INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Interface Scicos-SynDEx

Rachid Djenidi, Ramine Nikoukhah, Yves Sorel, Serge Steer

N° 4250

Septembre 2001

THÈME 4

ISSN 0249-6399 ISRN INRIA/RR--4250--FR

INRIA
ROCQUENCOURT

Interface Scicos-SynDEx

Rachid Djenidi, Ramine Nikoukhah, Yves Sorel, Serge Steer

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Metalau, Sosso

Rapport de recherche n° 4250 — Septembre 2001 — 68 pages

Résumé : *Scicos* est une boite à outils du logiciel libre de calcul scientifique Scilab, dédiée à la modélisation et la simulation de systèmes dynamiques. Ces systèmes, représentés sous forme de schéma-blocs, peuvent être potentiellement constitués d'éléments ayant un comportement de nature différente : continu, discret, événementiel, constituant ainsi des systèmes hybrides. Le modèle classique de ces systèmes est formé d'un modèle d'environnement physique continu, commandé par un modèle de contrôleur discret. Le fonctionnement du système dans son ensemble est simulé par l'exécution d'un programme sur une station de travail. Ce programme généré par le compilateur *Scicos*, traduit les mises à jour des blocs en fonction de l'ordonnancement déduit du schéma-bloc.

Les produits visés par les applications industrielles ne concernent que la mise en œuvre du modèle de contrôleur discret, modélisé par *Scicos*, qui doit être implanté en temps réel sur un calculateur. *SynDEx* est un logiciel libre permettant la spécification et la simulation d'une implantation optimisée d'algorithmes sur des architectures distribuées, en respectant des contraintes temps réel.

La possibilité de réaliser à la fois l'étude et automatiser la mise en œuvre de ces systèmes de commande discrets constitue donc une réponse à un réel besoin. D'où l'idée de réaliser l'interface entre *Scicos* et *SynDEx*, afin de pouvoir implanter à l'aide du logiciel *SynDEx* des algorithmes de commande, modélisés avec *Scicos*, sur des architectures matérielles en respectant des contraintes temps réel.

Mots-clés : Systèmes dynamiques hybrides, Modélisation, Simulation, Génération de code, temps réel

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Scicos-SynDEx interface

Abstract: *Scicos* is a toolbox of the free scientific software package for numerical computations *Scilab*, for modeling and simulation of dynamical systems. Such systems, designed with blocks diagram, can potentially include parts with different type of behavior: continuous, discrete and event, leading to hybrid systems. The typical model of these systems is composed of a model of continuous physical environment, interconnected with a model of discret controller. The whole system behavior is simulated by the execution of a program on a workstation. This program generated by the *Scicos* compiler manages blocks update, according to the scheduling obtained from the blocks diagram.

The products aimed by industrial applications only concern the implementation of the discret controller model designed with *Scicos*, and must be implemented in real-time on a computer. *SynDEx* is a system level CAD software, for rapid prototyping and optimizing the implementation of real-time embedded applications on multicomponent architectures.

The possibility to realize both design and automatically implement discret controller, constitutes an answer to a real need. Then, *Scicos* and *SynDEx* has been interfaced, in order to implement controller algorithms on hardware architectures while satisfying real-time constraints.

Key-words: Hybrid dynamical systems, Modeling, Simulation, Code generation, real-time

Table des matières

Les buts et motivations **i**

1 Le logiciel Scicos **1**
1.1 Les signaux . . . 1
1.1.1 Les types de signaux . . . 1
1.1.2 La terminologie . . . 3
1.1.3 Les signaux d'activations . . . 4
1.1.3.1 Le conditionnement par activation continue . . . 4
1.1.3.2 Le conditionnement par événements . . . 7
1.1.4 Les signaux réguliers . . . 7
1.1.5 L'ensemble des temps d'activations . . . 8
1.1.6 Les opérations sur les signaux réguliers . . . 8
1.1.7 Le conditionnement direct et hérité . . . 9
1.1.8 Le super-bloc . . . 10
1.1.9 La synchronisation des événements . . . 10
1.2 Les différents types de blocs . . . 15
1.2.1 Les blocs *Standards* . . . 16
1.2.2 Les blocs *Zcross* . . . 18
1.2.3 Les blocs *Synchro* . . . 18
1.2.4 Les blocs *Memo* . . . 19
1.2.5 L'ordre de mise à jour des registres . . . 20
1.2.6 Le bloc *Memo* (suite) . . . 21
1.3 La compilation de schéma bloc . . . 21
1.3.1 Les fonctions des blocs . . . 22
1.3.1.1 La fonction d'interface graphique . . . 22
1.3.1.2 La fonction de simulation . . . 22
1.3.2 La compilation d'un schéma bloc . . . 23
1.3.2.1 La première étape de compilation . . . 23
1.3.2.2 La deuxième étape de compilation . . . 23
1.4 La simulation de schéma bloc . . . 26
1.4.1 La procédure *cosini* . . . 27

1.4.2 La procédure *cossim* 27
1.4.2.1 La procédure *doit* 30
1.4.2.2 La procédure *cdoit* 30
1.4.2.3 La procédure *ddoit* 30
1.4.2.4 La procédure *edoit* 30
1.4.3 La procédure *cosend* 30
1.4.4 Le solveur *ODE* 31
1.4.4.1 L'évolution du temps continu 31
1.4.4.2 La détection de traversée de zéro 32
1.4.5 La gestion de l'agenda 32

2 Le logiciel SynDEx 35
2.1 Introduction 35
2.2 Définitions 36
2.2.1 Les signaux 36
2.2.2 Système réactif temps-réel 36
2.2.3 Le concept des langages synchrones 37
2.2.4 Architecture muti-processeurs 37
2.2.5 Parallélisme potentiel 38
2.2.6 Graphe et hypergraphe orientés 38
2.2.7 Problème NP-complet 39
2.3 La méthodologie AAA 39
2.3.1 Introduction 39
2.3.2 Algorithme : Graphe logiciel 40
2.3.3 Architecture : Graphe matériel 40
2.4 Transformation de graphes 41
2.4.1 Routage 41
2.4.2 Distribution 41
2.4.3 Ordonnancement 41
2.4.4 Adéquation 42
2.4.5 Heuristique 42
2.5 Syntaxe *SynDEx* 44
2.5.1 Instructions pour la spécification de l'algorithme 44
2.5.1.1 Instructions de déclaration de sommet de calcul 44
2.5.1.2 Instructions de déclaration de sommet mémoire 46
2.5.1.3 Instructions de connexion des sommets 46
2.5.1.4 Instruction d'exécution inconditionnelle 47
2.5.1.5 Instruction d'exécution conditionnelle 47
2.5.2 Appel de fonctions pour la génération de code 47
2.5.3 Fonctions existantes 47

3 Interface Scicos-SynDEx **49**
3.1 Les principes de l'interface 49
3.1.1 Les signaux 49
3.1.2 Le conditionnement 50
3.1.3 Les différences terminologiques 50
3.1.4 Les blocs 50
3.2 Les règles de traduction 51
3.2.1 Test sur les blocs 51
3.2.2 Test sur les paramètres des blocs 52
3.2.3 Déclarations de compilation 52
3.2.3.1 Identification des entrées et sorties 52
3.2.3.2 Liberté d'ordonnancement pour affiner le parallélisme 52
3.2.3.3 Identification des blocs 53
3.2.3.4 La fonction d'exécution inconditionnelle : "execroots" 53
3.2.3.5 La fonction d'exécution conditionnelle : "exec" 54
3.3 Un exemple de comparaison : 54
3.3.1 Les modifications dans *Scicos* 56
3.3.2 Les modifications pour *SynDEx* 56
3.3.3 Les vérifications 57
3.4 La procédure de transformations *Scicos-SynDEx* 57
3.4.1 Phase d'initialisation : fonction scv40 57
3.4.2 Déclaration des fonctions : *fonction scv41* 59
3.4.3 Constitution des connexions : *fonction scv42* 59
3.4.4 Constitution des exécutions : *fonction scv43* 59
3.4.5 Création des fichiers : *fonction scv44* 63
3.4.5.1 Le fichier *appli.m4h* 63
3.4.5.2 Le fichier *appli.m4x* 63
3.4.5.3 Le fichier *Makefile* 63
3.5 Le mode d'emploi 63
3.5.1 Interface SynDEx-Scicos 64
3.5.2 La génération de code dans SynDEx 64

Conclusion **65**

Table des figures

1 Une chaîne directe. ii

1.1 La représentation d'un système hybride dans l'éditeur *Scicos*. 2
1.2 Un signal *Scicos* et l'ensemble des temps d'activations du bloc. 4
1.3 Le conditionnement des blocs. 5
1.4 Le conditionnement continu dans la figure 1.3. 6
1.5 Les résultats de la simulation (cf.figure 1.3, page 5). 7
1.6 Une opération en temps discret sur les signaux : l'addition. 8
1.7 Le bloc *Clock* est un super-bloc. 10
1.8 L'héritage (intermédiaire) dans la figure 1.3. 11
1.9 L'héritage (total) dans la figure 1.3. 12
1.10 Le conditionnement par deux sources d'activations. 13
1.11 Une erreur à éviter. 13
1.12 Le conditionnement de la figure 1.11. 14
1.13 Le conditionnement réel du bloc 3 est hérité du bloc aval. 14
1.14 Les niveaux d'implications en fonction des super-blocs. 15
1.15 Le bloc Standard. 16
1.16 Le bloc Zcross. 19
1.17 Le bloc Synchro. 20
1.18 Le bloc Memo. 22
1.19 Les étapes de compilation d'un schéma bloc 23
1.20 Un schéma avec des blocs *Synchro* 24
1.21 Le conditionnement continu de la figure 1.20 (page 24) 25
1.22 La simulation dans *Scicos* 26
1.23 La procédure *cosini* 27
1.24 L'essentiel de la procédure *cossim* 28
1.25 Le fichier *doit* 29
1.26 La procédure *cosend* 30
1.27 La procédure *ODE* 31

2.1 Un système réactif 36
2.2 Un graphe orienté 39

2.3 Graphe matériel encapsulé 41
2.4 La méthodologie AAA 43

3.1 La représentation d'un diagramme dans l'éditeur *SynDEx*. 50
3.2 Bloc avec état discret. 54
3.3 L'égaliseur dans *SynDEx*. 55
3.4 L'égaliseur dans *Scicos*. 55
3.5 Entrées et sortie du soustracteur. 56
3.6 Mise en forme des données *Scicos* : procédure Scilab scv40. 58
3.7 Déclaration des fonctions *SynDEx* : fonction scv41. 60
3.8 Déclaration des inter-connexions en *SynDEx* : fonction scv42. 61
3.9 Déclaration des exécution en *SynDEx* : fonction scv43. 62
3.10 Étapes d'appel des sous programmes. 63

Liste des tableaux

1.1 Des éléments de terminologie. 3
1.2 Les différents “flags”. 21

2.1 Exemple de déclaration de sommet *SynDEx* 45
2.2 Récapitulatif des fonctions existantes dans Cexec 48

3.1 Type d’entrées/sorties. 49
3.2 Les principales différences de terminologies *Scicos/SynDEx*. 51
3.3 Préfixe de l’identificateur des variables. 52
3.4 Correspondance entre le type de bloc et le nom du sommet. 54

Les buts et motivations

Le travail décrit dans ce document est issu de la coopération entre les projets *METALAU et SOSSO*, de l'unité de recherche de l'INRIA à Rocquencourt. Plus précisemment il s'agit de l'interface entre la boite à outils *Scicos* du logiciel libre de calcul scientifique : *Scilab* et le logiciel d'aide à l'implantation temps réel multi-processeur d'algorithmes réactifs : *SynDEx*.

Scicos

Scicos (Scilab Connected Object Simulator, http://www-rocq.inria.fr/scilab) contient un éditeur graphique de type schéma-blocs, qui permet la modélisation et la simulation de systèmes dynamiques ([20], [7], [18], [21], [24], [19], [6], [8], [10], [11], [9]). Ces systèmes peuvent être potentiellement constitués d'éléments continus, discrets, événementiels ; réalisant ainsi des systèmes hybrides. L'exemple le plus classique de ces systèmes est un modèle d'environnement continu, commandé par un contrôleur discret, dont la simulation se traduit par l'exploitation d'un algorithme, défini par le compilateur *Scicos* et décrivant le fonctionnement du système dans son ensemble.

Théoriquement, un système hybride pourrait très bien être défini par *des lignes de code* servant d'entrée à un simulateur. Mais dans la réalité, de par la dimension sans cesse croissante de systèmes de plus en plus sophistiqués et complexes, l'utilisaton d'interfaces (éditeurs) graphiques est rendue nécéssaire. Ces éditeurs graphiques facilitent le travail de l'utilisateur mais compliquent la vie des concepteurs de simulateur.

Les spécificités du logiciel *SynDEx*, pour l'implantation d'algorithme sur des architectures matérielles distribuées, constituent une complémentarité à l'étude de systèmes de commande discrets par le logiciel *Scicos*. Cette possibilité de mise en œuvre mérite d'être exploitée, de manière à permettre l'étude et l'implantation d'algorithmes de commande de systèmes, dans une chaîne directe. D'où l'idée de réaliser une interface avec *Scicos* [7] (cf. figure 1, page ii).

SynDEx

L'étude du logiciel *SynDEx* est effectuée dans le souci d'obtenir des règles de traduction claires de la partie discrète de l'algorithme modélisée avec *Scicos*, en l'algorithme au format

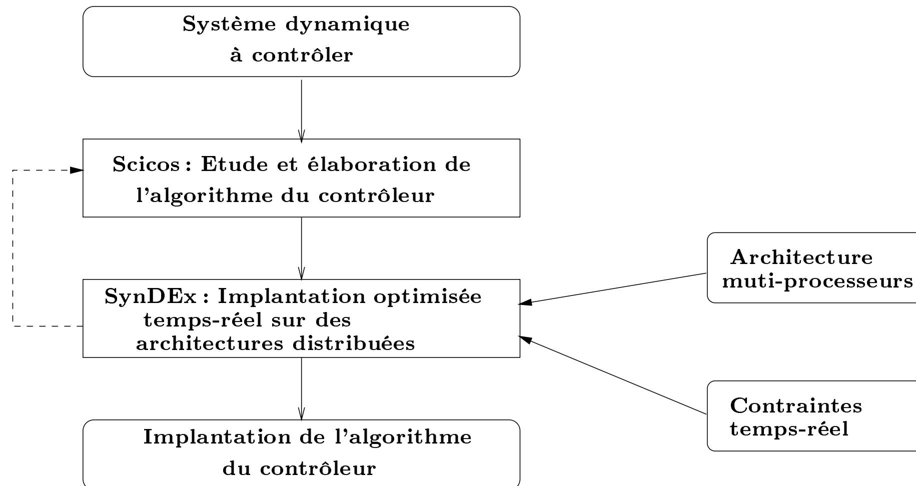


FIG. 1 – *Une chaîne directe.*

SynDEx à implanter en temps réels sur une architecture multiprocesseur spécifiée elle aussi avec *SynDEx*. A moyen terme, dans leurs développements futurs, la nécessité d'établir un format commun aux deux logiciels est à envisager pour une base de l'interface adaptable aux évolutions des deux logiciels.

Les besoins des systèmes embarqués imposent de plus en plus de conduire les études autour des systèmes complexes (automobiles, avions, usines, raffineries, centrales électriques, ...) de façon intégrée. La tendance actuelle pour la conception d'objets physiques, leur commande et le logiciel embarqué vise à une réalisation qui doit être conduite dans un environnement unifié. De manière classique, ces applications industrielles sont typiquement constituées d'une partie continue commandée par un contrôleur discret. *Scicos* est un logiciel de modélisation et de simulation, adapté à l'étude de tels systèmes. La mise en œuvre de l'algorithme de contrôle ainsi obtenu consiste en son implantation sur une architecture matérielle. Lorsque sont imposées des contraintes de temps réel, il devient nécessaire d'utiliser des architectures distribuées, afin d'exploiter le parallélisme potentiel de l'algorithme à implanter en fonction du parallélisme effectif de l'architecture. Pour exécuter cette tâche de manière efficace, l'utilisation d'un logiciel spécialisé se révèle nécessaire. Les principaux logiciels d'aide à l'implantation des systèmes embarquées sont présentés dans [13].

SynDEx (EXécutif Distribué SYNchrone, http://www-rocq.inria.fr/syndex) est un logiciel libre permettant la spécification et la simulation d'une implantation optimisée d'algorithmes sur des architectures distribuées, en respectant des contraintes temps-réel ([1], [5], [15], [16], [23], [14]). Il a été conçu et développé à l'unité de Recherche de l'INRIA à Rocquencourt.

Parmis les atouts principaux de *SynDEx* par rapport aux autres produits concurrents (CASCH, GEDAE, Ptolemy II, TRAPPER) [13], il faut retenir qu'il :

- propose des modèles cohérents pour spécifier l'algorithme et l'architecture matérielle. Cette dernière est décrite de manière détaillée afin de prendre en compte les communications interprocesseurs qui sont critiques ;
- offre la possibilité, avant la génération de code, de visualiser (simuler) un graphe temporel d'exécution des tâches ;
- génère un code intermédiaire afin que l'utilisateur, suivant une méthodologie conseillée, puisse implémenter son algorithme sur n'importe quelle architecture.

Chapitre 1

Le logiciel 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*[1]. L'analyse du formalisme est développé à travers l'étude des signaux et du fonctionnement interne des blocs.

1.1 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 1.1, on observe un système hybride formé :

- d'une partie évoluant en temps continu (le générateur de sinusoïde et l'intégrateur "1/s")
- d'une partie évoluant en temps discret (le système linéaire et l'oscilloscope), à la cadence de l'horloge .

1.1.1 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).

1. Institut de Recherche en Informatique et Systèmes Aléatoire

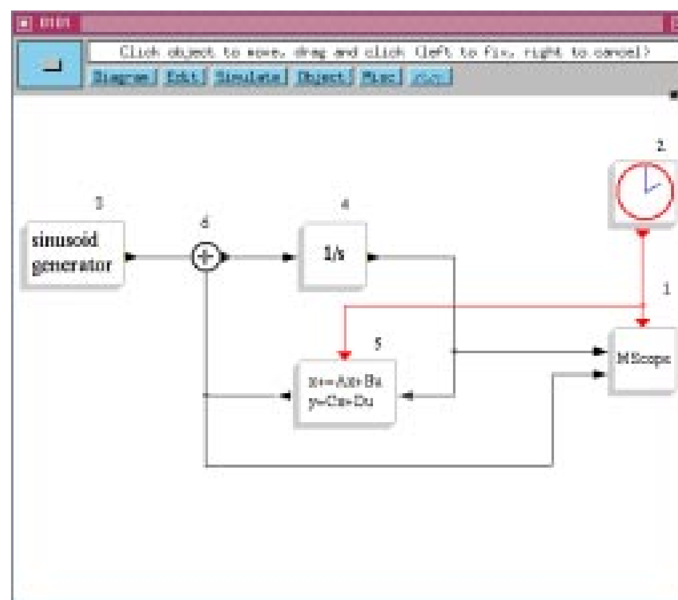


Fig. 1.1 – *La représentation d'un système hybride dans l'éditeur* Scicos.

1.1.2 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 :

- 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

TAB. 1.1 – *Des éléments de terminologie.*

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

1.1.3 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.

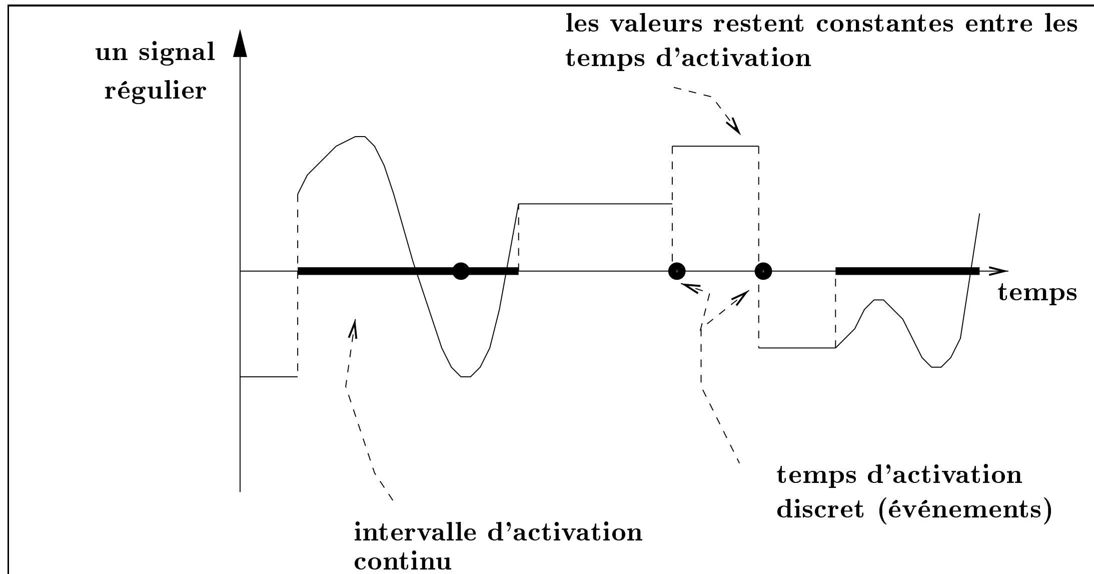


FIG. 1.2 – *Un signal* Scicos *et l'ensemble des temps d'activations du bloc.*

La figure 1.2 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*.

1.1.3.1 Le conditionnement par activation continue

Le fonctionnement continu des blocs est rendu possible grâce à un bloc fictif générant des activations continues [17]. Le conditionnement continu correspond à la mise à jour des blocs de manière continue. Dans l'exemple de la figure 1.4 (page 6), 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 1.1.7, 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.

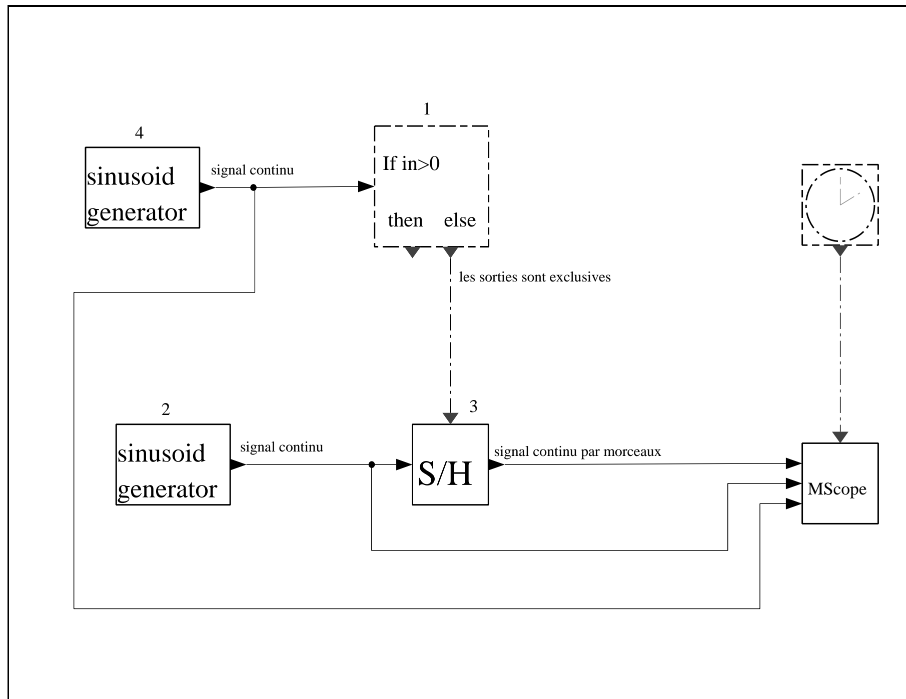


FIG. 1.3 – *Le conditionnement des blocs.*

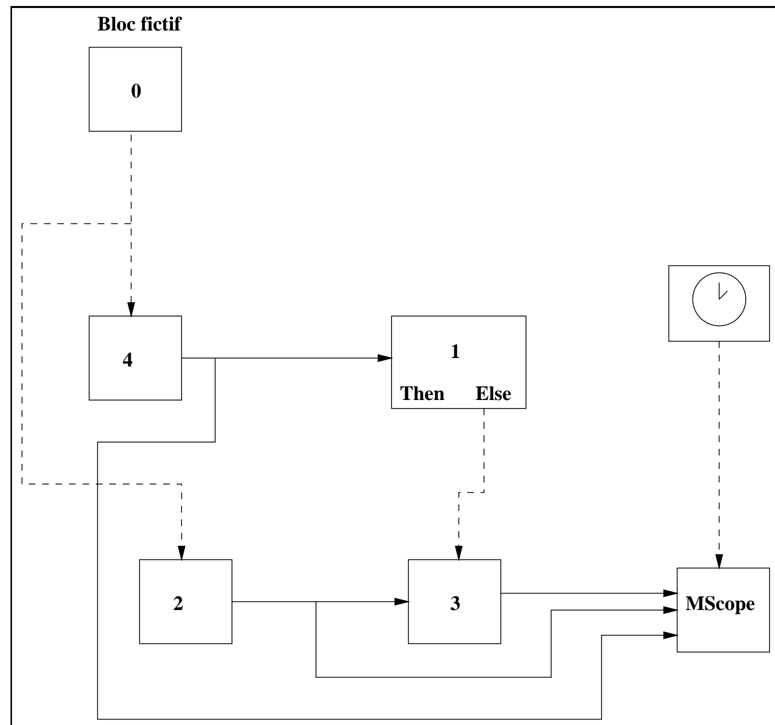


FIG. 1.4 – *Le conditionnement continu dans la figure 1.3.*

1.1.3.2 Le conditionnement par événements

Les événements sont représentés sur la figure 1.2 par des points qui marquent le caractère instantané de leur occurrence. L'exemple de la figure 1.1 (page 2) 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 1.3 (page 5) il n'y a que le bloc *MScope* qui est activé de manière discrète.

Sur la figure 1.5, 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.

FIG. 1.5 – *Les résultats de la simulation (cf.figure 1.3, page 5).*

1.1.4 Les signaux réguliers

Dans le schéma de la figure 1.3 (page 5), 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 du langage *Signal* [3]. 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 1.2, page 4).

1.1.5 L'ensemble des temps d'activations

Les temps d'occurrence des signaux d'activations de sortie sont contenus dans le registre des temps d'activations (cf. fig 1.15, page 16). Ces valeurs peuvent être programmées soit à l'initialisation, soit durant la simulation.

L'ensemble des temps d'activations T est l'union des segments et indices temporels pour lesquels le signal évolue. Sur la figure 1.2 (page 4) on observe que le signal à évolué pour des activations ponctuelles (événements) et pour des intervalles d'activation continue, traduisant un fonctionnement continu par morceaux. On remarque dans cette figure que l'occurrence d'une activation discrète peut tout à fait avoir lieu pendant une phase d'activation continue. De manière générale, l'expression de l'ensemble des temps d'activations T d'un bloc, peut s'écrire sous la forme suivante :

$$\{T_{bloc}\} = \sum_{k=i}^{f} t_k$$

Ici t_k indique la date d'occurrence de l'activation continue ou discrète, t_i *et* t_f $(f \geq i)$, désignant la date d'occurrence de l'activation respectivement initiale et finale.

1.1.6 Les opérations sur les signaux réguliers

Dans *Scicos*, les opérations sur les signaux s'effectuent à travers la *fonction de calcul* des blocs (cf. chap 1.3, page 21). Chaque bloc exécute sa *fonction de calcul* au rythme des activations qui le conditionnent. Prenons pour exemple un bloc *Scicos* exécutant l'addition de deux signaux ($\mathbf{E_1(t)}$ et $\mathbf{E_2(t)}$) via ses deux ports d'entrées (cf. figure 1.6, page 8).

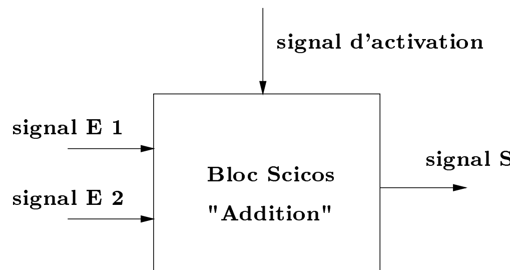


FIG. 1.6 – *Une opération en temps discret sur les signaux : l'addition.*

Les opérations sont exécutées à la cadence du conditionnement du bloc “Addition”. Le conditionnement est caractérisé par l'ensemble des temps d'activations, exprimé par $\mathbf{T_a}$. Nous obtenons en sortie du bloc, un signal régulier $\mathbf{S(t)}$, que spécifie l'expression suivante :

$$\{\mathbf{E_1(t)}, \mathbf{T_{E_1}}\} + \{\mathbf{E_2(t)}, \mathbf{T_{E_2}}\} \Leftrightarrow \{\mathbf{S(t)} = (\mathbf{E_1(t)} + \mathbf{E_2(t)}), \mathbf{T_a}\}$$

De manière plus générale, pour chaque bloc réalisant une opération de calcul entre ses n entrées régulières, avec une activation caractérisée par $\mathbf{T_G}$ (l'ensemble des temps d'activations conditionnant le bloc), le signal de sortie $\mathbf{S(t)}$ peut se traduire sous la forme :

$$G(\{S_1(t), T_1\}, ..., \{S_n(t), T_n\}) \Leftrightarrow \{S(t) = G(S_1(t), ..., S_n(t)), T_G\}.$$

1.1.7 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 1.3 (page 5) 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 1.3 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 1.8). Soulignons que le bloc *Clock* qui conditionne le bloc *MScope* est en fait un bloc *Delay* relié sur lui-même (cf. figure 1.7, page 10). 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* (cf. chap 1.1.8, page 10).

Nous verrons par la suite que l'héritage doit aussi prendre en compte les activations générée par le bloc *Clock* (cf. figure 1.9, page 12).

Dans l'exemple de la figure 1.10 (page 13) le conditionnement du bloc *Mux* est hérité de celui de ses trois entrées régulières. Les deux premières sont conditionnées par les blocs 1 et 3, tandis que la dernière est conditionnée par le bloc fictif. Ainsi, la notion d'héritage permet d'allèger de manière significative la représentation graphique, en évitant de faire apparaître les liaisons d'activations du bloc.

Une erreur à éviter concerne l'héritage de l'activation continue sur un bloc discret comme pour l'exemple de la figure 1.11 (page 13). En effet, pour ce schéma, dont l'héritage est explicité dans la figure 1.12 (page 14), la mise à jour de l'état discret du bloc 3 est conditionnée par le bloc fictif et par le bloc 1. En réalité nous verrons dans le chapitre 1.4.2, à la figure 1.24 (page 28) que la mise à jour du registre d'état discret n'est pas effectuée pour une activation continue, donc ici la mise à jour de l'état discret ne dépend que des activations du bloc 1. Autrement dit le bloc 3 n'hérite dans la pratique que du conditionnement du bloc aval (cf. figure 1.13, page 14), ce qui entraine donc un résultat de simulation différent de celui auquel on pourrait imaginer à première vue.

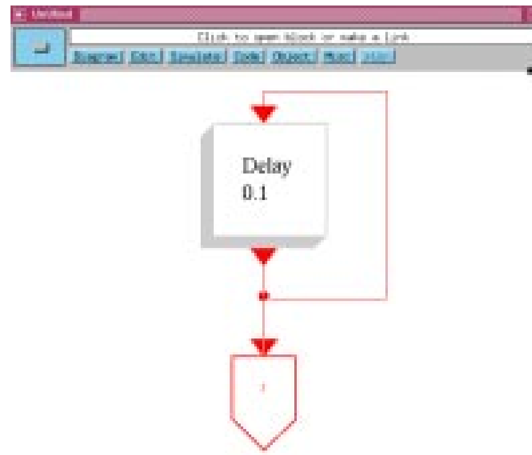


FIG. 1.7 – *Le bloc* Clock *est un super-bloc.*

1.1.8 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 (cf. fig. 1.14, page 15).

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.

Considérons l'exemple de la figure 1.14, constitué d'un bloc relié directement à un super-bloc A, qui contient lui-même deux blocs reliés au super-bloc B.

Le schéma bloc ainsi composé est caractérisés par trois niveaux :

- le niveau 2 pour le super-bloc B,
- le niveau 1 pour le super-bloc A,
- le niveau 0 pour le schéma bloc dans son ensemble.

1.1.9 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)

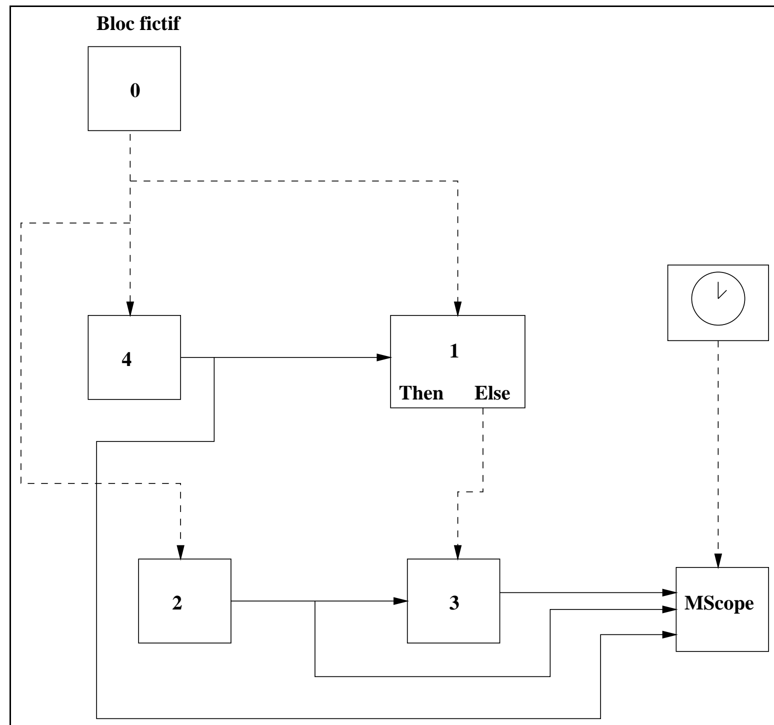


FIG. 1.8 – *L'héritage (intermédiaire) dans la figure 1.3.*

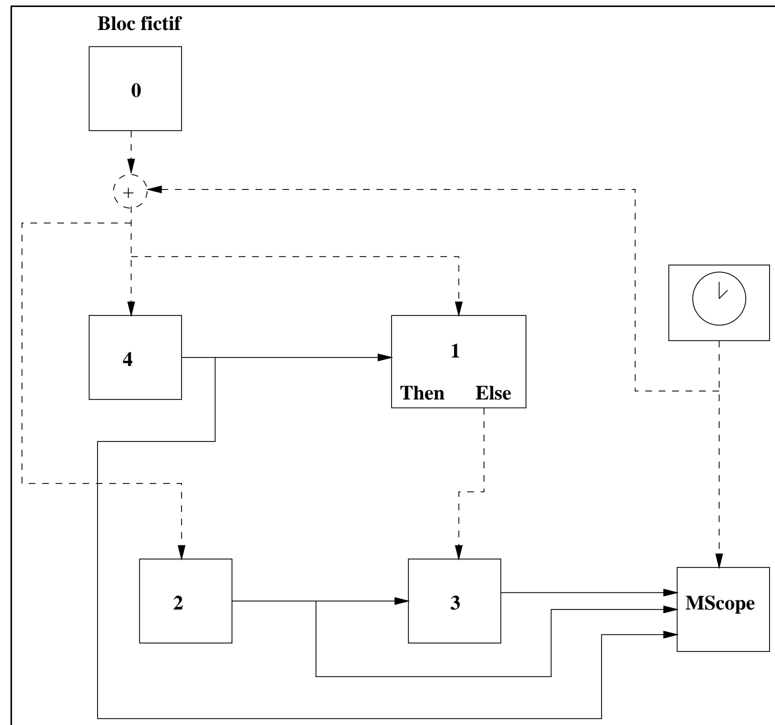


FIG. 1.9 – *L'héritage (total) dans la figure 1.3.*

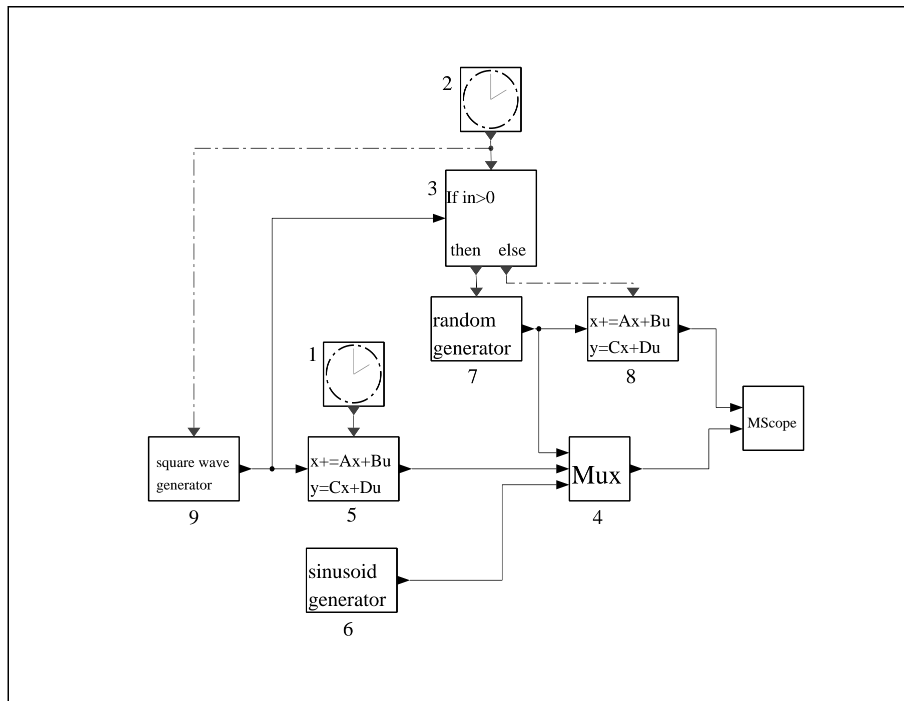


FIG. 1.10 – *Le conditionnement par deux sources d'activations.*

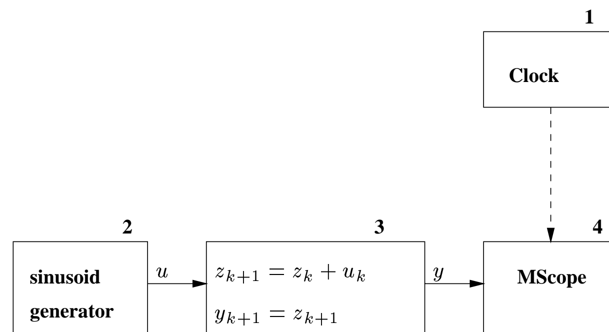


FIG. 1.11 – *Une erreur à éviter.*

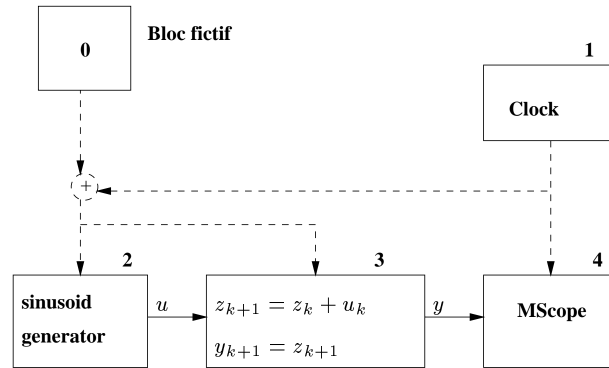


FIG. 1.12 – *Le conditionnement de la figure 1.11.*

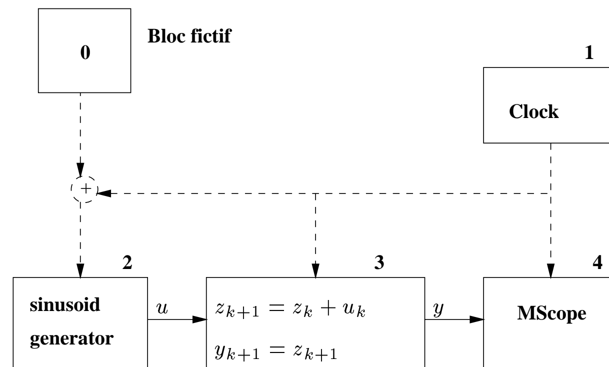


FIG. 1.13 – *Le conditionnement réel du bloc 3 est hérité du bloc aval.*

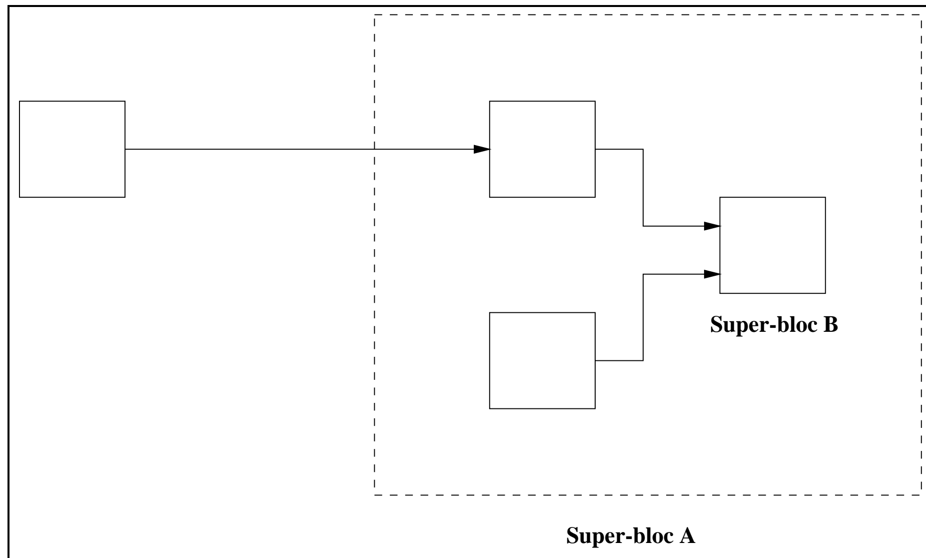


FIG. 1.14 – *Les niveaux d'implications en fonction des super-blocs.*

mais seront considérés et traités dans un ordre chronologique indéterminé ; il s'agit dans ce cas d'événements *simultanés*.

Dans l'exemple de la figure 1.10 (page 13), du fait du conditionnement continu du bloc 6, la synchronisation du signal sur la troisième entrée est assurée avec chacun des signaux réguliers des deux premières entrées.

1.2 Les différents types de blocs

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

- le bloc *"Standard"* qui représente le type de bloc le plus courant dans *Scicos*.
- le bloc *"Synchro"* qui est utilisé le plus souvent pour le conditionnement d'autres blocs,
- le bloc *"Zcross"* qui permet des applications liées à la détection de seuil,
- le 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 (cf. chap. 1.1.8, page 10). Soulignons qu'il ne s'agit que d'une facilité graphique et non d'un nouveau type de bloc.

1.2.1 Les blocs *Standards*

Les blocs *Standard* permettent de modéliser un fonctionnement en temps discret, en temps continu ou les deux à la fois. Comme l'indique la figure 1.15 (page 16), un bloc *Standard* peut être constitué de plusieurs registres. Les signaux de sortie sont stockées dans les registres de sortie y_i. Les deux registres d'état (*continu* x *et discret* z) 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 continu doit être conditionné par activation continue. Selon le cas, ces blocs possèdent des ports d'entrée et de sortie réguliers et d'activations.

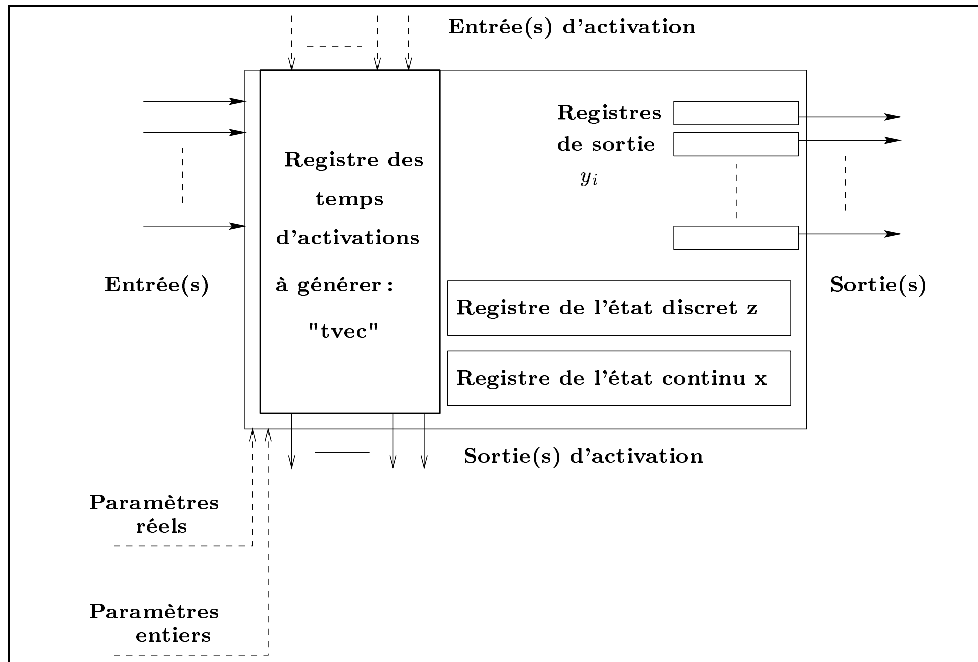


FIG. 1.15 – *Le bloc Standard.*

La mise à jour des registres se fait en fonction des signaux d'entrée et du conditionnement du bloc. En prenant pour vecteur d'entrée régulière u et pour vecteur de sortie régulière y, un bloc continu *Standard* impose, pendant la période d'activation continue, les relations (1.1)-(1.2).

$$\frac{dx}{dt} = f(t, x(t), z, u(t), p) \tag{1.1}$$

$$y(t) = h(t, x(t), z(t), u(t), p) \tag{1.2}$$

où f et h sont les fonctions spécifiques au bloc et p est le vecteur des paramètres constants.

Un événement au temps t_e (t_e^- étant le temps précédent l'occurrence de cet événement), peut provoquer un saut des états du bloc continu *Standard*, décrits par les équations suivantes :

$$x(t_e) = g_c(t_e, x(t_e^-), z(t_e^-), u(t_e), p, n_e) \tag{1.3}$$

$$zt_e = g_d(t_e, x(t_e^-), z(t_e^-), u(t_e), p, n_e) \tag{1.4}$$

où g_c et g_d sont les fonctions spécifiques du bloc. Ici $z(t_e^-)$ est la valeur précédente de l'état discret ; z demeure constant entre les temps d'activations.

n_e représente l'entier codant les ports activés ; en effet, si les ports d'entrée activés sont désignés par $i_1, i_2, \ldots, i_n$, alors

$$n_e = \sum_{j=1}^{n} 2^{i_j - 1}$$

Dans le cas, par exemple, d'un bloc possédant deux ports d'activations, la variable n_e peut prendre les valeurs suivantes :

- 0 si le bloc est activé par le bloc fictif (activation continue) ;
- 1 si le bloc est activé par le premier port d'entrée d'activations (en commençant par la gauche du bloc) ;
- 2 si le bloc est activé par le deuxième port d'entrée d'activations (à gauche du bloc) ;
- 3 si le bloc est activé de façon **synchrone**, par les deux ports d'entrée d'activations.

Si un événement active à l'instant t_e un bloc régulier, ce dernier peut éventuellement générer un signal d'activations de type événementiel à l'instant t_{evo}, défini par :

$$t_{evo} = k(t_e, x(t_e), z(t_e), u(t_e), p, n_e) \tag{1.5}$$

pour une fonction spécifique k du bloc et où t_{evo} est un vecteur de temps, dont chaque élément correspond à un port de sortie d'activations. Le vecteur de temps t_{evo} est contenu dans un registre de sortie d'activation *"tvec"*.

1.2.2 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 (cf. fig. 1.16, page 19).

L'équation 1.6 indique par le vecteur de temps t_{evo}, l'instant pour lequel un événement peut-être généré. Le vecteur t_{evo} est contenu dans le registre de sortie d'activation *"tvec"* du bloc *Zcross*.

$$t_{evo} = T_z(t_e, u(t_e), p) \tag{1.6}$$

Le bloc *ZCROSS* dans la palette *Threshold* constitue l'exemple de référence. En effet, ce bloc ne génère une activation que si son signal d'entrée traversent zéro (change de signe). Dans le cas d'un signal d'entrée vectoriel, il faut que la valeur de tous les éléments du vecteur, traversent zéro simultanément.

Un autre exemple est représenté par les blocs - *to* + et + *to* - qui génèrent une activation lorsque le signal de l'unique entrée traverse zéro, respectivement avec une pente positive et négative.

Une forme plus générale de ce type de bloc est proposée par le bloc *GENERAL* dans la palette *Threshold*. Il génère une activation si au moins un de ses signaux d'entrée traverse zéro. Ce qui donne, en fonction du nombre i de signaux d'entrée, le nombre n_{zc} de possibilités de traversée de surface suivant : $n_{zc} = 2 \cdot 2^i$.

Notons que les signaux d'entrée des blocs de détection de traversée de zéro ne doivent pas être constant de valeur nulle. En effet, la détection est basée sur les calculs d'un solveur numérique qui, dans un tel cas, s'engage dans des calculs très longs (voir sans issue) ; cette situation est donc à prohiber pour éviter que le solveur ne s'emballe pendant la simulation.

1.2.3 Les blocs *Synchro*

Les blocs *Synchro* (cf. figure 1.17, page 20) 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 u et de la fonction du bloc (par exemple if u >0), vers un des ports de sortie d'activations, codés par un entier n_s (cf. équ. 1.7). 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 *"tvec"*. Par contre on utilise un registre appelé *"ntvec"* pour contenir la valeur de n_s.

$$n_s(t_e) = l(t_e, u(t_e), p) \tag{1.7}$$

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*.

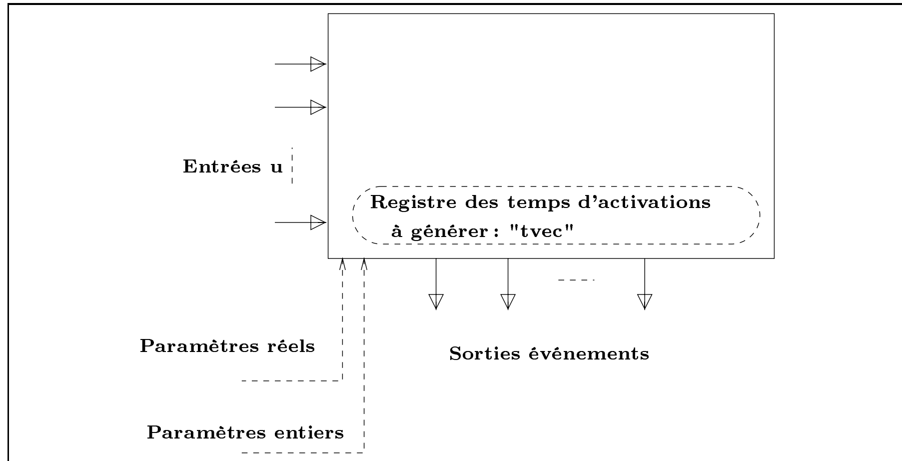


FIG. 1.16 – *Le bloc Zcross.*

L'activation d'entrée est synchrone avec l'activation générée par *l'union* des ports de sorties d'activations, avec N nombre total de ports d'activations de sortie :

$$\{T_{entree}\} = \{T_1\} \cup \{T_1\} \cup \cdots \{T_N\}$$

Par exemple pour un bloc *If then Else* le synchronisme des activations se traduit par :

$$\{T_{entree}\} = \{T_1\} \cup \{T_2\}$$

Des exemples de blocs de ce type sont les blocs *"event select"* et *"If-then-else"*.

1.2.4 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.

En fait, pour comprendre le fonctionnement des blocs *Memo*, nous avons besoin de connaitre *l'ordre des mises à jour des registres dans* Scicos. C'est la raison pour laquelle, nous reprendrons l'étude des blocs *Memo*, après le paragraphe suivant.

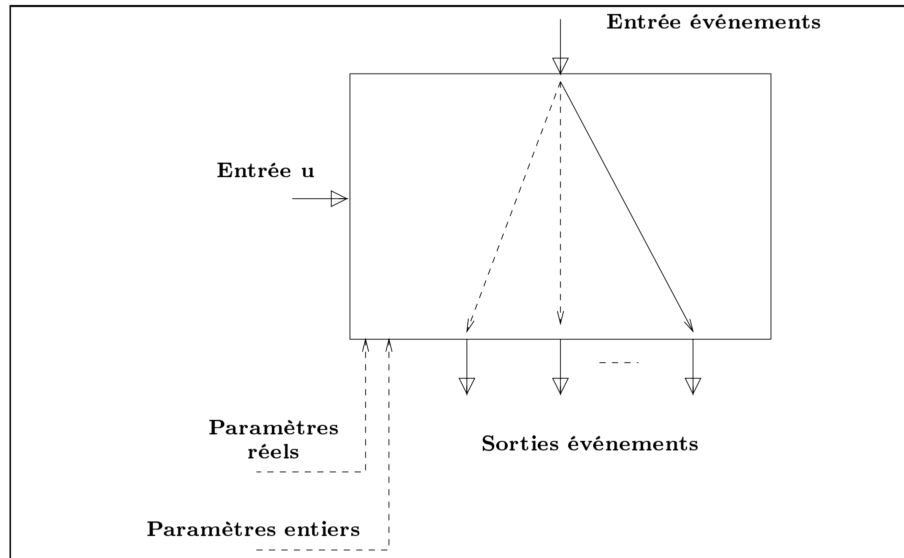


FIG. 1.17 – *Le bloc Synchro.*

1.2.5 L'ordre de mise à jour des registres

Comme nous l'avons vu précédemment, il existe différents registres selon les types de bloc :

- le registre des temps d'activations à générer : *tvec* ;
- le registre de l'état discret ;
- le registre de l'état continu ;
- les registres de sorties régulières.

En ce qui concerne l'évolution des signaux réguliers, deux types de registres sont utilisés : les registres de sorties régulières et les registres d'état(s) (si le bloc en est pourvu). Ces registres sont mis à jour au rythme de leur conditionnement (continu ou discret). La mise à jour des différents registres des blocs peut se faire dans des ordres différents en fonction de leur type ("régulier", "Zcross", "Synchro" et "Memo") et leur conditionnement. Cette double dépendance détermine le fonctionnement de chaque bloc en fonction d'opérations ordonnées. Le tableau 1.2 (page 21) détaille ces opérations qui sont exécutées pendant la phase de simulation (cf. chap 1.4, page 26). Les opérations suivantes de mise à jour des registres sont exécutées systématiquement dans le même ordre préétabli :

1. *avec "flag 1"*: Les registres de sortie des blocs concernés sont mis à jour en respectant la relation d'ordre (ordonnancement) définit par le schéma bloc.

2. *avec "flag 3"*: les événements, programmés dans *l'agenda* de simulation et contenus dans le registre *"tvec"* des blocs concernés, sont générés par les ports de sortie d'activations.
3. *avec "flag 2"*: le registre d'état discret des blocs concernés sont mis à jour en même temps, dans un ordre quelconque.

Les opérations avec "flag 0" ne sont effectuées que pour les blocs qui possèdent un registre d'état continu. En fait les sorties sont mises à jour chaque fois qu'un bloc "aval" doit être mis à jour, l'intégrateur d'*ODE* met à jour l'état continu.

Flag	*Opération*
0	Mise à jour de l'état continu
1	Mise à jour des sorties
2	Mise à jour de l'état discret
3	Génération des activations (de sorties)
4	Initialisation de simulation
5	Fin de simulation
6	Ré-initialisation

TAB. 1.2 – *Les différents "flags".*

1.2.6 Le bloc *Memo* (suite)

Dans le bloc *Memo*, la mise à jour du registre de sortie se fait, non pas au moment où à lieu celle des registres de sortie des autres type de blocs, mais au moment de la mise à jour de leur registre d'état discret. Ce qui se traduit pour le registre de sortie d'un bloc *"Memo"* par la recopie de la valeur de ses entrées au moment où les registres d'état discrets sont mis à jour (cf. chap. 1.2.5, page 20). Ce type de bloc ne peut et ne doit pas être conditionné par activation continue.

Si le bloc "Memo" est activé par un événement à l'instant t_e, la valeur de la sortie y (cf. équ. 1.8) est mise à jour au même instant. En dehors des occurrences d'activations la valeur de la sortie y reste constante.

$$y(t) = m(t_e, u(t_e^-), p) \tag{1.8}$$

La restriction que doit s'imposer l'utilisateur lors de l'utilisation d'un bloc de type Memo *est de toujours relier directement sa sortie à l'entrée d'un bloc* Synchro*, afin d'éviter toute confusion dans l'ordonnancement et de garantir par conséquent le résultat de la simulation.*

1.3 La compilation de schéma bloc

Ici nous présentons les deux fonctions caractérisant chaque bloc, ainsi que les principales étapes de compilation d'un schéma bloc dans *Scicos*.

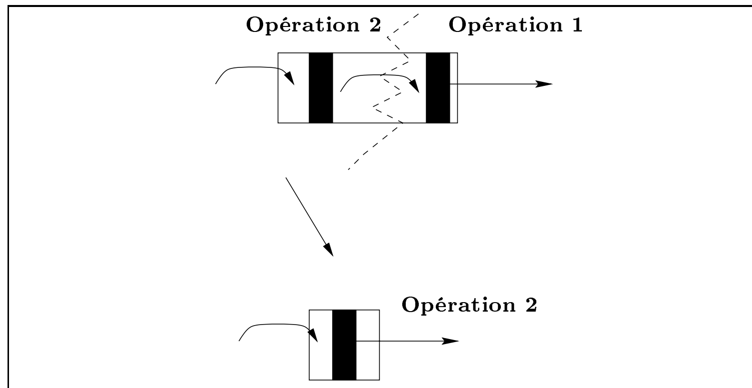


Fig. 1.18 – *Le bloc Memo.*

1.3.1 Les fonctions des blocs

Chaque bloc est systématiquement caractérisé par deux fonctions [22] :

1. une *fonction d'interface graphique*,
2. une *fonction de simulation*, appelée aussi *fonction de calcul*.

1.3.1.1 La fonction d'interface graphique

La *fonction d'interface graphique*, écrite en langage Scilab, définit pour chaque bloc :

- les caractéristiques graphiques : la géométrie du bloc, sa couleur de fond, son icône, le nombres et la taille de ses ports d'entrées et sorties (réguliers et d'activation), etc.
- les caractéristiques de l'interface avec l'utilisateur (GUI : Graphical User Interface) dans l'éditeur *Scicos*. Ces caractéristiques peuvent être diverses : la valeur initiale des états, les différents paramètres, ainsi que le dialogue permettant à l'utilisateur de modifier les propriétés du bloc, par un simple clic de souris.

1.3.1.2 La fonction de simulation

La *fonction de simulation* est utilisée par le simulateur pour exécuter la dynamique caractérisant le comportement des blocs. Elle doit permettre une exécution rapide, c'est la raison pour laquelle elle est souvent écrite en langage C ou Fortran. Cependant, dans sa phase de mise au point elle peut aussi être écrite en langage Scilab. La fonction de simulation est appelée au cours de la simulation pour effectuer plusieurs opérations. Chacune des ces opérations est codée, dans la liste d'appel de la fonction de simulation, par un drapeau (*flag*) (cf. tableau 1.2, page 21).

1.3.2 La compilation d'un schéma bloc

La compilation est une étape nécessaire pour coder les informations graphiques, architecturales et paramétriques d'un schéma bloc. La compilation s'exécute, une première fois, en deux étapes (cf. fig. 1.19, page 23). Par la suite, lorsque des modifications sont effectuées dans le schéma bloc (changements de paramètres, de la tailles des entrée et sorties ...), elles sont prises en compte de différentes manières.

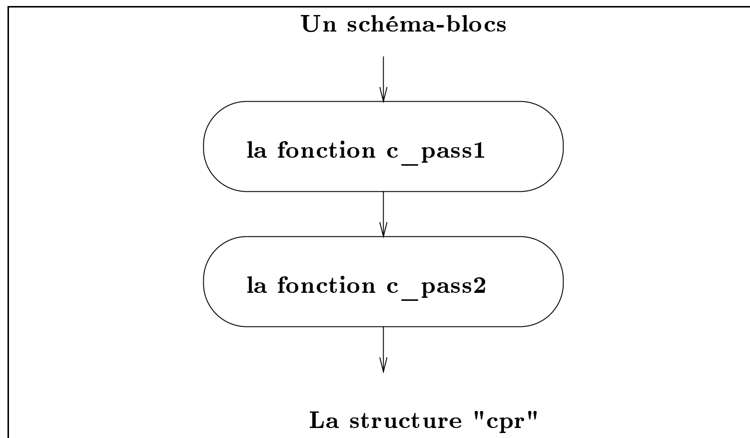


FIG. 1.19 – *Les étapes de compilation d'un schéma bloc*

1.3.2.1 La première étape de compilation

Cette étape est réalisée par la fonction Scilab *c_pass1* (/macros/scicos/c_pass1.sci). Son rôle consiste à exécuter :

- la vérification de la cohérence entre les paramètres des ports des blocs reliés ;
- la mise à plat de la hiérarchie des super-blocs ;
- la construction des tables de connections (régulières et d'activations) entre les blocs.

Le résultat obtenu est basé sur une nouvelle numérotation des blocs.

1.3.2.2 La deuxième étape de compilation

Cette étape est réalisée par la fonction Scilab *c_pass2* (/macros/scicos/c_pass2.sci). Elle consiste à construire l'algorithme d'ordonnancement des mises à jour des registres des blocs, dans le respect de la relation d'ordre définit par le schéma bloc.

L'ordonnancement tient compte de l'aspect hybride d'un schéma bloc, en traitant :

- *pour la partie continue*, un seul registre d'état continu constitué des registres d'état continu de chaque bloc. Le calcul de l'évolution des états est assuré par un solveur d'équations de type *ODE* ;
- *pour la partie discrète*, la gestion de la programmation des activations des blocs.

Pour cela on exploite deux propriétés spécifiées pour chaque bloc. La première indique si la sortie du bloc dépend directement de son entrée ($y = f(u)$) et la deuxième indique si le bloc est activé en continu ou non. Dans l'exemple de la figure 1.20 (page 24), seuls les blocs 5 et 7 sont activés en continu. Notons pour le bloc 2 que la matrice D est nulle, sa sortie ne dépend donc pas directement de son entrée.

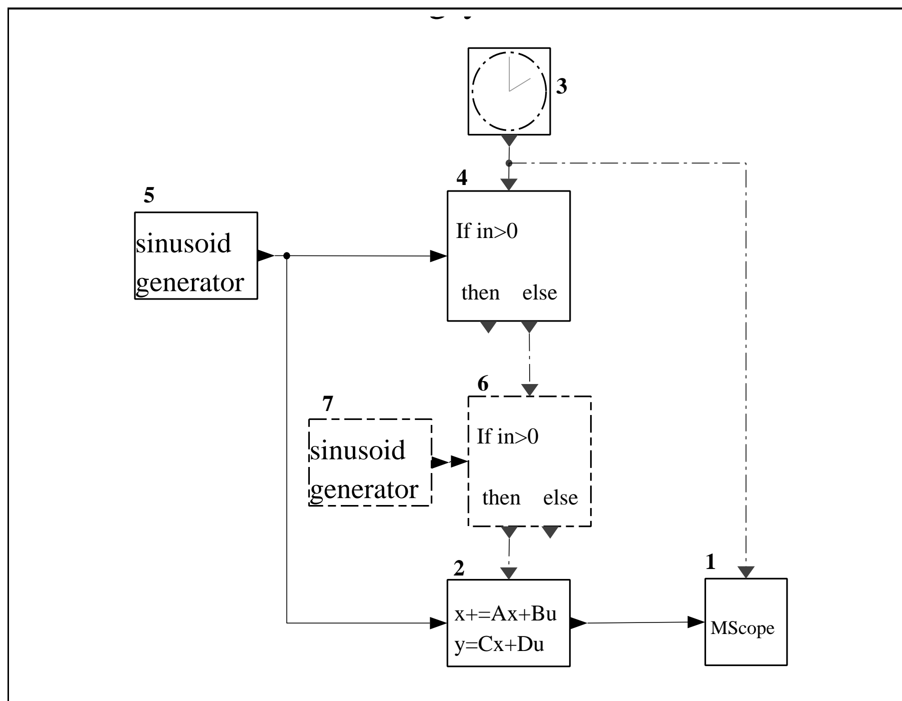


FIG. 1.20 – *Un schéma avec des blocs* Synchro

Le conditionnement continu dans la figure 1.20 est repésenté de manière explicite par le schéma de la figure 1.21 (page 25).

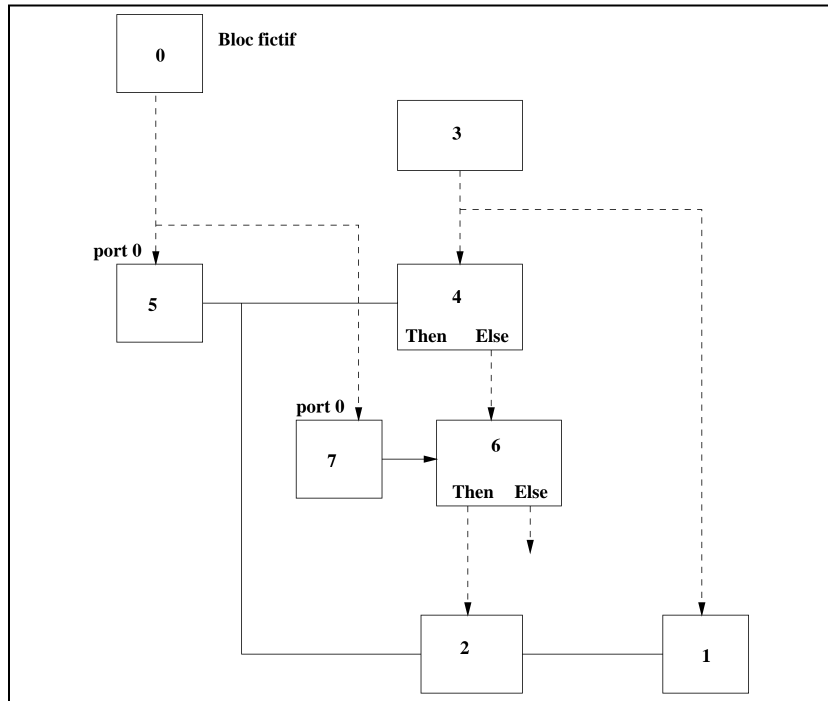


FIG. 1.21 – *Le conditionnement continu de la figure 1.20 (page 24)*

1.4 La simulation de schéma bloc

Comme il est décrit sur la figure 1.22 (page 26), la simulation d'un schéma blocs s'effectue par des appels à différentes procédures. Lorsque l'utilisateur lance une simulation, cela correspond à un appel successivement aux deux procédures suivantes :

1. *intcos*, qui gère l'interface entre les programmes du simulateur (procédure *scicos*) et le logiciel Scilab.
2. *scicos*, qui effectue, selon la valeur de "flag", différentes combinaisons d'appels à 3 procédures : *cosini, cossim, cosend*, dont le rôle est de mettre à jour les différents registres des blocs d'un schéma *Scicos*. Ces mises à jour sont effectuées :

   - à l'initialisation de la simulation, pour la procédure *cosini* (cf. chap. 1.4.1, page 27).
   - pendant la durée de simulation, pour la procédure *cossim* (cf. chap. 1.4.2, page 27).
   - afin de clore la simulation, pour la procédure *cosend* (cf. chap. 1.4.3, page 30).

   La combinaison de ces procédures permet à l'utilisateur, d'interrompre et de reprendre une simulation en cours.

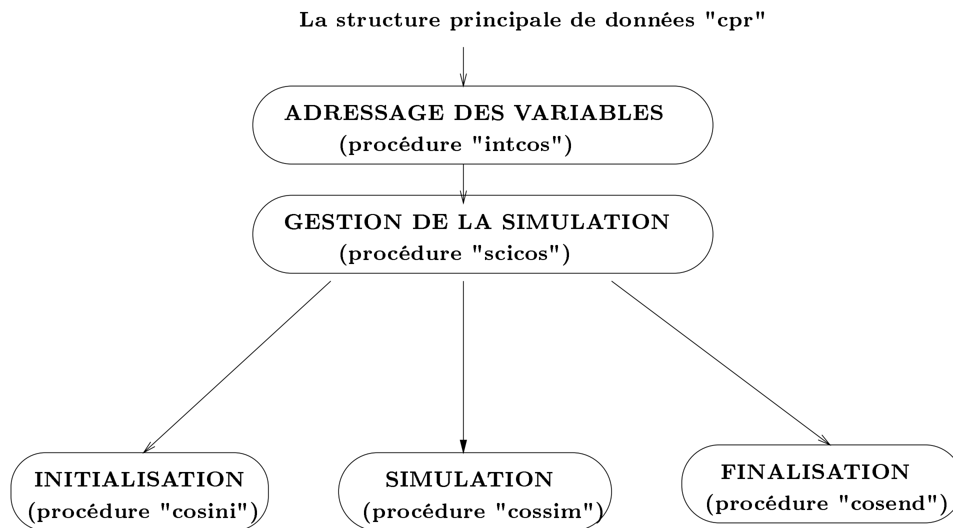


FIG. 1.22 – *La simulation dans* Scicos

Dans la suite de ce chapitre, nous présentons les principales procédure utilisées pour la simulation, dont celles précédemment citées.

1.4.1 La procédure *cosini*

Cette procédure réalise la phase d'initialisation de la simulation par les flags 4 et 6 (cf. figure 1.23, page 27).

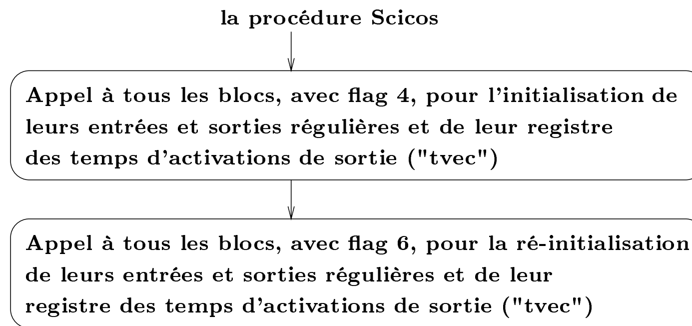


FIG. 1.23 – *La procédure* cosini

1.4.2 La procédure *cossim*

Dans les faits, l'essentiel de la simulation est assuré par la procédure *cossim*, qui gère les appels à des procédures de mise à jour des blocs, selon deux approches (cf. figure 1.24, page 28) :

- *continue*, avec la procédure *cdoit* qui met à jour les sorties régulières et d'activations (respectivement avec flag 1 et 3) nécessaires pour le calcul de l'état continu des blocs. Dans la procédure *cdoit*, on utilise :
  - la procédure *ODE* qui met en œuvre un solveur d'équations différentielles ordinaires pour les calculs d'intégration liés à l'état continu des blocs. L'état continu des blocs ainsi concernés est mis à jour par flag 0.
  - la procédure *doit* qui assure la récursivité de la procédure *cdoit* pour les blocs continus de type *Synchro*.
- *discrète*, avec la procédure *ddoit* qui met à jour les blocs discrets avec ou sans état discret, en fonction de l'occurrence des activations reçues. Les sorties régulières et d'activations sont mises à jour (respectivement par flag 1 et 3) avant les états discrets (par flag 2). On utilise la procédure *edoit* qui assure la récursivité de la procédure *ddoit* pour les blocs discrets de type *Synchro*.

Le fichier `doit` contient les quatres procédures *doit, cdoit, ddoit et edoit* (cf. figure 1.25, page 29).

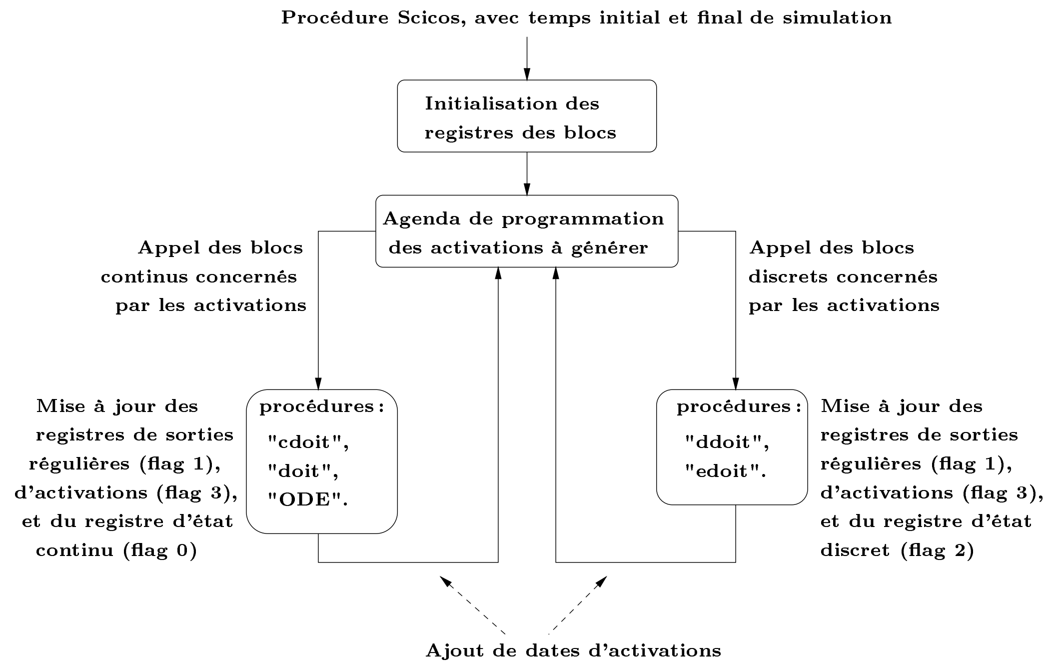


FIG. 1.24 – *L'essentiel de la procédure* cossim

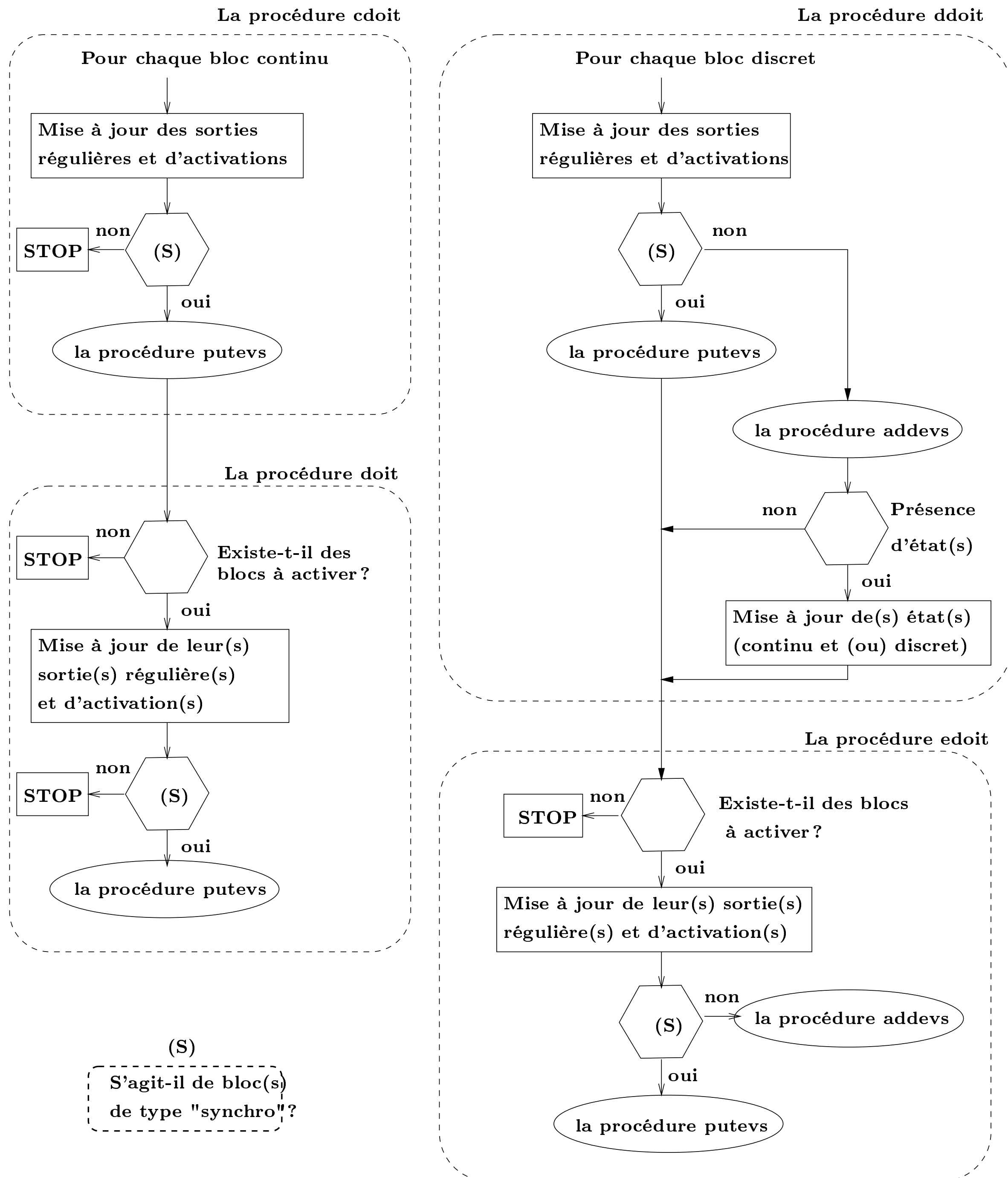


FIG. 1.25 – *Le fichier* doit

1.4.2.1 La procédure *doit*

La procédure *doit* intervient dans le traitement (mises à jour) des blocs discrets activés par un bloc continu de type *Synchro*, elle est appelée par la procédure *cdoit*. L'emploi de la procédure *doit* est destiné aux blocs devant générer une activation synchrone avec celle reçue. Ce traitement prioritaire et spécifique des activations se traduit par une gestion ramifiée verticalement (dans le sens des liens d'activations).

1.4.2.2 La procédure *cdoit*

La procédure *cdoit* gère la mise à jour des blocs continus. Elle fait appel à d'autres procédures *doit, putevs.*

1.4.2.3 La procédure *ddoit*

La procédure *ddoit* gère les mises à jour des blocs discrets. Elle fait appel à d'autres procédures *edoit, putevs, addevs.*

1.4.2.4 La procédure *edoit*

La procédure *edoit* intervient dans la mise à jour des blocs activés par un bloc discret de type *Synchro*, elle est appelée par la procédure *ddoit*. L'emploi de la procédure *edoit* est destiné aux blocs devant générer une activation synchrone avec celle reçue. Ce traitement est similaire à celui de la procédure *doit*.

1.4.3 La procédure *cosend*

Cette procédure permet de mettre à zéro, avec flag 5, le registre des temps d'activations de chaque bloc afin de terminer la simulation (cf. figure 1.26, page 30). Terminer la simulation correspond à une procédure d'arrêt, qui prend toute sa signification par exemple pour les blocs "Writef", écrivant dans un fichier qu'il est nécessaire de "fermer" pour valider leur contenu.

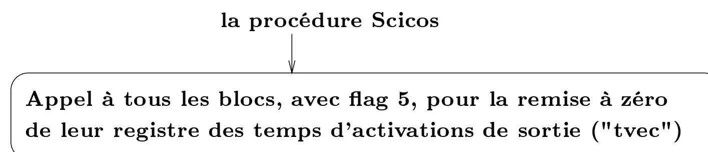


FIG. 1.26 – *La procédure* cosend

1.4.4 Le solveur *ODE*

Le solveur *ODE* est un programme d'intégration numérique des équations différentielles du schéma, qui permet de faire progresser le temps continu d'un pas et donnent la valeur de l'état à ce nouvel instant. (cf. figure 1.27, page 31).

Agenda de programmation
des activations à générer
Assure la récursivité de
de la procédure "odoit",
pour les blocs
de type "synchro"
procédure
"oodoit"
Assure la récursivité de
de la procédure "zdoit",
pour les blocs générant
une activation instantanée
procédure
"zzdoit"
Le bloc possède
un état continu
Mise à jour
des blocs avec état continu,
(par flag 1, 3, 0)
procédure
"lsodar"
procédure
"odoit"
Mise à jour
des blocs de type
"zcross" (par flag 1 et 3)
procédure
"zdoit"
STOP
non
Problème
d'intégration ?
oui
Interface entre
les formats
procédure
"grblk"
procédure
"lsoda"
Interface entre les formats
procédure
"simblk"
"lsodar" et "lsoda" : pour les calculs d'intégration

FIG. 1.27 – *La procédure* ODE

L'évolution du temps est obtenue en fonction d'un pas, calculé de manière interne et variable (suivant les calculs d'intégration) par la procédure *lsodar*.

La procédure *lsodar* est aussi utilisée pour les calculs liés à *la détection de traversée de zéro.*

1.4.4.1 L'évolution du temps continu

Les calculs liés à l'évolution du temps nécessite de fournir à la procédure *lsodar*, la dérivée de l'état continu du système à simuler. Cette dérivée est un vecteur fournit par la procédure *simblk* et composé de la dérivée pour chacun des blocs. Pour chaque bloc, la dérivée est une fonction de l'état continu du bloc et de l'entrée du bloc.

La procédure *simblk* permet de réaliser une interface entre le format de la procédure *lsodar* et celui de la procédure *odoit*. La procédure *odoit* est utilisée pour calculer la valeur de l'entrée des blocs ayant un état continu. Cette procédure fait appel à la procédure *oodoit*, qui assure une récursivité pour les calculs impliquant des blocs de type *Synchro*.

1.4.4.2 La détection de traversée de zéro

Les calculs liés à la détection de traversée de zéro nécessite de fournir à la procédure *lsodar*, une fonction ($y = f(x,t)$) qui retourne les valeurs de l'état continu du système à simuler. x est l'état continu, t est le temps continu et y est un vecteur dont le nombre d'éléments est fonction du nombre de blocs de type *Zcross* à simuler. y est le vecteur des valeurs d'équations de surface qui indique une position par rapport à la dérivée de l'état continu. La détection de traversée de zéro pour l'ensemble des blocs de type *Zcross* d'un système est considérée comme un ensemble de surfaces. La valeur des éléments de y varie entre trois possibilités : positive au dessus du seuil, négative en dessous et nulle sur le seuil. C'est la procédure *grblk* qui fournit l'équation $y = f(x,t)$ à la procédure *lsodar*. Si l'un des élément de la fonction $y = f(x,t)$ change de signe, la procédure *lsodar* engage une dichotomie du pas de temps avant et après la détection de traversée de zéro, de manière à détecter le bloc concerné et déterminer la date exacte du changement de signe.

Il est à noter que dans le cas particulier de détection de traversée de zéro pris à l'instant "zéro" même, la procédure *lsodar* se lance dans des calculs d'intégration particulièrement longs, tant et si bien que la simulation s'en trouve considérablement ralentie. Pour pallier ce problème il est nécessaire de faire avancer le temps d'intégration à l'instant suivant de manière à reprendre les calculs d'intégration, cette fois-ci avec la procédure *lsoda*. La procédure *lsoda* est adaptée uniquement aux blocs de type *Zcross*. Il est à signaler que nul besoin est de reprendre plus d'une fois cette tentative de décalage du temps d'intégration, car si elle échoue il s'agit réellement d'un problème d'intégration plus complexe. Dans ce cas la simulation est arrêtée et un message d'erreur invite l'utilisateur à faire des modifications dans le schéma blocs.

La procédure *grblk* permet de réaliser une interface entre le format de la procédure *lsodar* et celui de la procédure *zdoit*. La procédure *zdoit* est utilisée pour calculer la valeur de l'entrée des blocs de type *Zcross*. Cette procédure fait appel à la procédure *oodoit*, qui assure une récursivité pour les calculs impliquant des blocs de type *Synchro*.

1.4.5 La gestion de l'agenda

La gestion de l'agenda consiste pour chaque itération de l'algorithme de simulation, à programmer, par les procédures *doit, cdoit, ddoit* et *edoit*, les numéros de blocs devant générer une activation à l'itération suivante, ainsi que leur date d'occurrence.

L'*agenda* est le terme employé pour désigner la programmation des dates d'activations (continue et discrètes) générées par les blocs au cours de la simulation. De manière générale, ces activations dépendent des valeurs des signaux réguliers et sont donc programmées au fur et à mesure des itérations de l'algorithme de simulation.

L'*agenda* est initialisé par la fonction *init_ agenda*. Le rôle de cette fonction est l'initialisation des événements pré-programmés par les blocs ; par exemple le bloc *Clock* qui génere le premier événement à une date fixée par un paramètre.

Chapitre 2

Le logiciel SynDEx

2.1 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'executer des algorithmes réactifs. L'environnement *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 nulle 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 spécifier 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 œuvre dans le logiciel *SynDEx* (acronyme de la traduction anglaise d'EXecutif Distribué SYNchronisé) [23].

2.2 Définitions

2.2.1 Les signaux

Un signal, désigné par un nom, est caractérisé par la suite ordonnée de ses valeurs typées et par son horloge permettant une gestion implicite du temps (logique). Par exemple, $\mathbf{x}$ dénote une séquence infinie $\{x_t\}_{t\geq 0}$. Les signaux peuvent être caractérisés par des scalaires de type entier, réel, réel double précision et booléen auxquels s'ajoute le type spécifique *event* qui permet de déclarer un signal de type événement pur (un tel signal a pour valeur la constante booléenne "vraie"). D'autre part, ces signaux peuvent être de type vectoriel permettant de définir des vecteurs à une ou plusieurs dimensions, dont les éléments appartiennent à un même type de base. Par exemple, une matrice $n\ X\ m$ de booléens se déclare en [n,m] *logical* M. L'horloge d'un signal spécifie, relativement à une suite d'instants, les instants auxquels les valeurs du signal sont présentes. Contrairement aux variables dans les langages classiques, ces valeurs ne sont pas persistantes, c'est-à-dire qu'elles sont disponibles uniquement aux instants ainsi déterminés. Dans le respect strict des principes régissant les systèmes synchrones, il ne s'agit ici en aucune façon d'horloges absolues : le rôle des horloges est de permettre de parler des *relations temporelles* existant entre les divers signaux. En particulier, des contraintes de synchronisation et de logique peuvent être spécifiées dans les programmes.

2.2.2 Système réactif temps-réel

Un système *réactif* reçoit des informations en entrée, venant de l'environnement, appellées stimuli ou événements, effectue des opérations et réagit en produisant des événements en sortie, utilisables par l'environnement (cf. figure 2.1).

REMARQUE : Ces stimuli ou événements sont des signaux numériques du point de vue de *Scicos*, cela comprend les signaux réguliers discrêts et les signaux événementiels.

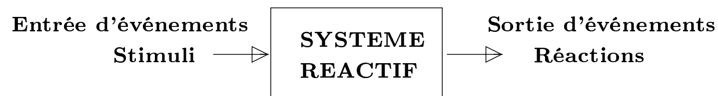


FIG. 2.1 – *Un système réactif*

Un système réactif est dit *temps-réel* s'il réagit en respectant les contraintes de temps liées à l'application traitée. Ces contraintes sont de deux types :

- **La latence** : intervalle de temps entre la réception de la 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.

Nota
La notion de temps réel ne peut donc pas être définie indépendamment du contexte dans lequel on se trouve. Une application temps-réel dans le domaine de la vision robotique impose une latence de l'ordre de la milliseconde avec une cadence du même ordre de grandeur. En revanche, dans le domaine de la météorologie par exemple, le temps de réponse admis est de l'ordre de l'heure ou de la journée.

2.2.3 Le concept des langages synchrones

Dans le langage flot de données *Signal*, le concept synchrone existe à travers deux notions : d'une part les systèmes réactifs et d'autres part la réaction instantanée avec ce qui la déclenche. Cela suppose que les calculs sont réalisés instantanément lors d'une récation ; on ignore l'aspect matériel et ses conséquences temporelles physiques, en vue d'être déterministe.

L'interface d'un système avec le monde extérieur au système est décrite par un ensemble de signaux d'entrée et un ensemble de signaux de sortie. A chaque signal est associée une séquence de valeurs ayant toutes le même type. Cette séquence définit un temps logique local au signal associant un instant logique à chaque valeur de la séquence. L'ensemble des instants logiques d'un signal est appelé horloge du signal. Le comportement du système peut être décrit à l'aide de mémoire d'état et d'une relation globale transformationnelle combinant les valeurs des signaux d'entrée du système et de sortie des mémoires pour produire les valeurs des signaux de sortie du système et d'entrée des mémoires. Le signal de sortie d'une mémoire d'état est obtenu à partir de son signal d'entrée en ajoutant une valeur initiale à sa séquence de valeurs, décalée d'un instant logique. L'absence de valeurs en entrée d'une relation entraîne une absence de valeur en sortie et donc une absence de dépendance entre entrée et sortie.

La relation globale peut être modélisée par un graphe flot de données dont les sommets sont les relations élémentaires et dont les arcs sont les signaux intermédiaires que l'on peut voir comme des relations de dépendance entre sommets (pour plus de détails, on peut consulter la thèse [4]). Les relations élémentaires et la relation globale ainsi que le graphe correspondant s'appellent "cycle d'états" en *Scicos*.

2.2.4 Architecture muti-processeurs

Les architectures multi-processeurs permettent d'augmenter la puissance de calcul par rapport à un mono-processeur, ou de distribuer géographiquement les calculs pour des raisons

de modularité et pour rapprocher ces calculs des capteurs et actionneurs, afin de diminuer globalement la longueur des câblages. Elles nécessitent de découper le programme de départ en autant de programmes que de processeurs et impliquent donc des communications entre processeurs. Chaque processeur doit communiquer des résultats de son travail à un autre processeur, ce transfert pouvant s'effectuer en parallèle avec une séquence d'instructions sur le CPU (Unité Centrale de Programmation).

Les communications inter-processeurs sont critiques dans la conception de système multi-processeurs. Les systèmes multi-processeurs à hautes performances nécessitent des transferts de données rapides entre processeurs.

2.2.5 Parallélisme potentiel

Lorsqu'un algorithme est spécifié à l'aide d'un langage de programmation impératif (purement séquentiel), un ordre total est défini sur l'exécution de toutes les opérations à réaliser alors que certaines de ces opérations, n'étant pas en relation de dépendances de données (les calculs de l'une ne dépendent pas des résultats des calculs de l'autre), pourraient être exécutées en parallèle. Il est préférable d'utiliser un langage déclaratif décrivant un ordre partiel sur les opérations. Dans ce cas, on n'impose un ordre d'exécution sur deux opérations, que lorsqu'elles sont en relation de dépendances de données. Cet ordre partiel définit le parallélisme potentiel de l'algorithme. Les opérations qui ne sont pas en relation pourront être affectées à des ressources différentes et donc, être éventuellement effectuées en parallèle, si le parallélisme effectif de l'architecture le permet. Dans le cas contraire il faudra imposer un ordre d'exécution compatible avec l'ordre partiel de l'algorithme. Il faut noter que si l'algorithme est spécifié de manière impérative, il sera nécessaire d'effectuer une analyse de dépendance pour l'exécuter sur une architecture parallèle. La notion d'algorithme utilisée ici est une extension de la notion habituelle telle que définie par Turing par exemple. D'une part on considère qu'un algorithme est associé à un ordre partiel plutôt que total, et d'autre part la réunion d'un ensemble d'algorithmes est aussi appelée algorithme [1].

Dans le cadre de l'interface *Scicos/SynDEx* le parallélisme est mis en évidence de façon explicite par la structure graphique dans le logiciel *Scicos*. L'algorithme qui en découle est à utiliser tel quel dans l'environnement SynDEX.

2.2.6 Graphe et hypergraphe orientés

Un graphe est constitué par un ensemble de sommets et un ensemble d'arcs reliant des couples de sommets. Un graphe orienté est un graphe pour lequel plusieurs arcs ne peuvent aboutir sur un même sommet (cf. figure 2.2).

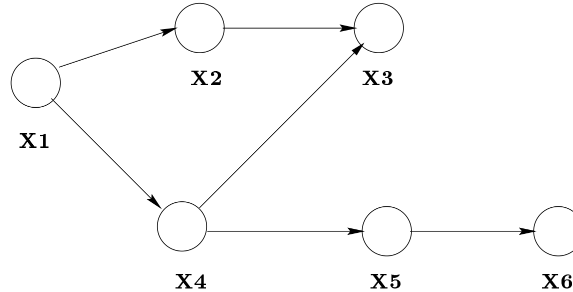
Un graphe orienté [12] $G = [X, U]$ est déterminé par la donnée :

- d'un ensemble X dont les éléments sont appelés des *sommets* ou des *nœuds*. Si $N = |X|$ est le nombre de sommets(de nœuds), on dit que le graphe G est d'ordre N. On supposera que les sommets sont numérotés $i = 1, ..., N$.

– d'un ensemble U dont les éléments $u \in U$ sont des couples ordonnés de sommets appelés des *arcs*. Si $u = (i, j)$ est un arc de G,i est l'extrémité initiale de u et$j$ l'extrémité terminale de u. On notera souvent $|U| = M$.

Graphiquement, les sommets peuvent être représentés par des points et $u = (i, j)$ sera représenté par une flèche joignant les deux points i et j (j étant la pointe de la flèche).

Un hypergraphe est une spécialisation d'un graphe, caractérisé par des hyperarcs (extension aux n_uplet de la notion d'arc qui est un couple de nœuds) afin de spécifier de la diffusion de données, par exemple : (X4, X3, X5) dans la figure 2.2.



Xi : Sommets ou nœuds reliés entre eux par des arcs

FIG. 2.2 – *Un graphe orienté*

2.2.7 Problème NP-complet

Un problème(P) [12], de taille(n) est dit NP-complet (Non Polynomial) si on ne peut pas trouver un algorithme(A) donnant la solution en un nombre(f) polynomial d'opération par rapport à la taille (variable) du problème. Ce qui sous-entend que la solution ne peut pas être un trouvée en un temps polynomial.

2.3 La méthodologie AAA

2.3.1 Introduction

La complexité des applications temps-réel embarquées nécessite à la fois des outils de spécification de haut niveau et des architectures multi-composants. Afin de réduire le nombre d'erreurs de spécification des algorithmes et de limiter au maximum les tests matériels, de nouvelles méthodes sont proposées. Elles permettent à l'utilisateur de se concentrer sur les aspects temporels qui sont cruciaux dans le domaine du temps-réel (réactivité du programme et temps de réponse contraint), d'étudier les relations entre le parallélisme potentiel au

niveau de l'algorithme et celui disponible au niveau de l'architecture, afin d'être déchargé de la programmation de bas niveau (exécutifs) souvent fastidieuse. La méthode AAA - Adéquation Algorithme Architecture - est une méthode de ce type, permettant d'aider à l'implantation d'un algorithme sur une architecture donnée, conduisant éventuellement à proposer des modifications de l'architecture (dimensionnement), ou à remettre en cause l'algorithme.

2.3.2 Algorithme : Graphe logiciel

L'algorithme est modélisé par un hypergraphe orienté. Chaque sommet du graphe (nœuds) représente une opération de calcul, d'entrées-sorties, de mémorisation ou de conditionnement.

Chaque arc reliant deux nœuds traduit :

- un transfert itératif de données (*flot de données*), établissant une précédence entre deux actions.
- un ordre partiel sur les opérations à réaliser, encore appelé *ordre d'exécution* ;

L'ensemble des arcs ainsi définis et l'ensemble des nœuds associés forment un *graphe de dépendance des données* ou encore un *graphe flot de données*. Les nœuds sans prédécesseurs (resp. sans successeurs) sont les nœuds d'entrée (resp. de sortie) ; ils représentent l'interface avec l'environnement.

Ce modèle met en évidence le parallélisme potentiel de l'algorithme (ordre partiel induit par les précédences du graphe) et la mémoire d'état. Cette dernière est représentée par l'ensemble des nœuds particuliers (*notés par un $*) qui permettent d'une part de mémoriser un élément du flot de données d'une exécution à l'autre du graphe et d'autre part de "casser" les circuits dans le graphe flot de donnés (cf. chap. 4). Le graphe flot de données est ré-exécuté chaque fois qu'une donnée se présente sur un nœud d'entrée.

2.3.3 Architecture : Graphe matériel

L'architecture est modélisée par un graphe non orienté représentant un réseau de processeurs MIMD : Multiple Instructions Multiple Data (chaque processeur effectue son programme sur ses propres données) ou SPMD : Single Program Multiple Data (plusieurs processeurs effectuent le même programme sur des données différentes), dont chaque sommet est un processeur et chaque arc est une liaison physique de communication bidirectionnelle qui permet des transferts de données entre les mémoires des processeurs, au besoin par l'intermédiaire d'une mémoire commune.

Un processeur comprend une unité de calcul, une unité d'interface avec l'environnement (E/S), une unité de communication pour chaque arc adjacent, une unité de mémoire partagée.

2.4 Transformation de graphes

L'implantation du graphe logiciel sur le graphe matériel est formalisé en termes des trois transformations de graphes suivantes : routage, distribution, ordonnancement.

2.4.1 Routage

Le routage est une transformation du graphe matériel. Le but de cette transformation est de définir les différentes *routes* permettant de communiquer d'un processeur à l'autre, lorsque ceux-ci ne sont pas reliés directement. Cela revient à transformer le graphe matériel afin qu'il soit complètement connecté. Bien que le graphe matériel soit nécessairement connexe, certains processeurs ne sont pas forcément reliés entre eux par des liens directs ou pas uniquement. Autrement dit, pour communiquer d'un processeur à un autre, on peut utiliser le lien direct (s'il existe) ou passer par un(des) processeur(s) intermédiaire(s) (cf.figure 2.3).

Lors du routage seul le graphe matériel est modifié.

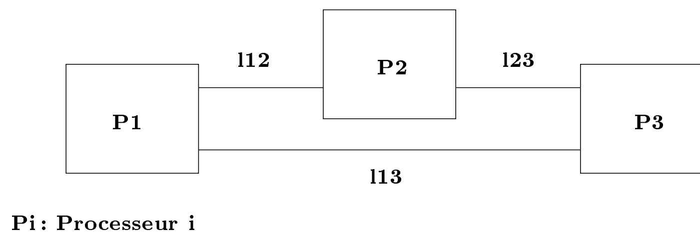


FIG. 2.3 – *Graphe matériel encapsulé*

2.4.2 Distribution

On appelle *distribution*, l'allocation *spatiale* des sommets du graphe logiciel aux sommets du graphe matériel et l'allocation *spatiale* des dépendances de données aux liens physiques.

A l'issue de la *distribution*, on doit pouvoir associer un nœud (resp. un arc) du graphe logiciel à un nœud (resp. un arc) du graphe matériel.

2.4.3 Ordonnancement

On appelle *ordonnancement*, l'allocation *temporelle* des sommets du graphe logiciel *distribués* aux sommets du graphe matériel et l'allocation *temporelle* des dépendances de données aux liens physiques.

Cette transformation consiste à étudier les relations d'ordre de tous les sous-graphes engendrés par les opérations de calcul ou de communication contraintes à être exécutées sur des *unités* de calcul ou de communication. Chaque unité est un automate séquentiel. La

relation associée à chaque automate doit donc être totale à l'issue de l'*ordonnancement* et elle doit comprendre l'ordre partiel associé aux arcs de dépendances de données du graphe logiciel de départ.

L'ordonnancement va donc consister à rajouter des arcs au graphe logiciel distribué. Ces nouveaux arcs, contrairement aux arcs initiaux vont représenter simplement les précédences d'exécution du graphe logiciel distribué et non plus des transferts de données.

2.4.4 Adéquation

L'algorithme, représenté par un graphe flot de données, possède un parallélisme potentiel. L'architecture, représentée par un graphe, possède un parallélisme effectif. L'implantation consiste, par transformations de graphes successives, à réduire le parallélisme potentiel au parallélisme effectif. Ces transformations représentent une *distribution* (*allocation spatiale*) et un *ordonnancement* (*allocation temporelle*) des calculs sur les processeurs et des communications sur les liaisons physiques inter-processeurs.

Le terme *Adéquation* sous-entend une mise en correspondance efficace du graphe de l'algorithme, appelé *graphe logiciel* et du graphe de l'architecture, appelé *graphe matériel*.

L'adéquation consiste à choisir parmi toutes les transformations possibles une transformation permettant de respecter les contraintes temps-réel tout en minimisant les composants utilisés. C'est à partir de cette distribution et de cet ordonnancement qu'un exécutif distribué temps-réel, permettant l'exécution de l'algorithme sur l'architecture, peut être généré par le logiciel *SynDEx* qui supporte cette méthodologie (cf.figure 2.3).

Etant donnés un graphe logiciel et un graphe matériel, on comprend facilement qu'il y a un grand nombre de transformations possibles. Le critère utilisé par les heuristiques [16], pour déterminer parmi toutes les transformations la transformation efficace, est lié en premier lieu aux aspects temps-réel, et en particulier à la latence qu'il est primordial de maîtriser dans le cas des systèmes réactifs réalisant de la commande de processus.

L'adéquation consiste à minimiser une fonction de coût, dépendant des temps d'exécution des opérations et des temps de transfert de données entre opérations. La fonction de coût peut faire intervenir d'autres paramètres tels que la mémoire nécessaire pour exécuter une opération ou un transfert de données, la consommation électrique etc. Les durées font partie de la caractérisation du modèle matériel. La caractérisation consiste à associer à chaque composant du graphe matériel l'ensemble des opérations qu'il est capable de réaliser, puis à chaque opération on associe une liste de caractéristiques contenant sa durée d'exécution, la mémoire nécessaire, la consommation etc.

2.4.5 Heuristique

La recherche de la distribution et de l'ordonnancement les plus efficaces est un problème NP-complet, la solution optimale ne peut être obtenue sûrement qu'en énumérant toutes les solutions. Cela prend un temps prohibitif dès que le problème posé n'est plus trivial (concrètement dès que les graphes logiciel et matériel ont plus d'une dizaine de nœuds).

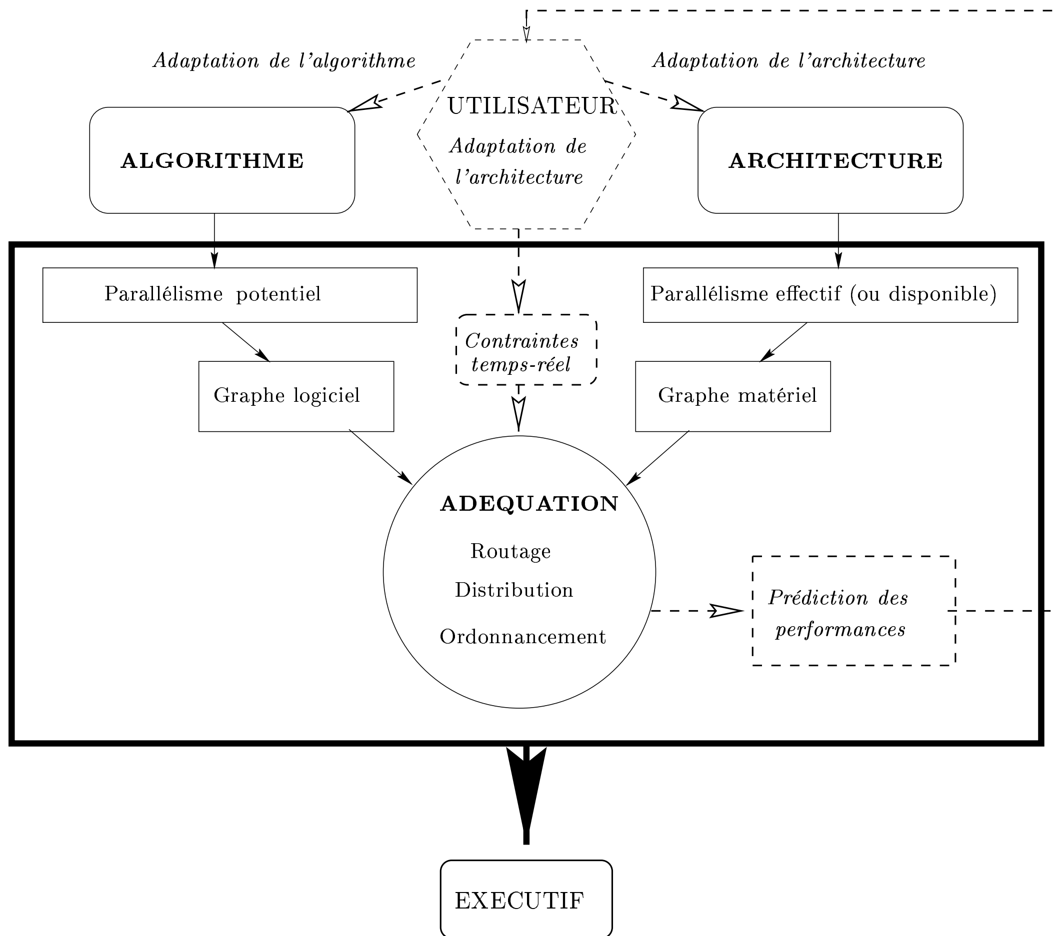


FIG. 2.4 – *La méthodologie AAA*

En d'autre termes, il n'existe pas d'algorithme s'exécutant en un temps polynomial pour trouver la solution optimal. On a généralement recours à des heuristiques qui donnent des solutions approchées, sous-optimales. *SynDEx* possède une heuristique qui propose donc une solution de ce type. La solution proposée par *SynDEx* peut correspondre dans certains cas à la solution optimale mais on ne le saura jamais puisqu'on n'aura pas évalué toutes les solutions.

2.5 Syntaxe *SynDEx*

Les travaux présentés ont été réalisé avec la version 4 de *SynDEx*.

Le graphe flot de données spécifiant l'algorithme d'application est décrit par une séquence d'instructions *SynDEx* de deux sortes :

- instructions de déclaration de sommets (sommets de l'hypergraphe)
- instructions de connexions ou d'execution (arcs orientés de l'hypergraphe)

Chaque instruction *SynDEx* est délimitée par des parenthèses et commence par un mnémonique désignant l'instruction, suivi des paramètres de l'instruction. Dans le cas des instructions de déclaration de sommets, le premier paramètre est un identificateur (chaîne de caractères alphanumérique commençant par une lettre) du sommet déclaré. Tous les identificateurs de sommets doivent être différents. De même, tous les identificateurs de ports d'un même sommet doivent être différents. *SynDEx* distingue les majuscules des minuscules dans les identificateurs. Tout texte apparaissant entre guillemets (double quote) est ignoré (commentaires). Une instruction de connexion ne peut pas précéder les instructions de déclaration des sommets qu'elle connecte. Dans la description syntaxique de chaque instruction, les éléments apparaissant entre <> sont non terminaux et sont décrit par ailleurs.

2.5.1 Instructions pour la spécification de l'algorithme

2.5.1.1 Instructions de déclaration de sommet de calcul

Syntaxe : (function <id> <proc> <duree> <interface>)

- <**id**> est l'identificateur du sommet de calcul.
- <**proc**> est l'identificateur de la procédure de calcul appelée par l'opération du graphe de l'algorithme. La procédure est écrite et compilée séparément par l'utilisateur.
- <**duree**> est la durée d'exécution de la procédure de calcul (entier positif, mettre 10 par défaut) utilisée pour l'optimisation de la distribution et de l'ordonnancement.
- <**interface**> est une liste de déclarations des ports d'entrée-sortie ; un port de sortie peut être connecté (cf instruction *connect*) à un ou plusieurs (diffusion) ports d'entrée d'autres sommets ; un port d'entrée peut soit porter une valeur constante, soit être connecté à une seule sortie. Chaque entrée est précédée du symbole ? et chaque sortie est précédée du symbole !.

L'identificateur <**proc**> ainsi que l'ordre et le type des entrées-sorties de l'interface doivent correspondre à la définition externe de la procédure de calcul. Les opérateurs arithmétiques

et logiques ont des noms réservés. Dans la liste ci-dessous, le mot clé *type* représente un type arithmétique (**integer ou real ou dpreal**).

Scilab	**SynDEx**	Syntaxe **SynDEx**
+	add	(function A add 10 type?e1?e2!s)
−	sub	(function S sub 10 type?e1?e2!s)
−	negate	(function N negate 10 type?e!s)
∗	mul	(function M mul 10 type?e1?e2!s)
/	div	(function D div 10 type?e1?e2!s)
<	less	(function LT less 10 type?e1?e2, logical !s)
>=	notless	(function GE noteless 10 type?e1?e2!s)
=	equal	(function EQ equal 10 type?e1?e2!s)
/ =	noteequal	(function NE notequal 10 type?e1?e2!s)
or	logor	(function OL logor 10 type?e1?e2!s)
and	logand	(function AL logand 10 type?e1?e2!s)
not	lognot	(function NL lognot 10 type?e1?e2!s)

TAB. 2.1 – *Exemple de déclaration de sommet* SynDEx

Note : il n'y a pas d'opérateurs <= ni > car les deux premiers sont obtenus en permuttant les arguments des opérateurs >= et < et le troisième est obtenu en utilisant *logical* pour **type** dans l'opérateur /=.

Dans <**interface**>, chaque déclaration d'entrée-sortie est constituée de :

- un identificateur de type, soit *logical*, soit *integer*, soit *real*, soit *dpreal* (réel double précision), soit un tableau mono - ou multi - dimensionnel composé de ces types primitifs (ex: **[2,6]integer**)
- un caractère **?** pour une entrée ou **!** pour une sortie
- un identificateur nommant l'entrée-sortie ou bien, dans le cas d'une entrée constante, sa valeur entre apostrophes.

Les identificateurs de la liste <**interface**> doivent être tous différents pour un sommet donné. Si plusieurs entrées ou sorties successives sont du même type, celui-ci peut n'être donné qu'une fois. Chaque fois qu'un nouveau type apparaît dans la liste, il doit être précédé d'une virgule. Exemple :

(function gain "calls" mul "dt" 10 "i/o" real ?'0.1'?i !o)

La valeur d'une entrée constante doit être compatible avec son type (ex: **integer ?'1'** ou **real ?'3.14'**). La valeur d'une entrée constante de type tableau est spécifiée séquentiellement, par indice ou par intervalle, avec les mots clé **in** (optionnel), **to** et **step** (optionnel) spécifiant respectivement la borne inférieure, la borne supérieure et l'incrément de l'intervalle. Par exemple, les deux entrées constantes suivantes sont équivalentes :

[2,6]integer ?'[[1]:[{to 6}:0], [2]:[{to 6}:1, {in 2 to 6 step 2}:2]]'

[2,6]integer ?'[[1]:[[1]:0, [2]:0, [3]:0, [4]:0, [5]:0, [6]:0], [2]:[[1]:1, [2]:2, [3]:1, [4]:2, [5]:1, [6]:2]]'

2.5.1.2 Instructions de déclaration de sommet mémoire

Syntaxe : (memory <id> <type>? <entree> <mem>! <sortie> init <init>)

- **<id>** est l'identificateur du sommet mémoire.
- **<type>** est le type de l'entrée.
- **<entree>** est l'identificateur du port d'entrée.
- **<sortie>** est l'identificateur du port de sortie.
- **<init>** est la valeur initiale de la mémoire.
- **<mem>** permet de définir un retard pur, une fenêtre glissante ou une fenêtre glissante retardée. Il définit avec **<type>** les types du port de sortie et de la valeur initiale de la mémoire :
  1. **$1**(retard de 1)
     type sortie=**<type>** et type init =**<type>**
     exemple:**(memory M1 real?i $1 !o init 0.0)**
  2. **$<n>**(retard de n)
     type sortie=**<type>** et type init = **[<n>]<type>**
     exemple:**(memory M2 integer?i $5 !o init [{to 5}:0])**
  3. **window <m>** (fenêtre glissante de m éléments)
     type sortie = **[<m>]<type>**, type init = **[<m>-1]<type>**
     exemple:**(memory M3 real?i window 8 !o init [{to 7}:0.0])**
  4. **$<n> window <m>** (fenêtre glissante de m éléments retardée de n)
     type sortie = **[<m>]<type>**, type init = **[<n>+<m>-1]<type>**
     exemple:**(memory M4 integer?i $5 window 8 !o init [{to 12}:0])**

Pour la syntaxe de **<type>** et de **<init>**, voir l'instruction de déclaration de sommet de calcul.

2.5.1.3 Instructions de connexion des sommets

Les connexions permettent de spécifier soit des transferts de données (flots) entre sommets, soit de spécifier le conditionnement (horloge) des sommets.

L'instruction de connexion entre ports :

Syntaxe : (connect <idSom1>/<idSor> <idSom2>/<idEnt>...)

Le port de sortie (<idSor> du sommet <idSom1>) est connecté au(x) port(s) d'entrée (<idEnt> du sommet <idSom2>).... Tous ces ports doivent avoir été déclarés avec le même type. Exemple :

(hinput H0 getH0 logical !o)
(hadd H1 logical ?i1 ?i2 !o)
(hmul H2 logical ?i1 ?i2 !o)
(connect H0/o H1/i1 H2/i1)

2.5.1.4 Instruction d'exécution inconditionnelle

Syntaxe : (execroots <idSom1> ...)
Les sommets <idSom1> ...) doivent être exécuter inconditionnellement à chaque instant logique.

2.5.1.5 Instruction d'exécution conditionnelle

Syntaxe : (exec <idSom1>/<idSortieLogical> <idSom2> ...)
Les sommets <idSom2> ...) ne sont exécutés que lorsque le port de sortie <idSortieLogical> (de type logical) du sommet <idSom1> porte la valeur logique vraie.

2.5.2 Appel de fonctions pour la génération de code

Le but de la génération de code, est de construire les fichiers exécutables pour chacun des processeurs de l'application décrite sous *SynDEx*. Cela consiste d'abord à générer les fichiers sources (en C par exemple) de chaque processeur du graphe, pour ensuite les compiler avec le compilateur correspondant au processeur.

A partir d'un graphe *SynDEx* appelé par exemple *appli.syn*, mettant en œuvre deux processeurs, nous obtenons, en déclenchant l'opération *SynDEx EXECUTIVE*, les fichiers *appli.m4, appli1.m4 et appli2.m4.*

Ensuite il faut créer le fichier des données d'entrée, par exemple *IN.dat*, ainsi que les fichiers suivants : *appli.UNIX, appli0.UNIX, appliio.c.*

Pour la suite le moyen le plus pratique est de se constituer un fichier *Makefile* qui permet (en tapant la commande "make") d'obtenir les fichiers suivants *appli.make, appli1*, appli1.c, appli2.c,appli1.o, appli2.o, appliio.o* et éventuellement le fichier de sortie *OUT.dat*.

2.5.3 Fonctions existantes

Un certain nombre de fonctions très utilisées sont définies dans le fichier Cexec, à l'instar des palettes prédéfinies dans *Scicos*. Il est donc très peu judicieux (notamment pour la gestion de l'initialisation et des durées d'exécution) de les redéfinir dans un schéma quelconque. Le tableau 2.2 récapitule l'ensemble de ces fonctions, avec des variables de type logical,integer,real ou dpreal.

Définition	**Fonction**
Fenêtre glissante **Retard**	(memory MEM real?i window9 !o init[to8:0.0]) (memory R2 real?i $1 !o init[to9:0.0])
Sous-échantillonage **Sur-échantillonage**	hmul hadd
Opération **arithmétique** **et logique**	NON,ET,OU,OPPOSEE,>=,< ADD(+),SUB(),MULT(*),DIV(/) DECREMENT 1(-1 itération))
Opération **sur les tableaux**	addition, soustraction multiplication
Traitement **du signal**	produit scalaire (realDotProduct), filtre IIR filtre adaptatif (realEqualizer)

TAB. 2.2 – *Récapitulatif des fonctions existantes dans Cexec*

Chapitre 3

Interface Scicos-SynDEx

3.1 Les principes de l'interface

Contrairement aux langages de programmation *C* et *Fortran*, *Scicos* et *SynDEx*, étant basés sur le langage *SIGNAL* [2], intègrent la gestion du temps et permettent d'exprimer la notion de parallélisme.

3.1.1 Les signaux

Dans *Scicos* il existe deux types de signaux *les signaux réguliers* et *les signaux d'activations* (cf. chap. 1.1.1, page 1). Les valeurs des signaux réguliers (flot de données) sont toujours présentes, contrairement aux signaux d'activations.

Dans *SynDEx* tous les signaux sont de type flot de données et sont appelés *événements* ordonnés. Historiquement, le choix de cette uniformisation a été décidé pour éviter d'éventuelles confusions entre les signaux de conditionnement et les signaux combinatoires. La figure 3.1 (page 50) montre un diagramme de la version 4 de *SynDEx*.

Type de signal	**Événement *SynDEx***
signal régulier	booléen, réel double précision
signal d'activation	booléen

TAB. 3.1 – *Type d'entrées/sorties.*

Dans *SynDEx* un *sommet* est l'équivalent à un bloc dans *Scicos*, à la différence que l'absence de valeurs en entrée d'un *sommet* implique l'absence de valeurs en sortie. De même qu'un sommet non activé ne présente aucune valeur en sortie, malgré la présence d'une valeur sur son entrée.

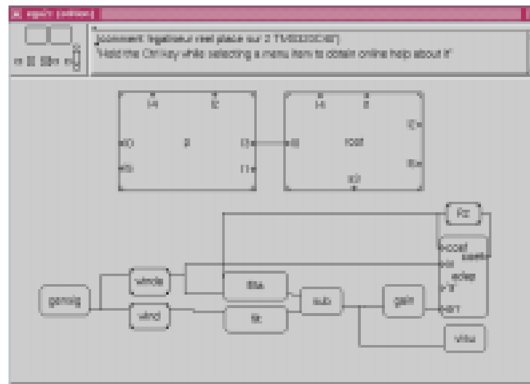


FIG. 3.1 – *La représentation d'un diagramme dans l'éditeur* SynDEx.

Un signal régulier *Scicos* peut correspondre à des événements de deux natures dans *SynDEx*: `booléen` ou `réel double précision`. En revanche, un signal d'activation *Scicos* ne peut être traduit que par un événement booléen dans *SynDEx*.

3.1.2 Le conditionnement

Dans *SynDEx* l'exécution d'une opération est, soit non conditionnée et elle s'exécute à l'horloge la plus fine (*execroots*), soit conditionnée et elle s'exécute à l'horloge qui est un sous-échantillonnage (*exec*) de l'horloge la plus fine. L'horloge la plus fine est égale à l'union de toutes les horloges.

Une opération consomme et produit des signaux. Un signal est une suite d'événements valués à laquelle est associée une horloge indiquant la présence ou l'absence de ces valeurs.

3.1.3 Les différences terminologiques

Le tableau 3.2 (page 51) indique les différences principales de terminologie entre *Scicos* et *SynDEx*.

3.1.4 Les blocs

De par leur concept, les types de blocs *Scicos* (cf. chap. 1.1.3, page 4) n'ont pas tous d'équivalent dans *SynDEx* et ne sont donc pas tous traduisibles. En effet, la notion de temps

Scicos	Syndex
signal régulier (valeurs rémanentes)	événements (valeurs fugitives)
signal d'activation (valeurs fugitives)	événements de conditionnement (valeurs fugitives)
lien	arc orienté
bloc	nœud ou sommet

TAB. 3.2 – *Les principales différences de terminologies* Scicos/SynDEx.

continu n'existant pas dans le logiciel *SynDEx*, les types de blocs suivants ne peuvent être traduit :

- les blocs avec état continu,
- les blocs de type "z".

De même que la notion de visualisation des signaux n'existant pas dans le logiciel *SynDEx*, les blocs *Scicos* de visualisation, à l'instar du bloc `Scope` sont à exclure.

Enfin le concept mono-horloge dans *SynDEx*, interdit l'existence de source d'activations telle que le bloc *Scicos* `CLOCK`.

3.2 Les règles de traduction

Dans ce chapitre nous définissons les modifications systématiques à apporter sur les types de blocs et leurs ports d'entrées et sorties. Ces modifications sont opérées dans une première phase de compilation de l'algorithme obtenu dans *Scicos*.

3.2.1 Test sur les blocs

Une série de test sur les type de blocs doit être effectuée afin de :

1. détecter et interdire l'utilisation des blocs *Scicos* suivant :
   - les blocs avec état continu,
   - les blocs de type "z",
   - les blocs de sortie de visualisation (exemple `Scope` ou `Mscope`).
2. vérifier l'existence d'une source unique d'activations : le bloc `Evtdly`, de le supprimer ainsi que les ports d'activation d'entrée sur les blocs auxquels il était relié. L'existence de plus d'une source d'activation génère une erreur.
3. détecter les blocs générateurs de valeur constante avant de les supprimer, puis d'ajouter une entrée constante au(x) bloc(s) au(x)quels il sont reliés. En effet dans *SynDEx* la notion d'événement généré de manière permanente n'existant pas, il se pose un problème concernant la traduction des blocs *Scicos* générant une valeur constante, d'où cette solution.

3.2.2 Test sur les paramètres des blocs

Les paramètres (conditions initiales ou autres), associés à chaque bloc *Scicos*, doivent être systématiquement transformés en port d'entrée du sommet *SynDEx* correspondant.

3.2.3 Déclarations de compilation

Dans *SynDEx*, les ports d'entrée et de sortie d'un même sommet ne doivent pas avoir le même *nom*, dans *Scicos*, les ports d'entrée et de sortie d'un même sommet ne doivent pas avoir le même *numéro*. Des modifications sur le repérage des ports d'entrées et de sorties s'impose :

3.2.3.1 Identification des entrées et sorties

L'identification des entrées et sorties pendant la phase de compilation consiste à rajouter un préfixe devant chaque numéro de port d'entrée et de sortie (régulier et d'activation) (cf. tableau 3.3, page 52).

Signal	**préfixé de**
d'entrée régulière	"e"
de sortie régulière	"s"
d'entrée d'événements	"iclk"
de sortie d'événements	"oclk"

TAB. 3.3 – *Préfixe de l'identificateur des variables.*

3.2.3.2 Liberté d'ordonnancement pour affiner le parallélisme

Dans certains cas de contraintes temps-réel restrictives, l'exploitation du parallélisme peut nécessiter la transformation d'un sommet en plusieurs autres sommets, de manière à permettre au logiciel *SynDEx* une plus grande liberté d'ordonnancement. Ces modifications peuvent concerner de manière systématique des blocs avec un état comme par exemple le bloc *DSSLTI*, qui implémente les équations d'états :

$$Z_{(k+1)} = A.Z_{(k)} + B.u_{(k)} \tag{3.1}$$

$$y_{(k+1)} = C.Z_{(k+1)} + D.u_{(k+1)} \tag{3.2}$$

Z est le vecteur de l'état discret, u est le vecteur de l'entrée régulière, y est le vecteur de la sortie régulière, A, B, C, D sont des matrices et z est l'opérateur retard. En opérant le changement de variable suivant :

$$Z_{(k)} = zZ_{(k-1)} \tag{3.3}$$

L'équation 3.2 étant valable à tout instant, nous obtenons les relations suivantes :

$$zZ_{(k)} = A.Z_{(k)} + B.u_{(k)} \tag{3.4}$$

$$y_{(k)} = C.Z_{(k)} + D.u_{(k)} \tag{3.5}$$

Pour une correspondance systématique avec les sommets de *SynDEx*, il faut donc envisager trois cas de figure :

1. *Sans état*, il s'agit d'un bloc effectuant des opérations purement combinatoires, pour lequel il n'y a aucune transformation à faire. Les matrices A, B et C sont nulles, le bloc devient un sommet *SynDEx* sans aucune modification.
2. *Sans combinatoire*, il s'agit d'un bloc ayant un état pur, pour lequel il n'y a aucune transformation à faire. La matrice C est diagonale et $D = 1$. Ce bloc est :
   - soit un retard pur, si tous les éléments de C sont nuls (sauf le dernier valant 1)
   - soit une fenêtre glissante, si tous les éléments de la diagonale de C valent 1.
3. *Avec état et combinatoire*, dans ce cas il est possible de prévoir la décomposition du bloc en trois sommets distincts représentant :
   - les matrices A et B.
   - les matrices C et D.
   - le retard $.

Cette décomposition en 3 sommets est ainsi représentée sur la figure 3.2 (page 54).

3.2.3.3 Identification des blocs

Après avoir effectué les modifications qui s'imposent sur certains blocs *Scicos* à l'instar du bloc "DSSLTI", leur traduction dans *SynDEx* impose de modifier leur identification.

L'identification des blocs consiste à rajouter pour chaque sommet, pendant la phase de compilation, un préfixe devant chaque numéro de bloc (cf. tableau 3.4, page 54).

3.2.3.4 La fonction d'exécution inconditionnelle : "execroots"

Cette fonction définit les sommets qui s'exécutent de manière inconditionnelle, sa syntaxe est la suivante :

(execroots sommet_i sommet_j sommet_n)

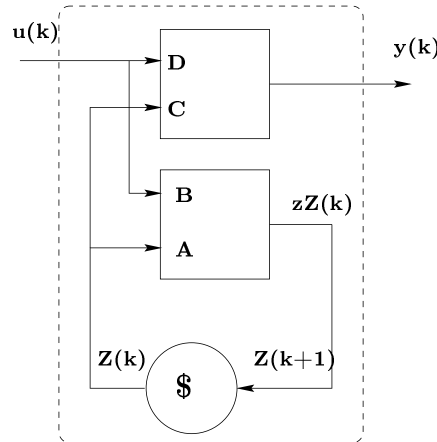


FIG. 3.2 – *Bloc avec état discret.*

type de blocs	nom de sommet, suivi du numéro du bloc dans *Scicos*
standard (sans état)	celui de la procédure dans *Scicos*
standard (avec état)	3 sommets : $, matAB, matCD
synchro	“Hinput”

TAB. 3.4 – *Correspondance entre le type de bloc et le nom du sommet.*

3.2.3.5 La fonction d'exécution conditionnelle : “exec”

Les relations de priorité d'exécution des opérations sur les sommets sont établies par la fonction “exec”, définissant ainsi le conditionnement. La syntaxe de cette fonction indique que la sortie du premier sommet conditionne les suivants :

(exec sommet_condition/sortie_x sommet_k sommet_l)

3.3 Un exemple de comparaison :

Le schéma de la figure 3.3 (page 55) est celui fourni à titre démonstratif dans l'environnement *SynDEx* : *ega2c*. Il modélise un filtre adaptatif dont l'algorithme est basé sur la méthode du gradient stochastique de pas μ. Les sommets *winda* et *win* sont des fenêtres glissantes sur le signal d'entrée $X(t)$: $V_{X(t)} = (X(t), X(t-1), \cdots, x(t-N))$, avec l'indice temporel t, l'ordre du filtre N. Le sommet *filta* réalise le calcul du filtre, avec l'indice vectoriel i :

$$Y(t) = \sum_{i=1}^{N} (H_i(t-i) \cdot V_{X_i(t)})$$

La mise à jour du coefficient H (gradient stochastique de pas μ) est effectuée par le sommet *adap* : $H_i(t) = H_i(t-1) + \mu \cdot E(t) \cdot V_{X_i}(t)$, avec le calcul de l'erreur exécuté par le sommet *sub* : $E(t) = Y_{ref} - Y(t)$. Dans cet exemple l'ordre du filtre est 9, nous l'avons ainsi reproduit et simulé dans l'environnement *Scicos*, afin de faire apparaître les principales différences fonctionnelles entre les deux concepts (cf. fig. 3.4, page 55). Pour remplacer la fonction d'exécution inconditionnelle "execroots", nous utilisons le bloc "Clock".

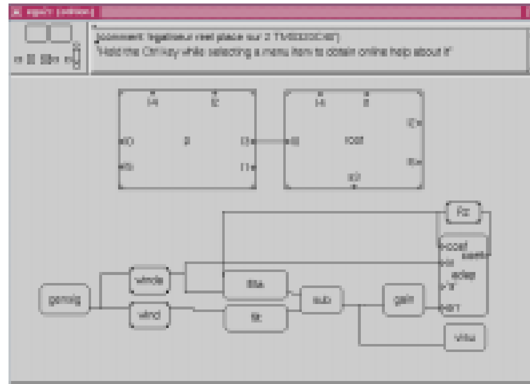


FIG. 3.3 – *L'égaliseur dans* SynDEx.

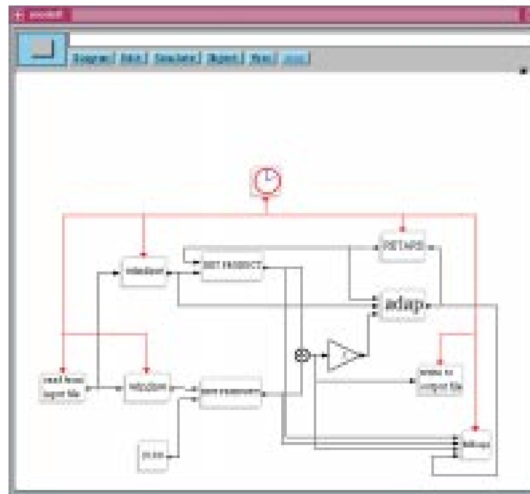


FIG. 3.4 – *L'égaliseur dans* Scicos.

3.3.1 Les modifications dans *Scicos*

Pour la circonstance, nous avons créé de nouveaux blocs : “Windows”, “Dotproduct”, “Retard” et “Adap”. Par rapport à la figure 3.4, nous avions modifié dans un premier temps bloc, le bloc “sinus”, de base dans *Scicos*, pour le rendre discret.

Notons au passage la possibilité de visualiser, avec *Scicos*, l'évolution des variables en fonction du temps (en abscisse). On observe ainsi sur la figure 3.5 (page 56) l'évolution des signaux en amont et en aval du bloc “Sommateur”.

On peut constater la stabilité du système, ainsi que l'évolution des 9 éléments du vecteur en entrée du bloc “RETARD”.

3.3.2 Les modifications pour *SynDEx*

Les modifications à apporter au schéma *Scicos* concernent le renommage des blocs identiques, la transformation des blocs WINDOW et RETARD qui deviennent des sommets “memory”, ainsi que le bloc de générateur de valeur constante qui doit être supprimé pour rajouter sa valeur sous la forme d'un port d'entrée supplémentaire du bloc DOTPRODUCT.

Afin de vérifier l'équivalence des schémas *Scicos* et *SynDEx*, nous comparons les résultats obtenus dans les fichiers de sortie. Pour cela nous remplaçons le bloc “sinus” par un bloc “readf” (lecture de fichier) qui contient le même fichier de valeurs aléatoires “/syndex/ega/gensig.dat” que le sommet “gensin” (lecture de fichier) de *SynDEx*.

Pour la sortie, on supprime le bloc “MScope” et on remplace le bloc “WRITE to OUTPUT FILE” par un fichier quelconque afin de le comparer à “/syndex/ega/visu.dat”.

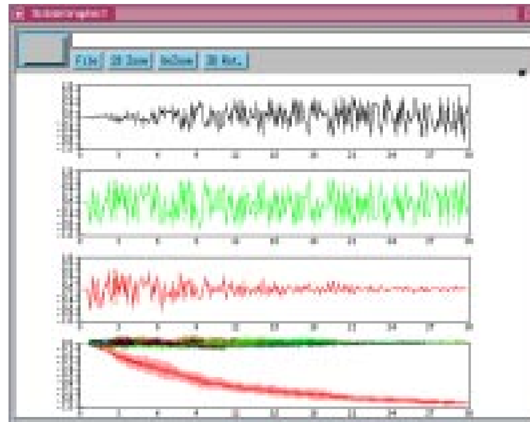


FIG. 3.5 – *Entrées et sortie du soustracteur.*

3.3.3 Les vérifications

A partir du fichier d'entrée "gensin.dat", nous disposons des valeurs de sortie dans le fichier "visu.dat". Nous proposons les deux comparaisons suivantes :

- le schéma identique est reproduit dans *Scicos* (cf. Fig. 3.4, page 55) et, à partir du même fichier d'entrée "gensin.dat", nous vérifions que nous obtenons, par la simulation, dans un fichier de sortie, strictement les mêmes valeurs que dans le fichier "visu.dat".
- A partir de ce même schéma *Scicos*, nous utilisons l'interface *Scicos/SynDEx* et nous obtenons à nouveau dans *SynDEx* un graphe de l'égaliseur, qui donne les mêmes valeurs que le fichier "visu.dat".

3.4 La procédure de transformations *Scicos-SynDEx*

Cette opération doit s'effectuer de manière systématique par le compilateur *Scicos*. Pour ce faire nous décrivons la liste de tests à effectuer sur les blocs :

3.4.1 Phase d'initialisation : fonction scv40

Cette fonction permet d'établir des listes et de mettre en forme différentes matrices concernant la nature des blocs *Scicos* et leurs inter-connexions dans le schéma-blocs (cf. la figure 3.6, page 58).

Le résultat obtenu est placé dans un fichier avec l'extension ".syn". Prenons un exemple quelconque avec un fichier "foo.syn", pour lequel sont d'abord déclarées les fonctions des sommets, puis leurs connexions ; le résultat est de la forme suivante :

La fonction *scv40* permet aussi de donner le nom de l'application traitée : exemple *appli*, ainsi que d'opérer différents test sur le schéma-blocs, notamment la présence de plus d'une horloge ou le type des blocs de sorties (visualisation proscrite).

Après la phase d'initialisation, la fonction *fonction scv40* fait appel aux fonctions Scilab suivantes (détaillées par la suite) :

- scv41 : pour la transformation des blocs en sommets ;
- scv42 : pour les connexions (régulières) des sommets ;
- scv43 : pour la définition de l'exécution conditionnelle (*exec*) et inconditionnelle (*execroot*) ;
- scv44 : pour la création des fichiers *appli.m4h*, *appli.m4x*, *Makefile*

Le résultat obtenu est placé dans un fichier avec l'extension ".syn". Prenons un exemple simple avec un fichier "foo.syn", dans lequel sont d'abord déclarées les fonctions des sommets, leurs connexions puis les mode d'exécution ; le résultat est de la forme suivante :

```
(memory dsslti1M dpreal ?zkp1 $1 !zk init 0)
(function dsslti1AB "calls" matAB "dt" 10 "i/o" dpreal ?e1, dpreal ?!zk)
```

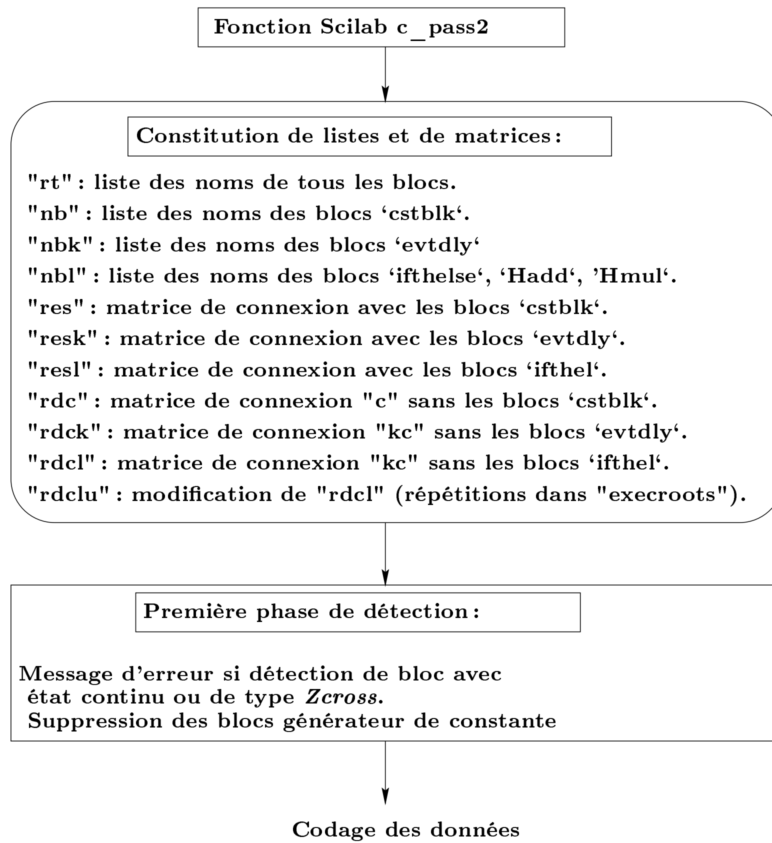


FIG. 3.6 – *Mise en forme des données* Scicos *: procédure Scilab scv40.*

```
(function dsslti1CD "calls" matCD "dt" 10 "i/o" dpreal ?e1, dpreal ?zk, dpreal!
s1)
(function ifthel2 "calls" ifthel "dt" 10 "i/o"  dpreal ?'1',logical !oclk1)
(function capteur1 "calls" capteur1 "dt" 10 "i/o" dpreal!s1)
(function actionneur1 "calls" actionneur1 "dt" 10 "i/o" dpreal ?e1)

(connect dsslti1CD/s1  actionneur1/e1)
(connect dsslti1AB/zk dsslti1M/zkp1)
(connect dsslti1M/zk dsslti1AB/zk dsslti1CD/zk)
(connect capteur1/s1  dsslti1AB/e1 dsslti1CD/e1)

(exec ifthel2/oclk1 dsslti1AB  dsslti1CD  dsslti1M)
(execroots  ifthel2 capteur1)
```

On reconnait que dans les tois premières lignes qu'il s'agit d'un bloc *Scicos* avec état, transformé en trois sommets *SynDEx*. Les sommets `capteur` et `actionneur` sont respectivement l'entrée et la sortie du système.

L'exécution des 3 sommets `dsslti` est conditionné avec la commande "exec" par la sortie "oclk1" du sommet `ifthel` (bloc Synchro).

L'exécution inconditionnelle concerne les sommets `ifthel` et `capteur`.

3.4.2 Déclaration des fonctions : *fonction scv41*

Cette fonction permet la transformation des blocs (sans état) *Scicos* en sommets *SynDEx*, de même que les blocs dynamiques discrets avec état discret (*DSSLTI* et *Num(z)/Den(z)*) en trois sommets ("matAB,matCD et $"). La figure 3.7 (page 60) détaille l'organigramme de transformation exécuté pour chaque bloc.

3.4.3 Constitution des connexions : *fonction scv42*

Cette fonction gère la déclaration des connexions existantes, et en crée d'autres dans le cas des trois sommets : matrice AB, matrice CD et retard $ (cf. fig. 3.2, page 54). La figure 3.8 (page 61) détaille l'organigramme de déclaration des inter-connexions dans *SynDEx*.

3.4.4 Constitution des exécutions : *fonction scv43*

Cette fonction gère la déclaration des exécutions conditionnelles *exec* et de l'exécution inconditionnelle *execroot*, ainsi que la création du fichier *appli.syn* qui caractérise la représentation graphique du schéma *SynDEx*. La figure 3.9 (page 62) détaille l'organigramme de déclaration des exécution conditionnelle (*"exec"*) et de l'exécution inconditionnelle (*"execroots"*) dans *SynDEx*.

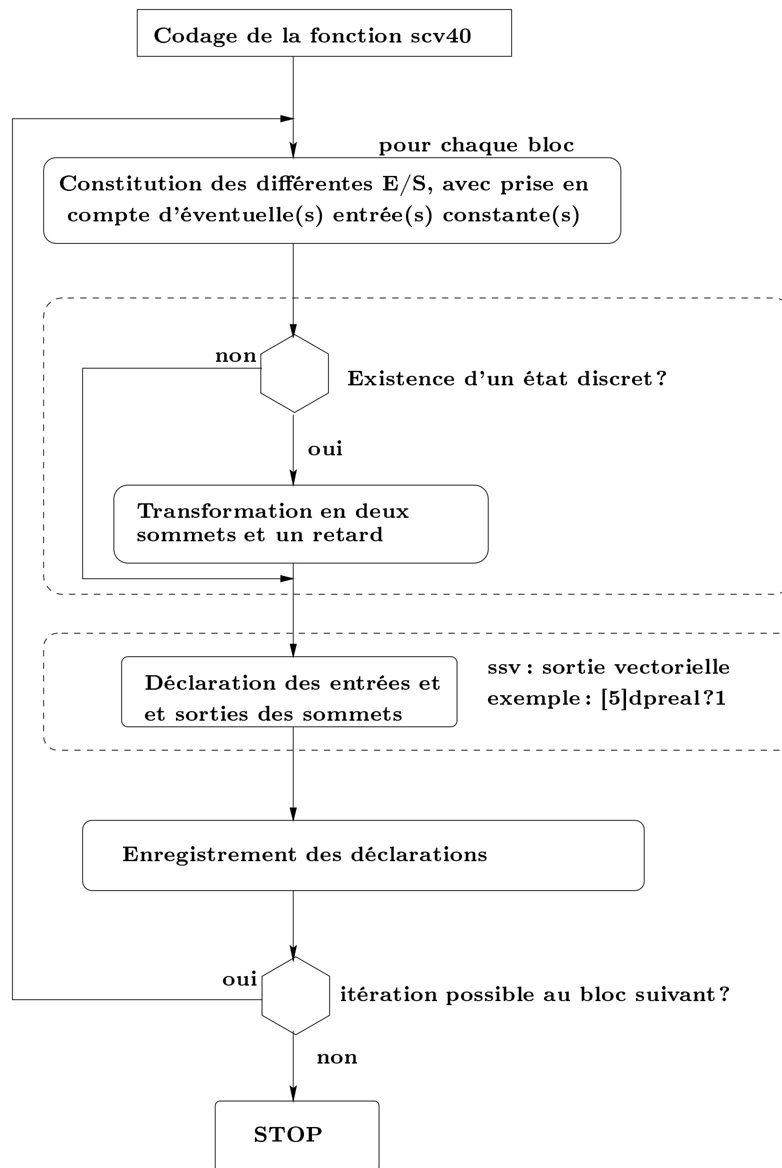


FIG. 3.7 – *Déclaration des fonctions* SynDEx *: fonction scv41.*

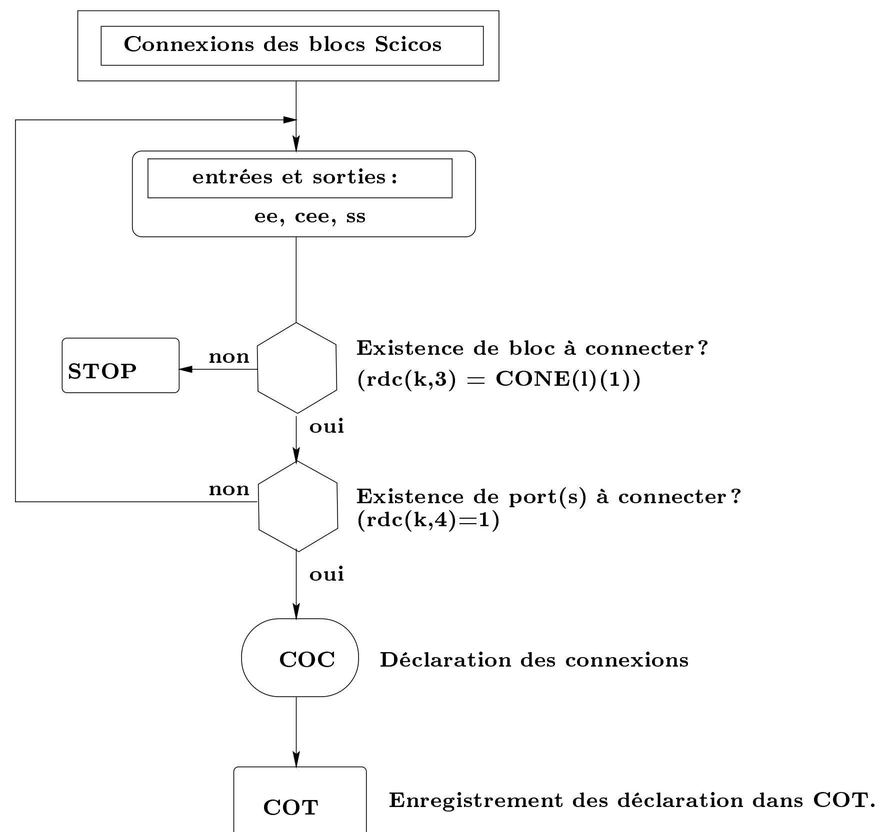


FIG. 3.8 – *Déclaration des inter-connexions en* SynDEx *: fonction scv42.*

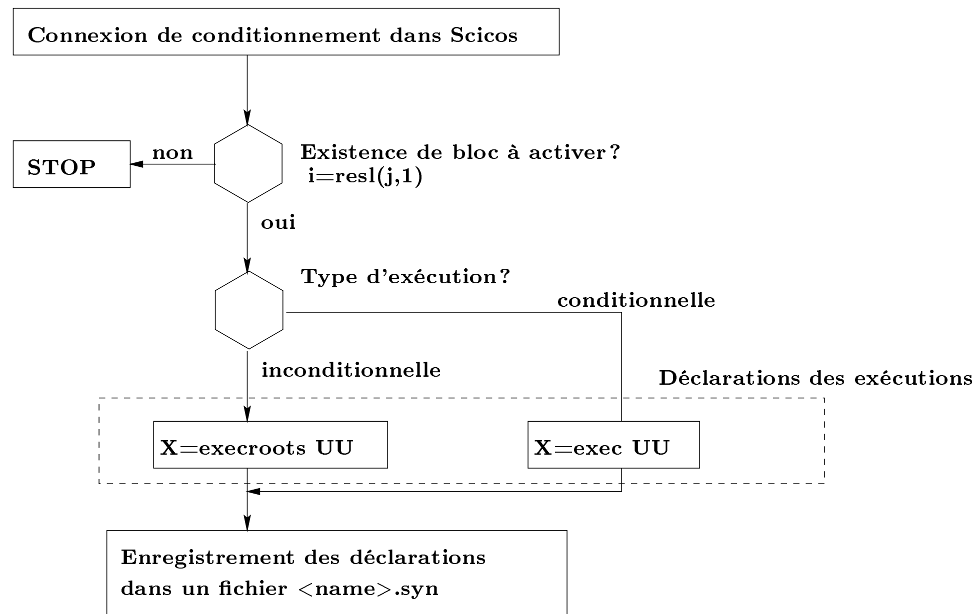


FIG. 3.9 – *Déclaration des exécution en* SynDEx *: fonction scv43.*

3.4.5 Création des fichiers : *fonction scv44*

Cette fonction finalise la constitution des 3 fichiers suivants :

3.4.5.1 Le fichier *appli.m4h*

Ce fichier contient les informations de paramétrage et d'adressage des différentes variables et procédures mises en œuvre dans le graphe *SynDEx*.

3.4.5.2 Le fichier *appli.m4x*

Ce fichier est composé des adresses (chemin d'accès) des différentes librairies contenant les procédures utilisées dans le graphe *SynDEx*.

3.4.5.3 Le fichier *Makefile*

Ce fichier contient les commandes *m4* pour générer tous les fichiers sources (compilés en C ou autre) pour l'application considérée. Dans *Scicos* l'exécution des procédures de simulation Fortran se fait directement. En revanche, dans *SynDEx*, il faut utiliser des macros d'exécutions générées par le processeur de langage macro *GNU M4* (cf. figure 3.10, page 63).

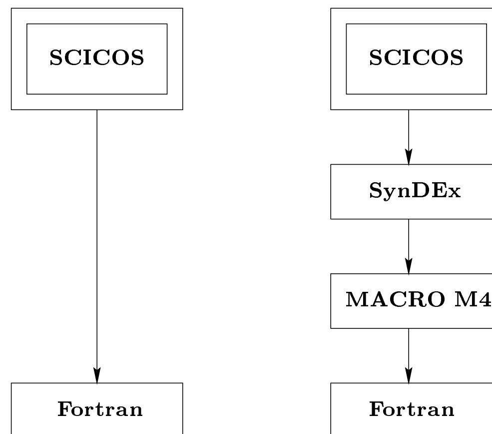


FIG. 3.10 – *Étapes d'appel des sous programmes.*

3.5 Le mode d'emploi

3.5.1 Interface SynDEx-Scicos

L'installation et l'utilisation de la boite à outils pour interfacer *Scicos* avec *SynDEx* à partir d'un super-bloc discret *Scicos*, est exactement semblable à celle de la boîte à outils *Génération de Code*. Toutes les informations nécessaires sont aussi disponibles sur le site :

http://www-rocq.inria.fr/scilab/contributions.html

On obtient ainsi dans la barre de menu un nouveau bouton *SynDEx* qui donne l'accès une unique sélection : *Interface*. D'un simple clic sur cette dernière, suivi d'un autre clic pour désigner le super-bloc concerné et le tour est joué.

En prenant pour exemple l'application du nom de *foo*, 4 fichiers sont ainsi créés automatiquement :

- foo. syn (cf. section. 3.4.4, page 59),
- foo.m4h (cf. section. 3.4.5, page 63),
- foo.m4x (cf. section. 3.4.5),
- Makefile (cf. section. 3.4.5).

L'utilisateur aura le soin d'adapter le fichier Makefile au type de code à générer (C, fortran, ou autre).

3.5.2 La génération de code dans SynDEx

La procédure de génération de code pour un graphe *SynDEx* consiste au lancement de l'opération `Execute`, dans le menu `Edit`. Ce qui permet d'obtenir la création de deux fichiers $< name > .m4$ et $< name > 1.m4$. Dans l'exemple de la figure 3.5 (page 56), nous avons *ega2c.m4* et *ega2c1.m4*. Ces deux fichiers vont servir par la suite, à l'aide du macro processeur GNU m4, à la création des fichiers suivants :

- *ega2c.m4h, ega2c.m4x, ega2c.make, Makefile et ega2cio.c.*
  Ce dernier fichier concerne les déclarations d'entrées/sorties.
- Si le graphe contient un sommet faisant appel à un fichier d'entrée (nommé par exemple "IN"), il faut alors créer un fichier *IN.dat*, contenant les valeurs d'entrée.

A l'aide du fichier *Makefile* (avec la commande : make -f ega2c.make), nous obtenons le code de l'exécutif sous la forme des fichers suivant : *ega2c1*, ega2c1.c, ega2c1.o, ega2cio.o.* L'exécution de l'application de notre exemple, permet de créer et d'écrire le résultat des valeurs de sortie dans le fichier *OUT.dat*.

Conclusion

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

- *Scicos*, pour la modélisation des systèmes dynamiques hybrides. Nous nous sommes particulièrement intéressé à la définition des types de signaux et à la description du fonctionnement des blocs. Nous avons donné un aperçu du traitement de la compilation de schéma-blocs et du processus de simulation.
- et *SynDEx*, d'aide à l'implantation temps réel multi-processeur d'algorithmes réactifs.

Dans un second temps nous avons proposé une nouvelle contribution permettant l'interfaçage du logiciel *Scicos* avec le logiciel *SynDEx*, ce qui permet de transformer directement dans le format du logiciel *SynDEx* un sous-ensemble discret (souvent de type régulation-contrôle) du système hybride complet specifié avec *Scicos* qui devient alors un algorithme à implanter avec *SynDEx* sur une architecture multiprocesseurs en respectant des contraintes temps réel .

Perspectives

Il est envisagé de poursuivre et de renforcer les fonctionnalités de l'interface entre les environnements logiciels : *Scicos* et *SynDEx*, afin de prendre en compte les aspects liés au conditionnement. Pour cela, on devra, d'un côté, définir une sémantique commune aux deux environnements afin de supprimer la rupture entre le niveau modélisation/simulation des automaticiens et le niveau implantation temps réel des informaticiens, en vue de permettre un bon niveau de traçabilité quand il s'agit de retrouver au niveau de la modélisation ce qui a causé une erreur, constatée au niveau de l'implantation. D'un autre côté on pourra faire remonter des informations, obtenues en étudiant le comportement temps réel simulé (architecture opérationnelle), au niveau de la simulation/modélisation (architecture fonctionnelle).

Cette réalisation, la première dans son genre permettra d'élaborer un environnement logiciel pour la modélisation/simulation/implantation, en utilisant des environnements standards largement répandus et en les adaptant, afin de résoudre au mieux les problèmes liés aux transferts d'informations entre l'automatique et l'informatique temps réel.

Bibliographie

[1] C. AIGLON, C. LAVARENNE, Y. SOREL, A. VICARD *Utilisation de SynDEx pour le traitement d'images temps-réel* Rapport de recherche 2968, Septembre 1996, INRIA, Rocquencourt, France

[2] A. BENVENISTE, G. BERRY *The synchronous approach to reactive and real-time systems.* Proceedings of the IEEE, 79(9):1270-1282, Sep. (1991)

[3] A. BENVENISTE *Compositional and Uniform Modeling of Hybrid Systems*, in IEEE Trans. Automat. Control, Vol. 43 (Apr. 1998): 579-584.

[4] L. BESNARD *Compilation de SIGNAL : horloges, dépendances, environnement* PhD thesis, Université de Rennes 1, 1992.

[5] P. BOURNAI, C. LAVARENNE, P. LE GUERNIC, O. MAFFEÏS, Y. SOREL *Interface SIGNAL-SynDEx* Rapport de recherche 2206, Mars 1994, INRIA, Rocquencourt, France

[6] C. BUNKS, J.P. CHANCELIER, F. DELEBECQUE, C. GOMEZ (EDITOR), M. GOURSAT, R. NIKOUKHAH, S. STEER, *Engineering and scientific computing with Scilab*, Birkhauser, 1999.

[7] R. DJENIDI, C. LAVARENNE, R. NIKOUKHAH, Y. SOREL, S. STEER *From Hybrid System Simulation to Real-Time Implementation*, in ESS'99. SCS, Erlangen, Germany, Oct. 1999.

[8] R. DJENIDI, R. NIKOUKHAH, S. STEER *A propos du formalisme Scicos*, 3e Conference francophone MOSIM'01 Conception, analyse et gestion des systemes industriels. SCS, Troyes, France, Avril 2001.

[9] R. DJENIDI *Formalisme de modélisation des systèmes dynamiques hybrides.* Thèse, 2001, Paris-12, France

[10] R. DJENIDI, R. NIKOUKHAH, S. STEER *Code generation in Scicos*, in ESM'2001. SCS, Prague, Czech Republic, June 2001.

[11] R. DJENIDI, R. NIKOUKHAH, S. STEER *A tool for verification in Scicos* (soumis) in ESS'2001. SCS, Marseille, France, Oct. 2001.

[12] M. GONDRAN, M. MINOUX. *Graphes et Algorithmes.* Eyrolles 1979.

[13] T. GRANDPIERRE *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* Thèse, 2000, Paris Sud Orsay, France

[14] C. Lavarenne, R. Reynaud, Y. Sorel *Spécification et validation à l'aide d'un langage synchrone d'un protocole d'appariement de données asynchrones.* Quatorzième Colloque GRESTI, Juan-Les-Pins, Septembre 1993.

[15] C. Lavarenne & Y. Sorel *SynDEx : Un environment de programmation pour multi-processeur de traitement du signal* Manuel de l'utilisateur version vo, INRIA-Rocquencourt France, Novembre 1989.

[16] C. Lavarenne, Y. Sorel *Performance Optimisation of Multiprocessor Real-Time Applications by Graph Transformations.* Parallel Computing 93, Grenoble, Septembre 1993.

[17] R. Nikoukhah, S. Steer *Conditioning in hybrid system formalism*, in ADPM, Dortmund, Germany, Sept. 2000.

[18] R. Nikoukhah, S. Steer, *Scicos: a hybrid system formalism* in ESS'99, Erlangen, Germany, Oct. 1999.

[19] R. Nikoukhah, S. Steer, *Hybrid system modelling and simulation*, Femsys'99, Munich, Germany, March 1999

[20] R. Nikoukhah, S. Steer, *Scicos a dynamic system builder and simulator*, in IEEE International Conference on CACSD, Dearborn, Michigan, 1996.

[21] R. Nikoukhah, S. Steer, *Hybrid systems: modeling and simulation*, COSY : Mathematical Modelling of Complex System, Lund, Sweden, Sept. 1996.

[22] R. Nikoukhah, S. Steer, *SCICOS : A Dynamic System Builder and Simulator* User's Guide - Version 1.0, Rocquencourt France, Juin 1997.

[23] Y. Sorel *Massively Parallel Computing Systemes with Real Time Constraints. The "Algorithm Architecture Adequation" Methodology.* Proc. Massively Parallel Computing Systems, the Challenges of General-Purpose and Special-Purpose Computing-Ischia Italy, May 1994.

[24] S. Steer, H. Jreij, R. Nikoukhah, *Scilab/Scicos : une application à la régulation des aménagements hydrauliques fluviaux*, 2AO 96, Paris, France, Nov. 1996.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399