Répartition de Circuits Séquentiels

Clément MÉNIER

31 juillet 2001

Stage effectué sous la direction d'**Alain Girault**
à l'INRIA Rhône-Alpes au sein du projet BIP.

Résumé :

Dans le cadre de la conception de programmes commandes action-réaction, les contraintes physiques impliquent fréquemment une programmation multi-sites. Pour faciliter le développement de tels projets, nous avons réalisé un générateur automatique de programmes distribués à partir d'une description des souhaits de répartition et d'un programme général unique. Partant de l'outil Ocrep existant, nous nous sommes intéressés à la répartition du format SC généré par ESTEREL, langage synchrone très utilisé dans la conception de tels projets. Nous avons aussi étudié les méthodes qu'il faudrait mettre en œuvre pour appliquer la répartition automatique au format CP généré par un nouveau compilateur ESTEREL.

Mots-clés : programmation distribuée, répartition, code séquentiel, SC, ESTEREL.

Remerciements :

Je remercie tout d'abord l'équipe BIP et l'INRIA Rhône-Alpes pour m'avoir accueilli dans leurs murs.

Je remercie particulièrement **Alain Girault**, mon maître de stage, qui m'a permis de réaliser ce stage dans des conditions excellentes, grâce à sa disponibilité et à son aide fréquente. Je remercie aussi **Daniel Simon** qui m'a présenté les utilisations concrètes d'ESTEREL et m'a permis de mieux comprendre les avantages de ce langage.

Je remercie **Gérard Berry**, **Yanis Bress** et **Xavier Fornari** de l'équipe du CMA et d'Esterel Technologies pour le temps qu'ils m'ont consacré à répondre à mes multiples questions. Je remercie aussi l'équipe de France Telecom R&D (**Daniel Weil**, **Etienne Closse**, **Jacques Pulou**) pour m'avoir accordé une partie de leur temps afin de m'expliquer le fonctionnement de leur compilateur, et de rechercher un compromis afin que je puisse travailler sur celui-ci.

Je voudrais enfin remercier toutes les personnes de l'INRIA que j'ai côtoyées durant ce stage et qui m'ont fait passer un moment très agréable. En particulier encore merci Alain pour le barbecue et la dégustation de vins, merci Fabien, Christine et Soraya pour avoir organisé les sorties restaurants et autres pique-niques.

Table des matières

1 Introduction

L'engouement pour des machines automatisées de plus en plus complexes implique une recherche de méthodes de développement plus aisées et plus sûres.

1.1 l'INRIA Rhône-Alpes et le projet BIP

Dans cette optique le projet BIP a été créé autour de la conception d'un robot bipède (Fig. 1) avec une collaboration étroite entre automaticiens et informaticiens. Celle-ci a abouti sur la réalisation d'outils de programmation (ORCCAD) adaptés aux besoins des roboticiens. Ces recherches ont aussi permis la création d'une société, Athys, offrant une panoplie de logiciels aidant à la conception de chaînes de production automatisées.

FIG. 1 – Le robot bipède.

1.2 Les Langages synchrones et les liens avec la commande robotique

Les langages synchrones sont basés sur le même principe que les circuits synchrones. Ils comportent une notion de cycle, défini par des tops d'horloge appelés ticks. A chaque cycle correspond une *réaction*, c.-à-d. l'exécution d'un ensemble d'actions. Ces langages s'avèrent très utiles dans le cadre de la programmation des systèmes temps-réels critiques tels qu'utilisés dans les projets robotiques.

1.3 Les différentes méthodes de programmation multi-sites

Les contraintes physiques, telles que l'éloignement excessif entre divers capteurs, imposent une programmation multi-sites, c.-à-d. le développement de plusieurs programmes tournant conjointement en plusieurs endroits. Pour réaliser cet ensemble de logiciels, deux méthodes s'offrent au développeur :

- la programmation **séparée** : chaque programme pour chaque site est créé de manière indépendante ;

– la programmation **répartie** : une fois entièrement conçu et développé, un programme global représentant le fonctionnement général des tâches à effectuer est réparti automatiquement en plusieurs programmes selon les spécifications du projet.

Dans le cadre de la programmation séparée, le programmeur a plus de liberté mais doit s'occuper lui-même des communications entre processus. En revanche la programmation répartie procure une facilité de conception accrue et des possibilités plus avantageuses pour vérifier le comportement global des tâches.

2 Le langage synchrone Esterel et ses différents formats de compilation

Le langage ESTEREL, conjointement développé au Centre de Mathématiques Appliquées de l'École des Mines de Paris et à l'INRIA Sofia-Antipolis par l'équipe de **Gérard Berry** et distribué par Esterel Technologies, permet une compilation vers plusieurs formats présentant divers intérêts :

OC (*Object Code*) **:** décrit un automate, dont chaque état correspond à une réaction représentée sous la forme d'un DAG (graphe orienté acyclique). Il est très efficace en rapidité mais a une taille potentiellement exponentielle par rapport à la taille du programme ESTEREL d'origine.

SC (*Sequential Circuit*) **:** décrit un circuit de type booléen. Il est moins efficace en rapidité mais conserve une taille raisonnable.

CP (*Control Point*) **:** nouveau format créé par une équipe de France Telecom R&D qui allie rapidité et taille minimale.

Les possibilités de compilation d'un format vers un autre sont représentées à la Figure 2.

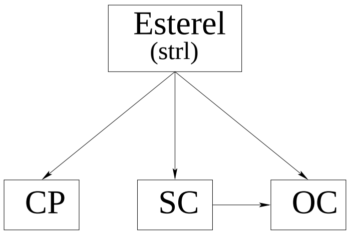


FIG. 2 – Les liens entre les différents formats d'ESTEREL

2.1 Le langage Esterel

ESTEREL est un langage de programmation impératif basé sur la gestion de signaux qui propose une méthode de conception différente des langages traditionnels. Certaines de ces particularités auront une influence sur la répartition :

- le **parallèle** : qui permet de préciser que plusieurs tâches sont à effectuer simultanément ;
- la **gestion des événements** : qui permet de contrôler l'exécution du programme directement en fonction des signaux reçus. Le lancement d'une tâche est conditionnée par les événements extérieurs.

Vous pouvez voir un exemple d'une utilisation de ces deux particularités dans le programme *prog.strl* en Annexe A.1. Le langage ESTEREL est largement décrit dans le manuel [4] et l'article [3].

2.2 Le format SC

Le format SC (décrit dans le rapport technique [2]) représente un circuit booléen sous la forme d'un graphe dont les nœuds sont des fonctions booléennes et les arêtes des fils. Il existe plusieurs type de fils :

- **standard** : propage instantanément l'influx électrique (0 ou 1) ;
- **action** : propage instantanément l'influx et exécute l'action à laquelle il correspond si son état passe de 0 à 1 ;
- **if** : évalue, si son état passe de 0 à 1, l'expression booléenne qui lui est rattaché et propage le résultat de cette évaluation ;
- **registre** : propage avec délai l'influx électrique, il attend le tick d'horloge suivant ;
- **input** : propage 1 si le signal correspondant est présent.

Le fil **input** est particulier car il n'a pas de prédécesseur : en effet il est exécuté à chaque tick d'horloge. Une réaction correspond à la remise à zéro de tous les fils en dehors des registres, puis à la propagation des influx électriques dans le circuit suivant les règles précédentes.

Un exemple de circuit *prog.sc* se trouve en Annexe A.2. Les nœuds \$0 et \$1 représentent respectivement les influx électriques constants 0 et 1.

2.3 Le format CP

Pour concilier rapidité d'exécution et taille minimale, des chercheurs de France Telecom R&D ont créé un nouveau compilateur d'ESTEREL vers le langage C. La méthode consiste à détecter des «points de contrôles» et à associer à chacun d'eux un ensemble d'actions, appelé tâche. Celles-ci ne seront exécutées que si leur point de contrôle associé est activé.

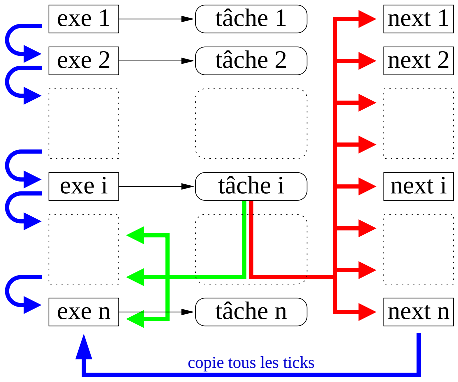


FIG. 3 – La méthode d'exécution du format CP

La Figure 3 représente à gauche le vecteur «EXE» des points de contrôles activés pour le tick en cours et à droite le vecteur «NEXT» des points activés pour le tick suivant. Une réaction correspond à l'exécution dans l'ordre croissant (de haut en bas) des tâches correspondant aux points de contrôles actifs. De plus les actions à l'intérieur des tâches peuvent influer sur le vecteur :

- «EXE» si la position du point modifié est postérieure au point courant (flèches vertes) ;
- «NEXT» n'importe où (flèches rouges).

Le fonctionnement précis de ce compilateur est décrit dans l'article [5].

3 L'outil Ocrep

La répartition automatique de programmes OC a été étudiée par **Paul Caspi**, **Alain Girault** et **Daniel Pilaud** et est décrite dans leur article [1]. Nous avons repris une partie du code de leur outil en ce qui concerne les déclarations d'en-tête communes aux deux formats OC et SC. Nous avons aussi conservé et amélioré l'utilisation des files FIFO pour la gestion des communications entre les différents sites.

3.1 L'algorithme de répartition

L'algorithme se décompose ainsi :

1. réplication et localisation ;
2. pour chaque état :
   (a) insertion des émissions (put) dans le DAG de l'état ;
   (b) insertion des réceptions (get) dans le DAG de l'état ;

La première étape consiste à associer à chaque action un ensemble de sites qui devront l'exécuter. Les actions de contrôle, telles que «if ... then ... else», sont exécutées par tous les sites pour garantir que tous les différents programmes suivent le même parcours dans le DAG et les mêmes transitions de l'automate. Les actions de calcul sont exécutées par les sites où sont localisées les variables qu'elles modifient. Nous dirons qu'une action appartient à un site s si ce site doit exécuter cette action.

L'étape 2a est effectuée en parcourant le DAG en «remontant» depuis les feuilles vers la racine. Lors de celui-ci sont analysées les dépendances de variables : une action a besoin d'une variable si cette dernière est nécessaire au calcul de la fonction d'entrée associée. Ces besoins sont propagés ainsi vers le haut. Le but est de repousser le plus tard possible (donc en réalité le plus tôt car le DAG est exécuté à partir de la racine en «descendant») l'émission d'une variable pour exploiter au mieux la latence du réseau. Si une action dépend d'une variable qui est déjà présente dans la liste des besoins nous ne rajoutons pas une dépendance supplémentaire. Ceci permet de supprimer des émissions inutiles dans le cas où deux actions ont besoin d'une même variable et que celle-ci n'est pas modifiée entre les deux.

Le placement des réceptions lors de l'étape 2b se fait par un parcours «descendant» lors duquel nous simulons l'état des files pour corriger quelques inversions possibles. Chaque *get* est ainsi placé le plus tard possible, juste avant l'action qui a besoin de la variable transmise. Nous maintenons à tout moment pour chaque site la liste des variables dont celui-ci a une valeur correcte en mémoire. Ceci permet de ne pas faire les *get* redondants éliminés lors de l'étape précédente.

Pour de plus amples détails sur cet algorithme, vous pouvez vous référer à l'article [1] ou bien à la partie 4.2 qui en détaille une version plus sophistiquée.

3.2 L'outil Asig

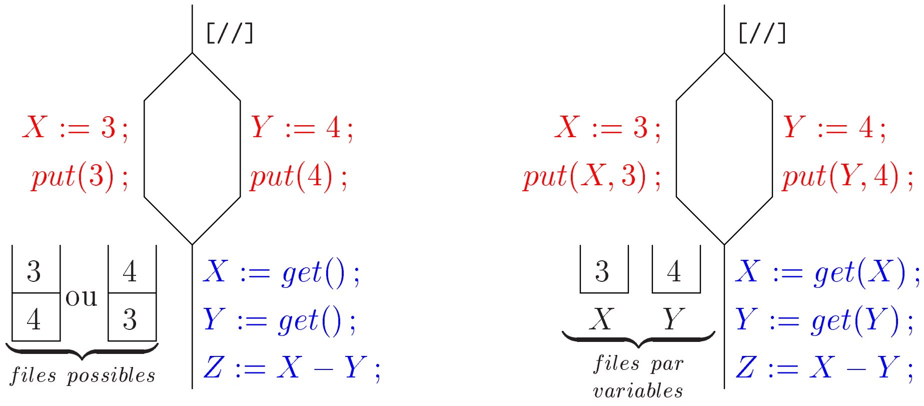
Étant donné l'impossibilité d'utiliser le simulateur fourni par Esterel Technologies à cause de la nécessité de créer des sockets de communication au démarrage des programmes, **Alain Girault** et **Serge Mani** ont réalisé un générateur automatique d'interfaces graphiques Tcl/Tk à partir d'un fichier OC, permettant en particulier une gestion des programmes répartis. Cet outil génère un *Makefile* pour chaque fichier OC en-capsulant la bibliothèque des primitives de communications.

4 La gestion des problèmes dus au parallélisme des codes SC et CP

Alors que le code OC est purement séquentiel, une certaine forme de parallélisme existe dans les formats SC et CP.

4.1 Contraintes dues à l'indéterminisme de l'ordre d'exécution

Un premier problème survient lors de la gestion des émissions et réceptions des variables à cause de l'indéterminisme lié à la structure parallèle. Considérons l'exemple de la figure 4(a) où les actions en rouge et celles en bleu sont exécutées sur deux sites distincts ; il est impossible de prévoir l'ordre dans lequel les instructions $put(3)$ et $put(4)$ sont exécutées, et par conséquent Z peut aussi bien valoir 1 que -1. Pour résoudre ce problème, nous avons décidé d'utiliser, entre chaque paire de sites, plusieurs files, une par variable, au lieu d'une globale. En effet nous savons qu'il est impossible pour des raisons de causalité qu'une même variable soit modifiée dans plusieurs branches en parallèle. Ainsi l'ordre d'émission pour une variable donnée est totalement déterministe.



(a) Le problème du à l'indéterminisme dans l'ordre des émissions-réceptions

(b) L'utilisation de files par variables

FIG. 4 – Les contraintes dues au parallélisme

Pour ne pas trop alourdir les programmes en utilisation mémoire, nous avons implémenté les n files par une seule file à double entrée. Ceci est illustré par la Figure 5.

Lors d'un appel à put, nous envoyons dans un même message le nom de la variable et sa valeur. Lors d'un get, nous ôtons de la file la première occurrence de celle-ci si cette variable est présente et sinon nous attendons

nom de la variable	valeur associée
X	4
Y	1
Z	2
X	5

$\Longleftrightarrow$

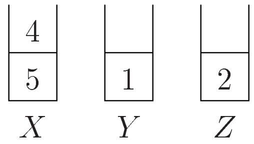


FIG. 5 – La simulation des n files par une seule file à double entrée

l'arrivée d'une donnée relative à cette variable en plaçant les autres dans la file.

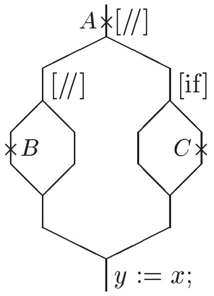
4.2 Un algorithme plus fin pour la gestion des dépendances de données

Le format CP assimile les tâches à des DAG en ajoutant par rapport au format OC la structure de contrôle «parallel ... end parallel», notée //. Nous avons donc décidé d'utiliser les mêmes méthodes que celles mises en oeuvre dans Ocrep en modifiant toutefois les étapes 2a et 2b de l'algorithme décrit à la partie 3.1.

4.2.1 L'algorithme de placement des émissions

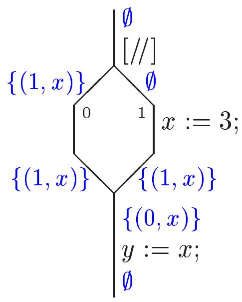
Comme dans l'algorithme de répartition d'Ocrep, nous plaçons les *put* en parcourant chaque DAG en «remontant». Alors que dans ce dernier, nous propagions des ensembles de variables, nous allons désormais propager des ensembles $Need_s$ dont les éléments sont des couples (a, x) où a est un entier naturel et x le nom d'une variable. a correspond au degré de parallélisme du besoin en x ce qui permet de préciser qu'un besoin peut être satisfait dans une des branches du parallèle mais pas dans toutes. Voici un exemple du degré de parallélisme d'un besoin :

- en A, on a $(0, x) \in Need_s$ car ce besoin n'est pas un besoin parallélisé ;
- en B, on a $(2, x) \in Need_s$ car ce besoin est situé deux niveaux de structure parallèle plus bas que l'action l'ayant engendré ;
- en C, on a $(1, x) \in Need_s$ car ce besoin n'est en réalité qu'à un degré 1 de parallélisation par rapport à l'action $y := x$.



Et voici un exemple de gestion d'un besoin parallélisé où x appartient au site s_0 et y au site s_1 :

L'action $y := x$ crée un besoin $\{(0, x)\}$. Celui-ci est propagé dans les deux branches parallèles mais voit son degré de parallélisation incrémenté. La branche 1 résout la dépendance et propage donc $\emptyset$. Ainsi au niveau du branchement séparatif // on a $(1, x) \in Need^0_{s_1}$ ce qui indique une dépendance en x non spécifique à cette branche. Or $x \notin Need^1_{s_1}$ donc ce besoin a été résolu par la branche 1 et nous l'éliminons en propageant $\emptyset$ au delà du branchement séparatif //.



Nous définissons sur ces ensembles $Need_s$, les opérateurs d'appartenance $\in$, d'ajout sans redondance $\lhd$ et d'intersection $\sqcap$ de la manière suivante :

$$x \in Need_s \iff \exists\, a, (a, x) \in Need_s$$

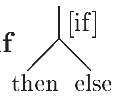
$$Need_s \lhd (a, x) = \begin{cases} Need_s \cup \{(a, x)\} & \text{si } x \notin Need_s \\ Need_s - \{(b, x)\} \cup \{(a, x)\} & \text{si } (b, x) \in Need_s \end{cases}$$

$$(c, x) \in Need^1_s \sqcap Need^2_s \iff \begin{cases} (a, x) \in Need^1_s \\ (b, x) \in Need^2_s \end{cases} \text{ et } c = min(b, a)$$

Étant donné que nous ne manipulons $Need_s$ que via ces opérateurs, ces ensembles ne peuvent pas contenir deux occurrences d'une même variable. Ainsi nous pouvons définir l'opération de suppression d'une variable x comme la suppression de l'unique couple ayant comme nom de variable x.

Nous créons aussi à chaque nœud des ensembles vides $GetStop_s$ destinés à contenir les variables dont il est impératif de faire un *get* immédiatement après ce nœud.

Nous plaçons au départ pour chaque site s et chaque feuille du graphe un ensemble vide $Need_s$. Puis nous propageons ces ensembles vers la racine de la manière suivante :

- lorsque l'on rencontre une action appartenant au site s et ayant besoin de la variable x nous rajoutons $(0, x)$ à $Need_s$;
- lorsque l'on rencontre une action qui modifie x, alors pour tout site s tq $x \in Need_s$, on insère après celle-ci l'action *put(s,x)* et on ôte x de $Need_s$;
- lorsque l'on rencontre un branchement séparatif **if** 

, alors pour tous les sites s :

1. on récupère $Need_s^{then}$ et $Need_s^{else}$ provenant des 2 branches ;
2. pour chaque branche b, on insère l'action *put(s,x)* pour tout $x \in Need_s^b - (Need_s^{then} \sqcap Need_s^{else})$;
3. on propage $Need_s^{then} \sqcap Need_s^{else}$;

– lorsque l'on rencontre un branchement jointif **if** then else [if], on duplique $Need_s$ que l'on propage dans les 2 branches ;

– lorsque l'on rencontre un branchement séparatif // [//] 1 2 3, alors pour chaque site s :
  1. on récupère $Need_s^i$ provenant des différentes branches ;
  2. on crée $Need_s := \emptyset$;
  3. puis nous construisons incrémentalement $Need_s$ en faisant, pour chaque variable x :
     – si $\exists!\, i, (0, x) \in Need_s^i$, c.-à-d. si une seule branche n'a réellement besoin de x, alors $Need_s := Need_s \lhd (0, x)$;
     – si il y a plusieurs i tq $(0, x) \in Need_s^i$, c.-à-d. si plusieurs branches ont besoin de x, alors nous propageons $Need_s := Need_s \lhd (0, x)$ et nous ajoutons x à $GetStop_s$ car il sera impératif de faire un *get* de x à cet endroit ;
     – si $\exists\, a \geq 1, \forall\, i, (a, x) \in Need_s^i$ ou $x \notin Need_s^i$, alors
     $\begin{cases} \text{si } \forall\, i, x \in Need_s^i \text{ alors } Need_s := Need_s \lhd (a-1, x) \\ \text{sinon le besoin a été résolu dans une des branches} \end{cases}$;
     – les autres cas ne peuvent pas se produire ;

– lorsque l'on rencontre un branchement jointif // 1 2 3 [//], alors pour chaque branche i et chaque site s on crée $Need_s^i$ de la manière suivante :
  – au départ $Need_s^i := \emptyset$;
  – puis $\forall i,\ \forall (a, x) \in Need_s,\ Need_s^i := Need_s \lhd (a + 1, x)$, car il faut incrémenter le degré de parallélisation de chaque besoin.

4.2.2 L'algorithme de placement des réceptions

L'algorithme de placement des *get* est identique à celui utilisé pour Ocrep avec comme seules particularités :

– si $x \in GetStop_s$ alors il faut insérer juste après ce nœud l'action *get(x)* appartenant au site s ; il ne faut pas repousser plus loin la réception de cette valeur car nous savons que plusieurs branches en parallèles ont besoin de cette variable ; sans cela nous serions obligé de faire plusieurs

get, ne pouvant garantir le fait qu'une action nécessitant x s'exécute après l'action *x := get(x)* placée sur une autre branche ;
– lors d'un branchement jointif //, il faut fusionner les listes des variables connues provenant des différentes branches ;
– l'utilisation de files par variables permet de se passer de la simulation des états des files lors du placement des *get*.

5 La réalisation de l'outil Screp

Pour réaliser cette outil, il nous a tout d'abord fallu créer un parser pour le format SC. Pour ce faire nous avons utilisé les outils *lex* et *yacc* en se servant des spécifications de ce format données dans [2].

5.1 La simulation des «input»

Lors de la compilation d'ESTEREL vers SC, les tests relatifs à la présence de signaux sont directement convertis en circuits booléens, la valeur de présence d'un signal étant propagée par son fil **input** : nous appellerons par la suite ces tests des expressions booléennes pures. Or dans un programme distribué, seuls les fils liés à un signal localisé sur le site du programme propagent un influx correct, les autres propageant 0. Ainsi l'évaluation de certains de ces tests sera incorrecte, et le programme réparti lui-même sera incorrect. Pour résoudre ce problème, une première méthode serait, au début de chaque réaction, d'envoyer à tous les sites la présence des signaux, les fils **input** propageant ainsi des valeurs correctes. Mais ceci n'est efficace que dans le cas où il n'y aurait que peu de signaux, ce qui n'est pas souvent le cas. Il nous faut donc trouver une méthode pour que les échanges de variables de présence ne se fassent que lorsqu'elles sont nécessaires.

Pour détecter ces besoins et permettre, à la demande, de rétablir un influx correct à la sortie d'un fil **input**, nous proposons de transformer préliminairement le circuit de la manière suivante :

1. détection des nœuds correspondant à un test relatif à la présence de signaux ;
2. création de blocs de simulation pour les fils **input** ;
3. connection des nœuds détectés aux blocs nécessaires.

Pour détecter les nœuds susceptibles de nécessiter la connaissance de certaines valeurs propagées par les fils **input**, nous utilisons le fait que ces nœuds sont des portes «AND» ayant une ou plusieurs entrées correspondant à des évaluations d'expressions booléennes pures et au moins une entrée dont l'influx ne dépend pas uniquement de la présence de signaux. Nous attribuons à chaque nœud un ensemble $S_{need}(n)$ destiné à contenir les signaux dont il est nécessaire de connaître la présence pour pouvoir calculer l'influx de sortie. Pour déterminer $S_{need}(n)$, nous mettons au départ $\forall n, S_{need}(n) = \emptyset$ puis, à partir de chaque fil **input**, nous parcourons partiellement le circuit en marquant les nœuds visités et en propageant un ensemble S de signaux de la manière suivante :

- au départ de chaque parcours, le fil **input** relatif au signal s propage $S = \{s\}$;
- un noeud n'ayant qu'une entrée propage aux nœuds suivants le même S qu'il a reçu ;
- dans le cas d'un noeud comportant plusieurs entrées :
  - on rajoute à $S_{need}(n)$ l'ensemble S reçu ;
  - si tous les noeuds connectés en entrée ont été visités, alors ce nœud appartient à une expression booléenne pure et donc nous propageons $S = S_{need}(n)$ puis remettons $S_{need}(n) = \emptyset$.

L'étape 2 consiste à faire pour chaque fil **input** la transformation décrite à la Figure 6 :

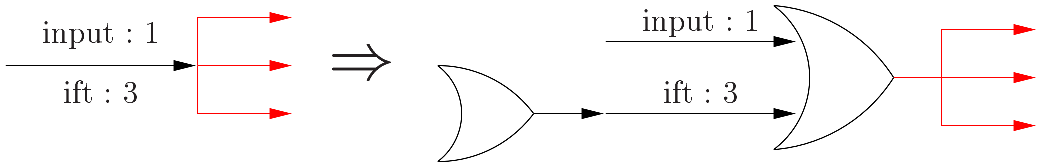


FIG. 6 – Transformation des fils **input** en blocs de simulation

L'étape 3 consiste à connecter chaque nœud n tel que $S_{need}(n) \neq \emptyset$ aux blocs de simulation nécessaires. Nous dupliquons ce nœud «AND» en éliminant les fils relatifs à des expressions booléennes pures et nous créons des fils entre ce nouveau nœud et les blocs de simulation des signaux appartenant à $S_{need}(n)$. Le résultat obtenu est représenté à la Figure 7 où le nœud rajouté est en bleu et les fils éliminés en rouge. Ainsi la simulation des fils **input** ne se fait que si le nœud n en a réellement besoin, ceci afin d'éviter des échanges de données inutiles. De plus on remarque que les blocs de simulation ne peuvent être exécutés au plus qu'une fois durant une réaction, ce qui est normal car la valeur de présence d'un signal est constante durant celle-ci ; on évite ainsi de nombreux échanges redondants.

5.2 Les modifications apportées à l'outil Asig

Pour pouvoir réutiliser l'outil Asig pour les programmes générés par Screp, il nous a fallu le modifier légèrement. Étant donné que les entêtes des programmes OC et SC sont quasiment identiques, il nous a simplement fallu effectuer quelques modifications dans le parser et supprimer l'analyse des corps de fichiers dont ne se sert pas Asig.

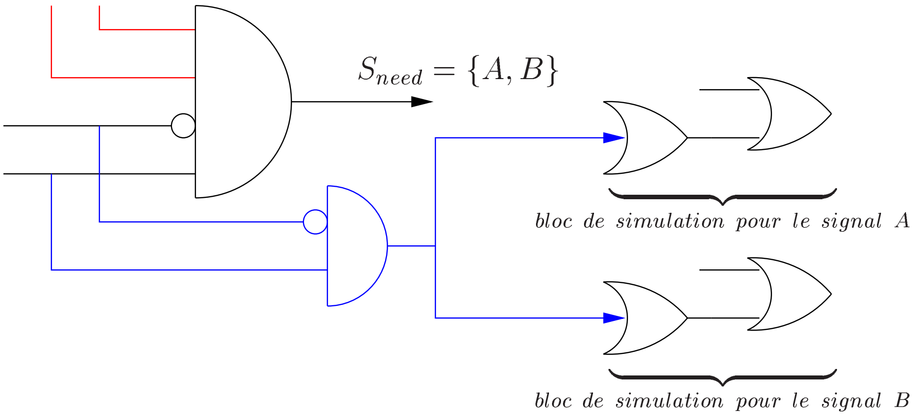


FIG. 7 – Connection des nœuds aux blocs de simulation des signaux dont ils dépendent

Ainsi nous avons pu tester nos programmes distribués de manière graphique comme nous pouvons le voir en Annexe A.3.

6 Conclusion et perspectives

6.1 Le travail réalisé

Nous avons pu mettre en avant la faisabilité d'une répartition automatique de programmes SC. Pour cela nous avons tout d'abord résolu le problème dû à l'indéterminisme de l'ordre d'exécution lors d'actions en parallèle par l'utilisation de files par variables. Nous avons aussi proposé une méthode pour permettre une détection des besoins des tests liés à la présence de signaux. Nous avons finalement réalisé un outil mettant en œuvre ces méthodes, qui nous a permis de valider la faisabilité réelle de la répartition automatique de programmes SC.

Nous avons aussi élaboré un algorithme de résolution des dépendances entre actions et variables localisées. Celui-ci insère des émissions et réceptions en essayant de trouver le meilleur compromis entre la minimisation du nombre des messages et l'utilisation optimale de la latence du réseau.

6.2 Les perspectives

En ce qui concerne la répartition des programmes SC, nous n'avons pas pu utiliser l'algorithme de la partie 4.2. En effet il est très difficile de détecter dans le circuit les structures de contrôles nécessaires à son bon fonctionnement. Notre outil ne permet donc qu'une optimisation lors d'actions se trouvant en séquence. Nous pourrions envisager de trouver une méthode de détection des structures de contrôles, ce qui permettrait de développer de meilleures techniques de résolution des dépendances.

En ce qui concerne la répartition des programmes CP, nous n'avons pas pu mettre en œuvre notre algorithme pour des raisons de propriété industrielle. Nous avons tout de même réussi à obtenir un compromis avec France Telecom pour que l'INRIA puisse développer l'outil Cprep dès qu'un accord entre France Telecom R&D et l'INRIA sera conclu.

6.3 les applications pratiques

La répartition de programmes générés par ESTEREL intéresse tout d'abord les concepteurs d'îlots robotiques. En effet dans le cas de la société Athys, il semble intéressant de créer un seul programme décrivant l'activité d'une chaîne de production et de le répartir ensuite en autant d'automates programmables que celle-ci devra contenir.

D'autres applications non roboticiennes sont apparues telles que la réalisation d'un client-serveur à l'aide du programme détaillant le protocole complet. Ceci permettrait d'appliquer des techniques de preuves de non inter-blocage sans être obligé de modéliser séparément le comportement de chacun des deux logiciels. De plus le format SC attire particulièrement les créateurs de puces électroniques en raison de la possibilité de créer aisément un compilateur de ce format vers silicium. Ainsi la répartition leur permettrait d'alléger la structure physique en scindant leurs programmes en une partie logicielle et une partie physique.

A Exemple : prog

A.1 prog.strl

```
module prog:
input I1, I2;
output O1 : integer, O2 : integer;
var N1 : integer, N2 : integer in
N1 := 0;
N2 := 0;
loop
[
  await I1;
  N1 := N1 + 1;
  emit O1(N1);
]
||
[
  await I2;
  N2 := N2 + 1;
  emit O2(N2);
]
end loop
end var
end module
```

A.2 prog.sc

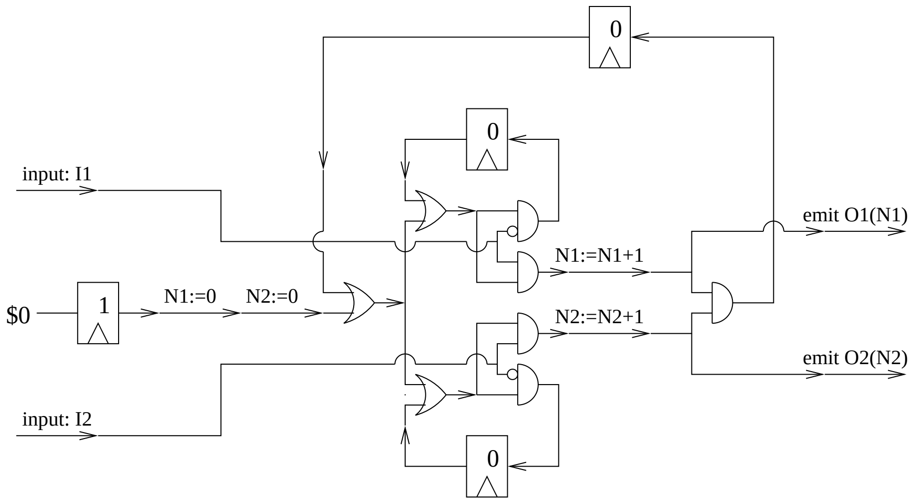
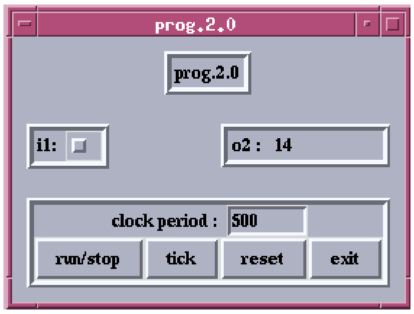
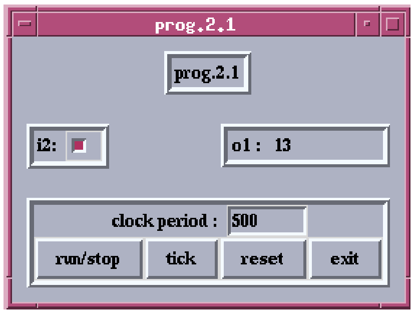


Fig. 8 – Le programme au format SC

A.3 Test des programmes distribués



(a) Le programme du site 0 comportant les signaux i1 et o2



(b) Le programme du site 1 comportant les signaux i2 et o1

FIG. 9 – La simulation des programmes répartis grâce à Asig

Références

[1] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3) :416–427, May/June 1999. Research report INRIA 3491.

[2] G.Berry. *The Sc Format Net Table (version sc6).* CMA, École des Mines et INRIA, Octobre 1995.

[3] G.Berry. The foundations of ESTEREL. *Proof, Language, and Interaction : Essays in Honour of Robin Milner, MIT Press*, 1998.

[4] G.Berry. *The Esterel v5 Language Primer, version 5.91.* CMA, École des Mines et INRIA, Juin 2000. http ://www.esterel.org/.

[5] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of esterel for real-time embedded systems. *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Novembre 2000.