0}^{\infty} f(k)z^{-k}$$

où z est un variable complexe défini par $z \equiv \sigma + j\omega$.

6.4 Placement de pôles

Un système linéaire temps invariant (LTI) ayant un nombre fini d'états s'écrit généralement de la façon suivante :

$$\begin{aligned} \frac{dx}{dt} &= Ax + Bu \\ y &= Cx \end{aligned}$$

Où la A matrice est une matrice $n \times n$, B une matrice $n \times p$, C une matrice $q \times n$, . Sous block diagramme il se dessine comme sur la figure 6.2.

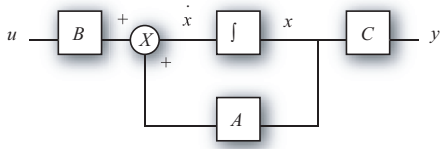


FIG. 6.2 – u est l'entrée, y la sortie, x l'état (ou mémoire)

La fonction de transfert d'un système linéaire temps invariant mono-entrée mono-sortie est une fonction rationnelle s'écrivant :

$$G(s) = C(sI - A)^{-1}B = \frac{N(s)}{D(s)} ,$$

où N et D sont deux polynômes.

On appelle pôles les zéros du polynôme $D(s)$, ils sont aussi les valeurs propres de la matrice A. La présence du pôle λ implique que la sortie du système $y(t)$ contient une composante de la forme $e^{\lambda t}$. Alors, si $Re(\lambda) > 0$ la sortie tend vers l'infini lorsque t tend vers l'infini. Le système est dit instable. Un système est asymptotiquement stable si et seulement si toutes les valeurs propres vérifient : $Re(\lambda_i) < 0$.

Considérons le système bouclé (u en feedback sur l'état) dans lequel $u = Kx + v$ le nouveau système s'écrit

$$\begin{aligned} \frac{dx}{dt} &= (A + BK)x + Bv \ , \\ y &= Cx \ . \end{aligned}$$

Le feedback étant à notre disposition on peut choisir ses coefficients de façon à placer les pôles du système bouclé où l'on veut. Par exemple rendre stable par feedback un système instable.

Un théorème indique sous quelles conditions sur les matrices (A, B, C) il est possible de placer les pôles du système où l'on veut.

Chapitre 7

Application de SynDEx à un problème d'automatique

Nous allons, maintenant, nous intéresser à la création d'un exemple complet d'automatique qui a été intégré dans le tutoriel de SynDEx. En plus de montrer le fonctionnement global de SynDEx, il met en relief l'utilité de l'Editeur de Code, intégré maintenant dans ce logiciel, et développé pendant le stage.

Le modèle de l'application est spécifié et simulé en Scicos, le logiciel de bloc diagrammes de Scilab. Ensuite, l'application Scicos est traduite, éventuellement de façon automatique avec la passerelle Scicos/SynDEx, en une application multiprocesseur de SynDEx. Cet exemple ne peut pas être entièrement traité par SynDEx seul car les bibliothèques ne contiennent pas toutes les fonctions d'automatique nécessaires. L'utilisateur doit rajouter du code à la main en utilisant les facilités de l'Editeur de Code.

7.1 Le modèle

Nous considérons un système constitué de deux voitures. La deuxième voiture $\mathcal{C}_2$ suit la première voiture $\mathcal{C}_1$ tentant de maintenir la distance l alors que la première accélère ou ralentit. Nous appelons $x_1(t)$ la position de $\mathcal{C}_1$, $x_2(t)$ la position de $\mathcal{C}_2$ plus l. $\dot{x_1}(t)$ et $\dot{x_2}(t)$ les vitesses des deux voitures. Nous considérons k_1 et k_2 comme étant les inverses des masses des voitures. Nous appelons $r(t)$ la référence en vitesse choisie par le chauffeur de $\mathcal{C}_1$. Enfin, nous supposons que nous sommes capable d'observer la vitesse de la première voiture ainsi que la distance entre les deux voitures.

Nous avons le système suivant d'ordre quatre :

$$\begin{aligned} \ddot{x_1} &= k_1 u_1 \\ \ddot{x_2} &= k_2 u_2 \\ y_1 &= \dot{x_1} \\ y_2 &= x_1 - x_2 \end{aligned} \tag{7.1}$$

Nous décomposons ce système en deux sous systèmes mono-entrée mono-sortie $S_1(u_1, y_1)$ et $S_2(u2, y2)$. Nous notons en majuscule les transformées de Laplace des variables utilisées. Nous avons $Y_1 = k_1 U_1/s$ et $Y_2 = (k_1 U_1 - k_2 U_2)/s^2$ où U_1 est vu par le deuxième système comme une perturbation à rejeter.

Une première rétro-action (feedback) $U_1 = \rho_1(R - Y_1)$ permet de stabiliser la vitesse de la première voiture autour d'une vitesse de référence. Le deuxième contrôleur est du type

proportionel-dérivé $U_2 = \rho_2 Y_2 + \rho_3 s Y_2$ (en fait, dans le diagramme suivant, nous supposerons que la dérivée de y_2 est aussi observée). Le coefficient ρ_1 est obtenu par placement du pôle de la première boucle :

$$Y_1 = \rho_1 k_1 R/(s + \rho_1 k_1).$$

Le coefficient ρ_2 et ρ_3 sont obtenus par placement des pôles de la fonction de transfert U_1 vers Y_2 dans le système en boucle fermé qui est donné par :

$$Y_2 = U_1 k_1/(s^2 + k_2\rho_3 s + k_2\rho_2).$$

7.2 Les contrôleurs des voitures

Le but du contrôleur de $\mathcal{C}_1$ est de suivre la vitesse de référence donnée par le conducteur. Il stabilise sa vitesse autour de sa référence en vitesse. En utilisant le placement de pôle, les gains valent respectivement : (0, -5, 0, 0, -5). Le contrôleur de la deuxième voiture stabilise la distance entre les deux voitures (stabilise y_2 autour de de 0). Par placement de pôles, les gains valent respectivement : (4, 4,-4,-4).

Le contrôleur de la deuxième voiture connaît la position et la vitesse de la dynamique de $\mathcal{C}_2$ et les renvoient à la première voiture de façon électronique (cette remarque vaut aussi pour la voiture $\mathcal{C}_1$).

7.2.1 Bloc diagrammes des contrôleurs

Nos contrôleurs sont relativement simples. Ils sont représentés sur les figures 7.1 et 7.3 sous Scicos et 7.2 et 7.4 sous SynDEx.

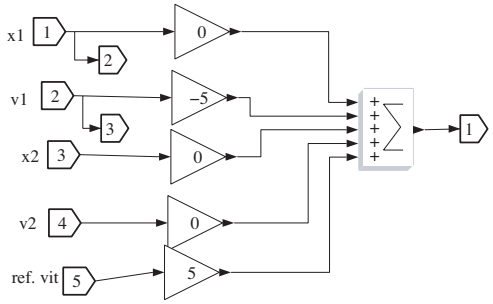


FIG. 7.1 – Le contrôleur de la première voiture.

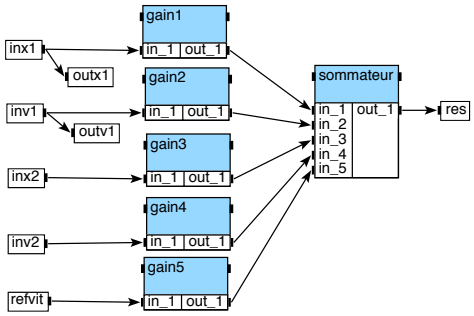


FIG. 7.2 – Le contrôleur de la première voiture.

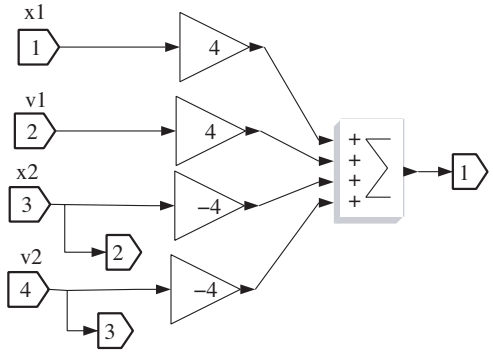


FIG. 7.3 – Le contrôleur de la deuxière voiture.

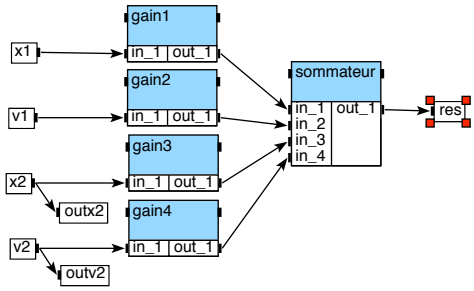


FIG. 7.4 – Le contrôleur de la deuxière voiture.

7.2.2 Edition du code source associé aux opérations SynDEx

Nous allons attacher du code C à chaque opérations : gain et sommateur n-aires. Le code généré est du C (processeur par défault).

La fonction gain

Un gain est une fonction qui multiplie la valeur de son port d'entrée par un coefficient donné en paramètre, nommé `GAIN`. Ouvrons l'Editeur de Code de la fonction gain et écrivons le code suivant dans la zone de texte associée à la phase `init` du processeur par défault.

```
@OUT(out_1)[0] = @IN(in_1)[0] * @PARAM(GAIN);
```

Les fonctions sommes

Nous avons des sommes avec des arités différentes. Ouvrons l'Editeur de Code de la fonction somme à cinq ports d'entrée (figure 7.2) et écrivons le code suivant dans la zone de texte associée à la phase `init` du processeur par défault.

```
@OUT(out_1)[0] = @IN(in_1)[0] + @IN(in_2)[0] + @IN(in_3)[0]
                              + @IN(in_4)[0] + @IN(in_5)[0];
```

Ouvrons l'Editeur de Code de la fonction somme à quatre ports d'entrée (figure 7.4) et écrivons le code suivant dans la zone de texte associée à la phase `init` du processeur par défault.

```
@OUT(out_1)[0] = @IN(in_1)[0] + @IN(in_2)[0] + @IN(in_3)[0] + @IN(in_4)[0];
```

7.3 Construction du modèle complet

Dans une véritable application, notre travail s'arrêterait avec les adéquations de SynDEx des deux contrôleurs sur leur architecture cible. Mais pour des raisons pédagogiques, nous allons simuler le système complet (à savoir avec la dynamique des voitures) afin de vérifier que notre application SynDEx donne les mêmes résultats que Scicos.

Après avoir inséré la référence en vitesse `ref_vit` (que l'on définira comme étant un générateur de signaux carrés) et deux sorties (`vitesse` de $\mathcal{C}_1$ et `distance` entre les deux voitures), la représentation finale de l'application sous SynDEx est montrée figure 7.5. Dans les sections suivantes nous étudions l'intérieur des boites hiérarchiques `voiture1` et `voiture2`.

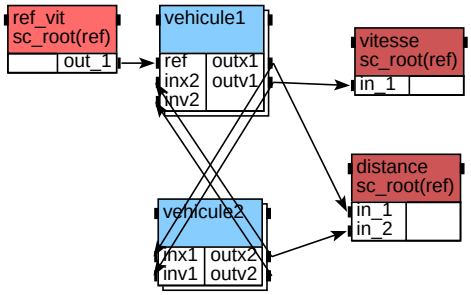


Fig. 7.5 – L'algorithme principal de SynDEx

7.3.1 Les véhicules et leurs contrôleurs

Dans les diagrammes suivants (de 7.6 à 7.9), les blocs dénotés par **meca** contiennent la dynamique d'une voiture. Donnons les contrôleurs des deux voitures.

La voiture $\mathcal{C}_1$ et son contrôleur

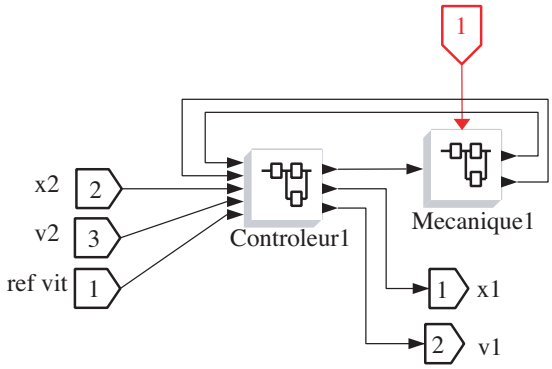


FIG. 7.6 – La dynamique de la voiture $\mathcal{C}_1$ et son contrôleur sous Scicos.

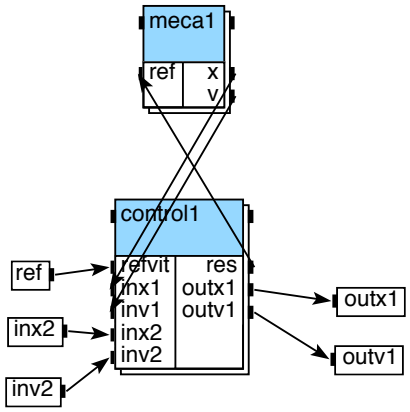


FIG. 7.7 – La dynamique de la voiture $\mathcal{C}_1$ et son contrôleur sous SynDEx.

La voiture $\mathcal{C}_2$ et son contrôleur

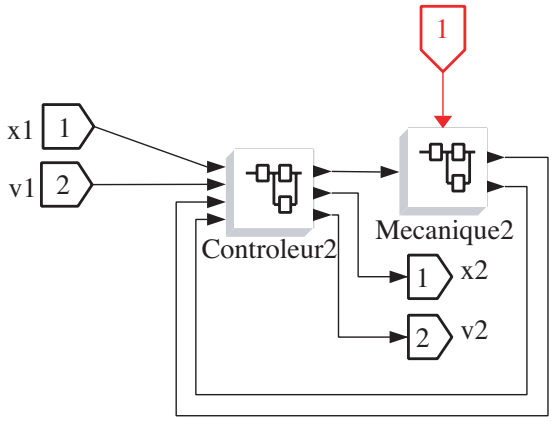


FIG. 7.8 – La dynamique de la voiture $\mathcal{C}_2$ et son contrôleur sous Scicos.

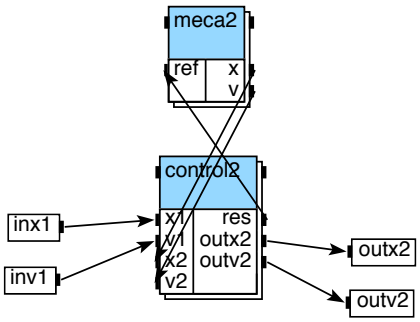


FIG. 7.9 – La dynamique de la voiture $\mathcal{C}_2$ et son contrôleur sous SynDEx.

7.3.2 Dynamique d'une voiture

La dynamique d'une voiture est représentée par un block diagramme Scicos (opérations SynDEx) dans la figure 7.10 (figure 7.11), où l'entrée 1 (`ref`) est l'accélération de la voiture. La première intégrale donne sa vitesse et la deuxième intégrale donne sa position.

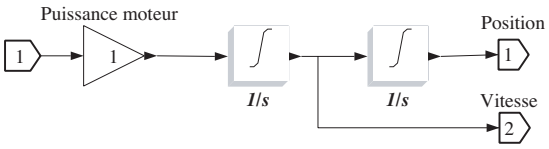


FIG. 7.10 – La dynamique d'une voiture dessinée en bloc diagramme de Scicos (temps continu).

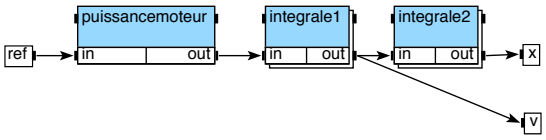


FIG. 7.11 – La dynamique d'une voiture dessinée avec opértions de SynDEx.

SynDEx manipule seulement des modèles temps discret (et donc, pas de temps continu) et il n'est pas capable de manipuler des boucles algébriques implicites. C'est pourquoi, une boucle SynDEx doit contenir au moins un délais ($1/z$). Par conséquent, notre application Scicos qui est un système dynamique temps continu doit être convertie en temps discret pour pouvoir être utilisée dans SynDEx.

L'équation différentielle $\dot{x} = u$ est discrétisée d'une façon simple en utilisant un schéma de Euler. Notons h le pas de discrétisation et x_0 une valeur initiale arbitraire. Le système discret s'écrit :

$$x_{n+1} - x_n = uh \tag{7.2}$$

Finalement, le système (7.2) est donné en Scicos (SynDEx) par la figure. 7.12 (7.13). Notons que la variable h, stockée dans le contexte de Scicos, est utilisée dans l'entrée du gain et dans la définition de l'horloge. Dans SynDEx, h est défini en tant que paramètre dans la définition d'un gain et l'horloge est utilisée directement dans le code source des opérations.

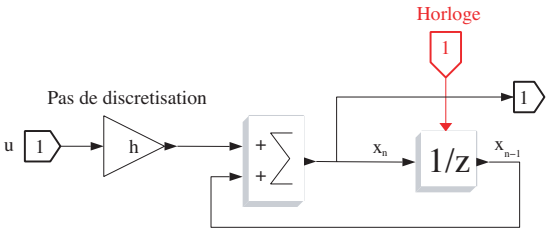


FIG. 7.12 – Une intégrale discrétisée dans Scicos (temps discret).

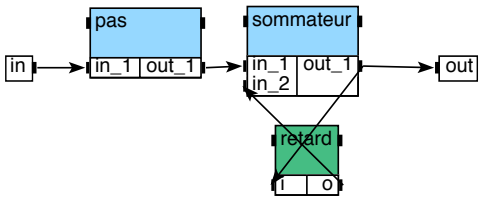


FIG. 7.13 – Une intégrale discrétisée dans SynDEx.

7.3.3 Edition du code source associé aux autres opérations

Nous allons associer du code C à l'entrée et aux deux sorties du système. Le code est généré pour le processeur par défaut.

Entrée

Dans l'application Scicos, une entrée est un générateur de signaux carrés. Sous SynDEx, on simule ce type de générateur par une lecture des valeurs dans un fichier texte (nommé **square_wave_generator.txt**). Nous utilisons pour cela les fonctions `fopen`, `fclose` et `fscanf` (provenant de la bibliothèque `stdio.h`). Nous utilisons également des assertions (provenant de la bibliothèque `assert.h`) pour s'assurer que le fichier a bien été ouvert.

Pour le moment, il faut supposer qu'il existe un tableau de flux `FILE*` (la structure retournée par la fonction `fopen`) appelé `fd_array` et une variable appellée `timer` pour simuler un pseudo timer. Une entrée à un paramètre appelé `POSI_ARRAY` pour se souvenir de la case du tableau où la structure de flux a été enregistrée.

Ouvrons l'Éditeur de Code de l'entrée et écrivons le code suivant dans la zone de texte de la phase `Init` du processeur par défaut.

```
timer = 0;

fd_array[@PARAM(POSI_ARRAY)] = fopen("square_wave_generator.txt", "r");
assert(fd_array@PARAM(POSI_ARRAY)] != NULL);
```

Parce que dans l'application Scicos, nous avons initialisé la période de l'horloge à 5 et défini le pas de discrétisation h à 0.001, nous devons, dans l'application SynDEx, envoyer 5000 fois de suite la même valeur (la 5000 ième fois permet de lire une nouvelle valeur dans le fichier). Pour compter, nous utilisons la variable `timer`.

Ecrivons le code suivant dans la zone de texte de la phase `Loop` du processeur par défaut.

```
timer = (timer + 1) % 5000;

if (timer == 1)
    fscanf(fd_array[@PARAM(POSI_ARRAY)], "%f\n", &data);
@OUT(out_1)[0] = data;
```

Nous devons libérer la mémoire en fermant le flux du fichier. Ecrivons le code suivant dans la zone de texte de la phase `End` du processeur par défaut.

```
fclose(fd_array[@PARAM(POSI_ARRAY)]);
```

Sortie de la vitesse

Une sortie enregistre les valeurs des états du système dans un fichier. C'est pourquoi elle a aussi, un paramètre appelé `POSI_ARRAY` pour se souvenir de la case du tableau où la structure de flux du fichier a été enregistrée. Ouvrons l'Éditeur de Code d'une sortie et écrivons le code suivant dans la zone de texte de la phase `Init` du processeur par défaut.

```
fd_array[@PARAM(POSI_ARRAY)] = fopen("actuator_@TEXT(@PARAM(POSI_ARRAY))", "w");
assert(fd_array[@PARAM(POSI_ARRAY)] != NULL);
```

La phase de boucle permet de stocker les valeurs des états du système.

```
fprintf(fd_array[@PARAM(POSI_ARRAY)], "%E\n", @IN(in_1)[0]);
```

Nous devons libérer la mémoire en fermant le flux du fichier. Ecrivons le code suivant dans la zone de texte de la phase `End` du processeur par défaut.

```
fclose(fd_array[@PARAM(POSI_ARRAY)]);
```

Sortie de la distance entre les voitures

Cette sortie contrairement à la vitesse a deux ports d'entrée (un port par position de voiture). Le code des phases `Init` et `End` sont identiques. Seul le code associé à la phase `Loop` diffère :

```
fprintf(fd_array[@PARAM(POSI_ARRAY)], "%E\n", (@IN(in_1)[0] - @IN(in_2)[0]));
```

7.3.4 Génération du noyau applicatif

Génération du fichier `mycar_sdc.m4x`

SynDEx génère le fichier `mycar_sdc.m4x` (où `mycar` est le nom de l'application) donné ci-dessous :

```
define(`sommateur5_def',`ifelse(
    processorType_,processorType_,`ifelse(
         MGC,`INIT',``'',
         MGC,`LOOP',``$6[0] = $5[0] + $4[0] + $3[0] + $2[0] + $1[0];'',
         MGC,`END',``'')')')

define(`sommateur4_def',`ifelse(
    processorType_,processorType_,`ifelse(
         MGC,`INIT',``'',
         MGC,`LOOP',``$5[0] = $4[0] + $3[0] + $2[0] + $1[0];'',
         MGC,`END',``'')')')

define(`sommateur2_def',`ifelse(
    processorType_,processorType_,`ifelse(
         MGC,`INIT',``'',
         MGC,`LOOP',``$3[0] = $2[0] + $1[0];'',
         MGC,`END',``'')')')

define(`gain_def',`ifelse(
    processorType_,processorType_,`ifelse(
         MGC,`INIT',``'',
         MGC,`LOOP',``$3[0] = $2[0] * $1;'',
         MGC,`END',``'')')')

define(`vitesse_def',`ifelse(
    processorType_,processorType_,`ifelse(
         MGC,`INIT',``fd_array[$1] = fopen("actuator_'$1`", "w");
                      assert(fd_array[$1] != NULL);'',
         MGC,`LOOP',``fprintf(fd_array[$1], "%E\n", $2[0]);'',
         MGC,`END',``fclose(fd_array[$1]);'')')')


define(`distance_def',`ifelse(
    processorType_,processorType_,`ifelse(
         MGC,`INIT',``fd_array[$1] = fopen("actuator_'$1`", "w");
                      assert(fd_array[$1] != NULL);'',
         MGC,`LOOP',``fprintf(fd_array[$1], "%E\n", ($1[0] - $2[0]));'',
         MGC,`END',``fclose(fd_array[$1]);'')')')

 define(`ref_vit_def',`ifelse(
     processorType_,processorType_,`ifelse(
         MGC,`INIT',``k = 0;
                      fd_array[$1] = fopen("square_wave_generator.txt", "r");
                      assert(fd_array[$1] != NULL);'',
         MGC,`LOOP',``k = (k + 1) % 5000;
                      if (k == 1)
                          fscanf(fd_array[$1], "%f\n", &data);
                     $2[0] = data;'',
         MGC,`END',``fclose(fd_array[$1]);'')')')
```

Génération manuelle du fichier `mycar.m4x file`

On ne peut pas utiliser directement le fichier générique **mycar.m4x** crée par SynDEx puis qu'il manque le code permettant d'initialiser les variables et l'appel à certaines bibliothèques. Il faut l'éditer en mettant le code suivant :

```
define(`dnldnl',``// '')
define(`NOTRACEDEF')
define(`NBITERATIONS',``20000'')

define(`proc_init_',`
    FILE *fd_array[10];
    float data;
    int timer;')

include(`mycar_sdc.m4x')

divert
    #include <stdio.h> /* for printf */
    #include <assert.h>
```

Où la macro `proc_init_` permet d'insérer les déclarations des variables locales entre la fonction `main` et les initialisations des opérations. Notons que la boucle principale du programme est définie de façon générique par une boucle de `NBITERATIONS` itérations. Dans notre exemple, `NBITERATIONS` est le nombre de lignes du fichier `square_wave_generator.txt` (qui, pour rappel, sert à simuler un générateur de signaux carrés). Enfin, les appels aux bibliothèques se font après l'appel du fichier `mycar_sdc.m4x`.

7.4 Compilation et simulation

7.4.1 Simulation sous Scicos

Scicos permet de simuler le modèle. Une fenêtre représentée par la figure 7.14. Les valeurs de trois états internes du système (axe des ordonnées) sont dessinées en fonction du temps (axe des abscisses). Nous avons :

- les vagues du générateur, de couleur rouge (bas),
- la vitesse de la voiture $\mathcal{C}_1$, de couleur noire (haut),
- la distance entre les deux voitures, de couleur vert (milieu).

D'après nos diagrammes, le système est stable et donc fonctionne correctement. Nous ne chercherons pas à améliorer le fonctionnement des contrôleurs.

7.4.2 Exécution de l'application SynDEx

Avec une architecture mono-processeur

SynDEx n'étant pas conçu pour simuler des modèles, nous devons supposer que le modèle fonctionne correctement. Cependant, dans notre cas, il est facile de comparer les résultats obtenus par SynDEx avec ceux obtenus par Scicos.

Dans cette sous-section nous supposerons que l'architecture est constituée d'un unique opérateur nommé **root**. Obtenir les exécutables à partir des codes générés par SynDEx se fait très simplement grâce au noyau générique fourni sur le site officiel de SynDEx. Tout d'abord, il

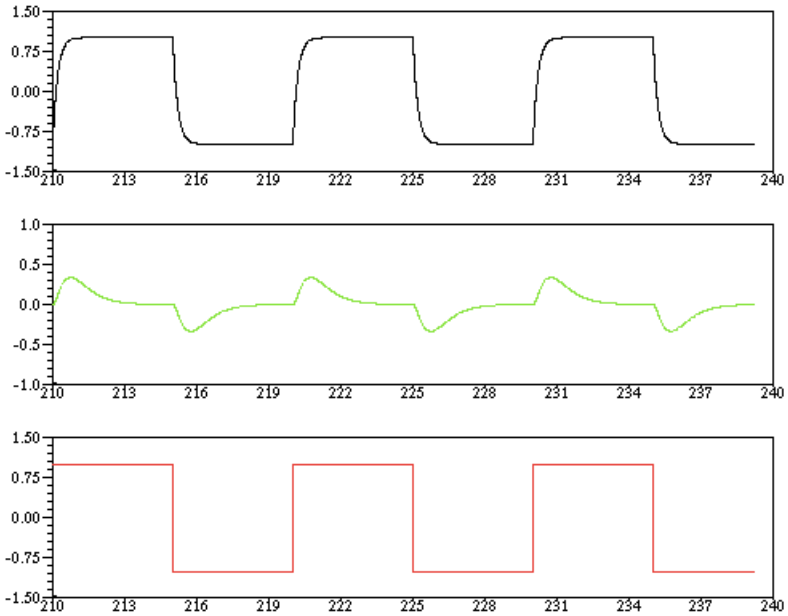


FIG. 7.14 – Diagramme obtenu avec les valeurs $[0, -5, 0, 0, -5]$ des gains du contrôleur de $\mathcal{C}_1$ et les valeurs $[4, 4, -4, -4]$ des gains du contrôleur de $\mathcal{C}_2$.

faut copier les bons dossiers `Unix_C_TCP` et `libraries` ainsi que leurs contenus (fichiers macros m4) dans le même répertoire contenant les codes m4 et m4x générés par SynDEx. Ensuite, il faut créer manuellement deux fichiers : **GNUmakefile** et **mycar.m4m**. Le makefile générique nommé **GNUmakefile** ressemble à :

```
A  = mycar

export Macros_Path = ./macros/Unix_C_TCP
export Libraries_Path = ./macros/libraries
export M4PATH = $(Macro\_Path):$(Libraries_Path)

CFLAGS = -DDEBUG
VPATH = $(M4PATH)

.PHONY: all
all: $(A).mk $(A).run

$(A).mk: $(A).m4 syndex.m4m U.m4m $(A).m4m
    gm4 $< >$@

root.libs =
p.libs =

include $(A).mk
```

Pour fonctionner ce makefile a besoin d'un fichier nommé **mycar.m4m** (m4m pour makefile). Ce fichier permet d'associer un nom d'opérateur de l'architecture à un nom d'hôte d'un ordinateur qu'on supposera pouvoir se connecter à distance avec.

```
define(`root_hostname_',Solarion.local)dnl
```

La commande `gmake` dans un terminal :
- lance le macro-processeur m4 sur tous les fichiers `root.m4*` et `mycar*.m4*` et obtient les fichiers correspondant en langage C,
- compile chaque fichier C en un exécutable,
- lance l'exécutable.

Dans notre cas, l'exécutable génère les fichiers de sauvegarde que l'on peut comparer avec ceux obtenus sous Scicos et vérifier qu'ils sont identiques.

Avec une architecture multi-processeurs

Pour simuler notre application, de la façon la plus réaliste possible, nous remplaçons l'ancienne architecture par une nouvelle contenant cinq opérateurs communicant par un média de communication. Comme nous n'avons pas le matériel nécessaire, l'architecture réelle sera simulée par cinq ordinateurs communiquant par TCP/IP (figure 7.15).

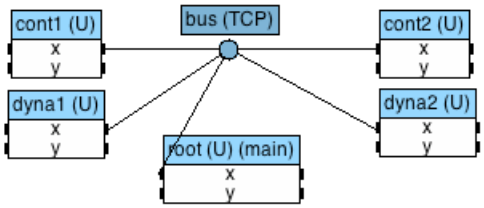


Fig. 7.15 – L'architecture avec cinq opérateurs.

Nous avons cinq opérateurs parce que nous avons deux contrôleurs, deux systèmes physiques ainsi qu'un appareil gérant l'entrée et les sorties du système.

Comme on l'a expliqué dans la partie contexte de ce rapport, SynDEx utilise une heuristique pour regrouper au mieux les opérations sur les opérateurs de l'architecture. Cependant SynDEx à la possibilité de placer arbitrairement une opération sur un opérateur : ce sont les **software componants**. Créons en cinq puis :
- associons le contrôleur de $\mathcal{C}_1$ avec l'opérateur nommé **cont1** et de $\mathcal{C}_2$ avec **cont2**,
- associons la partie mécanique de $\mathcal{C}_1$ avec **dyna1** et de $\mathcal{C}_2$ avec **dyna2**,
- regroupons l'entrée et les sorties sur un même opérateur nommé **root**.

L'adéquation et la génération de code ne posent aucun problème sous SynDEx. Pour compiler, il suffit de compléter le fichier `mycar.m4m` en associant le nom des nouveaux opérateurs avec le nom de la machine hôte associée.

La commande `gmake` tapée dans un terminal compile et lance automatiquement les cinq exécutables sur les cinq ordinateurs. Comme dans la simulation précédente la machine simulant l'opérateur **root** génère les fichiers de sauvegarde que l'on peut comparer avec ceux obtenus sous Scicos et vérifier que leurs contenus sont identiques.

Chapitre 8

Planning du stage

Le planning des travaux réalisés pendant les quatre mois de ce stage peut être résumé de la façon suivante :

- Une période de deux semaines pour comprendre la méthodologie AAA en lisant de la documentation. De maîtriser le fonctionnement de SynDEx en lisant des tutoriels.
- Deux périodes de deux semaines pour concevoir l'Éditeur de code sous la forme d'une IHM en CamlTk. Deux semaines pour finir de déboguer l'IHM et faire un portage sur Mac OS X.
- Trois semaines pour écrire un premier tutoriel concernant le fonctionnement de l'Éditeur de code.
- Durant les trois premiers mois, en parallèle au stage, j'ai étudié une introduction à l'automatique, j'ai appris à utilisé Scicos l'Éditeur de bloc diagrammes du logiciel Scilab. Cette initiative personnelle fut encouragée par mon maître de stage afin de créer un exemple en automatique pour détecter d'éventuels bogues dans SynDEx.
- Trois semaines furent nécessaire pour écrire un deuxième tutoriel afin d'expliquer le fonctionnement de mon exemple. Il simule le fonctionnement d'une voiture suivant une autre et gardant une distance alors que la première peut accélérer ou ralentir.
- J'ai participé à trois réunions avec un des utilisateurs-clients principaux de SynDEx.
- Le mois de décembre fut entièrement utilisé afin de rédiger la documentation de SynDEx et les rapports pour l'EPITA.

Le planning précis est donné dans les schémas 8.1, 8.2 et 8.3.

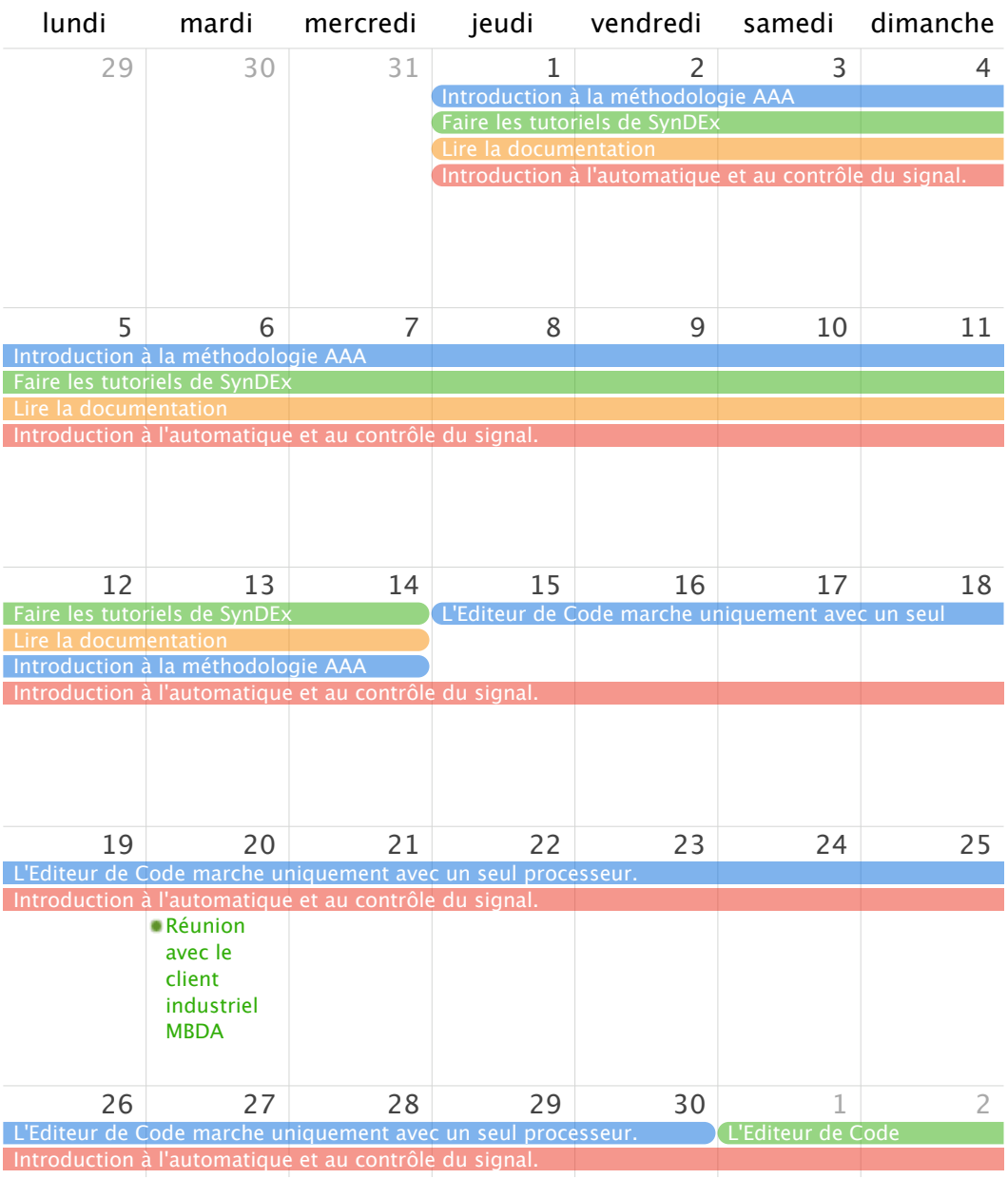


FIG. 8.1 – Planning de Septembre

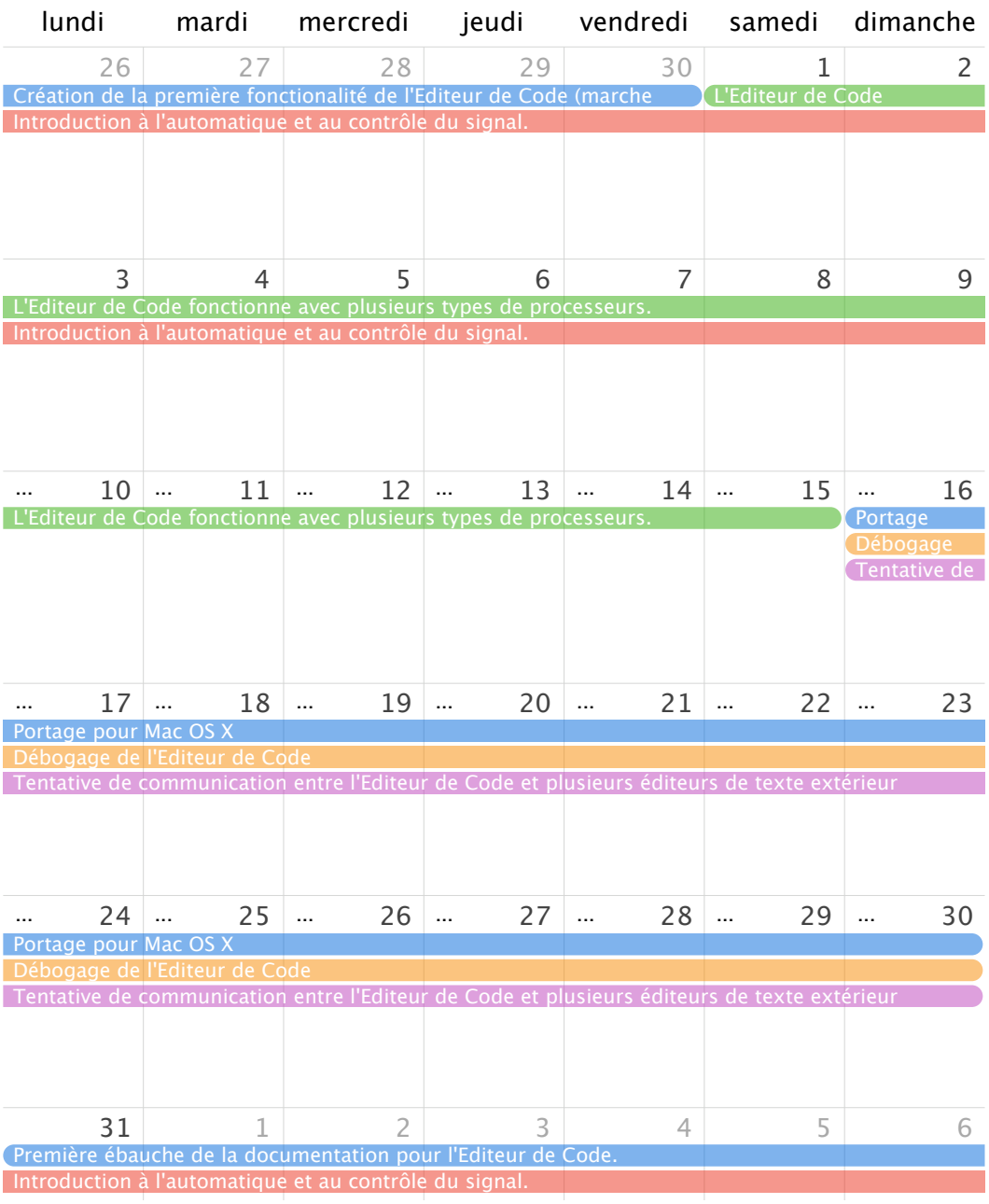


FIG. 8.2 – Planning de Octobre

FIG. 8.3 – Planning de Novembre

Conclusion

8.1 Appréciations des utilisateurs de l'Éditeur de Code

Ce travail était une demande par l'un (MBDA) des principaux utilisateur industriel de SynDEx dont les commentaires après utilisation ont été les suivants :

- Les ports d'entrées-sorties d'un opérateur ne sont plus désignés par leurs positions relatives mais désormais par leurs noms. Ceci permet d'éviter de nombreux problèmes lors de la génération de code lorsqu'elle devait être fait à la main.
- Le code source est maintenant lisible, puisque le squelette en langage m4 a disparu et que l'Éditeur de code permet de le formater.
- Le langage m4 est peu connu des développeurs et délicat à manipuler. Des erreurs d'équilibrage de guillemets conduisaient à des erreurs difficiles à identifier. Des mécanismes internes du m4 SynDEx n'ont plus à être connu du concepteur.
- Il n'y a désormais plus de manipulation séparée du code source associé et du modèle puisque tous les moyens d'édition de code sont directement intégrés dans l'IHM de SynDEx.

8.2 Expérience acquise

Ce stage m'a permis :

- de découvrir les problèmes de distribution et d'ordonnancement temps réel,
- de mieux comprendre les problèmes liés aux exécutifs temps réel,
- d'étudier un problème d'automatique,
- de manipuler les logiciels SynDEx, Scilab, Scicos,
- de progresser dans la connaissance du langage OCaml (pour l'anecdote : l'équipe AOSTE partage le même bâtiment que les créateurs de Caml) et ses outils (Camllex, Camlyacc, CamlTk).

8.3 Conclusion générale

Mon stage fut passionnant grâce à un cadre agréable et à la bonne humeur des membres du projet AOSTE. Le travail réalisé pendant le stage est déjà inclus dans la dernière version de SynDEx (version 6.8.4) disponible gratuitement sur le Web. Une application complète servant de tutoriel à SynDEx a été réalisée. Elle m'a permis de m'initier à l'Automatique. Malheureusement, par manque de temps je n'ai pas pu étudier en détail les problèmes d'ordonnancement qui sont au coeur de SynDEx.

Table des figures

Bibliographie

[1] Thierry Grandpierre, Christophe Lavarenne, Yves Sorel, *Modèle d'exécutif distribué temps réel SynDEx*, INRIA, 1998.

[2] Andreas Ermedahl, *Schedulability Analysis Assignment*, 2004.

[3] La page principale de SynDEx : http ://www-rocq.inria.fr/syndex/

[4] Stephen L. Campbell, Jean-Philippe Chancelier and Ramine Nikoukhah, *Modeling and Simulation in Scilab/Scicos*, Springer, 2005.

[5] La page principale de Scilab : http ://www-rocq.inria.fr/syndex/

[6] La page principale de Scicos : http ://www.scicos.org/

[7] Pierre Faure et Michel Depeyrot, *Eléments d'automatique*, Dunod, 1974.

[8] Karl Johan Åström *Control System Design* ME155A.

[9] La page principale de Karl Johan Åström : http ://www.control.lth.se/∼kja/

[10] Philipe Baptiste, Emmanuel Neron, Francois Soud *Modèles et algorithmes en ordonnancement*, Ellipses 2004.