Distribution de programmes synchrones : Le cas d'Esterel

Fabrice Peix

▶ **To cite this version:**

Fabrice Peix. Distribution de programmes synchrones : Le cas d'Esterel. Autre [cs.OH]. Université Nice Sophia Antipolis, 2004. Français. NNT : . tel-00327772

HAL Id: tel-00327772
https://theses.hal.science/tel-00327772v1

Submitted on 9 Oct 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Nice – Sophia Antipolis
UFR Sciences

Ecole Doctorale
Sciences et Technologies de l'Information et de la Communication

Thèse de Doctorat
Pour obtenir le titre de
Docteur en Sciences
de l'UNIVERSITE de Nice-Sophia Antipolis

Discipline : Sciences
Spécialité : Informatique

Préparée à l'Institut National de Recherche en Informatique
et en Automatique de Sophia Antipolis

par

Fabrice PEIX

Distribution de programmes synchrones : Le cas d'Esterel

Directeur de thèse : Robert DE SIMONE

Soutenue publiquement le 6 juillet 2004
à l'Institut National de Recherche en Informatique
et en Automatique de Sophia Antipolis

devant le jury composé de :

Mme.	Laurence PIERRE	Présidente	UNSA
MM.	Jean-Luc DEKEYSER	Rapporteur	LIFL
	Eric RUTTEN	Rapporteur	INRIA / Grenoble
	Jean-Pierre ELLOY	Examinateur	EC / Nantes
	Robert DE SIMONE	Directeur de thèse	INRIA Sophia Antipolis
	Yves SOREL	Examinateur	INRIA / Rocquencourt

Remerciments

Je remercie :
Christian Vander pour sa musique qui m'a soutenu dans ma rédaction.
Eric Vecchié et Loïc Henry-Gréard qui ont supporté les odeurs de tabac dans le bureau.
Robert de Simone qui n'a pas craqué devant mes frasques diverses et variées.
L'ensemble des personnes qui ont, par leurs conseils, fait que la rédaction de cette thèse soit arrivée à son terme.
Mon papa et ma maman qui m'on fait si intelligent.

Table des matières

Table des figures

"C'est à l'heure du commencement qu'il faut particulièrement veiller à ce que les équilibres soient précis"

Princesse Irulan, Manuel de Muad'Dib

Chapitre 1

Introduction et motivations

La présente thèse traite de l'implémentation répartie, sur des plates-formes embarquées, de programmes écrits dans le langage **Esterel**.

1.1 Langages synchrones et *distribution/répartition*

1.1.1 Contexte de réalisation

Le besoin d'avoir des outils permettant une distribution automatique ou semi-automatique de programmes, sur des architectures *matérielles/logicielles* embarquées, ([23]) correspond à l'évolution des infrastructures utilisées pour la réalisation des systèmes embarqués. En effet, la plate-forme d'implémentation utilisée pour la réalisation de certains systèmes embarqués a tendance à s'orienter vers des réseaux de processeurs hétérogènes. Cette évolution peut être rattachée à deux grandes tendances :

- **Minimisation des** *temps de réponses*
  Dans des domaines tels que l'avionique et l'automobile, cette tendance s'inscrit dans un besoin de rapprocher les *centres de calculs* des capteurs physiques afin de minimiser les flux de communication. Cette réduction permet de minimiser à la fois la durée des réactions (*durée entre la réception des entrées et l'émission des sorties*) et la capacité des moyens de communications.
- *software/hardware* **Co-design**
  Désormais une phase importante lors de la conception des systèmes embarqués consiste à déterminer la répartition de la gestion des fonctionnalités entre la partie logicielle et la partie matérielle (*co-design*). En effet, l'évolution permanente et rapide des fonctionnalités offertes par certains processeurs spécialisés (DSP, FPGA, SOC, ...) permet de transférer une partie de l'application de la plate-forme logicielle vers la plate-forme matérielle. Néanmoins la réalisation de tels systèmes est rendue de plus en plus ardue du fait que le nombre de composants et leurs complexités ne cessent de croître.

Actuellement, l'ensemble de ces opérations (design, implémentation *logicielle* versus implémentation *matérielle*, distribution) est réalisé à la "*pogne*". Le problème majeur de cette méthode est que dans le cas où les spécifications attendues ne sont pas au rendez-vous la seule solution, après des semaines, voire des mois d'optimisations acharnées mais inutiles, est de repartir *from scratch*. La réalisation d'un outil permettant de générer et de répartir du code directement à

partir du *design* constitue donc un enjeu majeur.

Toutefois, les difficultés de programmation que rencontrent les concepteurs d'applications distribuées ne constituent pas le problème central de la répartition. En effet, la validation de la mise en œuvre de telles applications est le véritable enjeu. Cette vérification de programmes distribués se heurte, en plus des problèmes classiques, à des problèmes spécifiques tels que : non déterminisme, équité, inter-blocage, ...

1.1.2 Les langages réactifs synchrones

Les langages réactifs synchrones et leurs outils offrent plusieurs avantages dans la réalisation des systèmes embarqués. Le design des systèmes peut être réalisé au moyen d'outils graphiques de haut niveau ([3], [2]). Le design de l'application terminé, la vérification des propriétés de ce dernier peut être réalisée ([10]), la génération du code étant finalement assurée par le compilateur. Ces langages offrent donc une chaîne complète et automatique allant du design jusqu'à l'implémentation, tout en donnant la possibilité d'effectuer des vérifications sur les spécifications de l'application.

Réaliser la distribution d'applications à partir de tels langages permet de tirer partie de leur nombreux avantages (sémantique formelle, outils de vérification, ...) mais aussi de pouvoir utiliser une partie du compilateur existant.

1.1.3 SynDEx

Le placement (distribution) de langages synchrones et plus particulièrement de programme **Esterel** peut être abordée de deux façons différentes. La première consiste à réaliser un logiciel générant directement du code distribué, la seconde consiste à passer par l'intermédiaire d'un logiciel spécialisé dans la distribution de code. L'inconvénient majeur de la première méthode réside dans la difficulté du choix de l'architecture cible, conjugué au surcroît de travail généré par la réalisation de la partie du logiciel réalisant la distribution. On trouvera dans le paragraphe suivant, consacré aux travaux connexes, une évocation des travaux existant sur la distribution de programmes **Esterel** en particulier et synchrones en général. La seconde solution semblait donc la plus raisonnable si nous pouvions trouver un logiciel réalisant de la distribution de programme dans lequel les spécificités d'**Esterel** pourraient être exploitées. Or le logiciel **SynDEx** [26] correspondait entièrement à ces critères. En effet, son atout majeur est de posséder le même modèle sous-jacent pour la représentation des algorithmes puisque le logiciel **SynDEx** utilise le modèle réactif synchrone pour la représentation des algorithmes (discrétisation du temps en instants logiques). Le fait que ce logiciel soit développé à l'**INRIA** a de plus facilité les discussions et les échanges pour la compréhension du fonctionnement interne du logiciel, compréhension indispensable pour réaliser une traduction qui permette à **SynDEx** d'exploiter au mieux les informations fournies par le langage **Esterel**.

1.1.4 Nos apports

Il y a déjà eu dans le passé une tentative pour réaliser un prototype de traducteur **Esterel/SynDEx** [33]. Mais cette traduction se situait au niveau de la représentation *"circuits"* des programmes **Esterel**, qui ne permet pas d'exploiter des informations importantes pour optimiser le *placement/ordonnancement*. C'est justement l'objet de notre étude que d'explorer les voies d'une traduction, depuis un format intermédiaire structuré des programmes **Esterel**

(**GRC**) vers **SynDEx**, qui profite pleinement des possibilités expressives des deux formalismes en améliorant ainsi le *placement/ordonnancement*.

Nous reviendrons brièvement sur les caractéristiques des travaux précédents, avant de considérer leurs faiblesses et les améliorations possibles.

1.1.4.1 Analyse de la précédente traduction

Nous allons analyser la traduction d'**Esterel** vers **SynDEx** existante afin de mettre en lumière les améliorations que fournit notre traduction.

- Parallélisme versus exclusion
  Un des problèmes majeurs lorsque la traduction est réalisée à partir de la représentation sous forme de circuit est que l'exploitation des informations d'exclusion entre les différentes parties du programme est rendue impossible par le modèle d'exécution des circuits qui est basé sur l'activation de l'ensemble du programme. En effet le principe d'évaluation des circuits (propagation des zéros) est de représenter le contrôle comme des données (type booléen), l'ensemble du programme est donc évalué soit par la propagation de valeur *vrai* (la partie est active) soit par la propagation de valeur *faux* (la partie est inactive).
- Séquence causale et flot de données
  La traduction actuelle ne se sert absolument pas du flot de données pour ordonnancer les opérations (actions sur les données). En effet, les dépendances utilisées pour fournir les informations d'ordonnancement au logiciel **SynDEx** sont celles fournies par l'opérateur de séquence causale d'**Esterel**. Or ces dépendances correspondent à une sur-approximation des dépendances réelles qui existent entre les opérations.
- Gestion des signaux
  Dans cette traduction la gestion des signaux et leurs spécificités ne sont pas utilisés puisque le format **DC** a déjà réalisé l'ensemble des traitements et il ne permet pas de façon simple de séparer la partie du circuit relative aux signaux du reste du flot de contrôle. Une gestion spécifique permettrait pourtant de minimiser les dépendances induites par les signaux. En effet, la version circuit étant basée sur l'activation de la totalité du programme (propagation des zéro) elle induit un surcroît de dépendances

1.1.4.2 Améliorations apportées à la traduction

Dans notre traduction, nous allons améliorer chacun des points évoqués ci dessus :

- Parallélisme versus exclusion
  L'expression du parallélisme est importante, afin que le logiciel **SynDEx** l'exploite lors du processus de répartition, mais elle ne doit pas se faire au détriment de l'expression de l'exclusion qui est indispensable pour obtenir une exécution efficace qui n'active pas la totalité du programme à chaque itération (propagation des zéros dans le circuit). Pour cela nous allons exploiter les informations d'exclusion fournies par la structure du programme **Esterel**, et regrouper les instructions au sein de blocs. La structuration de ces blocs est calquée sur celle du programme.
- Séquence causale et flot de données
  La substitution des dépendances de contrôle (celles induites par l'opérateur de séquence causale d'**Esterel**) par les dépendances de données permet d'introduire une forme de parallélisme sans pour autant supprimer l'expression de l'exclusion. Ce traitement constitue un élément important dans notre traduction.

- Gestion des signaux
  Pour que notre traduction fournisse au logiciel **SynDEx** une entrée plus adaptée au placement, elle doit impérativement avoir un traitement spécifique pour les signaux. Notre approche va consister à rajouter des instruction notifiant explicitement la non émission d'un signal. Le point central est que cet ajout d'instructions doit être réalisé en contraignant le moins possible l'ordonnancement.

1.1.5 Travaux connexes

L'étude de la distribution de programme **Esterel** a déjà été abordée dans de nombreux travaux, néanmoins dans l'ensemble de ces travaux le choix a été fait de directement réaliser la distribution sans passer par l'intermédiaire d'un logiciel particulier. On trouve notamment dans [12] et [11] une étude de la distribution de programme **Esterel** à partir de leur représentation sous forme d'automate basée sur les travaux présentés dans [16]. Cette méthode s'est incarnée dans le logiciel **ocrep** ([17]).

Une autre approche a consisté à réaliser des systèmes *GALS* (Globally Asynchronous Locally Synchronous Systems [13] [30]) à partir de la description d'**Esterel**. Nous allons rapidement présenter deux approches relativement semblables qui ont fait le choix de réaliser la distribution à partir de la représentation sous forme de circuits synchrones plutôt que sous forme d'automate. Ce choix a pour principale raison le fait que la représentation sous forme d'automate ne peut pas toujours être réalisée du fait de son coût en mémoire (représentation de l'espace d'état). On trouvera dans [15] des travaux présentant une implémentation des *circuits synchrones constructifs* (qu'ils soient acycliques ou non) comme un réseau de **CFSM** (Codesign Finite State Machines) à l'intérieur de **POLIS** ([4]). Les circuits sont *découpés* en *clusters* (qui sont ensuite implémentés comme des **CFSM**), chacun des ces *clusters* regroupant un certain nombre de portes logiques. Les deux inconvénients majeurs de cette approche sont :

- Les données doivent impérativement être des booléens.
- Elle ne propose pas de méthode pour déterminer la granularité des *clusters*.

On trouve par ailleurs dans [19] et [31] une approche permettant de gérer les *variables valuées* (entiers, flottant, ...) et qui de plus donne une méthode de distribution automatique de circuit, elle est fortement inspirée de [12] et [11]. Par contre, les circuits distribués se doivent, dans cette méthode, d'être acycliques. Cette dernière méthode a donné lieu à la réalisation du logiciel **screp** ([18]). De façon plus générale on trouve dans [6] une étude de la désynchronisation des programmes synchrones.

1.2 Présentation de l'étude

Nous allons présenter dans cette thèse la traduction de programme **Esterel** dans le format utilisé par **SynDEx** pour la représentation des algorithmes. Cet exposé se décomposera en deux parties, la première comportera une introduction aux systèmes réactifs synchrones. Nous nous intéresserons plus particulièrement au cas d'**Esterel** et de **SynDEx**. Cette première partie sera suivie par un intermède mettant en lumière les transformations réalisées par le format **GRC** en les mettant en perspective du futur travail de traduction. La seconde partie constituera l'exposé détaillé des transformations utilisées pour réaliser la connexion entre le langage **Esterel** et le logiciel **SynDEx**. Nous allons maintenant donner un plan détaillé de cette thèse chapitre par chapitre.

Première partie : État de l'art et fondements (chapitres 1 à 5)

Cette première partie se décompose en quatre chapitres. Dans le premier chapitre nous introduirons le paradigme *réactif/synchrone* et nous présenterons les divers modèles mathématiques sous-jacents. Ce chapitre se conclura par une courte présentation des langages, qu'ils soient déclaratifs (*flot de données*) ou impératifs (*flot de contrôle*), dans lesquels le paradigme *réactif/synchrone* s'est incarné.

Le chapitre suivant constituera une présentation (non exhaustive) du langage **Esterel**. On insistera en particulier sur la double spécificité du langage : la *réaction instantanée à l'absence d'un événement* ainsi que la possibilité d'avoir plusieurs émetteurs potentiels pour un même événement. On abordera ensuite les problèmes qu'introduit cette spécificité ainsi que ceux liés à la schizophrénie. On continuera par une courte présentation des principales sémantiques du langage. Nous finirons notre exposé sur le langage **Esterel** par une description de la chaîne de compilation du compilateur **Esterel** développé à l'**INRIA** et au **CMA** (Ecole des Mines de Paris).

Le chapitre 4 présentera en profondeur le format **GRC** à partir duquel nous avons décidé de réaliser la traduction. En effet, ce format intermédiaire, même s'il masque une partie de la structure du programme, est intéressant car il règle certains problèmes tels que la schizophrénie. Après une description détaillée de l'ensemble des structures constituant le format, nous conclurons par un exposé des modifications et extensions à apporter au format afin qu'il satisfasse nos besoins.

Cette première partie se terminera par le chapitre présentant la méthodologie **AAA** ainsi que le logiciel dans lequel elle s'incarne : **SynDEx**. Nous décrirons notamment le format utilisé par le logiciel **SynDEx** pour la représentation des algorithmes, en nous attachant particulièrement à la structure permettant l'expression d'un choix entre plusieurs sous-comportements exclusifs (conditionnement).

Entracte : chapitre 6

Dans cet entracte ne comportant qu'un seul chapitre, nous décrirons les mécanismes généraux sur lesquels repose notre traduction, en revisitant les principes de modélisation de la première partie à la lumière des besoins identifiés et des possibilités algorithmiques.

Deuxième partie : Traduction GRC vers SynDEx (chapitres 7-9)

Le premier chapitre de cette partie présente l'ensemble des modifications techniques, motivées par l'entracte, que nous avons effectuées sur le format **GRC** afin de pouvoir réaliser notre traduction. Ce chapitre se décompose en deux sous-parties. La première aborde notamment la prise en compte des dépendances de données, tandis que la seconde se concentre sur le problème de la *réaction instantanée à l'absence d'un événement* et la manière de le gérer au sein du logiciel **SynDEx**.

Le chapitre 8 présente les détails de la traduction algorithmique du format **GRC** *étendu* vers le format utilisé par **SynDEx** pour la représentation des algorithmes. On y décrit en détail les fonctionnalités algorithmiques, la décommposition de la traduction en phases et les modes de représentations.

Le dernier chapitre de cette partie présente une proposition d'extension au logiciel **SynDEx** qui permettrait une meilleure gestion des variables et des signaux.

Première partie

Etat de l'Art et Fondements

"Qui est le plus fou des deux, le fou ou alors le fou qui le suit ?"

Obi-Wan Kenobi, StarWars IV

Chapitre 2

Modèles synchrones

Nous allons présenter dans ce chapitre les modèles mathématiques sous-jacents au paradigme réactif synchrone ainsi qu'un descriptif rapide des langages dans lesquels il s'est incarné.

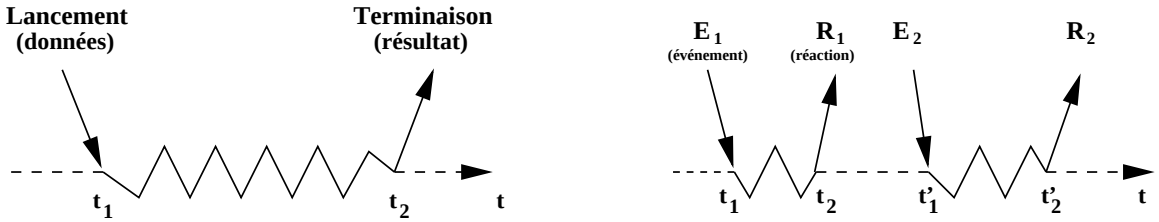
2.1 Le paradigme réactif synchrone

2.1.1 Définition

On trouvera dans [5] [25] une description détaillée de l'approche réactive synchrone. Toutefois on peut présenter les principes fondamentaux :

- *réactif*
  On parle de systèmes *réactifs* par opposition aux systèmes classiques dit transformationnels . On appelle système transformationnel (Figure 2.1(a)), un système qui produit des sorties en fonction de ses entrées avant de se terminer (Ex : un compilateur). De ce fait, les sorties dépendent uniquement des entrées, puisque le système n'a pas d'état interne. Lorsque l'on parle de systèmes *réactif* (Figure 2.1(b)), on parle de système réagissant, indéfiniment, à ses entrées en produisant des sorties. De plus les sorties d'un système réactif ne dépendent pas uniquement de ses entrées mais aussi de son état interne. A chaque réception d'entrée un nouvel état interne ainsi que les sorties sont calculés en fonction des entrées et de l'état interne courant.
- *synchrone*
  L'aspect synchrone affine la notion de systèmes réactifs en définissant la notion d'*instants*. En effet l'exécution de systèmes synchrones (Figure 2.2) se décompose en une succession de réactions (*instants*) délimitées par une horloge globale (le temps est discrétisé en une séquence d'intervalles disjoints et non interruptibles). De plus on suppose que ces instants logiques sont de durée nulle, les sorties du système sont produites à la réception des entrées (la réaction est simultanée aux événements qui la produise).

Il est important de noter que de tels systèmes permettent théoriquement des réactions à l'absence d'événements, puisqu'on peut savoir déterminer le statut cohérent de *présence/absence* de chaque événement pour un instant donné, au contraire des systèmes asynchrones où l'on ne sait pas faire la différence entre absent et *pas encore présent*.



(a) Exécution d'un système transformationnel (b) Exécution d'un système *réactif*

FIG. 2.1 – Système transformationnel vs système réactif

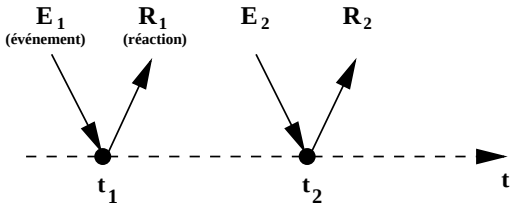
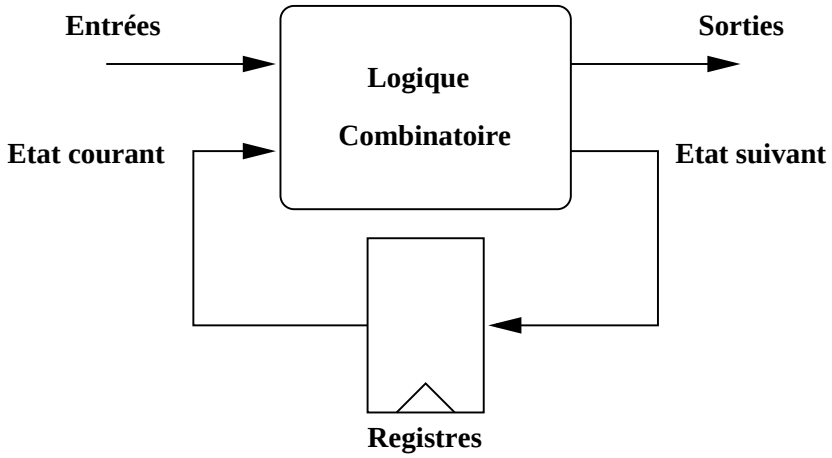


FIG. 2.2 – Systèmes *réactifs synchrones* (Discrétisation du temps en instants de durée nulle)

2.1.2 Modèles mathématiques

2.1.2.1 Les schémas de portes logiques (netlist)

Un des modèles naturellement utilisé pour la sémantique des systèmes *réactifs synchrones* correspond à un réseau de portes logiques (*circuits synchrones*), et dénommé usuellement *netList* dans la communauté de la conception de circuits numériques. Dans ce modèle le système est représenté par une liste d'équations définissant les valeurs de variables et de registres booléens comme fonctions logiques d'autres valeurs. Il existe quatre types d'objets (variables) constituant ces équations :



2.3: Représentation schématique d'un *circuit séquentiel synchrone*

- **entrées**
  Ces variables correspondent aux entrées du système, elles ne peuvent pas être affectées.
- **sorties**
  Ces variables correspondent aux sorties du système, elles ne peuvent pas être lues.
- **registres**
  Ces variables sont d'un type particulier puisqu'elles servent à encoder l'état du système de manière implicite.
- **variables** *locales*
  Ces variables correspondent à des résultats intermédiaires.

Il est important de noter que dans ce modèle on applique le principe d'assignation unique (*Single Static Assignement*) au sens où une variable représente la valeur d'une *entrée/sortie* d'une porte logique et ne prend qu'une valeur *logique/électrique* à chaque cycle. Dans la plupart des implémentations des *systèmes réactifs synchrones* les *cycles* combinatoires sont interdits. On définit comme *cycle* combinatoire une interdépendance entre variables dont aucune n'est un *registre*.

2.1.2.2 Les automate de Mealy

Un des modèles mathématiques de référence sur lequel repose l'approche réactive synchrone correspond aux automates de **Mealy** (qui sont eux-mêmes un modèle opérationnel classique d'interprétation des circuits synchrones). On rappelle la définition d'un automate de **Mealy** :

Un automate de **Mealy** est représenté par un n-uplet de la forme :

$$< I, O, S, \delta, \lambda > \text{ avec :}$$

I	: Un ensemble fini d'éléments appelés **signaux d'entrée**
O	: Un ensemble fini d'éléments appelés **signaux de sortie**
S	: Un ensemble fini d'éléments appelés **états**
$\delta : I \times S \rightarrow S$	: Une fonction appelée **fonction de transition**
$\lambda : I \times S \rightarrow O$	: Une fonction appelée **fonction de sortie**

Il est important de noter que dans cette représentation il n'y plus de variables locales et que l'ensemble des problèmes liés à la causalité ont obligatoirement été résolus (cycles sans registre). Plus précisément, la représentation explicite des états impose que les problèmes liés à la causalité soient résolus lors du calcul et de l'énumération des états atteignables (compilation). La représentation sous forme d'automates permet aussi d'exprimer directement la valeur des *sorties* en fonction de l'état courant et des *entrées* (absence de variables locales).

2.2 Les styles de langages

La programmation synchrone s'est incarnée dans plusieurs langages, on peut toutefois regrouper ces langages dans deux grandes familles.

2.2.1 Le style impératif

Cette famille de langages synchrones est celle qui se rapproche le plus des langages informatiques classiques. Elle se caractérise par une syntaxe impérative et des programmes se découpant en deux parties distinctes, la première regroupant l'ensemble des déclarations tandis que la seconde constitue le corps du programme (ensemble d'instructions constituant le programme).

Ces langages s'inscrivent dans une mise en avant de la partie contrôle, la partie traitement de *données* étant entièrement externalisée. En effet pour chaque opération présente dans le programme on définira une sortie qui conditionnera l'activation de l'opération. Une procédure identique sera réalisée pour les tests, au détail près que l'on associera une entrée/sortie pour permettre la lecture du résultat du test.

2.2.2 Le style déclaratif

Cette classe de langages peut être rapprochée des langages fonctionnels. En effet comme dans les langages fonctionnels, et à la différence de la classe des langages impératifs qui privilégient le contrôle, le traitement des données est placé au centre de ces langages.

Ces langages ont certaines propriétés spécifiques, telles que le respect du principe **SSA** (**S**ingle **S**tatic **A**ssignement). Plusieurs langages correspondent à ce style (**Lustre** [24] et **Signal** [27]). Le format utilisé, dans **SynDEx**, pour la représentation des algorithmes est aussi une incarnation de ce style de langage. Nous présenterons le logiciel **SynDEx** et son format au Chapitre 5.

"Etre ou ne pas être ! c'est là la question"

Un signal schizophrène

Chapitre 3

Le langage Esterel

Le langage **Esterel** a été conçu en 1982 par deux chercheurs, Jean-Paul Marmorat et Jean-Paul Rigault, au Centre de Mathématiques Appliquées de l'Ecole des Mines de Paris. La principale raison qui a poussé ces chercheurs à concevoir ce nouveau langage était le manque d'expressivité des langages existants en matière de contrôle et de contraintes temporelles. Peu de temps après, Gérard Berry a rejoint l'équipe avec comme première ambition de donner une sémantique rigoureuse au langage. Il a par la suite pris la tête de l'équipe et a dirigé les évolutions du langage et de son compilateur. Aujourd'hui la dernière version du compilateur (*v7*) est développée au sein de l'entreprise Esterel-Technologies à Sophia-Antipolis. La description qui suit ainsi que les travaux effectués ont été réalisés à partir de la version *v5* du compilateur, développée à l'**INRIA** et au **CMA** dans les équipes *Meije* puis *Tick*.

3.1 Description rapide du langage Esterel

3.1.1 Un petit exemple

```
module ABRO
input A,B,R
output O
loop
     await A
   ||
     await B;
   emit O
each R
end module
```

FIG. 3.1 – Un exemple de programme **Esterel** *ABRO*

L'exemple de la Figure 3.1 permet d'introduire la majorité des constructions du langage. Nous allons en faire une brève description qui sera suivie d'une explication détaillée de l'ensemble des instructions dont dispose le langage ([8]).

La construction englobante **loop** p **each** R permet de relancer l'exécution de p à chaque itération de R (préemption forte). Dans notre exemple le corps p correspond aux instructions :

await A
$\|$
await B **;**
emit O

La construction **await** $A \parallel$ **await** B permet d'attendre en parallèle les évènements (signaux) A et B, lorsque ces deux événement sont survenus et si l'événement R n'a pas eu lieu, l'instruction **emit** O (émission du signal O) est exécutée en séquence (**;**).

3.1.2 Modularité

Ce langage est impératif et offre la possibilité de décomposer les programmes sous forme de modules. Les modules sont divisés en deux parties, la première définissant l'interface de déclaration du module tandis que la seconde représente le corps du module à proprement parler (Figure 3.2).

module
EXEMPLE
input *noms* ;
output *noms* ;
corps du module
end module

FIG. 3.2 – Structure des modules

Le langage **Esterel** fournit au programmeur un ensemble de structures de contrôle de haut niveau pour la réalisation de systèmes réactifs synchrones. Les instructions constituant le langage sont de deux types :

- Les instructions de contrôles : ces instructions permettent de définir la partie *flot de contrôle* du programme.
- Les instructions de données : ces instructions permettent la manipulation de données (entier, flottant) et les opérations élémentaires à travers des traitements externes non décrits dans le programme **Esterel**.

3.1.3 Les instructions du langage

En plus des structures classiques des langages impératifs (**loop**, **if then else**, ...), **Esterel** offre la possibilité d'exprimer du parallélisme d'expression. Ce parallélisme est un élément important de modularité des processus synchrones. La communication entre les modules, ainsi mis en parallèle, est assurée au moyen d'envoi et de réception de signaux. L'envoi de ces signaux s'effectue par une diffusion (*broadcast*) instantanée à l'ensemble des autres modules.

3.1.3.1 Les instructions de contrôle

Dans les lignes qui suivent p et q font références à des instructions (ou ensemble d'instructions).

- **nothing**
  Cette instruction ne fait rien, elle passe le contrôle instantanément.
- **emit** S
  Emission du signal S.
- **pause**
  Stoppe l'exécution du programme pour l'instant courant. L'exécution du programme sera reprise à l'instant suivant. En d'autre termes, les **pause** permettent de découper le temps en instants logiques.
- **present** S **then** p **else** q **end**
  Teste la présence du signal S. Si le signal S est présent on exécute p, sinon on exécute q. La réaction instantanée à l'absence d'un signal (exécution de la branche *else*) constitue un des aspects spécifiques des langages synchrones.
- **suspend** p **when** S
  Suspend l'exécution de p lorsque S est présent, sauf lors du premier instant d'exécution de p.
- p **;** q
  Opérateur de séquence, passe instantanément le contrôle à q lorsque p se termine.
- **loop** p **end**
  Boucle infinie. L'instruction ou le bloc d'instruction p ne doit pas être instantané (toute exécution de p doit exécuter un **pause**).
- $p \parallel q$
  Opérateur parallèle. La composition de p et de q se termine lorsque p et q sont terminés.
- **trap** T **in** p **end**
  Le bloc se termine lorsque l'instruction p se termine ou lorsque l'interruption T est levée.
- **exit** T
  Instruction permettant de lever l'interruption T.
- **signal** S **in** p **end**
  Définition du signal local S dont la visibilité est restreinte à l'instruction p.

En plus de ces *briques* de base le langage offre la possibilité d'utiliser des opérations un peu plus évoluées tel que **await**, **abort** p **when** S, ... On trouvera dans 3.1.6,p.28 de plus amples imformations sur les instructions dérivées ainsi que leur expansion en langage *noyau*.

3.1.3.2 Les instructions de données

Il faut noter que bien que le langage permette la manipulation de données, l'interprétation des opérations sur les valeurs est complètement externalisée dans un langage généraliste. Plus précisément, aussi bien au niveau du formalisme que de l'implémentation, la partie effectuant effectivement les calculs sur les données correspond à une série de procédures et de fonctions pilotées par la partie contrôle du programme.

- **emit** S(*exp*)
  Cette instruction permet l'émission d'un signal valué. La valeur associée au signal est obtenue par **?***S*. Il faut noter que l'opérateur **?** implique que la valeur portée par le signal soit connue, c'est à dire que l'ensemble des émissions soit réalisé.
- X **:=** *exp*
  Affectation de *exp* à la variable X.
- **call** procédure (liste arguments passés par valeur)
  (liste arguments passés par référence)

Appel à une procédure externe.

– **if** *exp* **then** *p* **else** *q* **end**
L'expression *exp* doit être de type booléen. Si *exp* est vraie l'instruction *p* est exécutée, sinon on exécute *q*.

– **var** X : type **in** *p* **end**
Déclaration d'une variable locale dont la visibilité est restreinte à l'instruction *p*.

Les accès concurrents aux variables ne sont autorisés que dans le cas de lecture. Les programmes suivants illustrent les cas interdits :

Accès concurrent en écriture	Accès concurrent en lecture/écriture	Accès syntaxiquement concurrent en lecture/écriture [a]

x := 4
‖
x := 7
	x := 4
	‖
	y := x
		if i = 0 **then**
		x := 3
		end
		‖
		if i ≠ 0 **then**
		x := y
		end

[a] Si on n'interprète pas les tests de branchement

On notera que le dernier exemple est considéré comme incorrect du fait que la vérification s'effectue de manière syntaxique, l'exclusion entre les deux affectations n'est donc pas prise en compte.

3.1.4 Les signaux

Les signaux ont deux rôles, un pour la partie *contrôle* et un pour la partie *donnée*. Au niveau contrôle ils permettent l'activation conditionnée d'une partie du programme (**present then else**) et une certaine synchronisation (**await**). Au niveau donnée ils sont aussi l'unique moyen d'échanger des données valuées entre des modules mis en parallèles (**emit** S(*exp*)). Il est important de noter que ces données valuées se doivent d'être uniques, cohérentes, et calculées par une combinaison des émissions dans l'instant. Avec les conditions d'interdiction d'affectations concurrentes, ceci garantit l'exécution déterministe du programme.

3.1.5 Appel à des *sous*-modules

L'appel à des *sous*-modules (Figure 3.3) se fait au moyen d'une instruction spécifique.

– **run** *module_name* [*liste de renommage*]

Les renommages de noms de signaux servent à établir les connexions adéquates par nommage (par défaut un signal garde son nom).

3.1.6 Les instructions dérivées

En plus des instructions de base le langage offre un ensemble de primitives utiles pour la programmation qui peuvent être définies par macro-expansion. On trouvera dans la Figure 3.4

```
module M1
input I;
output O;
present  I then
   emit  O
end present
end module
```

```
module MAIN
interface
⋮
signal S1,S2 in
  ⋮
   run M1 [S1/I,S2/O])
  ⋮
end signal
end module
```

FIG. 3.3 – Appel à un sous module avec renomage de son interface

quelques instructions dérivées ainsi que leur expansion en **Esterel** *pur*. L'intérêt de définir de telles instructions est à la fois d'offrir des fonctionnalités de plus haut niveau mais aussi de pouvoir effectuer certaines optimisations lors de la compilation (en traduisant directement ces constructions fréquentes et intuitives plutôt que de les expanser).

3.1.7 Syntaxe du calcul de processus d'Esterel

Le langage **Esterel**, bien qu'adapté pour la programmation, est trop lourd pour la définition de règles de sémantique. Nous commencerons donc par définir une nouvelle syntaxe (équivalente) permettant d'écrire simplement des programmes **Esterel**. Cette syntaxe (Tableau 3.1) se rapproche de celle utilisée dans les calcul de processus. La principale différence avec la syntaxe du langage de programmation est la suppression du principe de nommage des exceptions (**trap**) et l'introduction d'un système de *code de complétion* servant à encoder les instructions **nothing**, **pause** et **exit**. On notera que le remplacement des *noms* des **trap** par un système de numérotation implique l'introduction d'un nouvel opérateur *shift* ($\uparrow$). En effet cette numérotation s'effectue de façon structurelle en fonction du *niveau d'imbrication* de l'exception levée.

nothing	0
pause	1
emit S	$!s$
present S **then** p **else** q **end**	$s?p,q$
suspend p **when** S	$s \supset p$
p **;** q	$p;q$
loop p **end**	p^*
$p \parallel q$	$p \mid q$
trap T **in** p **end**	$\{p\}$
	$\uparrow p$
exit T	k with $k \geq 2$
signal S **in** p **end**	$p \backslash s$

TAB. 3.1 – Syntaxe du calcul de processus d'**Esterel**

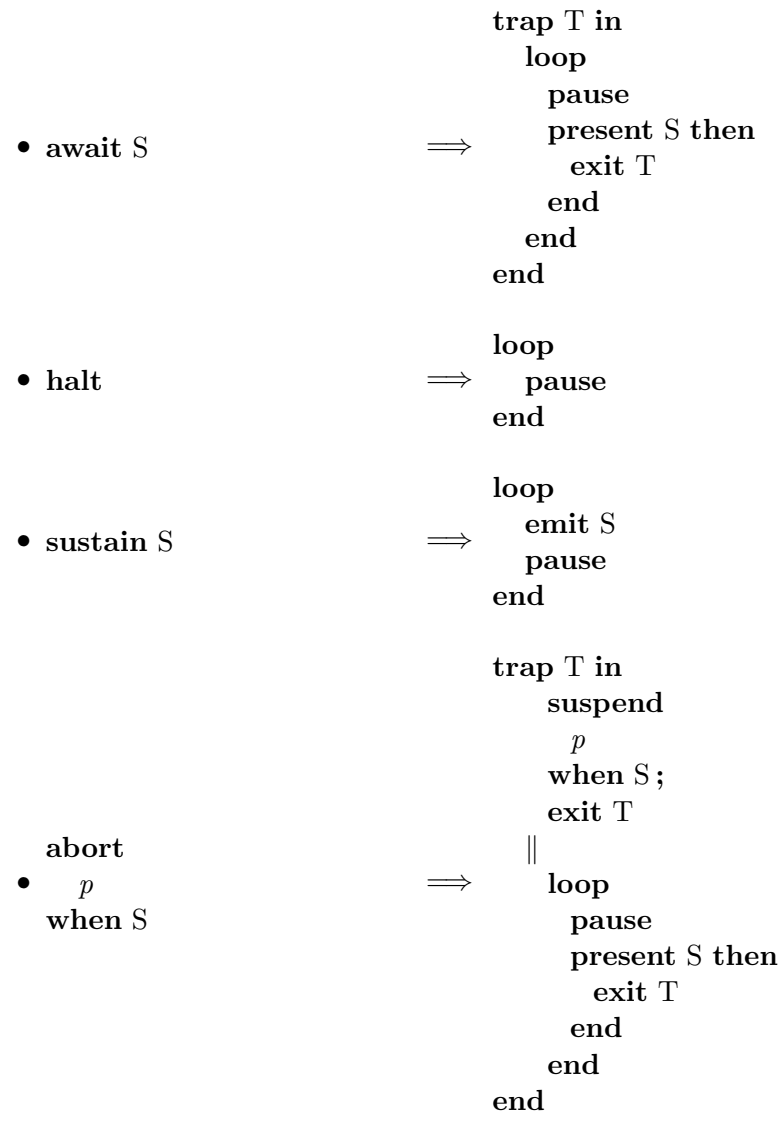


FIG. 3.4 – Liste non exhaustive d'instructions dérivées

3.1.7.1 Les codes de complétion

Nous allons brièvement décrire l'encodage des instructions **Esterel** et le système de code de complétion utilisé par cette nouvelle syntaxe. L'instruction **nothing** est encodée par **0**, l'instruction **pause** par **1** et les instructions de type **exit** T par **2** si T correspond au bloc **trap in end** le plus englobant. Dans le cas contraire l'instruction **exit** T s'encode par $n + 2$, n correspondant au nombre de blocs **trap in end** englobant le bloc correspondant à T. La figure 3.5 illustre le principe de traduction que l'on vient de décrire.

Le principe des codes de complétion est le suivant : à chaque instant les *threads de contrôle* retournent un code de complétion $k \geq 0$ lorsque leurs exécutions sont terminées (pour l'instant courant). Il faut noter que seules les instructions **nothing**, **pause** et **exit** génèrent effectivement des codes de complétion. Le code de complétion d'un bloc *parallèle* correspond au calcul du *max* des codes de complétion retournés par chacune des branches du parallèle. Plus précisément cela correspond aux règles suivantes :

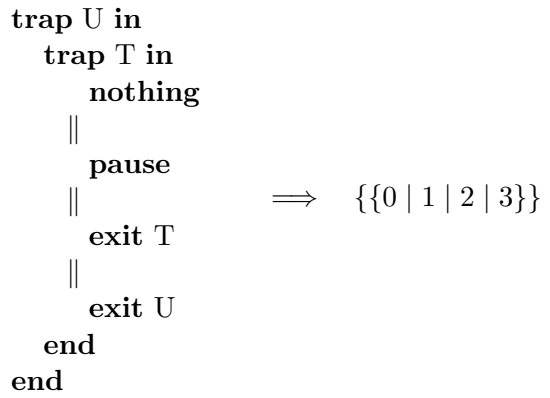


FIG. 3.5 – Illustration du système de code de complétion

Soit un bloc *parallèle* ayant n branches, on notera k_i le code de complétion de la branche i.

- Un bloc parallèle se termine si ses branches ont toutes terminé, ce qui correspond à : $max(k_0, k_1, \ldots, k_n) = 0$ ce qui est équivalent à $k_0 = k_1 = \ldots = k_n = 0$
- Un bloc parallèle *pause* si une de ses branches effectue une *pause* et si aucune n'émet d'exception (**trap**) ce qui correspond à : $max(k_0, k_1, \ldots, k_n) = 1$
- Un bloc parallèle émet une exception si au moins l'une de ses branches émet une exception, dans le cas où plusieurs branches émettent des exceptions le bloc parallèle émet l'exception correspondant au bloc **trap in end** le plus englobant.

3.1.7.2 L'opérateur *shift* (↑)

La perte des noms des **trap** lors de la traduction implique l'introduction d'un opérateur spécifique ($\uparrow p$). Cet opérateur ayant pour effet d'incrémenter de un l'ensemble des codes de complétion supérieurs à un émis par p. Cet opérateur permet ainsi d'assurer que $\{\uparrow p\}$ a un comportement équivalent à p.

3.2 Schizophrénie

3.2.1 Le problème

Le langage **Esterel** impose un certain nombre de restrictions (récursivité générale non présente, pas de création dynamique de parallèle, ...) afin de garantir une structure de contrôle statique. On pourrait penser qu'une conséquence de cette hypothèse est que chaque instruction ou opération n'est exécutée qu'une seule fois à chaque réaction. Ceci n'est pas tout à fait exact pour des raisons que nous allons développer. Néanmoins le nombre de fois où une instruction peut être (ré)exécutée au sein d'un même instant est fini et borné. On peut transformer les programmes **Esterel** pour que cette propriété soit vraie ce qui permet par la suite de faciliter les analyses et transformations effectuées sur le programme. Il est important de noter que ce problème apparaît lors d'une traduction structurelle de certains programmes **Esterel** (format **sc**, **grc**, ...). En effet dans l'exemple de la Figure 3.6 le signal S est dit *schizophrène* car dans une même réaction deux instances différentes du signal S peuvent coexister et avoir des statuts de présence différents. De manière plus générale il existe aussi des programmes dont une partie

du code est exécutée plusieurs fois au cours du même instant ce qui nécessite un traitement similaire.

```
module EXEMPLE :
input I;
loop
  signal S in
    present I then
      emit S
    end present
    pause;
    emit S
  end signal
end loop
end
```

FIG. 3.6 – Exemple de signal schizophrène

La Figure 3.7 illustre l'évaluation pas à pas d'une réaction du programme de la Figure 3.6. Dans la description de chaque étape qui suit, on appellera S_{old} l'instance du signal S existant au début de la réaction et S_{new} l'instance du signal S créée au cours de la réaction. De plus on suppose que lors de cette réaction le signal d'entrée I est absent.

- **Etape 1**
  Début de la réaction, l'instruction pause se termine.
- **Etape 2**
  Emission du signal S_{old}.
- **Etape 3**
  Destruction de l'instance S_{old} du signal S, fin de la visibilité du signal S et fin de boucle.
- **Etape 4**
  Nouvelle itération du loop, et création de l'instance S_{new} du signal S .
- **Etape 5**
  Test de présence du signal d'input I.
- **Etape 6**
  Fin de la réaction.

Au cours de cette réaction l'instance du signal S_{new} est absente alors que l'instance du signal S_{old} était présente.

3.2.2 Une solution : La réincarnation

Une solution *purement syntaxique* consiste à réécrire les **loop** afin de différencier l'instant initial d'activation de l'ensemble des autres instants. La Figure 3.8 illustre une réécriture possible.

Bien que correctes, ces transformations ont une complexité excessive dans l'ajout d'instructions, c'est pour cette raison que des techniques moins coûteuses, en termes de duplication de

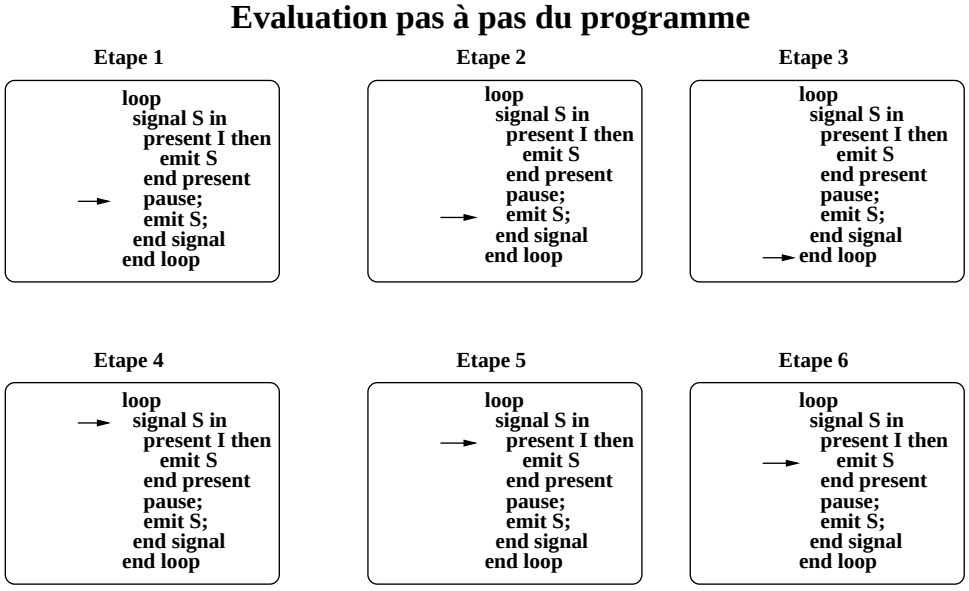


FIG. 3.7 – Evaluation pas à pas sur un cycle de réaction du programme de la Figure 3.6

$$\textbf{loop } p \textbf{ end loop} \longrightarrow \textbf{loop } p \textbf{; } p \textbf{ end loop}$$

FIG. 3.8 – Réécriture des **loop** pour résoudre les problèmes de schizophrénie

code, ont été etudiées. Intuitivement on se propose de ne dupliquer que les instructions posant problème, pour cela on définit deux notions permettant de caractériser certaines parties du corps des **loop** :

- **surface**
  Partie du corps d'un *loop* activée dans l'instant initial de son activation (ne comprend aucun **pause**).
- **profondeur**
  Partie du corps d'un *loop* activée lors des autres instants.

La **surface** est donc constituée uniquement d'instructions instantanées (pas d'instruction **pause**). On notera que la surface et la profondeur ne sont pas forcément disjointes. Le principe de la réincarnation est basé sur la séparation de la surface et de la profondeur permettant dans le cas de l'exemple de la Figure 3.6 de distinguer les deux intances du signal S (S_{old} et S_{new}). Il faut noter que dans le cas de programmes plus complexes faisant intervenir des **trap**, on peut être amené à effectuer une duplication de la surface, ce qui permet de gérer plus de deux instances du même signal.

Le problème de la schizophrénie n'apparaissant pas dans l'ensemble des boucles on peut généralement éviter de dupliquer la **surface** d'un **loop**. On pourra trouver dans [9] et [35] le détail des méthodes actuelles permettant de déterminer les cas ou l'on peut ne pas effectuer de duplication de la surface.

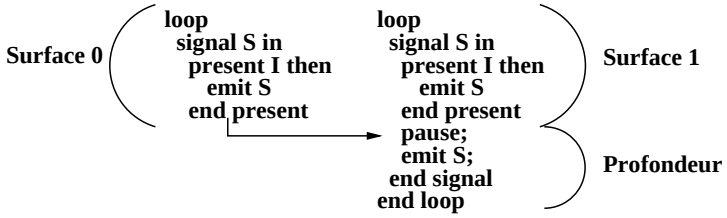


FIG. 3.9 – Séparation de la surface et de la profondeur

3.3 Absence et réaction instantanée à l'absence

Une des caractéristiques spécifiques du langage **Esterel** et plus généralement de l'hypothèse synchrone est le fait de pouvoir réagir à l'absence d'un signal. De plus il faut noter que cette réaction se fait de manière instantanée, cette construction est importante car elle permet de gérer les priorités et les préemptions. Dans le cas du langage **Esterel** ce problème est compliqué par la possibilité offerte par le langage d'avoir plusieurs émetteurs potentiels d'un même signal. Cette caractéristique impose, pour déduire qu'un signal est absent, de savoir déterminer, lors de l'exécution, quand plus aucune de ces émissions n'est exécutable dans la réaction courante. Nous verrons que dans un cadre séquentiel cette difficulté est dans la plupart des cas résolue par l'ordonnancement. Ceci n'est plus forcément vrai lorsque l'objectif est de distribuer le code réalisant l'évaluation des réactions.

Néanmoins ceci a pour conséquence de rendre possible l'écriture de programmes syntaxiquement corrects mais n'ayant aucun sens (*causalement* incorrects) :

```
present S then
   nothing
else
   emit S
end
```

De même cela introduit aussi la possibilité d'obtenir des programmes incorrects par composition parallèle de *modules* corrects (3.10). Ce dernier point interdisant la possibilité de vérifier la correction d'un programme de façon modulaire.

En fait ces deux exemples s'inscrivent dans un problème plus général qui revient à déterminer si le programme est *causal*. Le chapitre 3.4,p.34 explicite précisément la notion de programmes *réactif*, *déterministe*, *fortement déterministe* et *constructif* (correct du point de vue de la sémantique).

3.4 Sémantique SOS et causalité constructive

3.4.1 Sémantique comportementale logique

Cette sémantique directe donne une définition structurelle et opérationnelle (par opérateurs) du comportement réactif des programmes. Néanmoins elle est trop abstraite et ne prend pas en compte les problèmes de causalité. Cette sémantique sera par la suite raffinée dans une seconde sémantique (*sémantique constructive* 3.4.2,p.37) de façon à ce que les deux sémantiques coïncident sur les programmes causaux. La description précise et complète de cette sémantique sort du cadre de ce document, nous nous contenterons d'une introduction permettant de mettre

```
module M1 :
input S1 ;
output S2 ;
present S1 then
  emit S2
end
```

```
module M2 :
input S2 ;
output S1 ;
present S2 then
  emit S1
end
```

```
module MAIN :
interface
⋮
signal S1,S2 in
    run M1
  ||
    run M2
end signal
⋮
end module
```

FIG. 3.10 – Composition incorrecte de modules corrects

en lumière les notions importantes qu'elle définit. On trouvera dans ([7]) une description formelle et complète de cette sémantique. Nous commencerons par définir la notion d'événements puis nous exposerons précisement la manière d'interpréter les règles de réécriture.

3.4.1.1 Evénements

On définit un évènement E, comme la définition du statut de présence b ($b \in B = \{+, -\}$) pour chaque signal s appartenant à un ensemble de signaux S. On notera $E(s)$ le statut de s dans E, de plus on écrira $s^+ \in E(respectivement\ s^- \in E)$ si le statut de s dans E est $+$ (respectivement $-$). D'un point de vue opérationnel si $s^+ \in E(respectivement\ s^- \in E)$ cela veut dire que le signal S est déterminé comme présent (respectivement déterminé comme absent). Pour deux évènements E et E', on pourra écrire $E' \subset E$ si et seulement si $s^+ \in E' \Rightarrow s^+ \in E$.

Soit un évènement E d'un ensemble S, un signal s qui peut ne pas appartenir à S, et un statut de présence b appartenant à B. On définit l'opérateur $*$ tel que :

$E' = E * s^b$ est un événement dont l'ensemble S' est défini par $S' = S \cup \{s\}$. On a $E'(s) = b$ et $E'(s') = E(s')$ *pour* $s' \neq s$

On notera que l'introduction des événements dans la description de la sémantique logique n'est pas indispensable (la notion d'ensemble est suffisante). Néanmoins cette démarche rendra la transition de la sémantique logique vers la sémantique constructive plus aisée et plus pédagogique.

3.4.1.2 Format et interprétation des règles

Les sémantiques comportementales définissent une réaction d'un programme avec des règles de la forme :

$$P \xrightarrow[I]{O} P'$$

où O et I sont respectivement les événements d'entrée (*Input*) et les événements de sortie (*Ouput*). Le programme P' correspond au nouvel état atteint par P à la fin de la réaction. Plus précisément les relations représentant les transitions d'un état à un autre se décomposent en règles de la forme :

$$p \xrightarrow[E]{E',k} p'$$

On définit E comme étant l'événement définissant le statut des signaux *déclarés* dans le bloc p, E' est l'événement de l'ensemble des signaux émis par p lors de la réaction et pour finir k est le code de complétion de p pour cette réaction.

Étant donné un programme P dont le corps est p, on a la relation suivante :

$$P \xrightarrow[I]{O} P' \qquad \Longleftrightarrow \qquad p \xrightarrow[I \cup O]{O,0} p'$$

Définition 3.1 *Le programme P est réactif (respectivement déterministe) pour un ensemble d'événements d'entrée I s'il existe au moins (respectivement au plus) une transition $P \xrightarrow[I]{O} P'$. Le programme est correct pour un ensemble d'événements d'entrée I s'il est à la fois réactif et deterministe pour ce même ensemble d'événements d'entrée.*

3.4.1.3 Quelques exemples de règles

$$!s \xrightarrow[E]{\{s^+\},0} 0 \qquad \textit{(emit)}$$

$$\frac{s^+ \in E \quad p \xrightarrow[E]{E',k} p'}{s?p,q \xrightarrow[E]{E',k} p'} \qquad \textit{(present +)}$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad k \neq 0^a}{p* \xrightarrow[E]{E',k} p';p*} \qquad \textit{(loop)}$$

[a] Ceci permet d'éviter les boucles instantanées

3.4.1.4 Réactivité et déterminisme

Il faut noter que la définition de la correction d'un programme (le programme est réactif et déterministe) laisse un certain degré de non déterminisme dans la manière d'appliquer les règles. En effet le programme $(s?!s,0)\backslash s$ se réécrit en $0\backslash s$, néanmoins il est non déterministe dans le choix des règles à appliquer puisque le signal local s peut être émis ou ne pas être émis. Dans la sémantique constructive un tel indéterminisme interne sera interdit. On définit donc les programmes fortement déterministes comme :

Définition 3.2 *Un programme P est fortement déterministe pour un ensemble d'entrée I s'il est réactif et déterministe pour I et s'il existe une preuve unique de la transition $P \xrightarrow[I]{O} P'$.*

3.4.2 Sémantique comportementale constructive

La sémantique constructive (définie dans [7]) utilise des règles similaires à la sémantique logique (3.4.1,p.34), mais elle permet d'éviter de faire des suppositions sur le statut de présence des signaux en les raffinant par l'ajout de prédicats permettant de déterminer effectivement si un signal doit (*Must*) ou ne peut (*Cannot*) être émis. Plus précisément, c'est une approche plus opérationnelle qui effectue une exécution symbolique permettant de déterminer au fur et à mesure de son avance les signaux ne pouvant plus être émis (en fonction des informations déjà déterminées) ; ceci est réalisé en déterminant si les parties encore activables au cours de l'instant peuvent potentiellement émettre tel ou tel signal. Soit un événement d'entrée, si le calcul permet de déterminer un statut de présence pour l'ensemble des signaux, le programme sera déclaré constructif pour cet événement. Si le calcul échoue, le programme sera déclaré non constructif et donc incorrect.

3.4.2.1 Les fonctions *Must*, *Can* et *Cannot*

La fonction *Must* détermine ce qui doit être effectué dans la réaction $P \xrightarrow[O]{I} P'$. La fonction *Must* a la forme suivante :

$$Must(p, E) = < S, K >$$

où E est un événement partiel qui associe un statut $B_\perp = \{+, -, \perp\}$ à chaque signal, S est l'ensemble des signaux que p doit émettre et K est l'ensemble des codes de complétion que p doit retourner. L'ensemble K est soit vide si on ne peut déterminer aucune information, soit un singleton $\{k\}$ si p doit retourner k. Il faut noter qu'il est impossible que p doive retourner plus d'un code de complétion. On introduit les fonctions $Must_s$ et $Must_k$ permettant de calculer les deux éléments de la paire calculée par $Must$. On a donc :

$$Must(p, E) = < Must_s(p, E), Must_k(p, E) >$$

La fonction $Cannot^m$ est utilisée pour déterminer les parties non actives du programme, elle se décompose elle aussi en deux sous fonctions :

$$Cannot^m(p, E) = < Cannot^m_s(p, E), Cannot^m_s(p, E) > = < S, K >$$

Pour cette fonction S et K représentent respectivement l'ensemble des signaux que p ne peut pas émettre et l'ensemble des codes de complétion que p ne peut pas retourner lorsque l'évènement d'entrée est E. L'argument supplémentaire m ($m \in \{+, \perp\}$) indique si p doit être exécuté ($+$) ou s'il peut être exécuté ($\perp$)[1] lorsque p est exécuté dans l'environnement E. Il faut noter que le cas $m = -$ ne peut pas arriver puisque *Cannot* ne sera appelé que sur les *instructions* potentiellement exécutées. Dans la pratique on utilise plutôt la fonction $Can^m(p, E)$ que l'on définit comme le complément de $Cannot^m(E, p)$. Ce complément est défini comme le complément de chacune des fonctions constituant $Cannot^m(E, p)$, l'univers de la première fonction correspondant à l'ensemble des signaux visibles tandis que le second correspond au code de complétion potentiellement retourné par p.

[1] le cas $m = -1$ n'a pas de sens puisque $Cannot^m$ n'est exécuté que pour les parties potentiellement actives du programme

3.4.2.2 Définition partielle des fonctions *Must* et *Can*

- **Retour de code de complétion**
  $Must(k, E) = Can^m(k, E) = < \emptyset, \{k\} >$
- **Emission de signal**
  $Must(!s, E) = Can^m(k, E) = < \{s\}, \{0\} >$
- **present**

$$Must\left((s?p, q), E\right) = \begin{cases} Must(p, E) & si\ s^+ \in E \\ Must(q, E) & si\ s^- \in E \\ < \emptyset, \emptyset > & si\ s^\perp \in E \end{cases}$$

$$Can^m\left((s?p, q), E\right) = \begin{cases} Can^m(p, E) & si\ s^+ \in E \\ Can^m(q, E) & si\ s^- \in E \\ Can^\perp(p, E) \cup Can^\perp(q, E) & si\ s^\perp \in E \end{cases}$$

On trouvera dans [7] l'intégralité de la définition des fonctions *Must* et *Can*.

3.4.2.3 Utilisation des fonctions *Must* et *Cannot* dans les règles de la sémantique logique

Les règles de réécriture de la sémantique constructive sont les mêmes que les règles de la sémantique comportementale logique excepté pour les règles régissant les signaux locaux. En effet ces règles deviennent :

$$\frac{s \in Must_s(p, E \cup \{s^\perp\})\ p \xrightarrow[E \cup \{s^+\}]{E' \cup \{s^+\}, k} p' \quad s \notin S(E')}{p/s \xrightarrow[E]{E', k} p'/s} \qquad (sig\ +)$$

$$\frac{s \in Cannot_s^+(p, E \cup \{s^\perp\})\ p \xrightarrow[E \cup \{s^-\}]{E', k} p' \quad s \notin S(E')}{p/s \xrightarrow[E]{E', k} p'/s} \qquad (sig\ -)$$

3.5 Sémantique dénotationnelle trivaluée

Cette sémantique repose sur l'évaluation des circuits générés à partir du code source **Esterel**. Nous présenterons rapidement les principes de la traduction en circuit puis nous expliquerons les principes utilisés pour l'évaluation de ces circuits dans une logique trivaluée.

3.5.1 Traduction sous forme de circuit

Nous allons présenter de façon rapide et informelle la traduction des programmes **Esterel** en circuit. Lors de cette présentation nous ne tiendrons pas compte des problèmes liés à la schizophrénie ce qui permettra de décrire la traduction de façon structurelle. Nous allons dans un premier temps décrire l'interface commune à chaque instruction puis présenter la traduction de quelques instructions. Une fois de plus on pourra trouver dans [7] l'intégralité des définitions.

3.5.1.1 Description de l'interface du circuit associé à chaque instruction

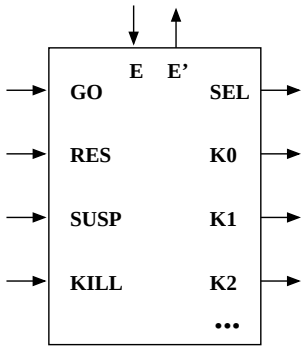


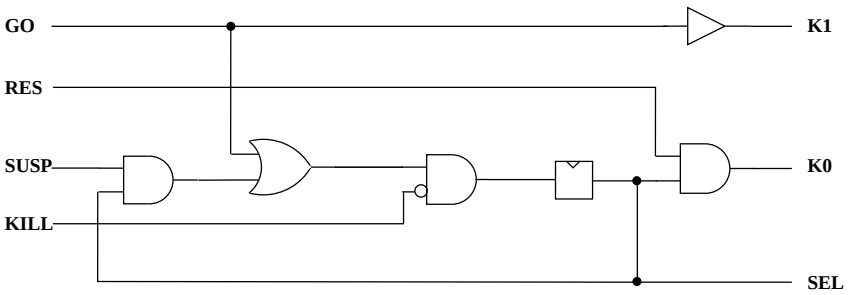
FIG. 3.11 – Interface du circuit associé à une instruction

L'interface des circuits que l'on associe à chaque instruction est présentée dans la Figure 3.11. L'ensemble des fils de gauche ainsi que le fil E sont des entrées tandis que les fils de droite et le fil E' sont des sorties. Voici une description de la signification de chacun de ces fils :

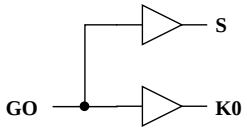
- Les fils d'entrée
  - Le fil **GO** est utilisé pour démarrer l'instruction.
  - Le fil **RES** est utilisé pour recommencer l'exécution de l'instruction.
  - Le fil **SUSP** est utilisé pour suspendre l'exécution de l'instruction.
  - Le fil **KILL** est utilisé pour *tuer* une instruction, ceci arrive lors de l'évaluation de l'instruction *exit*.
  - Les fils représentés par l'entrée **E** correspondent aux signaux visibles lors de l'exécution de l'instruction.
- Les fils d'entrée
  - Le fil **SEL** indique lorsqu'il est activé que l'un des *pause* contenu dans l'instruction est actif.
  - Les fil **K0,K1,...** correspondent aux codes de complétion de l'instruction.
  - Les fils représentés par la sortie $\mathbf{E}'$ correspondent aux signaux émis par l'instruction.

3.5.1.2 Circuit correspondant à quelques instructions

Voici pour commencer le circuit correspondant à l'instruction **pause** :



On trouvera ci dessous le circuit correspondant à l'instruction **emit** S :



3.5.2 Evaluation des circuits Esterel dans une logique trivaluée

Or	$true$	$false$	$\perp$
$true$	$true$	$true$	$true$
$false$	$true$	$false$	$\perp$
$\perp$	$true$	$\perp$	$\perp$

(a) Opérateur **Or** dans la logique trivalué

And	$true$	$false$	$\perp$
$true$	$true$	$false$	$\perp$
$false$	$false$	$false$	$false$
$\perp$	$\perp$	$false$	$\perp$

(b) Opérateur **And** dans la logique trivalué

Not	
$true$	$false$
$false$	$true$
$\perp$	$\perp$

(c) Opérateur **Not** dans la logique trivalué

FIG. 3.12 – Opérateur classique dans la logique trivalué

La traduction des programmes **Esterel** en circuit peut être représentée sous la forme de *netList*, c'est à dire une suite d'équations booléennes. Ce système d'équations booléennes a une solution unique trivaluée qui s'obtient par la combinaison de deux opérations :

a. La substitution des membres droit des équations du système.

b. La simplification des équations du système par application des règles suivantes :

$$\begin{aligned} \neg\neg f &\rightarrow f \\ f \vee 1 &\rightarrow 1 \\ f \wedge 0 &\rightarrow 0 \end{aligned}$$

Il est important de noter que ces opérations sont effectuées de manière traditionnelle par les **BDD**s (**B**inary **D**ecision **D**iagram). Il est important de noter que cette logique ne possède pas le tiers exclus, on a donc si $f = \perp$:

$$\begin{aligned} \neg f \ \vee f &= \perp \\ \neg f \ \wedge f &= \perp \end{aligned}$$

Les règles précédemment énoncées viennent simplement des tables de vérité des trois opérateurs classiques (**Or**,**And**,**Not**) dans la logique trivalués (Figure 3.12).

3.6 Compilation sous forme d'automate et de circuit

Le compilateur **Esterel** permet de générer les programmes sous forme exécutable selon plusieurs schémas d'exécution. Le modèle sous-jacent de l'ensemble de ces schémas est celui des machines de Mealy. Nous présenterons deux des principaux schémas qui sont la représentation sous forme d'automate *explicite* (représentation explicite des états) et la représentation sous forme de circuit (représentation implicite des états). La première de ces représentations a comme principal avantage la rapidité d'exécution tandis que l'avantage de la seconde est que la taille du code généré est polynomiale par rapport à la taille du code source (et dans les faits quasi linéaire).

3.6.1 Compilation sous forme de circuit

La compilation sous forme de circuit correspond à la mise en équation (*netList*) de la partie contrôle du programme. La partie donnée étant *externalisée*, l'activation des actions s'effectuent au moyen *d'output* spécifiques. L'intérêt majeur de cette représentation réside dans le fait que sa taille est linéaire par rapport à la taille du programme source. En effet, puisque les états du programme ne sont pas explicitement représentés on évite l'explosion combinatoire due à une représentation explicite. De plus la représentation sous forme de circuit permet d'effectuer un certain nombre de vérifications (souvent avec un coût exponentiel) au moyen d'outils tels que les **BDDs** . Il faut noter que dans sa version originale (**sc**) les équations ne sont pas triées, plus précisément cela veut dire que la constructivité du programme n'est pas assurée. Une version *améliorée* du format de représentation (**ssc**) correspond à une liste triée d'équations qui assure donc la constructivité du programme, néanmoins ce tri peut dans certains cas (programmes statiquement cycliques) correspondre à une exploration de l'espace d'état avec une réécriture partielle des équations (problème exponentiel).

3.6.2 Compilation sous forme d'automate

Dans ce schéma d'exécution, le programme **Esterel** est compilé sous forme d'automates explicites. Cette explicitation des états s'effectue à partir du format **sc** (représentation des circuits sous forme de *netlist*). Ce processus de compilation correspond à une exploration de l'espace d'états. Ce schéma de compilation possède un certain nombre d'avantages dont :

- Sûreté lors de l'exécution :
  Le fait d'expliciter l'espace des états atteignables lors de la compilation permet d'éviter des erreurs lors de l'exécution (erreurs détectées à la compilation). De plus, pour les mêmes raisons, la vérification de propriétés s'effectue aisément.
- Exécution efficace :
  Ce procédé de compilation consistant à effectuer une exécution partielle du programme lors de la compilation, l'exécution s'en trouve d'autant simplifiée et donc efficace (rapide). Plus précisément, du fait de l'explicitation des états, la compilation pré-effectue un certain nombre de *calculs* tel que :
  - Entrelacement séquentiel des instructions mises en parallèle
  - Décodage de l'état courant

– Statut de présence des signaux
– ...

Néanmoins ce procédé de compilation a un inconvénient majeur correspondant à ces avantages : l'explicitation des états étant exponentielle, la taille de la représentation du programme l'est aussi. C'est une des raisons principales qui fait que ce format n'est utilisé que pour des programmes dont la taille reste *raisonnable*.

3.7 La chaîne de compilation

La compilation des programmes **Esterel** est assurée par un ensemble de processeurs permettant ainsi la décomposition en plusieurs étapes de la chaîne de compilation (Figure 3.13). Voici une brève description des principaux formats générés par les différents processeurs :

- Les formats **ic** et **lc** :

  Ces formats (**ic** et **lc**) conservent la structure hiérarchique et séquentielle des programmes **Esterel**.

  Les principales tâches du processeur **strlic** sont :
  – Vérification de la correction syntaxique du programme.
  – Vérification de la non instantanéité des boucles.
  – Construction de l'ensemble des tables contenant les différents types d'objets (signal, action, variable, ...)[2]

  Le format **lc** correspond à la fusion des différents fichiers **ic** d'un programme **Esterel**.

- Le format **grc**[3] :

  Ce format est décrit plus en détail dans le Chapitre 4,p.45

- Le format **sc** et **ssc** :

  Ce format correspond à la transformation de la partie contrôle d'un programme **Esterel** en réseau de portes logiques (*circuit*). La partie flot de données du programme est activée par des *output* spécifiques du *circuit*. Dans ce format la partie contrôle du programme a totalement été *mise à plat*, de ce fait la structure du programme est entièrement perdue. Le format **ssc** est identique au format **sc** mis a part que les équations booléennes correspondant au *circuit* sont ordonnées.

- Le format **oc** :

  Ce format correspond à la transformation du programme **Esterel** en automate, il correspond à la construction explicite de l'ensemble des états.

[2]La plupart de ces tables se retrouvent dans les formats **sc**,**ssc** et **grc**
[3]Ce format n'est pas fourni dans la distribution standard

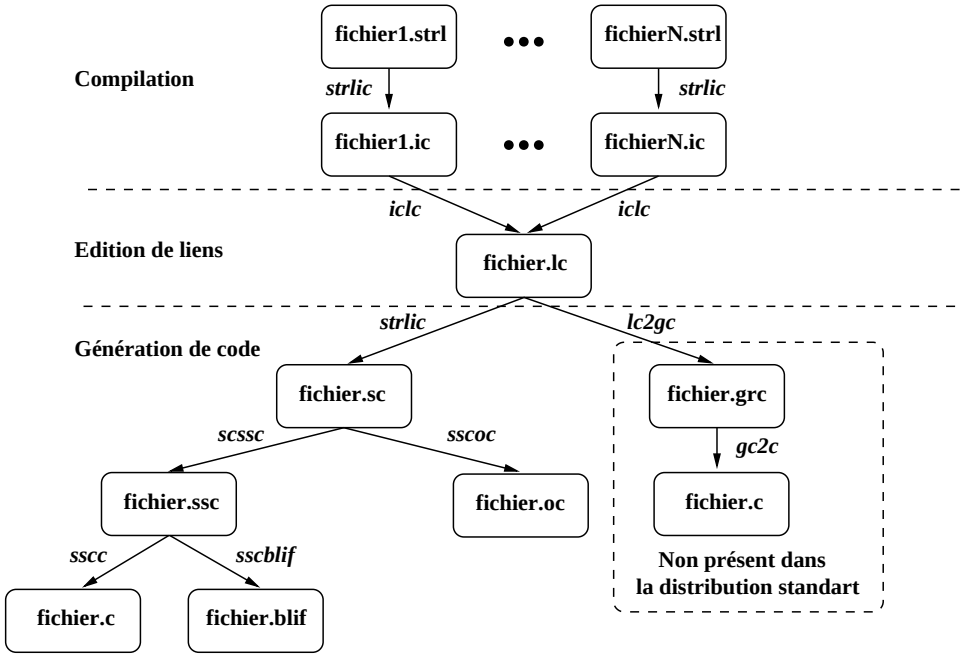


FIG. 3.13 – Représentation partielle de la chaîne de compilation

"Ce sont rarement les réponses qui apportent la vérité, mais l'enchaînement des questions"

Daniel Pennac

Chapitre 4

Le format GRC

Dans ce chapitre nous allons présenter le format **GRC** [32] qui est le format à partir duquel nous effectuons notre traduction vers le logiciel **SynDEx**. Nous conclurons ce chapitre par une analyse des évolutions et changements que nécessite le format **GRC** afin que sa traduction vers le format utilisé par le logiciel **SynDEx** soit possible et efficace.

4.1 Présentation

Le format **GRC** a été développé pour la simulation efficace de programmes **Esterel**. Plus précisément, l'objectif était d'éviter l'évaluation complète des programmes, en n'exécutant que les opérations de données (et si possible de contrôle) réellement actives. Toutefois, la taille des programmes produits ne devait pas *exploser* en fonction du nombre d'états (compilation sous forme d'automate). Dans ce format l'état des programmes **Esterel** est représenté de manière hiérarchique et explicite (*arbre de sélection* ou *Selection Tree*). Plus précisément cela veut dire que l'ensemble des configurations est explicite, l'aspect hiérarchique permettant d'éviter l'explosion combinatoire. Toutefois cette représentation implique un décodage de la totalité de l'état à chaque instant, ce qui a pour conséquence d'être moins efficace que l'exécution des automates explicites. L'aspect comportemental du programme étant quant à lui représenté par un graphe (*Control/Data flowgraph*). Plus précisément, ce graphe représente l'ensemble des comportements possibles au cours d'un instant, il est important de noter que ce graphe ne se modifie pas au cours de son évaluation. L'interprétation de ce graphe s'effectue en deux étapes, la première permet de déterminer l'état courant (consultation du *Selection Tree*), tandis que la seconde permet d'effectuer l'ensemble des opérations ainsi que la mise à jour de l'état pour l'instant suivant (modification du *Selection Tree*). La représentation hiérarchique de l'état permet d'activer seulement les parties *actives* du *Control/Data flowgraph*. La génération de code ne s'effectue qu'à partir du *Control/Data flowgraph*, le *Selection Tree* ne servant qu'à effectuer des optimisations (par annotation de ce dernier) et à conserver la structure initiale du programme. Ce format s'inscrit comme intermédiaire entre le programme **Esterel** source et sa représentation sous forme de circuit. Les propriétés principales de ce format sont :

- Représentation hiérarchique et explicite de l'état
- Résolution des problèmes liés à la réincarnation

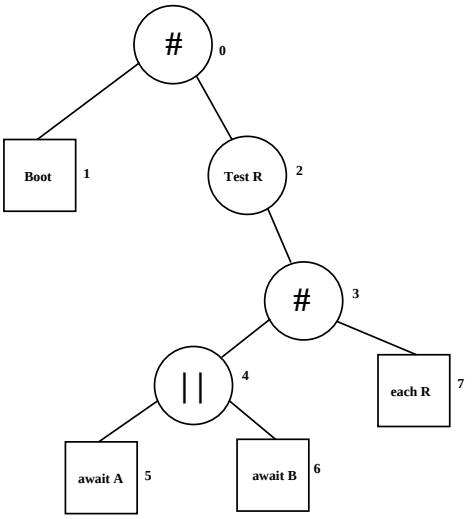
Néanmoins le format **GRC**, comme la majorité des formats intermédiaires, externalise la gestion des données et de ce fait effectue, au moyen de dépendances de contrôle, une sur-approximation

des dépendances de données. De plus le fait de n'activer qu'une partie du programme pose le problème du calcul du statut de présence des signaux (dans un contexte de distribution).

4.2 L'arbre de sélection

4.2.1 Représentation hiérarchique de l'état

La représentation hiérarchique de l'état est construite sous forme d'arbre, appelé *Selection Tree*. Cette structure est une abstraction de l'arbre de syntaxe abstraite du programme **Esterel**. Cet arbre est une composition de nœuds *exclusif/parallèle et référence* dont les feuilles correspondent aux *pause* du programme. La composition séquentielle d'instructions et les tests (*present* et *if then else*) sont représentés par les nœuds exclusifs (#), tandis que la composition parallèle d'instructions est naturellement représentée par les nœuds *parallèles* (||). Les instructions composées d'un seul argument (comme *loop* et *suspend*) sont représentées au moyen de nœuds appelés *référence* et n'ayant qu'un seul fil. Les instructions comportementales instantanées telles que *nothing* et *emit* ne sont pas représentées au sein de l'arbre de sélection (puisque cet arbre sert à encoder l'état).




```
sel:0 nonterm: exclusive: ( sel:1 , sel:2 )
sel:1   boot: pause:
sel:2   ref: ( sel:3 )
sel:3   exclusive: ( sel:4 , sel:11 )
sel:4   parallel: ( sel:5 , sel:8 )
sel:5   pause:
sel:6   pause:
sel:7   pause:
```

FIG. 4.1 – ABRO *Selection Tree* version graphique et textuelle

La Figure 4.1 représente la version graphique et textuelle de l'*arbre de sélection* du programme **ABRO** (Figure 3.1). On notera que chaque ligne de la version textuelle commence par *sel : n*, où *n* est le numéro du nœud. Bien que le programme ne comprenne pas de *pause* explicite, on se rappellera que les traductions en **Esterel** *noyau* des instructions **await** S et **loop** p **each** R comportent un pause.

4.2.2 Construction et interprétation de l'*arbre de sélection*

Construction

Chaque feuille de l'*arbre de sélection* correspond initialement à une instruction **pause** du programme. Cette représentation peut éventuellement se coder au moyen d'un booléen exprimant l'état d'activité du point de contrôle du programme auquel il correspond (*pause*).

Par convention, la valeur *vrai* correspond à l'activité du nœud tandis que la valeur *faux* correspond à son inactivité. Chacun de ces booléens sera lu et modifié lors de l'interprétation du *Control/Data flowgraph*, permettant ainsi de déterminer l'état courant et de mettre à jour l'état de l'instant suivant. La règle déterminant le statut d'activité des nœuds qui ne sont pas des feuilles est :

– Un nœud est actif si et seulement si l'un de ses fils est actif

Il faut noter que l'ensemble des configurations ne correspond pas forcément à un état atteignable du programme. De plus, pour que l'état décrit soit consistant il doit respecter la règle suivante :

– Un nœud de *sélection* (*exclusif*) ne peut avoir au plus qu'un de ses fils actif.

Optimisations

Comme l'*arbre de sélection* ne sert pas à produire de code (il sert uniquement à définir l'encodage de l'état) et afin de conserver la structure du programme auquel il fait référence, les différentes optimisations effectuées par analyse statique sur le format **GRC** ne suppriment pas de nœuds autres que les *pause* (feuilles de l'arbre). Néanmoins la possibilité de changer le type d'un nœud ou d'ajouter des annotations permet d'indiquer certaines propriétés. Dans la version actuelle du format il existe deux annotations (*void* et *nonterm*) en plus des quatre types de nœuds précédemment cités (*exclusif*, *parallèle*, *ref* et *pause*).

– **void**
  Cette annotation indique que le nœud (donc tous ses fils) est instantané. Cette annotation est par exemple ajoutée dans les cas suivants :
  – Si l'une des branches d'un parallèle est instantanée, le nœud lui correspondant est marqué comme *void*.

    emit S ‖**pause**

    Dans cet exemple la branche contenant **emit** S sera marquée *void*.
  – Si un nœud *exclusif* correspond à une séquence d'instructions instantanées il est transformé en *ref* et marqué *void*.

    nothing ; nothing
  – Si un pause ne peut jamais être activé (code mort) il est supprimé, on marque les instructions précédentes *void*.

    exit T **; pause**

– **nonterm**
  Cette annotation indique qu'une branche d'un parallèle ne termine jamais *d'elle même*. Plus précisément cela veut dire que sa durée de vie correspond à celle du parallèle. Cette optimisation est essentielle pour la minimisation du nombre de registres que l'on produit. En effet, le test d'activité et le registre l'encodant à ce niveau sont inutiles car redondants lorsqu'une branche est marquée *nonterm*.

  sustain S ‖ **await I ; exit T**

Interprétation

Le décodage de l'état de l'instant courant commence toujours avec un *arbre de sélection* consistant. Même si cette propriété peut ne plus être vérifiée au cours de l'instant, elle doit toujours l'être entre l'exécution de deux instants. Lors de l'interprétation du *Control/Data flowgraph* l'état courant est consulté et mis à jour au moyen de cinq primitives ; chacune de ces primitives prend un nœud de sélection en argument.

enter : Primitive qui affecte la valeur *true* à un nœud.
exit : Primitive qui affecte la valeur *false* à un nœud.

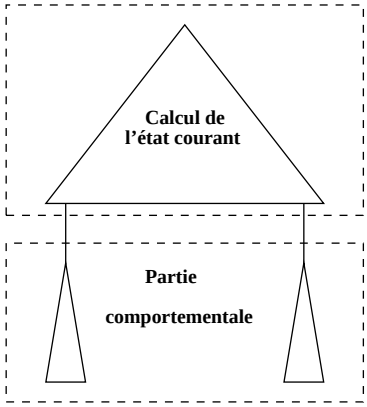


FIG. 4.2 – Décomposition du *Control/Data flowgraph* en deux parties

test : Primitive qui renvoie le statut de son argument.
switch : Primitive qui renvoie le seul fils actif de son argument (l'argument est impérativement un nœud *exclusif* actif).
sync : Primitive qui renvoie *true* si au moins l'un des fils de son argument est encore actif et *false* sinon. (l'argument est impérativement un nœud *parallèle* actif).

4.3 Le Control/Data flowgraph

Le *Control/Data flowgraph* est séparé en deux parties distinctes (Figure 4.2) :
- Partie initiale : Décodage de l'état
- Partie secondaire : Mise à jour de l'état entrelacé avec la partie comportementale (signaux/actions)

Cette séparation est primordiale du fait que la consistance de l'*arbre de sélection* n'est pas assurée durant l'intégralité de l'instant.

Ce graphe permet d'exprimer les dépendances causales (statiques) entre les différents nœuds qui le composent. Notre travail ne s'applique qu'à la classe des programmes dont la représentation **GRC** est acyclique. On pourra donc supposer que le *Control/Data flowgraph* (**GRC**) est un **DAG** (graphe acyclique orienté). Les nœuds composant le *Control/Data flowgraph* (**GRC**) peuvent être de six types différents (**Tick**, **Test**, **Join**, **Call**, **Switch**, **Sync**), ils sont connectés au moyen d'arcs pouvant être de deux types (**Arc de Contrôle** et **Arc de Donnée**). Chacun des nœuds possède une interface spécifique constituée de ports (ayant le même type que les arcs s'y connectant) *entrées/sorties* (décrit plus loin).

Arc de Contrôle
Ces arcs permettent le transfert du contrôle d'un nœud à un ou plusieurs autres nœuds.

Arc de Donnée[1]
Ces arcs correspondent aux émissions de signaux, ils ne se connectent en fait qu'aux **TestNode** qui sont les seuls à posséder des ports de données.

[1] Il faut bien noter que les arcs de donnée ne transportent que le statut de présence des signaux, la gestion des variables étant entièrement externalisée

Il faut noter que ce format (comme la grande majorité des autres formats intermédiaires d'**Esterel**) externalise[2] la gestion des variables. De ce fait le graphe ne contient qu'une sur-approximation (induite par le contrôle) des dépendances entre chaque action (Figure 4.3). Pour obtenir des dépendances plus fines il faut donc construire le flot de donnée correspondant à chaque variable.

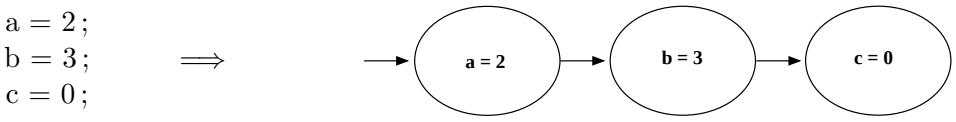


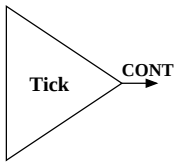
FIG. 4.3 – Dépendances entre actions

4.3.1 Les types de nœuds du *Control/Data flowgraph*

Mis à part le **SyncNode**, tous les nœuds sont activés par le port d'entrée **GO**. Pour ce qui est de la transmission du contrôle cela se fait pour la plupart des nœuds par l'intermédiaire du port de sortie **CONT**. On notera qu'un nœud active au plus un port de sortie à la fois (si un nœud a plusieurs ports de sortie ils sont forcément exclusifs). Par contre un port de sortie peut contenir plusieurs arcs (Ceci permet d'exprimer le parallélisme).

TickNode

Ce nœud est unique dans le *Control/Data flowgraph*, il représente le point d'entrée du graphe pour initier l'exécution du programme à chaque instant.

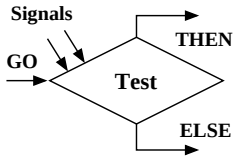


4.4: Tick Node

TestNode

Les nœuds de **Test** représentent les tests sur les signaux ou les données, l'activation du test est conditionnée par le port d'entrée *GO*. Une seule des sorties *then* et *else* est activée après l'évaluation du test. On ajoutera un port d'entrée (de type **Data**) au nœud pour chaque signal intervenant dans l'évaluation du test.

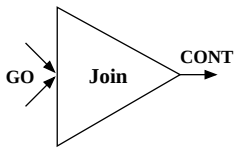
[2]La gestion des variables est assurée par un langage hôte tel que **C**, **C++**,...



4.5: Test Node

JoinNode

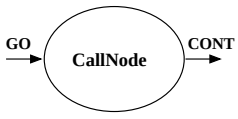
Les nœuds **Join** correspondent à la réunification de branches exclusives (Ex : Les branches d'un test). Ce type de nœud a un seul port d'entrée *GO* ou vient se connecter un ensemble de branches exclusives. Il passe instantanément le contrôle au port de sortie *CONT* lorsque l'une des branches entrantes est active.



4.6: Join Node

CallNode

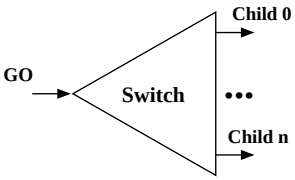
Ce type de nœud a un seul port d'entrée *GO* et un seul port de sortie *CONT*, il correspond à une opération sur des données appelée *action*. Cette *action* peut être une opération sur des données utilisateur mais aussi une primitive modifiant l'état. Cette action est exécutée après que le nœud ait reçu le contrôle et avant que le contrôle soit donné au port de sortie *CONT*. De plus à chaque action est associée une liste (*Access List*), éventuellement vide, correspondant à l'ensemble des actions devant être exécutées avant elle (afin de préserver la causalité). On notera que ce système *d'Access List* correspond à des dépendances ne respectant pas la structure du programme. En effet il n'est utilisé que pour la gestion des signaux valués (l'ensemble des émissions du signal devant précéder la consultation de sa valeur). Pour être plus précis, il assure que l'affectation de la variable associée au signal valué lors de(s) émission(s) soit réalisée avant la(les) lecture(s) de cette variable (Consultation de la valeur du signal).



4.7: Call Node

SwitchNode

Ce nœud associé avec les nœuds exclusifs de l'arbre de sélection(*Selection Tree*) permet de déterminer la branche active parmi un ensemble de branches exclusives. Ce nœud ne possède qu'un seul port d'entrée *GO* et un port de sortie $child_i$ pour chacune des branches.

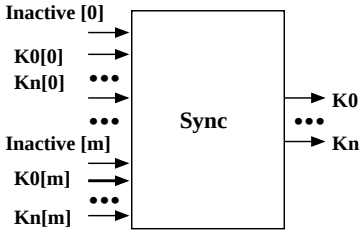


4.8: Switch Node

SyncNode

Les **SyncNode** ont pour fonction de calculer le niveau de terminaison d'un parallèle à partir de ceux de ses composants (chaque branche du parallèle). Les niveaux de terminaisons possibles (et mutuellement exclusifs) sont :
– Terminaison séquentielle normale (K_0 *valeur 0*).
– Non terminaison, le parallèle est toujours actif (K_1 *valeur 1*).
– Sortie avec levée d'exception (K_n *valeur n, n≥2*).
La règle en **Esterel** est que le niveau de terminaison global est le *max* de celui des composants (l'exception de haut niveau l'emporte sur la plus faible, ou sur la terminaison normale ou le blocage).

A chaque niveau de terminaison potentiellement généré par le parallèle correspond une sortie K_n. Lorsque le code de complétion c du parallèle est calculé, le handler correspondant est activé par la sortie K_c du parallèle.



4.9: Sync Node

4.3.2 Un exemple : ABRO

Voici la représentation graphique de l'*arbre de sélection* (Figure 4.10) et du *Control/Data flowgraph* (Figure 4.11) du programme **ABRO**.

4.4 Présentation intuitive de la génération de code séquentiel

Dans cette section, nous allons présenter de façon intuitive la génération de code séquentiel à partir de programmmes **GRC** acycliques [32]. On sait alors qu'il existe un (des) ordonnancement(s) statique(s) permettant une évaluation correcte du programme. Le problème correspond

```
module ABRO :
  input A,B,R;
  output O;
  loop
   [ await A || await B ];
   emit O;
  each R;
end module
```

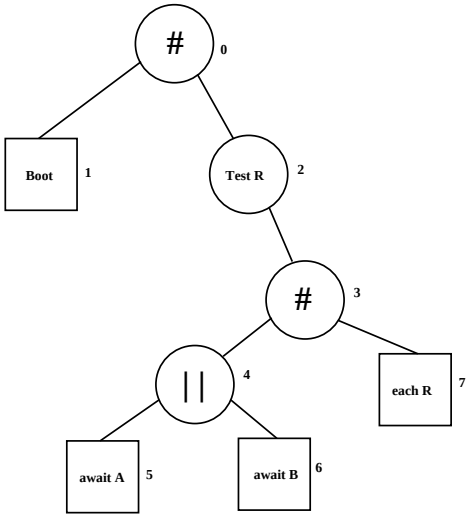


FIG. 4.10 – Le programme **ABRO** et son *arbre de sélection*

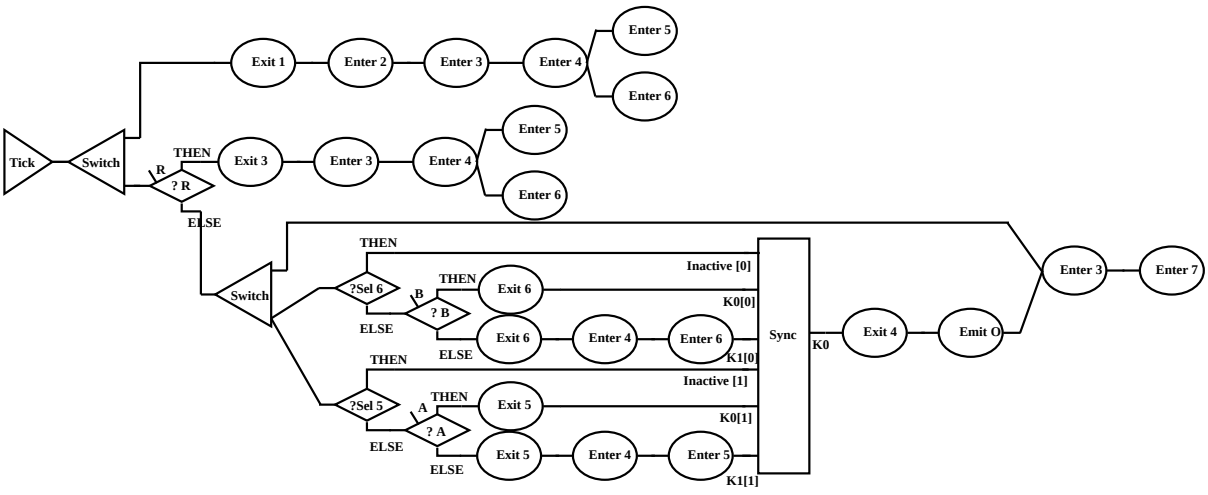


FIG. 4.11 – Le *Control/Data flowgraph* correspondant au programme **ABRO**

donc à trouver un des ces ordonnancements, qui satisfasse le critère global que toute valeur est établie avant d'être utilisée (suivant cet ordre).

En particulier, chaque ordonnancement valide se doit de satisfaire les dépendances induites par le flot de contrôle mais aussi celles induites par les émissions/réceptions de signaux. En particulier une des propriétés que doivent satisfaire les ordonnancements valides est que le statut de présence d'un signal n'est lu que lorsque la totalité des émissions de ce signal a été évaluée. Le problème se ramène donc à réaliser un *entrelacement* (minimisant les changements de contextes) des *threads* de contrôle qui respecte les contraintes imposées par les émissions/réceptions de signaux et les *Access List*.

4.5 Vers un format GRC intégrant le flot des données

Notre objectif étant d'utiliser le format **GRC** comme point de départ pour une transformation des programmes **Esterel** vers un formalisme orienté *dataflow* il nous faut raffiner ou étendre certaines notions. En effet le format **GRC**, malgré ses nombreux avantages, a quelques carences du fait qu'il a initialement été développé pour la génération de code séquentiel. On notera par exemple que certaines informations y restent implicites : dans le cadre de la génération de code séquentiel de programme acyclique et puisque les dépendances de contrôle expriment une sur-approximation des dépendances de données, le calcul d'un ordonnancement valide intégrant ces dépendances de contrôle suffit pour assurer que l'ensemble des affectations d'une variable précède leurs lectures. Ceci n'est plus vrai *par défaut* lorsque l'on désire répartir le code et créer un ordre partiel entre les instructions.

Nous présenterons dans le Chapitre 7,p.75 des modifications du format **GRC** portant sur les points suivant :

- Gestion des émissions/réceptions de signaux
  Dans le format **GRC** la gestion des émissions/réceptions de signaux n'est pas explicitement traitée, le format ne fait que fournir la liste des différentes émissions/réceptions présentes dans le programme (l'algorithme de génération de code séquentiel assure que les émissions précédent les réceptions). Dans une représentation orienté *dataflow* la gestion des *émissions/réceptions* de signaux se traduit par l'ajout d'arcs entre chaque *émission/réception* compatible (émission et réception du même signal). Dans un modèle d'exécution séquentiel ceci peut être réalisé au moyen d'une seule case mémoire booléenne initialement vide (*vrai*) que les émissions viennent *remplir* (mise à *faux*) et que les réceptions viennent consulter, l'ordonnancement s'assurant que les émissions sont bien exécutées avant les réceptions. Notre objectif étant de transformer ce graphe vers un modèle orienté *dataflow*, devant par la suite être distribué, cette méthode d'ordonnancement séquentiel ne peut plus être valide sans communication et synchronisation additionnelle. Il nous faudra donc étudier les différentes possibilités de gestion du calcul de l'absence/présence des signaux.

- Gestion et intégration des dépendances de données
  Les dépendances de données ne sont pas explicitement exprimées au sein du format **GRC**. En effet, le format **GRC** fournit au moyen de dépendances de contrôle une sur-approximation des dépendances de données. Ceci s'explique par le fait que la gestion des variables n'est pas directement assurée au niveau du format mais externalisée afin d'être traitée par un langage *hôte* (C,C++,...). Afin de pouvoir exprimer correctement les dépendances de données entre chaque opération il est donc indispensable d'intégrer cette gestion directement au cœur de la représentation.

Les dépendances de données peuvent avoir deux origines : les premières sont naturellement dûes à l'utilisation des variables par les opérations, tandis que les secondes correspondent à la transcription des informations contenues dans les *Access List* (présentes dans les **Callnode**), et portent sur les valeurs de données transportées par les signaux valués.
La construction des premières revient à exprimer le *datapath* de chaque variable comme de nouvelles dépendances. En essence on désire établir un arc du nœud où la variable est produite vers celui (ceux) où elle est consommée (on notera que la relation n'est pas univoque, il faudra donc étudier finement les possibilités de dépendances effectives).
La construction des secondes suit le même principe qui consiste à ajouter des arcs de dépendances de données entre le nœud effectuant la consultation de la valeur du signal et le(s) nœud(s) effectuant l'émission du signal. La spécificité de ces données est qu'elles peuvent être partagées par des *threads* concurrents, et qu'il faut donc calculer une combinaison consistante des valeurs émises.

- Ordonnancement
  Ce format ayant été développé à l'origine dans le but de générer du code séquentiel, l'expression du parallélisme *potentiel*, induit par les indépendances d'instructions de données (deux instructions portant sur des ensembles de variables disjoints), entre les instructions n'a pas du tout été traitée. Notre objectif final ici est la distribution du code sur des architectures matérielles possédant plus d'une unité de calcul. Il nous faut donc compléter la représentation du programme pour que celle-ci exprime plus finement les dépendances entre les instructions.

Nous présenterons dans le Chapitre 7,p.75 les ajouts et modifications que nous avons réalisés sur le format **GRC** afin de permettre une répartition plus efficace et une meilleure traduction vers le format utilisé par le logiciel **SynDEx**.

"Histoires de producteurs et de consommateurs, du producteur au consommateur, du producteur au consommateur, et des intermédiaires à plus savoir qu'en foutre, toute ton âme s'est usée sur ce chemin sans fin et sur ce va et vient ..."

Bertrand Cantat, L'Europe

Chapitre 5

La méthodologie AAA et le logiciel SynDEx

Dans ce chapitre nous allons présenter la méthodologie **AAA** et le logiciel **SynDEx** ([21] [26]). Notre exposé se concentrera surtout sur le format utilisé par le logiciel **SynDEx** pour la représentation des algorithmes ([22]), puisque notre objectif est la traduction de programme **Esterel** dans ce format.

5.1 Introduction

La méthodologie **AAA** (pour *adéquation algorithme architecture*) ([34]) s'inscrit dans le double objectif du prototypage rapide et de l'implémentation optimisée de systèmes temps réel distribués (Ex : application contrôle/commande, algorithmes de traitement du signal) afin de réduire les durées de réalisation. En effet une unification du cycle de développement (*spécification, simulation, implémentation*) entraîne une réduction du temps de conception. De plus cette méthodologie s'est concrétisée dans la réalisation du logiciel **SynDEx** qui permet la spécification d'algorithme d'application et la définition de réseaux de processeurs. Au départ ce logiciel a été développé pour permettre la spécification des algorithmes et des architectures au moyen d'une interface graphique. Toutefois, la possibilité d'exprimer des spécifications d'algorithme par l'intermédiaire d'un format textuel (sans se servir de l'interface graphique) ouvre la voie à la génération de telles spécifications par des logiciels externes. On notera que les algorithmes spécifiés au sein de **SynDEx** satisfont à l'hypothèse synchrone. Le placement de ces algorithmes correspond donc au placement du code correspondant à un cycle de réaction.

5.2 La méthodologie AAA

La méthodologie **AAA** a été développée afin de faciliter le développement d'applications complexes comme il en existe en traitement du signal. La modélisation des algorithmes (application) et des architectures (matériel) est réalisée au moyen de graphes. L'adéquation se réduit à la résolution d'un problème d'optimisation consistant à respecter certaines contraintes

(temps, consommation, etc ...) lors du placement de l'algorithme sur l'architecture. Les architectures sur lesquelles est effectué le placement étant hétérogènes, une description précise des caractéristiques mises en jeu au niveau du matériel est nécessaire.

5.2.1 Le modèle d'algorithme

Le modèle utilisé pour la représentation des algorithmes dans la méthodologie **AAA** est un *graphe de dépendances de données conditionné* ([36]). Il s'agit d'un *graphe orienté acyclique* dont les sommets sont des opérations partiellement ordonnées (*parallélisme potentiel*) par les dépendances de données qui les relient (arc orienté). L'aspect hiérarchique de la représentation est assuré par le fait que chaque opération peut à son tour être définie par un sous-graphe. Pour finir, le modèle offre la possibilité de définir des dépendances de conditionnement, où le conditionnement se définit comme un ensemble de sous graphes alternatifs dont l'exécution est déterminée par la valeur prise par la dépendance de conditionnement que l'on appellera par la suite condition d'activation. Ceci permet l'introduction d'une certaine forme de contrôle dans le flot de données, mais les variables servant à effectuer les choix ne sont pas clairement établies comme calculées pour du contrôle. Le choix du sous-graphe à exécuter est déterminé par la valeur d'une entrée spécifique. Le modèle d'exécution se décompose en trois phases, lors de la première phase l'environnement fournit l'ensemble des événements par l'intermédiaire des capteurs physiques, la seconde phase consiste à l'évaluation des opérations (permettant de déterminer les valeurs des **arcs de conditionnement**), la dernière phase consiste à informer l'environnement des événements produits au moyen des actionneurs physiques. Il faut noter que les événements sont typés, leurs valeurs sont comprises dans un ensemble bien défini (entier et réel par exemple). Ce modèle d'exécution correspond exactement à celui d'un *système réactif synchrone*.

Il faut noter que l'algorithme correspond en réalité à une répétition infinie d'un motif qui conduit à un graphe infini (Figure 5.1). De même un sommet du graphe peut représenter une répétition finie d'un autre motif. On pourra donc si les ressources matérielles le permettent (présence des capteurs et ressources cpu disponibles), transformer un certain nombre de répétitions temporelles en répétitions spatiales. L'utilisation de valeurs calculées dans des itérations précédentes s'effectue par l'introduction de nœuds spécifiques (**retard**).

5.2.2 Le modèle d'architecture

La description détaillée du modèle d'architecture utilisé par **SynDEx** sort du cadre de ce document, néanmoins nous allons le décrire rapidement, afin de définir quelques notions que nous utiliserons par la suite. On pourra trouver dans [20] et [22] une description détaillée du modèle.

Tout comme les algorithmes, les architectures sont définies au moyen de graphes. Mais, dans le cas des architectures, les nœuds représentent soit des unités de calculs (que l'on appellera par la suite **opérateurs**), soit des média de communication (*bus*,...). Les arcs définissent les interconnexions entre les opérateurs et les média.

5.3 Le logiciel SynDEx

Le logiciel **SynDEx** ([14]) correspond à la mise en œuvre de la méthodologie **AAA**. Il permet la spécification d'algorithmes et d'architectures au moyen d'une interface graphique.

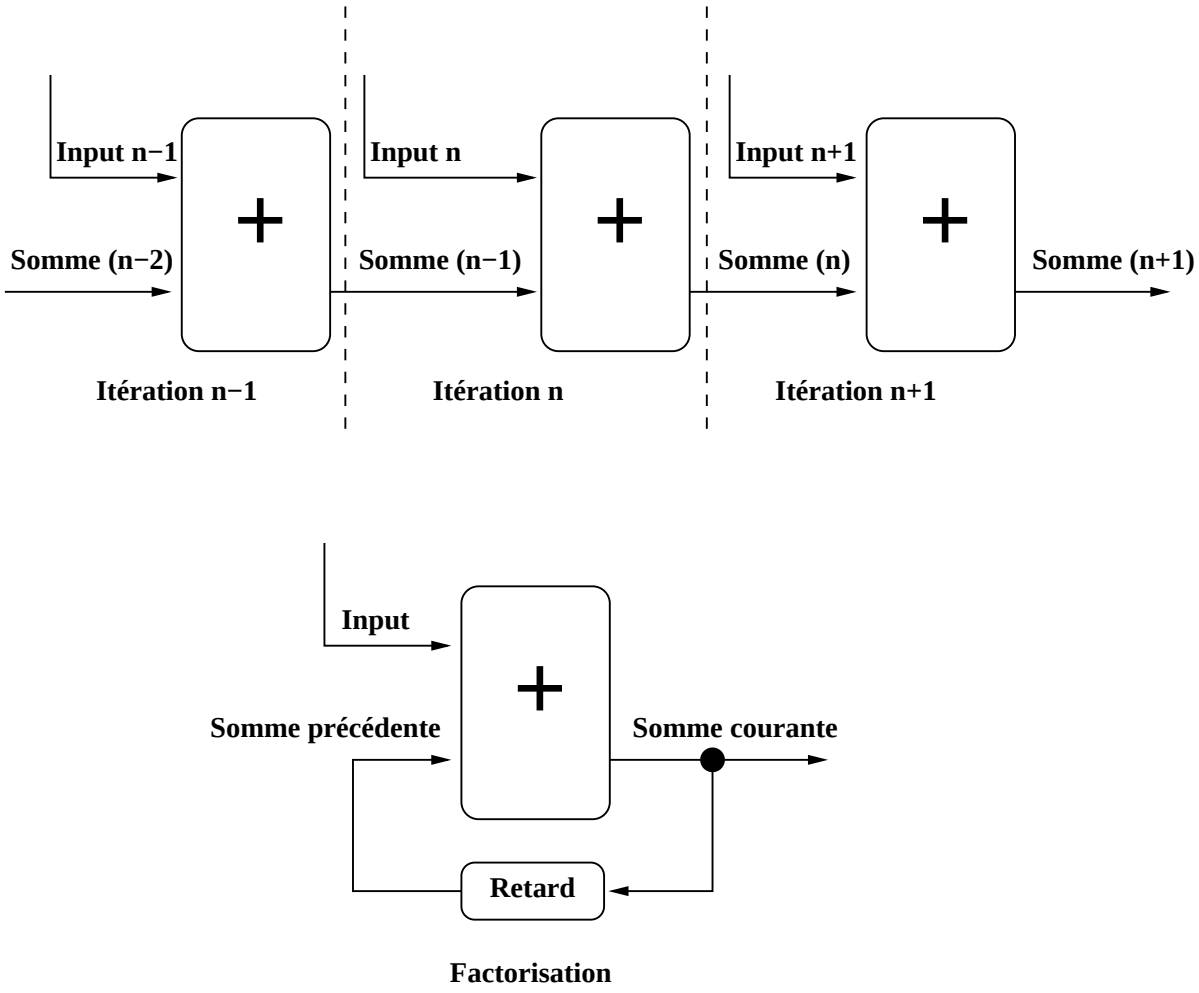


FIG. 5.1 – Le modèle d'algorithme de SynDEx

Après d'éventuelles contraintes de placement (association spécifique d'une opération et d'un processeur) il permet le calcul d'un ordonnancement valide ainsi que sa durée d'exécution. Il offre ensuite la possibilité de générer un ensemble de programmes (un programme par processeur constituant l'architecture), l'exécution de ces programmes sur leurs processeurs associés correspondant à la mise en œuvre de l'algorithme spécifié.

Algorithme

Les algorithmes sont spécifiés comme des graphes orientés acycliques (DAG) répétés infiniment. L'orientation des arcs exprime une relation (d'ordre d'exécution) entre les nœuds, les n-uplets de nœuds sont donc partiellement ordonnés. On défini le premier élément du n-uplets comme le nœud source tandis que les autres sont les nœuds destinations. Le nœud source produit une donnée consommée par le nœud destination.

Les nœuds du graphe correspondant à la définition de l'algorithme principal (*celui de plus haut niveau*) sont des références à des algorithmes, qui sont soit fournis avec le logiciel (correspondant aux opérations de base), soit définis par l'utilisateur. Les algorithmes définis par l'utilisateur sont, comme l'algorithme principal, définis à partir de références à d'autres

algorithmes offrant ainsi un aspect hiérarchique à la définition. De plus les définitions d'algorithme comportent des ports (*entrée/sortie*) correspondant à la consommation et à la production de données des opérations de bases.. D'un point de vue opérationnel les instructions d'un nœud sont démarrées lorsque l'ensemble des données d'entrées est disponible et le nœud produit les données de sortie à la fin de l'exécution de la séquence d'instructions le constituant.

Les références présentes dans les définitions d'algorithme peuvent être de cinq types :

constant
: Les constantes sont des références qui ne possèdent pas de port d'entrée et dont les ports de sortie émettent à chaque itération une valeur constante.

sensor
: Les senseurs sont des entrées venant de l'environnement qui produisent des valeurs à chaque itération. Ils correspondent à des capteurs physiques.

actuator
: Les actionneurs sont des sorties vers l'environnement qui consomment des valeurs à chaque itération. Ils correspondent à des actionneurs physiques.

délai ou **retard**
: Les délais permettent de mémoriser des valeurs pendant une ou plusieurs itérations, cette valeur peut alors être utilisée dans les répétitions suivantes.

fonction
: Les fonctions peuvent être soit définies dans les bibliothèques fournies avec le logiciel soit correspondre à des définitions réalisées par l'utilisateur (*permettant ainsi une structure hiérarchique*). Ces fonctions peuvent être de deux types :
  - **fonction non conditionnée :**
    Ces fonctions sont des opérations classiques, elle produisent un ensemble de *sorties* en fonctions de leurs *entrées*.
  - **fonction conditionnée :**
    Le comportement de ces fonctions est déterminé par la valeur d'une entrée spécifique[1] (**entrée de conditionnement**). On regroupe généralement les fonctions conditionnées par une même entrée au sein d'un même **nœud** (**nœud de conditionnement**). Cette fonctionalité permet d'introduire une certaine forme de contrôle au sein du flot de données.

Il existe deux types d'arcs permettant d'inter-connecter les nœuds :

strong data communication et execution precedence
: Ce type d'arc symbolise un transfert de données et impose aussi un ordre dans l'exécution des deux nœuds auquel l'arc est connecté. La source doit être exécutée avant la destination.

execution precedence
: Ce type d'arc ne correspond à aucun transfert de donnée, il impose uniquement un ordre dans l'exécution de la source et de la destination.

5.3.1 Les fonctions conditionnées

Les fonctions conditionnées (Figure 5.2) peuvent être vues comme un équivalent, en termes de dépendance de données, aux *if . . . then . . . else . . .* ou *switch . . . case . . .* des langages impératifs.

[1] Ces entrées doivent impérativement transporter des valeurs entières.

Toutefois ce modèle impose plusieurs règles lors de la réalisation des fonctions conditionnées liées au modèle flot de données sous-jacent.

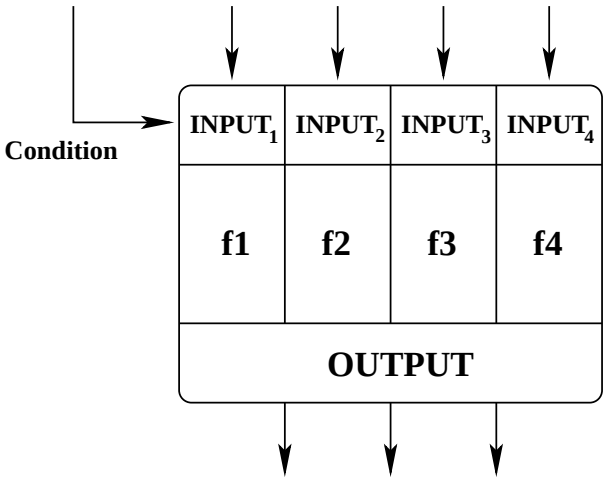


FIG. 5.2 – Les nœuds conditionnés dans SynDEx

1. La totalité des branches d'un conditionnement doit définir (affecter) la totalité de leurs sorties.
2. La totalité des *sorties* est partagée par les branches constituant le conditionnement. Une branche ne peut pas avoir de *sortie* spécifique. En d'autre terme, les sorties de chaque branche sont identiques. Plus précisément cela veut dire que dans un *if then else* les parties *then* et *else* doivent au minimum propager un valeur permettant ainsi de ne pas intérompre le flot de données.
3. L'*entrée de conditionnement* (qui définit la branche à exécuter) doit, lors de l'exécution, obligatoirement correspondre à une branche. Intuitivement, cela veut dire que *l'on ne sait pas ne rien faire.*

La première contrainte empêche toute gestion explicite de l'absence par l'utilisateur. Ainsi il est impossible de ne pas affecter une variable et donc de tester sa présence ou son absence.

La seconde contrainte associée à la première nous oblige à définir l'ensemble des variables à chaque itération même si elles ne sont pas utilisées (*gestion implicite de l'absence*). Pour être plus précis, cela veut dire que le modèle ne permet pas de définir des variables de façon contextuelle. Les variables sont définies pour l'ensemble du programme et doivent être affectées à chaque itération.

Comme nous le verrons dans le chapitre 5.3.2,p.60 les fonctions conditionnées ne constituent qu'un moyen d'exprimer de la hiérarchie conditionnée et ne changent en rien les dépendances de données du graphe d'algorithme. Néanmoins, du fait que ce modèle ne gère pas directement l'absence, il nous empêche d'exploiter de façon optimale les informations d'exclusion dont on dispose.

Ces contraintes correspondent au désir d'assurer la correction des algorithmes. En effet un conditionnement ne sachant pas déterminer localement si l'environnement (le reste de la description **SynDEx**) définit ou non l'ensemble de ses entrées, cette politique garantit que les

valeurs consommées sont effectivement produites. Il est important de noter que cette production/consommation de valeurs est uniforme et maximale ce qui ne permet pas de tenir compte des possibles caractéristiques du programme. Plus précisément, dans le cadre d'une traduction d'un formalisme de *haut niveau* (Ex : **Esterel**) vers le formalisme des algorithmes de **SynDEx** il n'est pas possible d'exploiter au mieux les informations d'exclusion entre les différentes parties du programme (obligation de produire des valeurs non utilisées). Nous présenterons dans le Chapitre 9,p.109 des modifications au modèle permettant une meilleure exploitation de ces informations en introduisant une certaine forme de gestion de l'absence.

5.3.2 Mise à plat et transformation du graphe d'algorithme

Une des tâches du logiciel **SynDEx** avant d'effectuer le placement/ordonnancement est la mise à plat du graphe d'algorithme. Nous allons décrire la mise à plat de l'algorithme pour les fonctions (conditionnées ou non). Nous ne décrirons pas la mise à plat des répétitions car cette construction n'est pas utilisée lors de la traduction. On trouvera toutefois une description exhaustive de la mise à plat des algorithmes dans [28]. Il est primordial de bien comprendre cette opération afin d'optimiser la traduction du format **GRC** vers le format utilisé par le logiciel **SynDEx**. La transformation des fonctions conditionnées est la plus importante puisque nous utiliserons cette construction afin de tirer partie des informations d'exclusion que nous fournit le format **GRC**.

5.3.2.1 Graphes conditionnés et fonctions

On définit les fonctions conditionnées comme étant les fonctions dont l'exécution dépend de la valeur d'une entrée spécifique (*Entrée de conditionnement*). Toutefois on peut étendre la notion de fonctions conditionnées à toutes les fonctions, en considérant que les fonctions originellement non conditionnées le sont par la valeur *vrai*.

Nous allons illustrer la transformation des fonctions conditionnées à partir d'un exemple (Figure 5.3). Après cette présentation intuitive des transformations effectuées nous décrirons précisément le schéma opérationnel de la transformation.

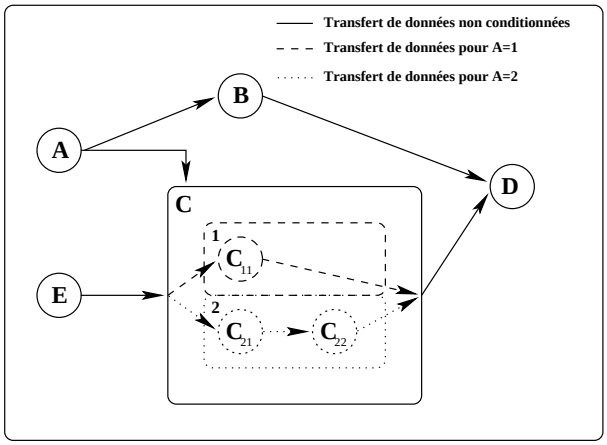


FIG. 5.3 – Graphe conditionné

Le graphe de la Figure 5.3 est constitué de quatre opérations non conditionnées (A,B,D et

E), et d'une opération conditionnée C. L'entrée de conditionnement est la sortie de l'opération A :

- **Si A=1 :** L'exécution de C est équivalente à l'exécution de C_{11}.
- **Si A=2 :** L'exécution de C est équivalente à l'exécution de C_{21} suivie de celle de C_{22}.

5.3.2.2 Le placement/ordonnancement d'un graphe conditionné

Nous allons décrire les spécificités des fonctions conditionnées lors de leur placement. En effet le placement d'une fonction conditionnée implique :

- Deux opérations peuvent être placées sur le même opérateur au même instant si et seulement si elles sont en exclusion mutuelle.
- Tout opérateur
  - ◇ sur lequel a été placé au moins une opération de calcul conditionnée
  - ◇ ou duquel part au moins une communication conditionnée
  - ◇ ou vers lequel arrive au moins une communication conditionnée,

  doit connaître la valeur de la dépendance de conditionnement, pour déterminer s'il doit ou non exécuter les opérations conditionnées.

La Figure 5.4 illustre la transformation du graphe de la Figure 5.3 en appliquant les règles ci-dessus.

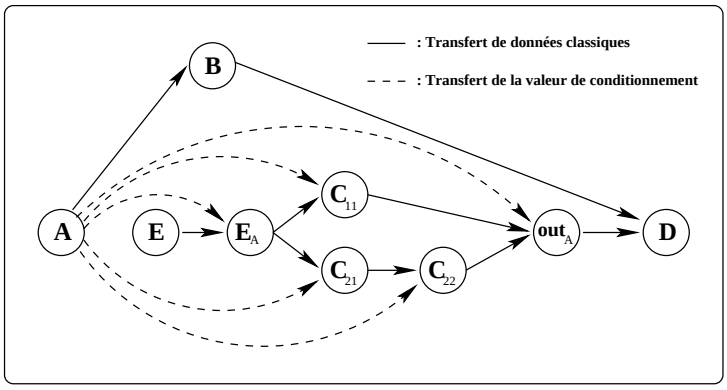


FIG. 5.4 – Transformation du graphe conditionné de la Figure 5.3

On peut noter l'ajout de deux nœuds non présents dans le graphe de départ (E_A et out_A). Le nœud E_A ne correspond en fait à aucune opération, son rôle est uniquement de contraindre la présence de la valeur de conditionnement (A) sur l'opérateur effectuant l'opération E (Transfert de données vers la fonction conditionnée). Le nœud out_A correspond par contre à une opération, son rôle est de déterminer la donnée réellement émise par les fonctions conditionnées (transfert de données vers l'extérieur de la fonction conditionnée).

5.3.2.3 Transformation d'un graphe conditionné

Dans les explications qui suivent l'ensemble des exemples correspond à la Figure 5.3.

Opération de calcul conditionnée

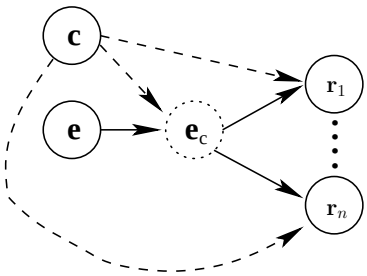
Dans le cas d'une opération de calcul o conditionnée par c (par exemple $o = C_{21}$ et $c = A$) il faut ajouter une dépendance de conditionnement $c \longrightarrow o$ pour contraindre la présence de la valeur de conditionnement sur l'opérateur implantant o.

Communication conditionnée

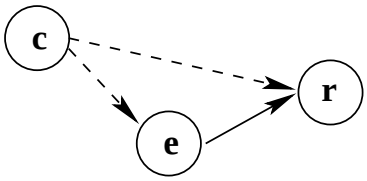
Dans le cas d'une opération de communication conditionnée, l'opérateur s implantant l'opération émettrice e et l'opérateur d implantant l'opération réceptrice r doivent connaître la valeur de l'entrée de conditionnement c.

En effet on peut distinguer trois types de communications conditionnées :

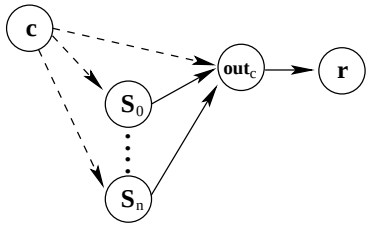
- **Communications entrantes**



  Ce cas correspond à une communication d'une opération e non conditionnée par c vers une opération r conditionnée par c (par exemple $e = E$ et $r_1 = C_{21}$ avec $c = A$). Pour s'assurer de la présence de la valeur de conditionnement sur les opérateurs s et d avant de commencer la communication, on insère un sommet noté e_c (E_A dans l'exemple) ayant pour prédécesseurs e et c et ayant comme successeurs r_i l'ensemble des nœuds utilisant la valeur produite par e. On notera que la valeur de sa sortie est naturellement égale à celle produite par e. Du point de vue du placement l'opération e_c est contrainte sur l'opérateur s.
- **Communications internes**



  Ce cas correspond à une communication de l'opération e vers l'opération r, les deux opérations étant conditionnées par c (par exemple $e = C_{21}$ et $r = C_{22}$ avec $c = A$). Dans ce cas la présence de c sur les opérateurs s et r est déjà assurée par les dépendances de conditionnement insérées entre l'entrée de conditionnement et les opérations de calcul (5.3.2.3,p.61).
- **Communications sortantes**



Ce cas correspond à une communication de l'opération S_0, conditionnée par c, vers l'opération r non conditionnée par c (par exemple $S_0 = C_{11}$ et $r = D$ avec $c = A$). On insère dans ce cas un sommet noté out_c, ayant pour prédécesseurs c (*dépendances de conditionnement*) et toutes les opérations (dépendances de données) conditionnées sans successeur conditionné (par c) . On notera que le sommet out_c est la sortie du sous graphe conditionné, son rôle est de déterminer quelle est la communication réellement active.

Entracte

Nous allons maintenant introduire et motiver les modifications et transformations successives appliquées au format **GRC** afin de l'amener vers un format utilisable par **SynDEx**. De plus ces transformations se doivent de permettre aux algorithmes, utilisés pour effectuer le placement, d'être efficaces. Nous commenterons également les extensions possibles de **SynDEx** en vue de permettre une meilleure expression de certaines caractéristiques du modèle **GRC**.

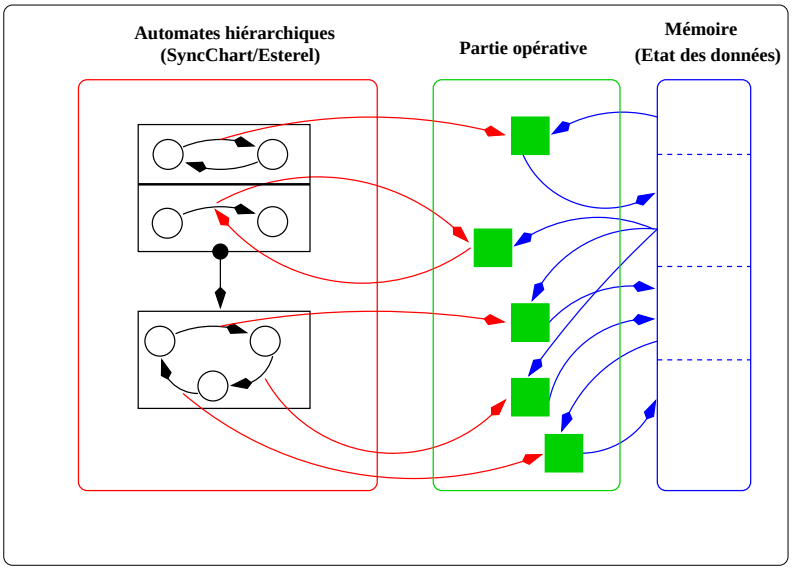
6.1 Aspects GRC

L'intérêt de partir du format **GRC** plutôt que du format source **Esterel** comme point de départ pour notre traduction vient essentiellement de deux points : le nombre de constructions y est limité, et d'une granularité plus fine que celle du langage (même réduit à ses primitives noyau) ; de plus, certains phénomènes perturbateurs comme la *«schizophrénie»* de parties de code ou de variables et signaux, est éliminée en amont de ce format (le précédent format **ic** du compilateur **Esterel** réglait également le premier problème, mais pas le second).

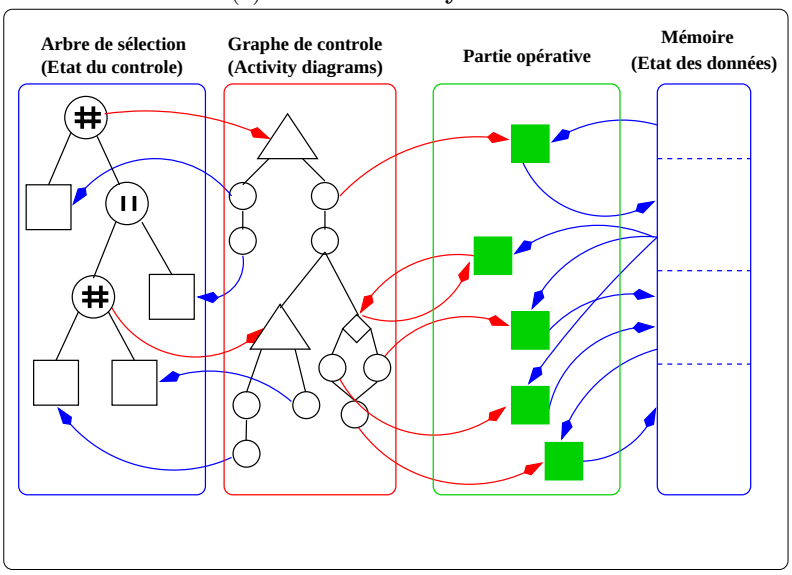
En revanche, un défaut mineur de l'utilisation de ce format en entrée de notre traduction est que la structure hiérarchique des programmes n'y est plus explicite. Mais, de fait, elle transparait tellement derrière des critères de *«graphes bien formés»* qu'on peut la récupérer aisément (même si cela demande des calculs simples parcourant le graphe). Par exemple, il est clair que les opérateurs de parallélisme et de choix sont bien imbriqués. De même, on peut typer les dépendances de contrôle entre celles provoquant des enchaînements en séquence, en parallèle ou en alternative (choix). Un autre défaut du format **GRC** est qu'il n'intègre pas certaines informations, comme les dépendances de signaux ou les dépendances de données induites par ces synchronisations. Le format devra donc être préalablement complété pour les inclure (alors qu'elles sont utilisées implicitement dans l'ordonnancement séquentiel [32])

Le format **GRC** traite essentiellement du flot de contrôle, obtenu à partir d'automates réactifs hiérarchisés décrits en **Esterel** ou dans sa forme graphique **SyncCharts** ([2] [3]). Le traitement des données reste implicite (appel à des opérations externes assimilables à des effets de bord), mais les informations nécessaires sont préservées, hors format **GRC** (Figure 6.1(b)). Elles seront ensuite exploitées lors de la production de code séquentiel, puisque celui-ci doit respecter l'ordonnancement imposé par ces opérations. Il en va de même pour les signaux, dont la communication doit aussi ordonnancer des comportements. Enfin les signaux interagissent avec les dépendances de données puisqu'ils peuvent transmettre des valeurs et que, dans le cas d'émissions multiples, la valeur conjointe doit résulter d'une combinaison précise des émissions effectivement présentes. Un premier travail sera d'expliciter ces relations dans le format, en superposition avec les flots de contrôle *"structurel"*. Il y aura une relation de dépendance de données (*«data flow»*) entre chaque instruction d'affectation *allouant* une valeur de variable, et chaque autre instruction d'affectation *utilisant* éventuellement la valeur calculée au dans la première affectation. L'idée est de remplacer les accès à une mémoire centralisée de données (Figure 6.1) par un flot de donnée (Figure 6.2) parcourant l'ensemble des opérations. On notera que le graphe de donnée ainsi construit est mis sous la forme **SSA** (*Single Static Asssignement*) ce qui procure une localité forte à chaque variable. Ce choix est motivé à la fois par le fait que le traitement et les optimisations que nous allons faire sont plus facilement réalisables lorsque le graphe de données satisfait ce principe, mais aussi du fait que **SynDEx** impose que les algorithmes qu'on lui fournit suivent ce modèle.

La deuxième transformation de notre traduction consistera donc à éliminer au maximum les relations de contrôle *séquentielles*, sachant qu'elles auront été auparavant nécessaires pour instituer les dépendances correctes et complètes entre opérations sur les données : On notera tou-



(a) Sous forme de **SyncChart**



(b) Dans le format **GRC**

FIG. 6.1 – Représentation schématique des programmes **Esterel** et **GRC**

tefois que cette suppression ne sera effective qu'après que la troisième transformation (décrite ci-dessous) ait été effectuée puisque ces dépendances seront de nouveau utilisées pour la réalisation de celle ci.

La troisième transformation viendra réintégrer une structuration explicite hiérarchique du graphe (avec ses dépendances de contrôle et de données juxtaposées mais néanmoins distinctes). Le résultat sera une hiérarchie de blocs d'activation. Cette hiérarchie reste inspirée de la structuration syntaxique au niveau des programmes **Esterel** (via les noeuds **SwitchNode** et **TestNode** du graphe), mais elle est de plus contrainte par les points de synchronisation induits par les dépendances de signaux (spécialement les tests de présence). Chaque bloc d'activation

sera activé par un événement unique, qui sera calculé par la partie du graphe de contrôle de son bloc parent. L'objectif est donc ici de *regrouper* les opérations de calcul et d'établissement du flot de contrôle activées sur les mêmes conditions d'activation. Au niveau des feuilles, les blocs d'activation *«élémentaires»* seront des réseaux d'opérateurs simples, de type bloc *«diagramme»*. L'effet des calculs locaux (au niveau père) des conditions d'activation sera toujours de choisir entre plusieurs blocs d'activation fils alternatifs. Il y a donc ici une *«relocalisation»* des comportements, où une condition d'activation vient lancer les calculs de données, ainsi que des calculs de contrôle qui permettent de décider de l'activation (ou non) de certains des *sous-blocs* du niveau inférieur. L'ambition est que le bloc dans son intégralité ne soit considéré pour le *placement/ordonnancement* que sous sa condition d'activation effective.

On trouvera au sein de la Figure 6.2 l'illustration et l'enchaînement des transformations que nous venons de décrire et qui poursuivent le chemin esquissé de la compilation des programmes **Esterel** en format **GRC** (Figure 6.1).

Il est important de noter que, à l'issu de la troisième transformation, la liste des signaux émis et testés en réception par un bloc d'activation (hors blocs feuilles) n'est **pas** homogène, au sens où des signaux peuvent être émis ou testés uniquement dans une des branches d'une alternative entre deux sous blocs d'activation. De même des variables peuvent être affectées différemment (ou pas du tout) dans ces diverses branches. Néanmoins les propriétés de bonne structuration, héritées par le graphe **GRC** de la structure initiale des programmes **Esterel**, font que les valeurs qui doivent être consommées sont garanties d'être produites (en **Signal/Lustre** cette propriété viendrait d'un calcul d'horloge adéquat, en **Esterel** elle repose principalement sur la rémanence des valeurs de signaux).

Le dernier problème notable dans cette traduction vient de la détermination (explicite et positive) de l'absence de signaux. Dans la compilation séquentielle de programmes **GRC** cette question disparaît du fait de l'ordonnancement total, qui permet de déterminer que, si le statut d'un signal est encore inconnu alors qu'il vient à être testé, c'est que le signal peut être considéré absent puisque toutes les émissions potentielles se situeraient *«en amont»* de cette instruction pour l'ordre total créé. Mais dans une implantation distribuée il faut savoir déterminer, pour chaque bloc *non activé*, quelles sont les possibilités d'émission *«non activée»* de ce fait, jusqu'à savoir atteindre la certitude que le signal ne puisse plus être émis, et donc soit déclaré absent s'il ne l'a jamais été dans l'instant. Synthétiser cette information demandera un certain nombre de communications supplémentaires, et asymptotiquement le fait de communiquer intégralement toutes les informations de non émission pour chaque occurrence syntaxique d'une instruction **emit**. Ceci est la faiblesse traditionnelle des implantations distribuées de systèmes synchrones (compliquée en **Esterel** des multi-émissions simultanées possibles sur un même signal). Il est parfois possible d'envisager certaines optimisations dans le nombre ou la position de ces communications, et nous en introduirons certaines.

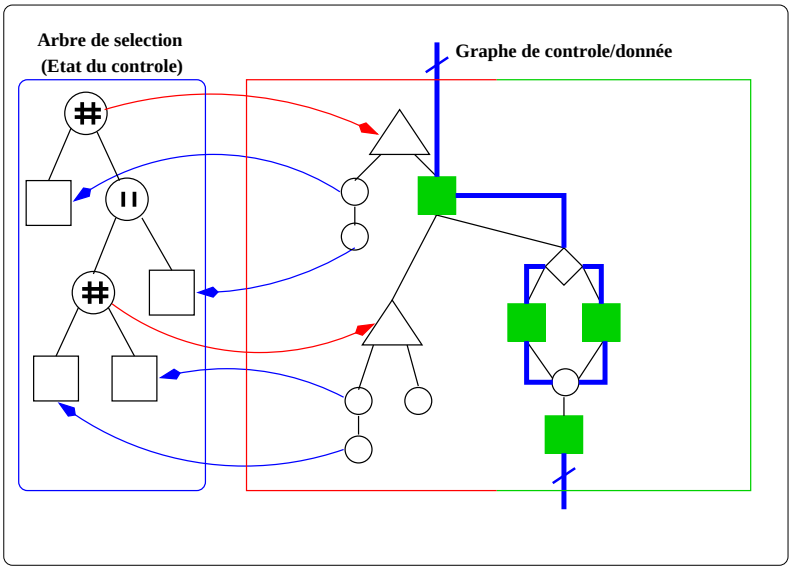
6.2 Aspects SynDEx

Le formalisme d'entrée de **SynDEx** (que nous nommerons ici *format* **SynDEx**) consiste essentiellement en un graphe de dépendances de données, contenant des alternatives entre sous graphes conditionnés par certaines dépendances qui constituent un embryon de contrôle, sans qu'existe néanmoins de distinction entre ces dépendances et les autres (on peut conditionner par n'importe quelle valeur dans un type de données fini). Du fait qu'on n'a aucune garantie sur des hypothèses de *«bonne formation»* des descriptions, le format et sa sémantique réclament que certaines choses soient homogènes entre les différentes branches (sous-blocs) d'un conditionne-

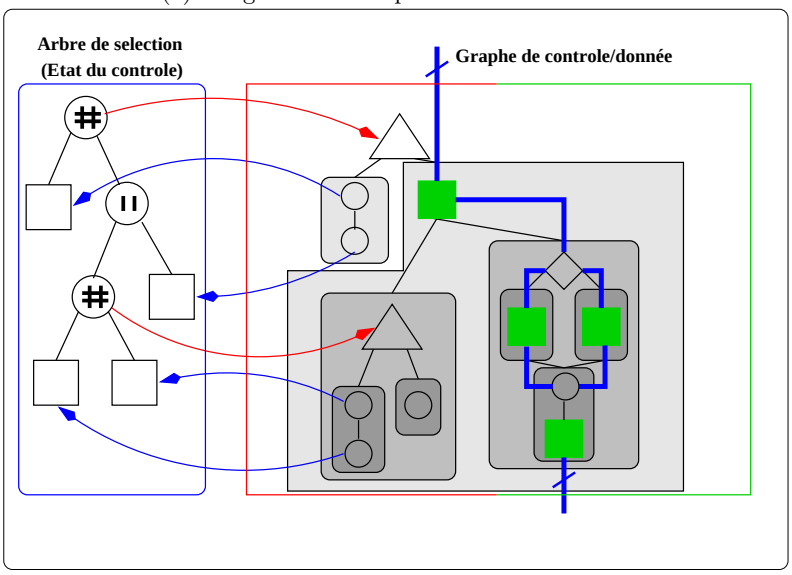
ment, afin de garantir justement que toute valeur utilisée soit définie. Sous certaines hypothèses ces conditions pourraient être levées, autorisant ainsi une meilleure économie de communication, voire une plus grande liberté d'ordonnancement et de placement des opérations sur des ressources architecturales. Nous esquissons dans le chapitre 9 certains arguments allant dans ce sens (qui remonteraient le niveau de modélisation du format **SynDEx** vers la sortie naturelle de nos transformations issues du format **GRC** produit par **Esterel**).

6.3 Conclusion

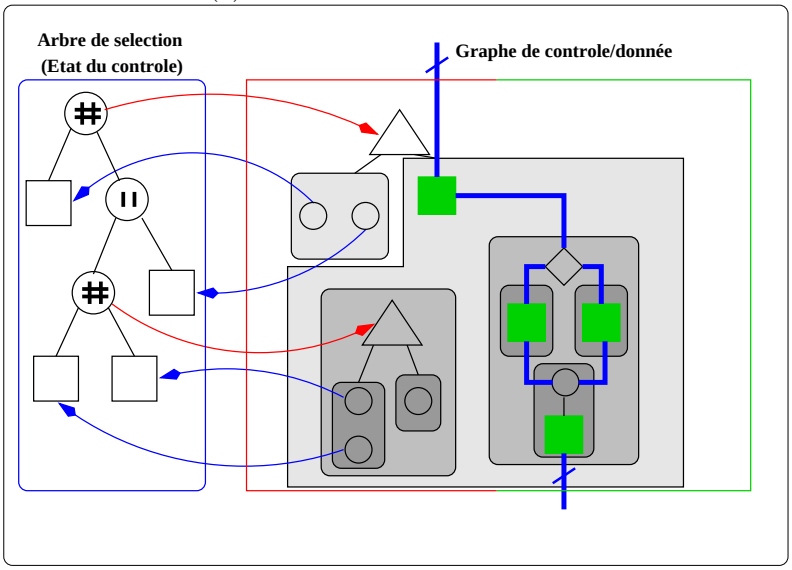
Pour résumer, notre démarche est la suivante : à partir d'un graphe *«purement contrôle»* **GRC**, nous introduisons d'abord des dépendances explicites de données, en utilisant pour se faire les dépendances de contrôle séquentielles. Puis nous réduisons ces dépendances de contrôle séquentielles, et nous localisons dans une hiérarchie les parties du flot de données qui ont des conditions d'activation identiques, avec une partie locale de contrôle qui sert à activer sélectivement les sous blocs de calculs de données alternatifs. Ce modèle peut être vu (à quelques détails près) comme un graphe conditionné **SynDEx**, dans lequel les parties calculant les conditions d'activation pilotant les conditionnements sont explicitement représentées par des graphes de contrôle locaux, implémentant la partie correspondante de contrôle hiérarchique du programme **Esterel** initial. La hiérarchie des blocs est d'une granularité plus fine qu'on ne la retrouve dans le langage **Esterel** (par exemple sous la forme des automates hiérarchiques en **SyncCharts**), du fait que chaque test de signal va introduire un nouveau bloc dont le statut de présence de ce signal sera la condition d'activation. Les différences restant à régler avec le format **SynDEx** exact sont le traitement adéquat de l'absence de signal, et la possibilité de ne définir qu'un sous-ensemble des valeurs, du moment que les valeurs utiles en font partie. Ceci est actuellement identifié et résolu, mais d'une manière qui pourrait donner encore lieu à des optimisations futures. Certaines de ces optimisations demanderaient l'extension du format SynDEx (et donc de ses algorithmes de placement/routage/ordonnancement), ce qui pourrait conduire à de nouvelles optimisations d'ordonnancement.



(a) Intégration des dépendances de données



(b) Création des blocs d'activation



(c) Suppression des dépendances induites par la séquence

FIG. 6.2 – Transformations successives du format **GRC**

Deuxième partie

Traduction GRC vers SynDEx

"Souvenez vous !...car hélas le destin de l'Homme est d'oublier..."

Merlin l'Enchanteur

Chapitre 7

Transformations effectuées sur le format GRC

Dans ce chapitre, nous allons présenter en plusieurs étapes les modifications que nous avons du réaliser sur le format **GRC**. Ces transformations sont indispensables pour rapprocher ce format de celui utilisé par le logiciel **SynDEx**.

Ainsi, dans un premier temps nous superposerons au graphe de contrôle pur, des dépendances supplémentaires, dues aux communications par signaux (Section 7.1,p.76) et dues aux opérations de données (Section 7.2,p.76). La partie données incluant également le traitement des valeurs de signaux puisque les communications dues aux signaux valués sont hybrides (elles font intervenir à la fois du contrôle et des données). Le résultat obtenu est un graphe mixte comprenant à la fois les dépendances de contrôle et de données, la nature (contrôle/donnée) des dépendances étant conservée.

Dans un second temps (Section 7.3,p.79) nous identifierons, au sein du graphe précédemment construit, des blocs d'opérations sur les données et le contrôle (ou blocs d'activité), tels que les dépendances de données y sont suffisantes pour garantir l'ordonnancement. Les dépendances de contrôle (séquentielles) deviennent donc partiellement redondantes et peuvent parfois être supprimées.

Au final nous obtiendrons un graphe hiérarchique (contrôle et données) avec des blocs d'opérations de données de granularité bien plus grande que la simple opération d'affectation. Ces blocs se retrouvent conditionnés et partiellement ordonnancés par une version simplifié du graphe de contrôle initial, qui génère les conditions d'activation et utilise les événements de tests de données pour réaliser ses branchements.

La simplification du graphe de contrôle sera réalisée après la construction des blocs d'activité, cette construction elle même basée sur le graphe dans lequel les dépendances de données ont été ajoutées.

Il sera important de conserver à l'esprit les quelques caractéristiques importantes du *control/-data flowgraph* du format **GRC** (Section 4.3,p.48) (à la lumière des opérations que nous voudrons ensuite effectuer). En effet ce graphe possède initialement plusieurs propriétés que nous allons exploiter au cours des transformations que nous allons réaliser. Le point central est que le graphe est *bien formé*, plus précisément cela veut dire que sa forme est calquée sur la structure hiérarchique du programme. Cette propriété nous permettra donc d'effectuer la majorité de nos transformations par des parcours récursifs en profondeur du graphe (dont la traduction vers le format utilisé par **SynDEx** au Chapitre 8,p.101).

7.1 *Intégration* des signaux dans le graphe de contrôle

Dans le format **GRC** originel les signaux sont assimilés à des données par opposition au flot de contrôle classique (séquence, parallèle, alternative *if-then-else*). Notre approche tend plutôt à considérer les signaux comme faisant pleinement partie du graphe de contrôle. Les signaux gardent néanmoins certaines propriétés spécifiques.

- ils ne peuvent pas activer un nœud par leur seule présence, le *vrai* contrôle structurel est nécessaire. Par contre les liens de communications dûs aux signaux *traversent* la structure du graphe de façon non structurée (une émission peut causer des réceptions n'importe où dans la *portée* du signal).
- Le calcul de la présence/absence d'un signal nécessite donc un travail spécifique (Section 7.6.1,p.92).

Afin de justifier cette approche nous allons séparer les signaux en deux catégories :

- **Signaux non valués**
  Ces signaux ne sont utilisés que pour effectuer de la synchronisation (**await** *S*) ou pour influencer le *flot* de contrôle (**present** *S* **then**), pour cette raison on peut aisément les assimiler à du contrôle.
- **Signaux Valués**
  Les signaux valués peuvent être vus comme la réunion d'une partie contrôle (le signal pur classique), auquel on associe une valeur, qui sera elle partie intégrante du flot de données. Ces variables nécessiteront un traitement particulier dû au fait que les variables associées aux signaux peuvent être vues comme des variables partagées. En effet la valeur de telles variables est une combinaison définie de l'ensemble des valeurs émises sur ce signal.

Néanmoins pour réaliser l'ajout des dépendances de données au graphe de contrôle une grande partie des algorithmes s'applique sur le graphe de contrôle *initial* (celui de **GRC**). Par la suite nous appellerons le graphe de contrôle initial le *graphe de contrôle restreint*.

7.2 Intégration des dépendances de données

7.2.1 Gestion des dépendances de données classiques

L'ajout des dépendances de données au sein du graphe de contrôle consiste à rajouter des arcs entre les nœuds (opérations) définissant la valeur d'une variable et les nœuds utilisant cette variable. L'algorithme utilisé pour ajouter les dépendances de données au *graphe de contrôle* repose sur le fait que l'accès aux variables ne peux s'effectuer que de façon séquentielle (les seuls accès concurrents autorisés sont les accès en lecture). Seules les variables attachées à des signaux valués ne respectent pas cette règle, un traitement spécifique leur est destiné (paragraphe suivant). Une fois cette hypothèse faite la construction des dépendances de données peut facilement être réalisée par un parcours en profondeur du *graphe de contrôle restreint*. En effet cette opération revient à construire le chemin de donnée (*datapath*) de chaque variable. L'algorithme[1] (Figure 7.1) utilisé est inspiré de ceux présents dans [29] et [1].

Le principe de l'algorithme est de parcourir le graphe de contrôle de façon récursive (lignes 9 et 10).

Pour chaque nœud de type **CallNode** rencontré lors du parcours on :

- Ajoute des dépendances de données en fonction des variables utilisées (lignes 3 et 4).

[1] La correction de cet algorithme repose sur le fait que le *graphe de contrôle* est acyclique.

On définit les éléments suivant :

Op(node)	: Opération effectuée par le **CallNode** node.
Def(op)	: Index de la variable *affectée/définie* dans l'affectation op.
Use(op)	: Ensemble des index de variables utilisées dans l'opération op.
CreateDataEdge(source, destination)	: Crée une dépendance de données entre le nœud source et le nœud destination.
Succ(node)	: Ensemble des successeurs du nœud node.
affectation[i]	: Le nœud ayant réalisé l'affectation de la variable ayant l'index i.

```
1  AddDataDep(node) {
2    si node est un CallNode alors
3      ∀ varIndex ∈ Use(op)
4        CreateDataEdge(affectation[varIndex], node)
5        si Op(node) est une affectation alors
6          affectation[Def(Op(node))] ← node
7        finsi
8    finsi
9    ∀ next ∈ Succ(node)
10     AddDataDep(next)
11 }
```

FIG. 7.1 – Ajout des dépendances de données au *Control/Data flowgraph*

– Si l'opération effectuée par le nœud est une affectation on met à jour le tableau $affectation$ (lignes 5 et 6).

La Figure 7.2 illustre le procédé sur un petit exemple.

7.2.2 Gestion des contraintes imposées par les *Access List*

Comme nous l'avons évoqué dans le paragraphe précédent, il est possible, dans le cas de variables attachées à des signaux, que des accès en *lecture/écriture* soient effectués par des *threads* concurrents. Le format **GRC** permet de gérer ce problème de lecture/écriture concurrentes en associant à chaque opération la liste des opérations devant être exécutées précédemment (*Access List*). Le système des *Access List* permet ainsi de sérialiser de façon correcte (du point de vue de la sémantique) l'ensemble des opérations effectuées sur ces variables *partagées* (on désire assurer que les écritures précèdent les lectures). On notera que de ce fait les dépendances exprimées par les *Access List* peuvent ne pas respecter la structure du programme.

La Figure 7.3 présente un module **Esterel** et la version modifiée et simplifiée du *control/data flowgraph* lui correspondant. Dans cet exemple on ajoute une dépendance de données entre l'instruction **emit** $S(10)$ et l'instruction **emit** $T(?S)$ afin que la valeur de S (notée $?S$) soit bien définie. On notera que l'arc symbolisant la dépendance de données due à l'*Access List* traverse le

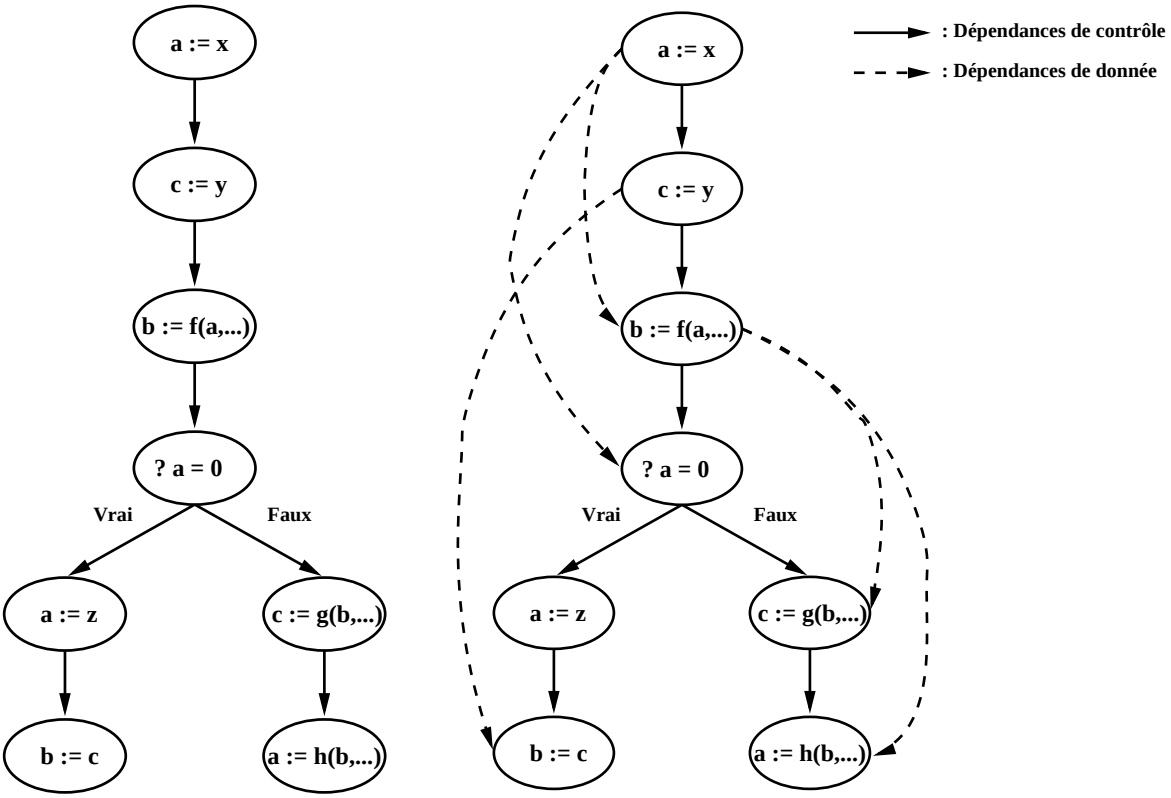
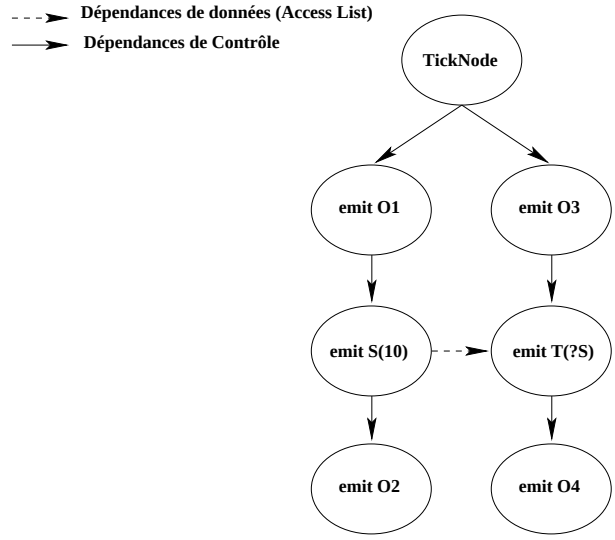


FIG. 7.2 – Ajout des dépendances de données



module M **:**
output *O1*,*O2*,*O3*,*O4* **;**
signal S : integer ,T : integer **in**
[**emit** *O1* **; emit** $S(10)$ **; emit** *O2*]
||
[**emit** *O3* **; emit** $T(?S)$ **; emit** *O4*]
end ;
end module

FIG. 7.3 – Exemple de module imposant des contraintes via l'*Access List*

graphe sans respecter la structure hiérarchique de ce dernier. On observera le même phénomène lors du traitement des signaux (7.6.1,p.92).

Nous allons maintenant décrire l'algorithme servant à ajouter ces dépendances. L'ajout de ces dépendances s'effectue comme précédemment par un parcours en profondeur du graphe de contrôle. On trouvera dans la Figure 7.4 le *pseudo-code* de l'algorithme correspondant à

On définit les éléments suivants :

GetAccessList(node)	: Renvoie *l'AccessList* associée au nœud node
CreateDataEdge(source, destination)	: Crée une dépendance de données entre le nœud source et le nœud destination
Succ(node)	: Ensemble des successeurs du nœud node

```
1 AddAccessListDataDep(node) {
2   si node est un CallNode alors
3     ∀ depNode ∈ GetAccessList(node)
4       CreateDataEdge(depNode, node)
5   finsi
6   ∀ next ∈ succ(node)
7     AddAccessListDataDep(next)
8 }
```

FIG. 7.4 – Ajout des dépendances de données induites par les *Access List* au *Control/Data flowgraph*

l'ajout de ces dépendances. Comme nous l'avons déjà dit cet algorithme effectue un parcours en profondeur du *graphe de contrôle* de façon récursive (lignes 6 et 7). Dans le cas ou le nœud est de type **CallNode** on parcourt l'ensemble des opérations présentes dans son *AccessList* afin d'ajouter les dépendances de données correspondant à chacune des ces opérations (lignes 3 et 4).

On notera que l'on peut réaliser les algorithmes 7.1 et 7.4 lors d'un même parcours du graphe de contrôle.

7.3 Les *blocs d'activité*

Afin de décrire les *blocs d'activité* et leur construction nous commencerons par définir la notion de condition d'activation. On appelle condition d'activation d'une instruction l'ensemble des configurations (état et signaux d'entrée) entraînant son exécution. L'objectif principal de notre traduction étant de n'activer que les instructions réellement exécutées, nous allons utiliser ces conditions d'activation pour réunir des instructions (ayant la même condition d'activation) au sein de blocs que nous appellerons par la suite *blocs d'activité*.

7.3.1 Présentation intuitive

L'objectif est de regrouper les instructions ayant les mêmes conditions d'activation au sein d'un même bloc. Pour cela nous allons séparer le graphe de contrôle en deux parties, la première constituée par la hiérarchie des **SwitchNode** (ces nœuds correspondant principalement au décodage de l'état courant) tandis que la seconde est constituée de regroupements d'instructions ayant les même conditions d'activation au sein de blocs (*bloc d'activité*). On notera que ces *blocs d'activité* seront structurés de façon hiérarchique. Pour être plus précis cette hiérarchie correspondra à celle des **Switchnode** et des **Testnode** du format **GRC** (Figure 7.5).

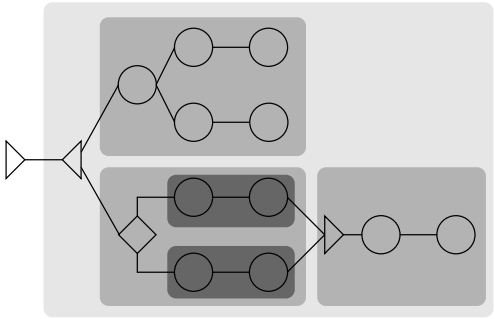


FIG. 7.5 – Structure hiérarchique des *blocs d'activité* calquée sur celle des **SwitchNode** et **TestNode**

7.3.2 Définition formelle des blocs d'activité

7.3.2.1 Propriétés du *graphe de contrôle*

Comme nous l'avons évoqué dans le Chapitre 4,p.45 le *graphe de contrôle* du format **GRC** est bien formé. On est donc assuré que la hiérarchie des **SwitchNode** et des **Testnode** a les propriétés suivantes :

- L'ensemble des chemins partant des arcs sortant d'un nœud de type **SwitchNode** (respectivement un **TestNode**) conflue vers un nœud unique ou ne conflue pas. Par la suite on nommera ce nœud : *GroupNode*[2]. Par exemple le *GroupNode* du graphe issu de la construction **Esterel** : **present** S **then** P **else** Q **end** est le nœud de terminaison du bloc. Du fait de cette définition un *GroupNode* sera donc toujours associé à un **SwitchNode** ou à un **TestNode**.
- Si un **SwitchNode** (respectivement un **TestNode**) n'a pas de nœud *GroupNode* les nœuds **SwitchNode** et **TestNode** qui le précèdent dans le graphe de contrôle n'ont pas de nœud *GroupNode*.
- Si un nœud **SwitchNode** ou **TestNode** appartient à une branche d'un parallèle alors si son nœud *GroupNode* existe il appartient à cette même branche parallèle.
- Si les branches d'un parallèle confluent vers un **SyncNode** alors les **SwitchNode** et **TestNode** présents dans ces branches ont un *GroupNode*.

7.3.2.2 Définition des *leaders*

Nous allons maintenant définir les nœuds constituant le point d'entrée des *blocs d'activité*, nous appellerons ces nœuds *leaders*. On déclare comme *leaders* les nœuds ayant l'une des propriétés suivantes :

- Le nœud est de type **TickNode** (unique source du graphe)
- Le nœud est successeur d'un **Switchnode**, d'un **Testnode** ou **SyncNode**[3].
- Le nœud est en *"tête"* d'une branche parallèle.
- Le nœud a plus d'un prédécesseur (Ex : **SyncNode**).

[2]On notera que *GroupNode* est un nom générique et ne présume en rien sur le type du nœud auquel il s'applique.

[3]Les *blocs d'activité* correspondent dans ce cas aux différents *handler* d'exception

7.3.2.3 Les *blocs d'activité*

Un *bloc d'activité* contient donc un graphe (une sous-partie du *graphe de contrôle*) dont l'unique source est un *leader*. Il nous reste à définir l'ensemble des nœuds appartenant à ce sous-graphe. On peut distinguer plusieurs cas en fonction de la nature du *leader*. Si le *leader* est :

- un nœud de type **TickNode**, le *bloc d'activité* est constitué de l'ensemble du graphe de contrôle.
- un successeur d'un nœud A de type **Switchnode** ou **Testnode** (Figure 7.6), on distingue alors deux cas :
  - Si le nœud A a un *GroupNode* associé, le graphe constituant le *bloc d'activité* correspond à l'ensemble des nœuds atteignables depuis le *leader* avec comme frontière le *GroupNode* associé à A (Figure 7.6(a)).
  - Si le nœud A n'a pas de *GroupNode* associé, le graphe constituant le *bloc d'activité* est constitué de tous les nœuds atteignables depuis le *leader* (Figure 7.6(b)).

(a) (b)

FIG. 7.6 – *Blocs d'activité* issus de **SwithNode**

- un successeur d'un nœud A de type **SyncNode**, le *bloc d'activité* est constitué de l'ensemble des nœuds atteignables depuis le *leader* jusqu'au premier *leader* (Figure 7.7).

(a) (b)

FIG. 7.7 – *Blocs d'activité* issus de **SyncNode**

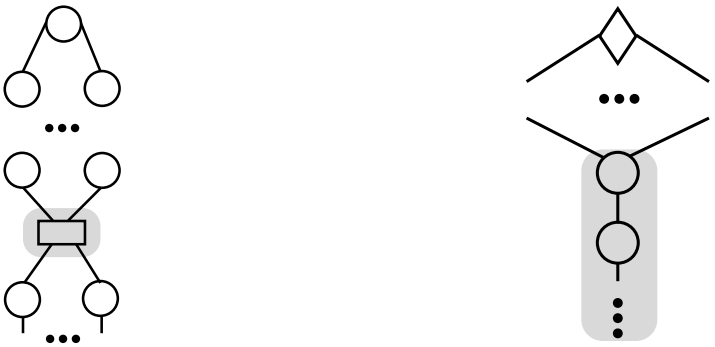
- une *tête* d'une branche parallèle (Figure 7.8), il y a deux cas :
  - Les branches du parallèle ont un nœud **SyncNode** associé, le graphe constituant le *bloc d'activité* est constitué de l'ensemble des nœuds atteignables depuis le *leader* avec

comme frontière le nœud **SyncNode** (Figure 7.8(a)).
– Les branches du parallèle n'ont pas un nœud **SyncNode** associé, le graphe constituant le *bloc d'activité* est constitué de l'ensemble des nœuds atteignables depuis le *leader* (Figure 7.8(b)).

(a) (b)

FIG. 7.8 – *Blocs d'activité* issus de branches parallèles

- un nœud ayant plus d'un prédécesseur (**SyncNode** ou *GroupNode*), le graphe constituant le *bloc d'activité* est constitué de tous les nœuds atteignables depuis le *leader* avec comme frontière le premier *leader* atteint (Figure 7.9). On notera que dans le cas d'un nœud **SyncNode** le *bloc d'activité* se réduit au seul nœud **SyncNode** puisque tous les nœuds successeurs d'un **SyncNode** sont des *leaders* (Figure 7.9(a)).



(a) (b)

FIG. 7.9 – *Blocs d'activité* issus de *GroupNode* ou **SyncNode**

On peut définir les *blocs d'activités* comme des **blocs de base étendus** (*"extended basic block"*), de même notre définition des *leader* correspond à celle trouvée dans [29]. La spécificité de notre construction est de relancer la recherche de *leaders* à l'intérieur des **blocs de base étendus** déjà trouvés. Ceci s'explique par le fait que nous associons la notion de **blocs de base étendus** à celle de *thread d'exécution*. Plus précisément l'activation d'un **bloc de base étendu** implique l'exécution des instructions et des *sous* **blocs de base étendus** qu'il contient. La hiérarchie des **blocs de base étendus** permet ainsi d'affiner au fil de la hiérarchie des blocs la granularité du nombre d'instructions à exécuter.

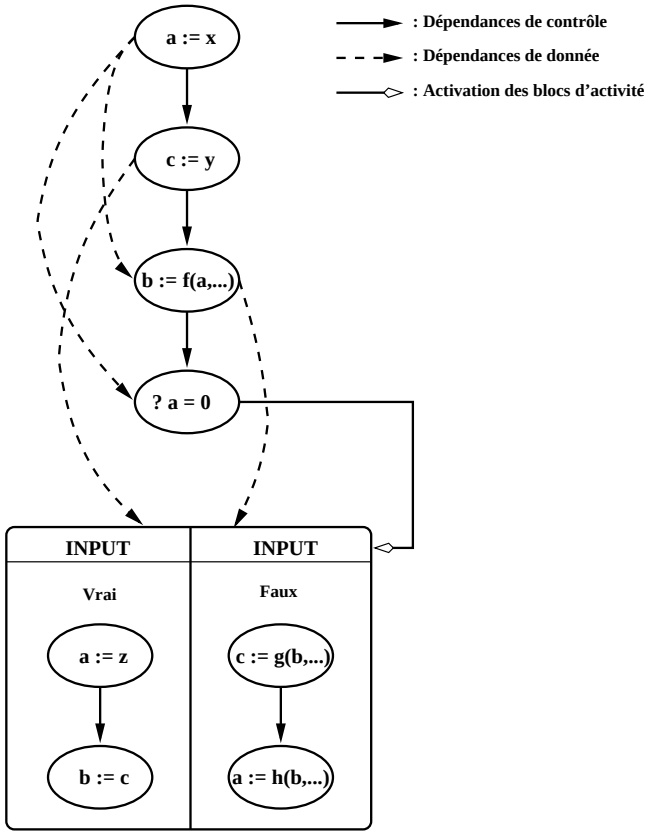


FIG. 7.10 – Création de blocs d'activité

7.4 Les *blocs conditionnés*

Notre objectif étant la traduction vers le format utilisé par **SynDEx** pour représenter les algorithmes, il nous faut adapter notre notion de *blocs d'activité* à celle de conditionnements utilisée dans le logiciel **SynDEx**. Pour cela nous allons regrouper certains *blocs d'activité* au sein d'une même structure que nous appellerons *blocs conditionnés*. Cette structure pourra par la suite aisément se calquer sur la notion de conditionnement présent dans **SynDEx**.

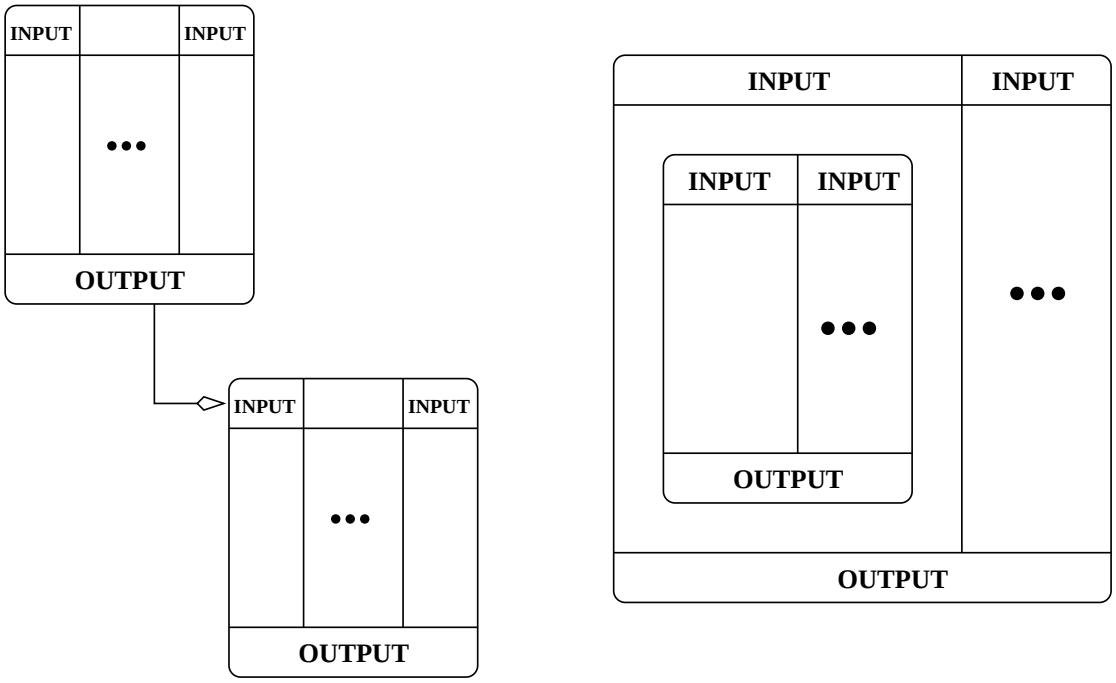
7.4.1 Définition

Nous allons maintenant décrire comment le regroupement des *blocs d'activité* au sein d'un même *bloc conditionné* est réalisé. On discerne deux cas :

- Les *blocs conditionnés* constitués de plusieurs branches. Ces *blocs conditionnés* sont constitués de *blocs d'activité* «frères» issus d'un même **TestNode**, **SwithNode** ou d'un **SyncNode**. Les branches de ces blocs sont exclusives entre elles et dépendent de la même condition (dont elles constituent les différentes conséquences possibles).
- Les *blocs conditionnés* constitués d'une seule branche. Ces *blocs conditionnés* sont constitué des *blocs d'activité* issus des branches des parallèles.

De même l'interface des *blocs conditionnés* se doit de respecter les règles imposées par **SynDEx** en vue de la traduction ultérieure (Section 5.3.1,p.58). Nous allons donc définir l'interface des *blocs conditionnés* : les *entrées* ne sont pas partagées et sont donc spécifiques à chaque *bloc*

d'activité; par contre les *sorties* seront communes à l'ensemble des *blocs d'activité*, l'ensemble des sorties correspond donc à l'union des sorties de chaque branche. On définira une *entrée* supplémentaire pour la condition elle même, qui définira le *bloc d'activité* devant être exécuté (cette entrée transporte une valeur entière correspondant à l'indice du *bloc d'activité* à exécuter).



(a) Première catégorie (b) Seconde catégorie

FIG. 7.11 – Différentes catégories de *blocs d'activité*

7.4.2 Construction des *blocs conditionnés*

D'un point de vue opérationnel, la construction des *blocs conditionnés* s'effectue par un parcours en profondeur du *graphe de contrôle restreint* en recherchant les nœuds *leaders*. La Figure 7.10 illustre les transformations effectuées sur le *control/data flowgraph* déjà modifié (ajout des dépendances de données) du format **GRC**.

Pour entrer un peu plus dans les détails de la construction des *blocs conditionnés* on peut remarquer que la définition des *blocs d'activité* nous permet de séparer les *leaders* en deux catégories :

- **Première catégorie**
  Les *leaders* ayant plus d'un prédécesseur.
- **Seconde catégorie**
  Les *leaders* ayant au plus un prédécesseur.

Ceci nous permet de distinguer deux types de *blocs conditionnés* en fonction de la nature des *leaders* des *blocs d'activité* les constituant. En effet, le type d'un *bloc conditionné* influe dans la manière dont ce même *bloc conditionné* s'intègre dans la hiérarchie (Figures 7.11(a) et 7.11(b)). On peut observer les propriétés suivantes :

- **Première catégorie**
  Ces *blocs conditionnés* sont au même niveau de la hiérarchie que le bloc qui les *"conditionne"* (Figure 7.11(a)).
- **Seconde catégorie**
  Ces *blocs conditionnés* peuvent être vus comme des *sous-blocs conditionnés* (Figure 7.11(b)).

7.4.3 Construction algorithmique des *blocs conditionnés*

Afin de donner une explication détaillée de l'algorithme permettant de construire les *blocs conditionnés* nous allons revenir sur une ou deux caractéristiques du format **GRC**.

7.4.3.1 Fonctions de base du format GRC

On se souvient qu'un nœud ne peut transmettre le contrôle qu'à un seul de ses ports de sortie (*OutPort*). Toutefois, un *OutPort* peut activer plusieurs nœuds, ce qui permet la gestion du parallèle. On définit donc deux fonctions :

- GetOutPort(`node`) : Fonction renvoyant l'ensemble des ports de sortie d'un nœud (peut être vide).
- GetParallelBranch(`port`) : Fonction renvoyant l'ensemble des nœuds activés par le *OutPort* `port` (ne peut pas être vide).

7.4.3.2 Fonctions de base pour la gestion des *blocs d'activité*

On définit un ensemble de fonctions permettant la gestion des *blocs d'activité* et des branches les constituant :

- CreateNewBlock() : Création d'un nouveau *bloc conditionné*.
- CreateNewBranch(`bloc`) : Création d'un nouveau *bloc d'activité* dans le *bloc conditionné* `bloc`. Le *bloc d'activité* créé devient le *bloc d'activité* sélectionné.
- AddNodeToBranch(`bloc`,`node`) : Ajoute le nœud `node` au *bloc d'activité* sélectionné dans le *bloc conditionné* `bloc`.
- FinishBranch(`bloc`) : Termine la création du *bloc d'activité* actuellement sélectionné.
- FinishBlock(`bloc`) : Termine la création du *bloc conditionné* et de ses *blocs d'activité*.

7.4.3.3 Explications détaillées de l'algorithme

Avant de détailler l'algorithme principal permettant la création des *blocs conditionnés*, nous allons présenter une sous-partie de celui-ci dont le rôle est la gestion du parallélisme présent dans le graphe de contrôle. La fonction ConstructParallelBranch, dont l'algorithme se trouve ci-dessous, a pour rôle de créer des *blocs conditionnés* associés aux branches d'un nœud parallèle.

```
1  ConstructParallelBranch(port, courant)
2  port : Le port de sortie que l'on traite.
3  courant : Le bloc d'activité courant.
4  debut
5    si | GetParallelBranch(port) | ≠ 1 alors
6      ∀n ∈ GetParallelBranch(port) {
7        newBlock ← CreateNewBlock()
8        AddNewBranch(newBlock)
9        syncNode ← ConstructCondBloc(n, newBlock)
10       FinishBlock(newBlock)
11     }
12     si syncNode ≠ NULL alors
13     – 1 : Gestion des nœuds de type SyncNode
14     sinon
15       retourner NULL
16     finsi
17   sinon
18    /* L'ensemble est un singleton car il n'y a q'une seule branche */
19    {n} ← GetParallelBranch(port)
20    retourner ConstructCondBloc(n, courant)
21   finsi
22 fin
```

Le rôle de cet algorithme est de créer des *blocs conditionnés* pour chacune des branches d'un parallèle. Ces *blocs conditionnés* ne contiennent qu'un seul *bloc d'activité*. Nous allons maintenant étudier plus précisément cet algorithme :

Les lignes 19 et 20 correspondent au cas où le *port de sortie* n'est pas un parallèle (un seul successeur). Ce cas se réduit à poursuivre la construction du *bloc d'activité* courant.

Etudions maintenant le cas où le *port de sortie* correspond à un parallèle. La première partie de l'algorithme (lignes 6-11) correspond à la création d'un *bloc conditionné* par branche du parallèle (ligne 7), chacun de ces blocs ne contient qu'un seul *bloc d'activité* (ligne 8). Chacun de ces blocs contient une branche du parallèle (lignes 9 et 10).

La seconde partie de l'algorithme détermine si les branches du parallèle on conflué vers un *SyncNode*[4] (lignes 12). Ceci permet de déterminer s'il faut poursuivre la construction des *blocs d'activité* Figure 7.8(a) (algorithme ci-dessous) ou si l'on est arrivé sur des puits du graphe (ligne 15) Figure 7.8(b).

[4]Le format **GRC** nous assure que si ce nœud existe il est unique

Voici la partie de l'algorithme gérant les **SyncNode** :

```
1     – 1 : Gestion des nœuds de type SyncNode
2     si syncNode ≠ NULL alors
3       AddNodeToBranch(courant, syncNode)
4       newBlock ← CreateNewBlock()
5       ∀ handler ∈ GetOutPort(syncNode) {
6         AddNewBranch(newBlock)
7         groupNode ← ConstructParallelBranch(handler, newBlock)
8       }
9       retourner groupNode
10    sinon
11      ...
```

Un nœud de type **SyncNode** correspond au calcul effectué pour la synchronisation des branches du parallèle. On commence donc par insérer le nœud dans le *bloc d'activité* contenant les branches du parallèle (ligne 3). Il faut ensuite créer un *bloc conditionné* contenant un *bloc d'activité* pour chaque *code de complétion* (*handler d'exception*) potentiellement émis par le nœud **SyncNode** (lignes 5-8). Une fois l'ensemble des *handler d'exception* traité on renvoie le premier *groupNode* rencontré (ligne 9). On est assuré que ce nœud, s'il existe est unique (le graphe est bien formé).

Nous allons maintenant décrire l'algorithme principal utilisé pour la création des *blocs conditionnés*. Afin d'en faciliter l'explication nous avons décomposé l'algorithme en plusieurs sous-parties :

```
1  ConstructCondBloc(node, courant)
2  node : Le noeud que l'on traite
3  courant : Le bloc d'activité courant.
4  debut
5    si node est un SwitchNode ou un TestNode
6    -- 1 : Gestion des successeurs des SwitchNode
     et des TestNode
7    sinon si node est un SwitchNode ou un GroupNode
8      retourner node
9    sinon
10   -- 2 : Le nœud courant n'intervient pas dans la création des
11           blocs d'activité.
12   finsi
13 fin
```

Cet algorithme distingue trois catégories de nœuds :

- **SwitchNode** et **TestNode** :
  Les successeurs de ces nœuds sont des *leaders*, on va donc créer un *bloc d'activité* par port de sortie du nœud et les réunir au sein d'un même *bloc conditionné*.

- **SyncNode** et *GroupNode* :
  Les successeurs de ces nœuds sont des *leaders*, et de plus ils correspondent à la terminaison du *bloc conditionné* précédent qui est un bloc de première catégorie (Figure 7.11(a)).
- L'ensemble des autres nœuds :
  Les nœuds n'ayant qu'un seul port de sortie. Ces nœuds vont simplement être intégrés au *bloc d'activité* courant.

Nous allons maintenant détailler la partie de l'algorithme effectuant le traitement des **SwitchNode** et **TestNode**.

```
1   -- 1 : Gestion des successeurs des SwitchNode
    et des TestNode
2   si node est un SwitchNode ou un TestNode
3     newBlock ← CreateNewBlock()
4     ∀port ∈ GetOutPort(node)
5       AddNewBranch(newBlock)
6       nextGroupNode ← ConstructParallelBranch(port, newBlock)
7     FinishBlock(newBlock)
8     si nextGroupNode ≠ NULL alors
9       newBlock ← CreateNewBlock()
10      AddNewBranch(newBlock)
11      retourner ConstructCondBloc(nextGroupNode, newBlock)
12    sinon
13      retourner NULL
14    fin
15  sinon si ...
```

Ces deux types de nœuds entraînent la création d'un *bloc conditionné* (ligne 3). On parcourt l'ensemble des ports de sortie (*OutPort*) du nœud afin de créer les différents *blocs d'activité* (lignes 4-6). On notera l'appel à la fonction *ConstructParallelBranch* permettant ainsi la gestion des branches en parallèle. On détermine ensuite s'il faut poursuivre le parcours du graphe (ligne 8). Dans le cas où l'on a atteint des puits du graphe (Figure 7.8(b)), on stoppe le parcours (ligne 13), alors que si l'on a atteint un *GroupNode*[5] (Figure 7.8(a)) on relance la construction des *blocs conditionnés* à partir de ce nœud (lignes 9-11).

La dernière partie de l'algorithme se charge du traitement de l'ensemble des autres nœuds. Leur caractéristique principale est le fait de n'avoir qu'un seul port de sortie.

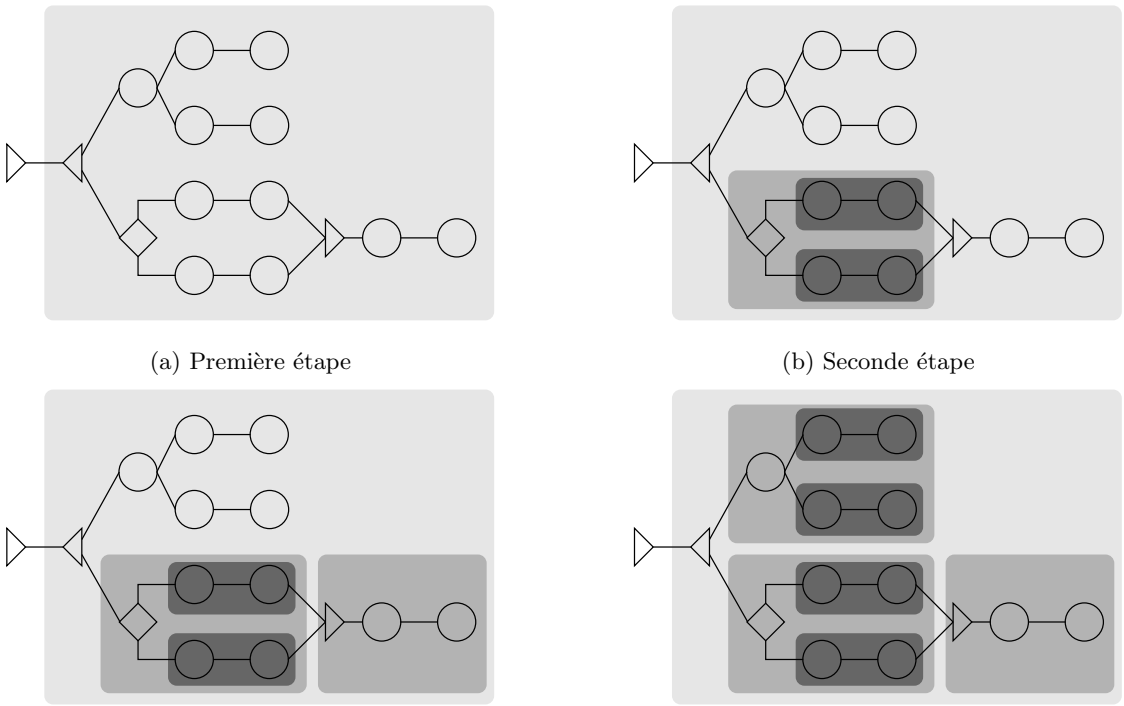
[5]Une fois encore le format **GRC** nous assure que ce nœud est unique

```
1     -- 2 : Le nœud courant n'intervient pas dans la création des
2              blocs d'activité.
3     sinon
4       AddNodeToBranch(courant, node)
5       si GetOutPort(node) ≠ ∅ alors
6         /* L'ensemble est un singleton car il n'y a qu'un seul port de sortie */
7         {port} ← GetOutPort(node)
8         retourner ConstructParallelBranch(port, courant)
9       sinon
10        retourner NULL
11      finsi
12    finsi
13  fin
```

La première action effectuée est l'ajout du nœud courant au *bloc d'activité* courant (ligne 4). La suite de l'algorithme consiste à récupérer le seul port de sortie du nœud (ligne 7) et à poursuivre la construction du *bloc conditionné* courant (ligne 8).

7.4.3.4 Un petit exemple

La Figure 7.12 illustre l'application de l'algorithme sur un exemple.



(a) Première étape

(b) Seconde étape

(c) Troisième étape

(d) Quatrième étape

FIG. 7.12 – Exemple de construction des *blocs d'activité*

Pour des raisons de simplicité nous avons décomposé la création des *blocs d'activité* en quatre étapes.

- Première étape (Figure 7.12(a)) :
  Cette première étape correspond à la Création du *bloc conditionné* ne contenant qu'un seul *bloc d'activité* issu du **TickNode**.
- Seconde étape (Figure 7.12(b)) :
  Dans cette seconde étape on crée un premier *bloc d'activité* issu du **SwitchNode**, ce *bloc d'activité* contenant lui même deux *sous-blocs d'activité* (issus du **TestNode**) réunis au sein d'un même *bloc conditionné*.
- Troisième étape (Figure 7.12(c)) :
  La troisième étape correspond à la construction du *bloc conditionné* constitué d'un seul *bloc d'activité* issu du **JoinNode** associé au **TestNode**.
- Quatrième étape (Figure 7.12(d)) :
  Dans cette dernière étape on construit le *bloc d'activité frère* du *bloc d'activité* créé à la seconde étape (Il seront réunis au sein d'un même *bloc conditionné*). Ce *bloc d'activité* contient deux *blocs conditionnés*, chacun de ces blocs étant constitué d'un seul *bloc d'activité* qui correspond aux branches du parallèle.

7.5 Suppression des dépendances de contrôle *inutiles*

On définit les éléments suivants :

Succ(node)	: Renvoie la liste des successeurs du nœud node.
RemoveArc(node)	: Suppression des arcs de contrôle issus de node.

```
1  SuppressDep(node)
2  node : Le nœud que l'on traite
3  debut
4    si node n'est pas un SwitchNode ou un TestNode alors
5      RemoveArc(node)
6    finsi
7    ∀ nextNode ∈ Succ(node)
8      SuppressDep(nextNode)
9  fin
```

FIG. 7.13 – Algorithme Supprimant les dépendances de contrôle inutiles

Dans la version initiale du format **GRC** une grande partie des dépendances de contrôle correspond à une sur-approximation des dépendances de données. Puisque nous venons d'intégrer les dépendances de données au graphe de contrôle, les dépendances correspondant à une sur-approximation des dépendances de données deviennent donc inutiles. Cette suppression des dépendances de contrôle *inutiles* permet ainsi de faire apparaître un certain niveau de parallélisme potentiel[6]. Les dépendances de contrôle que l'on supprime correspondent à l'opérateur

[6]Le format ayant été développé pour la génération de code séquentiel l'expression du parallélisme n'était pas

de séquence d'**Esterel** et ce sont les plus nombreuses (Figure 7.14). Les autres types de dépendances de contrôle sont :

- **Arc sortant des SwitchNode, TestNode**
  Ces dépendances correspondent à un choix effectué en fonction du résultat de tests qui sont effectués soit sur l'état courant soit sur des *variables* ou des *signaux*. Il ont été définis comme des *conditions d'activité* permettant de lancer l'exécution d'un sous-graphe (Figure 7.10). Ces dépendances sont les seules avec celles exprimant la séquence à être explicites dans le format **GRC**.
- **Émissions et réceptions des signaux**
  Les émissions et les réceptions de signaux imposent un ordre d'évaluation des différentes parties du graphe (7.6.1,p.92). De même que les *précédences d'exécution* imposées par les *AccessList*, ces dépendances ne sont pas explicites dans le format **GRC**.

On peut donc maintenant définir l'ensemble des dépendances (arcs) que l'on désire supprimer. Cette suppression va consister à parcourir le graphe de contrôle restreint et à supprimer l'ensemble des dépendances de contrôle qui ne sont pas issues de **SwitchNode** ou de **TestNode**. On trouvera dans la Figure 7.13 l'algorithme réalisant ces suppressions.

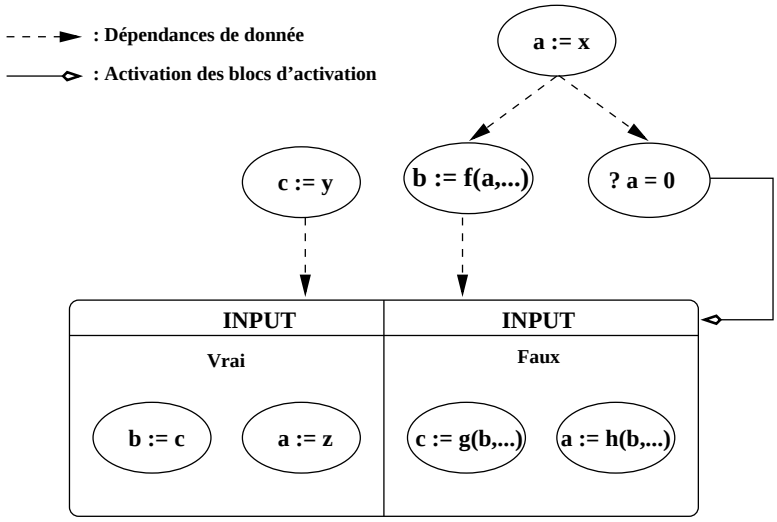


FIG. 7.14 – Suppression des dépendances de contrôle dus à la séquence

7.6 Ordonnancement

Le format **GRC** ne fixe pas un ordonnancement mais fournit un ensemble d'informations permettant d'en définir un valide par rapport à la sémantique d'exécution d'**Esterel**. En fait le format définit un ordre partiel correspondant à l'ensemble des ordonnancements valides. En plus du flot de contrôle (arc de choix des *blocs d'activation*) et des dépendances de données, la validité d'un ordonnancement est contrainte par les informations de précédence d'exécution et les *émissions/réceptions* de signaux.

primordiale.

7.6.1 Le statut de présence des signaux

La gestion du statut de présence des signaux est un point central d'**Esterel** qui permet à la fois les réactions **instantanées** à l'absence et les *lieux* multiples d'émissions. La cohérence de ce calcul impose un ordre partiel entre certaines instructions (*émissions* et *réceptions*). Le format **GRC** ne spécifie pas de méthode pour le calcul du statut de présence des signaux, mais uniquement des points d'entrée dans le graphe correspondant aux *réceptions/émissions* d'un signal. Ceci s'explique du fait que ce problème, dans un cadre séquentiel, se résout en trouvant un ordonnancement[7] total statique, dans lequel les écritures sont exécutées avant les lectures, ce qui permet de n'utiliser qu'une seule *case mémoire* pour la représentation du statut du signal (absent sauf s'il est émis). Dans le cadre d'une approche *dataflow*, avec comme objectif final la distribution, les dépendances doivent par contre être explicites pour assurer la cohérence du calcul.

On peut distinguer deux techniques de représentation des dépendances entre les *émissions/réceptions* de signaux, la première consistant à effectuer une *diffusion* (*broadcast*) de chaque émission vers chaque *réception* (Figure 7.15), la seconde consiste à *centraliser* le calcul du statut de présence de chaque signal pour le redistribuer à chaque réception (Figure 7.16)[8]. Toutefois la difficulté principale dans notre cas est que toutes les parties du programme ne sont pas exécutées à chaque instant, ce qui pose le problème de savoir quelle occurrence d'**emit** on doit attendre avant de calculer le statut de présence du signal.

Dans tous les cas, la méthode utilisée pour permettre la gestion de l'absence est fortement dépendante du modèle d'exécution sous-jacent. Nous présenterons dans 7.6.1.2,p.93 les principaux modèles d'exécution possible et la méthode de gestion de l'absence leur correspondant.

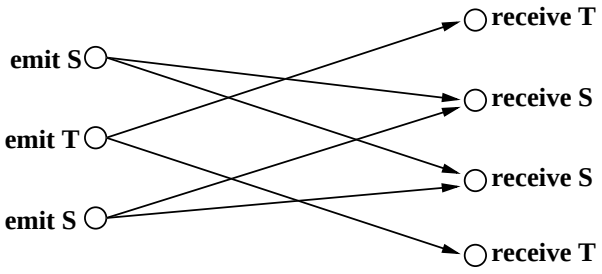


FIG. 7.15 – Implémentation des émissions de signaux par *broadcast*

7.6.1.1 Organisation des collecteurs de signaux

Pour la réalisation de notre traduction nous avons choisi de calculer le statut de présence des signaux au moyen de collecteurs qui déterminent la valeur de présence du signal à partir des émissions potentielles. En effet notre objectif étant la distribution, cette méthode de calcul permet de regrouper le statut de présence des signaux au sein d'une même *donnée*, ce qui permet de réduire les échanges d'informations entre les différentes parties du programme.

L'organisation des collecteurs de signaux s'effectue en respectant l'aspect hiérarchique des nœuds *parallèle* du *Control/Data flowgraph* du format **GRC**. En effet c'est le seul cas où un

[7] Ceci n'est pas vrai pour les programmes corrects cycliques, mais actuellement tous les compilateurs récrivent ces programmes en des formes acycliques équivalentes

[8] Dans les figures 7.15 et 7.16 on appelle *receive* les opérations testant le statut d'un signal tel que **await**, **present**, **suspend**, ...

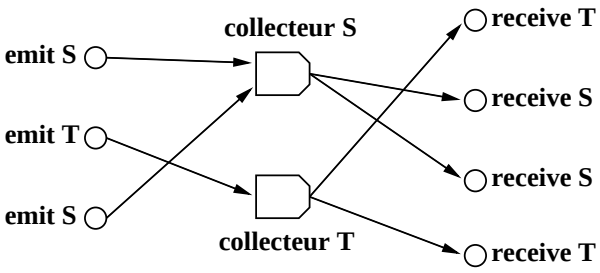


FIG. 7.16 – Implémentation des émissions de signaux au moyen de *collecteurs*

signal peut être émis de plusieurs sources différentes au cours d'un même instant. La Figure 7.17 illustre la construction hiérarchique dans la mise en place des collecteurs.

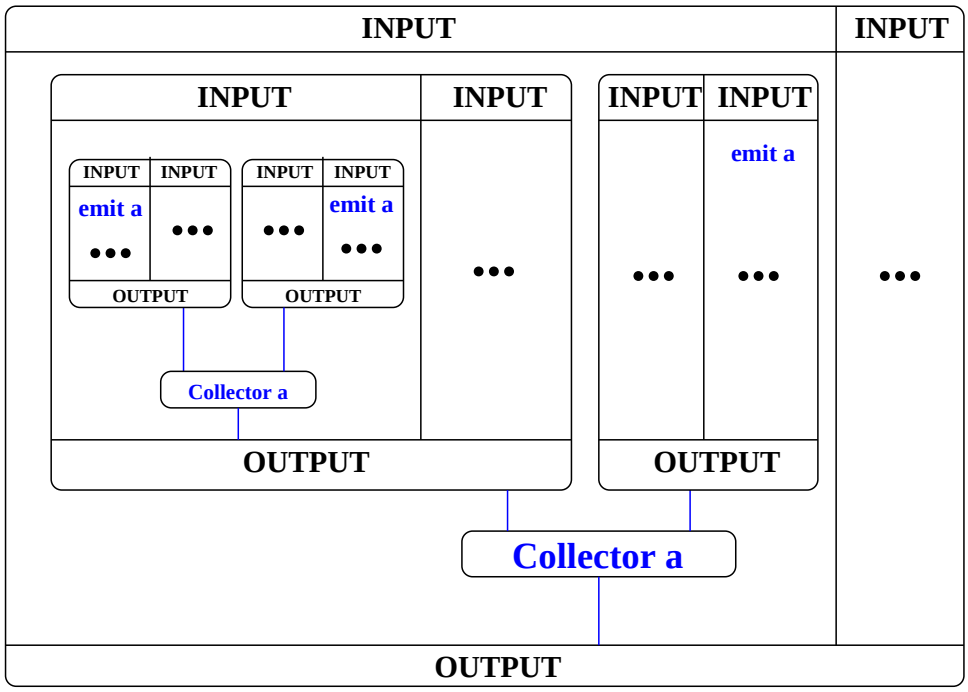


FIG. 7.17 – Organisation hiérarchique des collecteurs de signaux

Afin de minimiser les communications on ne *remonte* pas le statut de présence des signaux jusqu'au plus haut niveau mais uniquement jusqu'au nœud *parallèle* dont les branches contiennent l'ensemble des utilisations du signal (émission/réception). Ceci correspond fréquemment à la déclaration de signal local. On notera que cette optimisation se fonde néanmoins sur l'utilisation réelle des signaux et non pas sur la syntaxe du langage.

7.6.1.2 Modèles d'exécutions et collecteurs de signaux

On notera que l'implémentation des collecteurs peut être réalisée de plusieurs façons différentes en fonction du modèle d'exécution, c'est-à-dire des possibilités offertes par la plate-forme cible (*logicielle* ou *matérielle*). Nous allons faire une présentation des principaux modèles d'exécution

et de l'implémentation des collecteurs leurs correspondant.

- **Activation de la totalité du programme**
  Cette méthode est utilisée dans la traduction en circuit où l'on peut supposer que toutes les instructions de contrôle sont activées simultanément (propagation électrique). Le calcul du statut de *présence/absence* d'un signal est basé sur l'activation de l'ensemble des parties du programme(propagation des *zéro*). De façon plus explicite cela veut dire que les émissions ne devant pas être activées sont parcourues tout de même et émettent la valeur 0, tandis que les émissions réellement actives émettent des 1. Dans cette méthode le statut de présence est vu comme une donnée (0 *ou* 1), cette valeur étant calculée à chaque réaction. Le collecteur ne peut déterminer l'absence du signal que lorsque toutes les émissions ont émis la valeur 0. Pour des raisons d'uniformité dans la représentation de la causalité, l'émission de la présence d'un signal n'est effectuée que lorsque toutes les émissions ont défini leur valeur alors que la sémantique permettrait de déclarer le signal présent dès la réception d'un 1 (émission active).

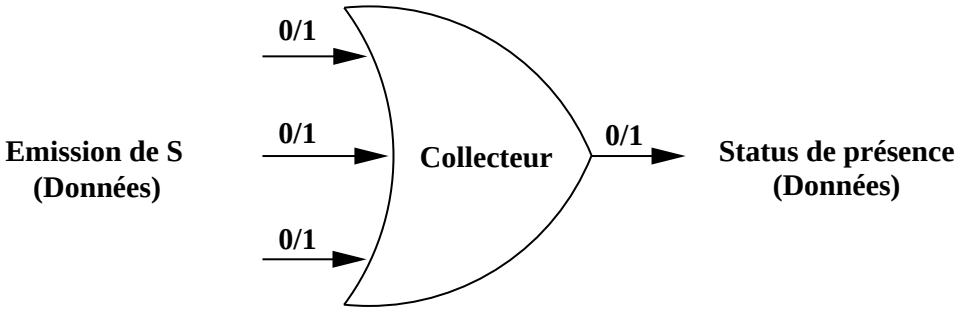
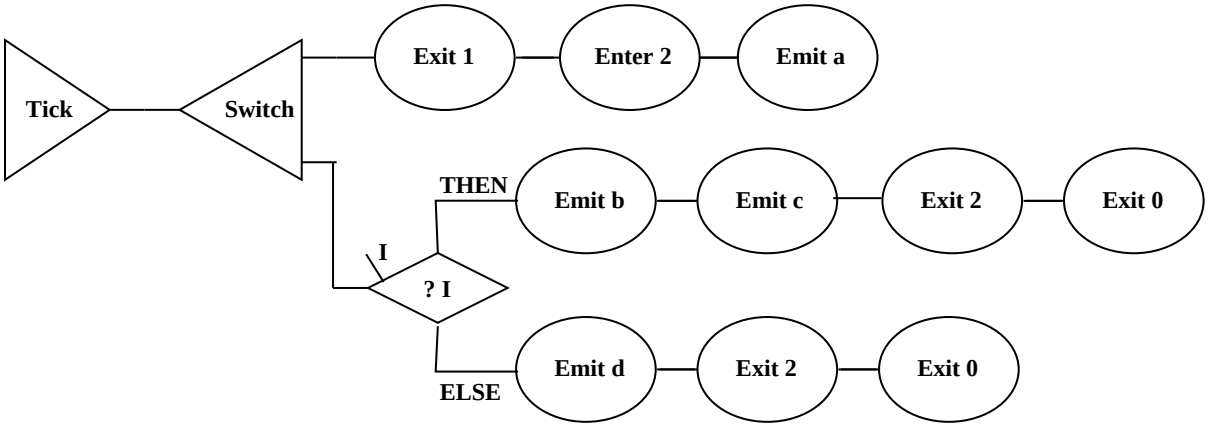


FIG. 7.18 – Collecteur sans gestion de l'absence

  L'utilisation de ce modèle dans le cadre de notre traduction nous imposerait d'activer la totalité du programme à chaque instant ce qui ne nous permettrait pas d'exploiter les informations d'exclusion fournies par le langage **Esterel**. Ceci est d'autant plus pénalisant que notre objectif est la distribution, cette activation totale engendrerait donc un sur-coût dans la gestion des communications.
- **Notification explicite de l'absence**
  Dans ce modèle le programme n'est plus activé dans sa totalité à chaque réaction, au lieu de cela les parties actives du programme notifient explicitement l'absence des signaux quelles n'émettent pas. La figure 7.19 permet d'illustrer le propos. Les instructions *emit* x correspondent naturellement à l'émission du signal x tandis que les instructions *emit* $\overline{x}$ correspondent à la notification explicite de l'absence de x.
  Le collecteur servant à déterminer le statut de présence du signal fonctionne sur le même principe que pour le modèle précèdent (Figure 7.18), la différence entre les modèles étant constituée par la méthode utilisée pour la notification de l'absence mais pas dans son traitement. Il faut noter que le principal inconvénient de ce modèle est le fait qu'il a pour conséquence une augmentation du nombre d'instructions total bien que le nombre d'instructions réellement exécutées soit inférieur au modèle précédent. Toutefois on montrera dans 7.6.1.3,p.96 que le nombre d'instructions ajoutées est dans la majorité des cas quasi linéaire.
- **Gestion réelle de l'absence**
  Dans ce modèle la gestion de l'absence n'est plus réalisée par une notification explicite mais




```
module EXP:
input I;
output a,b,c,d;
  emit a;
  pause;
  present I  then
    emit b;
    emit c;
  else
    emit d;
  end present
end module
```

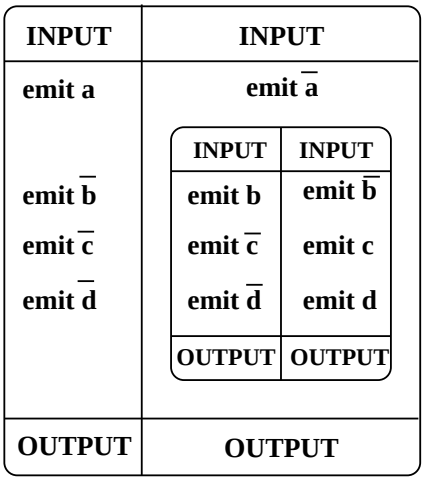


FIG. 7.19 – Notification explicite de l'absence des signaux

gérée directement au niveau du modèle d'exécution. En effet les signaux, dans ce modèle, ne sont plus assimilés à des données mais à du contrôle. Les émissions de signaux sont considérées comme des traces d'activation de points de contrôle. Le rôle des collecteurs dans ce modèle est d'émettre 1 si l'un des points de contrôle est atteint et 0 sinon. Ceci impose d'ajouter un point de synchronisation indiquant que l'ensemble des émissions potentielles a été *dépassé*. On peut donc voir ces collecteurs de signaux comme la réunion de deux opérateurs. L'un émettant des 1 (le signal est présent) tandis que l'autre émet des 0 (le signal est absent). L'activation du premier opérateur (le signal est présent) est conditionnée par une horloge correspondant à l'union des horloges régissant l'activation des émissions de signaux. L'activation du second opérateur est évidemment conditionnée par une horloge complémentaire.

7.6.1.3 Réalisation des collecteurs de signaux

Dans le cadre de la traduction du format **GRC** vers le format des algorithmes de **SynDEx** le choix du modèle de collecteur par *notification explicite de l'absence* est le choix le moins pessimisant. En effet le logiciel **SynDEx** ne permettant pas une gestion réelle de l'absence, le

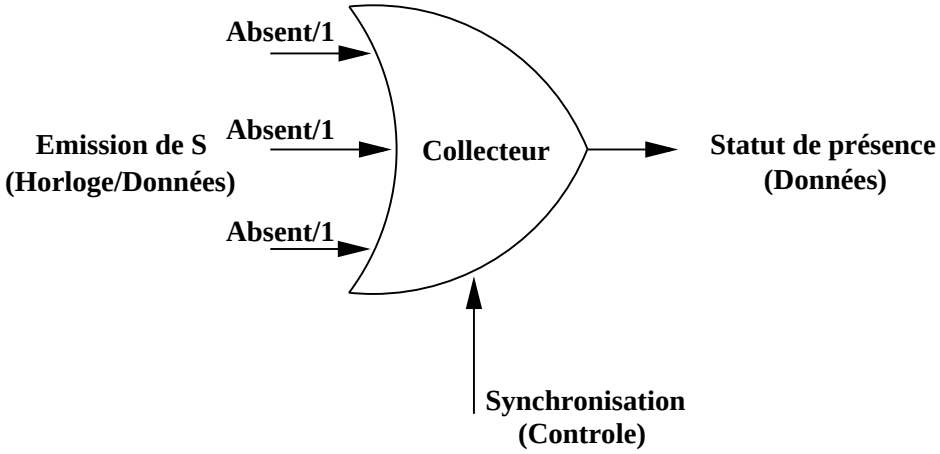


FIG. 7.20 – Collecteur avec gestion de l'absence

dernier modèle de collecteur que nous avons présenté nous est interdit. De plus, puisque l'un des objectifs de la traduction est d'éviter l'activation de la totalité du programme, le seul modèle qui respecte à la fois les objectifs de la traduction et les contraintes du format cible est celui correspondant à la *notification explicite de l'absence*. Dans un premier temps nous allons évaluer le nombre d'instructions que l'implémentation *naïve* de cette technique rajoute au code généré. Nous étudierons ensuite une optimisation permettant de réduire ce nombre.

Évaluation du nombre d'instructions ajoutées

Afin d'évaluer le coût de la notification explicite de l'absence nous allons déterminer une sur approximation du nombre d'instructions que ce modèle de collecteur rajoute au programme initial. Nous allons d'abord nous intéresser au cas d'un *bloc d'activation* simple pour ensuite étudier le cas d'une hiérarchie de *blocs d'activation*.

- *Blocs d'activation* **simples**

  Dans le cas d'un *bloc d'activation* simple (pas de hiérarchie) il faut, dans le pire des cas, rajouter pour chaque émission présente dans une branche d'un *bloc d'activation* une notification d'absence dans l'ensemble des autres branches du *bloc d'activation*.

  Propriété 7.1 *Le nombre d'instructions rajoutées pour la gestion de la notification explicite de l'absence est majoré par :*

  $$nbEmit \times (nbBranche - 1)$$

  avec :
  nbEmit : *Nombre d'émissions dans le bloc d'activation*
  nbBranche : *Nombre de branches du bloc d'activation*

- **Hiérarchie de** *blocs d'activation*

  Dans le cas d'une hiérarchie de *blocs d'activation* l'estimation est un peu plus complexe. Intuitivement pour chaque émission présente dans la hiérarchie il faut ajouter une instruction (notification de l'absence) dans chaque alternative que l'on a rencontrée pour arriver

jusqu'à cette émission. Les Figures 7.21 et 7.22 illustrent les modifications effectuées afin de permettre une notification explicite de l'absence.

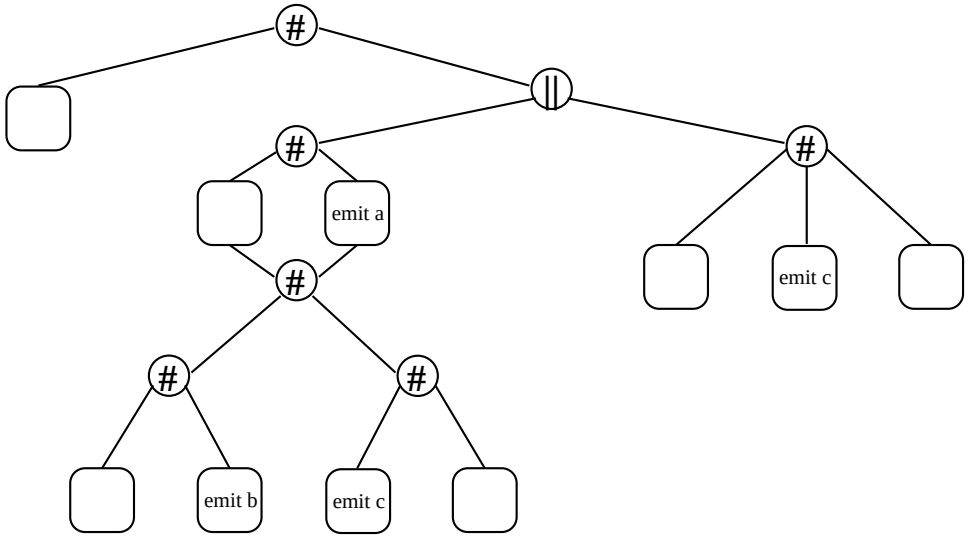


FIG. 7.21 – Hiérarchie des *blocs d'activation* avec émission de signaux

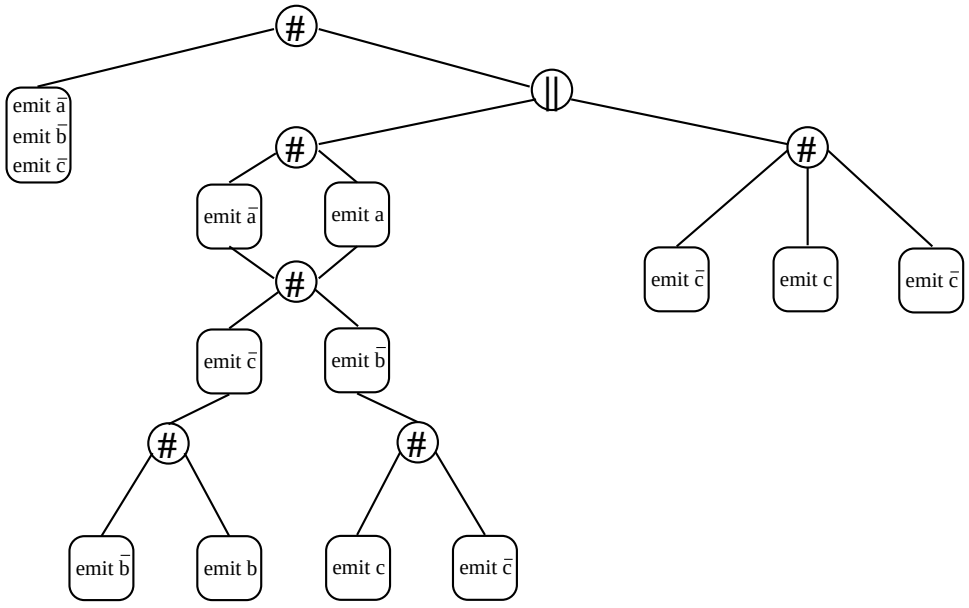


FIG. 7.22 – Hiérarchie des *blocs d'activation* modifiée afin de gérer la notification explicite de l'absence

Il est intéressant de noter que le pire des cas correspond à ce que l'ensemble des émissions se trouve dans les feuilles de la hiérarchie. Il faut, en effet, dans ce cas rajouter des notifications dans l'ensemble des alternatives de la hiérarchie.

Plus formellement, pour chaque émission il faut notifier explicitement l'absence dans l'ensemble des branches exclusives avec la branche dans laquelle se trouve l'émission. Le

nombre total de branches exclusives correspond au nombre de puits du graphe auquel il faut ajouter le nombre de *tests* dont les deux branches sont instantanées. Le nombre de puits est majoré par le nombre de *pause* du programme initial, le nombre de branches exclusives est donc majoré par la somme du nombre de *pause* et du nombre de *tests* du programme.

Propriété 7.2 *Le nombre d'instructions ajoutées dans un programme pour la gestion de la notification explicite de l'absence est majoré par :*

$$nbEmit \times (nbAltBranche)$$

avec :
nbEmit : *Nombre d'émissions dans le programme*
nbAltBranche : *Nombre de branches exclusives*

On peut essayer d'affiner la notion de branche exclusive en ne comptabilisant que celles que l'on a *ignorées* dans la hiérarchie. On obtient alors la propriété suivante :

Propriété 7.3 *Le nombre d'instructions ajoutées dans un programme pour la gestion de la notification explicite de l'absence est majoré par :*

$$nbEmit \times (MaxBlocBranche - 1) \times MaxProdondeur$$

avec :
nbEmit : *Nombre d'émissions dans le programme*
$MaxBlocBranche$: *Le Nombre maximum de branches dans les blocs d'activation*
$MaxProdondeur$: *Profondeur maximum des hiérarchies de blocs d'activation*

Nous allons maintenant évaluer le nombre d'instructions ajoutées par rapport à la taille du programme initial. Pour cela il faut évaluer la taille de *nbEmit* et *nbAltBranche* par rapport à la taille du programme. On détermine trivialement que *nbEmit* est de l'ordre de la taille du programme. De même *nbAltBranche* est lui aussi de l'ordre de la taille du programme. En effet *nbAltBranche* représente le nombre de branches alternatives présentes dans le programme et une branche alternative est soit issue d'un **pause**, soit issue d'un **test**. On ajoute donc dans le pire des cas n^2 instructions au programme initial (n taille du programme). Néanmoins dans la pratique on constate que cette modification ajoute un nombre quasi linéaire d'instructions au programme. En effet pour chaque **emit** on exécute au plus une fois sa notification d'absence puisque les différentes instances se trouvent dans des branches exclusives.

Une optimisation : Réorganisation des *blocs d'activation*

Une optimisation possible pour réduire le nombre d'instructions produites pour la gestion de la notification explicite de l'absence consiste en une réorganisation des *blocs d'activation*. Intuitivement l'idée est de factoriser la notification de l'absence en scindant certains *blocs d'activation*. Dans le cas de la Figure 7.23 on constate que dans la hiérarchie initiale le nombre d'instructions ajoutées est de 9 (ensemble des *notifications explicites de l'absence*), tandis que dans la version modifiée le nombre d'instructions ajoutées est de 6 (5 *notifications explicites de l'absence* et 1 test sur l'état). Le choix de la branche à factoriser peut par exemple s'effectuer en fonction du nombre de *notifications explicites de l'absence* quelle engendre. Il est important de noter que cette technique tend à produire une hiérarchie de *blocs d'activation* en *peigne*. Il

est clair qu'une telle organisation de la hiérarchie des *blocs d'activation* n'est pas optimale pour minimiser le nombre de duplications. Néanmoins sa construction étant simple, elle permet une réduction notable du nombre de duplications sans réaliser un algorithme coûteux en temps. Il faut toutefois noter que cette réorganisation engendre des duplications de tests et donc peut engendrer un surcoût de transfert de donnée.

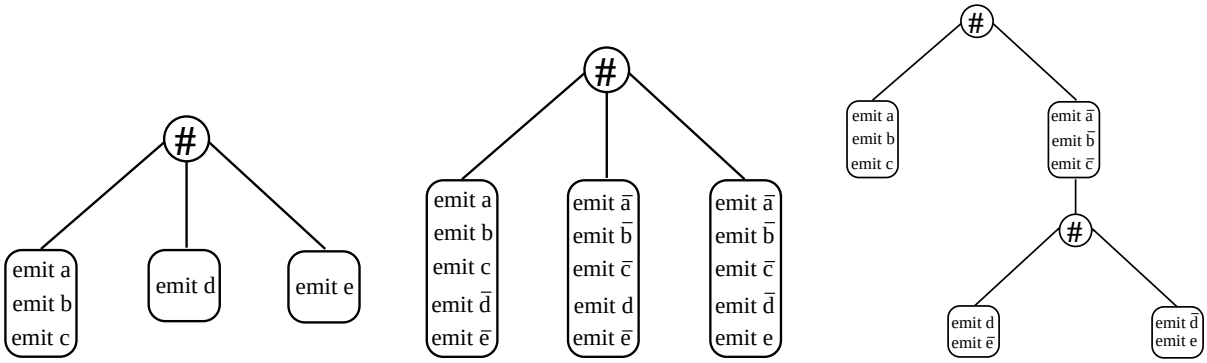


FIG. 7.23 – Réorganisation des *blocs d'activation* afin de réduire le nombre de notifications de l'absence

"There are 10 kinds of people in the world, those who speak binary - and those who don't"

Petit Linuxien pervers

Chapitre 8

Détails de la traduction du format

La traduction que nous allons présenter est réalisée à partir du format **GRC** étendu que nous avons présenté dans le chapitre 7. Le point central de cette traduction est la correspondance que l'on établit entre les **blocs conditionnés** et le conditionnement du logiciel **SynDEx**.

8.1 Mise en œuvre de la traduction

8.1.1 Détail de la traduction

Nous allons dans un premier temps décrire rapidement les aspects principaux de la traduction (encodage de l'état, flot de donnée, ...) puis décrire précisément la traduction de chaque type de nœud présent dans le format **GRC** étendu.

8.1.1.1 Principes généraux de la traduction

- **Encodage de l'état**
  Les **retards** (*registres*) servant à encoder l'état du programme se décomposent en deux groupes, ceux du premier groupe servent à stocker des *entiers* tandis que ceux du second stockent des booléens.
  - **Retard de type entier**
    A chaque nœud *exclusif* du *selection Tree* on associe un **retard**. La valeur contenue dans ce retard sert à déterminer quelle branche du **bloc conditionné**, issu du **SwitchNode** associé à ce nœud exclusif, est active. La valeur *zero* sert à exprimer que le **SwitchNode** est inactif (aucune de ses branches n'est active).
  - **Retard de type booléen**
    A chaque branche de parallèle du *selection Tree* qui n'est pas marquée comme *nonterm* (signifiant que la branche a une durée de vie correspondant à celle du parallèle), on associe un **retard** indiquant le statut d'activation de la branche. La valeur *vrai* indique que la branche est encore active tandis que la valeur *faux* indique que l'exécution de la branche est terminée.
- **Gestion des variables de donnée**
  On fait correspondre à chaque *variable* un **retard** permettant de stocker sa valeur au cours de l'exécution du programme. Toutefois, on notera que ce **retard** n'est pas nécessaire dans

le cas où la valeur d'une variable n'est jamais utilisée d'un instant sur l'autre (Variables *intermédiaires*).

– **Contexte général de la traduction**
La traduction repose sur la correspondance entre les **blocs conditionnés** (issus des nœuds **Switchnode** et **Testnode**) et les **conditionnements** du format de **SynDEx** (Figure 6.2). Cette traduction fait correspondre la hiérarchie des **blocs conditionnés** à une hiérarchie de **conditionnements**.

8.1.1.2 Traduction de chaque type de nœud

Voici le détail de la traduction de chaque type de nœud apparaissant dans le *control/data flow graph* du format **GRC** étendu que nous avons présenté au Chapitre 7. On décrira si nécessaire les interfaces des nœuds ainsi produits. Nous commencerons par présenter le traitement des *blocs conditionnés* puis nous présenterons la traduction des autres types nœuds.

- **blocs conditionnés**
On traduit naturellement les **blocs conditionnés** en **conditionnement SynDEx**. L'interface des nœuds de conditionnement ainsi créés se décompose en quatre parties :
  – **Ports de contrôle**
  Les ports d'entrée correspondent aux *consultations/décodages* de l'état effectués dans les **fonctions conditionnées** constituant le **conditionnement** tandis que les ports de sorties correspondent à des mises à jour de l'état.
  – **Ports de donnée**
  Ces ports correspondent à la lecture et à la mise à jour des **retards** associés aux *variables* que l'on effectue dans chacune des **fonctions conditionnées**. Il faut noter que dans le cas où une variable n'est pas affectée dans la totalité des **fonctions conditionnées**, on est obligé d'effectuer une *lecture* afin de propager l'ancienne valeur de cette variable dans les **fonctions conditionnées** qui ne la modifie pas (Figure 8.1).

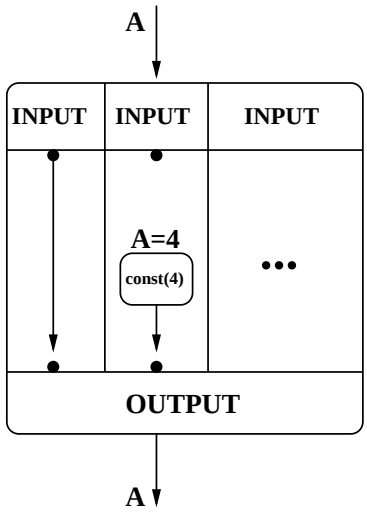


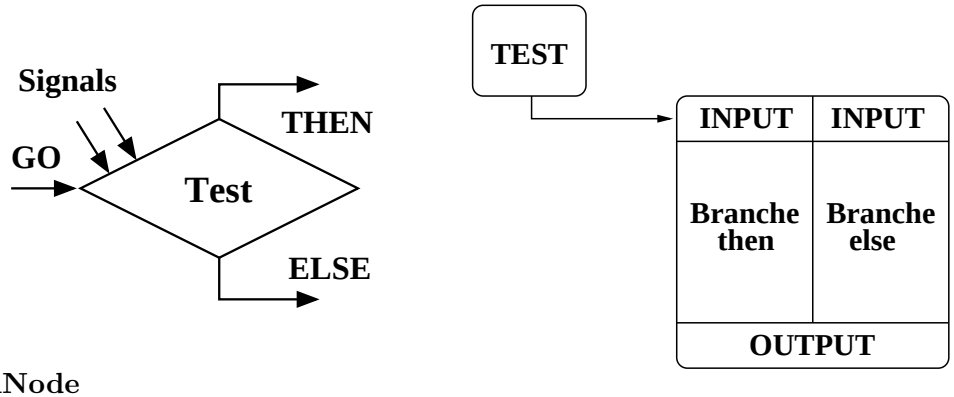
FIG. 8.1 – Propagation des valeurs non modifiées

  – **Ports de signaux**
  Ces ports permettent la consultation et la mise à jour du statut de présence des signaux.

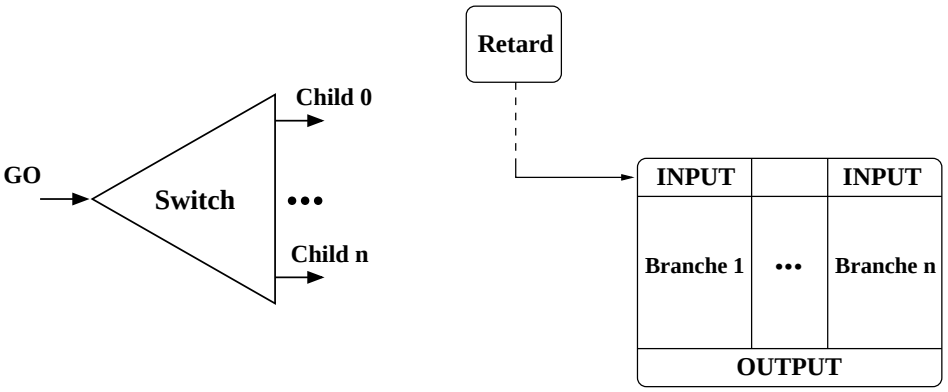
– **Port de conditionnement**
Port permettant de conditionner le nœud de conditionnement.

Les *blocs conditionnés* correspondent à deux types de nœuds (**TestNode** et **SwitchNode**). Nous allons décrire les spécificités de chacun de ces nœuds lors de la création des conditionnements.

– **TestNode**
Le **Port de conditionnement** des *blocs conditionnés* issus de **TestNode** est, comme nous l'avons vu dans le chapitre 7, *alimenté* par la sortie du nœud **TestNode** qui correspond à l'évaluation du test.



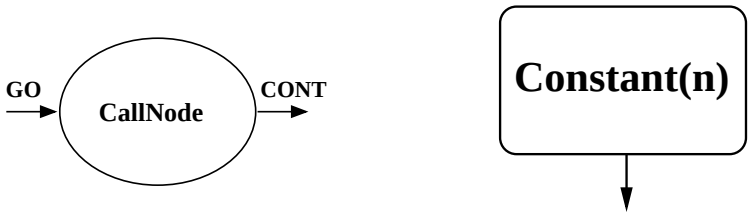
– **SwitchNode**
Dans le cas des conditionnements correspondand à des *blocs conditionnés* issus de **SwitchNode**, le **port de conditionnement** est alimenté par la sortie du retard (registre de type entier) associé au nœud exclusif du *selection tree* qui correspond à ce **SwitchNode**.



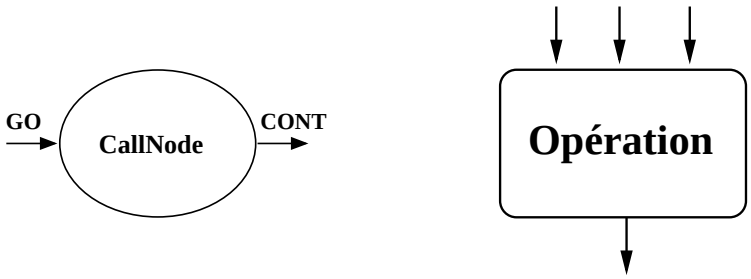
- **CallNode**

Les actions associées aux **Callnodes** peuvent être de deux types, elles correspondent soit à une action agissant sur les *variables de contrôle* du programme, soit à une opération sur les *variables de données* du programme.

– **Action sur les** *variables de contrôle*
Dans le cas où l'action agit sur les *variables de contrôle*, elle correspond donc à l'affectation d'une valeur **constante** à l'un des retards servant à encoder l'état. De ce fait, le nœud **CallNode** est traduit en une **constant SynDEx**. On notera que ce type d'action entraîne l'ajout d'un port de sortie de type contrôle à l'interface du conditionnement englobant.

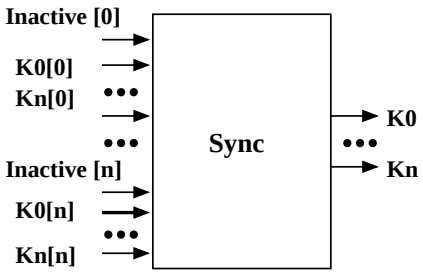


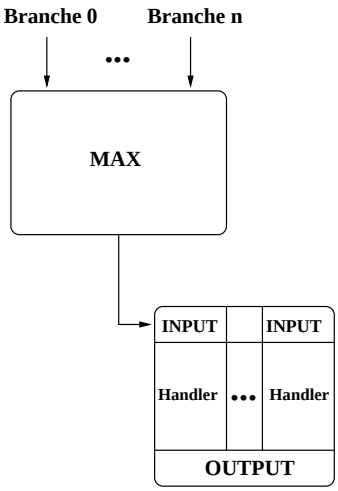
– **Action sur les** *variables de données*
Pour les actions sur les données (*affectation*), la traduction consiste à produire une **fonction** ayant comme *entrée* l'ensemble des *variables* intervenant dans l'expression affectée (Cette fonction peut se résumer à une constante).



Il faut noter que, du fait du principe *d'assignation unique* présent dans le modèle du format **SynDEx**, dans le cas où plusieurs affectations s'effectuent de manière séquentielle on ne connecte que la *dernière* (du point de vue de la causalité).

- **SyncNode**
Du fait que l'on ne traite pas les programmes cycliques, le rôle du **SyncNode** se ramène à calculer le *max* des codes de complétion de chaque branche. Le code de complétion du **SyncNode** ainsi calculé sert de condition pour déterminer quel gestionnaire d'exception il faut exécuter, ceci est assuré par un *bloc conditionné* dont l'entrée de conditionnement correspond à la sortie de la fonction *max*.





- **Ticknode** et **Joinnode**

Ces types de nœuds ne donnent lieu à aucune opération. Ils sont ignorés lors de la traduction. Mais on se rappellera qu'ils ont servi auparavant, soit pour indiquer la racine du graphe (**TickNode**), soit pour marquer la confluence de deux branches exclusives (**Joinnode**).

8.2 Optimisations

8.2.1 Optimisation sur les retards

Nous allons présenter deux façons de disposer les retards au sein de la hiérarchie des conditionnements **SynDEx**. La première permettra une réduction potentielle des coûts de communication en augmentant le nombre de registres tandis que la seconde permettra une réduction du nombre de registres en augmentant potentiellement le coût des communications.

8.2.1.1 Minimisation des coûts de communication

Cette optimisation a pour but de minimiser les communications entre les opérations permettant de consulter et de modifier la valeur des retards (contrôle et donnée) et les retards eux-mêmes. Cette optimisation s'appuie sur le fait que l'utilisation des retards est souvent locale. En effet on observe :

- **Retards correspondant au contrôle**
  La localité de ces retards est inhérente au format **GRC**. En effet la représentation hiérarchique de l'état induit par nature une localité dans l'utilisation des structures de données encodant l'état. On va donc utiliser cette hiérarchie pour *descendre* les registres le plus bas possible dans la hiérarchie.
- **Retards correspondant aux variables**
  La localité des retards correspondant aux variables de données correspond naturellement au *scope* d'utilisation de ces variables dans le *Control/Data flowgraph*.

L'algorithme (Figure 8.2) utilisé pour déterminer l'espace d'utilisation des retards est le même dans les deux cas (contrôle et donnée). Il consiste à rechercher la source du sous graphe minimal dans lequel est utilisé un retard, sachant que cette source doit être un nœud **SwitchNode**. L'algorithme établissi une association entre les nœuds **SwitchNode** et les registres devant être générés à son niveau.

Nous commencerons par définir les propriétés des nœuds que l'on cherche. Nous rappelons que notre objectif est de trouver le nœud **SwitchNode** dominant l'ensemble des utilisations d'un retard. Pour cela nous allons chercher le nœud **SwitchNode** ayant les propriétés suivantes :

- Au moins deux des *branches* du nœud contiennent des nœuds utilisant le retard.
- Tous les nœuds **SwitchNode** ancêtres ont au maximum une de leurs branches qui contient des nœuds utilisant le retard.

En réalité ceci n'est pas tout à fait vrai puisqu'il arrive parfois que la traduction de certains programmes **Esterel** donne un *Control/Data flowgraph* utilisant un retard dans le décodage de l'état (Figure 8.3). Il faut donc rajouter une condition :

- Aucun nœud ancêtre n'utilise le retard.

Nous allons maintenant donner une explication détaillée de l'algorithme :

On définit les éléments suivant :

AddToGenList(node, r)	: Ajoute le registre d'indice r à la liste des registres à générer au niveau du nœud node.
OrOfAllOtherSet(registerSet, i)	: Effectue une union de tous les ensembles du tableau registerSet sauf celui de l'indice i.
ChildIndex(child)	: Si child est le successeur d'un nœud SwitchNode renvoie son indice dans la liste des successeurs.
Succ(node)	: Ensemble des successeurs du nœud node.
RegUse(node)	: Ensemble des registres modifiés par le nœud node.

```
1   ComputeRegisterLocalisationTable(node, registerSet, find)
2   {
3     si node est un SwitchNode alors
4       ∀child ∈ Succ(node) {
5         ComputeRegisterLocalisationTable(child,
6                                          subRegisterSet[ChildIndex(child)],
7                                          find)
8       }
9       pour i ← 1 a | Succ(node) | faire
10        ∀r ∈ subRegisterSet[i] faire
11          si r ∈ OrOfAllOtherSet(subRegisterSet, r)‖r ∉ find alors
12            AddToGenList(node, i)
13            find ← find ∪ {r}
14          finsi
15        finpour
16      finpour
17    sinon
18      ∀fils ∈ Succ(node) {
19        ComputeRegisterLocalisationTable(fils, registerSet, find)
20      }
21      si node est CallNode alors
22        registerSet ← registerSet ∪ RegUse(node)
23        find ← find − RegUse(node)
24      finsi
25    finsi
26  }
```

FIG. 8.2 – Algorithme permettant de trouver la source du sous-graphe minimum où est utilisé un retard, permettant ainsi l'association du nœud **SwitchNode** et des retards que l'on doit générer à son niveau

Le calcul des associations entre les nœuds **SwitchNode** et les retards devant être générés à leur niveau s'effectue par un parcours en profondeur du *Control/Data flowgraph*. On discerne deux cas :

- Le nœud que l'on traite est un **SwitchNode** (lignes 4-14) :

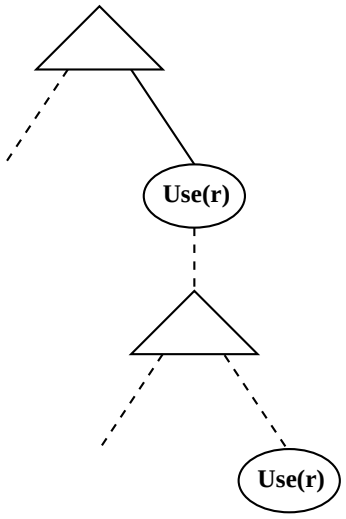


FIG. 8.3 – Utilisation d'un retard dans le décodage de l'état

Dans ce cas on calcule les retards utilisés dans chacune des branches du nœud (lignes 4-6). On détermine ensuite les registres utilisés par plus d'une branche (lignes 7-14). On effectue l'association entre le nœud courant et chacun de ces registres. Le test $find[r]$ permet de vérifier si l'on se trouve dans la situation illustrée par la Figure 8.3.

- Le nœud que l'on traite n'est pas un **SwitchNode** (lignes 16-23) :
  On commence par calculer les registres (retards) utilisés par les successeurs du nœud courant (lignes 16-18), puis dans le cas où ce dernier est un nœud **CallNode** on met à jour l'ensemble des registres utilisés (lignes 19-22). De plus, on indique que les éventuelles associations déjà effectuées pour le registre utilisé par le nœud courant ne sont plus valables (ligne 21), ce qui permet de gérer la situation illustrée par la Figure 8.3.

8.2.1.2 Minimisation du nombre de registres de contrôle

Nous allons présenter une alternative à la disposition des retards de contrôle présentée dans le paragraphe précédent. Ici l'objectif n'est plus la minimisation des communications mais celle des retards nécessaires pour effectuer la distribution. L'idée générale est d'utiliser les informations d'exclusion pour associer plusieurs nœuds **SwitchNode** (exclusifs entre eux) à un même registre (Figure 8.4).

Le nombre de retards nécessaires en utilisant cette méthode se calcule facilement puisqu'il correspond au nombre de retards maximum pouvant être *actifs* au même instant.

Cette disposition des retards a un inconvénient majeur, qui est d'éloigner les retards des opérations qui les utilisent ce qui peut entraîner un surcoût au niveau des communications.

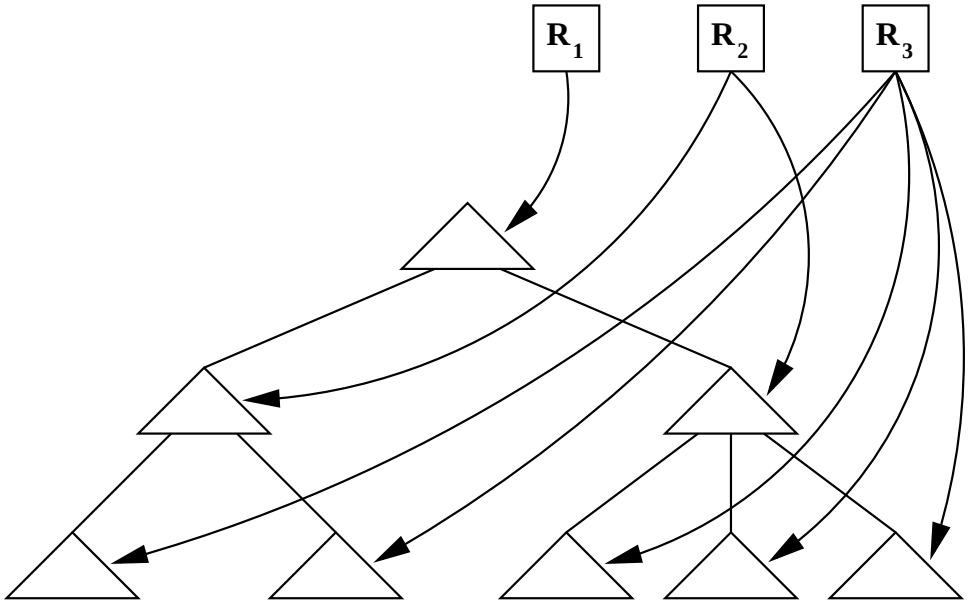


FIG. 8.4 – Schéma général de la minimisation du nombre de registres de contrôle

"En essayant continuellement on finit par réussir. Donc plus ça rate, plus on a de chances que ça marche."

Devise Shadok

Chapitre 9

Proposition d'extension du logiciel SynDEx

La réalisation d'un compilateur de programmes **Esterel** vers **SynDEx** a mis en lumière certaines différences entre les modèles sous-jacents à ces deux logiciels. Le langage **Esterel** fournit certaines hypothèses, qui n'ont pu être totalement utilisées lors de la traduction des programmes **Esterel** vers le format **SynDEx**, pour permettre une optimisation complète des communications. En particulier, une étude de la causalité du programme permet d'assurer que les *variables* utilisées sont bien définies pour l'instant présent. Ces problèmes viennent essentiellement de la façon dont est conçu le conditionnement : en effet toutes les branches doivent partager leurs sorties, ce qui nous oblige à propager la valeur de l'ensemble des variables, quel que soit le contexte d'exécution.

9.1 Les conditionnements revisités

Bien que **SynDEx** introduise une certaine forme de contrôle dans le graphe représentant le flot de données (au moyen des conditionnements), il ne permet pas de gestion réelle de l'absence (qu'elle soit implicite ou explicite). Afin de remédier à ce manque on se propose de changer le modèle utilisé pour le conditionnement pour le raffiner. L'ancien modèle devient alors un cas particulier du nouveau modèle.

9.1.1 Le modèle actuel du conditionnement

Comme nous l'avons vu au Chapitre 5.3.1,p.58 le modèle actuel du conditionnement (Figure 9.1) impose un certain nombre de règles :

1. Toutes les branches d'un conditionnement doivent définir (affecter) la totalité de leurs sorties.
2. Toutes les *sorties* sont partagées par les branches constituant le conditionnement. Une branche ne peut pas avoir de *sortie* qui lui soit spécifique.
3. L'*entrée de conditionnement* (qui définit la branche à exécuter) doit, lors de l'exécution, obligatoirement correspondre à une branche. Intuitivement, cela veut dire que *l'on ne sait*

pas ne rien faire. On notera que cette propriété n'est pas vérifiée à la compilation mais uniquement lors de l'exécution.

La première contrainte empêche toute gestion explicite de l'absence par l'utilisateur. En d'autres termes, il est donc impossible de ne pas définir une variable (affectation) et donc de tester sa présence ou son absence.

La seconde contrainte associée à la première nous oblige à définir l'ensemble des variables à chaque itération même si elles ne sont pas utilisées (*gestion implicite de l'absence*). Pour être plus précis, cela veut dire que le modèle ne permet pas de définir des variables de façon contextuelle. Les variables sont définies pour l'ensemble du programme et doivent être affectées à chaque itération.

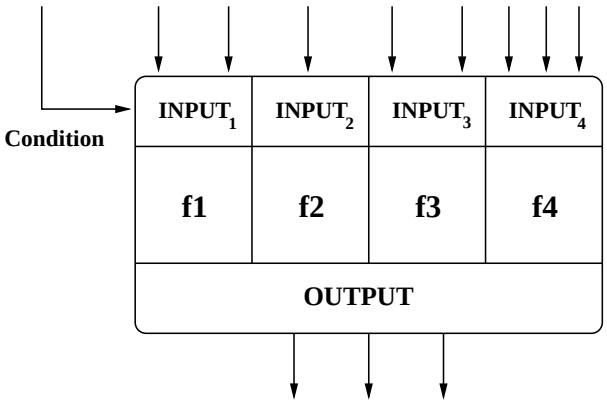


FIG. 9.1 – Le conditionnement dans sa version actuelle

Ceci a pour conséquence que toute gestion de l'absence doit être émulée (comme nous l'avons fait), le modèle ne permettant pas d'exprimer la notion d'absence. On peut toutefois séparer le problème de gestion de l'absence en deux sous-problèmes :

1. **Pouvoir ne pas émettre une donnée :**
   Ce problème consiste à donner la possibilité de ne pas définir l'ensemble des sorties d'un conditionnement, cela revient à avoir des sorties spécifiques pour les branches des conditionnements.
2. **Pouvoir tester explicitement l'absence :**
   Ce problème peut être résolu par l'introduction d'un nouveau type de dépendance entre les opérations permettant la propagation de valeurs par défaut en cas de non affectation. Toutefois il sera nécessaire de préciser le schéma d'exécution afin de déterminer à *quel moment* cette valeur doit être émise.

9.1.2 Un nouveau conditionnement

9.1.2.1 Présentation

Nous allons dans un premier temps introduire un nouveau modèle de conditionnement permettant une gestion implicite de l'absence dans les graphes flot de données utilisés par **SynDEx**.

Dans ce modèle, à la différence du modèle actuel, les sorties d'un conditionnement ne seront pas communes à chaque branche (Figure 9.2).

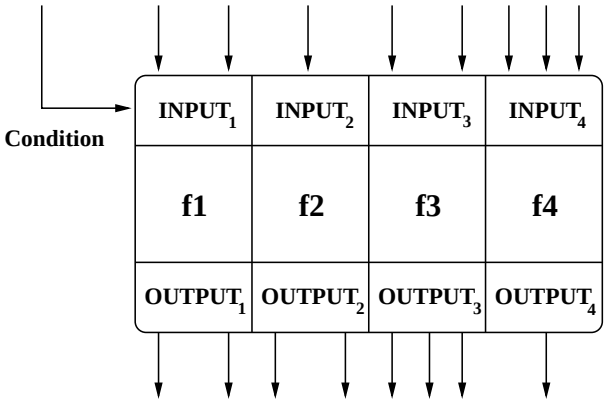


FIG. 9.2 – Une nouvelle version du conditionnement

On notera que le fait de ne pas contraindre toutes les branches à définir l'ensemble des sorties du conditionnement peut amener a priori à ce qu'une opération utilise une donnée qui n'a pas été produite et se retrouve alors indéfinie. Cette nouvelle forme de conditionnement impose donc de réaliser un certain nombre de vérifications supplémentaires sur l'algorithme pour vérifier sa correction. Néanmoins dans le cadre de la traduction d'**Esterel** vers **SynDEx** cette vérification est inutile puisque les propriétés d'**Esterel** nous assurent par construction de la correction de l'algorithme du point de vue des *production/consommation* de variables.

9.1.2.2 Mise à plat du *nouveau* conditionnement

On rappelle (Figure 9.3) le principe de la *mise à plat des sorties d'un conditionnement* définie dans le chapitre 5.3.2,p.60 :

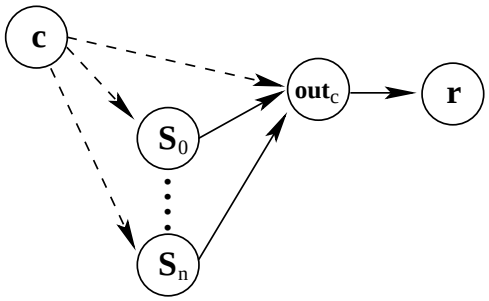


FIG. 9.3 – Détail des communications sortantes du conditionnement lors de la mise à plat de l'algorithme

Dans la version initiale du conditionnement l'ajout du nœud out_c correspondait au besoin de déterminer quelle communication est active. Son rôle dans le nouveau conditionnement ne change pas, hormis le fait qu'il est désormais possible qu'il n'y ait aucune communication active.

9.1.3 Étude de la correction des algorithmes

Nous allons définir les propriétés de correction de l'algorithme du point de vue de la *production/consommation* de données. Pour cela, nous commencerons par définir les `Conditions` ($\mathscr{C}$) qui vont nous permettre de définir l'activité des *opérations* du graphe d'algorithme **SynDEx** (on appelle indifféremment *opération* l'exécution d'un nœud ou la réalisation d'une communication). Intuitivement une `Condition` exprime une contrainte sur la valeur d'une *variable de conditionnement*. Afin de définir l'activité d'une opération nous allons faire correspondre à chaque opération du graphe une proposition logique (`Condition d'activation` $\mathfrak{Ca}$) dont les termes sont des `Conditions`. On pourra alors déterminer si chaque variable utilisée est effectivement produite.

On notera que la valeur associée à une *variable de conditionnement* n'est pas libre (9.1.1), elle doit impérativement appartenir à un ensemble de valeurs *valides*. On appelle cet ensemble *domaine de validité* et on le définit ainsi :

Définition 9.1 *On note $\mathscr{D}(c, C)$ le domaine de validité d'une variable de conditionnement c pour un conditionnement C l'ensemble des valeurs de c auxquelles correspond une fonction conditionnée dans C.*

L'introduction de la notion de *domaine de validité* est due au fait que les *variables de conditionnement* sont typées comme des valeurs entières. Il n'y a pas, au moment de la compilation, de vérification que les valeurs prises par une *variable de conditionnement* correspondent bien à une branche du conditionnement. On peut avoir, pour une même variable, des *domaines de validité* différents en fonction du conditionnement auquel elle est associée. On trouvera dans la Figure 9.4 un exemple correspondant à cette situation. En effet, dans cet exemple le *domaine de validité* de la variable de conditionnement est soit $\{1, 2\}$ lorsqu'elle est associée au sous-conditionnement de gauche, soit $\{3, 4\}$ lorsqu'elle est associée au sous-conditionnement de droite.

Définition 9.2 *On définit une* `Condition` *($\mathscr{C}$) comme un triplet (C, v, val) où C est un conditionnement, v est sa variable de conditionnement et val une valeur entière qui appartient au domaine de validité $\mathscr{D}(C, v)$. On dira qu'une condition est vraie si pour une valuation des variables du graphe on a $v = val$. On notera $\mathscr{C}_{tt}$ une* `Condition` *unique qui est toujours vraie.*

Comme nous l'avons évoqué précédemment, les `Conditions` vont être utilisées comme les termes de propositions logiques, que l'on nommera `Conditions d'activation` ($\mathfrak{Ca}$), permettant de définir l'activité des opérations du graphe d'algorithme.

Notation 9.1 *On appelle* `Condition d'activation` *notée $\mathfrak{Ca}(o)$ la proposition logique définissant l'activité de l'opération o.*

Nous allons maintenant définir comment calculer les `Conditions d'activation` pour les différentes opérations du graphe d'algorithme.

9.1.3.1 Construction des `Conditions d'activation` pour les nœuds

Le calcul des `Conditions d'activation` de chaque nœud s'effectue en parcourant la hiérarchie du graphe d'algorithme. Plus précisément le calcul commencera par calculer les `Conditions d'activation` du nœud *racine* dans la hiérarchie pour descendre vers les *feuilles* (opérations

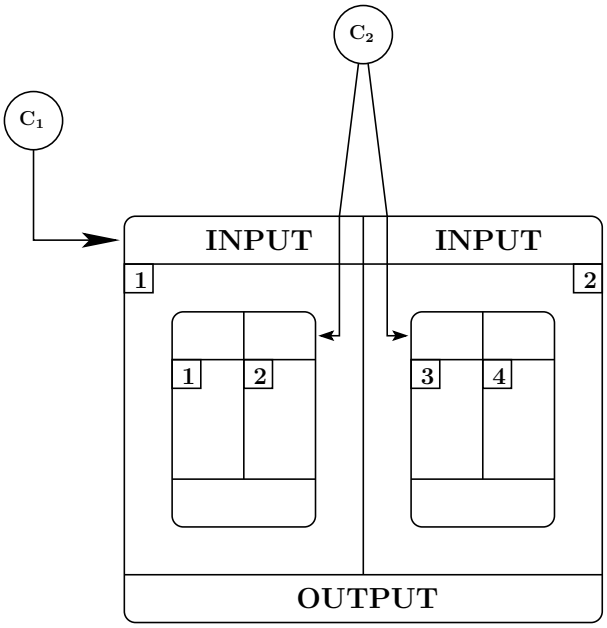


FIG. 9.4 – Exemple de variable de conditionnement ayant des domaines de validité différents en fonction du conditionnement auquel elle est associée

atomiques, sensor, ...). La fonction principale (*main*) étant toujours active, sa `Condition d'activation` ($\mathfrak{Ca}$(main)) est égale à $\mathscr{C}_{tt}$.

Le calcul des `Conditions d'activation` des nœuds est effectué sur la version non aplatie du graphe, car la construction des `Conditions` impose de connaître les *conditionnements*.

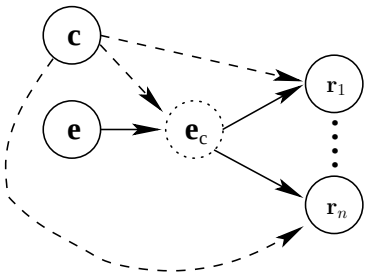
On peut séparer les nœuds en deux catégories, les nœuds conditionnés et les nœuds non conditionnés. On obtient donc deux règles dans le calcul des `Conditions d'activation`.

- **Nœuds non conditionnés**
  La `Condition d'activation` ($\mathfrak{Ca}(n)$) d'un nœud non conditionné (n) défini dans un nœud N est égale à :
  $$\mathfrak{Ca}(n) \;=\; \mathfrak{Ca}(N)$$
  avec $\mathfrak{Ca}(N)$: `Condition d'activation` de N.
- **Nœuds conditionnés**
  Soit un nœud conditionné n, défini dans un nœud N, dont l'entrée de conditionnement est I_{cond}. On notera $\{b_1, \ldots, b_n\}$ l'ensemble de ses branches (*fonctions conditionnées*), tel que la branche b_i est activée si $I_{\text{cond}} = i$. La `Condition d'activation` d'une branche b_i est définie par :
  $$\mathfrak{Ca}(b_i) \;=\; \mathfrak{Ca}(N) \wedge (n, I_{\text{cond}}, i)$$
  avec $\mathfrak{Ca}(N)$: `Condition d'activation` de N.

Comme nous allons le voir dans le paragraphe suivant les `Conditions d'activation` sur les communications vont, au contraire du cas des nœuds, être calculées sur la version aplatie du graphe d'algorithme. Pour permettre ceci il nous faut définir auparavant les `Conditions d'activation` des nœuds supplémentaires que l'on va introduire lors de cette mise à plat.

Lors de la mise à plat de l'algorithme on introduit deux types de nœuds. Le premier type de nœud permet de gérer les communications entrantes d'un *conditionnement*, tandis que le second gère les communications sortantes. Nous allons maintenant définir la méthode de calcul des `Conditions d'activation` pour chacun des ces types de nœuds.

- Communications entrantes

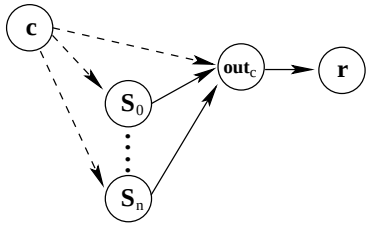


On définit la `Condition d'activation` du nœud e_c gérant l'envoi des données du nœud e dans un conditionnement C comme :

$$\mathfrak{Ca}(e_c) = \bigvee_{i=1}^{n} \mathfrak{Ca}(r_i)$$

avec $(r_1, \ldots, r_n)$: L'ensemble des nœuds du conditionnement C utilisant la valeur produite par n.

- Communications sortantes



On définit la `Condition d'activation` du nœud out_c gérant l'envoi des données depuis un conditionnement C vers le nœud r comme :

$$\mathfrak{Ca}(\text{out}_c) = \bigvee_{i=1}^{n} \mathfrak{Ca}(S_i)$$

avec $(S_1,\ldots,S_n)$: L'ensemble des nœuds du conditionnement C produisant la valeur consommée par r.

9.1.3.2 Construction des `Conditions d'activation` pour les communications

Nous allons maintenant décrire la méthode de calcul des `Conditions d'activation` des variables (communication) qui sont représentées par des dépendances de données (arcs) dans le graphe **SynDEx**. Ce calcul va s'effectuer sur la version aplatie du graphe d'algorithme, ce qui permet de le simplifier. En effet, une communication va être active si le nœud source est actif. On a donc pour une communication c du nœud n_1 vers le nœud n_2 :

$$\mathfrak{Ca}(c) = \mathfrak{Ca}(n_1)$$

avec $\mathfrak{Ca}(n_1)$: `Condition d'activation` du nœud n_1.

On définit :

$\mathsf{Branch}(\mathfrak{n})$	:	Ensemble des branches d'un conditionnement $\mathfrak{n}$.
$\mathsf{Sub}(\mathfrak{n})$	:	Liste des nœuds constituant le nœud $\mathfrak{n}$.
$\mathfrak{Ca}(\mathfrak{n})$	:	`Condition d'activation` du nœud $\mathfrak{n}$.
$\mathsf{CondVar}(\mathfrak{n})$	:	Variable de conditionnement du nœud de conditionnement $\mathfrak{n}$.
$\mathsf{ValCond}(\mathfrak{b})$	:	Valeur de la variable de conditionnement pour que la branche $\mathfrak{b}$ soit active.

```
1  ConstructNodeActivation(n)
2   si n est atomique alors
3     /* vide */
4   sinon
5     si n est un conditionnement alors
6       ∀b ∈ Branch(n) faire
7         ∀s ∈ Sub(b) faire
8           Ca(s) = Ca(n) ∧ (n, CondVar(n), ValCond(b))
9           ConstructNodeActivation(s)
10    sinon
11      ∀s ∈ Sub(n) faire
12        Ca(s) = Ca(n)
13        ConstructNodeActivation(s)
14    finsi
15  finsi
```

FIG. 9.5 – Construction des `Listes de Conditions` des nœuds du graphe d'algorithme

9.1.3.3 Conditions de correction d'un algorithme

Intuitivement un algorithme **SynDEx** est correct, du point de vue des *production/consommation* de variables, si lorsqu'une fonction s'exécute l'ensemble des ses entrées est défini. On notera que cette contrainte s'applique à la version aplatie de l'algorithme, où les fonctions se réduisent donc aux opérations *atomiques*. Cela veut dire que la `Condition d'activation` d'une opération doit être incluse dans celle de chacune de ses entrées. Ceci indique la propriété de correction que doit vérifier l'algorithme :

Définition 9.3 *Un algorithme A est correct si et seulement si :*

$$\forall o \in O, \forall i \in I(o),\ \mathfrak{Ca}(o) \subseteq \mathfrak{Ca}(i)$$

avec :

O	*: L'ensemble des opérations atomiques de l'algorithme A.*
$I(o)$	*: L'ensemble des variables d'entrée de l'opération atomique o.*

9.1.3.4 Mise en œuvre

La vérification de la cohérence des algorithmes **SynDEx** du point de vue de la *production/-consommation* de variables se décompose en deux étapes. Le première consiste à calculer les

On définit :

$\mathsf{E}(\mathfrak{n})$	:	Ensemble des communications *entrantes* du nœud $\mathfrak{n}$
$\mathfrak{Ca}(o)$	:	`Condition d'activation` de l'opération o (communication ou exécution)
$\mathsf{Succ}(\mathfrak{n})$	:	Ensemble des successeurs du nœud $\mathfrak{n}$
$\mathsf{Pred}(\mathfrak{n})$	:	Ensemble des prédécesseurs du nœud $\mathfrak{n}$
$\mathsf{Comm}(\mathfrak{n})$	:	Ensemble des communications réalisées par le nœud $\mathfrak{n}$

```
1  AlgoVerify(n, mark)
2    n : Le nœud que l'on traite
3    mark : L'ensemble des nœuds déjà traités
4    /* On vérifie que les prédécesseurs de n ont été traités */
5    ∀p ∈ Pred(n)
6      si p ∉ mark alors
7        retourner
8      fin
9    /* On vérifie que les données consommées sont produites */
10   ∀e ∈ E(n)
11     si Ca(n) ⊈ Ca(e) alors
12       erreur
13     finsi
14   /* On définit la condition de production de chaque
15      communication effectuée par le nœud n */
16   ∀c ∈ Comm(n)
17     Ca(c) ← Ca(n)
18   /* On ajoute n à l'ensemble des nœuds traités */
19   mark ← mark ∪ n
20   /* On continue la vérification pour les successeurs de n */
21   ∀n_s ∈ Succ(n)
22     AlgoVerify(n_s, mark)
```

FIG. 9.6 – Vérification de la validité de l'algorithme

`Conditions d'activation` de l'ensemble des nœuds de l'algorithme (Figure 9.5). La seconde étape consiste à calculer les `Conditions d'activation` des communications et à vérifier la cohérence des *production/consommation*.

On trouvera dans la Figure 9.6 l'algorithme permettant de déterminer si un graphe d'algorithme **SynDEx** est correct du point de vue de la *production/consommation* de variables. Cet algorithme renvoie un code d'erreur dans le cas ou il détecte un problème de cohérence dans le graphe d'algorithme **SynDEx**.

L'idée principale est de parcourir le graphe d'algorithme de façon récursive en partant des nœuds sources du graphe (**capteur**) jusqu'au nœuds puits (**actionneur**) et le point central est de ne traiter un nœud que lorsque tous ses prédécesseurs ont déjà été traités.

Nous allons maintenant procéder à une explication commentée de l'algorithme. Les lignes 5-8 vérifient que les `Conditions d'activation` des entrées du nœud que l'on traite ont été

calculées. Les lignes 10-13 s'assurent que la `Condition d'activation` du nœud courant (`n`) est bien incluse dans celles de ses entrées (cœur de l'algorithme). Les lignes 16-17 définissent les `Conditions d'activation` des sorties du nœud courant (`n`). La ligne 19 marque le nœud courant comme traité (les conditions de ses sorties sont calculées) et pour finir les lignes 21-22 poursuivent le traitement sur les successeurs du nœud courant (`n`).

9.1.4 Utilisation de ce nouveau conditionnement dans la traduction GRC/-SynDEx

Nous allons introduire une utilisation possible de ce nouveau type de conditionnement dans la traduction que nous avons présentée au Chapitre 8. La réalisation de ce nouveau conditionnement permettrait une gestion des variables beaucoup plus efficace. En effet, dans la traduction actuelle on se rappellera que l'on a été obligé, dans le cas où une variable n'est pas affectée dans la totalité des **fonctions conditionnées** d'un conditionnement, d'effectuer une lecture afin de propager l'ancienne valeur de cette variable dans les branches qui ne la modifient pas (Figure 9.7).

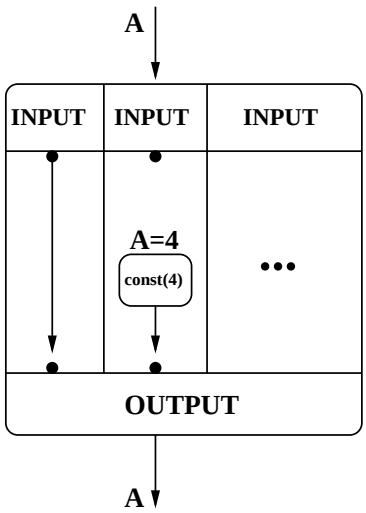


FIG. 9.7 – Propagation des valeurs non modifiées

Avec l'introduction de ce nouveau conditionnement, il n'est plus nécessaire de réaliser cette opération puisque chaque branche peut désormais avoir des sorties spécifiques. Cette modification permet ainsi une réduction du nombre de communications que l'on effectue.

9.2 Introduction de l'absence dans le modèle

Nous allons présenter une deuxième proposition d'extension du modèle régissant **SynDEx**, cette extension ayant pour objectif l'introduction d'une gestion explicite de l'absence. Pour cela nous commencerons par exposer les ajouts, puis les modifications du modèle d'exécution. Ces modifications peuvent être vues comme une amélioration du nouveau modèle de conditionnement exposé précédemment.

9.2.1 Ajout d'un type de transfert de donnée

Afin de permettre une gestion explicite de l'absence dans le graphe d'algorithme **SynDEx** nous proposons l'introduction d'un nouveau type de transfert de données entre les opérations (*optional data communication and execution precedence*). Les deux caractéristiques principales de ce transfert de données sont :

- **Affectation optionnelle et valeur par défaut**
  Ce nouveau type d'arc permet que le nœud *destination* effectue une lecture sans que le nœud *source* ait réalisé d'affectation. Dans ce cas le nœud destination lit une valeur spécifique associée à l'arc (valeur par *défaut*). Cette valeur sera spécifiée par l'utilisateur et devra évidemment être du même type que la variable transportée par l'arc.
- **Précédence d'exécution**
  Ce nouveau type d'arc impose un ordre d'exécution entre le nœud *source* et le nœud *destination*. Plus précisément si le nœud *source* doit être exécuté, il doit l'être avant le nœud *destination*. Cette caractéristique est identique à celle des arcs de type *strong data communication and execution precedence*.

Ce type d'arc peut être vu comme une extension du type **strong data communication and execution precedence** présent dans la version actuelle de **SynDEx** (Voir 5.3,p.56).

L'intérêt de l'ajout de ce type d'arc est de permettre de tester explicitement l'absence. En effet, avec l'ajout de ce type d'arc, tester l'absence revient à tester si la variable transportée par l'arc est égale à la valeur par défaut.

9.2.1.1 Mise à plat du graphe d'algorithme

Nous allons présenter la mise à plat d'un algorithme comprenant des arcs *optional data communication and execution precedence*. Lors de cette mise à plat nous introduirons une opération (nœud) correspondant à l'émission de la valeur par défaut pour chacun des arcs *optional data communication and execution precedence*. Avant cela nous allons introduire la notion de *complément* d'une `Condition` qui nous servira à définir les `Conditions d'activation` des nœuds effectuant l'émission par défaut.

Complément d'une `Condition`

Une `Condition` exprime une contrainte sur une variable (la `Condition` est vraie si la variable vaut une certaine valeur). Le complément d'une `Condition` se définit donc naturellement comme la négation de cette contrainte. On se souviendra que la valeur d'une *variable de conditionnement* n'est pas libre, elle doit appartenir à son *domaine de validité*.

Définition 9.4 *On définit donc le complément* $\overline{\mathscr{C}}$ *d'une* `Condition` $\mathscr{C} = (C, v, val)$ *comme :*

$$\overline{\mathscr{C}} = \bigvee_{\forall w \in \mathscr{D}(v,C), w \neq val} (C, v, w)$$

Détail de la mise à plat

Les Figures 9.8 et 9.9 illustrent la mise à plat d'un algorithme contenant des arcs *optional data communication and execution precedence*.

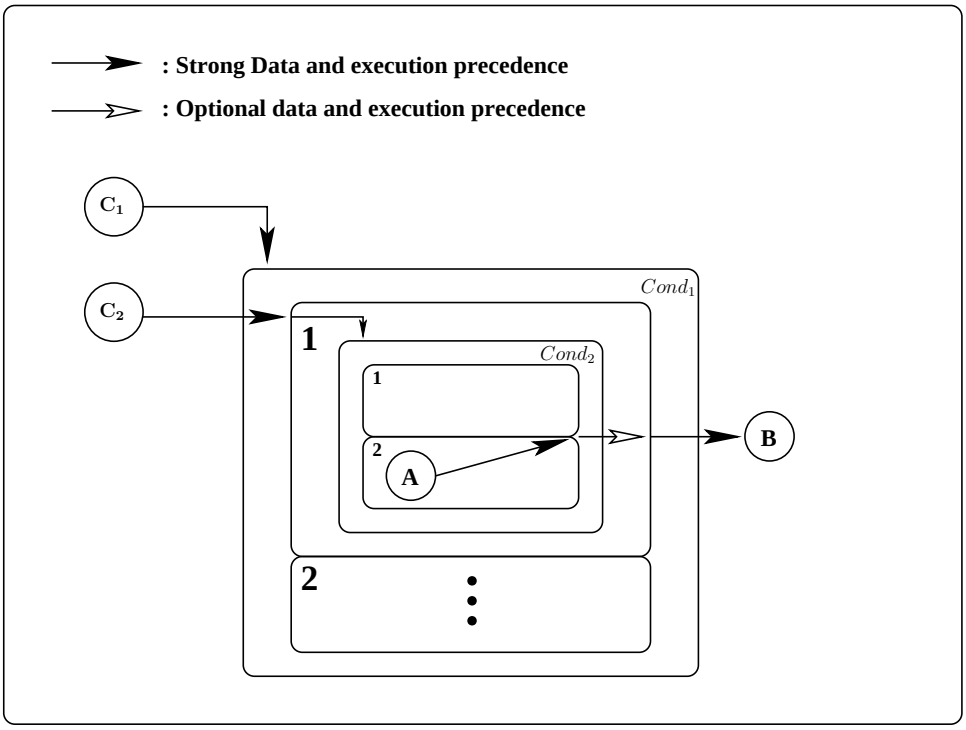


FIG. 9.8 – Gestion des arcs *execution precedence and optional data communication* lors de la mise à plat de l'algorithme

Le point important de l'algorithme de la Figure 9.8 est que l'unique sortie du conditionnement Cond_2 est un arc *optional data communication and execution precedence*.

On trouvera dans la Figure 9.9 la mise à plat de cet algorithme. On notera, par rapport à la mise à plat classique (Section 5.3.2) l'introduction d'un nœud supplémentaire ($\mathbf{A}_d$) correspondant à l'émission de la valeur par défaut lorsque l'opération du nœud **A** n'est pas exécutée. Nous allons définir quand la valeur par défaut doit être émise. Il faut bien évidemment que la sortie ne soit produite par aucun nœud mais il faut aussi que le conditionnent soit actif. En effet on ne désire pas que la valeur par défaut soit produite si le conditionnement n'est pas actif.

Dans notre exemple, puisque **A** est le seul émetteur, on décide donc que la `Condition` du nœud $\mathbf{A}_d$ effectuant l'émission par défaut est :

$$\mathfrak{Ca}(\mathbf{A}_d) = (C_1, 1, Cond_1) \wedge (C_2, 1, Cond_2)$$

Nous allons maintenant définir de façon plus générale la `Condition d'activation` de l'opération (introduite lors de la mise à plat de l'algorithme) effectuant l'émission par défaut. Comme nous l'avons déjà évoqué, ce noeud doit évidement être activé lorsqu'aucun des nœuds effectuant l'émission n'est activé et si le conditionnement effectuant l'émission par défaut est actif.

La Figure 9.10 illustre la *mise à plat* d'un conditionnement Cond, déjà présentée par la Figure 9.3. On remarquera l'introduction du nœud S_d symbolisant l'émission par défaut. On définit donc la `Conditions` $\mathfrak{Ca}(S_d)$ du nœud S_d :

Définition 9.5 *La* `Condition d'activation` *(*$\mathfrak{Ca}(S_d)$*) du nœud (*S_d*) chargé de réaliser l'émission*

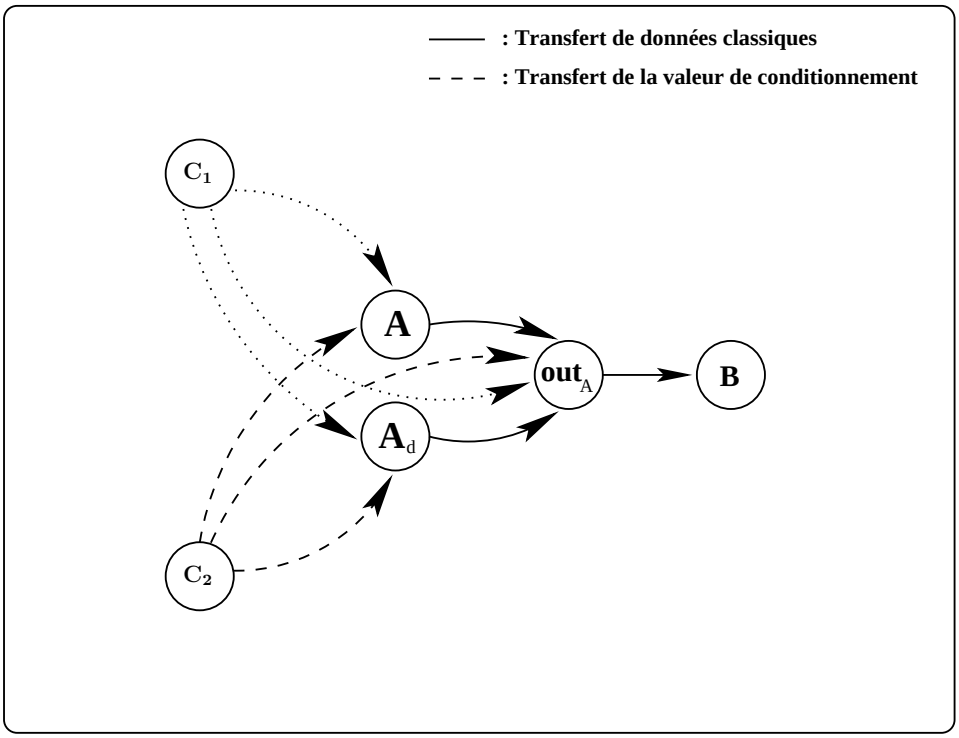


FIG. 9.9 – Mise à plat de l'algorithme de la Figure 9.8.

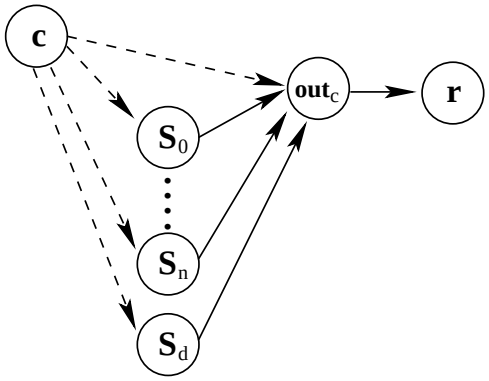


FIG. 9.10 – Détail des communications sortantes du conditionnement lors de la mise à plat de l'algorithme dans le cas d'arcs *execution precedence and optional data communication* .

par défaut dans un conditionnement Cond est :

$$\mathfrak{Ca}(S_d) = \overline{\bigvee_{i=1}^{n} \mathfrak{Ca}(S_i)} \wedge \mathfrak{Ca}(Cond)$$

Cette définition s'explique aisément, $\overline{\bigvee_{i=1}^{n} \mathfrak{Ca}(S_i)}$ exprime qu'il n'y a pas d'émetteur actif tandis que $\mathfrak{Ca}(Cond)$ permet de s'assurer que le conditionnement est actif. La conjonction des deux expressions permet donc d'exprimer exactement ce que l'on désire.

Si l'on revient à notre exemple on obtient :

$$\mathfrak{Ca}(A_d) = \overline{\mathfrak{Ca}(A)} \wedge \mathfrak{Ca}(Cond_1)$$

$$\Leftrightarrow \mathfrak{Ca}(A_d) = \overline{(C_1, 1, Cond_1) \wedge (C_2, 2, Cond_2)} \wedge (C_1, 1, Cond_1)$$

$$\Leftrightarrow \mathfrak{Ca}(A_d) = \overline{(C_1, 1, Cond_1)} \vee \overline{(C_2, 2, Cond_2)} \wedge (C_1, 1, Cond_1)$$

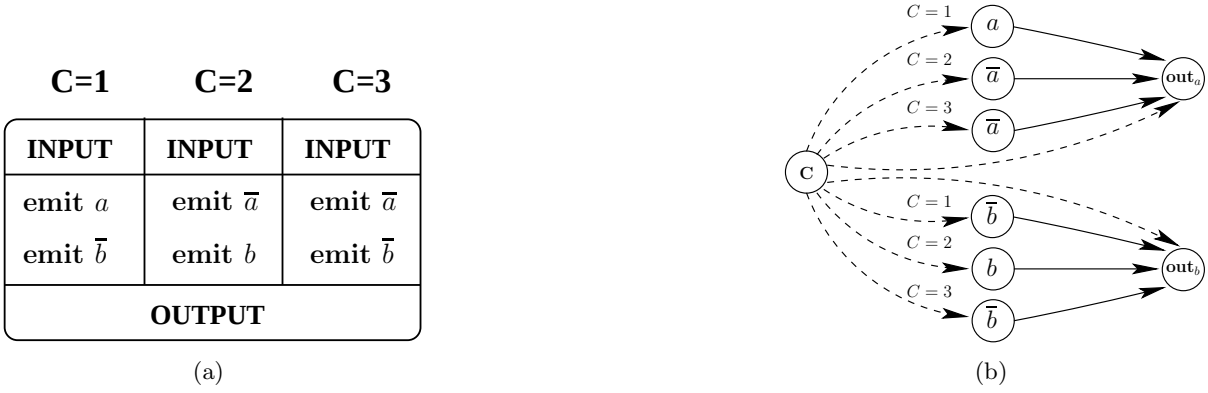
$$\Leftrightarrow \mathfrak{Ca}(A_d) = [(C_1, 2, Cond_1) \vee (C_2, 1, Cond_2)] \wedge (C_1, 1, Cond_1)$$

$$\Leftrightarrow \mathfrak{Ca}(A_d) = (C_1, 1, Cond_1) \wedge (C_2, 1, Cond_2)$$

qui correspond bien à ce que l'on avait espéré.

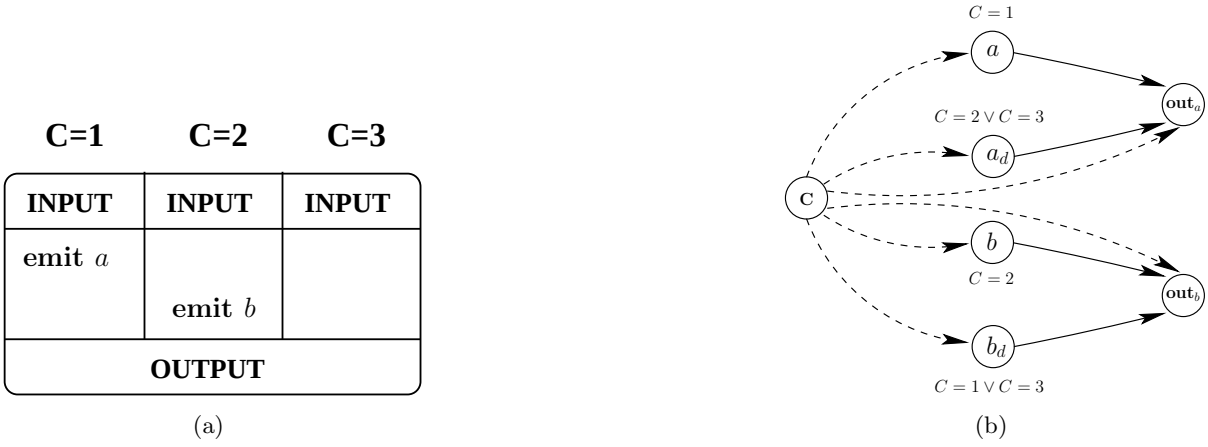
9.2.1.2 Correction de l'algorithme

Il est important de noter que l'introduction de ce nouveau type d'arcs n'impose pas de changer le calcul (Figure 9.6) permettant de déterminer la correction d'un algorithme **SynDEx**.



(a) (b)

FIG. 9.11 – Notification explicite de l'absence des signaux



(a) (b)

FIG. 9.12 – Utilisation des arcs optional data communication and execution precedence dans la gestion de l'absence des signaux

9.2.2 Utilisation de ce type d'arcs dans la traduction GRC/SynDEx

Pour conclure ce chapitre, nous allons présenter une utilisation possible de ce nouveau type d'arcs dans la traduction que nous avons présentée au Chapitre 8. Ce nouveau type d'arcs permet une certaine forme d'amélioration dans la gestion des émissions de signaux. En effet, dans la version actuelle de la traduction on se rappelle que l'on avait dû rajouter des notifications explicites de l'absence (Figure 9.11). Ce nouveau type d'arcs permet d'éviter de faire cette opération (Figure 9.12), l'ajout d'instruction n'est que masqué puisqu'il reste effectif mais de façon moins importante, en effet on a factorisé la notification explicite de l'absence d'un signal. On notera que dans le cas de la gestion des signaux au moyen des arcs de type *optional data communication and execution precedence* on n'utilise pas la possibilité introduite précédemment d'avoir des sorties spécifiques.

On notera que l'utilisation de ce type d'arcs n'a pas pour but d'améliorer les performances du code généré mes plutôt de réduire la taille de ce dernier. En effet la factorisation effectuée permet le regroupement de plusieurs comportements au sein d'une seule instruction mais de ce fait la `Condition d'activation` de cette instruction s'en trouve complexifiée et donc potentiellement plus longue à calculer.

"Les trois points terminateurs me font hausser les épaules de pitié. A-t-on besoin de cela pour prouver que l'on est un homme d'esprit, c'est-à-dire un imbécile ? Comme si la clarté ne valait pas le vague, à propos de points !"

Lautréamont, Poésies II.

Chapitre 10

Conclusions et perspectives

10.1 Conclusions

Nous avons présenté dans cette thèse une traduction des programmes **Esterel** dans le format utilisé par le logiciel **SynDEx** pour la représentation des algorithmes. Ce travail s'inscrit dans le désir d'obtenir un outil permettant d'automatiser les opérations allant de la conception de programmes jusqu'à leur implémentation. En effet le langage **Esterel** permet la conception ([3]) de programmes de haut niveau ainsi que leur vérification ([10]), tandis que le logiciel **SynDEx** permet de réaliser la distribution de programmes, exprimés sous forme *flot de donnée*, sur des réseaux de processeurs. La réalisation de cette interface permet ainsi d'obtenir une chaîne quasi automatique partant du *design* jusqu'à l'implémentation.

Cette traduction peut être décomposée en deux étapes distinctes. La première étape a consisté à transformer le format **GRC** depuis un modèle purement flot de contrôle dans lequel les opérations sur les données étaient externalisées, vers un modèle mixte où les dépendances de données sont parfois venues se substituer à une partie du flot de contrôle. De plus, lors de cette transformation, on a créé des blocs hiérarchiques (*bloc d'activité*) servant à transcrire les relations d'exclusion entre les différentes parties du programme.

La seconde étape a consisté à traduire notre format **GRC** étendu dans le format utilisé par **SynDEx**. Cette traduction repose sur une correspondance entre les *blocs d'activité* et le conditionnement de **SynDEx**.

La réalisation d'un prototype réalisant cette traduction a constitué une part importante de cette thèse. Elle se base sur l'exploitation du compilateur **Esterel** réalisé par Dumitru Potop-Butucaru durant son doctorat ([32]).

10.2 Perspectives

On peut imaginer deux grands axes dans les évolutions et modifications que l'on pourra réaliser sur la version actuelle du compilateur. Le premier correspond évidemment à des ajouts d'optimisation dans le processus de traduction, tandis que le second s'inscrit plus dans un désir

d'amélioration du processus de conception de façon globale, ce qui peut nécessiter une réécriture partielle ou totale du compilateur.

10.2.1 Amélioration de la traduction actuelle

Cette traduction bien que correcte peut sûrement être encore améliorée. Une analyse plus fine des programmes **Esterel** et de leurs propriétés peut amener à la réalisation d'optimisations, en particulier dans la gestion de l'absence des signaux. Par exemple, une analyse dynamique des programmes (évaluation partielle) pourrait conduire à découvrir des propriétés supplémentaires sur les émissions de signaux, permettant ainsi une amélioration dans leur gestion.

10.2.2 Remontée d'informations au niveau source

Le deuxième axe correspond à une amélioration du processus de conception en permettant la remontée d'informations du logiciel **SynDEx** vers le code source **Esterel**. En effet dans un contexte où la taille et la complexité des applications sont croissantes, la remontée d'informations au sein du code source devient un enjeu majeur.

Le logiciel **SynDEx** lors du processus de distribution réalise une estimation (maximisation) de la durée maximum d'un *cycle*. Dans le cas où cette durée est supérieure aux contraintes de réalisation l'outil que nous avons présenté ne permet pas de façon simple la mise en relation des structures présentes dans **SynDEx** avec le code source du programme **Esterel**.

En rajoutant un moyen d'associer les représentations **SynDEx** avec les structures d'**Esterel** au niveau source on pourrait ainsi déterminer les parties du programme initial qui *posent* problème et envisager une réécriture prenant en compte ces informations. Cette amélioration peut être réalisée en effectuant une modification de notre compilateur ainsi que celui permettant la génération du code **GRC**. On peut néanmoins imaginer une réécriture du compilateur afin qu'il parte directement du source **Esterel**, ceci rendrait plus facile la mise en correspondance des structures **Esterel** et **SynDEx**.

Bibliographie

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Charles André. SyncCharts : A visual representation of reactive behaviors. RR 95-52, I3S, 1995.

[3] Charles André. Representation and analysis of reactive bahaviors : A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29. IEEE-SMC, july 1996.

[4] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software co-design of embedded systems : the POLIS approach.* Kluwer Academic Publishers, 1997.

[5] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, volume 79(9), pages 1270–1282, 1991.

[6] Albert Benveniste, Benoit Caillaud, and Paul le Guernic. From synchrony to asynchrony. In *International Conference on Concurrency Theory*, pages 162–177, 1999.

[7] Gérard Berry. The constructive semantics of pure Esterel. Draft book available at http ://www.esterel-technologies.com/v3/ ?id=18162, July 1999.

[8] Gérard Berry and the Esterel Team. The esterel v5.91 system manual. Available at http ://www.esterel-technologies.com/v3/ ?id=18162, June 2000.

[9] Gérard Berry. The Esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[10] Amar Bouali. Xeve : an esterel verification environment. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 500–504. Springer-Verlag, 1998.

[11] P. Caspi and A. Girault. Execution of distributed reactive systems. In S. Haridi, K. Ali, and P. Magnusson, editors, *First International Conference on Parallel Processing, EURO-PAR'95*, volume 966 of *LNCS*, pages 15–26, Stockholm, Sweden, August 1995. Springer Verlag.

[12] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, USA, October 1994. ISCA.

[13] D.M. Chapiro. *Globally Asynchronous Locally Synchronous Systems.* PhD thesis, Stanford Universiry, October 1984.

[14] Julien Forget, Christophe Lavarenne, and Yves Sorel. *SynDEx v6 - User Manual.* http ://www-rocq.inria.fr/syndex/v6/manual/manual.html.

[15] E. Sentovich G. Berry. An implementation of constructive synchronous constructive programs in polis. *Formal Methods in System Design*, 17(2) :135–161, 2000.

[16] A. Girault. *Sur la Répartition de Programmes Synchrones.* Phd thesis, INPG, Grenoble, France, January 1994.

[17] A. Girault and P. Caspi. *Ocrep.* http ://www.inrialpes.fr/bip/people/girault/Ocrep/index.html.

[18] A. Girault and C. Ménier. *Screp.* http ://www.inrialpes.fr/bip/people/girault/Screp/index.html.

[19] A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *2nd International Workshop on Embedded Software, EMSOFT'02*, volume 2491 of *LNCS*, pages 266–281, Grenoble, France, October 2002. Springer-Verlag.

[20] Thierry Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* PhD thesis, Paris XI Orsay, Paris, France, 2000.

[21] Thierry Grandpierre, Christophe Lavarenne, and Yves Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *CODES'99 7th International Workshop on Hardware/Software Co-Design*, Rome, Italy, May 1999.

[22] Thierry Grandpierre and Yves Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *First ACM & IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[23] Rajiv Gupta, Santosh Pande, Kleanthis Psarris, and Vivek Sarkar. Compilation techniques for parallel systems. *Parallel Comput.*, 25(13-14) :1741–1783, 1999.

[24] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, volume 79(9), pages 1305–1320, 1991.

[25] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer academic Publishers, 1993.

[26] Christophe Lavarenne, Omar Seghrouchni, Yves Sorel, and Michel Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *European Control Conf.*, Grenoble, France, July 1991.

[27] P. LeGuernic, T. Gauthier, M. LeBorgne, and C. LeMaire. Programming real-time applications with signal. In *Proceedings of the IEEE*, volume 79(9), pages 1321–1336, 1991.

[28] Christophe Macabiau. *SynDEx v6 : Heuristiques.*

[29] Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[30] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 52. IEEE Computer Society, 2000.

[31] C. Ménier. Répartition de circuits séquentiels. Rapport de stage, ENS-Lyon, Lyon, France, July 2001.

[32] Dumitru Potop-Butucaru. *Optimizations for faster simulation of Esterel programs.* PhD thesis, Ecole des Mines, Paris, France, 2002.

[33] Mihaela Sighireanu. The specification of DC2SDX translator. Rr, INRIA, 1999. http ://www.liafa.jussieu.fr/ sighirea/DC2SDX/Sighireanu-99-c.ps.gz.

[34] Yves Sorel. Massively parallel systems with real time constraints, the algorithm architecture adequation methodology. In *Conf. on Massively Parallel Computing Systems*, Ischia, Italy, May 1994.

[35] O. Tardieu and R. de Simone. Curring schizophrenia by program rewriting in `Esterel`. In *MEMOCODE'04*, july 2004.

[36] Annie Vicard and Yves Sorel. Formalization and static optimization for parallel implementations. In *DAPSYS'98 Workshop on Distributed and Parallel Systems*, Budapest, Hungary, September 1998.

Résumé

La réalisation des systèmes embarqués modernes a fait naître le besoin de techniques et outils d'aide à la distribution automatique ou semi-automatique de code. En effet, l'évolution de la plate-forme *matérielle/logicielle* utilisée pour la réalisation de tels systèmes a tendance à s'orienter vers des réseaux et architectures de processeurs hétérogènes. Dans le cadre des langages synchrones, qui offrent déjà de nombreux avantages liés à une sémantique formelle (*Model based design*, outils de vérification, ...), la possibilité de distribution optimisée de code rendrait ces technologies encore plus « *attrayantes* ».

Notre travail a consisté à rapprocher le langage **Esterel** de logiciels existants permettant d'effectuer de la distribution de code. Notre choix final s'est porté sur le logiciel **SynDEx**, puisque ce dernier a l'immense avantage de reposer sur les même fondements théoriques (*systèmes synchrones*) que le langage **Esterel**.

Bien que reposant sur les mêmes fondements théoriques, le langage **Esterel** et le logiciel **SynDEx** se distinguent par le fait que le premier correspond à un style de langage orienté *flot de contrôle* tandis que le second utilise pour la représentation des algorithmes un style orienté *flot de données*. Dans ce contexte, une partie de notre travail a consisté à transformer la représentation des programmes **Esterel** vers une représentation orientée *flot de données*.

Abstract

The realization of modern embedded systems increase the requirements of automatic or semi-automatic code distributing tools. Indeed the evolution of the *hardware/software* platform use for the realization of such systems turn toward heterogeneous chip network.

Within the context of the synchronous language, all of their advantages, the possibility of distributing optimized code make this technologies more *attractive*.

Our works consist connect the `Esterel` language with existing software, which can make the code distribution. Our final choice is the `SynDEx` software, because this software use the same theoretical foundation (synchronous systems) than `Esterel` language.

Even, if the `SynDEx` software and the `Esterel` language use the same theoretical foundation, the `Esterel` language and the `SynDEx` software is different in the way they represent algorithm. Indeed the `Esterel` language is a imperative and control oriented language, while the `SynDEx` software is data flow oriented. In this circumstances, one part of our work consisted to transform the representation of `Esterel` program to data flow representation. Furthermore, in the goal of increase the evaluation of programs we exploit the structural information of the program to minimize the really active part of the program.