Collection

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Interface SIGNAL-SynDEx

Patricia BOURNAI - Christophe LAVARENNE
Paul LE GUERNIC - Olivier MAFFEÏS - Yves SOREL

N° 2206
Mars 1994

PROGRAMME 2

Calcul symbolique, programmation et génie logiciel

PROGRAMME 5

Traitement du signal, automatique et productique

INRIA
ROCQUENCOURT

Interface SIGNAL-SynDEx

Patricia BOURNAI*
Christophe LAVARENNE**
Paul LE GUERNIC*
Olivier MAFFEÏS*
Yves SOREL**

Programmes 2 et 5 — Calcul symbolique, programmation et génie logiciel — Traitement du signal, automatique et productique
Projets EPATR et SOSSO

Rapport de recherche n° 2206 — Mars 1994 — 49 pages

Résumé : Le langage synchrone SIGNAL permet de spécifier et de vérifier en terme d'ordre d'événements, un algorithme de traitement du signal et d'image ou de commande de processus. L'environnement logiciel SynDEx permet de spécifier d'une part un algorithme de ce type et d'autre part une architecture multi-processeur, puis de réaliser une implantation optimisée sous contraintes remps réel. Il était donc naturel d'interfacer ces deux logiciels. Pour cela, nous avons défini les règles permettant de traduire un programme SIGNAL compilé, afin de l'utiliser comme spécification d'algorithme dans SynDEx. L'implantation réalisée à partir de cet algorithme sous SynDEx conserve les propriétés vérifiées en SIGNAL.

Mots-clé : langages synchrones, SIGNAL, SynDEx, temps réel, implantation, multi-processeur, traitement du signal, traitement d'image, commande de processus

(Abstract: pto)

*. Projet EPATR, INRIA Rennes IRISA
**. Projet SOSSO, INRIA Rocquencourt

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

SIGNAL-SynDEx Interface

Abstract: The SIGNAL language allows the specification and the verification in terms of ordering of events, of a signal and image processing or a process control algorithm. The SynDEx software allows on one hand the specification of an algorithm of this type and on the other hand the specification of a multi-processor architecture, then it is possible to perform an optimized implementation under real-time constraints. It was natural to interface both softwares. In order to do this, we defined rules such as, a compiled SIGNAL program can be translated in an algorithm specification usable by SynDEx. The derived implementation performed with SynDEx preserves the properties proved whith SIGNAL.

Key-words: synchronous languages, SIGNAL, SynDEx, real-time, implementation, multi-processor, signal processing, image processing, process control

Table des matières

1 Introduction

L'étude que nous présentons ici s'inscrit dans le cadre général de la conception sûre de systèmes temps réel devant fonctionner sur des architectures souvent distribuées. 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, dans lequel calculs et communications internes sont supposés de durée nulle. Seuls les événements de communication du programme avec son environnement sont significatifs dans la détermination de l'écoulement du temps. Dans ce cadre, le langage **SIGNAL**, développé à l'IRISA, permet de spécifier un algorithme décrivant un système temps réel et d'en étudier ses propriétés temporelles.

Actuellement, le compilateur **SIGNAL** offre les services suivants:

- description graphique ou textuelle d'un algorithme sous la forme d'un ensemble d'expressions sur signaux définissant chacune un signal ou son horloge
- traduction du programme en un graphe de dépendances entre signaux (Graphe aux Dépendances Conditionnées ou GDC) dont chaque sommet représente un signal (et implicitement l'expression qui le définit) et dont les arcs représentent les dépendances (introduites par les expressions) conditionnées par les horloges des signaux
- vérification de l'absence de cycle instantané dans le GDC
- réduction et hiérarchisation des horloges pour produire un Graphe Conditionné Hiérarchisé
- actuellement, génération de code séquentiel C ou FORTRAN

D'autre part, la complexité sans cesse croissante des applications et les contraintes remps réel, conduisent à utiliser des architectures multi-processeur (parallèles, distribuées) lorsqu'il s'agit d'exécuter des algorithmes réactifs. L'environnement **SynDEx** fournit une aide à l'implantation remps réel multi-processeur de ces algorithmes en déchargeant au maximum l'utilisateur des tâches lourdes de programmation bas niveau (système). **SynDEx** offre les services suivants:

- description graphique de l'algorithme sous la forme d'un graphe flot de données conditionné
- description graphique d'un multi-processeur sous la forme d'un graphe
- distribution et ordonnancement optimisant le temps de réponse de l'algorithme sur le multi-processeur
- prévision et visualisation des performances remps réel de l'algorithme sur le multi-processeur
- génération d'exécutif pour fonctionnement en remps réel sur un multi-processeur
- mesures des performances remps réel

De plus, **SynDEx** assure qu'une spécification correcte dans le cadre synchrone conduit à une implantation correcte sur une machine multi-processeur et respecte des contraintes remps réel.

SIGNAL et **SynDEx** sont deux environnements complémentaires, ce qui nous amène à les interfacer. Ce document décrit la méthode d'interface.

2 Principes de l'interface

Les principes de base communs à **SIGNAL** et **SynDEx** sont le concept Synchrone et l'utilisation du flot de données conditionné [1], [4].

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 mémoires 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.

Le concept Synchrone consiste à considérer comme **simultanés** un instant logique d'un signal de sortie et un instant logique d'un signal d'entrée lorsque la valeur de la sortie **dépend** de la valeur de l'entrée. Relativement à un instant logique d'un signal, un autre signal est dit présent s'il a un instant logique simultané (resp. absent s'il n'en a pas). Deux signaux sont synchrones s'ils ont la même horloge, c'est-à-dire si leurs instants logiques sont simultanés deux à deux. Dans ce cadre, la relation globale se décompose, en introduisant des signaux intermédiaires, en relations élémentaires de trois types. Chaque type de relation est décrit relativement à un instant logique d'un de ses signaux d'entrée:

- relation avec calcul introduisant des dépendances inconditionnelles entre entrée(s) et sortie(s). La valeur du signal d'entrée est utilisée dans le calcul de la valeur du signal de sortie. La valeur de la sortie est simultanée avec la valeur de l'entrée et de toutes les autres entrées qui participent à la relation. Les signaux d'entrée et de sortie sont synchrones. Ces relations correspondent aux **fonctions immédiates** de **SIGNAL**.

- relation sans calcul introduisant une dépendance conditionnelle (dans l'instant logique considéré, il y a ou non une dépendance). La valeur de la sortie est égale à la valeur de l'entrée lorsque la valeur de l'autre entrée (signal booléen) qui conditionne la dépendance lui est simultanée et vraie. L'horloge du signal de sortie est incluse dans l'intersection des horloges des deux signaux d'entrée. Ces relations correspondent aux **when** de **SIGNAL**.

- relation sans calcul introduisant une dépendance inconditionnelle. La valeur de la sortie est égale à la valeur d'une entrée si l'autre entrée ne lui est pas simultanée. Si les deux entrées sont simultanées, l'une des deux est prioritaire, c'est elle qui est choisie. L'horloge du signal de sortie est égale à l'union des horloges des deux signaux d'entrée. Ces relations correspondent aux **default** de **SIGNAL**.

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 [2]). Les relations élémentaires et la relation globale ainsi que le graphe correspondant s'appellent "processus" en **SIGNAL.**

L'absence de valeur en entrée d'une relation entraîne une absence de valeur en sortie et donc une absence de dépendance entre entrée et sortie. Cette absence de dépendance peut être provoquée soit par l'absence d'une entrée externe soit par un sommet **when.**

Chaque partie régulière du graphe se représente simplement en répétant un sous-graphe appelé "motif" en SynDEx ou "tableau de processus" en **SIGNAL.** Cette répétition de motif peut être factorisée afin d'une part de réduire le volume de la spécification, et d'autre part de pouvoir transformer la répétition en itération (transformation spatiale → temporelle) [5] .

Le compilateur **SIGNAL** génère le Graphe Conditionné Hiérarchisé (GCH) à partir de ce graphe de la manière suivante :

1. Pour chaque ensemble de signaux d'entrées externes synchrones (synchrones à cause de contraintes internes) le compilateur crée un signal booléen et y associe l'ensemble des sommets transitivement successeurs de cet ensemble de signaux d'entrée. Tous les signaux booléens ainsi crées sont synchrones et les instants vrai de chacun définissent les occurences (l'horloge) des signaux d'entrée de l'ensemble associé.

2. De même, à chaque signal de type booléen entrée de conditionnement d'un (ou plusieurs) sommet(s) **when**, on associe l'ensemble des sommets transitivement successeurs de ce(s) sommet(s) **when.**

3. Lorsque deux de ces ensembles sont en intersection sans que l'un soit inclus dans l'autre, on leur soustrait leur intersection qu'on associe à un nouveau signal booléen généré par un nouveau sommet ajouté par le compilateur au graphe initial. Ce nouveau sommet, soit un **when** soit un **default**, prend pour entrées les deux signaux booléens auxquels sont associés les deux ensembles. Si les deux ensembles sont inclus dans un troisième, le nouveau sommet est associé au booléen auquel est associé ce troisième ensemble.

4. Tous les ensembles résultant de cette transformation sont deux à deux soit disjoints soit inclus l'un dans l'autre, ce qui permet de définir une arborescence des inclusions à laquelle correspond la hiérarchie du GCH.

Actuellement, le compilateur **SIGNAL** peut engendrer une représentation textuelle du GCH, suivant la syntaxe **SIGNAL**, dans un fichier noté "GCH.sig" dans la suite du rapport. S'interfacer avec **SynDEx** consiste pour le compilateur **SIGNAL** à produire un autre fichier texte contenant une liste d'instructions **SynDEx** décrivant le GCH. Ce fichier, noté "GCH.syn" par la suite, sera importé dans l'environnement **SynDEx.**

3 Description et interprétation du GCH

Le GCH est le graphe flot de données conditionné avec hiérarchisation des conditionnements engendré par la compilation d'un programme **SIGNAL.** Le GCH est présenté informellement ci-dessous à travers ses deux formes GCH.sig et GCH.syn. On donne pour chaque exemple une version en **SIGNAL** suivie d'une version en **SynDEx** séparées par un signe ⇒.

La syntaxe complète de **SynDEx** est définie dans la section 4 et pour la syntaxe de **SIGNAL**, on peut consulter [3]. Nous donnons ici juste quelques précisions pour pouvoir comprendre le paragraphe suivant : pour toutes les instructions **SynDEx** de déclaration de sommet, le premier champ

de l'instruction correspond au type du sommet (par exemple, `function`, `memory`, `extract`...) et le deuxième champ correspond au nom du sommet qui doit être unique (dans les exemples ci-dessous, `N1`, `N2` ...).

3.1 Les sommets du GCH

Les types des signaux d'entrée-sortie dans les instructions **SynDEx** de déclaration des sommets du GCH sont les mêmes que ceux de **SIGNAL**. Le GCH est constitué des différents sommets suivants :

- fonction immédiate (expressions arithmétiques ou logiques ou fonctions externes).
```
(| x:=a+b |) where integer a,b,x
  ⇒ (function N1 add 10 integer?a?b!x)
(| c:=x<0 |) where integer x; logical c
  ⇒ (function N2 less 20 integer?x?'0', logical!c)
(| x:=sin{a} |) where dpreal a, x
  ⇒ (function N3 sin 30 dpreal?a!x)
```
Les entiers 10, 20, 30 correspondent aux durées d'exécution des fonctions de calcul, en microsecondes.

- retard et fenêtre glissante sur un signal (mémoire).
```
(| zx:=x$1 |) where real x, zx init 0.0
  ⇒ (memory M1 real?x $1!zx init 0.0)
(| znx:=x$2 |) where real x, znx init [[1]:0.0, [2]:1.0]
  ⇒ (memory M2 real?x $2!znx init [[1]:0.0, [2]:1.0])
(| zwx:=x window 4 |) where real x; [4]real zwx init [{to 3}:0.0]
  ⇒ (memory M3 real?x window 4!zwx init [{to 3}:0.0])
(| znwx:=x$2 window 4 |) where real x; [4]real znwx init [{to 5}:0.0]
  ⇒ (memory M4 real?x $2 window 4!znwx init [{to 5}:0.0])
```

- extraction d'un élément d'un signal de type tableau.
```
(| s:=e[i] |) where [4]real e; integer i; real s
  ⇒ (extract E integer?i, [4]real?e, real!s)
```

- remplacement d'un élément d'un signal de type tableau.
```
(| s:=[i to 4:e[i], [j]:r] |) where [4]real e, s; real r; integer j
  ⇒ (replace R real?r, integer?j, [4]real?e!s)
```

Les quatre types de sommets suivants concernent les tableaux de processus. Si `PN` est défini par :
`array I to N of P with indexation end`
alors `PN` est un tableau de processus dont les éléments P_i sont des instanciations du processus `P`.

- décomposition d'un signal vectoriel en entrée d'un tableau de processus.
```
(| array I to 4 of ...:=...e... with e end |) where [4]real e
  ⇒ (fork F [4]real?e, real!fe)
```
Si une entrée `e` de `PN` est présente dans les indexations (`with e`), alors elle est implicitement indexée par `i` dans `P` (`e` "perd" une dimension).

- composition d'un signal en sortie d'un tableau de processus.
```
(| array I to 4 of s:=... with s end |) where [4]real s
```

⇒ (join J real?s, [4]real!js)
Pour toute sortie s de P possédant le type μ, PN possède une sortie js de type $[1...N]\mu$, telle que js[i] est la sortie s de P_i.

- indice d'un motif.
  (| array ii to 4 of ... end |)
  ⇒ (index I integer!ii)
  Le sommet index n'est engendré que si l'indice intervient dans un calcul dans P.

- dépendance inter-processus dans un tableau de processus.
  (| array I to 4 of s:= s ... with s[0]:ini end |) where real ini; [4]real s
  ⇒ (iterate L incr 4 real?ini?s!sout)
  (| array I to 4 of s:= s ... with s[.]:ini end |) where real ini; [4]real s
  ⇒ (iterate L decr 4 real?ini?s!sout)
  Si s est une entrée de P présente dans les indexations sous la forme with s[0]:ini (resp. with s[.]:ini), chaque entrée s de P_i est indexée implicitement par $i+1$ (resp. $i-1$); P doit posséder une sortie de nom s et chaque entrée s est connectée (est égale) à la sortie de même nom possédant le même indice. L'entrée s d'indice 0 (resp. $N+1$) est égale à INI.

- dépendance conditionnée (sous-échantillonnage).
  (| y:=x when c |) where [4]integer x, y; logical c
  ⇒ (when W5 logical?c, [4]integer?x!y)

- dépendance inconditionnelle avec priorité (mélange).
  (| z:=x default y |) where real x, y, z
  ⇒ (default D6 real?x?y!z)

- intersection de signaux booléens de conditionnement, sommet rajouté par le compilateur SIGNAL.
  (| H_3_H:=H_2_H when H_1_H |)
  ⇒ (hmul H3 logical?H2H?H1H!H)

- union de signaux booléens de conditionnement, sommet rajouté par le compilateur SIGNAL.
  (| H_6_H:=H_5_H default H_4_H |)
  ⇒ (hadd H6 logical?H5H?H4H!H)

- présence d'un signal d'entrée externe représentant de l'ensemble des entrées externes qui lui sont synchrones, sommet rajouté par le compilateur SIGNAL.
  (| H_7_H:=event ext |)
  ⇒ (hinput H7 hext logical!H)

- sommet composé (sous-graphe) permettant de faire ressortir par encapsulation la hiérarchisation dans le GCH.sig. Ce type de sommet est inutilisé dans le GCH.syn car la hiérarchisation y est mise en évidence autrement (voir le paragraphe 3.3).

- mémorisation d'un signal avec changement d'horloge.
  (| y:=x cell c |) where integer x, y; logical c
  ⇒ (cell C logical?c, integer?x!y)

3.2 Les arcs du GCH

Les arcs représentent des transferts de données entre sommets. Chacun des sommets a un ensemble de ports d'entrée et de sortie, qui sont les points de connexion entre sommets. Les ports interconnectés portent le même signal. Un port de sortie peut être connecté à plusieurs ports d'entrée (diffusion), par contre un port d'entrée ne peut être connecté qu'à un seul port de sortie. Les connexions inter-sommets se font implicitement par identité de noms entre instructions **SIGNAL**. Elles se font explicitement en **SynDEx** au moyen de l'instruction `connect`.

`(| x:=abs{a} | y:=f{x} | z:=g{x} | s:=y+z |)` ⇒

```
(function N1 abs 5 real ?a !x)
(function N2 f 20 real ?e !s)
(function N3 g 15 real ?e !s)
(function N4 add 5 real ?a ?b !s)
(connect N1/x N2/e N3/e)
(connect N2/s N4/a)
(connect N3/s N4/b)
```

3.3 La hiérarchisation des conditionnements

Dans GCH.sig, la hiérarchisation apparaît implicitement, d'une part à travers les expressions de "calcul d'horloge" qui génèrent des "signaux d'horloge" (when unaires sur signaux booléens et when et default entre signaux d'horloge), d'autre part à travers les instructions `synchro` entre les signaux d'horloge et les autres signaux. La hiérarchisation est par ailleurs mise en évidence par une encapsulation regroupant dans un sommet composé les sommets ayant la même horloge (l'horloge d'un sommet est l'union des horloges de ses entrées).

Dans GCH.syn, la hiérarchisation apparaît explicitement à travers les instructions `exec` dont chacune représente l'association entre un signal booléen et un ensemble de sommets, comme décrit à la fin du paragraphe 2. Les sommets `hinput`, `hadd` et `hmul` qui n'apparaissent pas dans les instructions `exec`, constituent les racines de la hiérarchie des conditionnements (ils génèrent tous des signaux booléens) et sont regroupés dans une instruction `execroots` (voir les exemples donnés dans le paragraphe 8). Les sommets exécutés inconditionnellement dans le corps d'un tableau de processus sont regroupés entre accolades dans les instructions `execroots` ou `exec`.

3.4 Interprétation du GCH.syn

A chaque exécution du graphe, les sommets de l'instruction `execroots` sont exécutés inconditionnellement et chaque signal booléen dans une instruction `exec` conditionne l'exécution de l'ensemble des sommets qui lui sont associés. L'ordre d'exécution est déterminé par les précédences entre sommets. Les précédences du graphe flot de données établissent un ordre partiel qui laisse de la liberté pour le distribuer et l'ordonnancer sur une architecture multi-processeur [6].

4 Syntaxe 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 connexion ou d'exécution (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 alpha-numérique 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 décrits par ailleurs.

4.1 Instructions de base

4.1.1 Instruction 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 le processus, écrite et compilée séparément par l'utilisateur.
- `<duree>` est la durée d'exécution de la procédure de calcul en microsecondes (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.

L'identificateur `<proc>` ainsi que l'ordre et le type des entrée-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`).

SIGNAL	SynDEx	Exemple de déclaration de sommet 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 notless 10 type?e1?e2, logical!s)
=	equal	(function EQ equal 10 type?e1?e2, logical!s)
/=	notequal	(function NE notequal 10 type?e1?e2, logical!s)
or	logor	(function OL logor 10 logical?e1?e2!s)
and	logand	(function AL logand 10 logical?e1?e2!s)
not	lognot	(function NL lognot 10 logical?e!s)

Note : il n'y a pas d'opérateurs <= ni > ni xor car les deux premiers sont obtenus en permutant 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]]'
```

4.1.2 Instruction 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>`, 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.

4.1.3 Instruction de déclaration de sommet "when"

Syntaxe : `(when <id> logical?<cond>, <type>?<entree>!<sortie>)`

- `<id>` est l'identificateur du sommet when
- `<cond>` est l'identificateur du port d'entrée de conditionnement
- `<type>` est le type commun à `<entree>` et à `<sortie>`
- `<entree>` est l'identificateur du port d'entrée ou une valeur constante
- `<sortie>` est l'identificateur du port de sortie, prenant la valeur de `<entree>` lorsque `<cond>` porte la valeur logique vraie.

Pour la syntaxe de `<type>` et de `<entree>` quand il s'agit d'une constante, voir l'instruction de déclaration de sommet de calcul.
Exemple :

```
(when W1 logical ?c, integer ?i !o)
```

4.1.4 Instruction de déclaration de sommet "default"

Syntaxe : `(default <id> <type>?<prio>?<def>!<sortie>)`

- `<id>` est l'identificateur du sommet default
- `<type>` est le type commun à `<prio>`, à `<def>` et à `<sortie>`
- `<prio>` est l'identificateur du port d'entrée prioritaire
- `<def>` est l'identificateur du port d'entrée par défaut ou une valeur constante
- `<sortie>` est l'identificateur du port de sortie, prenant la valeur de `<prio>` ou à défaut celle de `<def>`

Pour la syntaxe de `<type>` et de `<def>` quand il s'agit d'une constante, voir l'instruction de déclaration de sommet de calcul.
Exemple :

```
(default D2 dpreal ?ip ?id !o)
```

4.1.5 Instruction de déclaration de sommet "cell"

Syntaxe : (cell <id> logical?<cond>, <type>?<entree>!<sortie>)

- <id> est l'identificateur du sommet cell
- <cond> est l'identificateur de l'entrée de conditionnement
- <type> est le type commun à **<entree>** et à **<sortie>**
- <entree> est l'identificateur du port d'entrée de mise à jour de la mémoire
- <sortie> est l'identificateur du port de sortie, prenant la valeur de **<entree>** à chaque occurence de <entree> ou la dernière valeur de **<entree>** à chaque occurence **true** de <cond>

Pour la syntaxe de <type>, voir l'instruction de déclaration de sommet de calcul.
Exemple :

```
(cell C logical ?c, dpreal ?i !o)
```

4.2 Instructions de calcul d'horloge

Ce sont les sommets rajoutés par le compilateur (voir fin du paragraphe 2).

4.2.1 Instruction de déclaration de sommet "hinput"

Syntaxe : (hinput <id> <proc> logical!<sortie>)

- <id> est l'identificateur du sommet hinput
- <proc> est l'identificateur de la procédure de lecture de l'entrée externe booléenne, écrite et compilée séparément par l'utilisateur.
- <sortie> est l'identificateur du port de sortie

Exemple :

```
(hinput H0 getH0 logical !o)
```

4.2.2 Instruction de déclaration de sommet "hmul"

Syntaxe : (hmul <id> logical?<entree1>?<entree2>!<sortie>)

- <id> est l'identificateur du sommet hmul
- <entree1> et <entree2> sont les identificateurs des deux ports d'entrée
- <sortie> est l'identificateur du port de sortie

Exemple :

```
(hmul H1 logical ?i1 ?i2 !o)
```

4.2.3 Instruction de déclaration de sommet "hadd"

Syntaxe : `(hadd <id> logical?<entree1>?<entree2>!<sortie>)`

- `<id>` est l'identificateur du sommet hadd
- `<entree1>` et `<entree2>` sont les identificateurs des deux ports d'entrée
- `<sortie>` est l'identificateur du port de sortie

Exemple :

```
(hadd H2 logical ?i1 ?i2 !o)
```

4.3 Instructions d'accès aux éléments de signaux de type tableau

Pour les instructions **extract** et **replace**, **type** représente le type (**logical** ou **integer** ou **real** ou **dpreal**) des éléments du tableau et `[<d1>, ..., <dn>]` représente la liste des n dimensions (constantes entières) du tableau.

Les instructions **extract** et **replace** n'accèdent qu'à la première dimension `<d1>` des tableaux et donc extraient/remplacent des sous-tableaux à $n-1$ dimensions (c'est-à-dire des éléments du tableau si $n = 1$).

Pour le futur, on peut envisager des extensions de la syntaxe des instructions **extract** et **replace** permettant d'extraire ou de remplacer un ensemble d'éléments consécutifs ou désignés par un tableau d'indices.

4.3.1 Instruction de déclaration de sommet "extract"

Syntaxe :

$n > 1$: `(extract <id> integer?<i>, [<d1], <d2>, ...]<type>?<ae>, [<d2>, ...]<type>!<s>)`

$n = 1$: `(extract <id> integer?<i>, [<d1>]<type>?<ae>, <type>!<s>)`

- `<id>` est l'identificateur du sommet extract
- `<i>` est l'identificateur du port d'entrée d'indexation (obligatoirement de type **integer**)
- `<type>` est le type des éléments du port de sortie `<s>` et du port d'entrée `<ae>`
- `<d1>` est la constante entière première dimension du type tableau du port d'entrée `<ae>`

La sortie `<s>` prend la valeur du sous-tableau (ou de l'élément pour $n = 1$) d'indice `<i>` du tableau d'entrée `<ae>`. L'une des deux entrées peut être de valeur constante. Exemples :

```
(extract X1 integer ?i, [5]real ?e, real !s)
(extract X2 integer ?'3', [5]real ?e, real !s)
(extract X3 integer ?i, [2]real ?'[[1]:0.5, [2]:2.0]', real !s)
(extract X4 integer ?i, [5,2,3]real ?e, [2,3]real !s)
```

4.3.2 Instruction de déclaration de sommet "replace"

Syntaxe :

$n > 1$: `(replace <id> [<d2>, ...]<type>?<e>, integer?<i>, [<d1>, <d2>, ...]<type>?<ae>!<as>)`

$n = 1$: `(replace <id> <type>?<e>, integer?<i>, [<d1>]<type>?<ae>!<as>)`

- `<id>` est l'identificateur du sommet replace
- `<type>` est le type des éléments du port d'entrée `<e>`, du port d'entrée `<ae>` et du port de sortie `<as>`
- `<i>` est l'identificateur du port d'entrée d'indexation (obligatoirement de type **integer**)
- `<d1>` est la constante entière première dimension du type tableau commun aux ports d'entrée `<ae>` et de sortie `<as>`

La sortie tableau `<as>` prend la valeur de l'entrée tableau `<ae>` sauf le sous-tableau (l'élément pour $n = 1$) d'indice `<i>` qui prend la valeur de l'entrée `<e>`. L'une ou deux des trois entrées peut être de valeur constante. Exemples :

```
(replace R1 real ?r, integer ?i, [5]real ?e !s)
(replace R2 real ?'0.0', integer ?i, [5]real ?e !s)
(replace R3 real ?'0.0', integer ?'3', [5]real ?e !s)
(replace R4 real ?'1.0', integer ?i, [5]real ?'[{to 5}:0.0]' !s)
(replace R5 [2,3]real ?r, integer ?i, [5,2,3]real ?e !s)
```

4.4 Instructions de factorisation de motifs de graphes répétitifs

Les instructions **fork**, **join** et **iterate** d'un même motif doivent avoir les mêmes dimensions (des tableaux pour **fork** et **join**, du motif pour **iterate**), et les instructions **iterate** d'un même motif doivent toutes établir des connexions inter-motifs dans le même sens. La traduction en SynDEx d'un tableau de processus SIGNAL comporte toujours au moins une des trois instructions **join**, **fork** ou **iterate**.

Pour les instructions **fork** et **join**, **type** représente le type (**logical** ou **integer** ou **real** ou **dpreal**) des éléments du tableau et `[<d1>, ..., <dn>]` représente la liste des n dimensions (constantes entières) du tableau.

Les instructions **fork** et **join** n'accèdent qu'à la première dimension `<d1>` des tableaux et donc énumèrent/collectent des sous-tableaux à $n - 1$ dimensions (c'est-à-dire des éléments du tableau si $n = 1$).

4.4.1 Instruction de déclaration de sommet "fork"

Syntaxe :

$n > 1$: `(fork <id> [<d1>, <d2>, ...]<type>?<ae>, [<d2>, ...]<type>!<s>)`

$n = 1$: `(fork <id> [<d1>]<type>?<ae>, <type>!<s>)`

- `<id>` est l'identificateur du sommet fork
- `<type>` est le type des éléments du port de sortie `<s>` et du port d'entrée `<ae>`

- <d1> est la constante entière première dimension du port d'entrée <ae> ; c'est aussi la borne supérieure du motif.

L'entrée <ae> est soit une constante tableau, soit connectée à la sortie d'un sommet externe au motif. Pour chaque répétition i du motif (de 1 à <d1>), la sortie <s> prend la valeur de <ae>[i].
Exemples :

```
(fork F1 [20]real ?e, real !s)
(fork F2 [20,5]real ?e, [5]real !s)
```

4.4.2 Instruction de déclaration de sommet "join"

Syntaxe :

$n > 1$: (join <id> [<d2>, ...]<type>?<e>, [<d1>, <d2>, ...]<type>!<as>)
$n = 1$: (join <id> <type>?<e>, [<d1>]<type>!<as>)

- <id> est l'identificateur du sommet join
- <type> est le type des éléments du port d'entrée <e> et du port de sortie <as>
- <d1> est la constante entière première dimension du type tableau du port de sortie <as>

La sortie tableau <as> est connectée à l'entrée d'un sommet externe au motif répétitif. Pour chaque répétition du motif (de 1 à <d1>), l'entrée <e> donne sa valeur à l'élément de <as> dont l'indice est celui du motif.
Exemples :

```
(join J1 real ?e, [20]real !s)
(join J2 [5]real ?e, [20,5]real !s)
```

4.4.3 Instruction de déclaration de sommet "index"

Syntaxe : (index <id> integer!<i>)

- <id> est l'identificateur du sommet index
- <i> est l'identificateur de la sortie (obligatoirement de type integer)

Pour chaque répétition du motif, la sortie <i> prend la valeur de l'indice du motif.
Exemple :

```
(index I integer !i)
```

4.4.4 Instruction de déclaration de sommet "iterate"

Syntaxe :

dans le sens croissant : (iterate <id> incr <dim> <type>?<ini>?<e>!<s>)
dans le sens décroissant : (iterate <id> decr <dim> <type>?<ini>?<e>!<s>)

- <id> est l'identificateur du sommet iterate
- <dim> est la dimension du motif (constante entière)

- <type> est le type commun aux ports d'entrée <ini> et <e> et de sortie <s>. <dim> peut être un type tableau.

Cette instruction permet de spécifier une connexion inter-motifs (qui apparaît dans le graphe factorisé du motif de l'array comme un cycle à travers un sommet iterate ; c'est l'analogue "instantané" du retard). incr ou decr spécifie le sens de la connexion, incr dans le sens croissant, decr dans le sens décroissant des indices du motif répété <dim> fois. La sortie <s> prend la valeur soit de l'entrée <ini> pour l'indice 1 (cas incr) ou <dim> (cas decr) du motif, soit de l'entrée <e> pour les autres valeurs de l'indice du motif. L'entrée <ini> peut être de valeur constante.
Exemples :

```
(iterate I1 incr 20 real ?z ?e !s)
(iterate I2 incr 20 [5]real ?'[{to 5}:0.0]' ?e !s)
(iterate I1 decr 50 integer ?'0' ?e !s)
```

4.5 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.

4.5.1 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)
```

4.5.2 Instruction d'exécution inconditionnelle

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

4.5.3 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.

4.5.4 Extension de la syntaxe des instructions execroots et exec

Les sommets exécutés inconditionnellement dans le corps d'un motif doivent être regroupés entre accolades dans les instructions **execroots** ou **exec** ; les sommets **fork**, **index**, **iterate** et **join** doivent toujours apparaître à l'intérieur des accolades relatives au motif qu'ils spécifient.

5 Règles de traduction SIGNAL-SynDEx

5.1 Format de présentation

5.1.1 Syntaxe abstraite de SIGNAL

Dans la suite du document, les lignes commençant par ♠ et ◊ concernent la syntaxe abstraite de **SIGNAL**, les identificateurs en majuscule représentent les phylums (ou classes d'arbres) et les identificateurs en gras les opérateurs des sous-arbres. Les règles commençant par ♠ ont, en partie gauche du ::= un nom de phylum et en partie droite, une suite de phylums ou d'opérateurs qui définissent le phylum de la partie gauche.
Par exemple, la règle :

♠ SIGN ::= IDENT **init**

signifie que les sous-arbres appartenant au phylum SIGN sont, soit des sous-arbres appartenant au phylum IDENT , soit des sous-arbres ayant **init** comme opérateur.
les règles commençant par ◊ ont, en partie gauche de ⟶ le nom de l'opérateur d'un sous-arbre et en partie droite la liste des phylums ou des opérateurs de ses fils. Le nombre de termes en partie droite définit donc l'arité du sous-arbre en question.
Par exemple, la règle :

◊ **corp** ⟶ [*expression sur* PROC] [**dec_sig**]* [**exte**]*

signifie que le sous-arbre **corp** (corps d'un processus) a trois fils, le premier appartenant au phylum [*expression sur* PROC] (expression de processus), le deuxième étant une liste (éventuellement vide) de sous-arbres d'opérateur **dec_sig** (déclarations de signaux locaux) et le troisième une liste (éventuellement vide) de sous-arbres d'opérateur **exte** (déclarations de processus externes).
Dans la règle suivante :

◊ **t_vect** ⟶ [*expression sur* TEMP] $^{+}$ I_TYPE

le $^{+}$ indique qu'il s'agit d'une liste de [*expression sur* TEMP] , cette liste ayant au moins un élément. Lorsque le caractère $^{+}$ est remplaçé par *, la liste peut être vide.
Dans la règle suivante :

◊ **parc** ⟶ [IDENT]$^{0|1}$ BORN

le premier fils du sous-arbre d'opérateur **parc** est, soit un sous-arbre appartenant au phylum IDENT , soit absent.

5.1.2 Règles de décompilation SIGNAL-SynDEx

Exemple :

decinput:

$$\_\ (\vdash \textbf{event} \dashv, H) \ \mapsto \ \boxed{(}\ \boxed{\text{hinput}}\ \square\ extinnom(H)\ \square\ getnom(H)\ \square\ \boxed{\text{logical}}\ \square\ \boxed{!}\ getnom(H)\ \boxed{)}$$

$$-\ (T, X) \ \mapsto \ hinnode(getsighorl(X), getnom(X))\ xinnode(type(T), X)$$

Le nom de la règle (ou de la fonction de décompilation) est écrit en italique (dans l'exemple ci-dessus *decinput*) et les lignes commençant par un • définissent la règle. La partie à gauche de

$\mapsto$ correspond aux paramètres de la règle, c'est-à-dire à des sous-arbres du graphe tandis que la partie droite est le résultat de l'application de la règle sur les paramètres. Un paramètre entre $\vdash$ $\dashv$ correspond à un sous-arbre **SIGNAL** précis (avec le nom de l'opérateur ainsi que ses fils). Dans l'exemple ci-dessus, l'application de la fonction *decinput* sur le sous-arbre $\vdash$ **event** $\dashv$ (qui représente le type **event**) et sur l'horloge H engendre l'affichage des mots-clés $\boxed{(}$, $\boxed{\text{hinput}}$, $\square$ (espace), l'appel de la fonction $extinnom(H)$, l'affichage de $\square$, l'appel de la fonction $getnom(H)$ etc... Lorsque le type est différent du type **event** et que le second paramètre est un signal, l'application de la fonction *decinput* engendre (deuxième ligne commençant par •) l'appel à la fonction $hinnode(getsighorl(X), getnom(X))$ suivie de l'appel à $xinnode(type(T), X)$. Les mots-clés sont toujours entourés par un rectangle.

5.2 Généralités

- Un programme **SynDEx** sera obtenu à partir du GCH dont le fichier $NOM\_TRA.SIG$ est une forme externe, par la traduction $process(\vdash GCH \dashv)$ dans un fichier $NOM.SYN$ où NOM est le nom du programme **SIGNAL**.
- Les expressions sur processus imbriquées sont récursivement depliées
  - dans $\vdash$ **feed**$^+$ $E_1 \ldots E_n \dashv$, tout E_i de la forme $\vdash$ **feed**$^+$ $E_{i1} \ldots E_{in} \dashv$ est expansé sur place en la concaténation des expansions de chaque E_{ij}.
- au niveau du graphe, les appels de processus locaux n'existent plus puisqu'ils ont été expansés sur place. Les seuls processus restants sont externes.

5.3 Identificateurs

Tous les noms de sommets doivent être différents. Ils sont fournis par les fonctions :

1. $exthnom(h)$ pour une horloge,
2. $intinnom(x)$ pour le nom du sommet contenant l'appel à la fonction de lecture d'une variable x ($extinnom(x)$ est le nom externe de cette fonction)
3. $intoutnom(x)$ pour le nom du sommet contenant l'appel à la fonction d'écriture d'une variable x ($extoutnom(x)$ est le nom externe de cette fonction)
4. $intfunom(f)$ pour le nom du sommet contenant l'appel à la fonction dont le nom externe est f .
5. $nomn(x)$ pour le nom du sommet contenant le calcul de la variable x .
6. $nome(x)$ pour le nom du sommet contenant l'extraction d'une dimension du signal vectoriel x .
7. $nomf(x)$ pour le nom du sommet *fork* associé à une entrée x d'un tableau de processus.
8. $nomi(x)$ pour le nom du sommet *iterate* associé à une entrée x d'un tableau de processus.
9. $nomj(x)$ pour le nom du sommet *join* associé à une sortie x d'un tableau de processus.

Ils sont construits à partir des identificateurs **SIGNAL** :

♠ [*expression sur* TEMP] ::= IDENT

♠ IDENT ::= **nom ident**

◇ **ident** ⟶ **nom** [SUFF]*

♠ SUFF ::= **cst_ent nom**

◇ **nom** ⟶ implemented as IDENTIFIER

nom:

- (⊢ **nom** α ⊣) ↦
  α *est une chaîne de caractères directement transcrite en SynDEx*
- (⊢ **ident** X ⊢ ⊣ ⊣) ↦ $nom(\vdash X \dashv)$
- (⊢ **ident** X ⊢ S_1 $S_2 \ldots$ ⊣ ⊣) ↦ $nom(\vdash X \dashv)$ ⊡
  $nom(\vdash$ **ident** ⊢ **nom** S_1 ⊣ ⊢ $S_2 \ldots$ ⊣ ⊣)
  En **SIGNAL**, *un identificateur est un nom commençant par une lettre suivie d'un nombre quelconque de caractères significatifs parmi les lettres, les chiffres, et le caractère _ dont chaque occurrence est suivie par une lettre ou un chiffre. Par exemple, SIG_A1_5 est un identificateur* **SIGNAL**.

5.4 Programme

◇ **process** ⟶ IDENT **inte corp**

process:

- (⊢ **process** ID ES CO ⊣) ↦ $inte(ES)$ $corp(CO)$ $connexions(ES, CO)$
  Le résultat est mis dans le fichier de nom $nom(ID)$.SYNDEX

◇ **corp** ⟶ [*expression sur* PROC] [**dec_sig**]* [**exte**]*

Au niveau du **GCH**, *les seuls sous-processus sont les processus externes.*

corp:

- (⊢ **corp** EP DSL DPE ⊣) ↦ $decnode(EP)$
  Pour chaque expression de processus EP , *le graphe fournit un contexte (signaux locaux et processus externes) dans lequel est évalué* EP *et qui devrait indicer la fonction decnode(). Pour alléger cette présentation, nous omettrons, dans la suite, les paramètres* DSL *(déclarations de signaux locaux) et* DPE *(déclarations de processus externes) directement accessibles à partir de chaque sommet du graphe.*

5.5 Interface du programme

◇ **inte** ⟶ [**dec_sig**]* [**dec_sig**]* [**dec_sig**]*

◇ **dec_sig** ⟶ TYPE [*expression sur* SIGN] $^+$

◇ **dec_sig** ⟶ **event** [IDENT]$^+$

inte:

- $(\vdash$ **inte** $\vdash \dashv$ *LTE LTS* $\dashv) \mapsto$ *declinout*(*LTE*, ?) *declinout*(*LTS*, !)
  La liste de paramètres est vide pour le GCH car les paramètres ont été remplacés par leurs valeurs effectives.

declinout:

- $(\vdash \dashv, sens) \mapsto$
  Un programme peut n'avoir ni entrée ni sortie

- $(\vdash L \dashv, sens) \mapsto$ *decelinout*$(L, sens)$

- $(\vdash L_1\ L_2 \ldots \dashv, sens) \mapsto$ *decelinout*$(L_1, sens)$
  declinout$(\vdash L_2 \ldots \dashv, sens)$

decelinout:

- $(\vdash$ **dec_sig** $T \vdash X \dashv \dashv, sens) \mapsto$ *decextsig*$(T, X, sens)$

- $(\vdash$ **dec_sig** $T \vdash X_1\ X_2 \ldots \dashv \dashv, sens) \mapsto$ *decextsig*$(T, X_1, sens)$
  decelinout$(\vdash$ **dec_sig** $T \vdash X_2 \ldots \dashv \dashv, sens)$

decextsig:

- $(T, X, ?) \mapsto$ *decinput*(T, X)

- $(T, X, !) \mapsto$ *decoutput*(T, X)

5.5.1 Déclaration de signaux

La traduction des déclarations locales fournit les fonctions utilisées pour la production des sommets.

♠ `SIGN ::= IDENT` **init**

◇ **init** ⟶ `IDENT` [*expression sur* `SIGN`]

Toute méta fonction $MF(x)$, dénotant *gettype*(x), *getinit*(x), *getnom*(x) est appliquée sur la liste des déclarations pour trouver la déclaration correspondante $\vdash$ **dec_sig** $T\ x \dashv$. On la décrit à ce niveau.

getnom:

- $(\vdash$ **dec_sig** $T \vdash$ **init** $X\ V \dashv \dashv) \mapsto$ *nom*(X)

- $(\vdash$ **dec_sig** $T\ X \dashv) \mapsto$ *nom*(X)

gettype:

- $(\vdash$ **dec_sig** $T\ X \dashv) \mapsto$ *type*(T)

getinit:
- (├ **dec_sig** *T* ├ **init** *X V* ┤ ┤) ↦ *deceval(V)*

- (├ **dec_sig** *T X* ┤) ↦ *zero(type(T))*
  Rend une valeur 0 arbitraire pour le type T.

deceval:
- (├ *V* ┤) ↦
  Rend le littéral de la valeur V.

5.5.2 Types des signaux

Les types simples sont les suivants :

♠ TYPE ::= I_TYPE

♠ I_TYPE ::= **logical integer real dpreal**

Ils sont décompilés sous la forme suivante ;

type:
- (├ **event** ┤) ↦ logical

- (├ **logical** ┤) ↦ logical

- (├ **integer** ┤) ↦ integer

- (├ **real** ┤) ↦ real

- (├ **dpreal** ┤) ↦ dpreal

Pour les types tableaux, ils ont en **SIGNAL** la syntaxe suivante :

♠ TYPE ::= t_vect

◊ t_vect ⟶ [*expression sur* TEMP] $^{+}$ I_TYPE

[*expression sur* TEMP] $^{+}$ est ici une expression constante qui fournit la liste des bornes supérieures du tableau.
Ils sont décompilés de la manière suivante :

type:
- (├ t_vect *LI TEL* ┤) ↦ [*deceval(LI)*] *type(TEL)*
  LI est la liste des bornes supérieures du tableau et TEL est le type de ses éléments.

5.5.3 Entrées du programme

decinput:

$(\vdash \textbf{event} \dashv, H) \mapsto$ (hinput) □ *extinnom*(*H*) □ *getnom*(*H*) □ (logical) □ (! *getnom*(*H*))

$(T, X) \mapsto$ *hinnode*(*getsighorl*(*X*), *getnom*(*X*)) *xinnode*(*type*(*T*), *X*)

hinnode:

- $(h, \alpha) \mapsto$ (hinput) □ *exthnom*(*h*) □ α □ (logical) □ (! *getnom*(*h*))

  Lorsque plusieurs signaux d'entrée ont la même horloge h, une seule instruction hinput est engendrée pour cette horloge.

xinnode:

- $(t, x) \mapsto$ (function) □ *infoinfunc*(*x*) □ *t* □ (! *getnom*(*x*))

infoinfunc:

- $(x) \mapsto$ *intinnom*(*x*) □ *extinnom*(*x*) □ *delta*(*x*)

intinnom:

- $(x) \mapsto$ [r] *nom*(*x*)

extinnom:

- $(x) \mapsto$ [r] *nom*(*x*)

exthnom:

- $(h) \mapsto$ [rh] *hnum*(*h*) [h]

hnum:

- $(h) \mapsto \alpha$

  La chaîne de caractères représentant l'entier unique attribué à *h* par le compilateur.

getsighorl:

- $(x) \mapsto$

  Fournit l'horloge du signal x

5.5.4 Sorties du programme

decoutput:

- $(T, X) \mapsto$ *xoutnode*(*type*(*T*), *X*)

xoutnode:

- $(t, \vdash \textbf{init}\ X\ V \dashv) \mapsto$ (function) □ *infoutfunc*(*X*) □ *t* □ (? *getnom*(*X*))

- $(t, x) \mapsto$ (function) □ *infoutfunc*(*x*) □ *t* □ (? *getnom*(*x*))

infoutfunc:

- $(f) \mapsto$ *intoutnom*(*f*) □ *extoutnom*(*f*) □ *delta*(*f*)

intoutnom:

- $(x) \mapsto$ [w] *nom*(*x*)

extinnom:
- $(x) \quad \mapsto \quad \boxed{w} nom(x)$

5.5.5 Évolutions à court terme

- Un type externe serait nécessaire. Il pourrait avoir la forme
  ⊢ **external** *NOMTYPE* ⊢ *X_1 X_2 ...* ⊣ ⊣ , dont on devra connaître la représentation ou les fonctions associées.

5.6 Expressions sur processus.

5.6.1 Déclaration de processus externe

Les déclarations de processus n'ont pas de traduction directe en **SynDEx** ; on n'en voit que les appels. En conséquence, les noms des signaux formels ont disparu et ont été remplacés par les paramètres effectifs. On suppose ici que les types des paramètres effectifs sont exactement ceux des formels. Pour une fonction externe (lecture, écriture,...) la fonction ci-dessous doit être définie :
delta:
- $(f) \quad \mapsto$
  Cette fonction donne la durée d'exécution de f; la valeur à engendrer par défaut est 10. Dans une version ultérieure, il faudra prévoir une table pour que le générateur puisse calculer ce temps. Cette table dépendant du langagage cible, elle sera dans un fichier séparé.

Les seuls processus locaux admis sont des déclarations d'externes :

◇ **exte** ⟶ IDENT **inte**

5.6.2 Invocation de processus externe

♠ [*expression sur* PROC] ::= **appe inst**

◇ **appe** ⟶ REF_PRO [*expression sur* TEMP] *

◇ **inst** ⟶ IDENT ⊢ ⊣

♠ REF_PRO ::= IDENT **inst**

decnode:

- (⊢ **appe** ⊢ **inst** P ⊢ ⊣ ⊣ LSE ⊣) $\mapsto$ *decnode*(⊢ **appe** P LSE ⊣)

- (⊢ **inst** P ⊢ ⊣ ⊣) $\mapsto$ *decnode*(⊢ **appe** P ⊢ ⊣ ⊣)

- (⊢ **appe** P ⊢ ⊣ ⊣) $\mapsto$ *decnode*(⊢ **appe** P *getlinput*(P) ⊣)

- (⊢ **appe** P LSE ⊣) $\mapsto$ *funcnode*(*getdecl*(P,!), P, LSE)

  1. *Les connexions sont gérées par ailleurs.*
  2. *Les signaux effectifs ne peuvent être que des valeurs constantes ou des identificateurs de signaux.*

funcnode:

- (LSS, P, LSE) $\mapsto$ [(function] $\square$ *infofunc*(P) $\square$ *funces*(LSS, LSE)]

  Une expression E_i de LSE qui ne serait ni un identificateur ni une constante engendre la création de sommets auxiliaires.

infofunc:

- (P) $\mapsto$ *intfunom*(P) $\square$ *nom*(P) $\square$ *delta*(P)

intfunom:

- (f) $\mapsto$

  Nom unique délivré pour chaque occurrence syntaxique d'appel à la fonction f.

getdecl:

- (P,?) $\mapsto$ LTE

- (P,!) $\mapsto$ LTS

  On suppose que la déclaration de P est ⊢ **exte** P ⊢ **inte** ⊢ ⊣ LTE LTS ⊣ ⊣

funces:

- (LTE,⊢ ⊣) $\mapsto$ *ldecinout*(LTE,?)

- (⊢ ⊣, LTS) $\mapsto$ *ldecinout*(LTS,!)

- (LTE, LTS) $\mapsto$ *ldecinout*(LTE,?) [,] *ldecinout*(LTS,!)

ldecinout:

- (⊢ L_1 ⊣, *sens*) $\mapsto$ *dectinout*(L_1, *sens*)

- (⊢ L_1 L_2 ... ⊣, *sens*) $\mapsto$ *dectinout*(L_1, *sens*) [,] *ldecinout*(⊢ L_2 ... ⊣, *sens*)

Changements d'horloge

♠ [*expression sur* TEMP] ::= **extr mele**

◇ **mele** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **extr** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

decexpr:

- $(X,\vdash$ **mele** $E_1\ E_2 \dashv) \mapsto$ [(default $\square$ *nomn*(X) $\square$ *funces*($\vdash E_1\ E_2 \dashv, X$))]

- $(X,\vdash$ **extr** $E_1\ C_2 \dashv) \mapsto$ [(when $\square$ *nomn*(X) $\square$ *funces*($\vdash E_1\ C_2 \dashv, X$))]

♠ [*expression sur* TEMP] ::= **evnt when1**

◇ **when1** ⟶ [*expression sur* TEMP]

◇ **evnt** ⟶ [*expression sur* TEMP]

decexpr:

- $(H,\vdash$ **when1** $C_1 \dashv) \mapsto$
  N'engendre rien puisque booléens et horloges sont confondus en SynDEx. Toute référence à l'horloge H, dans le graphe Signal, sera représentée, en SynDEx, par le booléen C_1.

- $(H,\vdash$ **event** $X \dashv) \mapsto$
  n'engendre rien sauf si X est une entrée du programme (voir le paragraphe 5.5.3).

Variables

♠ [*expression sur* TEMP] ::= **memo**

◇ **memo** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

decexpr:

- $(X,\vdash$ **memo** $E_1\ C_2 \dashv) \mapsto$ [(cell $\square$ *nomn*(X) $\square$ *funces*($\vdash E_1\ E_2 \dashv, X$))]

5.7.3 Les expressions logiques et arithmétiques

Actuellement, en **SynDEx**, les expressions logiques et arithmétiques sont considérées comme des appels à des fonctions externes. La décompilation de ces expressions est donc similaires à celle des appels de fonctions (voir le paragraphe 5.6.2).

decexpr:

- $(X, \vdash op\ E_1 E_2 \dashv) \mapsto decexpr(nouvocc(X), E_i)$

- $(X, \vdash op\ E_1 \dashv) \mapsto decexpr(nouvocc(X), E_1)$
  Si une sous-expression n'est pas un identificateur ou une constante, un sommet supplémentaire est engendré pour son calcul en utilisant un signal nouvocc(X) créé à partir de X ; le type de ce signal nouvocc(X) est celui qui est déterminé par l'opérateur ou bien c'est celui de X .

Opérateurs logiques

♠ [*expression sur* TEMP] ::= **ou et non**

◇ **ou** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **et** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **non** ⟶ [*expression sur* TEMP]

decexpr:

- $(X, \vdash \textbf{ou}\ E_1\ E_2 \dashv) \mapsto$ (function □ *infoope*(**ou**) □ *funces*($\vdash E_1\ E_2 \dashv, X$))

- $(X, \vdash \textbf{et}\ E_1\ E_2 \dashv) \mapsto$ (function □ *infoope*(**et**) □ *funces*($\vdash E_1\ E_2 \dashv, X$))

- $(X, \vdash \textbf{non}\ E_1 \dashv) \mapsto$ (function □ *infoope*(**non**) □ *funces*($\vdash E_1\ E_2 \dashv, X$))

infoope:

- (**ou**) ↦ *intfunom*(**ou**) □ logor □ *delta*(**ou**)

- (**et**) ↦ *intfunom*(**et**) □ logand □ *delta*(**et**)

- (**non**) ↦ *intfunom*(**non**) □ lognot □ *delta*(**non**)

intfunom:

- (*ope*) ↦
  Nom unique délivré pour chaque occurrence de l'opérateur ope

♠ [*expression sur* TEMP] ::= **diff egal infe i_egal supe s_egal**

◇ **diff** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **egal** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **infe** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **i_egal** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **supe** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **s_egal** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

decexpr:

- $(X, \vdash$ **diff** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**diff**) $\square$ *funces*($\vdash E_1\ E_2 \dashv, X$))]
- $(X, \vdash$ **egal** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**equal**) $\square$ *funces*($\vdash E_1\ E_2 \dashv, X$))]
- $(X, \vdash$ **infe** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**infe**) $\square$ *funces*($\vdash E_1\ E_2 \dashv, X$))]
- $(X, \vdash$ **i_egal** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**i_egal**) $\square$ *funces*($\vdash E_1\ E_2 \dashv, X$))]
- $(X, \vdash$ **supe** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**supe**) $\square$ *funces*($\vdash E_2\ E_1 \dashv, X$))]
- $(X, \vdash$ **s_egal** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**s_egal**) $\square$ *funces*($\vdash E_2\ E_1 \dashv, X$))]

infoope:

- (**diff**) $\mapsto$ *intfunom*(**diff**) $\square$ [notequal] $\square$ *delta*(**diff**)
- (**egal**) $\mapsto$ *intfunom*(**egal**) $\square$ [equal] $\square$ *delta*(**egal**)
- (**infe**) $\mapsto$ *intfunom*(**infe**) $\square$ [less] $\square$ *delta*(**infe**)
- (**i_egal**) $\mapsto$ *intfunom*(**i_egal**) $\square$ [notless] $\square$ *delta*(**i_egal**)
- (**supe**) $\mapsto$ *intfunom*(**supe**) $\square$ [less] $\square$ *delta*(**supe**)
- (**s_egal**) $\mapsto$ *intfunom*(**s_egal**) $\square$ [notless] $\square$ *delta*(**s_egal**)

Opérateurs arithmétiques

- ♠ [*expression sur* TEMP] ::= **nega divi moin mult plus puis**
- ◇ **nega** ⟶ [*expression sur* TEMP]
- ◇ **divi** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]
- ◇ **moin** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]
- ◇ **mult** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]
- ◇ **plus** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◇ **puis** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

decexpr:

- $(X, \vdash$ **nega** $E_1 \dashv) \mapsto$ [(function $\square$ *infoope*(**nega**) $\square$ ***funces***(E_1, X))]
- $(X, \vdash$ **divi** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**divi**) $\square$ ***funces***$(\vdash E_1\ E_2 \dashv, X)$)]
- $(X, \vdash$ **moin** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**moin**) $\square$ ***funces***$(\vdash E_1\ E_2 \dashv, X)$)]
- $(X, \vdash$ **mult** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**mult**) $\square$ ***funces***$(\vdash E_1\ E_2 \dashv, X)$)]
- $(X, \vdash$ **plus** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**plus**) $\square$ ***funces***$(\vdash E_1\ E_2 \dashv, X)$)]
- $(X, \vdash$ **puis** $E_1\ E_2 \dashv) \mapsto$ [(function $\square$ *infoope*(**puis**) $\square$ ***funces***$(\vdash E_1\ E_2 \dashv, X)$)]

infoope:

- (**nega**) $\mapsto$ *intfunom*(**nega**) $\square$ [negate] $\square$ *delta*(**nega**)
- (**divi**) $\mapsto$ *intfunom*(**divi**) $\square$ [div] $\square$ *delta*(**divi**)
- (**moin**) $\mapsto$ *intfunom*(**moin**) $\square$ [sub] $\square$ *delta*(**moin**)
- (**mult**) $\mapsto$ *intfunom*(**mult**) $\square$ [mul] $\square$ *delta*(**mult**)
- (**plus**) $\mapsto$ *intfunom*(**plus**) $\square$ [add] $\square$ *delta*(**plus**)
- (**puis**) $\mapsto$ *intfunom*(**puis**) $\square$ [power] $\square$ *delta*(**puis**)

5.7.4 Les expressions vectorielles

♠ [*expression sur* VECT] ::= [ELE_VEC]$^+$

♠ ELE_VEC ::= **ele_vec ens_vec**

◇ **ele_vec** ⟶ [*expression sur* TEMP] $^+$ [*expression sur* TEMP]

◇ **ens_vec** ⟶ [**parc**]$^+$ D_ELE_VEC

♠ D_ELE_VEC ::= [*expression sur* TEMP] **ele_vec**

◇ **parc** ⟶ [IDENT]$^{0\|1}$ BORN

♠ BORN ::= **born**

◇ **born** ⟶ [TEMP]$^{0||1}$ [TEMP]$^{0||1}$ [TEMP]$^{0||1}$

Les expressions vectorielles ont la même syntaxe en **SIGNAL** et en **SynDEx** .
Exemple :

```
(|coef := [to 9:0.0, [5]:1.0] when h8h|) ⇒
(when COEF logical?H8H, [9]real?'[to 9:0.0, [5]:1.0]'!coef)
```

5.7.5 Les indexations ou éléments de tableau

♠ [*expression sur* TEMP] ::= **elem**

◇ **elem** ⟶ IDENT [*expression sur* TEMP] $^+$

decexpr:
- $(X,\vdash$ **elem** $T \vdash L_1\ L_2 \ldots \dashv \dashv) \mapsto$ *declextract*$(X,\vdash$ **elem** $T \vdash L_1\ L_2 \ldots \dashv \dashv)$

declextract:
- $(X,\vdash$ **elem** $T \vdash I_1\ I_2 \ldots \dashv \dashv) \mapsto$ *decextract*$(nouvocc(X),\vdash$ **elem** $T \vdash I_1 \dashv \dashv)$ *declextract*$(X,\vdash$ **elem** $T \vdash I_2 \ldots \dashv \dashv)$

decextract:
- $(X,\vdash$ **elem** $T \vdash I_1 \dashv \dashv) \mapsto$ ⟦(⟧ ⟦extract⟧ □ *nome*(X) □ *funces*$(I_1,\vdash \dashv)$ □ *funces*$(T,\vdash \dashv)$ □ *gettypextr*(T) □ *synputget*$(X,!)$ ⟦)⟧

En **SynDEx**, *l'instruction* **extract** *n'extrait que la première dimension du tableau ; pour extraire plusieurs dimensions, par exemple* $X := TAB[I,J,K]$, *on engendre donc une cascade de trois instructions* **extract**, *la première ayant TAB et I comme entrées, la seconde la sortie du premier* **extract** *et J comme entrée et la troisième la sortie du second* **extract** *et K comme entrées. A chaque fois, le tableau TAB en entrée perd une dimension.*

gettypextr:
- $(\vdash$ **dec_sig** $DIM\ X \dashv) \mapsto$ *typextr*(DIM)

typextr:
- $(\vdash$ **t_vect** $\vdash L_1\ L_2 \ldots \dashv TEL \dashv) \mapsto$ ⟦[⟧ *deceval*$(\vdash L_2 \ldots \dashv)$ ⟦]⟧ *type*(TEL)

  $\vdash L_1\ L_2 \ldots \dashv$ *est la liste des bornes supérieures du tableau et TEL est le type de ses éléments. typextr() supprime la première dimension du type tableau passé en paramètre.*

5.7.6 Les expressions constantes

Une expression constante apparaît comme argument d'une déclaration de tableau, d'une instance de processus, d'une initialisation de retard. Une expression constante a la syntaxe d'une expression sur signaux; une expression constante est soit un littéral soit un identificateur d'expression constante soit récursivement une expression, dont les arguments sont des expressions constantes, construite avec les operateurs de [*expression sur* TEMP] , [*expression sur* VECT] .

♠ [*expression sur* TEMP] ::= **cst_ent** **cst_dp** **cst_reel**

◊ **cst_ent** ⟶ implemented as STRING

decexpr:
- (⊢ **cst_ent** N_{10} ⊣) ↦ ['] N_{10} [']

cst_ent *est une chaine de chiffres représentant un entier en base 10 directement transcrite en SynDEx.*

Constantes numériques

◊ **cst_reel** ⟶ [*constante sur* **P_REE**] [*constante sur* **REL**] $^{0||1}$

decexpr:
- (⊢ **cst_reel** E_1 E_2 ⊣) ↦ *decexpr*(E_1) [e] *decexpr*(E_2)

◊ **cst_dp** ⟶ [*constante sur* **P_REE**] [*constante sur* **REL**] $^{0||1}$

decexpr:
- (⊢ **cst_reel** E_1 E_2 ⊣) ↦ *decexpr*(E_1) [d] *decexpr*(E_2)

♠ [*constante sur* **P_REE**] ::= **cst_ent** **frac**

◊ **frac** ⟶ [**cst_ent**]$^{0||1}$ [**cst_ent**]$^{0||1}$

decexpr:
- (⊢ **frac** E_1 E_2 ⊣) ↦ *decexpr*(E_1) [.] *decexpr*(E_2)

♠ [*constante sur* **REL**] ::= **cst_ent** **n_cst_ent**

◊ **n_cst_ent** ⟶ **cst_ent**

decexpr:
- (⊢ **n_cst_ent** N_{10} ⊣) ↦ ['] [-] N_{10} [']

Constantes booléennes

♠ [*expression sur* **TEMP**] ::= **false** **true**

◊ **false** ⟶

◊ **true** ⟶

decexpr:
- (**false**) ↦ [false]

- (**true**) ↦ [true]

5.8 Les tableaux de processus

◇ **moti** ⟶ **ind_mot** [*expression sur* PROC] [ENT_MOT]*

◇ **ind_mot** ⟶ IDENT [*expression sur* TEMP]

♠ ENT_MOT ::= IDENT **prec suiv**

◇ **suiv** ⟶ IDENT IDENT

◇ **prec** ⟶ IDENT IDENT

decmoti:
- (⊢ **ind_mot** ⊢ *I BORNE* ⊣ *E* [ENT_MOT]* ⊣) ↦ *decind(I) decnode(E) decsigmoti([ENT_MOT]*) *decsortmoti(getdecl(E,!))*

decind:
- (*I*) ↦ [(index □ *nomn(I)* □ *funces*(⊢ ⊣ , *I*))]
  Ce sommet n'est engendré que si l'indice I est utilisé dans des calculs à l'intérieur du tableau de processus.

decsigmoti:
- (*E*) ↦ [(fork □ *nomf(E)* □ *funces*(*E*,⊢ ⊣) □ *gettypextr(E)* □ [! *getnom(E)*])]

  Ce sommet est engendré pour toute entrée E du tableau de processus apparaissant dans la partie WITH sous la forme: with E

- (⊢ **prec** *E INI* ⊣) ↦ [(iterate □ *nomi(E)* □ [incr] □ *gettype(E)* □ [? e] *getnom(INI)* □ [? e] *getnom(E)* □ [!] *getnom(E)*)]

  Ce sommet est engendré pour toute entrée E du tableau de processus apparaissant dans la partie WITH sous la forme: with E[0]:INI

- (⊢ **suiv** *E INI* ⊣) ↦ [(iterate □ *nomi(E)* □ [decr] □ *gettype(E)* □ [? e] *getnom(INI)* □ [? e] *getnom(E)* □ [!] *getnom(E)*)]

  Ce sommet est engendré pour toute entrée E du tableau de processus apparaissant dans la partie WITH sous la forme: with E[.]:INI

decsormoti:
- (*E*) ↦ [(join □ *nomj(S)* □ *funces*(*S*,⊢ ⊣) □ *gettypextr(S)* □ [!] *getnom(S)*)]

  Ce sommet est engendré pour toute sortie du tableau de processus.

5.9 Les expressions sur horloges rajoutées par le compilateur SIGNAL

Au cours de la construction du GCH, le compilateur **SIGNAL** engendre des nouveaux sommets d'horloges (voir le paragraphe 2). Ces sommets sont, soit des sommets `when`, soit des sommets `default`. Ils sont traduits, en **SynDEx**, par des instructions `hmul` (intersection de signaux booléens) et des instructions `hadd` (union de signaux booléens).

decexpr:

- $(H, \vdash$ **mele** $H_1\ H_2\ \dashv)$ $\mapsto$ [(hadd] $\square$ $exthnom(H)$ $\square$ [logical]
  $\square$ [?] $getnom(H_1)$ $\square$ [?] $getnom(H_2)$ $\square$ [!] $getnom(H)$ [)]

- $(H, \vdash$ **extr** $H_1\ H_2\ \dashv)$ $\mapsto$ [(hmul] $\square$ $exthnom(H)$ $\square$ [logical]
  $\square$ [?] $getnom(H_1)$ $\square$ [?] $getnom(H_2)$ $\square$ [!] $getnom(H)$ [)]

5.10 Connexions

Pour chaque signal x produit dans un sommet N , diffusé aux sommets $N_1 \ldots N_m$ où il est respectivement identifié par les signaux $x_1 \ldots x_m$ (la diffusion peut concerner plusieurs signaux d'un même sommet), la description de la diffusion est faite par le sommet **SynDEx** suivant :

[(connect] $\square$ N[/]x $\square$ $lnodesign(N_1 \ldots N_m, x_1 \ldots x_m)$ [)]

La fonction *lnodesign*() est définie par :
lnodesign:

- $(N, x) \mapsto N[/]x$

- $(N_1\ N_2 \ldots, x_1\ x_2 \ldots) \mapsto N_1[/]x_1 \square lnodesign(N_2 \ldots, x_2 \ldots)$

Il est possible d'éclater les instructions **connect** ; par exemple, l'instruction :

(connect N/x N1/x1 N2/x2)

est équivalente aux deux instructions suivantes :

(connect N/x N1/x1)
(connect N/x N2/x2)

Les références en avant ne sont pas autorisées, c'est-à-dire que toute référence à un sommet N dans une instruction **connect** doit être précédée de la définition de ce sommet.

5.11 Contrôles

Les sommets **SynDEx hinput**, ainsi que les sommets **hadd** et **hmul** qui proviennent de **default** et de **when** entre signaux dont les horloges ne dépendent d'aucune horloge commune, constituent les racines de la hiérarchie des conditionnements et sont regroupés dans une instruction execroots :

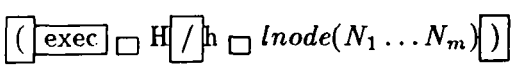
[(execroots] $\square$ $lnode(N_1 \ldots N_m)$ [)]

La fonction *lnode*() est définie par :
lnode:

- $(N) \mapsto N$

- $(N_1\ N_2 \ldots) \mapsto N_1 \square lnode(N_2 \ldots)$

Pour chaque horloge h , produite dans un sommet H , qui est l'horloge d'un ensemble de sommets $N_1 \ldots N_m$, la description du contrôle d'activation est faite par le sommet **SynDEx** suivant :



On retrouve ici la hiérarchie des horloges sous la forme d'une hiérarchie des booléens de contrôle.

6 Instructions SIGNAL non interprétables en SynDEx

♠ [*expression sur* PROC] ::= **ferm**

◊ **ferm** ⟶ [*expression sur* PROC] [IDENT]$^+$

L'opérateur **ferm** n'existe que pour des horloges récursivement définies : la génération **SynDEx** serait alors incorrecte.

♠ [*expression sur* TEMP] ::= **conc**

[*expression sur* TEMP] est utilisée pour les paramètres d'instances, les initialisations de retard et les parties droites d'affectation.

♠ [*expression sur* VECT] ::= **conc** [ELE_VEC]$^+$ **elem** IDENT

◊ **conc** ⟶ [*expression sur* VECT] [*expression sur* VECT]

Actuellement, la concaténation de vecteurs n'est pas traduite en **SynDEx** .

♠ [*expression sur* TEMP] ::= **nomb** **nomb_a** **nomb_d**

◊ **nomb_d** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◊ **nomb_a** ⟶ [*expression sur* TEMP] [*expression sur* TEMP]

◊ **nomb** ⟶ [*expression sur* TEMP]

Actuellement, les compteurs ne sont pas décompilés en **SynDEx**.

♠ I_TYPE ::= **complex**

◊ **complex** ⟶ implemented as SINGLETON

♠ [*expression sur* TEMP] ::= **cst_comp**

◊ **cst_comp** ⟶ [*constante sur* P_COM] [*constante sur* P_COM]

♠ [*constante sur* P_COM] ::= [*constante sur* REL] **n_cst_ree** **cst_reel**

♠ [*constante sur* REE] ::= **cst_reel**

◇ **n_cst_ree** ⟶ [*constante sur* REE]

Le type complexe n'existe pas en **SynDEx** .

7 SIGNAL non basique

On énumère ici pour mémoire, les primitives de **SIGNAL qui sont traduites en SIGNAL** basique.

♠ [*expression sur* PROC] ::= **modi_e modi_s obli vali para$^+$ seri$^+$ cond**

◇ **seri$^+$** ⟶ [*expression sur* PROC] *

◇ **para$^+$** ⟶ [*expression sur* PROC] *

◇ **vali** ⟶ [*expression sur* PROC] [IDENT]$^+$

◇ **obli** ⟶ [*expression sur* PROC] [IDENT]$^+$

◇ **modi_s** ⟶ [*expression sur* PROC] [**modi**]$^+$

◇ **cond** ⟶ [*expression sur* TEMP] [*expression sur* PROC] $^{0\|1}$ [*expression sur* PROC] $^{0\|1}$

◇ **modi_e** ⟶ [*expression sur* PROC] [**modi**]$^+$

◇ **modi** ⟶ IDENT IDENT

♠ TYPE ::= **implicit**

◇ **implicit** ⟶ implemented as SINGLETON

◇ **meta** ⟶

◇ **comment_s** ⟶ [**comment**]*

◇ **comment** ⟶

8 Exemples de transformations SIGNAL-SynDEx

Pour chaque exemple, on donne la spécification en langage **SIGNAL**, puis sa traduction en instructions **SynDEx**, puis la topologie du graphe flot de données correspondant.

Le premier exemple "compteur" est le cas d'école représentatif de la classe des processus ayant plusieurs horloges racines de la hiérarchie des horloges. Les autres exemples déclinent les cas de graphes à motif répétitif et n'ont tous qu'une seule horloge racine dont le sommet n'est pas représenté dans la topologie du graphe correspondant.

8.1 Compteur

L'exemple suivant est un compteur avec remise à zéro qui présente tous les cas d'instructions, excepté celles concernant les signaux de type tableau et les motifs répétitifs.

Programme SIGNAL

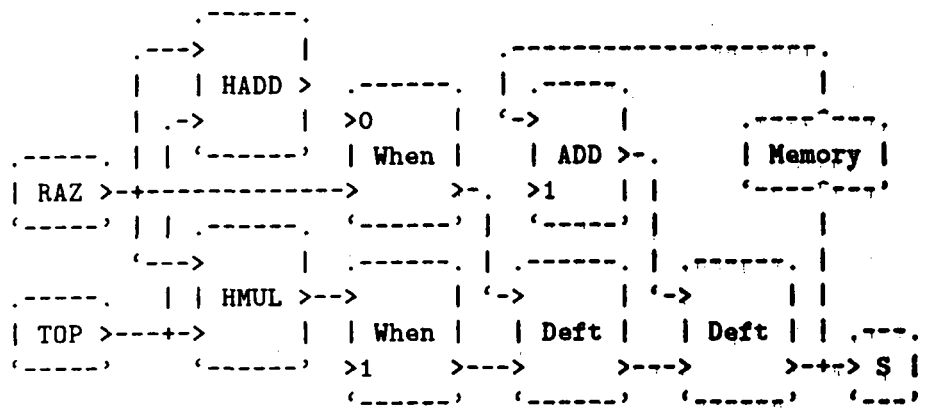
```
process cpt = { ? event top, raz ! integer s }
(| s := (1 when top when raz) default (0 when raz) default (zs + 1)
 | zs := s $ 1
 | synchro{top default raz, s}
 |)
where integer zs init 0
end
```

Graphe Conditionné Hiérarchisé GCH.sig

```
process CPT_TRA= { ? event TOP_1, RAZ_2 ! integer S_3 }
(| H_7_H:= RAZ_2 when TOP_1
 | H_9_H:= when( ( not H_7_H )default RAZ_2 )
 | H_13_H:= when( ( not RAZ_2 )default TOP_1 )
 | H_18_H:= RAZ_2 default TOP_1
 | synchro { H_18_H, S_3 }
 | H_18_H( )
 |)
where  event H_18_H, H_13_H, H_9_H, H_7_H
  process H_18_H= { ? event H_18_H, H_13_H, H_9_H, H_7_H ! integer S_3 }
  (| synchro { H_18_H, ZS_5 }
   | ZS_5:= S_3 $1
   | S_3:= (1 when H_7_H) default (0 when H_9_H) default (ZS_5+1 when H_13_H)
   |)
  where integer ZS_5 init 0
  end
end
```

Traduction en SynDEx GCH.syn

```
(hinput TOP1 RTOP logical !top)
(hinput RAZ2 RRAZ logical !raz)
(function WS3 WS 10 integer ?s)
(hmul H7H  logical ?raz ?top !h7h)
(hadd H18H logical ?raz ?top !h18h)
(memory ZS5 integer ?s $1 !zs init 0)
(when X6X logical ?h7h, integer ?'1' !x6x)
(default X7X integer ?x6x ?'0' !x7x)
(function X8X add 10 integer ?zs ?'1' !x8x)
(default S3 integer ?x7x ?x8x !s)
(connect TOP1/top H7H/top H18H/top)
(connect RAZ2/raz H7H/raz H18H/raz)
(connect H7H/h7h X6X/h7h)
(connect ZS5/zs X8X/zs)
(connect X6X/x6x X7X/x6x)
(connect X7X/x7x S3/x7x)
(connect X8X/x8x S3/x8x)
(connect S3/s ZS5/s WS3/s)
(execroots TOP1 RAZ2 H7H H18H)
(exec H7H/h7h X6X)
(exec RAZ2/raz X7X)
(exec H18H/h18h WS3 ZS5 X8X S3)
```

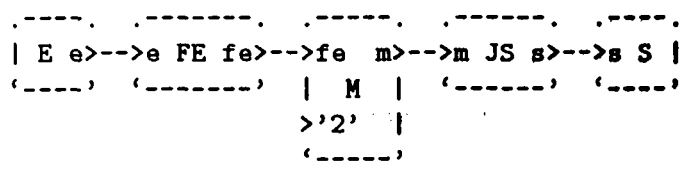


8.2 Multiplication d'un vecteur par une constante

Un exemple simple avec seulement un **fork** et un **join**:

```
process vecMul2 = { ? [5]integer e ! [5]integer s }
(| array i to 5 of  s := 2*e  with e end
 |)
end

(hinput Hroot RHroot logical !h)
(function E RE 10 integer !e)
(function S WS 10 integer ?s)
(fork FE [5]integer ?e, integer !fe)
(function M mul 10 integer ?fe ?'2' !m)
(join JS integer ?m, [5]integer !s)
(connect E/e FE/e) (connect FE/fe M/fe)
(connect M/m JS/m) (connect JS/s S/s)
(execroots Hroot)
(exec Hroot/h E {FE M JS} S)
```

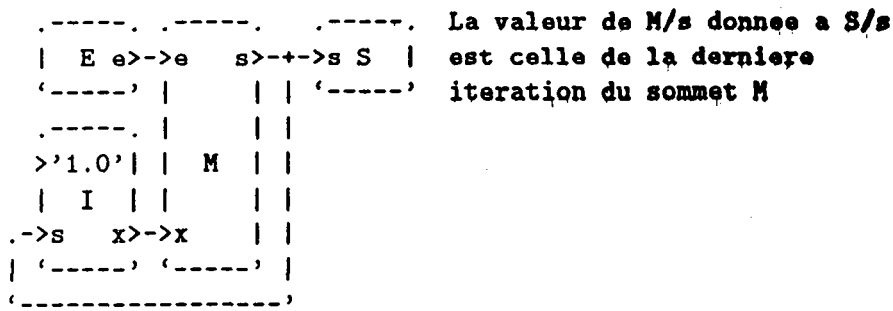


8.3 Calcul de la puissance 9ème de la valeur d'un signal

Un exemple simple avec seulement un sommet **iterate (croissant)**:

```
process power9 = { ? real e ! real s }
(| un := 1.0
 | array i to 9 of  x := x * e  with x[0]:un end
 | s := x[9]
 |)
where  [9]real x
end

(hinput Hroot RHroot logical !h)
(function E RE 10 real !e)
(function S WS 10 real !s)
(function M mul 10 real ?e ?x !s)
(iterate I incr 9 real ?'1.0' ?s !x)
(connect E/e M/e)
(connect I/x M/x)
(connect M/s I/s S/s)
(execroots Hroot)
(exec Hroot/h E {M I} S)
```

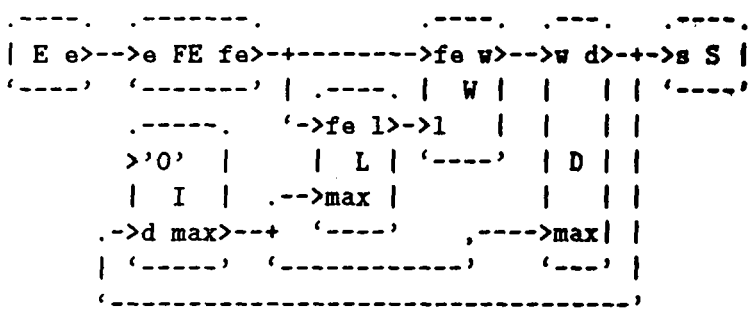


8.4 Extraction de l'élément positif maximum d'un tableau

Un exemple avec **fork**, **iterate** (décroissant) et **conditionnement** :

```
process vecMax = { ? [8]integer e ! integer s }
(| zero := 0
 | array i to 8 of  max := e when e>max default max  with max[.]:zero, e end
 | s := max[1]
 |)
where  integer zero; [8]integer max
end

(hinput Hroot RHroot logical !h)
(function E RE 10 [8]integer !e)
(function S WS 10 integer !s)
(fork FE [8]integer ?e, integer !fe)
(function L less 10 integer ?max ?fe, logical !l)
(when W logical ?l, integer ?fe !w)
(default D integer ?w ?max !d)
(iterate I decr 8 integer ?'0' ?d !max)
(connect E/e FE/e) (connect FE/fe L/fe W/fe)
(connect L/l W/l) (connect W/w D/w)
(connect D/d I/d S/s) (connect I/max L/max D/max)
(execroots Hroot)
(exec Hroot/h E {FE L D I} S) (exec L/l W)
```



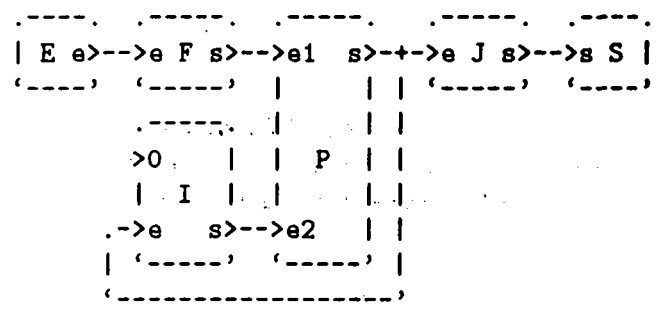
La valeur D/d donnee a S/s est celle de la derniere iteration du sommet D

8.5 Histogramme des valeurs d'un tableau

Un exemple simple avec **fork**, **join** et **iterate**:

```
process histo = { ? [3]integer e ! [3]integer s }
(| zero := 0
 | array i to 3 of s := s + e with s[0]:zero, e end
 |)
where  integer zero
end

(hinput Hroot RHroot logical !h)
(function E RE 10 [3]integer ?e)
(function S WS 10 [3]integer !s)
(fork F [3]integer ?e, integer !s)
(function P add 10 integer ?e1 ?e2 !s)
(iterate I incr 3 integer ?'0' ?e !s)
(join J integer ?e, [3]integer !s)
(connect E/e F/e) (connect F/s P/e1)
(connect P/s I/e J/e) (connect I/s P/e2) (connect J/s S/s)
(execroots Hroot)
(exec Hroot/h E {F P I J} S)
```

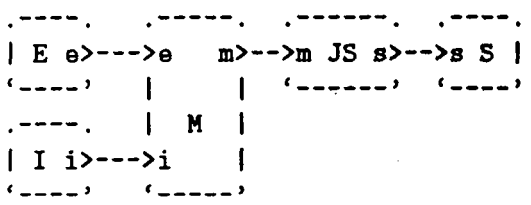


8.6 Génération d'un vecteur de multiples de l'index d'itération

Un exemple simple avec un **index** et un **join**:

```
process mulVecI = { ? integer e ! [4]integer s }
(| array i to 4 of  s := i*e  with s end
 |)
end

(hinput Hroot RHroot logical !h)
(function E RE 10 integer !e)
(function S WS 10 [4]integer ?s)
(index I integer !i)
(function M mul 10 integer ?i ?e !m)
(join JS integer ?m, [4]integer !s)
(connect E/e M/e) (connect I/i M/i)
(connect M/m JS/m) (connect JS/s S/s)
(execroots Hroot)
(exec Hroot/h E {I M JS} S)
```



8.7 Tri par insertion

Un exemple avec motifs imbriqués :

```
process triInsert = (integer N) { ? [N]integer e ! [N]integer s }
(| array i to N of
     array j to N of
     (| c := zy>y and j<i
      | zys := zy when c default y
      | ys := y when c or j>i default zy
      |) ! zys:zy, ys:y
     with y, zy[0]:e end
   with y[0]:e, e end
 | s := y[N]
 |)
where [N,N]logical c; [N,N]integer y, zy
end
```

Traduction SynDEx pour N=5 :

```
(hinput Hroot RHroot logical !h)
(function E RE 10 [5]integer !e)
(function S WS 10 [5]integer ?s)
(fork FE [5]integer ?e, integer !fe)
(iterate Y incr 5 [5]integer ?e ?iy !y) (index I integer !i)
(fork XY [5]integer ?y, integer !xy)
(iterate ZY incr 5 integer ?fe ?zys !zy) (index J integer !j)
(function L1 less 10 integer ?xy ?zy, logical !l1)
(function L2 less 10 integer ?j ?i, logical !l2)
(function C logand 10 logical ?l1 ?l2 !c)
(when W1 logical ?c, integer ?zy !w1)
(default ZYS integer ?w1 ?xy !zys)
(function L3 less 10 integer ?i ?j, logical !l3)
(function L4 logor 10 logical ?c ?l3 !l4)
(when W2 logical ?l4, integer ?xy !w2)
(default YS integer ?w2 ?zy !ys)
(join IY integer ?ys, [5]integer !iy)
(connect E/e FE/e Y/e) (connect FE/fe ZY/fe) (connect Y/y XY/y)
(connect XY/xy L1/xy ZYS/xy W2/xy) (connect I/i L2/i L3/i)
(connect ZY/zy L1/zy W1/zy YS/zy) (connect J/j L2/j L3/j)
(connect L1/l1 C/l1) (connect L2/l2 C/l2) (connect C/c W1/c L4/c)
(connect W1/w1 ZYS/w1) (connect ZYS/zys ZY/zys)
(connect L3/l3 L4/l3) (connect L4/l4 W2/l4) (connect W2/w2 YS/w2)
(connect YS/ys IY/ys) (connect IY/iy Y/iy S/s)
(execroots Hroot)
(exec Hroot/h E {FE Y {XY ZY L1 L2 C ZYS L3 L4 YS IY}} S)
(exec C/c W1) (exec L4/l4 W2)
```

8.8 Fonction de recentrage de l'énergie de la réponse d'un filtre

Un exemple avec motifs successifs, index et extract:

```
process recentrage = (integer N) { ? [N]real e ! [N]real s }
(| zero := 0.0
 | array i to N of % calcul de l'energie cumulee de e %
     energie := energie + e*e
   with energie[0]:zero, e end
 | energieMoyenne := energie[N]/2.0
 | un := 1
 | array i to N of % recherche de l'index de l'energie moyenne %
     index := (i when energie<energieMoyenne) default index
   with index[0]:un, energie end
 | di := N/2 - index[N] % recentrage autour de l'energie moyenne %
 | s := [{i to N}: (0.0 when i<=di or i>N+di) default e[i-di]]
%----------
  Remarque : l'expression ci-dessus est equivalente \'{a} :
  | array i to N of s := (0.0 when i<=di or i>N+di) default e[i-di] with s end
----------%
 |)
where
  real zero, energieMoyenne; [N]real energie;
  integer un, di; [N]integer index
end
```

Traduction SIGNAL/SynDEx pour N=5:

```
(hinput Hroot RHroot logical !h)
(function E RE 10 [5]real !e)
(function S WS 10 [5]real !s)

(fork FE [5]real ?e, real !s)
(function M mul 10 real ?e1 ?e2 !s)
(function A add 10 real ?e1 ?e2 !s)
(iterate I1 incr 5 real ?'0.0' ?e !s)
(join IE real ?e, [5]real energie)
(function EM div 10 real ?e ?'2.0' !energieMoyenne)

(fork FEN [5]real ?energie, !e)
(function L less 10 real ?e ?energieMoyenne, logical !b)
(index II2 integer !ii2)
(when W2 logical ?b, integer ?ii2 !iw)
(default D2 integer ?iw ?indexp !index)
(iterate I2 incr 5 integer ?'1' ?index !indexp)

(function DI sub 10 integer ?"5/2"'2' ?index !di)
(function NDI add 10 integer ?'5' ?di !ndi)
(index II3 integer !ii3)
```

```
(function NL notless 10 integer ?di ?ii3, logical !nl)
(function LL less 10 integer ?ndi ?ii3, logical !ll)
(function OR logor 10 logical ?nl ?ll !or)
(when W3 logical ?or, real ?'0.0' !w3)
(function SI sub 10 integer ?ii3 ?di !index)
(extract EX integer ?index, [5]real ?e, !real ex)
(default D3 real ?w3 ?ex !d3)
(join IS real ?d3, [5]real !s)

(connect E/e FE/e EX/e)
(connect FE/s M/e1 M/e2) (connect M/s A/e1)
(connect A/s I1/e D/e) (connect I1/s A/e2) (connect I1/s IE/e)
(connect IE/energie FEN/energie) (connect EM/energieMoyenne L/energieMoyenne)
(connect FEN/e L/e) (connect L/b W/b)
(connect II2/ii2 W/ii2) (connect W/iw D/iw)
(connect D2/index I2/index DI/index) (connect I2/indexp D2/indexp)
(connect DI/di NDI/di NL/di SI/di) (connect NDI/ndi LL/ndi)
(connect II3/ii3 NL/ii3 LL/ii3 SI/ii3)
(connect NL/nl OR/nl) (connect LL/ll OR/ll)
(connect OR/or W3/or) (connect W3/w3 D3/w3)
(connect SI/index EX/index) (connect EX/ex D3/ex)
(connect D3/d3 IS/d3) (connect IS/s S/s)

(execroots Hroot)
(exec Hroot/h E {FE M A I1 IE} EM {FEN L D2 I2} DI NDI
               {II3 NL LL OR SI EX D3 IS})
(exec L/b W2 II2) (exec OR/or W3)
```

Références

[1] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire,
"Programming Real-Time Applications with **SIGNAL**", Proceedings of the IEEE, volume 79, numéro 9, pages 1321-1336, septembre 1991.

[2] L. Besnard,
"Compilation de Signal : horloges, dépendances, environnement"
Thèse de l'Université de Rennes 1, France, septembre 1992.

[3] P. Bournai, B. Chéron, T. Gautier, B. Houssais, P. Le Guernic,
"SIGNAL Manual"
Publication interne IRISA n°745, Rennes, France, juillet 1993.

[4] C. Lavarenne, Y. Sorel,
"Performance Optimization of Multiprocessor Real-Time Applications by Graph Transformations"
Proceedings Parallel Computing 93, Grenoble, septembre 1993.

[5] C. Lavarenne, C. Milan, M. Paindavoine, G. Richard, Y. Sorel,
"Implantation d'algorithmes de traitement d'images sur une architecture multi-DSP avec l'environnement d'aide à l'implantation SynDEx"
Actes Quatorzième Colloque GRETSI, Juan-Les-Pins, septembre 1993.

[6] C. Lavarenne, Y. Sorel,
"Optimisation et génération d'exécutifs distribués remps réel pour algorithmes spécifiés avec les langages Synchrones"
Actes Real-Time Systems, Paris, janvier 1994.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399

RR-2206