DEA SETI
Ecole Doctorale STITS
Université de Paris XI

RAPPORT DE STAGE

De la Spécification Multi-Modèle d'Algorithme à l'Implantation Optimisée sur Architecture Embarquée avec Contraintes Temps Réel

Nikom SUVONVORN
Projet OSTRE
INRIA
31 Mars - 30 Septembre 2003

Supervisé par

Yves SOREL
INRIA - Domaine de Voluceau - Rocquencourt B.P.105
78153 Le Chesnay Cedex - France
Tél : (1) 39 63 55 11 - email : Yves.Sorel@inria.fr

Remerciements

Je tiens à remercier tout d'abord Yves Sorel, Directeur de Recherche à l'INRIA, de m'avoir accueilli dans l'équipe OSTRE durant ce stage.

Je remercie Nicolas Pernet, doctorant de l'équipe OSTRE, d'avoir su m'écouter et me conseiller par ses nombreuses idées et commentaires.

Je tiens enfin à remercier l'ensemble de l'équipe OSTRE, et tout particulièrement les ingénieurs pour les questions auxquelles ils ont pris le temps de répondre.

Contents

Remerciements **1**

Introduction **7**

I Etat de l'art 9

1 Introduction à AAA/SynDEx **11**

1.1 Introduction . . . 11
1.2 Systèmes temps réel . . . 11
1.2.1 Définition d'un système réactif temps réel embarqué . . . 12
1.2.2 Besoin d'architectures spécialisées . . . 12
1.2.3 Nécessité d'outils de spécification . . . 12
1.3 Méthodologie *AAA* . . . 13
1.3.1 Principes généraux de la méthode . . . 13
1.3.2 Modèles d'algorithmes et d'architectures . . . 14
1.3.3 Heuristique de distribution et d'ordonnancement . . . 15
1.3.4 Génération d'exécutif . . . 15
1.3.5 Présentation de SynDEx . . . 16

2 Introduction à Scilab/Scicos **20**

2.1 Les signaux . . . 20
2.1.1 Les types de signaux . . . 20
2.1.2 La terminologie . . . 20
2.1.3 Les signaux d'activations . . . 21
2.1.4 Les signaux réguliers . . . 22
2.1.5 Le conditionnement direct et hérité . . . 23
2.1.6 Le super-bloc . . . 24
2.1.7 La synchronisation des événements . . . 24
2.2 Les différents types de blocs . . . 25
2.2.1 Les blocs Standards . . . 25
2.2.2 Les blocs Zcross . . . 26
2.2.3 Les blocs Synchro . . . 26
2.2.4 Les blocs Memo . . . 27
2.2.5 Restriction au domaine nous concernant . . . 27

II Graphe hiérarchique d'unités fonctionnelles 29

1 Graphe hiérarchique d'unités fonctionnelles **31**

1.1 Définition de l'unité fonctionnelle . . . 31
1.1.1 Sommet "switch" . . . 32
1.1.2 Sommet "func$_i$" . . . 32
1.1.3 Sommet "selector" . . . 33

1.2 Graphe d'unités fonctionnelles 33
1.3 Graphe hiérarchique d'unités fonctionnelles 33
1.4 Fonctions spéciales 35
1.4.1 Fonction "sensor" 35
1.4.2 Fonction "actuator" 35
1.4.3 Fonction "delay" 35
1.4.4 Fonction "super" 35
1.4.5 Fonction "dummy" 35
1.4.6 Fonction "root" 35

2 Optimisation du graphe hiérarchique d'unités fonctionnelles 36
2.1 Détection et élimination de la redondance des "switch" 36
2.2 Détection et élimination de la redondance des "selector" 37
2.3 Détection et élimination des tests inutiles 38

III Application 41

1 Transformation du graphe Scicos en un graphe d'algorithme SynDEx 43
1.1 Transformation du graphe de Scicos en un graphe hiérarchique d'unités fonctionnelles 43
1.1.1 Transformation des blocs "Synchro" 44
1.1.2 Transformation des blocs "Standard" 45
1.2 Traduction du graphe hiérarchique d'unités fonctionnelles en un graphe d'algorithme SynDEx 49
1.2.1 Le conditionnement dans SynDEx 49
1.2.2 Traduction 51
1.3 Implémentation de la transformation 54
1.4 Exemples 58

Conclusion 64

List of Figures

1.1 Définition d'un système réactif temps réel 12
1.2 Un exemple de graphe flot de données 14
1.3 Un exemple d'architecture 15
1.4 principes de SynDEx 16
1.5 Graphe de l'algorithme de l'application ega2c 17
1.6 Graphe de l'architecture l'application ega2c 18
1.7 Graphe temporel pour l'application ega2c 19

2.1 La représentation d'un système hybride dans l'éditeur Scicos. 21
2.2 Un signal Scicos et l'ensemble des temps d'activations du bloc. 22
2.3 Le conditionnement des blocs. 23
2.4 Le conditionnement continu dans la figure **??** 24
2.5 Les résultats de la simulation (cf.figure **??**) 25
2.6 L'héritage (intermédiaire) dans la figure **??** 26
2.7 L'héritage (total) dans la figure **??** 27

1.1 L'unité fonctionnelle. 32
1.2 Graphe d'unités fonctionnelles. 34

2.1 (a) Le graphe hiérarchique d'unités fonctionnelles ; (b) le même graphe après élimination de la redondance du sommet "switch". 37
2.2 (a) Le graphe hiérarchique d'unités fonctionnelles ; (b) le même graphe après élimination de la redondance des sommets "selector". 38
2.3 (a) Un graphe hiérarchique d'unités fonctionnelles ; (b) le même graphe après suppression du test de la donnée c de valeur 0 et du sommet "dummy" de l'unité fonctionnelle consommatrice et remplacement du sommet "$func_2$" de l'unité fonctionnelle productrice par une fonction "dummy" ; (c) le graphe après inclusion de l'unité fonctionnelle consommatrice dans le sommet "super" de l'unité fonctionnelle productrice ; (d) le sous-graphe du sommet "super". 40
2.4 (a) Un graphe hiérarchique d'unités fonctionnelles ; (b) le même graphe après suppression du test de la donnée c de valeur 0 et du sommet "dummy" de l'unité fonctionnelle consommatrice et remplacement du sommet "$func_2$" de l'unité fonctionnelle productrice par une fonction "dummy" ; (c) le graphe après inclusion de l'unité fonctionnelle consommatrice dans le sommet "super" de l'unité fonctionnelle productrice et duplication le sommet "selector" pour chacune des sorties du sommet "super" ; (d) le sous-graphe du sommet "super". 40

1.1 (a) Le bloc "if_then_else" ; (b)(c) l'unité fonctionnelle "if_then_else". 44
1.2 (a) Le bloc "sampling_and_hold" ; (b)(c) l'unité fonctionnelle "sampling_and_hold". 45
1.3 Un graphe Scicos. 46
1.4 La transformation du graphe Scicos (la figure **??**) en un graphe hiérarchique d'unités fonctionnelles. 47

1.5 L'optimisation du graphe hiérarchique d'unités fonctionnelles (la figure ??) ; (a)(b)(c) représentant des étapes différentes de l'optimisation et (d) représentant le sous-graphe du sommet "super". 48
1.6 Un sommet conditionné dans SynDEx. 50
1.7 (a) Un graphe d'unités fonctionnelles représente le sommet conditionné de la figure ?? ; (b) le graphe d'unités fonctionnelles optimisé. 50
1.8 La traduction du graphe hiérarchique d'unités fonctionnelles (la figure ??) en un graphe d'algorithme SynDEx. 53
1.9 Un graphe Scicos. 59
1.10 La transformation du graphe Scicos (la figure ??) en un graphe hiérarchique d'unités fonctionnelles. 60
1.11 L'optimisation du graphe hiérarchique d'unités fonctionnelles (la figure ??). 62
1.12 La traduction du graphe hiérarchique d'unités fonctionnelles (la figure ??) en un graphe d'algorithme de SynDEx. 63

Introduction

Afin de valider conjointement ma dernière année ESME-Sudria, Majeure l'informatique, et le DEA Système Eléctronique et Traitement d'Information, j'effectue un stage de 6 mois dont le sujet est "De la Spécification multi-modèle d'algorithme à l'implantation optimisée sur architecture embarquée avec contraintes temps réel", qui se déroule à l'INRIA, sous la responsabilité de Mr Yves Sorel.

L'équipe OSTRE (Optimisation des Systèmes Temps Réel Embarqués), dont il a la direction, a développé un outil SynDEx, acronyme de **Syn**chronized **D**istributed **Ex**ecutive, qui a pour but de pouvoir, à partir d'un algorithme graphe flot de données et d'un graphe d'architecture, réaliser l'adéquation, c'est à dire l'implantation optimisée de l'algorithme sur l'architecture, et générer un code distribué synchronisé pour systèmes répartis hétérogènes avec respect de contraintes temps réel.

Contexte

Faisant suite à l'IRIA créé en 1967, l'INRIA, acronyme d'Institut Nationale de Recherche en Informatique et Automatique, est un établissement public à caractère scientifique et technologique placé sous la double tutelle du ministre chargé de la Recherche et de l'Industrie. Son objectif est d'effectuer une recherche de haut niveau, et d'en transmettre les résultats aux étudiants, au monde économique et aux partenaires scientifiques et industriels. A Rocquencourt, près de Versailles, se trouve le siège de l'INRIA ainsi qu'une de ses cinq unités de recherche. Les autres unités de recherche se situent en Lorraine (Nancy/Metz), à Rennes, à Sophia-Antipolis et à Grenoble. Les chercheurs en mathématiques, automatique et informatique de l'INRIA collaborent dans les quatre thèmes suivants :

- réseaux et systèmes,
- calcul symbolique et génie logiciel,
- intéraction homme-machine, données, connaissances,
- simulation et optimisation de systèmes complexes.

Dans chaque thème existe plusieurs équipes qui étudient plusieurs axes de recherches. L'équipe OSTRE fait partie du thème "réseaux et systèmes".

Mon stage se situe dans le cadre du développement de SynDEx, logiciel d'aide à la conception niveau système reposant sur la méthodologie AAA. SynDEx ne permet, jusqu'à présent, la spécification d'algorithme que sous forme de graphes flot de données. Ce formalisme est bien adapté à la représentation d'algorithme de traitement de données comme les lois de commande ou les filtres. Mais, bien que le principe de conditionnement ait été introduit et, avec lui une certaine manière d'inclure du contrôle dans les algorithmes, exprimer un algorithme de contrôle complexe avec ce formalisme reste fastidieux. Les graphes flot de contrôle (Réseaux de Petri, Statecharts) étant plus adaptés pour cela, il est intéressant de permettre le multi-formalisme pour la spécification d'algorithme : exprimer la partie traitement de données en graphe flot de données et la partie contrôlant les changements de modes, de paramètres, en graphe flot de contrôle. Le format flot de données permettant une distribution plus efficace que le format flot de contrôle, il faut unifier

la spécification multi-formalisme en une spécification flot de données pour une implantation optimisée. Une traduction d'automates SyncCharts en flot de données SynDEx a été réalisée dans ce but [?].

Le stage s'articule autour du problème de spécification multi-modèle selon les points suivants :

- réflexion sur la manière de traduire un modèle flot de contrôle ou incluant du contrôle en un modèle flot donnée,
- Etude et réalisation d'une interface Scilab/Scicos → SynDEx,
- Optimisation du graphe d'algorithme généré en terme de nombre de sommets et de test, et donc du temps de calcul global.

Méthodologie

Dans ce rapport, nous présentons une méthode de transformation des modèles flot de contrôle en flot de données en utilisant un graphe intermédiaire. L'idée d'utilisation de graphe intermédiaire pour la transformation est très employée dans l'analyse de données et la transformation de programme : "static single assignment (SSA)" [?], "dependence flow graphe (DFG)" [?][?] et "Sparse Evaluation Graphs (SEGs)" [?]. Dans notre transformation, nous proposons un type de graphe intermédiaire appelé *graphe hiérarchique d'unités fonctionnelles.*

La transformation se déroule en trois étapes :

- transformation du modèle flot de contrôle SynCharts et de la partie contrôle de Scilab/Scicos en un graphe hiérarchique d'unités fonctionnelles,
- optimisation du graphe hiérarchique d'unités fonctionnelles,
- traduction du graphe hiérarchique d'unités fonctionnelles en un graphe d'algorithme SynDEx.

Dans la troisième partie, nous présenterons la transformation particulière Scilab/Scicos → SynDEx.

Part I

Etat de l'art

Chapter 1

Introduction à AAA/SynDEx

1.1 Introduction

La complexité sans cesse croissante des applications et les contraintes temps-réel conduisent à utiliser des architectures multi-processeurs (parallèles, distribuées) lorsqu'il s'agit d'exécuter des algorithmes. Le logiciel *SynDEx* (**Syn**chronous **D**istributed **Ex**ecutive) fournit une aide à l'implantation temps-réel multi-processeur de ces algorithmes en déchargeant au maximum l'utilisateur des tâches lourdes de programmation bas niveau (système).

Les classes d'applications visées sont les systèmes temps-réel intégrant traitement du signal et contrôle-commande complexes (systèmes embarqués, robotique, militaire), les systèmes d'interface homme-machine en systèmes de contrôle (conduite de procédés, surveillance) et les systèmes temps réel de l'information (systèmes de transport, de transmission, reconnaissance de formes).

Dans ce contexte où la sûreté de fonctionnement joue un rôle capital, il est indispensable de disposer d'un ensemble d'outils permettant de décomposer la réalisation d'une application en plusieurs étapes : conception et validation des algorithmes indépendamment de toute architecture, implantation progressive sur une architecture, validation. L'indépendance vis-à-vis d'une structure hôte particulière est obtenue par l'utilisation d'un langage de type synchrone (le langage *SIGNAL*, développé à l'**IRISA**), dans lequel calculs et communications internes sont supposés de durée null pendant un instant logique (correspond au traitement d'un signal flot de donnée pendant une itération). Seuls les événements de communication du programme avec son environnement sont significatifs dans la détermination de l'écoulement du temps.

Une interface *Signal/SynDEx* a déjà été développée afin d'exploiter le parallélisme potentiel d'un algorithme lors de son implantation en temps réel. Comme dans le langage flot de données *Signal, Scicos* permet de sécifier du parallélisme potentiel de façon explicite. Nous présentons dans ce chapitre les principes de base de *SynDEx* et de la méthodologie appelée Adéquation Algorithme Architecture (AAA), mise en oeuvre dans le logiciel *SynDEx*.

1.2 Systèmes temps réel

L'informatique a longtemps traité des problèmes pour lesquels le temps qui s'écoule n'intervenait pas ; seul un délai raisonnable de production des résultats était souhaité. Cette situation a évolué du fait que les applications industrielles en informatique sont de plus en plus complexes et soumises à des contraintes temporelles rigoureuses. L'informatique cherche maintenant à résoudre des problèmes pour lesquels on ne peut plus ignorer la notion de temps : les traitements doivent être effectués dans un temps borné. Les solutions informatiques ne sont donc envisageables qu'avec la mise en œuvre de nouveaux concepts intégrant les spécificités temps réel.

1.2.1 Définition d'un système réactif temps réel embarqué

Figure 1.1: Définition d'un système réactif temps réel

C'est un système qui interagit continuellement avec son environnement. Il doit produire des réactions à des stimuli venant de l'extérieur, ceci dans un temps borné. Il doit être capable de stocker les diverses informations venant de l'extérieur et de les traiter dans un délai qui ne nuit pas au contrôle de l'application en cours. Réagir trop tard peut conduire à des conséquences catastrophique pour le système lui-même ou son environnement. Un système fonctionne donc en temps réel à chaque fois qu'il sera question de contraintes de temps et que ces dernières devront être respectées. Ces contraintes peuvent être de deux types :

- *la latence* : intervalle de temps entre la réception d'une donnée engendrée par un stimulus et l'émission de la donnée engendrée par une réaction à l'issue du traitement,
- *la cadence* : intervalle de temps qui sépare la réception, par le système, de deux stimuli consécutifs.

En plus de ces contraintes de performances temps réel, le système est soumis à des contraintes technologiques d'embarquabilité (volumes, poids, consommation électrique, etc...) et des contraintes de coût d'étude et de fabrication incitant de manière générale à minimiser les ressources nécessaires à la réalisation.

Pour le concepteur d'un tel système, il s'agit donc de garantir que le comportement du système corresponde bien aux spécifications et que les contraintes soient respectées. Par exemple, les images acquises par un capteur CCD d'un système d'aide à la conduite dans les automobiles, sont traitées par un calculateur embarqué. Il repère les véhicules qui le précèdent pour avertir dans un délai borné le conducteur par une alarme sonore et visuelle lorsque la position relative des véhicules ne respecte plus la distance de sécurité.

1.2.2 Besoin d'architectures spécialisées

Le concepteur d'un système temps réel doit choisir une architecture matérielle capable de répondre en termes de puissance de calcul aux contraintes temps réel et d'embarquabilité.

Bien que les processeurs d'usage général soient de plus en plus performants, certaines applications de traitement du signal et des images nécessitent une puissance de calcul largement supérieure à celle actuellement disponible sur les processeurs les plus rapides, utilisés au cœur des stations de travail.

C'est pourquoi, pour atteindre les objectifs imposés par ces demandes en puissance de calcul, il faut recourir à des calculateurs à architecture parallèle. Malheureusement, les contraintes temps réel et d'embarquabilité sont parfois tellement fortes que les processeurs disponibles sur le marché ne suffisent plus. Cela conduit donc à utiliser, en complément des processeurs, des circuits intégrés spécialisés.

1.2.3 Nécessité d'outils de spécification

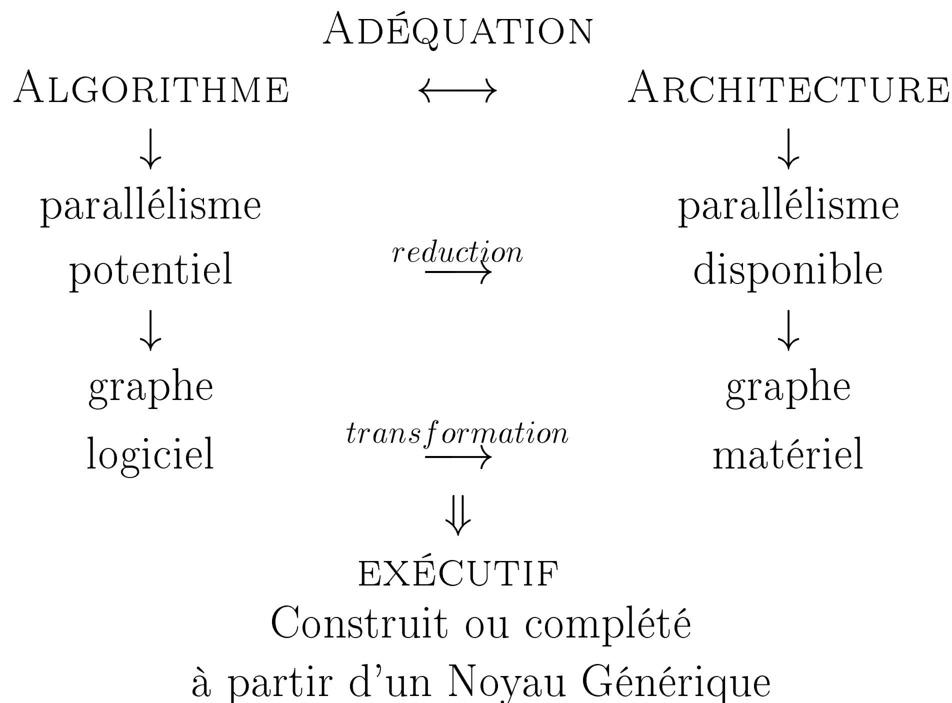
La programmation dans le domaine temps réel est loin d'être maîtrisable avec les outils utilisés actuellement. Bien que le matériel soit de plus en plus performant, il est souvent utilisé à la limite de ses possibilités car les algorithmes utilisés sont de plus en plus complexes. Par ailleurs, cette augmentation de complexité, tant au niveau des algorithmes que du matériel, entraîne une probabilité plus grande d'erreurs de conception et de pannes logicielles. Par conséquent, outre les architectures

multi-processeurs et les circuits intégrés spécialisés, on a besoin d'outils de spécification de haut niveau et d'aide à l'implantation sous contraintes de ces spécifications.

La méthodologie AAA, basée sur des modèles de graphes, a été développée dans ce but. Cette méthodologie se limite actuellement à la description d'architectures à base de processeurs mais une extension aux circuits intégrés non programmables (ASIC, FPGA) est en cours. Vous pourrez constater que grâce à cette méthodologie, l'utilisateur peut consacrer beaucoup plus de temps à la conception de son algorithme qu'à son implémentation puisque des outils logiciels associés à cette méthodologie lui apportent de l'aide. Il est ainsi déchargé de la programmation bas niveau (génération automatique d'exécutif temps réel) souvent très laborieuse et surtout très coûteuse en temps de développement.

1.3 Méthodologie *AAA*

1.3.1 Principes généraux de la méthode



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 termes 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. Il sera donc question ici d'une méthode d'optimisation.

Cette méthodologie a été concrétisée dans un logiciel appelé SynDEx.

Nous présenterons donc dans les sections suivantes les différentes étapes qui permettent d'implanter l'algorithme sur l'architecture tout en respectant les contraintes de temps. Nous consacrerons la fin de ce chapitre à la présentation du prototype d'environnement graphique SynDEx qui supporte cette méthode.

1.3.2 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). 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 soit 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. Dans le cas d'un conditionnement, la sortie n'est produite que si l'entrée booléenne est vraie. Par contre, la sortie d'un retard précède son entrée.

Les flots de données représentent un transfert itératif de données établissant ainsi une précédence d'exécution entre deux opérations.

Ce modèle permet de mettre en évidence le parallélisme potentiel de l'algorithme ainsi que sa mémoire d'état.

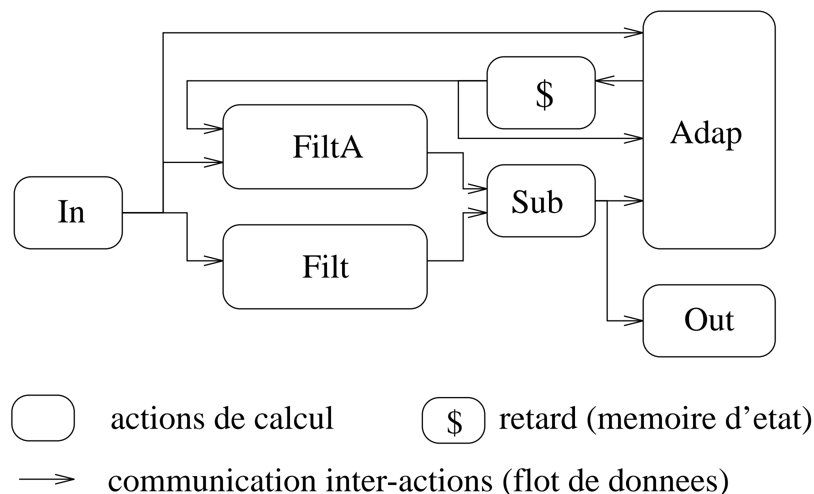


Figure 1.2: Un exemple de graphe flot de données

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 pas connecter directement deux processeurs ou deux médias (voir figure **??**). 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.

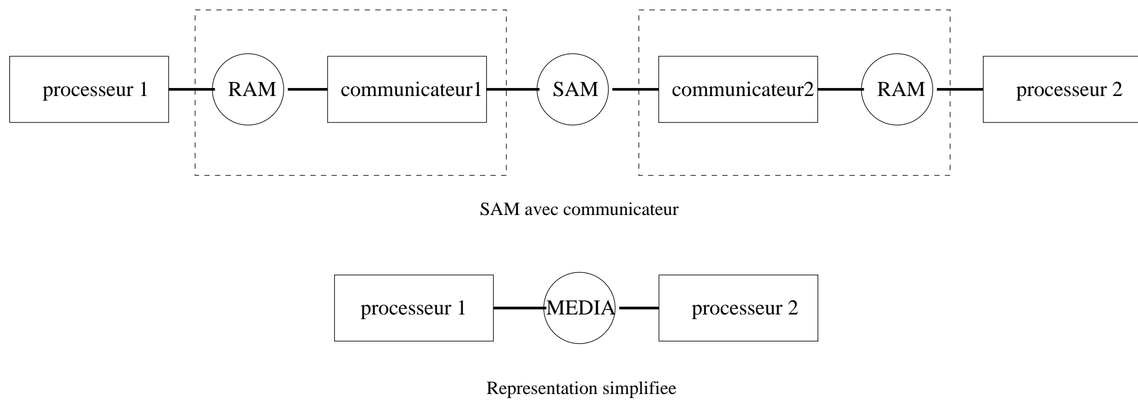


Figure 1.3: Un exemple d'architecture

Implantation de l'algorithme sur l'architecture

Comme nous l'avons déjà expliqué au début de ce chapitre, les modèles de graphes utilisés pour spécifier l'algorithme et l'architecture conduisent à formaliser l'implantation en terme de transformations de graphes. La réduction est obtenue en transformant progressivement le graphe flot de données modélisant l'algorithme jusqu'à ce qu'il corresponde au graphe matériel modélisant l'architecture. Ces transformations représentent une distribution et un ordonnancement des opérations sur les processeurs et des communications inter-processeurs sur les liaisons physiques.

1.3.3 Heuristique de distribution et d'ordonnancement

Afin que l'implantation proposée précédemment soit satisfaisante, il est indispensable de réaliser une Adéquation Algorithme Architecture. Celle-ci consiste à respecter d'une part l'ordre des événements défini lors de la spécification de l'algorithme et d'autre part les contraintes temps réel. Pour cela, nous devrons choisir 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, l'heuristique actuellement utilisée est un algorithme glouton dont chaque étape alloue une opération à un processeur, route les éventuelles communications inter-processeurs c'est-à-dire créée des opérations de communication et alloue chacune d'elles à une liaison physique. L'ordonnancement des opérations de calculs ou de communications 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.

1.3.4 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 permettant 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 coûteux 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 nous avons choisi des exécutifs statiques.

Comme l'exécutif est généré 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.

A présent, nous allons pouvoir vous présenter le logiciel SynDEx qui supporte la méthodologie d'Adéquation Algorithme Architecture et qui génère l'exécutif de chaque processeur.

1.3.5 Présentation de SynDEx

SynDEx (**Syn**chronized **D**istributed **Ex**ecutive) est un environnement graphique interactif de développement pour applications temps réel de traitement du signal et des images et de contrôle de processus supportant la méthodologie d'Adéquation Algorithme Architecture. Il assure que les spécifications de l'algorithme et de l'architecture conduisent à une implantation correcte sur une machine multi-processeur et respectent les contraintes temps réel. A partir d'une implantation satisfaisante, il génère automatiquement un exécutif fiable, libérant ainsi l'utilisateur des tâches lourdes de programmation bas niveau (système) (figure **??**).

SynDEx permet tout d'abord de spécifier l'algorithme et l'architecture matérielle. Pour cela, on saisit à l'aide de la souris le graphe de l'algorithme (figure **??**) et le graphe d'architecture multi-processeur (en haut de la figure **??**). SynDEx exécute ensuite l'heuristique d'optimisation réalisant l'adéquation entre l'algorithme et l'architecture en optimisant le temps de réponse. Après l'exécution de cette heuristique, le logiciel produit un diagramme temporel (figure **??**) qui permet une visualisation faisant apparaître les communications inter-processeurs et pour chaque processeur l'ordonnancement et la durée des opérations et des communications qu'il exécute.

Le temps se déroule de haut en bas avec une colonne par processeur (`root` et `p1`) ainsi qu'une colonne par média de communication (`B` est un 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 donnée dans la fenêtre principale de SynDEx. Enfin, SynDEx génère automatiquement un exécutif distribué, séquençant sur chaque processeur les opérations de calcul qui y ont été placées ainsi que les opérations de communication avec les processeurs voisins, séquencées en parallèle avec les calculs et synchronisées avec eux pour partager en exclusion mutuelle les tampons de données à communiquer. Cet exécutif est généré sous forme de macro-code traduit par le macro-processeur standard GNU "m4" au moyen d'un jeu de macros extensible et facilement portable, aussi bien en

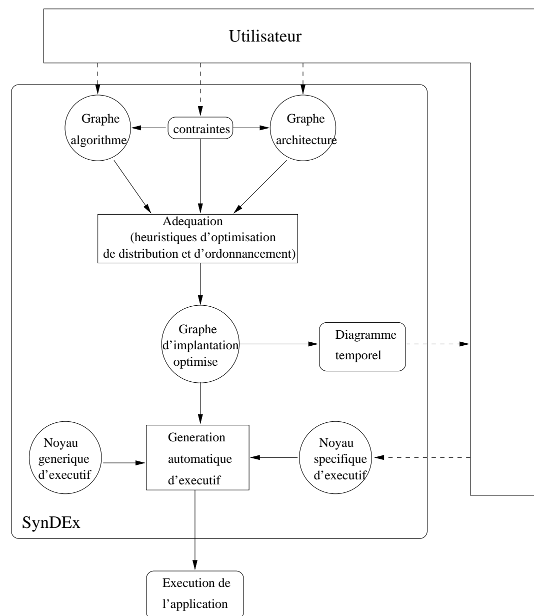


Figure 1.4: principes de SynDEx

assembleur qu'en langage de haut niveau.

Comme l'exécutif alloue les ressources statiquement, nous devons connaître à priori les durées d'exécution des différentes opérations de calcul du programme. Pour cela, nous exécutons tout d'abord le programme d'application sur une machine monoprocesseur compilé avec des instructions de chronométrage qui permettent ainsi de connaître les durées d'exécution de toutes les opérations. Ensuite, nous pouvons implanter l'algorithme sur une machine multi-processeur. On utilise comme chronomètre les horloges temps réel des processeurs.

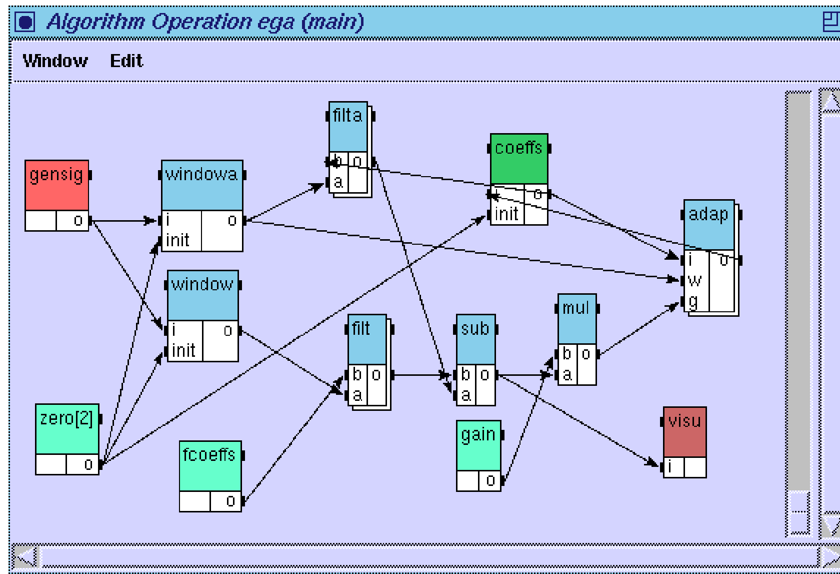


Figure 1.5: Graphe de l'algorithme de l'application ega2c

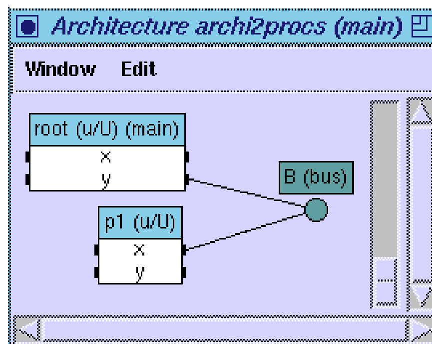


Figure 1.6: Graphe de l'architecture l'application ega2c

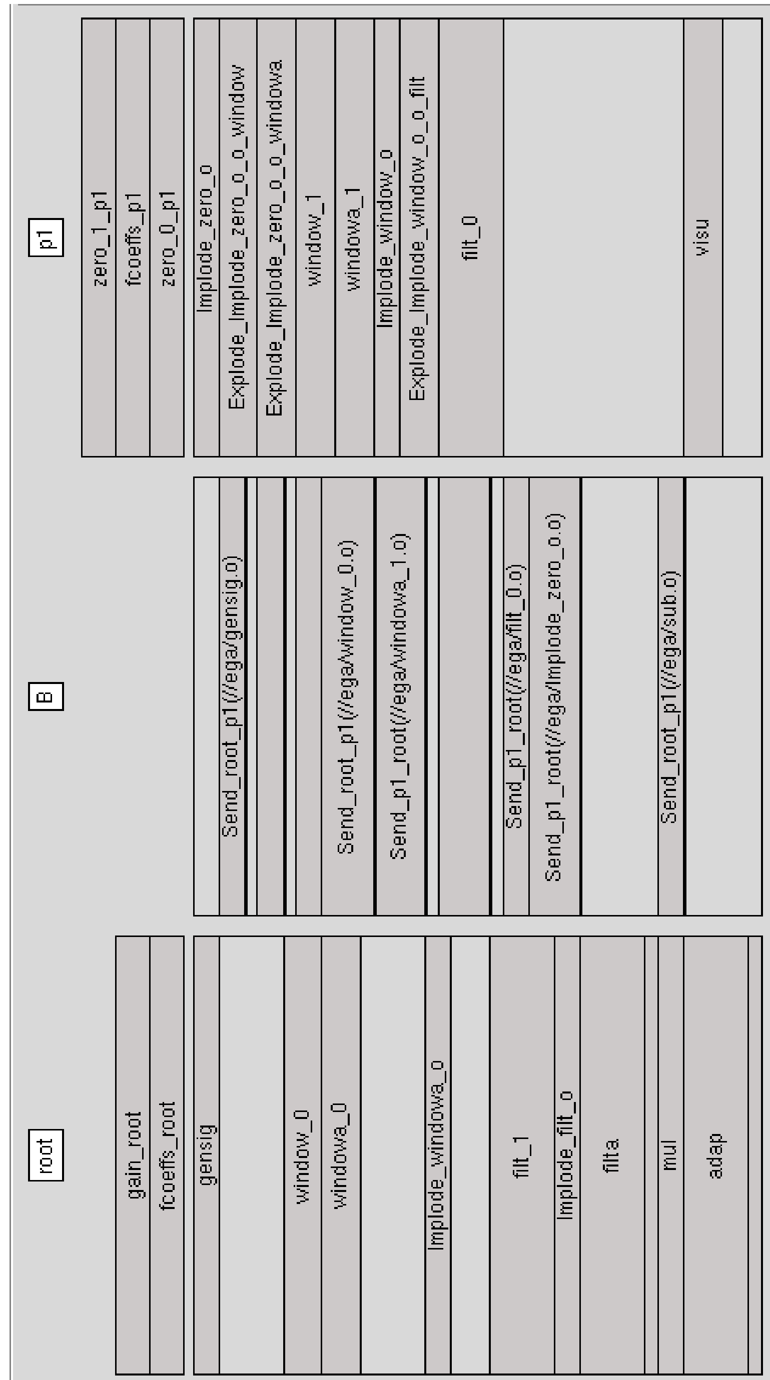


Figure 1.7: Graphe temporel pour l'application ega2c

Chapter 2

Introduction à Scilab/Scicos

Dans ce chapitre nous présentons le formalisme de modélisation des systèmes dynamiques hybrides utilisé dans *Scicos*. Ce formalisme est basé en partie sur le langage *SIGNAL* et son extension au temps continu, développé à l'*IRISA*. L'analyse du formalisme est développé à travers l'étude des signaux et du fonctionnement interne des blocs.

2.1 Les signaux

Dans *Scicos* les systèmes sont modélisés sous la forme de schéma bloc, les blocs sont une représentation graphique de fonctions à exécuter. Les blocs sont reliés entre eux via leurs entrées et sorties par les liaisons véhiculant des signaux.

Sur l'exemple de la figure **??**, on observe un système hybride formé :

- d'une partie évoluant en temps continu (le générateur de sinusoïde et l'intégrateur intégrateur1/s),
- d'une partie évoluant en temps discret (le système linéaire et l'oscilloscope), à la cadence de l'horloge.

2.1.1 Les types de signaux

A l'instar des liaisons, il existe deux types différents de signaux :

- d'activations (les liaisons sont situées au dessus et au dessous des blocs),
- réguliers (les liaisons sont situées à gauche et à droite des blocs).

2.1.2 La terminologie

Dans un souci de clarté, en fonction des aspects *continu* et *discret*, nous spécifions la caractéristique de certaines notions, récurrentes tout au long de ce document (temps, signal, bloc, activation, conditionnement). Ce besoin de précision peut s'exprimer, par exemple, à travers une question candide mais légitime : *"Comment qualifier le conditionnement d'un bloc par un train d'activations discrètes , émises de manière ininterrompues sur un intervalle de temps?"* Il apparaît ici important d'une part, de préciser clairement la distinction entre activation et événement et d'autre part, d'exprimer la formalisation des aspects *continu* (DESS) et *discret* (DTSS) et par conséquent leur traitement pour la simulation.

Dans *Scicos* les activations (ou *signaux d'activations*) conditionnent les blocs ; ce qui se traduit par la mise à jour des blocs selon la nature des activations. Les activations sont caractérisées par rapport à leur nature :

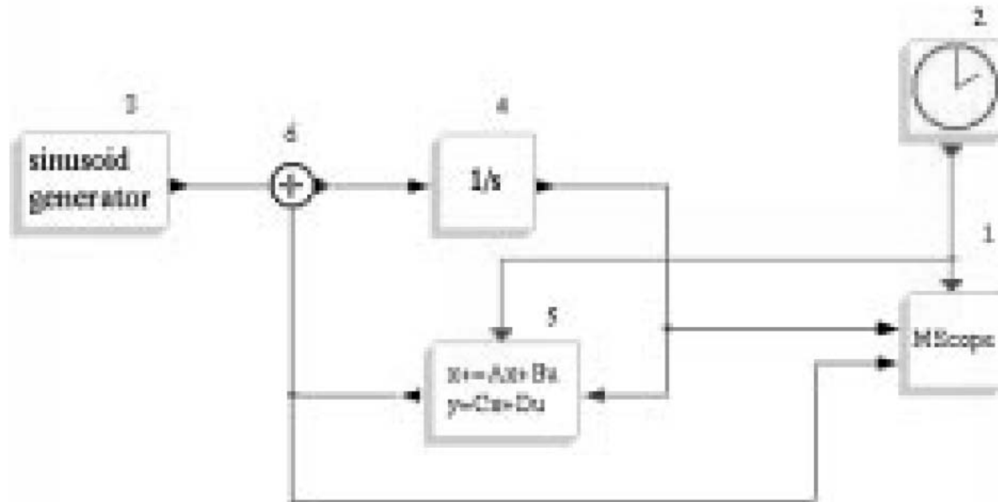


Figure 2.1: La représentation d'un système hybride dans l'éditeur Scicos.

- soit elles sont *continues* : il s'agit alors d'une *activation continue* pendant un intervalle de temps (continue par morceaux). Par extension, on parlera de *temps continu* pour l'intervalle de temps considéré. Si un bloc est conditionné par des activations continues, son fonctionnement est continu.
- soit elles sont *discrètes* : il s'agit alors d'*événements*. L'occurrence des événements est instantanée, par extension on parlera de *temps discret*. Ces événements peuvent être générés de manière périodique (par une horloge), pour un conditionnement échantillonné. Pour répondre à la question posée, si un bloc est conditionné par des événements son fonctionnement est discret. Même si ces événements sont générés avec des temps d'occurrence très proches pendant un intervalle de temps, le conditionnement et le fonctionnement du bloc n'en sont pas moins discret.

Le tableau 1.1 résume les notions importantes de la terminologie employée tout au long de ce document.

Notions	Caractéristiques
activation	signal de conditionnement (continu ou discret)
événement	activation discrète d'occurrence instantanée
temps continu	fonctionnement continu
temps discret	fonctionnement discret
bloc continu	prévu pour être conditionné en temps continu
bloc discret	prévu pour être conditionné en temps discret

Nous verrons par la suite que les blocs de type Standard peuvent être à la fois continu et discret.

2.1.3 Les signaux d'activations

Les signaux d'activations sont générés par les ports de sortie d'activation (situés en bas des blocs). Un événement a une occurrence instantanée et non rémanente.

La figure **??** montre un signal régulier en sortie d'un bloc quelconque. Ce bloc est conditionné suivant deux types d'activations : l'*activation continue par intervalle* et des *événements*.

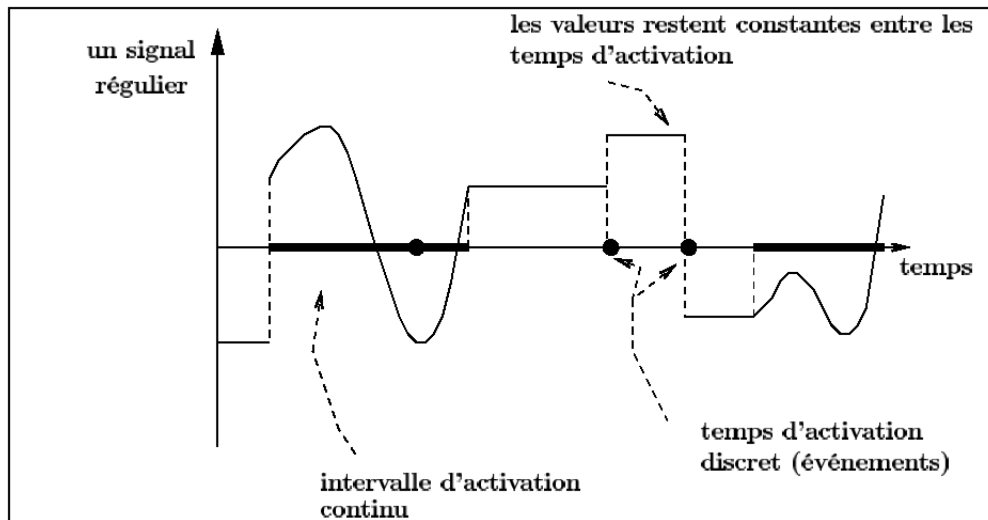


Figure 2.2: Un signal Scicos et l'ensemble des temps d'activations du bloc.

Le conditionnement par activation continue

Le fonctionnement continu des blocs est rendu possible grâce à un bloc fictif générant des activations continues. Le conditionnement continu correspond à la mise à jour des blocs de manière continue. Dans l'exemple de la figure **??**, les blocs numérotés 2 et 4 sont activés de manière continue sur l'intervalle complet de temps de simulation. Nous verrons par la suite dans la section **??**, que le bloc 1 est aussi conditionné par le bloc fictif. En revanche le bloc 3 est activé de manière continue sur un intervalle de temps dont la durée dépend à la fois de la valeur du signal sinusoïdal (bloc 4) et de la fonction du bloc 1. Ce dernier réalise un sous-échantillonnage de l'activation du bloc fictif pour conditionner le bloc 3.

Le conditionnement par événements

Les événements sont représentés sur la figure ?? par des points qui marquent le caractère instantané de leur occurrence. L'exemple de la figure ?? montre que les blocs, représentant le système linéaire ainsi que *MScope*, évoluent lors des événements générées par l'horloge (générateur d'événements périodiques). Sur la figure ?? il n'y a que le bloc *MScope* qui est activé de manière discrète.

Sur la figure **??**, lorsque la valeur de la troisième courbe est négative, la valeur de la première courbe recopie celle de la deuxième. Lorsque la valeur de la troisième courbe est positive, la valeur de la première reste constante.

2.1.4 Les signaux réguliers

Dans le schéma de la figure ??, les blocs 2, 3 et 4 génèrent un signal régulier par leur port de sortie régulière (situés à droite de chaque bloc). Les blocs 1, 3 et *Mscope* reçoivent des signaux réguliers par leurs ports d'entrée régulière (situés à gauche de chaque bloc). Un signal régulier représente le résultat de calcul de la fonction que traduit le bloc. Ce signal est continu par morceaux, les discontinuités correspondant aux événements.

L'ensemble des indices temporels associé à chaque signal régulier est appelé temps d'activations et correspond à la durée de temps pendant laquelle le signal peut évoluer. Cette notion est inspirée

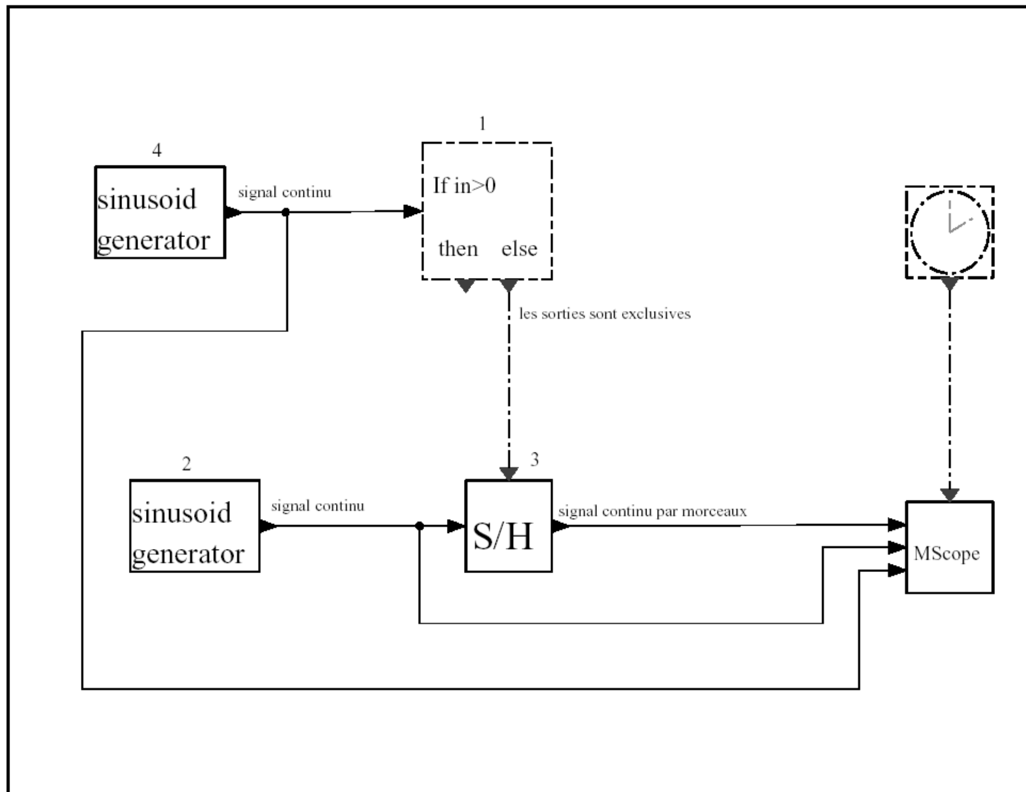


Figure 2.3: Le conditionnement des blocs.

du langage Signal. Mais en dehors de leur temps d'activations, les signaux réguliers sont présents et restent constants. En pratique, le temps d'activation est l'union des intervalles de temps et des points isolés appelés événements (cf. figure ??).

2.1.5 Le conditionnement direct et hérité

Le conditionnement hérité dans *Scicos* est une facilité graphique qui permet de réduire le nombre de connections d'activations dans un schéma bloc. Typiquement, dans l'exemple de la figure ?? le bloc If_then_Else est un bloc sans port d'entrée d'activations, conditionné par les activations de son signal d'entrée. On considère qu'il hérite du conditionnement en temps continu du bloc sinusoid generator. Le bloc If_then_Else a donc un fonctionnement en temps continu.

Dans l'exemple de la figure ?? le conditionnement du bloc 1 est hérité de celui de son entrée régulière, alors que celui du bloc 3 est explicitement défini par son entrée d'activation (cf. figure **??**). Soulignons que le bloc Clock qui conditionne le bloc MScope est en fait un bloc Delay relié sur lui-même. Le bloc Delay génère une activation dont la date d'occurrence est retardée par rapport à celle de l'activation reçue par en entrée du bloc. En programmant une activation initiale et en reliant l'entrée et la sortie d'activation, on obtient un générateur d'activation, représenté par le supe-bloc Clock. Nous verrons par la suite que l'héritage doit aussi prendre en compte les activations générée par le bloc Clock (**??**).

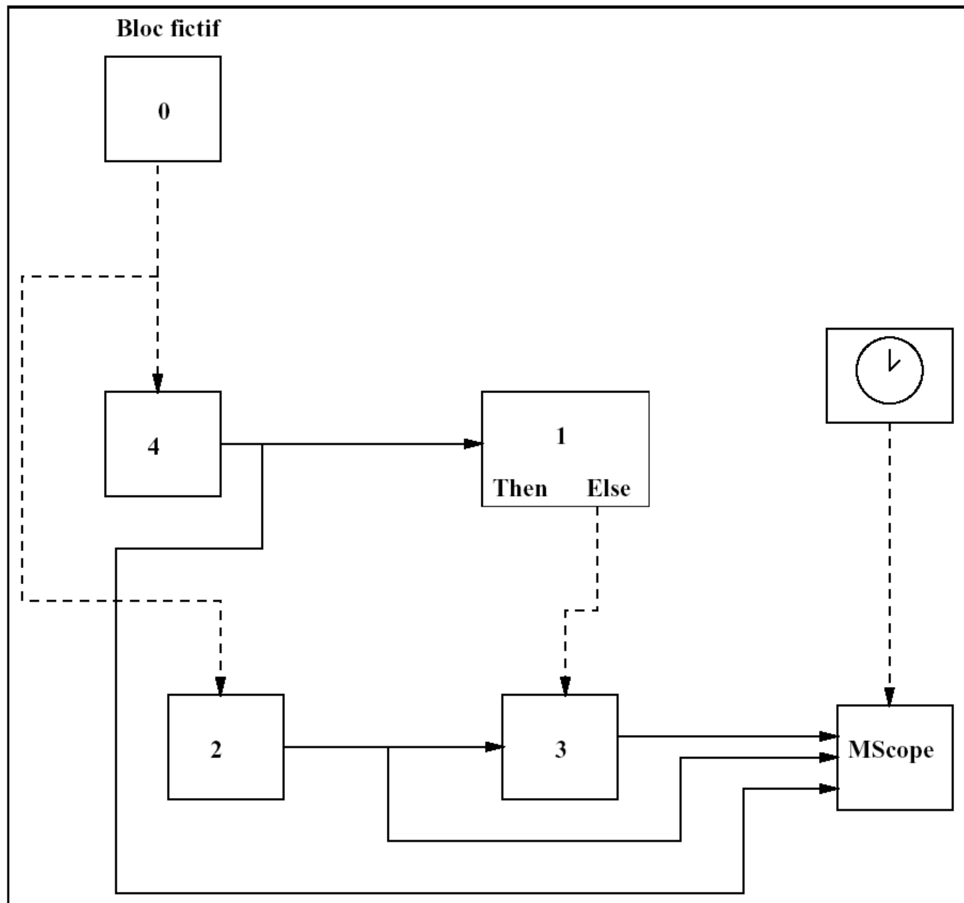


Figure 2.4: Le conditionnement continu dans la figure ??

2.1.6 Le super-bloc

Un super-bloc est un concept essentiellement graphique, permettant la représentation d'un sous-ensemble de schéma bloc, sous la forme d'un bloc. Ce principe de réduction graphique admet d'ailleurs qu'un super-bloc puisse contenir d'autres super-blocs. De cette manière on peut tout à fait envisager qu'un super-bloc contienne une imbrication de plusieurs super-blocs. Ce concept permet ainsi une certaine latitude dans la simplification des représentations graphiques.

2.1.7 La synchronisation des événements

L'occurrence d'un événement est définie par un temps chronométrique ; considérer que deux événements sont synchrones c'est admettre qu'ils ont le même temps chronométrique et la même chronologie d'occurrence. Dans Scicos, pour que deux événements soient synchrones, il faut qu'ils soient générés par des sources ayant une même et seule origine commune. Dans le cas contraire, les temps d'occurrence peuvent être identique (au sens chronométrique)mais seront considérés et traités dans un ordre chronologique indéterminé ; il s'agit dans ce cas d'événements simultanés.

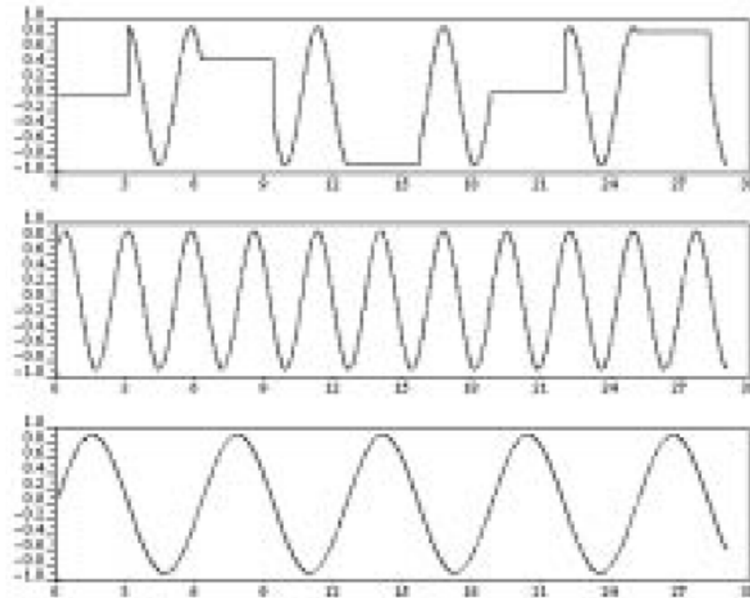


Figure 2.5: Les résultats de la simulation (cf.figure ??)

2.2 Les différents types de blocs

Les signaux sont générés par des blocs, définis suivant 4 types distincts :

- le bloc bloc Standard qui représente le type de bloc le plus courant dans Scicos,
- le bloc bloc Synchro qui est utilisé le plus souvent pour le conditionnement d'autres blocs,
- le bloc bloc Zcross qui permet des applications liées à la détection de seuil,
- le bloc bloc Memo dont l'utilisation très spécifique est destinée à des situations de boucle algébrique.

La conception de schémas blocs est obtenue par la construction et l'inter-connection de blocs basés sur ces quatre types. Un certain nombre de blocs sont proposés dans Scicos et disponibles dans des fenêtres graphiques désignées sous le terme de palettes.

Dans la représentation de schémas Scicos, il est parfois nécessaire (c'est le cas pour des schémas utilisant un nombre relativement grand de blocs), de pouvoir regrouper certains blocs sous-ensembles de blocs de manière synthétique. On utilise pour cela une facilité graphique : le super-bloc, qui permet de regrouper un ensemble de blocs sous l'apparence d'un bloc. Soulignons qu'il ne s'agit que d'une facilité graphique et non d'un nouveau type de bloc.

2.2.1 Les blocs Standards

Les blocs Standard permettent de modéliser un fonctionnement en temps discret, en temps continu ou les deux à la fois. Un bloc Standard peut être constitué de plusieurs registres. Les signaux de sortie sont stockées dans les registres de sortie. Les deux registres d'état (continu et discret) non observables de l'extérieur du bloc, indiquent que ce type de bloc peut modéliser plus que de simples systèmes dynamiques continus.

Un bloc Standard dont la dynamique utilise son registre d'état discret doit être conditionné par des événements. De même qu'un bloc Standard dont la dynamique utilise son registre d'état

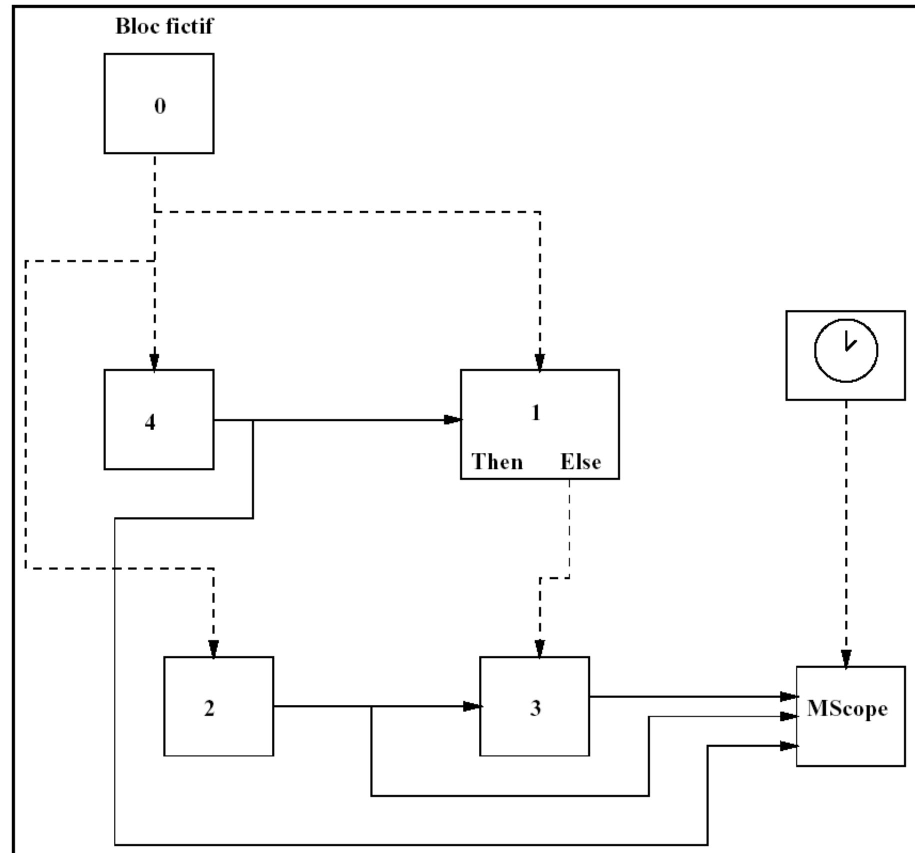


Figure 2.6: L'héritage (intermédiaire) dans la figure ??

continu doit être conditionné par activation continue. Selon le cas, ces blocs possèdent des ports d'entrée et de sortie réguliers et d'activations.

2.2.2 Les blocs Zcross

Les blocs Zcross permettent la détection de la traversée d'un seuil par leurs signaux d'entrées. Ce type de bloc ne possède ni port d'entrée d'activations, ni port de sortie régulier. Une activation de type événementiel est générée dès qu'un des signaux d'entrée du bloc change de signe.

2.2.3 Les blocs Synchro

Les blocs Synchro sont utilisés dans le conditionnement des blocs. Ils sont caractérisés par le synchronisme des activations reçues et celles générées. Cette spécificité peut permettre des applications de sous-échantillonnage. L'activation générée est dirigée, en fonction de la valeur du signal d'entrée et de la fonction du bloc ,vers un des ports de sortie d'activations. Du fait de la nécessité de synchronisme, l'activation à générer doit être traitée immédiatement. C'est la raison pour laquelle il n'est pas nécéssaire d'avoir à stocker son temps d'occurrence dans le registre. De manière générale, la fonction d'un bloc Synchro fait qu'il ne peut être générée une activation que par un seul port de sortie à la fois parmi les ports de sorties d'activations. On parle de sorties *exclusives.*

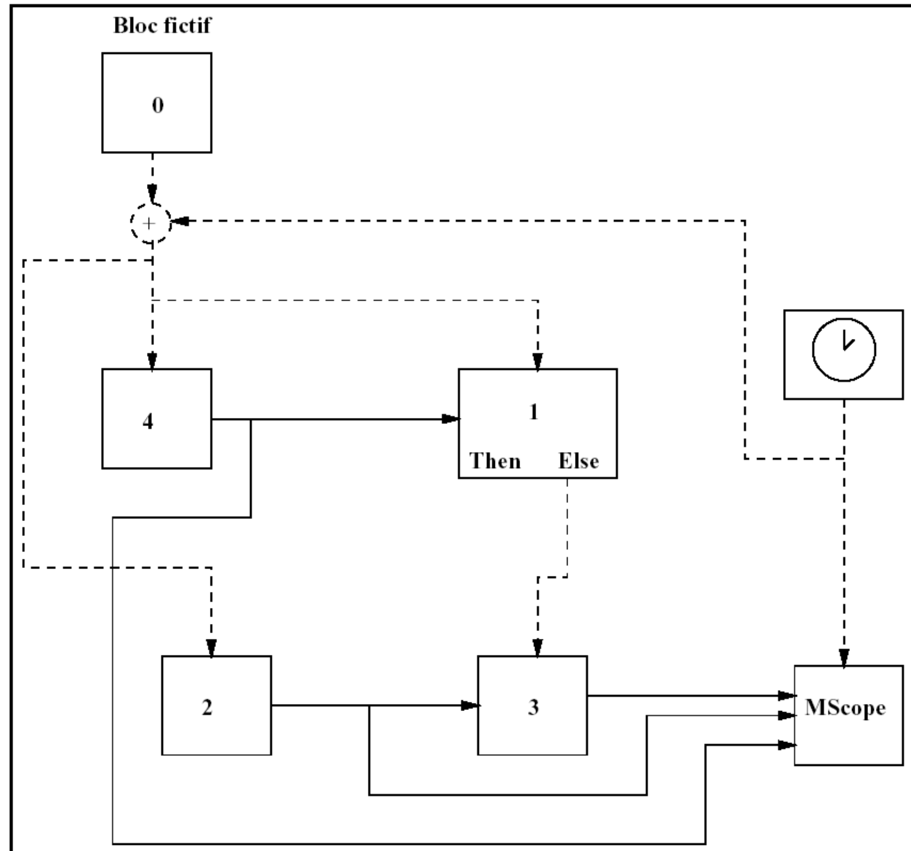


Figure 2.7: L'héritage (total) dans la figure ??

2.2.4 Les blocs Memo

Les blocs Memo ont été conçu, à l'origine, pour permettent de résoudre des problèmes liés à des erreurs de causalité (ou boucles algébriques). Ces problèmes conceptuels peuvent se présenter à l'utilisateur, lors de la transcription des algorithmes, sous forme de schémas blocs. D'autres applications utilisant ce type de bloc peuvent, cependant, être envisagées, en tenant compte, toutefois, de l'ordre de la mise à jour des blocs. En effet, la particularité des blocs Memo réside dans le fait qu'ils sont constitués uniquement d'un registre de sortie, qui est utilisé à la place d'un registre d'état.

2.2.5 Restriction au domaine nous concernant

Scicos permet les modélisations mêlant discret et continu. Néanmoins, la partie continue est réservée à la modelisation de phénomènes physiques et leur simulation. Cette partie ne peut faire l'objet d'une implémentation, au sens informatique du terme, sans être discrétisée. Notre objectif étant une traduction Scilab/Scicos→SynDEx et donc l'implémentation de modèles Scicos, nous nous restreindrons par la suite à la partie discrète des modèles Scicos.

Part II

Graphe hiérarchique d'unités fonctionnelles

Chapter 3

Graphe hiérarchique d'unités fonctionnelles

3.1 Définition de l'unité fonctionnelle

Une *unité fonctionnelle* est un graphe conditionné de dépendances de données. C'est un hypergraphe orienté. Elle se spécifie par un ensemble de sommets connectés, où chaque sommet est une fonction de calcul, et chaque arc est une dépendance de données entre *une fonction productrice* et une ou plusieurs *fonctions consommatrices*[?].

Définition 1 Soit $fu = [O,\ D]$ est une unité fonctionnelle où :

- $O = \{sw,\ f_0,\ f_1, \cdots,\ f_{n-1},\ sl\}$ est en ensemble d'éléments, appelés des *sommets*. C'est un graphe de $(n+2)$ème ordre où :
  - sw est un sommet de type "switch",
  - f_i est un sommet de type "func$_i$",
  - sl est un sommet de type "selector".
- D est un ensemble d'arcs, appelés *dépendances de données intra-unité fonctionnelle*, définis par la matrice d'incidence sommets-arcs suivante :

$$D = \begin{pmatrix} +1 & +1 & \cdots & +1 & 0 & 0 & \cdots & 0 \\ -1 & 0 & \cdots & 0 & +1 & 0 & \cdots & 0 \\ 0 & -1 & \cdots & 0 & 0 & +1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 & 0 & 0 & \cdots & +1 \\ 0 & 0 & \cdots & 0 & -1 & -1 & \cdots & -1 \end{pmatrix}$$

  Telle que chaque colonne correspond aux arcs $(s_0,\ s_1, \cdots,\ s_{n-1},\ o_0,\ o_1, \cdots,\ o_{n-1})$ et chaque ligne aux sommets $(sw,\ f_0,\ f_1, \cdots,\ f_{n-1},\ sl)$. Les éléments de la matrice peuvent prendre les valeurs $+1$, -1 et 0 où :
  - $+1$ origine de l'arc,
  - -1 extremité de l'arc,
  - 0 pas de sortie, ni d'entrée.
- le fonctionnement d'une unité fonctionnelle défini dans [?] par :
  - $s_0,\ s_1, \cdots,\ s_{n-1} = switch(s)$,

- $o_i = func_i(s_i,\ v_i)$,
- $c = selector(s,\ o_0,\ o_1, \cdots,\ o_{n-1})$.

Une unité fonctionnelle permet de représenter le comportement conditionné d'un ensemble de fonction "func$_i$". Elle se caractérise par ses entrées $(s,\ v_0,\ v_1, \cdots,\ v_{n-1})$ et sa sortie (c), et par les fonctions qui calculent la valeur de la sortie à partir de celles des entrées. Une unité fonctionnelle a une entrée spéciale (s), appelée *dépendance de conditionnement.* La dépendance de conditionnement porte une valeur, appelée *donnée de conditionnement*, qui permet au sommet "switch" de choisir parmi des fonctions alternatives, le sommet "func$_i$" qui sera exécuté.

Pour spécifier le comportement d'un système, un graphe comportant plusieurs unités fonctionnelles peut être utilisé. Le système spécifié est réactif [?], c'est à dire qu'il réagit à des stimuli provenant de l'environement extérieur en produisant des actions qui peuvent à leur tour modifier l'environnement. Les stimulis externes n'étant pas bornés en nombre dans le temps, le nombre de réactions du système ne peut pas l'être non plus. Ainsi le graphe spécifiant le système est implicitement répété infiniement. Le graphe ne représente que le comportement du système pour l'une de ces répétitions infinies. Si un sommet du graphe doit consommer lors de la $n^{ème}$ répétition du graphe une donnée produite lors de la $(n-1)^{ème}$, on utilise un sommet "delay" qui produit la donnée consommée lors de la répétition précédente.

Une unité fonctionnelle peut consommer des données produites par ses prédécesseurs. Ceux-ci peuvent être d'autres unités fonctionnelles, des sommets "sensor" ou des sommets "delay". La sortie d'une unité fonctionnelle peut être consommée par des successeurs pouvant être d'autres unités fonctionnelles, des sommets "actuator" ou des sommets "delay". La figure **??** représente une unité fonctionnelle.

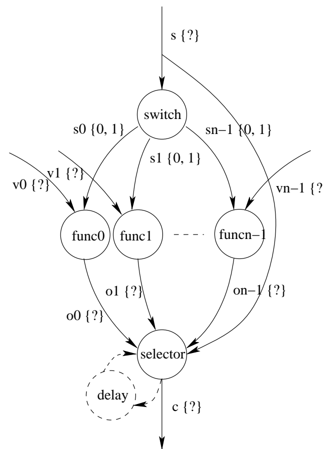


Figure 3.1: L'unité fonctionnelle.

3.1.1 Sommet "switch"

Le sommet "switch" consomme une donnée (s) et produit les données $(s_0,\ s_1, \cdots,\ s_{n-1})$ respectivement pour ses successeurs $(f_0,\ f_1, \cdots,\ f_{n-1})$. Chacune de ces données (s_i) est un booléen $(0 \rightarrow fausse$ et $1 \rightarrow vraie)$. On a $s_i = 1$ si $i = s$, sinon $s_i = 0$. La donnée s_i sera utilisée par le sommet "$func_i$" pour s'exécuter si sa valeur est vraie.

3.1.2 Sommet "func$_i$"

Le sommet "func$_i$" consomme les données (s_i, v_i) et produit une donnée (o_i). Cette fonction ne s'exécute (consommation de v_i et production de o_i) que si la valeur de s_i est vraie. Dans ce cas, on a $o_i = f_i(v_i)$ et la valeur est significative. Et si la valeur de s_i est fausse, on produit la valeur *null*.

3.1.3 Sommet "selector"

Le sommet "selector" consomme les données (s, o_0, $o_1, \cdots$, o_{n-1}) et produit une donnée (c). Le but de ce sommet est de recopier la donnée (o_i) ayant une valeur significative sur sa sortie. Pour connaître l'entrée (o_i) qu'il doit prendre en compte, il utilise la valeur de la *donnée de conditionnement* (s). Ainsi on a $c = o_s$ sauf si $o_s = null$. Dans ce cas, la sortie c doit garder la valeur qu'elle avait lors de la précédente répétition infinie et pour cela, le sommet *selector* utilise un sommet delay. Par la suite le sommet "delay" utilisé par chaque sommet "selector" sera implicite et non représenté sur les graphes. Le fait qu'une donnée garde sa précédente valeur est appelé *rémanence*.

3.2 Graphe d'unités fonctionnelles

La spécification fonctionnelle d'une application peut être obtenue par la composition de plusieurs unités fonctionnelles. Cette composition consiste à combiner les unités fonctionnelles en un graphe, appelé *graphe d'unités fonctionnelles*. On appelle *dépendance de données inter-unités fonctionnelles conditionnante* un arc reliant le sommet "selector" d'une unité fonctionnelle au(x) sommet(s) "switch" d'une ou plusieurs autre unité(s) fonctionnelle(s). L'unité fonctionnelle productrice produit la *donnée de conditionnement* sur cet arc, qui est consommée par une ou plusieurs unité(s) fonctionnelle(s) consommatrice(s). Cet arc symbolise le fait que la sortie (c) peut conditionner les autres unités fonctionnelles.

Dans un graphe d'unités fonctionnelles, on peut avoir un autre type d'arc, appelée *dépendance de données inter-unités fonctionnelles non-conditionnante*. Cet arc peut représenter une dependance de données entre :

- un sommet "selector" d'une unité fonctionnelle et un sommet "func$_i$" d'une autre unité fonctionnelle,
- un sommet "selector" d'une unité fonctionnelle et un sommet "delay" ou "actuator",
- un sommet "sensor" et un sommet "func$_i$" d'une unité fonctionnelle,
- un sommet "delay" et un sommet "func$_i$" d'une unité fonctionnelle.

Cet arc symbolise le fait qu'une unité fonctionnelle peut échanger (consommation et production) des données avec l'extérieur, que ce soit avec une autre unité fonctionnelle ou avec des sommets "externes" tel que les "delay", "sensor" et "actuator".

Définition 2 Soit $G_{fu} = [FU_a,\ D_a]$ est un graphe d'unités fonctionnelles où :

- FU_a est l'ensemble des unités fonctionnelles du graphe, $FU_a = \{fu_i\}_{1 \leq i \leq n}$, Card $FU_a = n$ où n est le nombre d'unités fonctionnelles,
- D_a est l'ensemble des arcs de *dépendance de données inter-unités fonctionnelles conditionnante* et de *dépendance de données inter-unités fonctionnelles non-conditionnante* du graphe.

La figure **??** est un exemple du graphe d'unités fonctionnelles.

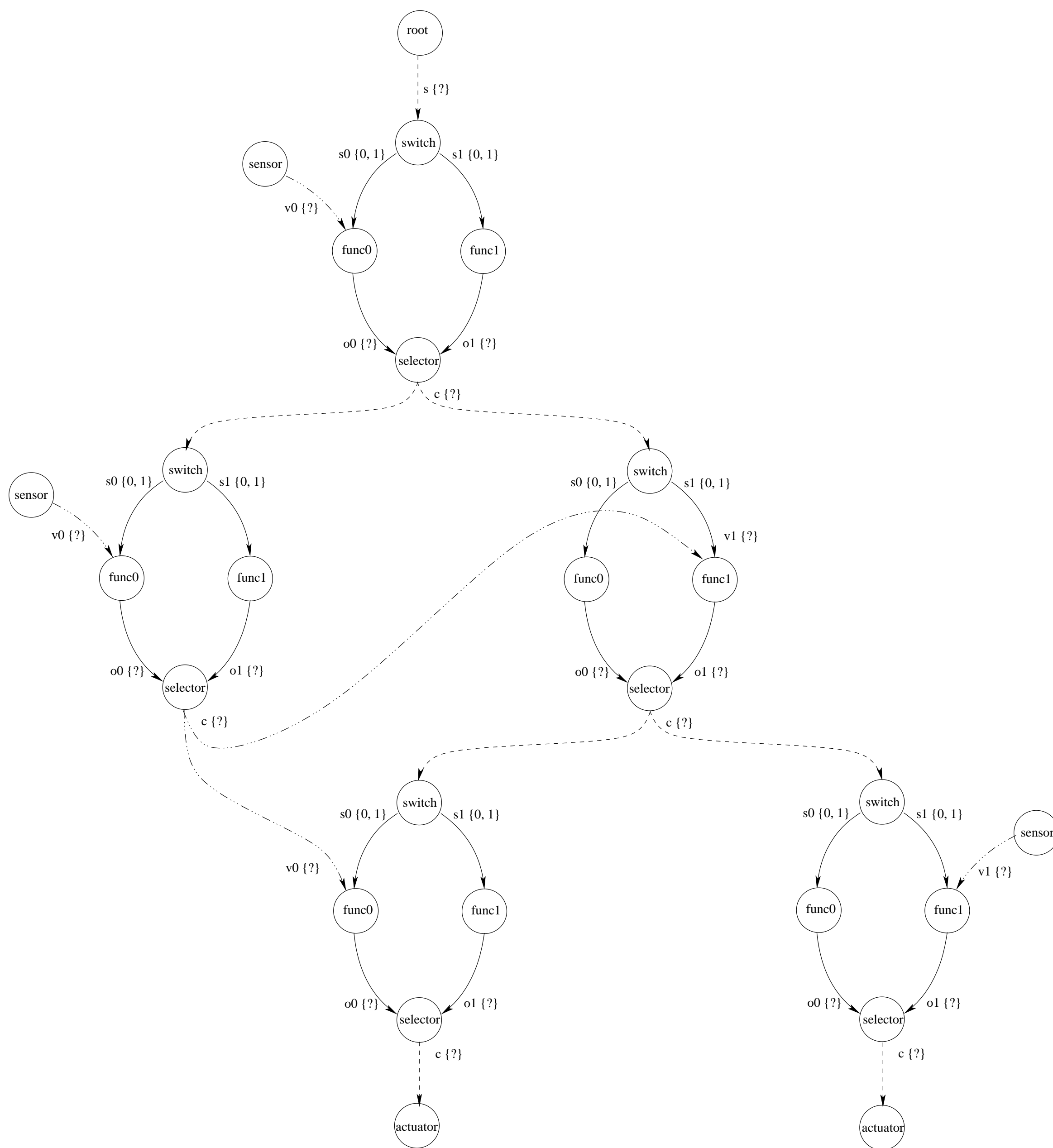


Figure 3.2: Graphe d'unités fonctionnelles.

3.3 Graphe hiérarchique d'unités fonctionnelles

Le graphe d'unités fonctionnelles est composée d'un ensemble d'unités fonctionnelles où chaque unité fonctionnelle représente un comportement conditionné. Ce comportement est caractérisé par les sommets "func$_i$". Le sommet "func$_i$" peut être définie par la fonction "super" qui permet de spécifier un sous-graphe, sous forme de graphe d'unités fonctionnelles. Ainsi ce sommet introduit la notion de hiérarchie. Le graphe d'unités fonctionnelles utilisant des sommets "super" est appelé *graphe hiérarchique d'unités fonctionnelles.*

3.4 Fonctions spéciales

3.4.1 Fonction "sensor"

La fonction "sensor" est une fonction de calcul avec une sortie et sans entrée. Il s'agit de représenter une entrée d'application, interface avec l'environnement extérieur. Cette fonction ne peut produire qu'une donnée, consommée par une ou plusieurs unité(s) fonctionnelle(s).

3.4.2 Fonction "actuator"

La fonction "actuator" est une fonction de calcul avec une entrée et sans sortie. Il s'agit de représenter une sortie d'application, interface avec l'environnement extérieur. Elle consomme une donnée produite par une unité fonctionnelle.

3.4.3 Fonction "delay"

La fonction "delay" est une fonction possédant une entrée et une sortie. Cette fonction produit à sa sortie la donnée consommé en entrée lors de la précédente répétition infinie du graphe. Lorsqu'une unité fonctionnelle a besoin de consommer, lors de la $n^{ème}$ répétition infinie du graphe, une donnée produite lors de la $(n-1)^{ème}$ répétition infinie, il faut intercaler ce type de fonction.

3.4.4 Fonction "super"

La fonction "super" introduit la notion de hiérarchie dans le graphe hiérarchique d'unités fonctionnelles. Permettant de spécifier un sous-graphe, elle est utilisée à la place de la fonction "func$_i$".

3.4.5 Fonction "dummy"

La fonction "dummy" est une non-fonction dans le sens où elle ne fait aucune action. Utilisée à la place de la fonction "func$_i$" dans l'unité fonctionnelle, elle consomme une donnée (s_i) sur son arc d'entrée et produit la valeur *null* sur sa sortie (o_i).

3.4.6 Fonction "root"

La fonction "root" est une fonction qui est utilisé pour définir l'horloge du graphe hiérarchique d'unités fonctionelles. Elle ne possède pas d'entrée et produit la valeur 1 sur sa sortie lors de chaque répétition infinie du graphe.

Chapter 4

Optimisation du graphe hiérarchique d'unités fonctionnelles

Dans le chapitre précédent, nous avons présenté le graphe hiérarchique d'unités fonctionnelles. Ce graphe nous permet de spécifier le fonctionnement d'une application. Cependant, ce graphe n'est pas optimal car certains sommets sont employés de manière rédondante. Afin de diminuer le nombre de sommet et ainsi la complexité du graphe, il convient de détecter et d'éliminer cette redondance.

4.1 Détection et élimination de la redondance des "switch"

Dans un graphe hiérarchique d'unités fonctionnelles, nous avons vu qu'une *dépendance de données inter-unités fonctionnelles conditionnante* permet à une unité fonctionnelle productrice fu_p de produire une *donnée de conditionnement* consommée par des unités fonctionnelles consommatrices fu_c. Dans ces unités fonctionnelles consommatrices, il peut y avoir rédondance de sommets "swtich" que nous proposons de détecter et éliminer de la manière suivante.

Définition 3 Soit fu_1, et fu_2 deux unités fonctionnelles tel que :

- $fu_1 = [O_1,\ D_1]$ et $O_1 = \{sw_1,\ f_{1_0},\ f_{1_1}, \cdots,\ f_{1_{n-1}},\ sl_1\}$,
- $fu_2 = [O_2,\ D_2]$ et $O_2 = \{sw_2,\ f_{2_0},\ f_{2_1}, \cdots,\ f_{2_{n-1}},\ sl_2\}$.

Définition 4 Les unités fonctionnelles fu_1 et fu_2 sont exclusives si et seulement si, $\forall i,\ 0 \leq i \leq n-1$, au moins l'un des deux sommets "func$_{1_i}$" et "func$_{2_i}$" est une fonction "dummy".

Règle 1 Les sommets "switch" sw_1 et sw_2 appartenant respectivement aux unités fonctionnelles fu_1 et fu_2 sont redondants si et seulement si les unités fonctionnelles fu_1 et fu_2 sont exclusives et consomment la même *donnée de conditionnement* produite par une unité fonctionnelle fu_p.

La règle 1 permet de détecter la redondance des sommets "switch" dans un graphe hiérarchique d'unités fonctionnelles afin de les éliminer et d'optimiser ainsi le graphe. Pour cela, on remplace les sommets "switch" sw_1 et sw_2 par un seul sommet "switch" sw_c dont l'entrée s_c est la même que sw_1 et sw_2, et les sorties $s_{c0},\ s_{c1}, \cdots,$ et s_{cn-1} sont définies par l'union des sorties des sommets "switch" sw_1 et sw_2, soit $s_{ci} = s_{1i} \cup s_{2i},\ \forall i,\ 0 \leq i \leq n-1$. En outre, il faut également insérer sur chaque *dépendance de données inter-unités fonctionnelles non-conditionnante* entre fu_1 et fu_2, une fonction "delay" afin de préserver le comportement du graphe.

Cette technique permet de réduire le nombre des sommets "switch" utilisé dans le graphe et donc de diminuer le temps de calcul nécessaire à l'évaluation de celui-ci. En contre-partie, les

sommets "delay" ajoutés augmentent l'espace mémoire nécessaire.

La figure **??** donne un exemple de détection et d'élimination de sommets "switch". La figure (a) est un graphe hiérarchique d'unités fonctionnelles composé de trois unités fonctionnelles. L'unité fonctionnelle qui se trouve en haut du graphe produit une donnée de conditionnement qui est consommée, via une *dépendance de données inter-unités fonctionnelles conditionnante*, par les deux autres unités fonctionnelles. De plus, en observant les sommets "func$_i$" de ces deux unités fonctionnelles, on en déduit qu'elles sont exclusives. Les sommets "switch" de ces deux unités fonctionnelles sont donc rédondants. On peut éliminer cette rédondance et obtenir le graphe qui apparait sur la figure (b). Il ne comporte plus que deux sommets "switch" au lieu de trois et un sommet "delay" a été insérer sur la *dépendance de données inter-unités fonctionnelles non-conditionnante* qui reliait les deux unités fonctionnelles exclusives de la figure (a).

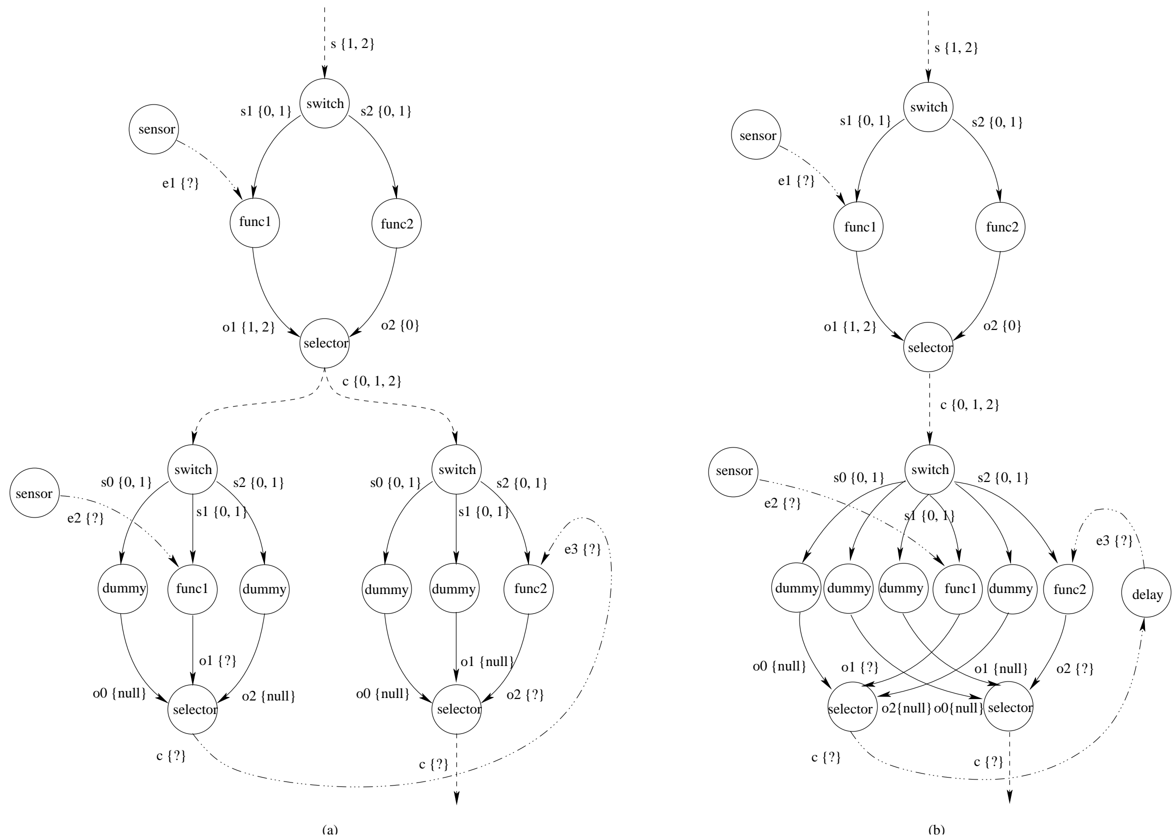


Figure 4.1: (a) Le graphe hiérarchique d'unités fonctionnelles ; (b) le même graphe après élimination de la redondance du sommet "switch".

4.2 Détection et élimination de la redondance des "selector"

La détection de la redondance de sommets "selector" utilise le même principe que celui de détection de redondance des sommets "switch".

Règle 2 Les sommets "selector" sl_1 d'unité fonctionnelle fu_1 et sl_2 d'unité fonctionnelle fu_2 sont redondant si et seulement si ces unités fonctionnelles sont exclusives et consomment la *donnée de conditionnement* produite par la même unité fonctionnelle fu_p.

On déduit des règles 1 et 2 que si les sommets "switch" de deux unités fonctionnelles sont redondants, alors leurs sommets "selector" le sont aussi. Bien que la règle 2 permette de détecter la redondance des sommets "selector", leur élimination pose un problème de comportement. En effet, il n'est possible de supprimer uniquement que les sommets "selector" redondants dont les sorties sont consommées par des fonctions "actuator". Dans ce cas, on remplace les sommets "selector" sl_1 et sl_2 par un seul sommet "selector" sl_c dont les entrées o_{c0}, $o_{c1}, \cdots$, et o_{cn-1} sont définies par l'union des entrées des sommets "selector" sl_1 et sl_2 soit $o_{ci} = o_{1i} \cup o_{2i}$, $\forall i,\ 0 \leq i \leq n-1$, et sa sortie c_c est consommée par l'ensemble des consommateurs des sommets "selector" sl_1 ou sl_2. Lorsque l'on élimine les sommets "selector" de deux unités fonctionnelles, celles-ci ne comportent déjà plus qu'un seul sommet "switch", la rédondance de "switch" ayant été éliminée. Passer de deux sommets "selector" à un seul entraine de passer de de $2n$ sommets "func" à n. Pour cela, on supprime, pour i allant de 0 à $n-1$, le sommet "func$_{1i}$" si c'est un sommet "dummy", sinon le sommet "func$_{2i}$".

La figure **??** présente un exemple d'élimination de la redondance de sommets "selector". La figure (a) est un graphe hiérarchique d'unités fonctionnelles après élimination de la redondance de sommets "switch". Il est est composé de trois unités fonctionnelles, une unité fonctionnelle productrice et deux unités fonctionnelles consommatrices, ne comportant plus qu'un seul sommet switch pour deux. La figure (b) montre le graphe obtenu après élimination de la redondance de sommets "selector".

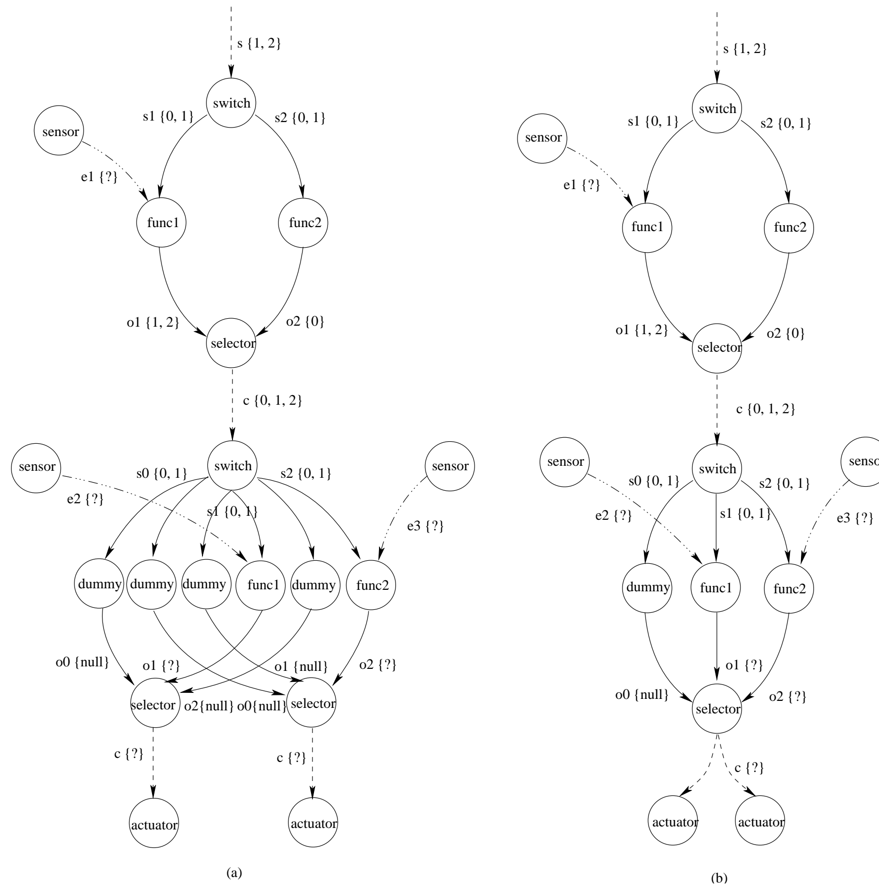


Figure 4.2: (a) Le graphe hiérarchique d'unités fonctionnelles ; (b) le même graphe après élimination de la redondance des sommets "selector".

4.3 Détection et élimination des tests inutiles

Dans un graphe hiérarchique d'unités fonctionnelles, il est possible qu'une unité fonctionnelle fu_c soit la seule consommatrice de la *donnée de conditionnement* s_c qui est produite par une unité fonctionnelle productrice fu_p. La donnée s_c est consommée par le sommet "switch" sw_c de l'unité fonctionnelle consommatrice et est testée pour choisir le sommet "func$_{cs_c}$" qui doit être activé. Or, il arrive que certains de ces tests soient inutiles et nous pouvons les détecter de la manière suivante :

Règle 3 Le test de la *donnée de conditionnement* s_c de valeur r par le sommet "switch" w_c est inutile si et seulement si les sommets "func$_{cr}$" sont tous les fonctions "dummy".

Définition 5 Soit r une des valeurs possibles de la donnée s_c dont le test est inutile au sens de la règle 3 :

- la fonction "func$_{p_r}$" est la fonction "func$_{p_i}$" (f_{p_i}) de l'unité fonctionnelle productrice qui produit la valeur r,
- la fonction "func$_{p_n}$" est la fonction "func$_{p_i}$" (f_{p_i}) de l'unité fonctionnelle productrice qui produit toute les valeurs possibles de la *donnée de conditionnement* c_p sauf r.

Pour éliminer la redondance de test de la donnée s_c de valeur r, nous procédons de la manière suivante :

- supprimer le test de la donnée s_c de valeur r du sommet "switch" sw_c,
- supprimer les sommets "func$_{cr}$",
- supprimer l'entrée o_{cr} du sommets "selector" sl_c
- remplacer le sommet "func$_{p_r}$" par une fonction "dummy",
- remplacer la fonction "func$_{p_n}$ par une fonction "super" contenant la fonction "func$_{p_n}$" et l'unité fonctionnelle consommatrice ainsi que toutes les unités fonctionnelles qu'elle conditionne par des *dépendances de données inter-unités fonctionnelles conditionnantes.* La donnée produite par le sommet "func$_{p_n}$" est consommée par l'unité fonctionnelle consommatrice.
- dupliquer le sommet "selector" de l'unité fonctionnelle productrice si le sommet super possède plus d'une sortie.

Les entrées du sommet "super" sont l'union des entrées de la fonction "func$_{p_n}$" d'unité fonctionnelle productrice (s_{p_i}, et v_{p_i}) et des entrées des sommets "func" de l'unité fonctionnelle consommatrice (v_{c0}, $v_{c1}, \cdots,$ et v_{cn-1}). Ses sorties sont celles de l'unité fonctionnelle consommatrice et des unités fonctionnelles qu'elle conditionne. Chaque sortie de la fonction "super" o_{p_i} est consommée par un sommet "selector" différent, ce qui peut impliquer la dupplication de celui existant dans l'unité fonctionnelle productrice.

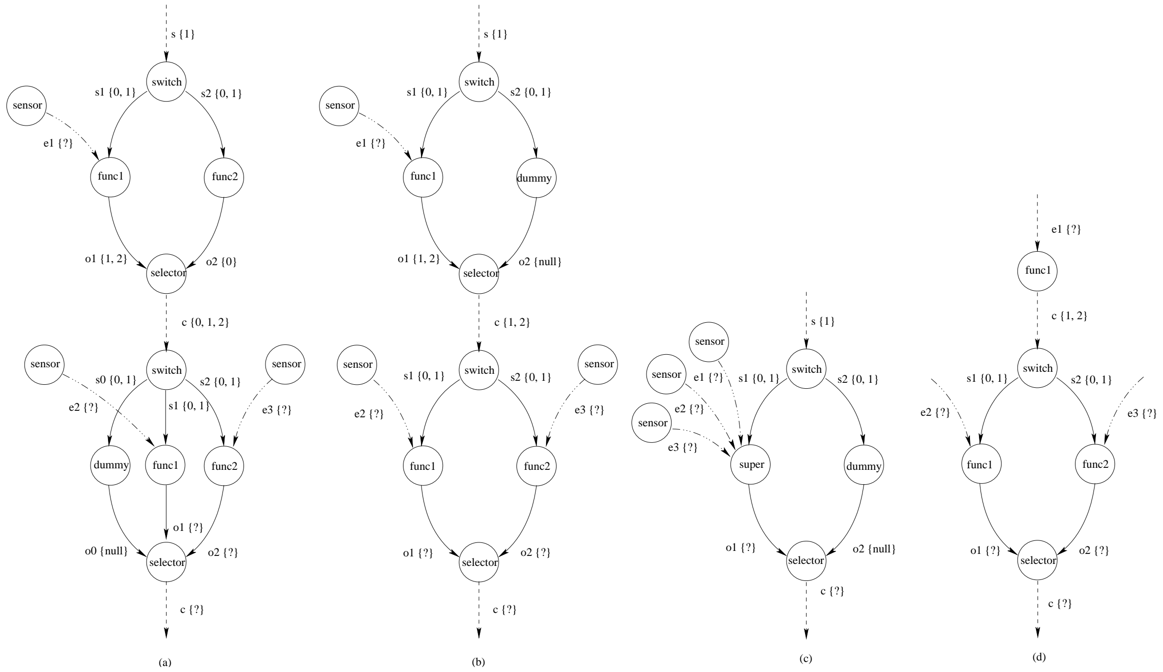


Figure 4.3: (a) Un graphe hiérarchique d'unités fonctionnelles ; (b) le même graphe après suppression du test de la donnée c de valeur 0 et du sommet "dummy" de l'unité fonctionnelle consommatrice et remplacement du sommet "$func_2$" de l'unité fonctionnelle productrice par une fonction "dummy" ; (c) le graphe après inclusion de l'unité fonctionnelle consommatrice dans le sommet "super" de l'unité fonctionnelle productrice ; (d) le sous-graphe du sommet "super".

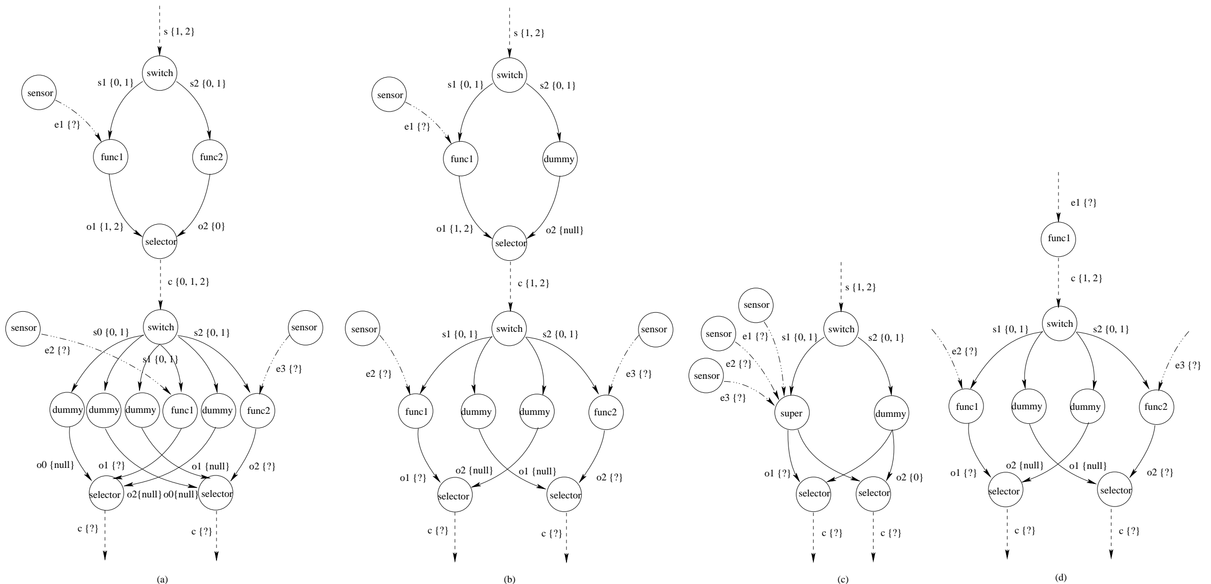


Figure 4.4: (a) Un graphe hiérarchique d'unités fonctionnelles ; (b) le même graphe après suppression du test de la donnée c de valeur 0 et du sommet "dummy" de l'unité fonctionnelle consommatrice et remplacement du sommet "$func_2$" de l'unité fonctionnelle productrice par une fonction "dummy" ; (c) le graphe après inclusion de l'unité fonctionnelle consommatrice dans le sommet "super" de l'unité fonctionnelle productrice et duplication le sommet "selector" pour chacune des sorties du sommet "super" ; (d) le sous-graphe du sommet "super".

Part III

Application

Chapter 5

Transformation du graphe Scicos en un graphe d'algorithme SynDEx

Dans le chapitre précédent, nous avons introduit la notion de graphe hiérarchique d'unités fonctionnelles. Dans cette partie, nous montrons comment de tels graphes peuvent être utilisés pour transformer un graphe Scicos en un graphe SynDEx. Cette transformation se déroule en trois étapes :

- transformer le graphe Scicos en un graphe hiérarchique d'unités fonctionnelles,
- optimiser le graphe hiérarchique d'unités fonctionnelles,
- traduire le graphe hiérarchique d'unités fonctionnelles en un graphe d'algorithme de SynDEx.

Seules la première et la troisième étape seront décrites ici, l'optimisation du graphe hiérarchique d'unités fonctionnelles ayant été présentée dans le précédent chapitre. A la fin du chapitre, nous présentons l'implémentation de cette transformation.

5.1 Transformation du graphe de Scicos en un graphe hiérarchique d'unités fonctionnelles

Nous avons vu qu'un graphe Scicos est composé d'un ensemble de blocs et de connections. Dans ce graphe un bloc peut recevoir des signaux sur ses entrées régulières et génèrer des signaux sur ses sorties régulières (partie traitement de donnée). Il peut également émettre des événements sur ses sorties d'activations et être activé par des événements sur ses entrées d'activations (partie contrôle). En effet son exécution dépends des événements qu'il reçoit sur ses entrées d'activation. Ce comportement peut être représenté par une ou plusieurs unités fonctionnelles. Il est donc possible de transformer un graphe Scicos en un graphe hiérarchique d'unités fonctionnelles où les *données de conditionnement* des unités fonctionnelles correspondent aux évenements permettant d'activer les blocs Scicos.

Définition 6 On définit les valeurs de la *donnée de conditionnement* correspondant aux événements de la manière suivante :

- 0 représente non événement,
- 1 représente un événement émis par un bloc "Synchro" sur son premier port d'activation,
- 2 représente un événement émis par un bloc "Synchro" sur son deuxième port d'activation.

Les connections régulières entre les blocs sont transformées en *dépendances de données inter-unités fonctionnelles non-conditionnantes* et les connections d'activations entre les blocs deviennent

des *dépendances de données inter-unités fonctionnelles conditionnantes* entre les unités fonctionnelles.

Dans un graphe Scicos, les blocs discrets se répartissent en trois catégories : les blocs "Synchro", les blocs "Standard", les blocs "Memo". Chaque type de bloc est transformé en une ou plusieurs unités fonctionnelles.

5.1.1 Transformation des blocs "Synchro"

Les blocs "Synchro" sont utilisés pour conditionner l'exécution d'autres blocs en émettant des événement. Le bloc "if_then_else" est un exemple de bloc "Synchro" et se traduit de la façon suivante.

Bloc "if_then_else"

Le bloc "if_then_else" possède un port d'entrée régulier *in*, un port d'entrée d'activation e et deux ports de sortie d'activation out_1 (then) et out_2 (else). Lorsque le port e est activée par un événement, la valeur lue sur le port *in* est comparée à la valeur 0. Si la valeur lue sur le port *in* est supérieure à zéro, un événement est émis sur le port out_1 ; sinon un événement est généré sur le port out_2. Si le bloc n'est pas activé, aucun événement n'est généré en sortie.

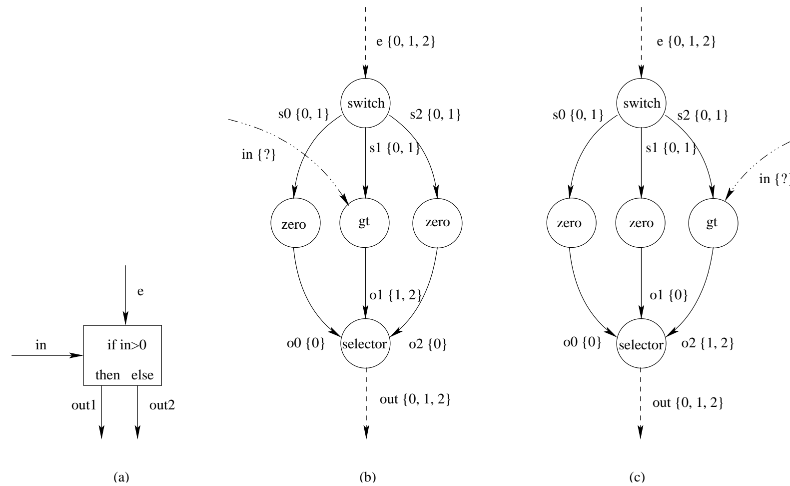


Figure 5.1: (a) Le bloc "if_then_else" ; (b)(c) l'unité fonctionnelle "if_then_else".

Suivant les cas, l'entrée e peut être connectée soit à la première sortie d'activation d'un bloc "Synchro", soit à la deuxième, ce qui modifie la façon dont il est traduit en une unité fonctionnelle. La figure **??** (b) est le résultat de la transformation dans le premier cas. C'est une unité fonctionnelle qui est composé de cinq sommets : les sommets "switch", "$func_0$", "$func_1$", "$func_2$", et "selector" où les sommets "$func_0$" et "$func_2$" sont les fonctions "zero" et le sommet "$func_1$" est la fonction "gt". Le fonctionnement de la fonction "gt" est de comparer la valeur de *in* en entrée avec la valeur 0. Elle produit la valeur 1 en sortie si la valeur de *in* est supérieur à 0 sinon elle produit la valeur 2. Le fonctionnement de la fonction "zero" est de produire la valeur constante 0 en sortie. Cette unité fonctionnelle possède deux entrées e, *in* et une sortie *out*. L'entrée e est la *donnée de conditionnement* qui peut porter les trois valeurs possible 0, 1, et 2, qui corresponds à la définition 6. Selon la valeur de e, cette unité fonctionnelle produit la valeur 1 (then) ou 2 (else) si la valeur de e égale à 1 sinon elle produit la valeur 0 (bloc non activé). Ainsi, l'unité fonctionnelle possède le même comportement que le bloc "if_then_else".

La figure ?? (c) est le résultat de la transformation dans le deuxième cas. C'est une unité fonctionnelle qui est composé par la même nombre et le même type de sommets que celle du premier cas. La différence est que c'est la fonction "$func_2$" qui est la fonction "gt" parce que cette fonction doit être exécuté uniquement quand la *donnée de conditionnement* est égale à 2. En effet, d'un cas à l'autre, seul la valeur d'activation diffère.

5.1.2 Transformation des blocs "Standard"

Le bloc "Standard" dans un graphe de Scicos réaliser un traitement sur des données régulières lorsqu'il est activé. Si le bloc n'est pas activé, les valeurs de ses sorties restent inchangées, par rémanence. Comme la transformation des blocs "Synchro", la transformation de bloc "Standard" diffère selon la manière dont il est activé.

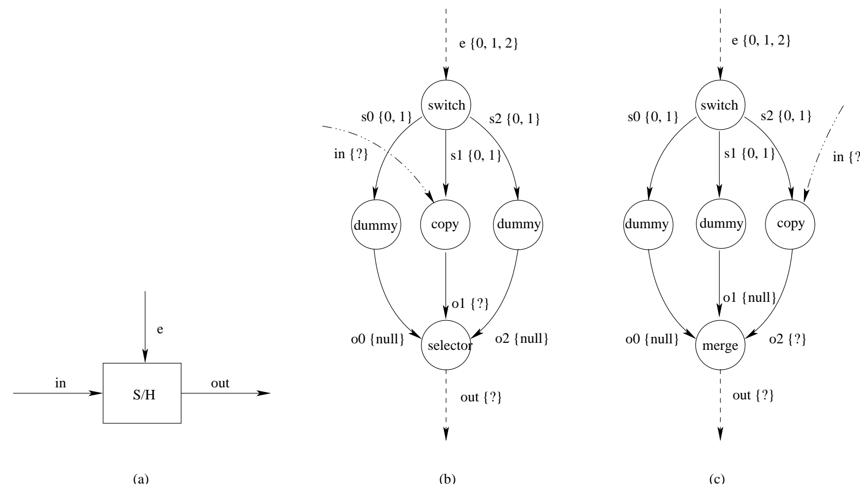


Figure 5.2: (a) Le bloc "sampling_and_hold" ; (b)(c) l'unité fonctionnelle "sampling_and_hold".

La figure **??** montre la transformation du bloc "sampling_and_hold". Le fonctionnement de ce bloc consiste à copier la valeur lue sur son entrée régulière *in* sur sa sortie régulière *out* s'il est activé par un événement e. Selon la manière dont il est activé, on obtient l'unité fonctionnelle de la figure (b) ou (c). Cette unité fonctionnelle est composé de cinq sommets : les sommets "switch", "$func_0$", "$func_1$", "$func_2$", et "selector". Pour le premier cas, les sommets "$func_0$" et "$func_2$" ("$func_0$" et "$func_1$" pour le deuxième cas) sont les fonctions "dummy" et le sommet "$func_1$" ("$func_2$" pour le deuxième cas) est une fonction "copy" qui copie la valeur de *in* sur sa sortie $o1$ ("$o2$" pour le deuxième cas). Ainsi, le fonctionnement de l'unité fonctionnelle est de copier la donnée *in* sur la sortie *out* si la valeur de la *donnée de conditionnement* e est égale à 1 (2 dans le deuxième cas), sinon de reproduire la même donnée. Cette unité fonctionnelle se comporte donc comme le bloc "sampling_and_hold".

La figure **??** montre un example du graphe Scicos. La transformation de ce graphe en un graphe hiérarchique d'unités fonctionnelles est représentée par la figure **??** et la figure **??** montre son optimisation.

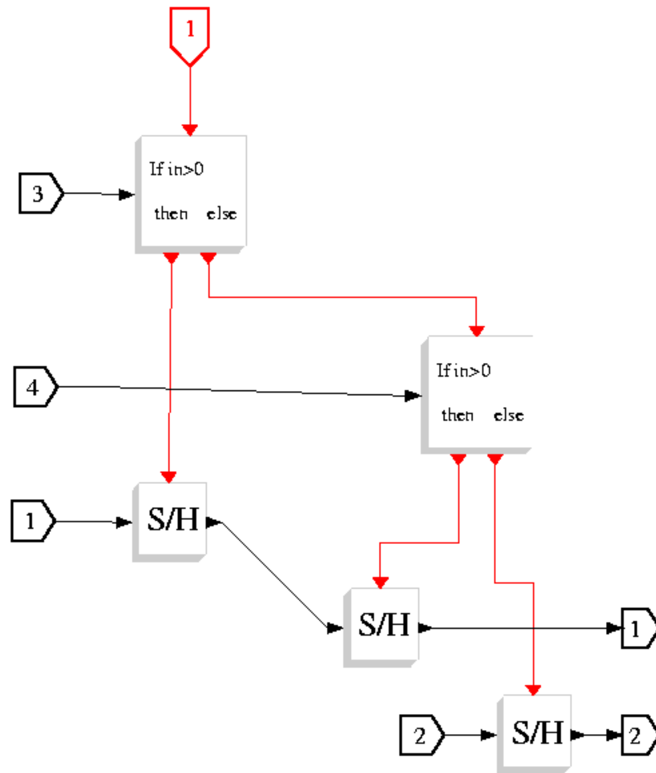


Figure 5.3: Un graphe Scicos.

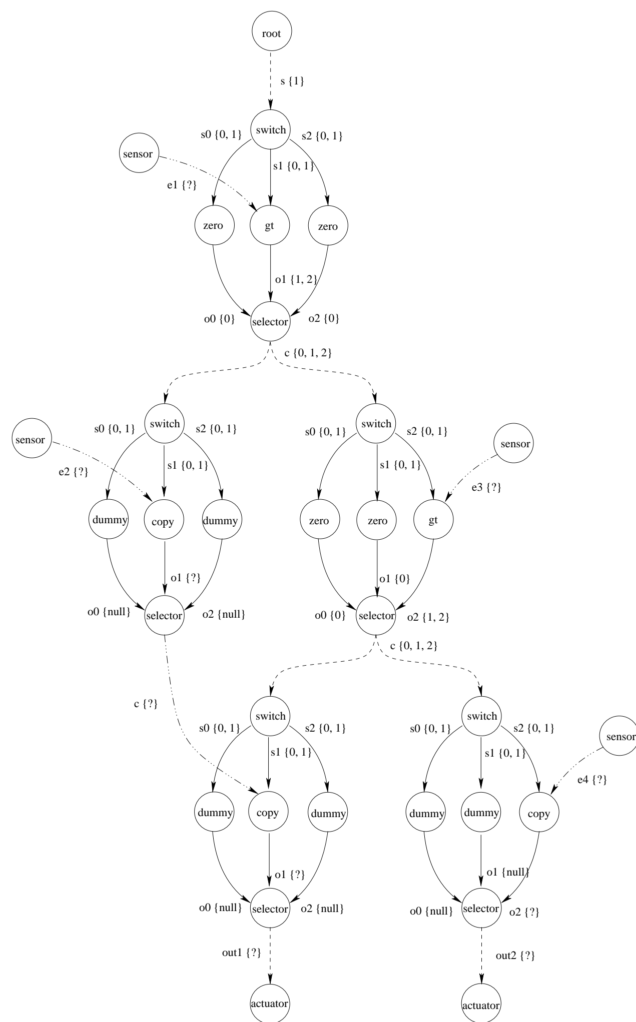


Figure 5.4: La transformation du graphe Scicos (la figure ??) en un graphe hiérarchique d'unités fonctionnelles.

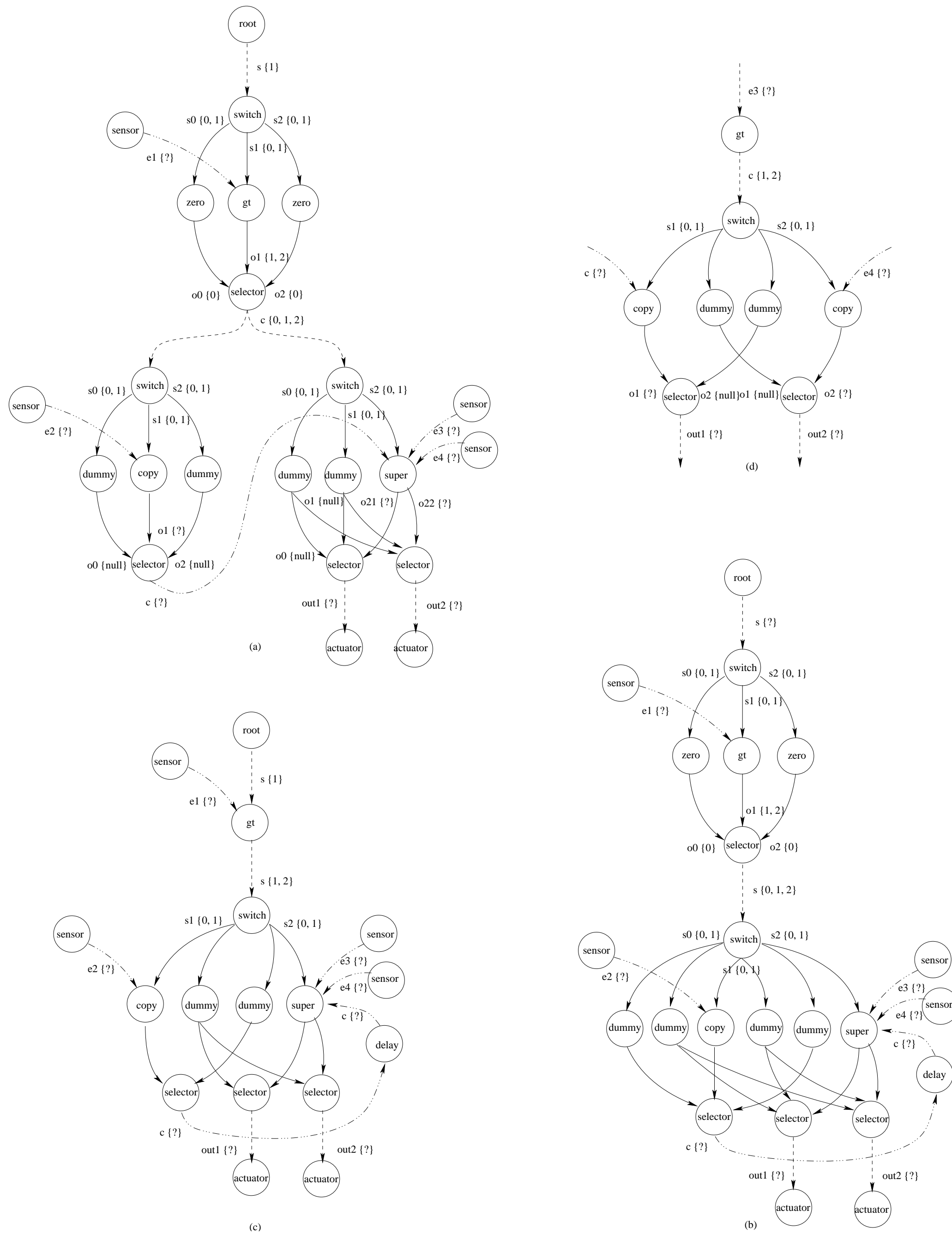


Figure 5.5: L'optimisation du graphe hiérarchique d'unités fonctionnelles (la figure ??) ; (a)(b)(c) représentant des étapes différentes de l'optimisation et (d) représentant le sous-graphe du sommet "super".

5.2 Traduction du graphe hiérarchique d'unités fonctionnelles en un graphe d'algorithme SynDEx

Dans cette section nous présentons le procédé de la traduction du graphe hiérarchique d'unités fonctionnelles en un graphe d'algorithme SynDEx. Nous présenterons d'abord le principe du conditionnement dans SynDEx, ainsi que la façon dont il peut être représenté sous forme de graphe hiérarchique d'unités fonctionnelles. Ensuite, nous décrirons comment chaque élément du graphe hiérarchique d'unités fonctionnelles issu du graphe Scicos est traduit en un élément du graphe d'algorithme SynDEx.

5.2.1 Le conditionnement dans SynDEx

Principes

Dans un graphe d'algorithme SynDEx, un sommet peut être conditionné. Pour cela, il possède une dépendance de donnée particulière appelée *dépendance de conditionnement.* Pour chacune des valeurs que peut prendre la dépendance de conditionnement, un sous-graphe de ce sommet est spécifié. Chacun de ces sous-graphes peut, parmis les entrées et sorties du sommet conditionné, utiliser des entrées (aucune, une partie ou toutes) et des sorties (aucune, une partie ou toutes).

La figure **??** montre un exemple de conditionnent. On y distingue deux sous-graphe d'un même sommet qui possède une entrée pour la dépendance de condtionnement ainsi que deux autres entrées et deux sorties.

L'exécution d'un sommet conditionné se déroule en 3 phases :

- on acquiert les entrées du sommet et on teste l'entrée de conditionnement,
- selon la valeur de cette dernière, on exécute le sous-graphe d'algorithme correspondant,
- on produit les sorties pour chaque port de sortie, quel que soit le sous-graphe ayant été exécuté.

Il est donc possible qu'une sortie soit produite alors que le sous-graphe exécuté ne l'a pas utilisée. Dans ce cas, c'est la dernière valeur de cette sortie qui est reproduite, faisant apparaître un phénomène de rémanence.

Représentation en graphe d'unités fonctionnelles

Un sommet conditionné dans SynDEx peut être représenté par un graphe d'unités fonctionnelles qui comporte autant d'unités fonctionnelles que le sommet conditionné comporte de sorties. L'entrée s de ces unités fonctionnelles est la dépendance de conditionnement et les sous-graphes deviennent les sommets func_i de ces unités fonctionnelles. La figure **??** montre le graphe d'unités fonctionnelles représentant le sommet conditionné de la figure ?? ainsi que le graphe optimisé correspondant. Si un sommet conditionné peut être représenté sous forme de graphe d'unités fonctionnelles, il est possible, à l'inverse, de déduire d'un graphe d'unités fonctionnelles, un graphe d'algorithme SynDEx comportant des sommets conditionnés.

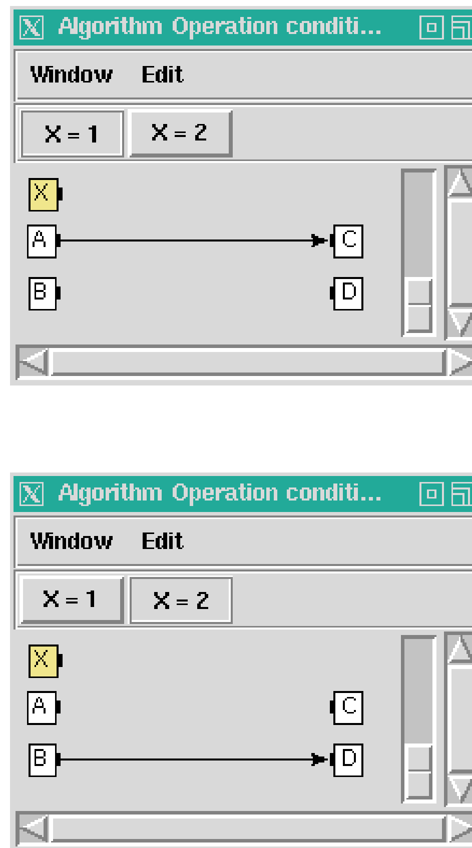


Figure 5.6: Un sommet conditionné dans SynDEx.

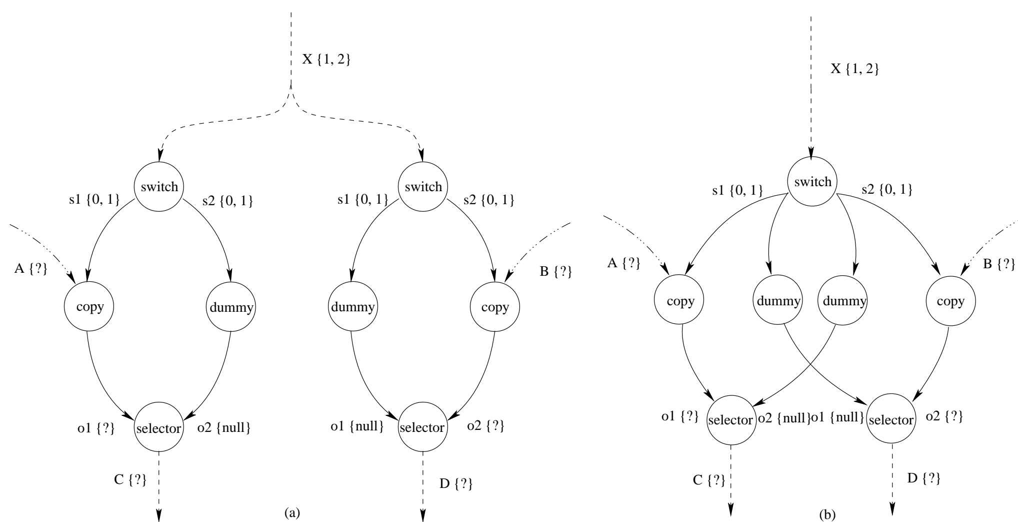


Figure 5.7: (a) Un graphe d'unités fonctionnelles représente le sommet conditionné de la figure **??** ; (b) le graphe d'unités fonctionnelles optimisé.

5.2.2 Traduction

Traduction des unité fonctionnelles

Chaque unité fonctionnelle devient un sommet conditionné du graphe d'algorithme SynDEx où :

- l'entrée s de l'unité fonctionnelle devient le port d'entrée du conditionnement ;
- les autres entrées de l'unité fonctionnelle deviennent des ports d'entrée ;
- les sorties de l'unité fonctionnelle deviennent les ports de sortie ;
- pour chaque valeur que s peut prendre dans le graphe d'unités fonctionnelles, si la fonction func_i n'est pas une fonction "dummy", elle devient le sous-graphe corresponant au cas où la dépendance de conditionnement est égal à cette valeur. Si la fonction func_i est une fonction "dummy", ce cas n'apparait pas dans le sommet conditionné.
- les connections s_i de l'unités fonctionnelles n'apparaissent pas, les o_i deviennent des arcs entrant sur les ports de sorties et les connection v_i deviennent des arcs sortant des ports d'entrées.

  Remarque la fonction "switch" et "selector" d'une unité fonctionnelle sont traduites implicitement par le conditionnement dans SynDEx.

Traduction des dépendances de données

Les dépendances de données inter-unités fonctionnelles conditionnantes et non-conditionnantes sont traduites par des arcs de dépendance de donnée dans un graphe d'algorithme SynDEx.

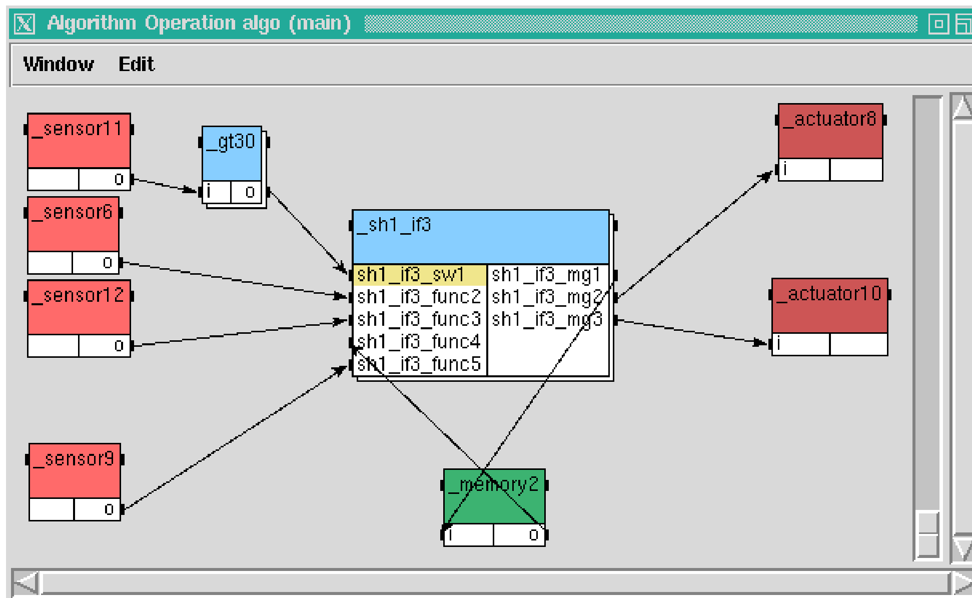
Tranduction des fonctions spéciales

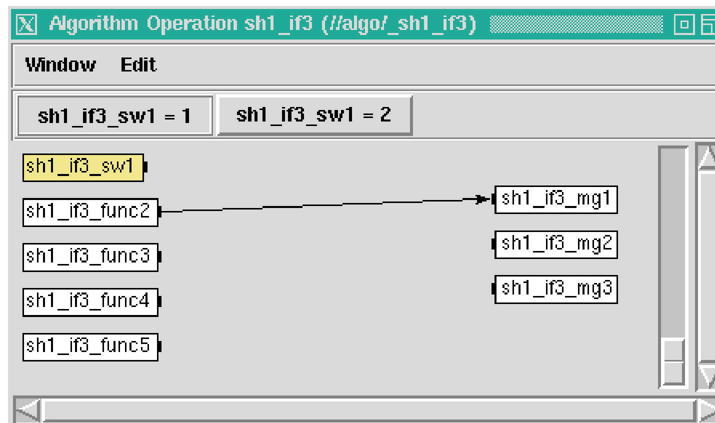
On traduit chaque fonction spéciale de la manière suivante :

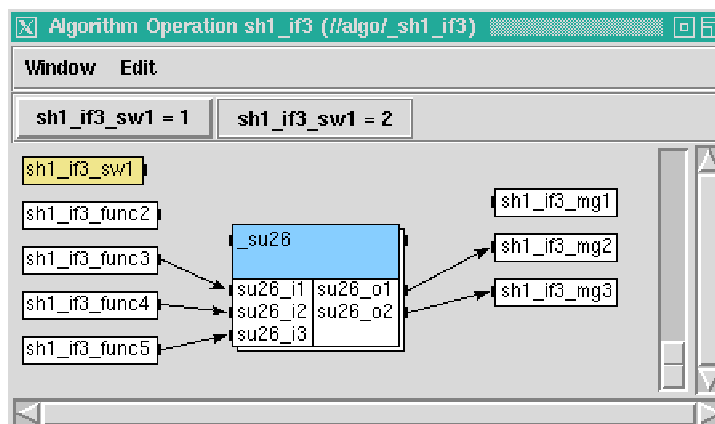
- la fonction "sensor" est traduite par un opérateur "sensor",
- la fonction "actuator" est traduite par un opérateur "actuator",
- la fonction "delay" est traduite par un opérateur "delay",
- la fonction "super" est traduite par un sommet comportant un sous-graphe (hiérarchie),
- la fonction "dummy" est ignorée,
- la fonction "root" est ignorée.

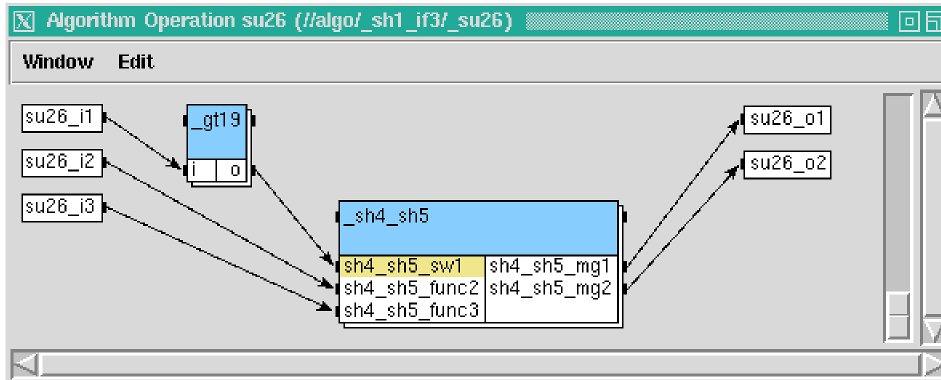
Exemple

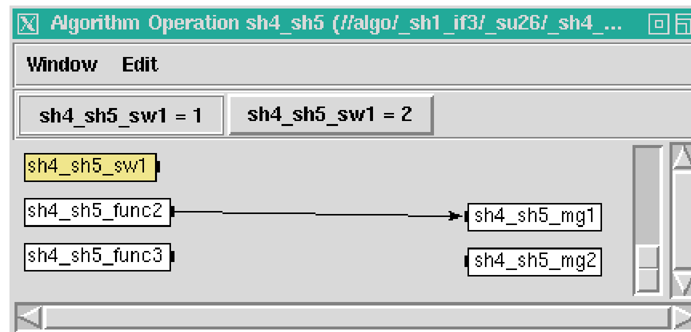
La figure **??** montre la traduction du graphe hiérarchique d'unités fonctionnelles dans la figure ??.


Algorithm Operation algo (main)
Window
Edit
_sensor11
_gt30
_actuator8
_sensor6
_sh1_if3
sh1_if3_sw1
sh1_if3_func2
sh1_if3_func3
sh1_if3_func4
sh1_if3_func5
sh1_if3_mg1
sh1_if3_mg2
sh1_if3_mg3
_sensor12
_actuator10
_sensor9
_memory2


Algorithm Operation sh1_if3 (//algo/_sh1_if3)
Window
Edit
sh1_if3_sw1 = 1
sh1_if3_sw1 = 2
sh1_if3_sw1
sh1_if3_func2
sh1_if3_func3
sh1_if3_func4
sh1_if3_func5
sh1_if3_mg1
sh1_if3_mg2
sh1_if3_mg3


Algorithm Operation sh1_if3 (//algo/_sh1_if3)
Window
Edit
sh1_if3_sw1 = 1
sh1_if3_sw1 = 2
sh1_if3_sw1
sh1_if3_func2
sh1_if3_func3
sh1_if3_func4
sh1_if3_func5
_su26
su26_i1
su26_i2
su26_i3
su26_o1
su26_o2
sh1_if3_mg1
sh1_if3_mg2
sh1_if3_mg3





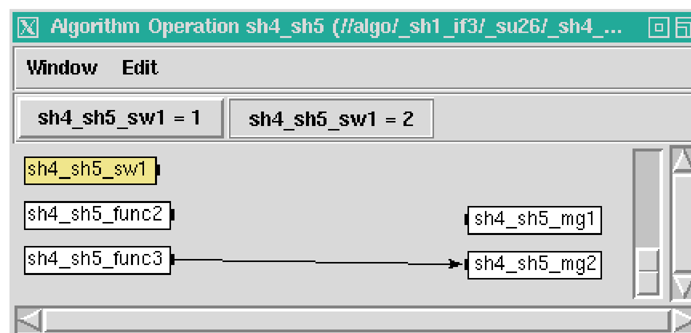


Figure 5.8: La traduction du graphe hiérarchique d'unités fonctionnelles (la figure ??) en un graphe d'algorithme SynDEx.

5.3 Implémentation de la transformation

Dans la section précédente, nous avons présenté les principes de la transformation du graphe Scicos en un graphe d'algorithme SynDEx. Dans cette section, nous présentons comment ces principes ont été implémentés. Nous avons choisis d'implémanter cette transformation directement dans Scicos de manière à permettre, à partir de Scicos, de générer des fichiers SynDEx. Pour cela, nous avans développé cette transformation en utilisant le langage *Scilab*.

Après la compilation d'un graphe Scicos, les variables suivantes sont accessibles :

- *clkconnect* est une matrice, elle contient les informations sur les connections d'activation entre les blocs,
- *connectmat* est une matrice, elle contient les informations sur les connections régulières entre les blocs,
- *bllst* est une list d'objets, elle contient les informations sur chaque bloc du graphe.

Pour les matrices *clkconnect* et *connectmat*, chaque ligne de la matrice définit une connection entre deux blocs où les colonnes 1 et 3 sont respectivement les numéros d'identification du premier et du deuxième bloc. Et ils se connectent via les ports définis par les colonnes 2 et 4 de la matrice.

Les matrices *clkconnect* et *connectmat* suivantes sont obtenus par la compilation de la figure ??. La deuxième ligne de la matrice *clkconnect* nous indique par exemple qu'il y a une connection d'activation entre le $2^{ème}$ port de sortie du bloc numéro 2 et le 1er port d'entrée du bloc numéro 3.

```
clkconnect  =
!   2.    1.    1.     1. !
!   2.    2.    3.     1. !
!   3.    1.    4.     1. !
!   3.    2.    5.     1. !
!   7.    1.    2.     1. !
!   3.    1.    8.     1. !
!   3.    2.    10.    1. !

connectmat  =
!   11.    1.    2.     1. !
!   12.    1.    3.     1. !
!   1.     1.    4.     1. !
!   6.     1.    1.     1. !
!   4.     1.    8.     1. !
!   9.     1.    5.     1. !
!   5.     1.    10.    1. !
```

L'implémentation de la transformation se déroule en 4 étapes :

Etape 1 A partir de la matrice *clkconnect* et la liste d'objet *bllst*, on crée une structure de données de type arbre où chaque noeud de l'arbre est une donnée-objet de type *tlist* (un objet du langage Scilab). Cette donnée-objet contient les informations d'une unité fonctionnelle obtenu par la transformation d'un bloc de la liste d'objets *bllst*. De plus, il contient les *dépendances de données inter-unités fonctionnelles conditionnantes* entre les unités fonctionnelles obtenus par la transformation des connections d'activation entre les blocs.

La donnée-objet *tlist* d'une unité fonctionnelle est définie de la manière suivante :

```
objTree=tlist(["obj","objNumber","objName", "objShortName", "objType",..
             "childClk",...
             "inputPorts", "outputPorts",..
             "funcSwitch0", "funcSwitch1", "funcSwitch2"],..
              objNumber, objName, objShortName, objType,..
```

```
childClk,...
inputPorts, outputPorts,..
funcSwitch0, funcSwitch1, funcSwitch2);
```

Où on a :

- *objNumber* est un numéro d'identification, différent pour chaque unité fonctionnelle,
- *objName* est un nom,
- *objShortName* est un diminutif de *objName*,
- *objType* est un type d'unité fonctionnelle,
- *childClk* est une structure de données qui contient les fils,
- *inputPorts* est une structure de données qui contient les ports d'entrées,
- *outputPorts* est une structure de données qui contient les ports de sorties.
- *funcSwitch0* est une structure de données qui contient les fonctions "$func_0$",
- *funcSwitch1* est une structure de données qui contient les fonctions "$func_1$",
- *funcSwitch2* est une structure de données qui contient les fonctions "$func_2$".

Etape 2 En utilisant la matrice *connectmat*, on modifie les connections entre les données-objets dans la structure de données (créé par l'étape 1), autrement dit on connecte la *dépendance de données inter-unités fonctionnelles non-conditionnante* entre des unités fonctionnelles.

Remarque L'étape 1 et 2 permet d'obtenir une structure de données de type arbre qui représente un graphe hiérarchique d'unités fonctionnelles.

La structure de données suivante correspond au graphe hiérarchique d'unités fonctionnelles non-optimisé de la figure ??.

```
=> N: -1 Name: root SName: rt Type: d
   => N: 7 Name: activin SName: rt7 Type: d
      => N: 2 Name: ifthel SName: if2 Type: s
         |->Input Ports
            Name: if2_sw1 Type: 0 Obj: if2 N: 2 Port: 1 N: 7 Port: 1
            Name: if2_func2 Type: 2 Obj: if2 N: 2 Port: 2 N: 11 Port: 1
         |->Output Ports
            Name: if2_mg1 Type: 0 Obj: if2 N: 2 Port: 1 N: 1 Port: 1
            Name: if2_mg1 Type: 0 Obj: if2 N: 2 Port: 1 N: 3 Port: 1
         |->Switch0 Function
            => N: 23 Name: zero SName: zo23 Type: f
               |->Output Ports
                  Name: o Type: 0 Obj: zo N: 23 Port: 1 N: 2 Port: 1
         |->Switch1 Function
            => N: 22 Name: gt SName: gt22 Type: f
               |->Input Ports
                  Name: i Type: 0 Obj: gt N: 22 Port: 1 N: 2 Port: 2
               |->Output Ports
                  Name: o Type: 0 Obj: gt N: 22 Port: 1 N: 2 Port: 1
         |->Switch2 Function
            => N: 24 Name: zero SName: zo24 Type: f
               |->Output Ports
                  Name: o Type: 0 Obj: zo N: 24 Port: 1 N: 2 Port: 1
         => N: 1 Name: samphold SName: sh1 Type: d
            |->Input Ports
               Name: sh1_sw1 Type: 0 Obj: sh1 N: 1 Port: 1 N: 2 Port: 1
```

```
      Name: sh1_func2 Type: 2 Obj: sh1 N: 1 Port: 2 N: 6 Port: 1
   |->Output Ports
      Name: sh1_mg1 Type: 1 Obj: sh1 N: 1 Port: 1 N: 4 Port: 2
   |->Switch1 Function
      => N: 13 Name: copy SName: cp13 Type: f
         |->Input Ports
            Name: i Type: 0 Obj: cp N: 13 Port: 1 N: 1 Port: 2
         |->Output Ports
            Name: o Type: 0 Obj: cp N: 13 Port: 1 N: 1 Port: 1
   |->Switch2 Function
      => N: 14 Name: dummy SName: dm14 Type: f
         |->Output Ports
            Name: o Type: 0 Obj: dm N: 14 Port: 1 N: 1 Port: 1
=> N: 3 Name: ifthel SName: if3 Type: s
   |->Input Ports
      Name: if3_sw1 Type: 0 Obj: if3 N: 3 Port: 1 N: 2 Port: 1
      Name: if3_func2 Type: 2 Obj: if3 N: 3 Port: 2 N: 12 Port: 1
   |->Output Ports
      Name: if3_mg1 Type: 0 Obj: if3 N: 3 Port: 1 N: 4 Port: 1
      Name: if3_mg1 Type: 0 Obj: if3 N: 3 Port: 1 N: 5 Port: 1
   |->Switch0 Function
      => N: 20 Name: zero SName: zo20 Type: f
         |->Output Ports
            Name: o Type: 0 Obj: zo N: 20 Port: 1 N: 3 Port: 1
   |->Switch1 Function
      => N: 21 Name: zero SName: zo21 Type: f
         |->Output Ports
            Name: o Type: 0 Obj: zo N: 21 Port: 1 N: 3 Port: 1
   |->Switch2 Function
      => N: 19 Name: gt SName: gt19 Type: f
         |->Input Ports
            Name: i Type: 0 Obj: gt N: 19 Port: 1 N: 3 Port: 2
         |->Output Ports
            Name: o Type: 0 Obj: gt N: 19 Port: 1 N: 3 Port: 1
   => N: 4 Name: samphold SName: sh4 Type: d
      |->Input Ports
         Name: sh4_sw1 Type: 0 Obj: sh4 N: 4 Port: 1 N: 3 Port: 1
         Name: sh4_func2 Type: 1 Obj: sh4 N: 4 Port: 2 N: 1 Port: 1
      |->Output Ports
         Name: sh4_mg1 Type: 3 Obj: sh4 N: 4 Port: 1 N: 8 Port: 1
      |->Switch1 Function
         => N: 15 Name: copy SName: cp15 Type: f
            |->Input Ports
               Name: i Type: 0 Obj: cp N: 15 Port: 1 N: 4 Port: 2
            |->Output Ports
               Name: o Type: 0 Obj: cp N: 15 Port: 1 N: 4 Port: 1
      |->Switch2 Function
         => N: 16 Name: dummy SName: dm16 Type: f
            |->Output Ports
               Name: o Type: 0 Obj: dm N: 16 Port: 1 N: 4 Port: 1
   => N: 5 Name: samphold SName: sh5 Type: d
      |->Input Ports
         Name: sh5_sw1 Type: 0 Obj: sh5 N: 5 Port: 1 N: 3 Port: 1
         Name: sh5_func2 Type: 2 Obj: sh5 N: 5 Port: 2 N: 9 Port: 1
      |->Output Ports
         Name: sh5_mg1 Type: 3 Obj: sh5 N: 5 Port: 1 N: 10 Port: 1
      |->Switch1 Function
         => N: 18 Name: dummy SName: dm18 Type: f
            |->Output Ports
               Name: o Type: 0 Obj: dm N: 18 Port: 1 N: 5 Port: 1
```

```
|->Switch2 Function
   => N: 17 Name: copy SName: cp17 Type: f
      |->Input Ports
         Name: i Type: 0 Obj: cp N: 17 Port: 1 N: 5 Port: 2
      |->Output Ports
         Name: o Type: 0 Obj: cp N: 17 Port: 1 N: 5 Port: 1
```

Etape 3 On optimise le graphe hiérarchique d'unités fonctionnelles. Cette optimisation se fait en partant des feuilles et en remontant l'arbre. Son déroulement est le suivant :

- appliquer la règle 1 : Détecter et éliminer la redondance des fonctions "switch" (cf. la section **??**) sur l'ensemble d'un niveau de l'arbre,
- appliquer la règle 3 : Détecter et éliminer les testes inutiles.(cf. la section **??**) sur l'ensemble d'un niveau de l'arbre.
- réitérer l'étape 3 sur le niveau supérieur de l'arbre.

Remarque L'étape 3 permet d'obtenir un graphe hiérarchique d'unités fonctionnelles optimisé.

La structure de données suivante correspond à celle (la figure ??) après optimisation.

```
=> N: -1 Name: root SName: rt Type: d
   => N: 7 Name: activin SName: rt7 Type: d
      => N: 30 Name: gt SName: gt30 Type: f
         |->Input Ports
            Name: i Type: 2 Obj: if2 N: 30 Port: 1 N: 11 Port: 1
         |->Output Ports
            Name: o Type: 0 Obj: if2 N: 30 Port: 1 N: 29 Port: 1
         => N: 29 Name: samphold_ifthel SName: sh1_if3 Type: u
            |->Input Ports
               Name: sh1_if3_sw1 Type: 0 Obj: sh1 N: 29 Port: 1 N: 30 Port: 1
               Name: sh1_if3_func2 Type: 2 Obj: sh1 N: 29 Port: 2 N: 6 Port: 1
               Name: sh1_if3_func3 Type: 2 Obj: if3 N: 29 Port: 3 N: 12 Port: 1
               Name: sh1_if3_func4 Type: 4 Obj: sh4 N: 29 Port: 4 N: 2 Port: 1
               Name: sh1_if3_func5 Type: 2 Obj: sh5 N: 29 Port: 5 N: 9 Port: 1
            |->Output Ports
               Name: sh1_if3_mg1 Type: 4 Obj: sh1 N: 29 Port: 1 N: 2 Port: 1
               Name: sh1_if3_mg2 Type: 3 Obj: sh4 N: 29 Port: 2 N: 8 Port: 1
               Name: sh1_if3_mg3 Type: 3 Obj: sh5 N: 29 Port: 3 N: 10 Port: 1
            |->Switch1 Function
               => N: 13 Name: copy SName: cp13 Type: f
                  |->Input Ports
                     Name: i Type: 0 Obj: cp N: 13 Port: 1 N: 29 Port: 2
                  |->Output Ports
                     Name: o Type: 0 Obj: cp N: 13 Port: 1 N: 29 Port: 1
               => N: 28 Name: dummy SName: dm28 Type: f
                  |->Output Ports
                     Name: o Type: 0 Obj: dm N: 28 Port: 1 N: 29 Port: 2
                     Name: o Type: 0 Obj: dm N: 28 Port: 1 N: 29 Port: 3
            |->Switch2 Function
               => N: 14 Name: dummy SName: dm14 Type: f
                  |->Output Ports
                     Name: o Type: 0 Obj: dm N: 14 Port: 1 N: 29 Port: 1
               => N: 26 Name: super SName: su26 Type: u
                  |->Input Ports
                     Name: su26_i1 Type: 0 Obj: gt N: 26 Port: 1 N: 29 Port: 3
                     Name: su26_i2 Type: 0 Obj: sh4 N: 26 Port: 2 N: 29 Port: 4
                     Name: su26_i3 Type: 0 Obj: sh5 N: 26 Port: 3 N: 29 Port: 5
                  |->Output Ports
```

```
         Name: su26_o1 Type: 0 Obj: sh4 N: 26 Port: 1 N: 29 Port: 2
         Name: su26_o2 Type: 0 Obj: sh5 N: 26 Port: 2 N: 29 Port: 3
      => N: 19 Name: gt SName: gt19 Type: f
         |->Input Ports
            Name: i Type: 6 Obj: gt N: 19 Port: 1 N: 26 Port: 1
         |->Output Ports
            Name: o Type: 0 Obj: gt N: 19 Port: 1 N: 25 Port: 1
         => N: 25 Name: samphold_samphold SName: sh4_sh5 Type: u
            |->Input Ports
               Name: sh4_sh5_sw1 Type: 0 Obj: sh4 N: 25 Port: 1 N: 19 Port: 1
               Name: sh4_sh5_func2 Type: 6 Obj: sh4 N: 25 Port: 2 N: 26 Port: 2
               Name: sh4_sh5_func3 Type: 6 Obj: sh5 N: 25 Port: 3 N: 26 Port: 3
            |->Output Ports
               Name: sh4_sh5_mg1 Type: 6 Obj: sh4 N: 25 Port: 1 N: 26 Port: 1
               Name: sh4_sh5_mg2 Type: 6 Obj: sh5 N: 25 Port: 2 N: 26 Port: 2
            |->Switch1 Function
               => N: 15 Name: copy SName: cp15 Type: f
                  |->Input Ports
                     Name: i Type: 0 Obj: cp N: 15 Port: 1 N: 25 Port: 2
                  |->Output Ports
                     Name: o Type: 0 Obj: cp N: 15 Port: 1 N: 25 Port: 1
               => N: 18 Name: dummy SName: dm18 Type: f
                  |->Output Ports
                     Name: o Type: 0 Obj: dm N: 18 Port: 1 N: 25 Port: 2
            |->Switch2 Function
               => N: 16 Name: dummy SName: dm16 Type: f
                  |->Output Ports
                     Name: o Type: 0 Obj: dm N: 16 Port: 1 N: 25 Port: 1
               => N: 17 Name: copy SName: cp17 Type: f
                  |->Input Ports
                     Name: i Type: 0 Obj: cp N: 17 Port: 1 N: 25 Port: 3
                  |->Output Ports
                     Name: o Type: 0 Obj: cp N: 17 Port: 1 N: 25 Port: 2
```

Etape 4 On traduit le graphe hiérarchique d'unités fonctionnelles optimisé en un graphe d'algorithme de SynDEx (cf. la section **??**).

5.4 Exemples

La figure **??** montre un example du graphe Scicos. La transformation de ce graphe en un graphe hiérarchique d'unités fonctionnelles est illustré par la figure **??** alors que la figure **??** montre son optimisation. Enfin, la figure **??** montre le graphe SynDEx obtenu.

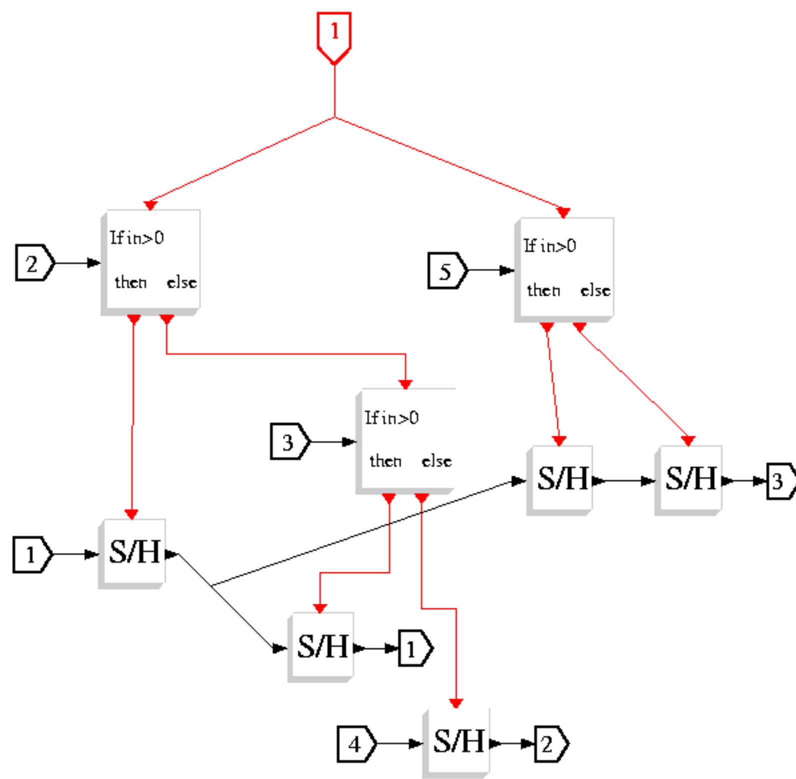


Figure 5.9: Un graphe Scicos.

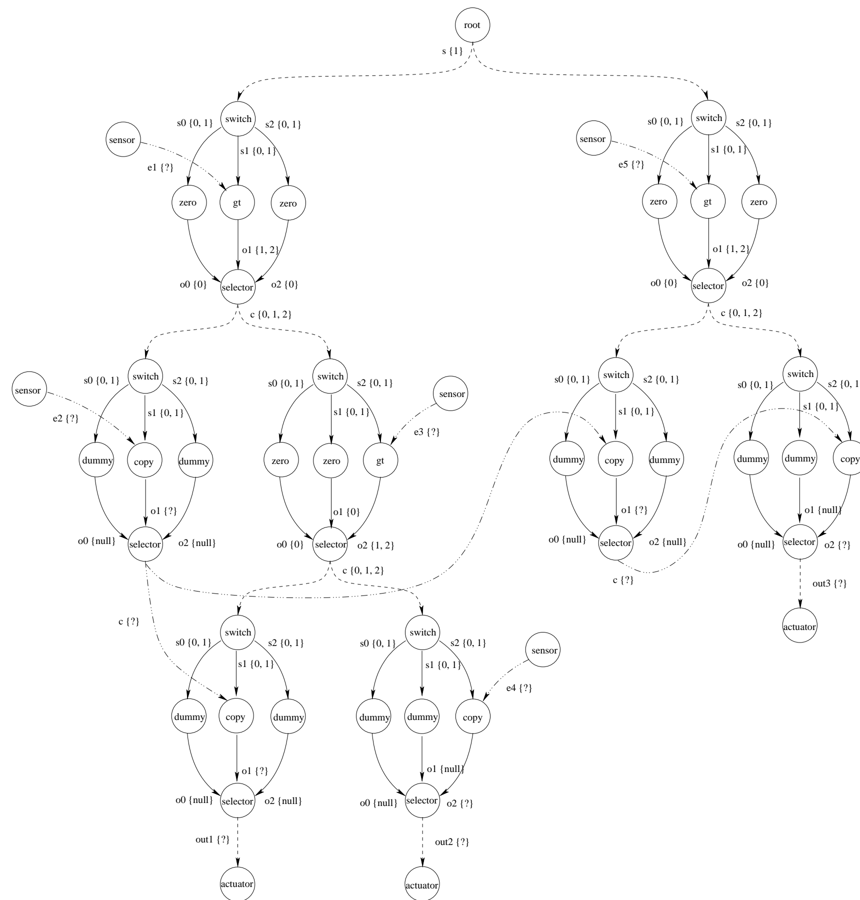
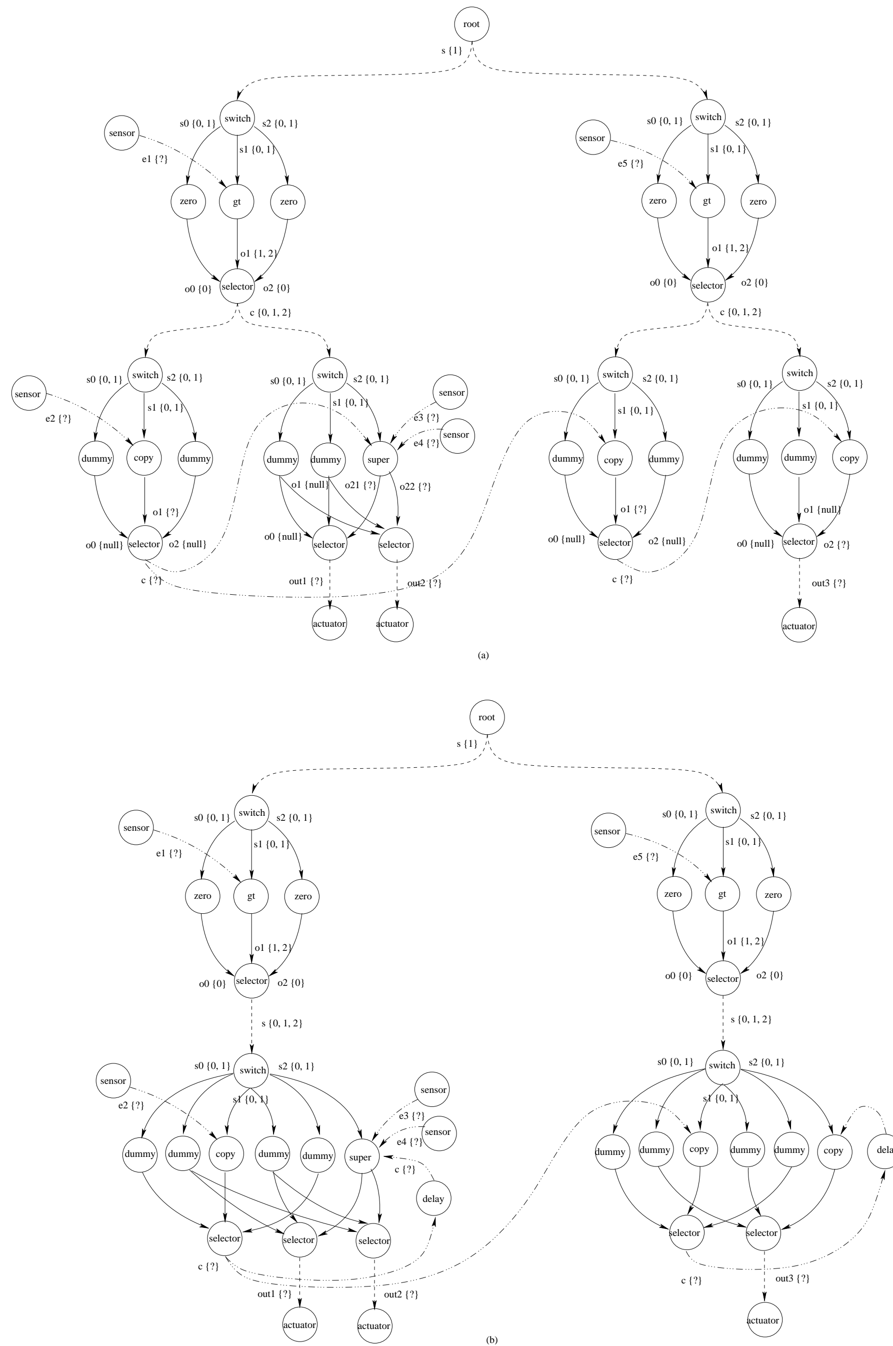


Figure 5.10: La transformation du graphe Scicos (la figure ??) en un graphe hiérarchique d'unités fonctionnelles.



(a)

(b)

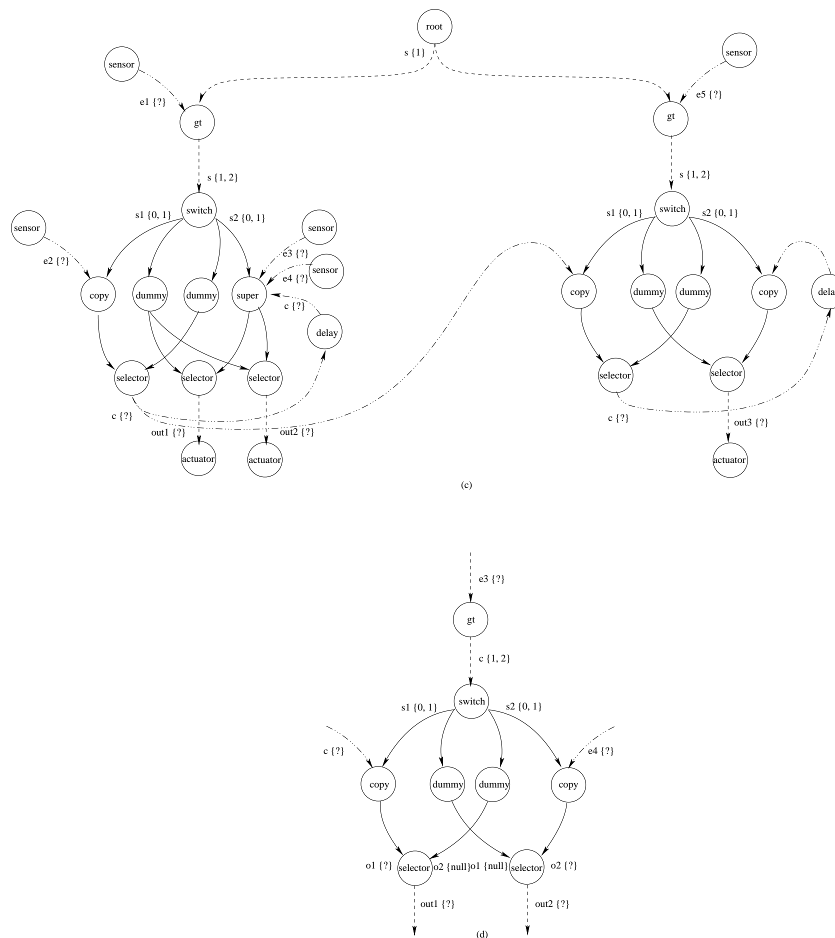


Figure 5.11: L'optimisation du graphe hiérarchique d'unités fonctionnelles (la figure ??).

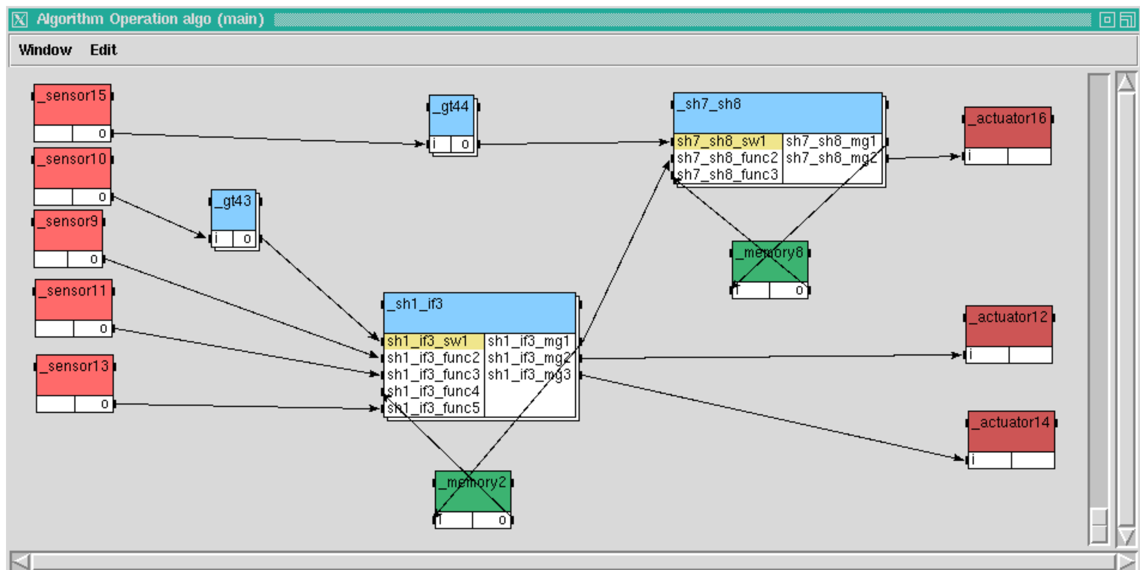


Figure 5.12: La traduction du graphe hiérarchique d'unités fonctionnelles (la figure ??) en un graphe d'algorithme de SynDEx.

Conclusion

Dans ce rapport, nous avons présenté dans un premier temps le formalisme des logiciels :

- *SynDEx*, d'aide à l'implantation temps réel multi-processeur des algorithmes,
- *Scicos*, pour la modélisation des systèmes dynamiques hybrides. Nous nous sommes particulièrement intéressés et restreints à la définition des types de signaux et à la description du fonctionnement des blocs discrets.

Ensuite, nous avons proposé le formalisme de *graphe hiérarchique d'unités fonctionnelles* comme type de graphe intermédiaire pour la transformation des graphes. Pour cela, nous avons décrit comment il permettait de décrire du contrôle tout en étant un formalisme flot de données. De plus, nous avons défini des règles pour minimiser le nombre de tests composant ce type de graphe.

Enfin, nous avons présenté une application de ce formalisme, son rôle de format intermédiaire dans la transformation de graphes Scicos en graphes d'algorithme SynDEx. Nous avons expliqué comment le graphe Scicos était traduit en graphe hiérarchique d'unités fonctionnelles, puis optimisé avant d'être transformé en graphe d'algorithme SynDEx. Une fois cette transformation validé théoriquement, nous avons développé un programme, écris en langage Scilab et inclu dans le logiciel Scicos. Ceci nous permet de valider par la pratique la méthode de transformation par l'obtention de code généré par SynDEx dont le comportement correspondait au spécification et simulation réalisées sous Scicos. Néanmoins, ce programme n'est qu'un prototype, traduisant qu'un nombre restreint de blocs Scicos et devra être complété.

Dans le cadre d'un projet RNTL nommé ECLIPSE, incluant les entreprises PSA et CS Communications & Systèmes, la transformation sera complétée selon les règles définies lors de mon stage. Par ailleurs, le formalisme de graphe hiérarchique d'unités fonctionnelles pourra servir d'interface lors de la traduction d'autres formalismes incluant du contrôle. Ces nouvelles transformations seront en outre plus faciles à prouver en utilisant ce formalisme. Enfin, différents graphes, provenant de différentes transformations, pourront être combinés dans ce format intermédiaire avant de traduire l'ensemble en graphe d'algorithme SynDEx.

Bibliography

[1] Nicolas Pernet. Spécification multi-formalisme d'algorithmes de contrôle-commande. Rapport de stage, INRIA, septembre 2002.

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[3] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: an Algebraic Approach to Program Dependencies. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 445–467. MIT Press, Cambridge, MA, 1991.

[4] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[5] Jong Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. *Conferrence Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[6] Thierry Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* PhD thesis, Université de Paris Sud, Spécialité électronique, 30/11/2000.

[7] David Harel and Amir Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*. Springer Verlag, New York, 1985.