Pascal AUBRY
IRISA, Pascal.Aubry@irisa.fr

Sylvain MACHARD
INRIA/IRISA, Sylvain.Machard@irisa.fr

Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu
35042 Rennes Cedex

1 INTRODUCTION

Basés sur l'hypothèse d'un temps logique discret, les langages synchrones (SIGNAL [11, 16], LUSTRE [9, 13], ESTEREL [4, 8]) ont prouvé leur efficacité pour la conception d'application temps réel critiques sûres. Ils sont caractérisés par une sémantique forte offrant des techniques de vérification et de preuve.

SIGNAL, un des langages synchrone [2, 3], est déclaratif et orienté flot de données. Dans ce langage, un *signal* est une succession de valeurs ; les instants de présence de ce signal sont appelés *horloges*. Tout au long de la conception, jusqu'à la phase finale de compilation [1, 17], le compilateur SIGNAL [5] utilise un seul formalisme appelé *Graphe Flot de Données Synchronisé*[1].

Pour différentes raisons, telles la délocalisation géographique de capteurs, l'accélération de la fréquence de réponse ou une plus grande tolérance aux fautes, les applications temps-réel nécessitent une production de code distribué. Nous allons présenter une méthode de distribution fonctionnelle des programmes, et prouver qu'elle peut se faire par transformations successives du GFDS, ce qui permet de conserver ce formalisme jusqu'à la phase finale de génération de code distribué.

Après une rapide présentation du langage SIGNAL en nous intéressant plus particulièrement au GFDS, nous allons donner une méthode de propagation de directives de distribution du code source jusqu'au GFDS. Nous allons ensuite montrer comment les communications peuvent être introduites dans le GFDS, et comment la partie "contrôle" des programmes SIGNAL peut être distribuée pour obtenir sur chaque processeur, un sous-graphe indépendant des autres. Finalement, nous montrerons des implémentations possibles et des perspectives de ce travail.

2 SIGNAL

2.1 Le langage SIGNAL

SIGNAL est un langage synchrone, de style déclaratif, orienté flot de données. Un programme SIGNAL décrit des processus communiquant à travers des suites de valeurs typées indicées par le temps : *les signaux*. Un signal `X` dénote la succession $(\mathtt{x}_t)_{t \in I\!N_{\setminus\{0\}}}$.

Noyau de SIGNAL

Le noyau du langage SIGNAL [7] inclu des opérateurs sur les signaux et des opérateurs sur les processus. Quatre types d'opérateurs agissent sur les signaux :

- les **fonctions instantanées** regroupent toutes les fonctions arithmétiques et logiques usuelles, étendues aux signaux. Si le symbole f dénote une fonction n-aire sur des signaux et $[\![f]\!]$ la fonction correspondante agissant sur les valeurs. Le processus SIGNAL `Y := ` f`{X1,..,XN}` indique que
$$\forall t \geq 1 \;\; \mathtt{y}_t = [\![f]\!](\mathtt{x1}_t, ..., \mathtt{xn}_t)$$
Ces fonctions imposent que tous les opérateurs soient présents au même instant.
- les **décalages temporels**, `ZX := X $ 1` ou `ZX` est le signal de même horloge que `X`, dont la valeur à l'instant t est celle de `X` à l'instant $t-1$.
$$\mathtt{ZX}_0 = \mathtt{v0}, \quad \forall t \geq 1 \;\; \mathtt{ZX}_t = \mathtt{X}_{t-1}$$
`v0` est une valeur initiale constante associée à la déclaration de `ZV`.
- le **sous-échantillonnage** `Y := X when C` spécifie que `Y` à la valeur de `X` quand `X` est présent, et quand `C` est présent et a pour valeur *vraie*.
- le **mélange prioritaire** `X default Y`, où `X` et `Y` sont des signaux de même type, correspond à une combinaison des flots de données. C'est le signal égal à `X` quand `X` est présent, égal à `Y` sinon.

Les quatre opérateurs précédents spécifient les processus de base. La spécification de processus plus complexes est réalisé grâce à l'opérateur de composition parallèle : la composition de deux processus `P1` et `P2` est notée :

`(| P1 | P2 |)`

Dans cette composition, les noms communs à `P1` et `P2` réfèrent les mêmes signaux, ce sont les liens de communication entre `P1` et `P2`. La composition parallèle est un opérateur associatif, commutatif et idempotent.

Le dernier opérateur du noyau permet de réduire la **portée d'un signal** : dans le processus `P()/X`, le signal `X` est local à `P`.

---

[1] appelé par la suite GFDS

Ces opérateurs, souvent utilisés par le programmeur, sont construits sur les opérateurs de base.

- l'**extraction d'horloge** `C := event X` rend explicite l'horloge d'un signal, il est équivalent à

  ```
  C := (X=X)
  ```

- la **synchronisation** entre signaux induit une nouvelle contrainte au programme. `synchro{X, Y}` impose à `X` et `Y` d'avoir les mêmes horloges. Ce processus est équivalent à

  ```
  (| C := (event X) = (event Y) |) /C
  ```

- la **mémorisation** permet au programmeur de conserver la dernière valeur d'un signal aux occurrence *vraie* d'un booléen. `X cell B`, où `B` est un flot booléen, est le signal d'horloge `(event X) default (when B)` ayant pour valeur celle de `X` si `X` est présent, sinon la dernière valeur de `X`. Le processus est équivalent à :

  ```
  (|synchro{Y,(event X) default (when B)}
   | ZX := X $ 1
   | Y  := X default ZX
   |) /ZX
  ```

La spécification d'un programme SIGNAL est indépendant de l'architecture. Cette liberté provient de l'approche synchrone et du style équationnel/flot de donnée du langage. La section suivante décrit le processus de compilation et la représentation finale : le GFDS. Par la suite, nous allons montrer comment des transformations de ce graphe peuvent mener à une implémentation distribuée.

2.2 Graphe Flot-de-Données Synchronisées

Puisque SIGNAL est équationnel, le compilateur n'est pas un simple traducteur d'un code de haut niveau en un code exécutable. C'est un système formel capable de raisonner sur la logique et les horloges.

- La première étape de la compilation consiste à réécrire le programme source dans le langage noyau.

- Des nœuds horloge sont alors créés, rassemblant tous les signaux synchrones d'un programme. Une horloge h peut être :

  - l'**horloge d'un signal** `X`, elle est alors notée $\hat{\mathtt{X}}$ indiquant les instants de présence du signal `X`.
  - l'**échantillonnage** positif (resp. négatif), noté $[b]$ (resp. $[\neg b]$), d'un signal booléen `b`, indiquant les instants ou `b` est présent, et a pour valeur *vrai* (resp. *faux*).
  - la **borne supérieure** (resp. **inférieure**) de deux horloges h_1 et h_2, notée $h_1 \vee h_2$ (resp. $h_1 \wedge h_2$), indique les instants où l'une ou l'autre (resp. les deux) horloges sont présentes.
  - la **complémentaire** d'une horloge h_1 dans une horloge h_2, notée $h_2 \ominus h_1$, indiquant les instants ou h_2 est présente et h_1 absente.

  cul d'horloge. Par analyse des horloges du programme, il y a construction *d'arbres d'horloges* en plaçant une horloge sous son horloge-mère[2]. Le résultat du calcul d'horloge est une forêt d'arbre dans lequel les horloges racines sont mutuellement indépendantes. Si une seule horloge racine est présente, la forêt est réduite à un seul arbre ; ceci signifie qu'il existe une horloge plus rapide que toutes les autres. Pendant l'analyse, les cycles dans la définition d'horloges sont détectés et les contraintes d'horloges sont établies.

- à une horloge h sont accrochées des *instructions* (équations SIGNAL) explicitant les calculs à effectuer lorsque h est présente. Une instruction peut être :

  - une **définition de signal**, de la forme `X :=` *exp*, où *exp* est une expression sur signaux.
  - l'**appel d'une fonction externe** `F`.
  - un **retard**.

Ces instructions induisent des dépendances entre les signaux. Puisque les signaux ne sont pas toujours présents, les dépendances sont étiquetées par des horloges : une dépendance entre deux signaux `X` et `Y` est effective seulement aux instants de leur horloge de dépendance h :

$$\mathtt{X} \xrightarrow{h} \mathtt{Y}$$

L'horloge h est incluse dans $\hat{\mathtt{X}}$ et $\hat{\mathtt{Y}}$, ce qui signifie que la dépendance est effective seulement si les deux signaux `X` et `Y` sont présents. Deux propriétés appliquées sur les dépendances, caractérisent la sérialisation :

$$\mathtt{X} \xrightarrow{h_1} \mathtt{Y} \xrightarrow{h_2} \mathtt{Z} \Longrightarrow \mathtt{X} \xrightarrow{h_1 \wedge h_2} \mathtt{Z}$$

et le parallélisme :

$$\left.\begin{array}{l} \mathtt{X} \xrightarrow{h_1} \mathtt{Y} \\ \mathtt{X} \xrightarrow{h_2} \mathtt{Y} \end{array}\right\} \Longrightarrow \mathtt{X} \xrightarrow{h_1 \vee h_2} \mathtt{Z}$$

La fermeture transitive sur les dépendances du graphe révèle des circuits.

$$\mathtt{X}_1 \xrightarrow{h_1} \mathtt{X}_2 \xrightarrow{h_2} \cdot \xrightarrow{h_{n-1}} \mathtt{X}_n \xrightarrow{h_n} \mathtt{X}_1$$

en appliquant la propriété de sérialisation, l'horloge de ce circuit est $h = \bigwedge_i h_i$; si $h = \emptyset$ (horloge vide), le circuit n'est jamais effectif et l'ordonnancement du calcul dépend des horloges h_i. De tels circuits sont rejetés à la compilation, à cause du coût d'analyse.

Dans cette rapide description, nous voyons les deux aspects du GFDS : une hiérarchie d'horloge pouvant être utilisée pour réduire le contrôle des calculs, et un graphe flot de données dans lequel les dépendances des signaux sont étiquetées par des horloges.

3 DIRECTIVES DU CODE SOURCE AU GFDS

Notre motivation n'est pas une distribution quantitative [14] mais fonctionnelle des programmes, nous cherchons à implémenter des programmes en fonction de contraintes de localisation des instructions. Ces contraintes de localisation seront introduites par des directives dans le code source, par cette méthode, un seul formalisme est vu par l'utilisateur. Le lecteur

---

[2] Une horloge h ne peut être présente si sa mère ne l'est pas

GNAL peuvent se réferer à [6, 14].

Un des intérêts de SIGNAL est la possibilité de définir un modèle de processus pouvant être instancié plusieurs fois. Au moment de la compilation, ces modèles de processus sont expansés, notre problème est donc de propager les directives du code source jusqu'au GFDS.

3.1 Directives de distribution

Puisque ces directives sont des informations utilisées seulement pour la génération de code distribué, nous les représenterons sous la forme de pragmas tels qu'ils sont définis dans le manuel de référence de SIGNAL V4 [12]. Les pragmas de topologie sont utilisés seulement par le processus principal, si de tels pragmas existent dans un sous-processus, ils sont ignorés.

Le premier pragma que nous introduisons, appelé *pragma de topologie* (**Topology_PRAGMA**), est utilisé pour décrire le système sur lequel le programme sera exécuté. Sa syntaxe est :

♠ **Topology-PRAGMA** ::=
- $\boxed{\textbf{topology}}$ $\boxed{”}$ *O.S.* $\boxed{,}$ *machine* $\boxed{,}$ *alias* $\boxed{”}$ •
- $\boxed{\textbf{topology}}$ $\boxed{”}$ *processor* $\boxed{”}$ •
- $\boxed{\textbf{topology}}$ $\boxed{\{}$ *signal*-**name** $\boxed{,}$ ... $\boxed{\}}$ $\boxed{”}$ *processor* $\boxed{”}$ •

- La première forme spécifie les processeurs sur lesquels le programme sera exécuté.

- La seconde forme spécifie où l'horloge principale du programme devra être lue. Si ce pragma est absent, le processeur choisit est le premier déclaré.

- La dernière forme spécifie la localisation des signaux d'entrée et de sortie du programme. Par défaut, les signaux d'entrée et de sortie sont localisés sur le premier processeur déclaré.

Les *pragmas d'exécution* (**Run_on_PRAGMA**) définis par :

♠ **Run_on-PRAGMA** ::=
• $\boxed{\textbf{run\_on}}$ $\boxed{\,}$ $\boxed{\{}$ *equation*-**Label** $\boxed{,}$ ... $\boxed{\}}$ $\boxed{\,}$ $\boxed{”}$ *processor* $\boxed{,}$ ... $\boxed{”}$ •

permettent à l'utilisateur d'assigner une instruction sur un processeur. Sans labels, ce pragma spécifie qu'un processus s'exécutera toujours sur le même processeur.

En l'absence de ces pragmas, le processus s'exécutera sur le processeur spécifié lors de l'instanciation. Si le processus principal d'un programme n'est pas assigné sur un processeur avec cette commande, il s'exécutera sur le premier défini.

Nous voulons offrir la possibilité de localiser certaines instructions d'un processus sur des processeurs autres que celui de localisation l'instanciation. Pour cela, nous allons introduire la notion de *paramètre de localisation.* La définition d'un pragma d'exécution servant à localiser un sous-processus est étendue pour permettre de spécifier la liste des *processeurs-paramètres.* Au sein du sous-processus, ces paramètres sont accessibles par `#n`, ou `n` est un entier positif désignant le numéro d'ordre du paramètre dans la liste. Le processeur d'instanciation des sous-processus peut ainsi être désigné par `#0`. Si dans un pragma d'exécution, le numéro de paramètre désigné est supérieur au nombre de paramètres pour l'instanciation, le processeur par défaut est assigné.

Lors de la compilation, des signaux intermédiaires sont introduits ; si les entrées des appels externes ne sont pas des signaux, ils sont remplacées par des signaux temporaires, les opérandes polychrones d'un opérateur monochrone et les opérandes polychrones de l'opérateur **when** sont remplacés de la même façon. Si l'utilisateur désire placer l'une de ces évaluations sur un processeur différent, il peut le faire explicitement. Nous imposons donc à ces signaux d'être exécutés sur le même processeur que l'instruction de départ.

Le calcul des horloges doit aussi être placé sur un processeur, mais cette opération ne peut être faite syntaxiquement. En effet, un principe du calcul d'horloge est l'unification : lorsque le compilateur détermine que deux horloges ont les mêmes instants de présence, elles sont unifiées. Des conflits peuvent donc apparaître au moment de l'unification si les horloges ont étés placées sur des processeurs différents. Donc, l'assignation des horloges est faite lors de la production de la hiérarchie.

L'horloge principale étant lue, elle doit être placée manuellement par l'utilisateur. L'algorithme d'assignation que nous avons choisit est le suivant :

- Si h est conditionnée par un signal booléen `B` ($h :=$ `when B` ou $h :=$ `when not B`), h est placée sur le même processeur que `B`.

- Si h est la borne supérieure ($h = h_1 \vee h_2$) ou inférieure ($h = h_1 \wedge h_2$) d'horloges, si h_1 et h_2 sont placées sur le même processeur, h est aussi placé sur ce processeur, sinon, elle est placée sur le processeur de son horloge mère dans la hiérarchie.

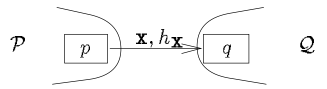
- Si h est le complémentaire d'une autre horloge h', h est placé sur le même processeur que h'.

4 COMMUNICATIONS DANS LE GFDS

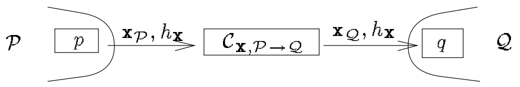
Nous supposons dans cette section que tous les nœuds (horloges et signaux) du graphe ont étés assignés sur un processeur. La méthode que nous proposons consiste à introduire des noéuds de communication des données, extraire les sous-graphes indépendants et purifier la hiérarchie.

4.1 Communications entre deux nœuds

Considérons un signal $\mathtt{x}$ produit par un nœud p sur un processeur $\mathcal{P}$. Ce signal est consommé par un nœud q sur un processeur $\mathcal{Q}$ à une horloge $h_{\mathtt{x}}$ inclue dans $\hat{\mathtt{x}}$.

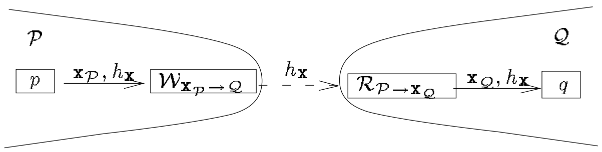


Si $\mathcal{P}$ et $\mathcal{Q}$ sont deux processeurs différents, il est nécessaire de communiquer $\mathtt{x}$ du processeur $\mathcal{P}$ au processeur $\mathcal{Q}$. Nous représentons cette communication en introduisant un *nœud de communication* $\mathcal{C}_{\mathtt{x},\mathcal{P}\to\mathcal{Q}}$. Pour ne pas avoir de conflit de notation des signaux, le signal $\mathtt{x}$, produit par le nœud p est nommé $\mathtt{x}_{\mathcal{P}}$ et un nouveau signal $\mathtt{x}_{\mathcal{Q}}$, produit par le nœud, est introduit :



mage de flot.

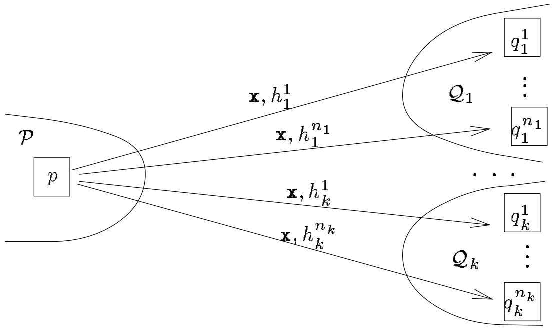
Le nœud de communication introduit précédemment peut être coupé en deux, un nœud d'écriture $\mathcal{W}_{\mathbf{x}_{\mathcal{P}} \to \mathcal{Q}}$ sur le processeur $\mathcal{P}$ et un nœud de lecture $\mathcal{R}_{\mathcal{P} \to \mathbf{x}_{\mathcal{Q}}}$ sur le processeur $\mathcal{Q}$.



Pour garantir la dépendance entre les nœuds p et q, il est nécessaire d'introduire une dépendance à l'horloge $h_{\mathbf{x}}$ entre les nœuds $\mathcal{W}_{\mathbf{x}_{\mathcal{P}} \to \mathcal{Q}}$ et $\mathcal{R}_{\mathcal{P} \to \mathbf{x}_{\mathcal{Q}}}$.

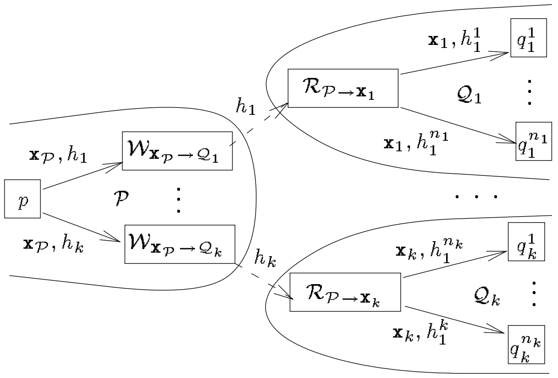
4.2 Communications dans un graphe complet

Dans cette sous-section, nous allons montrer comment le principe décrit précédemment peut être étendu à un graphe complet tel que celui-ci :



Pour cela, nous allons considérer le cas général d'un nœud p sur un processeur $\mathcal{P}$ produisant un signal $\mathbf{x}$ consommé par différents nœuds localisés sur un ensemble de processeurs $\mathcal{Q}_{i \in \{1..k\}}$. La difficulté introduite pour la transformation d'un tel sous-graphe est induite par la présence de plusieurs horloges. Un choix possible pour préserver les dépendances aux bonnes horloges, est d'introduire une communication pour chaque processeur $\mathcal{Q}_i$. La dépendance d'horloge entre le nœud p et chaque nœud $\mathcal{C}_{\mathbf{x}, \mathcal{P} \to \mathcal{Q}}$, est $h_i = \bigvee_{j=1}^{n_i} h_i^j$.

Comme dans la section 4.1, les nœuds de communication peuvent être vus comme étant de simple renommage (de $\mathbf{x}_{\mathcal{P}}$ vers $\mathbf{x}_{\mathcal{Q}}$). L'introduction des nœuds de lecture/écriture est alors sans problèmes grâce à la dépendance entre un nœud de lecture et le nœud d'écriture associé :



chaque arête d'un graphe SIGNAL. Le résultat est un nouveau graphe dans lequel les dépendances entre deux nœuds localisés sur des processeurs différents ne sont plus des dépendances de données, mais simplement des précédences temporelles.

Chaque sous-graphe est localisé sur un processeur. Les sous-graphes indépendants peuvent alors être extraits du graphe d'origine pour obtenir des programmes "indépendants".

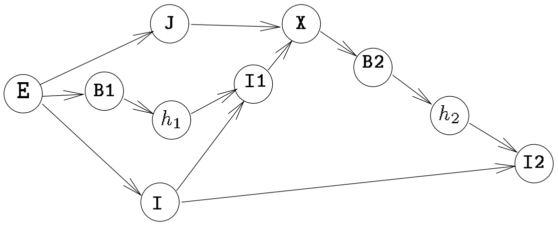
Présenté de cette façon, le problème de la distribution semble très simple. En fait, nous avons intentionnellement caché deux principaux problèmes : la distribution du contrôle et la conséquence de l'extraction des sous-graphes en programmes indépendants sur la sémantique d'origine du programme. Ces problèmes sont explicités dans les sections suivantes.

4.3 Horloges de communication

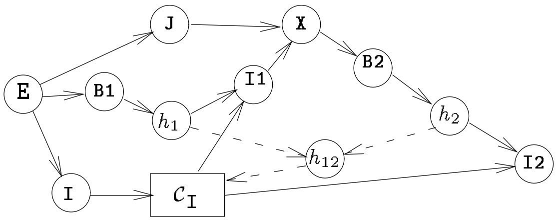
Considérons l'exemple suivant :

```
(| { I, B1, J } := f{ E }
 | I1 := I when B1
 | I2 := I when B2
 | X  := I1 default J
 | B2 := g{ X }         |)
```

Le processus de compilation introduit les horloges h_1 et h_2 dénotant respectivement les instants où les booléens `B1` et `B2` sont vrais. Pour rendre le GFDS aussi clair que possible, nous ne mentionnons pas l'horloge principale du programme, synchrone avec `E`, `I`, `B1`, `J`, `X` et `B2` :



Supposons que le programmeur veut produire `I` sur un processeur $\mathcal{P}$ et `I1` et `I2` sur un processeur $\mathcal{Q}$, un nœud de communication $\mathcal{C}_{\mathtt{I}}$ entre $\mathcal{P}$ et $\mathcal{Q}$ est nécessaire. La borne supérieure des horloges de consommation est $h_1 \vee h_2$, elle est notée h_{12}.



L'introduction de h_{12} entraîne un circuit dans le graphe. Après analyse, le circuit n'est jamais effectif parce que son horloge de dépendance est nulle. Ce problème déjà rencontré dans [10] n'est pas encore résolu.

Puisque l'exactitude de l'algorithme précédent dépend du choix de l'horloge de communication, nous choisissons $h_{\mathcal{Q}}$ de telle sorte qu'il n'y a pas de circuit ni de faux-circuit introduit puisqu'ils ne peuvent pas être pris en compte pour la génération de code finale. Si placer les nœuds de communication à la borne supérieure des horloges d'utilisation introduit des circuits, nous choisissons une horloge plus rapide, au plus égale à $\hat{\mathbf{x}}$, n'introduisant pas de cycles.

Deux nœuds de lecture/écriture associés sont activés à la même horloge. Cette horloge doit donc être présente sur chaque processeur. On suppose que les signaux ne doivent pas être dupliqués, l'horloge est produite sur un processeur et doit donc être communiquée à l'autre : *la communication des données implique la communication du contrôle.*

Ces horloges ne peuvent pas être communiquées à leur propre horloge, mais à une horloge plus rapide (leurs horloges de communication). Ceci implique la présence de cette horloge de communication sur le processeur émetteur et celui récepteur : *la communication du contrôle implique d'autres contrôles de communication.* Bien sur, le phénomène est borné puisque la profondeur de chaque horloge est fixée à la compilation.

Pour optimiser la distribution, une réduction de la hiérarchie d'horloge sur chaque processeur est nécessaire, et explicitée ci-dessous.

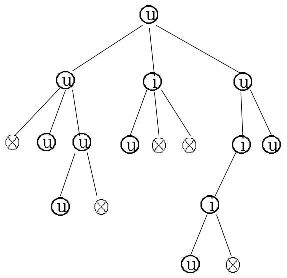
5.1 Extraction des horloges inutiles

Le premier pas de simplification de la hiérarchie est l'élimination des horloges inutiles. Pour savoir quelles horloges peuvent être immédiatement supprimées, nous introduisons la classification suivante :

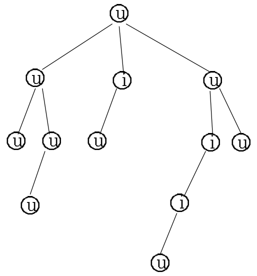
- Une horloge h est dite *nécessaire* sur un processeur $\mathcal{P}$ si et seulement si au moins une des assertions suivantes est vérifiée :
  - une instruction doit être exécutée sur le processeur $\mathcal{P}$ quand h est présente.
  - un signal produit sur $\mathcal{P}$ dépend de h.
  - une autre horloge produite sur $\mathcal{P}$ dépend de h.
- Pour garder la richesse du graphe initial, nous disons d'une horloge h qu'elle est *utile* si elle est nécessaire ou si au moins deux sous-hiérarchie de h possèdent des horloges nécessaires. Les horloges utiles doivent être présentes sur $\mathcal{P}$.
- À l'opposé, une horloge h est dite inutile sur un processeur $\mathcal{P}$ si et seulement si :
  - Il n'y a pas d'instruction à exécuter quand h est présente.
  - Aucun signal sur $\mathcal{P}$ ne dépend de h.
  - Aucune horloge utile produite sur $\mathcal{P}$ ne dépend de h.
  - Toutes les sous-horloges de h sont inutiles sur $\mathcal{P}$.
- Les autres horloges du programme, qui sont ni utiles, ni nécessaires, sont des horloges sans instructions sur $\mathcal{P}$, sans successeur produit sur $\mathcal{P}$, ayant des sous-horloges utiles. Ces horloges sont dites intermédiaires.

Les horloges inutiles peuvent donc être extraites de la hiérarchie d'horloge sans aucun problèmes. La hiérarchie résultante est dites *purifiée*. Considérons la hiérarchie d'horloge sui-

projetée sur un processeur $\mathcal{P}$:



Dans cette hiérarchie, les horloges utiles, inutiles et intermédiaires sur $\mathcal{P}$ sont représentée respectivement par les symboles ⓤ, ⊗ et ⓘ. Après extraction des horloges inutiles, le graphe purifié est :

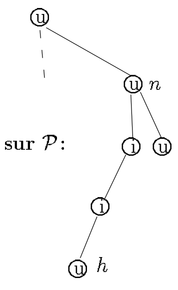


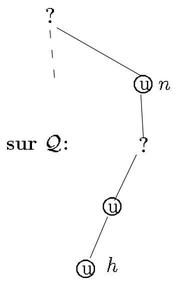
Comme les horloges utiles doivent être présentes sur $\mathcal{P}$, réduire la hiérarchie purifiée signifie extraire les horloges intermédiaires.

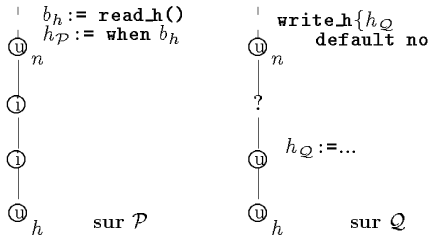
5.2 Extraction des horloges intermédiaires

Considérons une horloge h utile pour le processeur $\mathcal{P}$. Si elle n'est pas produite mais utilisée sur $\mathcal{P}$, elle doit donc être lue du processeur $\mathcal{Q}$ la produisant. Le problème est de savoir à quelle horloge elle doit être lue (horloge que nous allons noter r). Cette horloge doit être ancêtre de h dans la hiérarchie. Si nous voulons extraire du graphe le maximum d'horloge intermédiaires en préservant la hiérarchie, le meilleur choix pour r est l'ancêtre de h utile sur $\mathcal{P}$ le plus près de h, noté n (de cette façon, toutes les horloges intermédiaire entre n et h pourront être supprimées). L'horloge h doit être écrit et lu à la même horloge, n doit être présent sur $\mathcal{Q}$, ce qui n'est pas toujours vérifié dans la pratique.

Si n est utile sur $\mathcal{Q}$, situation présentée sur l'exemple suivant :



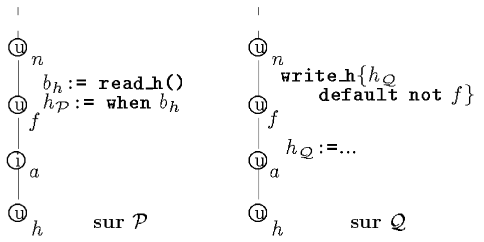




Si toutes les horloges peuvent être communiquées de cette façon, le gain est optimal et toutes les horloges intermédiaires peuvent être supprimées.

Si n n'est pas utile sur $\mathcal{Q}$,

Comme h est produite sur $\mathcal{Q}$, son ancêtre immédiat a est utile sur $\mathcal{Q}$. Il existe donc au moins une horloge entre n et h utile sur $\mathcal{Q}$. Nous avons choisi de communiquer h à la plus rapide des horloges, notée f, entre n et h. Cette horloge intermédiaire f devient utile sur $\mathcal{P}$. Dans l'exemple précédent, si a est différent de f, elle peut être extraite de $\mathcal{P}$:



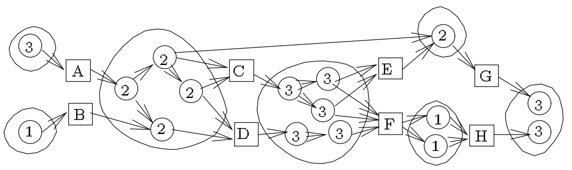
Dans le pire des cas, cette solution marque de nouvelles horloges comme utiles dans la hiérarchie seulement le long des ancêtres de h. Puisque des communications peuvent être introduites dans toute la hiérarchie, la modification de l'arbre d'horloge doit se faire des feuilles vers la racine. De plus, les horloges doivent être traitées une par une pour chaque processeurs : si nous appliquons cette méthode processeurs par processeurs sur la hiérarchie, nous ne bénéficions pas des horloges introduites par les communications nécessaires aux autres processeurs.

5.3 Horloges principales

Prenons q l'horloge utile la plus rapide sur un processeur $\mathcal{P}$ après purification de la hiérarchie, seulement les valeurs indiquant la présence de q ont besoin d'être transmises à $\mathcal{P}$ et l'horloge de communication peut être q elle-même.

6 EXTRACTION DES SOUS-GRAPHES

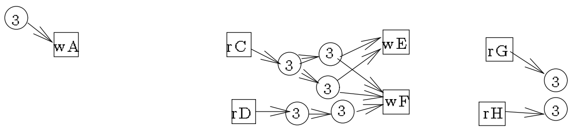
Nous avons dit précédemment que la transformation conserve toutes les propriété du graphe initial. Par une simple extraction de sous-graphes nous perdons la dépendance temporelle entre les nœuds de lecture/écriture nouvellement introduits et ceci peut conduire à des implémentations incorrectes. Considérons le graphe suivant :



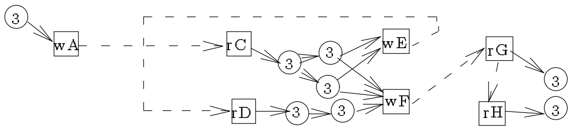
sont représentés respectivement par des cercles et des carrés. Chaque nœud de communication correspond à un nœud d'écriture et au nœud de lecture associé.

6.1 Extraction directe

Si nous extrayons du graphe les nœuds localisés sur le processeur 3, nous obtenons le sous-graphe suivant :



qui peut, en l'absence d'information supplémentaires, conduire à des implémentations incorrectes. La figure suivante montre un renforcement possible des dépendances à l'instant effectué par un ordonnancement statique :

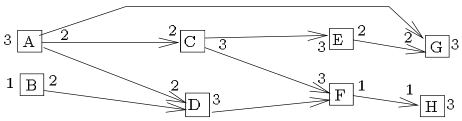


Manifestement, ce renforcement qui n'introduit pas de cycle au niveau du sous-graphe, étant donné qu'il existe des dépendances entre les nœuds de lecture/écriture, conduit à un interblocage. Ce problème déjà rencontré dans [17] est résolu par une méthode "d'abstraction du graphe". Pour le résoudre, nous explicitons les dépendances entre les nœuds d'écriture et de lecture du GFDS complet par une fermeture transitive sur chaque processeur, et ajoutons l'ensemble de dépendances minimales entre les nœuds de lecture/écriture de chaque processeur afin d'extraire les sous-graphes indépendants.

6.2 Renforcement des sous-graphes

Pour être sur que n'importe quel renforcement des dépendances par un scheduler conduit à une exécution correcte, nous devons ajouter des lépendances minimales entre les nœuds de lecture/écriture du graphe qui ne resteignent pas le paralélisme naturel de l'application mais donnent un graphe plus contraint donc plus facile à mettre ne œuvre.

Pour cela, nous rendons explicites les dépendances entre les nœuds de communication. Appliqué sur le graphe précédent, nous obtenons les dépendances suivantes :



Nous évaluons, pour chaque processeur i du programme, les informations suivantes :

- $\mathcal{D}_i(w)$, la liste des nœuds de lecture r localisés sur le processeur i précédés par w, tel qu'il n'y a pas de chemin entre w et r contenant un nœud de lecture localisé sur I.

processeur i précédés par w, tel qu'il existe au moins un chemin de w à r contenant un nœud de lecture localisé sur i.

Les dépendances minimales désirées sont celles à partir d'un nœud d'écriture w vers tous les nœuds de lecture de $\mathcal{D}_i(w)$.

7 SIMULATION

Toutes les modifications précédentes sur le GFDS nous ont conduit à obtenir un sous-graphe par processeur dont la composition (au sens SIGNAL) est équivalente au graphe initial. Mais de nouvelles possibilités, spécifiques aux systèmes distribués apparaissent pour la simulation. Sur un système monoprocesseur, l'exécution d'instants logiques consécutifs est exclusive, on génère alors un code statiquement séquencé. Sur un système distribué, sans informations supplémentaires, le processeur produisant l'horloge la plus rapide du programme peut être prêt à exécuter un instant alors que l'instant logique précédent est en cours d'exécution. Ceci est en parfaite contradiction avec la sémantique de SIGNAL, mais il est toutefois possible de concevoir des implémentations menant aux résultats suivants :

- une exécution synchronisée, où l'exécution d'un instant ne peut commencer si celle de l'instant précédent n'est pas terminée.
- une exécution faiblement désynchronisée autorisant un chevauchement des instants logique au même instant physique.

7.1 Mise en œuvre synchronisée

Nous souhaitons dans cette sous-partie, que l'exécution préserve la sémantique observationnelle de SIGNAL : l'exécution d'un instant ne peut commencer tant que celle de l'instant précédent n'est pas terminée. Ceci n'est possible que si le processeur $\mathcal{P}$ produisant l'horloge la plus rapide du programme est informé de la terminaison de l'exécution de l'instant courant de tous les processeurs.

Pour cela, nous devons faire précéder (resp. suivre) les nœuds sans prédécesseurs (resp. successeurs) des graphes par un nœud de départ (resp. fin). Si l'exécution du nœud de départ d'un instant n'est pas effectuée avant celle du nœud de fin de l'instant précédent, il n'y a pas chevauchement des instants logiques.

Cette première solution est dans la logique d'une synchronisation forte. Il est possible d'envisager un relâchement de synchronisation en autorisant un nombre borné de chevauchements entre instants logiques (voir 7.2).

Nous voulons implémenter le graphe instantané (GFDS) de manière ininterromptible (procédurale). Le schéma d'exécution se doit de respecter trois impératifs : fidélité, sûreté et efficacité.

fidélité : l'exécution doit respecter les dépendances de la spécification ;

sûreté : l'exécution doit toujours être possible. En particulier, l'ajout de nouvelles dépendances ne doit pas créer de cycles ;

efficacité : le schéma d'exécution ne doit pas laisser le programme en attente d'une réception si d'autres instructions peuvent être exécutées ;

scheduler dynamique, nous allons effectuer des regroupements de nœuds qui seront implémentés de manière procédurale.

Nous voyons tout d'abord sous quelles conditions un ensemble de nœuds peut être implémenté de manière procédurale tout en respectant les impératifs ci-dessus. Nous pouvons ensuite parler de leur abstraction et de leur mise en œuvre.

Considérons le graphe $\mathcal{G} = < \mathcal{N}, \mathcal{C}, \rightarrow >$ sur un ensemble de nœuds $\mathcal{N}$, ayant un ordre partiel dit de *sérialisation* ($x \rightarrow y$ indique que l'exécution de y ne peut avoir lieu avant celle de x) et un ensemble des nœuds critiques $\mathcal{C}$ (nœuds dont l'exécution est déclanchée par l'environnement).

Extraction des procédures

En l'absence de nœuds critiques, un ensemble de nœuds $\mathcal{P}$ peut être implémenté de manière procédurale si et seulement si il est non ré-entrant pour la relation de sérialisation :

$$\forall (x,y) \in \mathcal{P}^2,\ \forall z \in \mathcal{N},\ (x \neq y \text{ et } x \rightarrow z \rightarrow y) \implies z \in \mathcal{P}$$

En présence de nœuds critiques, les conditions de regroupement doivent être affinées afin de satisfaire le critère d'efficacité : la procédurisation ne doit pas empêcher certains séquencements initialement possibles d'un nœud (critique ou non) vers un nœud critique. En plus de la condition précédente, la relation pour laquelle les procédures doivent être non ré-entrantes est la fermeture transitive de l'union des sérialisations initialiales et des sérialisations possibles des nœuds (critiques ou non) vers les les nœuds critiques. Cette relation, notée $\mapsto$, est définie par :

$$\forall (x,y) \in \mathcal{N}^2,\ x \mapsto y \iff \left(\begin{array}{c} x \rightarrow y \\ \text{ou} \\ y \in \mathcal{C} \text{ et } \neg(x \rightarrow y) \end{array} \right)$$

Soit $\mathcal{P}$ un sous ensemble de $\mathcal{N}$, $\mathcal{P}$ peut être implémenté de manière procédurale si et seulement si :

$$\forall (x,y) \in (\mathcal{P} \cap \mathcal{C})^2,\ \forall z \in \mathcal{C},\ (x \neq y \text{ et } x \mapsto^* z \mapsto^* y) \Rightarrow z \in \mathcal{P}$$

Abstraction des procédures

Tout ensemble de nœuds formant une procédure $\mathcal{P}$ est abstrait en un nouveau nœud considéré comme atomique, et critique s'il contient au moins un nœud critique. L'abstraction des sérialisations est faite par l'union des sérialisations de tous les nœuds de la procédure.

Recherche des procédurisations possibles

Elle se fait tout d'abord par l'abstraction des *lignées* [15] sur nœuds critiques, dont on peut montrer qu'elles sont implémentables statiquement.

La suite de la procédurisation menant à un arbre de procédures se fait par application de règles de regroupement.

Mise en œuvre des procédures

En l'absence de nœuds critiques, toute procédure peut être implémentée de manière statique si l'ordonnancement choisi mène à une exécution correcte de celle-ci.

En présence de nœuds critiques, un renforcement de la sérialisation est acceptable s'il ne force pas l'inter-séquencement entre des nœuds critiques. En particulier, la mise en œuvre statique d'une procédure est impossible dès que la procédure contient deux nœuds critiques non contraints (non sérialisés entre eux).

Si les temps d'exécution des graphes sur les processeurs sont variables, ce qui est le cas dans la pratique, le chevauchement des instants peut permettre de remplacer un temps d'attente suite à une exécution rapide par le début d'exécution de l'instant suivant. Ceci est possible puisque les GFDS ont une partie du contrôle dépendant des données.

Cette extension de la distribution s'orienterait vers des schémas d'exécution désynchronisés des programmes SIGNAL, dont la validation nécessite l'étude approfondie des comportements dynamiques des applications synchrones distribuées. Cette étude est actuellement en cours.

8 PERSPECTIVES

Nous venons de présenter une distribution fonctionnelle des programmes SIGNAL. Partant de directives de répartition au niveau du source, elle permet à l'utilisateur de ne voir qu'un seul formalisme : les équations sur signaux.

Après diverses transformations, le compilateur produit pour chaque processeur cible un graphe flot de données synchronisé, la composition de tous ces graphes donnant un programme SIGNAL dont la composition a la même sémantique observationnelle que le programme d'origine.

La distribution fonctionnelle est faite uniquement sur critères qualitatifs, pour répondre à des exigences de topologie du système (délocalisation de capteurs par exemple). L'algorithme de purification des hiérarchies distribuées semble cependant applicable aux outils de distribution sur critères qualitatifs. Certaines hypothèses de départ de ce travail, comme la non-duplication des nœuds de calcul, pourraient aussi être remises en cause, couplées à des critères quantitatifs pour mener à des implémentations encore plus efficaces, tout en gardant les bénéfices de la programmation synchrone (vérification notamment).

Enfin, le schéma d'exécution montré ici est celui d'exécutions synchronisées distribuées. Le relâchement du synchronisme apparaît comme la plus prometteuse des perspectives de simulation de SIGNAL. Cependant, la validation de schémas d'exécutions désynchronisées, ne respectant plus la sémantique observationnelle de SIGNAL, nécessite une connaissance fine des comportements dynamiques, qui sont actuellement à l'étude.

RÉFÉRENCES

[1] Belhadj (Mohammed). – *Conception d'architectures en utilisant* SIGNAL *et VHDL.* – France, Thèse de PhD, Université de Rennes 1, décembre 1994.

[2] Benveniste (Albert) et Berry (Gérard). – The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, vol. 79, n° 9, September 1991, pp. 1270–1282.

[3] Benveniste (Albert), Caspi (Paul), Le Guernic (Paul) et Halbwachs (Nicolas). – Data-Flow Synchronous Languages. *In : Lecture Notes in Computer Science 803, Proc. of the REX School/Symposium, Noordwijkerhout, Netherlands*, éd. par de Bakker (J.W.), de Roever (W.P.) et Rozenberg (G.). pp. 1–45. – Springer–Verlag.

[4] Berry (Gerard), Couronné (Philippe) et Gonthier (Georges). – Synchronous programming of reactive systems: an introduction to ESTEREL. *In : Programming of future generation computers*, éd. par Fuchi (K.) et Nivat (M.). ICOT INRIA, pp. 35–56. – North-Holland.

pendances, environnement. – France, Thèse de PhD, Université de Rennes 1, septembre 1992. in french.

[6] Bournai (Patricia), Lavarenne (Christophe), Le Guernic (Paul), Maffeïs (Olivier) et Sorel (Yves). – *Interface SIGNAL-SynDEx.* – Research report n° 2206, Rennes, INRIA France, March 1994.

[7] Bournai (Patricia), Chéron (Bruno), Gautier (Thierry), Houssais (Bernard) et Le Guernic (Paul). – SIGNAL *manual.* – Technical Report n° 745, IRISA, juillet 1993.

[8] Boussinot (Frédéric) et De Simone (Robert). – The ESTEREL language. *Proceedings of the IEEE*, vol. 79, n° 9, September 1991, pp. 1293–1304.

[9] Caspi (Paul), Pilaud (Daniel), Halbwachs (Nicolas) et Plaice (John Alexander). – LUSTRE: A declarative language for programming synchronous systems. *In : 14th ACM Symposium on Principles of Programming Languages*, pp. 178–188. – Munich, 1987.

[10] Chéron (Bruno). – *Transformations syntaxiques de programmes* SIGNAL. – France, Thèse de PhD, Université de Rennes 1, Septembre 1991.

[11] Gautier (Thierry), Le Guernic (Paul) et Besnard (Loïc). – SIGNAL: a declarative language for synchronous programming of real-time systems. *In : Functional programming languages and computer architecture*, éd. par Kahn (Gilles), pp. 257–277. – Lecture Notes in Computer Science, 274, Springer-Verlag, 1987.

[12] Gautier (Thierry), Le Guernic (Paul) et Dupont (François). – SIGNAL *V4 : manuel de référence (version préliminaire).* – Publication interne n° 832, IRISA, June 1994.

[13] Halbwachs (Nicolas), Caspi (Paul), Raymond (Paul) et Pilaud (Daniel). – The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, vol. 79, n° 9, September 1991, pp. 1305–1321.

[14] Lavarenne (Christophe), Segrouchni (O.), Sorel (Yves) et Sorine (Michel). – The SYNDEX software environment for real-time distributed systems design and implementation. *In : European Control Conference*, pp. 1684–1689.

[15] Le Goff (Bernard) et Le Guernic (Paul). – The granules, glutton: An idea, an algorithm to implement on multiprocessor. *In : STACS 88, Lectures Notes in Computer Science*, éd. par Wirsing (R. Cori M.). AFCET. – Bordeaux France, February 1988. Volume 294.

[16] Le Guernic (Paul), Gautier (Thierry), Le Borgne (Michel) et Le Maire (Claude). – Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, vol. 79, n° 9, septembre 1991, pp. 1321–1336.

[17] Maffeïs (Olivier). – *Ordonnancements de graphes de flots synchrones; Application à* SIGNAL. – France, Thèse de PhD, Université de Rennes 1, janvier 1993.