Mémoire de **D**iplôme d'**E**tudes **A**pprofondies
de
l'école doctorale **S**ciences et **T**echnologies de l'**I**nformation et de la **C**ommunication
Université de **N**ice – **S**ophia **A**ntipolis

Réseaux et **S**ystèmes **D**istribués

Bertrand BLANC

Traduction d'Esterel Pur v5 en VHDL Comportemental

1 juillet 2002

Institut **N**ational de **R**echerche en **I**nformatique et **A**utomatique Sophia-Antipolis
Action TICK: Etude et Implantation des Systèmes Réactifs Synchrones

encadreurs:

Robert de Simone – Laurence Pierre

Table des matières

Introduction et Motivation

Le travail de DEA s'int`egre au projet de l'´equipe TICK qui se focalise sur les activit´es li´ees au langage synchrone Esterel. Esterel, Lustre et Signal sont les trois principaux repr´esentants de la classe des langages synchrones, n´es du fruit de la recherche de laboratoires français.

La nature des formalismes synchrones permet de mod´eliser, programmer et valider des applications embarqu´ees `a temps critique. Divers outils de simulation, de v´erification ou de synth`ese ont ´et´e d´evelopp´e depuis une vingtaine d'ann´ees. Le langage a ´evolu´e jusqu'`a aujourd'hui avec la derni`ere version libre implant´ee par le compilateur v5. L'outil graphique XES permet de simuler un programme Esterel en lui donnant une s´erie d'entr´ees: l'utilisateur observe `a chaque r´eaction les signaux ´emis. XEVE permet la v´erification de propri´et´es du syst`eme. Elles sont sp´ecifi´ees par un *observeur* ´ecrit en Esterel. Le moteur de v´erification va essayer d'exhiber une s´equence d'entr´ees falsifiant la propri´et´e, si il ´echoue, la propri´et´e est satisfaite: la description Esterel est correcte.

Esterel est actuellement compil´e en syst`emes d'´equations bool´eennes, elles mêmes formant une repr´esentation textuelle des sch´emas de portes logiques des circuits digitaux. Ce niveau de granularit´e dans la cible de traduction apparait cependant trop fin du point du vue de la conservation de la structure d'automate repr´esent´ee par un programme Esterel, qui disparait totalement. La probl´ematique consiste donc `a proposer une alternative mettant en relation Esterel et le langage de description de mat´eriel, VHDL, de sorte `a conserver le haut niveau de structure comportementale tel qu'exprim´e en Esterel.

Une des motivations de cette traduction vient du fait que VHDL a ´et´e invent´e dans un but de simulation efficace de circuits synchrones. La compilation actuelle d'Esterel en ´equations tri´ees, demande d'´evaluer toutes les ´equations `a l'ex´ecution, même si les parties du programme correspondantes sont inactives, ce qui est un frein `a une simulation/ex´ecution efficace en nombre d'instructions.

Une seconde motivation est de pr´eserver la structuration hi´erarchique du programme Esterel d'origine vers le code produit, justement pour permettre de conserver ce gain de n'ex´ecuter que des processus actifs dans le cycle de simulation

Actuellement, le compilateur Esterel g´en`ere un code dont les instructions sont difficiles `a associer `a celles du programme Esterel d'origine. Le sujet propos´e traite donc de la traduction d'un programme Esterel vers une architecture VHDL comportementale.

La proposition *IEEE P1076.6/D2.01, Draft Standard For VHDL Register Transfer Level Synthesis* d´ecrit le sous-ensemble de VHDL accept´e par les outils de synth`ese actuellement distribu´es sur le march´e tel que DC de Synopsys. Le travail r´ealis´e dans le cadre du stage de DEA permettra d'aboutir `a une transformation d'Esterel en VHDL. Etablir un lien, `a un haut niveau d'abstraction, entre Esterel et VHDL permettra d'aboutir `a des mod´elisation mixtes, `a l'utilisation des outils de CAO de chaque langage pour l'autre, ...

Présentation des chapitres

Esterel et VHDL

Les deux premiers chapitres pr´esentent le langage source: Esterel Pur v5 et le langage cible: VHDL 93. Nous les situons en exhibant leur pouvoir d'expression et leur s´emantique respective de simulation. Esterel est un langage r´eactif synchrone fond´e sur plusieurs hypoth`eses fortes permettant d'assurer la coh´erence entre toutes les activit´es li´ees au langage: simulation, compilation, optimisation, v´erification, ... et ´egalement l'interpr´etation dans des mod`eles math´ematiques (automates r´eactifs ou circuits synchrones). VHDL est d´edi´e `a la description de circuits asynchrone dans un environnement de simulation orient´e ´ev`enements. L'analyse de ces deux langages permet d'exhiber une repr´esentation orient´ee simulation VHDL des constructions d'Esterel.

Détermination d'un format Esterel intermédiaire qui conserve la structure du programme Esterel d'origine

Le troisi`eme chapitre est consacr´e `a la pr´esentation de plusieurs formats interm´ediaires Esterel offrant divers avantages. Mais lequel d'entre eux choisir permettant une traduction la plus facile possible en VHDL? Plusieurs crit`eres d´eterminants entrent en jeu: la conservation de la structure du programme Esterel, une API disponible permettant le parcours de l'arbre syntaxique abstrait, la r´esolution potentielle des probl`emes de duplication n´ecessaire de code pour d´esambigüer des comportements, probl`emes connus en Esterel sous les noms de schizophr´enie et r´eincarnation. Notre choix s'est port´e sur le format GRC qui regroupe un ensemble de propri´et´es satisfaisantes dans la vision VHDL.

Détermination de la traduction des instructions de ce format en VHDL de sorte à préserver la sémantique d'origine

Le troisi`eme chapitre traite en seconde partie de l'expression des nœuds du format GRC dans un syst`eme de processus orient´e moteur de simulation VHDL, respectant la s´emantique de simulation Esterel. J'ai donn´e une s´emantique aux nœuds, exprim´ee par un ensemble de signaux, de processus parall`eles et diverses constructions atomiques. Un programme Esterel exprim´e par ce syst`eme de processus m'a permis ensuite de me consacrer `a la composition d'instructions, leur r´eduction et la r´esolution des probl`emes de causalit´e ou de r´eaction `a l'absence.

Production de code VHDL

Le quatri`eme chapitre montre la mani`ere dont j'ai traduit les instructions du syst`eme de processus repr´esentant un programme Esterel en VHDL. Les phases successives de traduction s'appuient sur le même exemple depuis Esterel jusqu'`a VHDL, montrant la qualit´e du code obtenu par le compilateur que j'ai r´ealis´e en C++ durant le stage.

Perspectives

En conclusion, nous ferons le comparatif entre autres en termes de nombre de registres et nombre de signaux d'un exemple sp´ecif´e en Esterel compil´e dans deux styles de code VHDL. Le premier, orient´e comportemental, est le fruit du projet de DEA et le second, plus bas niveau, repr´esente la liste d'´equations tri´ees dans une syntaxe VHDL. Puis, nous terminerons par les perspectives offertes par ce travail dans le cadre de projets plus ambitieux.

Chapitre 1

Présentation d'Esterel

Esterel est un langage r´eactif synchrone cr´e´e par G´erard Berry et d´evelopp´e en collaboration entre l'**I**nstitut **N**ational de **R**echerche en **I**nformatique et **A**utomatique (INRIA) et le **C**entre de **M**ath´ematiques **A**ppliqu´ees (CMA) voila plusieurs ann´ees. Il permet de d´ecrire de mani`ere intuitive les parties contrôle de syst`emes r´eagissant `a des stimuli provenant de l'environnement ext´erieur.

Esterel est le sujet actif de nombreux th`emes de recherche ainsi que d'utilisation industrielles.

- L'action INRIA-TICK dirig´ee par Robert de Simone propose de nouveaux formats de simulation d'Esterel, de simplifications, de traduction vers d'autres langages tels que VHDL ou Syndex, utile pour une vision *co-design* ou partitionnement logiciel-mat´eriel.
- L'´equipe SPORTS de Charles Andr´e, cr´eateur du formalisme graphique *SyncCharts* bas´e sur Esterel, s'oriente dans une direction logiciel et UML.
- France Telecom R&D propose des solutions de simulation logicielles tr`es rapides, permettant de simuler quelques parties contrôles dans le domaine des Telecom.
- Steven Edwards, au sein des laboratoires Synopsys, `a de même propos´e des implantations rapides permettant la simulation logicielle de circuits.
- La soci´et´e Esterel Technologies industrialise le langage, les outils, l'expertise et le conseil aupr`es de grands groupes:
  - *Semiconducteurs et Télécom*: ST Microelectronics, Texas Instrument, Thales, Motorola, France Telecom
  - *Automobile, transport et énergie*: Audi, RATP, Schneider Electric, EDF, ...
  - *Aérospatiale et Défense*: Eurocopter, Dassault, Airbus, Zodiac, ...
- Dans le monde acad´emique:
  - *Amériques*: Universit´e de Columbia (USA), Universit´e du Minnesota (USA), Universit´e de Berkeley (USA), Universit´e de Calgary (Canada)
  - *Europe*: Ecole des Mines de Nantes, Universit´e de Sheffield (UK), ENST, Universit´e de Kiel (Allemagne), ESSI, Universit´e Linkoping (Su`ede), Ecole d'ing´enieurs de Gen`eve (Suisse), Universit´e de Budapest (Hongrie)
  - *Asie*: Universit´e de Singapore (Inde), Universit´e de Cor´ee, Institut indienne des technologies (Inde)

Afin d'illustrer diff´erentes propri´et´es ou constructions syntaxiques des langages trait´es, nous allons nous appuyer sur un exemple choisi pour sa simplicit´e même si il ne refl`ete pas du pouvoir expressifs global des langages Esterel ou VHDL. Il s'agit d'un additionneur 1-bit $add$ `a trois entr´ees: i et j les deux bits `a additionner avec la retenue entrante c_{in} et deux sorties: s la somme et c_{out} la retenue sortante.

$$\forall t \in \mathbb{R},\ \forall c_{in}(t), i(t), j(t) \in \mathbb{B},\ [\![add(i(t),j(t),c_{in}(t))]\!] \ =\ ((c_{in}(t) + i(t) + j(t))\ mod\ 2,\ (c_{in}(t) + i(t) + j(t))\ /\ 2)$$

la fonction add doit explicitement être appel´ee — et `a tout moment $t \in \mathbb{R}$ — en lui pr´ecisant ses param`etres. L'ex´ecution de add suspend l'ex´ecution du programme l'ayant appel´e pour une dur´ee $\epsilon \in \mathbb{R}$ ind´etermin´ee mais born´ee. Le couple (s, c_{out}) rendu par le calcul de add est la somme des entr´ees (s) et la retenue sortante (c_{out}).

L'approche synchrone est fond´ee sur des hypoth`eses permettant de s'affranchir de ces consid´erations. Un plus grand pouvoir d'expressivit´e est ainsi offert au programmeur avec par exemple l'introduction du parall´elisme.

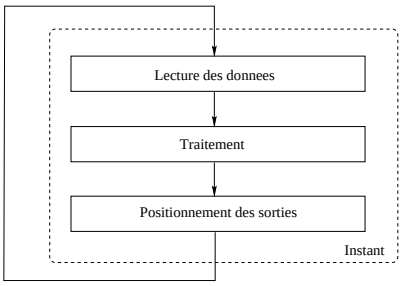


FIG. 1.1 –. Boucle de simulation d'un instant

La boucle de simulation dans l'*instant* est la s´equence de trois actions: lecture des entr´ees, traitement, puis positionnement des sorties.

1.1 Constructions Esterel Pur v5

Un utilisateur doit pouvoir exprimer simplement un ´el´ement tel que l'appel `a une routine ou une op´eration arithm´etique ´el´ementaire. Esterel permet l'expression de constructions arithm´etiques ou conditionnelles sur des types primitifs et des variables apparent´es aux langages imp´eratifs, ou *data-flow*.

Une *sémantique constructive* a ´et´e d´efinie pour un noyau d'instructions Esterel: Esterel Pur. Par la suite, sur la demande des utilisateurs, plusieurs instructions ont ´et´e rajout´ees, macros construites `a partir du noyau Esterel ou constructions orient´ees *données* sans s´emantique. Nous nous focalisons ici sur les instructions du noyau Esterel v5 — derni`ere version d´evelopp´ee `a l'INRIA —, les autres d´epassant le cadre de l'expos´e [8, 12].

1.1.1 Exemple

Un additionneur 2-bits est sp´ecifi´e en Esterel et comment´e (figure 1.2). Il est constitu´e d'additionneurs 1-bit (figure 1.1), de même sp´ecifi´e en Esterel et comment´e. En utilisant des *macros* Esterel, ces sp´ecifications auraient ´et´e plus concises et auraient plus ressembl´e `a un style de programmation imp´eratif traditionnel.

1.1.2 Syntaxe

Dix primitives syntaxiques sont d´efinies [8, 12].

La séquence
La s´equence est faite entre deux instructions p et q par l'op´erateur **;**: $p; q$. La premi`ere instruction $p$ est instantan´ement d´emarr´ee lorsque la s´equence d´ebute, et s'ex´ecute jusqu'`a sa terminaison ou la lev´ee d'une *exception*. Lorsque p termine, q d´ebute imm´ediatement et la s´equence se comporte de la même façon que q.

Le parallèle
L'instruction de parall´elisation || est *synchrone*. Les signaux sont partag´es entre tous les *threads* — ou *branches* — leur permettant de communiquer.

```
[p || q]; r
```

Les instructions p et q s'ex´ecutent en parall`ele et lorsqu'elles ont toutes les deux termin´ees alors le contrôle est en s´equence instantan´ement donn´e `a r.

Emission d'un signal
Une ´emission instantan´ee d'un signal s est r´ealis´ee par l'instruction **emit** s. L'´emission termine instantan´ement et donne le contrôle dans l'instant `a la prochaine instruction, lui permettant de s'ex´ecuter.

module add1bit:	$add1bit$ est le nom du composant: le *module*.
input i, j, cin;	L'interface d´eclare 3 entr´ees et 2 sorties
output s, cout;	pouvant prendre soit le *statut présent* si
	le signal est pr´esent, soit les *statut absent*
	si le signal n'est jamais *émis* au cours de
	l'*instant*. Le signal d'entr´ee implicite: **tick**
	une fois activ´e marque le d´ebut de l'instant.
loop	Le comportement du module se reproduit `a chaque *instant*.
	Une boucle marque explicitement la reproduction du
	comportement au cours des instants.
present i **then**	Si le signal i est *présent* alors
present j **else**	si le signal j est *présent* alors rien ne se produit
emit s	si le signal j est *absent* alors le signal s est ´emis.
end present	
end present	Toutes les instructions d'Esterel sont clôtur´ees de sorte
	`a lever toutes les ambigüit´es de pr´ec´edence.
||	L'op´erateur de parall´elisme d´eclare en *parallèle synchrone* les
	diverses instructions.
present j **then**	
present i **else**	Cette instruction est similaire `a la pr´ec´edente.
emit s	
end present	Ainsi, le calcul de la somme est repr´esent´e par deux instructions
end present	syntaxiquement concurrentes car elles peuvent affecter en même
	temps s. Cependant, par construction, une seule pourra ´emettre
	s dans l'instant.
||	
present i **then**	Les trois prochaines instructions concurrentes calculent la
present j **then**	retenue c_{out} en fonction des trois entr´ees.
emit cout	Ici, c_{out} pourra être ´emis par deux instructions. Doit-on
end present	prot´eger ou *résoudre* le signal c_{out}?
end present	Les hypoth`eses de *diffusion instantanée des signaux* et
||	du *durée nulle de l'instant* ont permis de donner un sens
present j **then**	`a l'´emission d'un tel signal dit *signal pur*. Si un tel
present cin **else**	signal est ´emis dans une instruction concurrente alors ce signal
emit cout	est *présent*. Si une autre instruction concurrente
end present	l'´emet aussi dans l'instant, alors se signal est toujours pr´esent
end present	sans aucune ambigüit´e. Traiter d'Esterel Pur permet de s'abstraire
||	des r´esolutions de signaux portant des variables (des valeurs).
present i **then**	
present cin **else**	
emit cout	
end present	
end present	
end loop	
end module	

TAB. 1.1 –. Additionneur 1-bit en Esterel

module add2bits:	Ce module mod´elise l'additionneur 2-bits.
input i, j;	Ces entr´ees repr´esentent le premier nombre $i + 2.j$.
input k, l;	Le second nombre est $k + 2.l$.
output s1, s2, cout;	La sortie repr´esente le nombre $s_1 + 2.s_2$ et c_{out} la retenue.
loop	
signal cout1, cin **in**	Un signal interne c_{out_1} est d´eclar´e de sorte `a ce que les diverses instructions concurrentes puissent communiquer entre elles via ce signal. Un second signal interne c_{in} est n´ecessaire pour positionner l'entr´ee c de l'additionneur 1-bit du calcul des bits les moins significatifs.
run add1bin[	Le module $add1bit$ sp´ecifi´e au-dessus est instanci´e ici.
signal i / i;	Ses arguments sont des signaux que nous devons lier aux signaux
signal k / j;	que nous utilisons dans le module courant.
signal cin / cin;	Nous remarquons que le signal c_{in} est toujours *absent*
signal cout1 / cout;	car il n'est jamais *émis* dans ce module.
signal s1 / s]	
||	En parall`ele, nous instancions le module $add1bit$ une seconde
run add1bin[	fois avec les signaux repr´esentant les bits de poids fort
signal j / i;	des 2 nombres.
signal l / j;	
signal cout1 / cin;	
signal cout / cout;	A chaque instant, $add2bits$ positionnera ses sorties c_{out},
signal s2 / s]	s_1 et s_2 `a ***pésent*** ou *absent* en fonction
end signal	du *statut* des entr´ees i, j, k et l.
end module	En repr´esentation logique, *absent* est le potentiel bas not´e 0, et ***pésent***, le potentiel haut not´e 1.

TAB. 1.2 –. Additionneur 2-bits en Esterel

La boucle

Une boucle est d´efinie par **loop p end loop** o`u le corps est $p$, une instruction Esterel. Le corps $p$ red´emarre instantan´ement `a sa terminaison, de mani`ere infinie. Le corps d'une boucle n'est pas autoris´e `a se terminer instantan´ement lorsqu'il est d´emarr´e. Cette condition est *statique*: il ne peut pas exister dans l'instant un chemin direct d'ex´ecution du d´ebut jusqu'`a la fin de p, le chemin ne pouvant pas être dynamique.

La fin de l'instant

Une fin d'instant simple est la terminaison du programme en un instant (sans boucle). Cependant un programme ne se limite pas `a être ex´ecut´e qu'une seule fois (besoin de boucle). Un autre aspect r´eside dans le repr´esentation du syst`eme: c'est un automate. Lorsque le programme termine de s'ex´ecuter dans un instant, il termine dans un ´etat. Lors du prochain instant, il reprendra son ex´ecution `a partir de l'´etat dans lequel il s'´etait arrêt´e. La primitive **pause** termine l'instant courant dans un ´etat et reprendra au prochain instant `a ce point d'arrêt: un ´etat est donc un ensemble de points d'arrêt actifs.

Les boucles ne peuvent pas terminer instantan´ement: il est impos´e de casser les cycles instantan´es `a l'aide d'une instruction **pause** dans le corps.

```
loop
   pause;
   emit S
end loop
```

Par exemple, la macro **sustain** ´emet `a chaque instant le signal $S$. Lorsque la boucle est activ´ee (outre le premier instant), $S$ est ´emis puis termine instantan´ement en donnant en s´equence le contrôle `a l'instruction suivante. **pause** fige l'´etat du corps de la boucle de sorte qu'au prochain instant — au prochain *tick* — il rende le contrôle l`a o`u il l'avait laiss´e. L'ex´ecution reprend en terminant le corps de la boucle, revient au d´ebut, ´emet S et sauvegarde de nouveau l'´etat courant du corps de la boucle. Ceci sera ind´efiniement r´ealis´e.

Au travers de l'instruction Esterel *sustain*, nous venons de montrer que **pause** sauvegardait l'´etat courant de l'ex´ecution d'une instruction. On voit ici apparaitre le lien qu'il existe entre un *registre* — logiciel en termes de variables ou mat´eriel en termes de flip-flop-D — et l'instruction **pause**. En effet, cette instruction est la seule mani`ere de repr´esenter un registre. Toutes les autres constructions syntaxiques comme **await** qui intuitivement sauvegarde un ´etat ou marque un point d'attente sont construites avec des primitives de base dont **pause**.

Le test de présence d'un signal

L'instruction **present S then p else q end present** une fois activ´ee donne instantan´ement le contrôle `a une de ses branches *p* ou *q* en fonction de l'´evaluation du statut du signal $S$ test´e. Une des branches *p* ou *q* peut être omise, mais au moins une d'entre elles doit être sp´ecifi´ee. La branche omise correspond `a l'instruction **nothing** qui rend intantan´ement le contrôle.

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

Si le signal S est pr´esent dans l'instant courant alors le signal O est ´emis et l'instruction **present** termine. Si S est absent alors la branche *else* r´esuite `a **nothing** obtient le contrôle —*réaction à l'absence* — et le rend de sorte `a ce que **present** termine.

Déclaration locale d'un signal

Les branches mises en concurrence s'´echangent des informations par le moyen de signaux. L'instruction **signal S in p end signal** permet de d´eclarer un signal local S de sorte `a ce que les branches puissent communiquer. Les instructions Esterel ´etant clôtur´ees, la notion de port´ee – *scope* – de S dans p existe:

```
signal S in              signal S in
   p                        r
||               ||     ||
   q                        s
end signal               end signal
```

Les instructions p et q communiquent via le signal S, diff´erent de celui servant `a faire communiquer r et s. Dans cette instruction, p et q ne peuvent pas communiquer avec r et s via le signal S.

Cette notion de *scope* de signaux a amen´e la notion de *r´eincarnation* des signaux dans les boucle. Des *instances fraiches* des signaux sont disponibles au d´ebut du corps de chaque boucles. Nous n'allons pas d´evelopper ces notions inutiles pour la compr´ehension du document dont les d´etails se trouvent en [12].

Préemption
Une exception d´efinit un point de sortie pour son corps:

```
trap T in
   p
end trap
```

Le *scope* de T s'applique `a son corps $p$. L'instruction **exit** *T* dans $p$ donne instantan´ement le contrôle `a l'instruction suivant la fin du *scope* de *T*. Une nouvelle d´eclaration d'un **trap** cache celle la plus ext´erieure.

Le *trap* permet de terminer une boucle via l'instruction *exit* lev´ee dans le corps de la boucle ou par une *branche* parall`ele.

La suspension **suspend p when S** dans l'instant d'une section de code p pr´eempt´ee par le signal S permet de d´efinir avec **trap** des macros de pr´eemption habituellement utilis´ees **abort p when S** et **weak abort p when S**.

1.2 Langage réactif orienté contrôle

La mod´elisation d'un probl`eme quelconque aboutit `a la mise en parall`ele de deux syst`emes. Pourquoi devrions-nous nous interdire cette solution naturelle au profit d'un style s´equentiel, fonctionnel ou axiomatique?

Cette section pr´esente les hypoth`eses fondamentales d'Esterel qui ont permis, entre autres, de donner `a un programme Esterel une repr´esentation orient´ee VHDL.

1.2.1 Notion d'instant

$\mathbb{R}$ est ´echantillon´e sur $\mathbb{N}$:

$$\forall n \in \mathbb{N},\ \forall t \in [\![n,\ n+1[\![,\ \forall c_{in}(t), i(t), j(t) \in \mathbb{B},\ [\![add(i(t),j(t),c_{in}(t))]\!] \ =\ [\![add(i(n),j(n),c_{in}(n))]\!]$$

Hypoth`ese synchrone: l'instant $n$ est une tranche de temps telle que les entr´ees acquises `a l'instant n restent inchang´ees jusqu'au prochain instant $n+1$, en assurant que la p´eriode d'´echantillonage soit suffisament grande pour que le calcul puisse terminer en rendant un r´esultat. Alors la dur´ee de calcul $\epsilon \in \mathbb{R}$ est r´eduite `a $\epsilon \in [\![n,\ n+1[\![$. Les activit´es d'entr´ees-sorties et de calcul sont altern´ees de mani`ere coh´erente: toutes les entr´ees sont prises en compte et trait´ees.

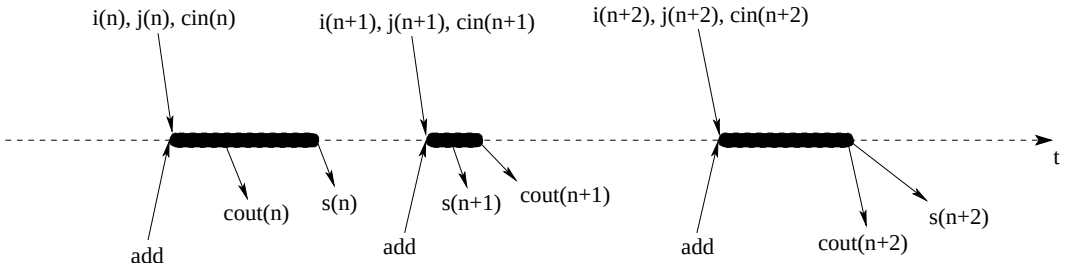


FIG. 1.2 –. Appel à la fonction add sur trois instants consécutifs

La figure 1.2 illustre la notion d'instant. Au cours du temps t, trois ´echantillons sont r´ealis´es sur les entr´ees i, j et c_{in} aux instants n, $n+1$ et $n+2$ initi´es par l'appel `a la fonction add. Les sorties c_{out} repr´esentant la retenue et s, la somme, sont calcul´ees au cours de l'instant. Contrairement aux langages avec horloge explicite, l'horloge est implicite — l'intervalle de temps n'est pas pr´ecis´e — avec r´eactions successives sur appel de add. La dur´ee d'un instant varie en fonction de plusieurs param`etres tels que la difficult´e `a calculer une instance d'entr´ees ou la charge de calcul du syst`eme qui peut ne pas simplement être limit´e au calcul de add.

L'expression du parall´elisme a ´et´e introduite grâce `a la notion d'*instant commun* qui assure que toutes les branches d´emarrent en même temps sur la pr´esence du signal marquant le d´ebut de l'instant (ici add).

Ce mod`ele correspond `a une r´ealit´e physique, en termes de circuits, mod´elisant les parties contrôles qui ne sont pas consid´er´ees par les langages g´en´eralistes de description de circuits.

Composant et interface

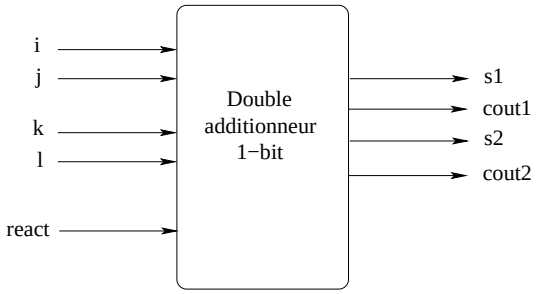


FIG. 1.3 –. Système réactif synchrone

Lorsque cette fonction de r´eaction sera appel´ee depuis l'ext´erieur — par un autre programme, par l'utilisateur, par un stimulus de l'environnement —, les entr´ees seront fix´ees pour toute la dur´ee de l'instant. Nous devons ainsi connaitre toutes les entr´ees du syst`eme donn´ees aux diverses fonctions de calcul de sorte `a pouvoir les figer d`es le d´ebut de l'instant. Le syst`eme est donc d´ecrit comme une bôte ayant des entr´ees et des sorties. L'appel `a la fonction de r´eaction provient de l'environnement ext´erieur, comme les entr´ees, nous pouvons donc consid´erer cet appel comme une nouvelle entr´ee d´eclanchant un instant de calcul. L'ensemble des entr´ees et des sorties est appel´e *interface* du syst`eme — ou du composant —.

La figure 1.3 pr´esente l'interface du double additionneur 1-bit pr´esent´e au-dessus. i, j, k et l sont les entr´ees du syst`eme. Les sorties sont c_{out_1}, c_{out_2}, s_1 et s_2. $react$ est l'entr´ee permettant de d´ebuter un nouveau cycle de calcul.

Système infini

L'appel `a une fonction locale (ici add) depuis une autre fonction n'est plus possible. La fonction de r´eaction marquant le d´ebut de l'instant, assure que tous les *threads* d´emarrent en même temps selon la boucle de r´eaction 1.1. Le d´ebut de l'instant est initi´e de façon g´en´erale par un stimulus externe au syst`eme (ici $react$). De plus, lorsqu'un instant est termin´e, nous d´esirons que le composant serve de nouveau pour de nouveaux cycles de calculs. Ainsi, g´en´eralement, un syst`eme synchrone attend le d´ebut d'un instant, lit les entr´ees, r´ealise les calculs et positionne les sorties puis boucle ind´efiniement dans cette s´equence d'actions. La figure 1.3 pr´esente en plus de son interface cette notion implicite de r´eutilisabilit´e infinie.

Signaux

L'instant commun a permis d'introduire un op´erateur de parall´elisme. Comment les diff´erentes *branches* communiquent-elles entre elles? Ils utilisent des variables partag´ees appel´ees *signaux*. En r`egle g´en´erale, les signaux ne peuvent être affect´es que dans une seule et unique branche par instant. Un m´ecanisme de *résolution* ou de *combinaison* des signaux doit être offert par le langage de sorte `a pouvoir les affecter dans plusieurs branches parrall`eles en même temps. De mani`ere g´en´erale, ces langages permettent d'exprimer fortement le parall´elisme entre divers calculs, ainsi, toutes les communications se feront par l'interm´ediaire de signaux. D´esormais, nous consid`ererons les entr´ees, les sorties et les communications internes au syst`eme comme des signaux.

Représentation logicielle

Un programme Esterel peut se synth´etiser soit en logiciel soit en mat´eriel.

Le double additionneur 1-bit 1.3 pr´esente une interface. Est-ce possible de repr´esenter le syst`eme interne sous forme logicielle? La boucle infinie peut être traduite en

```
forever do
```

```
   if react then
      read_inputs(i, j, k, l);
      compute();
      write_outputs(s, cout);
   end if
done
```

Sur la pr´esence de l'entr´ee $react$, les entr´ees de l'interface sont lues par $read\ inputs$, le corps du calcul, $compute$, est ensuite lanc´e, puis finalement les sorties sont positionn´ees par $write\ outputs$. Cette boucle est infiniement it´er´ee.

En supposant être sur une architecture mono-processeur, nous devons s´equentialiser l'expression du parall`ele `a l'int´erieur de la fonction $compute$:

```
compute() is
   add(0, i, j, cout1, s1);
   add(0, k, l, cout2, s2);
end compute
```

Ici, les appels `a $add$ sont parall`eles, les mettre en s´equence ne nuit en rien `a la coh´erence des r´esultats obtenus sur les sorties. Les deux instructions $add$ auraient pu être interverties sans affecter ni les r´esultats ni l'´ech´eance du calcul. Que ce passe-t-il dans le cas o`u les instructions soient concurrentes, ou bien que les entr´ees d'une instruction d´ependent des sorties d'une autre? Ces questions seront trait´ees un peu plus loin dans la section 2.3 relative au d´eterminisme de ces langages.

Représentation matérielle

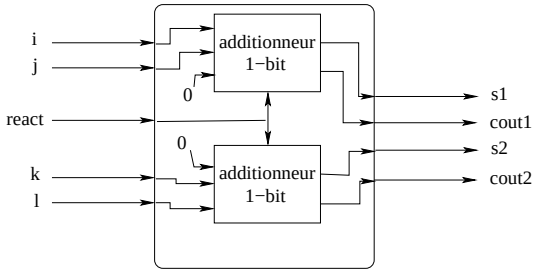


FIG. 1.4 –. Système réactif synchrone sur une architecture matérielle

Le double additionneur 1-bit 1.3 se traduit bien sur une architecture mat´erielle: le parall´elisme exprim´e dans un langage synchrone se retrouve dans sa repr´esentation mat´erielle (figure 1.4). Deux instances d'un additionneur 1-bit calculent simultan´ement les sorties en temps born´e. Nous remarquons que les composants sont sensibles `a l'entr´ee $react$: ce sont des composants *clockés* ou *composants digitaux synchrones*.

1.2.2 Durée nulle de l'instant

Les langages r´eactifs synchrones proposent par la notion d'instant de fractionner le temps continu en temps discret. Chaque tranche discr`ete est appel´ee instant ou cycle. A la fin de chaque instant, le syst`eme `a n´ecessairement termin´e les calculs en positionnant les sorties `a des valeurs stables. A cette notion, nous avons rajout´e celle d'*instant commun* qui nous permet d'introduire aux langages un op´erateur de parall´elisme. Chaque branche parall`ele est appel´ee *thread* — au sens le plus g´en´eral: entit´es de calcul ind´ependantes qui peuvent communiquer entre elles —, synchronis´ees par une entr´ee commune provenant de l'environnement ext´erieur. Dans une repr´esentation logicielle, nous parlerons d'*appel à la fonction de réaction* alors que dans une approche mat´erielle ou distribu´ee, nous emploierons le terme de *signal de synchronisation*, *horloge* ou *clock*.

Une sous-classe de langages r´eactifs synchrones comme Esterel [8] font l'hypoth`ese de la *durée nulle de l'instant*. Le temps est discr´etis´e et, lors de l'appel `a la fonction de r´eaction, les entr´ees sont fig´ees d'une part et les sorties instantan´ement positionn´ees aux valeurs rendues apr`es le calcul.

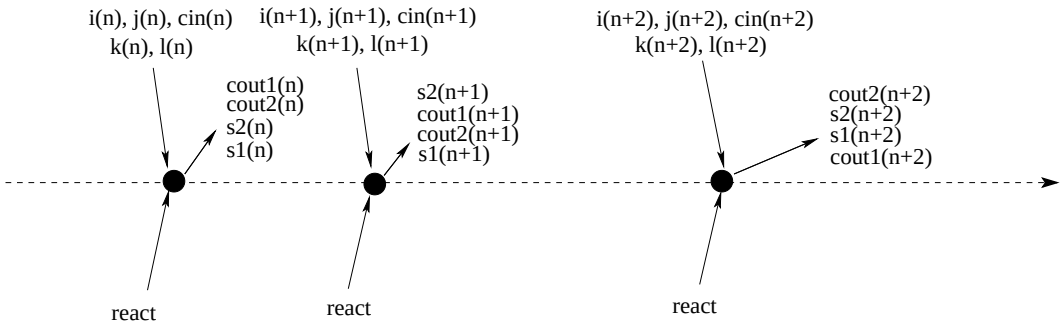


FIG. 1.5 –. Système réactif synchrone avec durée nulle de l'instant

La figure 1.5 pr´esente le paradigme synchrone avec la dur´ee nulle de l'instant. Cette hypoth`ese permet de s'abstraire de la dur´ee de calcul aboutissant `a donner une valeur aux sorties entrainant de multiples ouverture sur des syst`emes de preuves axiomatiques. Le programme ´evolue au cours du temps discret. Consid´erer la dur´ee de l'instant nulle permet de mettre en ´equations le programme: ´evaluation symbolique et s´emantique op´erationnelle en sont les principaux gains `a moindre coût. Lors d'une implantation logicielle ou mat´erielle, la dur´ee maximale de l'instant devra être d´etermin´ee de sorte `a ce que le calcul puisse terminer dans la dur´ee allou´ee `a l'instant.

Diffusion instantanée des signaux

La dur´ee nulle de l'instant permet de s'abstraire de la dur´ee du calcul en positionnant les sorties instantan´ement. Les *threads* ´emettent et recoivent les signaux instantan´ement. Un effet implicite de l'hypoth`ese de la dur´ee nulle de l'instant est la *diffusion instantanée des signaux*.

Réaction à l'absence

La division du temps entre instants permet de d´eterminer instantan´ement les signaux *présents*: les signaux d'entr´ee qui ont ´et´e positionn´es *émis* et les signaux interne de communication qui ont eux auusi ´et´e ´emis de sorte `a propager une information vers les *threads* qui attendent cette information. De mani`ere sym´etrique, les signaux qui ne sont pas *présents* sont dit *absents*. L'hypoth`ese faite par le langage Esterel est que les *threads* peuvent r´eagir `a l'absence d'un signal de la même mani`ere qu'ils le font `a la pr´esence.

1.3 Spécifier, simuler, prouver

1.3.1 Spécifier

Esterel permet de programmer des syst`emes en termes de contrôles dans le but de les v´erifier par la simulation et la v´erification de propri´et´es caract´eristiques exhib´ees. Un programme Esterel peut être vu comme un automate hi´erarchique ayant

1. des registres d'arrêt entre les r´eactions: ils permettent de m´emoriser en fonction de l'historique des entr´ees, l'´etat courant du cycle.
2. une fonction de transition sur les sorties: les entr´ees du cycle courant g´en`ereront des sorties, en fonction de l'´etat courant dans lequel l'automate se trouve.
3. une fonction de transition sur les ´etats: les entr´ees du cycle courant et l'´etat courant de l'automate le font passer dans un nouvel ´etat pour le cycle de simulation suivant.

Dans une ultime phase le syst`eme sp´ecifi´e et v´erifi´e pourra être compil´e en divers langages dont les plus repr´esentatifs sont:

– C ANSI pour être embarqu´e sur ASIC, ou utilis´e tel quel dans les domaines du logiciel (contrôles d'interfaces graphiques par exemple).

– C++ dans une optique logicielle ou g´enie logiciel.
– VHDL et Verilog pour être synth´etis´e en circuit.

Faiblesses de la tarduction en langage cible
Un programme Esterel est actuellement compil´e en syst`eme d'´equations tri´ees selon un tri topologique. La structure d'automate telle qu'elle a ´et´e pr´esent´ee plus haut engendrerait une explosion combinatoire sur le nombre des ´etats. Cette solution a eu son heure de gloire mais a dû être abandonn´ee. Le syst`eme d'´equations est ordonnanc´e – s´equentialis´e – afin que chaque variable soit d´efinie avant d'être utilis´ee (tout programme ne permettant pas ceci est incorrect). Ainsi, quelque soit le langage cible (C, C++, VHDL, ...), le syst`eme d'´equations est traduit de la même mani`ere (`a la syntaxe pr`es) en un syst`eme – ou circuit — combinatoire synchrone avec sauvegarde de l'´etat courant.

Une premi`ere faiblesse est que la structure du programme Esterel donn´ee par le concepteur a totalement disparue. Pourquoi arriver `a un niveau si bas[16] — proche des portes logiques ´el´ementaires en termes de circuits, ou d'instructions assembleurs de base en terme logiciel — en occultant la structure donn´ee au programme d'origine?

La seconde faiblesse qui d´ecoule de l'ordonnancement du syst`eme d'´equations est qu'`a tout instant toutes les ´equations sont ´evalu´ees. Ceci signifie qu'un nombre non n´egligeable d'´equations est vainement ´evalu´e. Conserver la structure initiale du programme Esterel d'origine permettrait d'activer uniquement les parties actives. En conservant un nombre de lignes produites apr`es compilation lin´eaire par rapport au nombre de lignes du code source Esterel: le temps de simulation et d'ex´ecution en serait largement am´elior´e.

Fort de ces constatations, un des buts est ici de proposer un système de compilation d'Esterel dans le langage VHDL en conservant la structure d'un programme Esterel initial afin de n'exécuter uniquement, ou presque, les parties réellement actives.

1.3.2 Simuler

Un syst`eme sp´ecifi´e en Esterel est ensuite ex´ecut´e de sorte `a en v´erifier la coh´erence comportementale. Plusieurs simulateurs sont disponibles, XES d´evelopp´e `a l'INRIA est de domaine public, le simulateur d'Esterel Studio, produit phare de la gamme d'outils d'Esterel Technologies, est un produit commencial.

La simulation est un point crucial car un passage `a l'´echelle des programmes Esterel implique un fort coût en ressources logicielles: m´emoire, CPU, ... Ainsi, des tentatives pour n'ex´ecuter que les parties utiles sont faites dans ce domaine depuis longtemps avec comme avanc´ees r´ecentes:

– Steven Edwards [2] (laboratoires Synopsys) propose un syst`eme purement logiciel de simulation des parties actives d'un programme.
– Le CNET (France Telecom R&D) a r´ealis´e un compilateur (SAXO [10]) produisant un code simulable 100% logiciel (FAST C) jouant sur le pouvoir d'expressivit´e d'un langage imp´eratif s´equentiel (C).
– Dumitru Potop [11] (INRIA-CMA) propose lui aussi un style d'´evaluation acc´el´erant la simulation en activant que les sections utiles.

Ces travaux sont des pr´eliminaires au travail r´ealis´e ici. Leur expressivit´e est ´etudi´ee dans le chapitre 3 consacr´e aux *formats internes*.

Programmes cycliques

Toutes les m´ethodes de simulation actuelles sont fond´ees sur une hypoth`ese forte des langages synchrones: *toute réaction termine dans un délai borné*. Sans cette hypoth`ese, les notions d'*instant, instant commun* et *durée nulle de l'instant* seraient fausses. Cependant, les constructions syntaxiques d'Esterel permettent de sp´ecifier un programme cyclique i.e. un programme qui ne termine pas dans l'instant. Le fait qu'un programme soit acyclique est une condition suffisante de correction.

Sémantique d'Esterel

Les compilateurs actuels v5 implantent Esterel et sa *sémantique formelle constructive* [12]. Les constructions Esterel sont associ´ees `a une repr´esentation logique, combinaison de portes et de registres. Ces constructions peuvent être combin´ees pour donner une repr´esentation logique ´equivalante: un programme Esterel est ´equivalant `a un circuit logique. Ainsi, compiler un

programme Esterel revient `a r´esoudre un syst`eme d'´equations logiques en appliquant les techniques et les outils d´edi´es au domaine de la logique.

- Un programme cyclique peut être constructif: production de code logique simulable.
- Les ´equations tri´ees sont une traduction de cet ensemble de portes et de registres logiques.
- La traduction en syst`eme d'´equations logiques s´equentialise (tri topologique) un programme Esterel: la structure initiale est perdue.

Un des buts est ici, de proposer une traduction d'Esterel en VHDL de sorte à conserver la sémantique constructive de la simulation du programme Esterel initial et de l'exprimer en VHDL, dans sa propre sémantique de simulation. Nous ´edudirons ici l'expression de programmes acycliques.

1.3.3 Prouver

La simulation d'un programme Esterel est fond´ee sur la s´emantique constructive du programme repr´esent´e comme un ensemble d'op´erations bool´eennes. Ainsi, des propri´et´es sur le programme peuvent être prouv´ees en appliquant des algorithmes prenant en entr´ee un syst`eme d'´equations bool´eennes. Les outils actuels XEVE (CMA-INRIA en domaine public) et les outils de preuve propos´es dans la Suite logicielle d'Esterel Studio sont bas´es sur les **B**inary **D**ecision **D**iagrams. Les outils produits par Esterel Technologies tendent n´eanmoins `a introduire de nouveaux moteurs de preuve bas´es sur SAT.

1.4 Synthèse

Esterel est un langage *réactif synchrone* structur´e fond´e sur plusieurs *hypothèses fortes*. Il poss`ede une *sémantique constructive* permettant plusieurs traitements *logiques*, *simplifications* et *preuves de propriétés* en utilisant des algorithmes propres au domaines de la *logique* ou du *hardware*. Il permet de *spécifier* les parties contrôle de syst`emes synchrones complexes, de les *simuler* puis de les compiler en C embarqu´e, en C++ pour du logiciel ou en VHDL/Verilog pour la synth`ese.

Esterel est compil´e en un syst`eme d'*équations logiques* permettant la s´equentialisation du code. Cependant, `a tout instant, toutes les ´equations du syst`emes sont ´evalu´ees. **Notre but est de conserver la structure exprimée par le programme Esterel acyclique initial de sorte à ce que seules les parties actives du programme soient simulées**.

Le syst`eme d'´equations logiques est directement traduit en VHDL, exprimant une s´equence d'op´erations sur des portes et des registres [16]. **Notre but est de traduire en VHDL un programme Esterel acyclique en conservant les comportements décrits**.

Esterel subit plusieurs ajouts de constructions syntaxiques de sorte `a accroitre son pouvoir d'expressivit´e. Ces nouvelles instructions se construisent `a partir d'un *noyau de dix primitives de base* ayant une repr´esentation en *sémantique constructive*. Nous ne traiterons donc ici que de ce noyau: *Esterel Pur v5*.

Chapitre 2

VHDL

***V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage* (VHDL) [13] est un langage de description de mat´eriel `a divers niveaux qui pourra ensuite être synth´etis´e en portes logiques et registres (***R**egister **T**ransfer **L**evel*) ou transistors. VHDL s'inscrit au même titre que son concurrent Verilog dans les langages de CAO de circuits permettant la simulation du syst`eme d´ecrit avant synth`ese. L'int´eret port´e `a ce langage r´eside dans le fait que de nombreux logiciels de partitionnement logiciel/mat´eriel, d'optimisation, ... le prennent en entr´ee.

2.1 Langage de description

Le d´eveloppement de langage a ´et´e entrepris en 1981 par le D´epartement de la d´efense des USA impliquant largement les industriels dans le processus de standardisation IEEE aboutissant en 1987. Plusieurs groupes de travail sur la synth`ese de haut niveau, la simulation digital/analogique, la v´erification formelle, ... sont cr´e´es. L'IEEE apporte une r´evision en 1993 avec VHDL93 (oppos´e `a VHDL87). Ces groupes continuent de nos jours `a travailler dans un grand nombre de domaines donnant lieu `a de nouvelles r´evisions.

2.1.1 Styles de programmation

VHDL offre trois styles de programmation: niveau algorithmique comportemental, niveau structurel et niveau flot de donn´ees. C'est un langage tr`es verbeux permettant un grand pouvoir d'expressivit´e. Fortement typ´e il permet de manipuler des tableaux, des entiers, ... et pas que des bits.

Niveau algorithmique

C'est le niveau plus proche de la mani`ere de programmer des informaticiens. Les instructions sont s´equentielles avec l'expression de la condition, du choix, l'appel `a des fonctions et proc´edures, variables, ... Les pointeurs, effets de bord et flux d'entr´ees/sorties permettent de programmer dans un style imp´eratif ADA ayant largement inspir´e le langage. L'additionneur 1-bit est sp´ecifi´e par

```
if i = '1' xor j = '1' xor cin = '1' then s <= '1' else s <= '0' end if;
if i = '1' and j = '1' then cout <= '1' else
   if i = '1' and cin = '1' then cout <= '1' else
      if j = '1' and cin = '1' then cout <= '1' else
         cout <= '0';
      end if;
   end if;
end if;
```

Il est facile de reconnaitre ici un style de programmation imp´erative s´equentielle. Notons cependant le symbole d'affectation <= qui permet de positionner le signal VHDL typ´e en partie gauche `a une valeur par l'´evaluation d'une expression en partie droite.

Niveau flot de données

Ce niveau est proche de celui exprim´e par les langages synchrones *data-flow*. Tous les signaux apparaissant en partie gauche des affectations <= sont modifi´es en prenant les valeurs calcul´ees en partie droite. A tout instant $t \in \mathbb{R}$, l'additionneur 1-bit sp´ecifi´e en ´equations flot de donn´ees, produit des valeurs sur ses sorties s et c_{out}

```
s    <= i xor j xor cin;
cout <= (i and j) or (i and cin) or (j and cin);
```

Niveau structurel

Ce niveau s'adresse particuli`erement aux ´electroniciens qui prennent VHDL comme langage de sp´ecification de circuits. Il permet d'interconnecter des composants fournis par des librairies ou ´ecrits par le concepteur, de sorte `a construire un syst`eme avec la vision mat´erielle. Les instances des composants ´evoluent implicitement en parall`ele. L'additionneur 1-bit est sp´ecifi´e par

```
Xor(i, j, sig0);
Xor(cin, s);
And(i, j, sig1);
And(i, cin, sig2);
And(j, cin, sig3);
Or(sig1, sig2, sig4);
Or(sig3, sig4, cout);
```

On reconnait facilement ici une description mat´erielle en termes de circuits logiques illustr´ee par la figure 2.1.

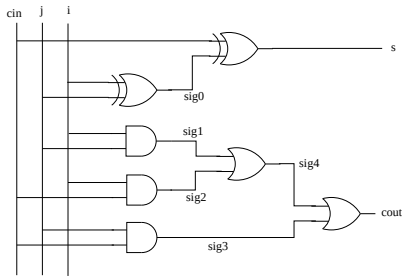


FIG. 2.1 –. Additionneur 1-bit en circuit

2.1.2 Modularité et hiérarchie

VHDL est modulaire: il permet de sp´ecifier l'interface d'un composant en pr´ecisant les *ports* d'entr´ees et de sorties. Un syst`eme peut être vue comme un bôte noire, une *entité*, connect´ee `a d'autres entit´es via les ports. La mani`ere de calculer les sorties en fonction des entr´ees d´epend du style de programmation utilis´e par le programmeur. A une entit´e peut être associ´e un nombre quelconque d'*architectures* qui traduit l'int´erieur de la bôte noire, ´ecrites dans diff´erents styles: *architecture comportementale*, *architecture structurelle* ou *architecture flot de données*. Les diff´erents styles peuvent être m´elang´es: nous utiliserons par exemple pour repr´esenter un syst`eme combinatoire p´eriodique avec registres VHDL un niveau flot de donn´ees pour la partie combinatoire, et un niveau comportemental pour l'´evaluation p´eriodique des registres.

L'additionneur 1-bits $add1bit$ d´eclare ses ports:

```
entity add1bit is
   port ( i, j    : in bit;
          cin     : in bit;
          s, cout : out bit
        );
end entity add1bit;
```

L'additionneur 2-bits utilise deux instances de ce composant. Une architecture structurelle — ´ecrite avec le niveau de programmation structurel — exploite pleinement ce type de composants mat´eriels. Une architecture flot de donn´ees — ´ecrite avec le niveau de programmation flot de donn´ees — d´ecrit un ensemble de signaux (en partie gauche) constamment modifi´es par les ´equations de la partie droite: ce composant ne peut pas s'utiliser dans une telle architecture. Finalement, l'architecture comportementale ne peut pas non plus exploiter ce composant. Elle permet une programmation algorithmique de plus haut niveau.

Architecture comportementale

Elle est compos´ee d'une collection de *processus* parall`eles communiquant entre eux via des signaux. Chaque *processus* est ´ecrit avec un niveau de programmation comportementale. Les processus sont le moyen d'exprimer l'activation d'un comportement sur un signal. Il apparait dans la *liste de sensitivité* du processus, d´evelopp´e plus loin. L'utilisation de cette liste est fortement li´ee `a la *sémantique de simulation* du moteur de simulation VHDL. L'additionneur 2-bits s'´ecrirait en comportemental synchrone de la mani`ere d´ecrite dans le tableau 2.1. Nous porterons une attention toute particul`ere `a la construction syntaxique des processus LSB et MSB.

```
entity add2bits is
  port (
    i, j      : in bit;
    k, l      : in bit;
    s1, s2, cout: out bit;
    clock     : in bit
  );
end entity add2bits;
architecture behav of add2bits is
  signal cout1 : bit := bit'low;

begin
LSB: process(clock) is begin
  if clock = '1' then
    s1 <= i xor k;
    cout1 <= i and k;
  end if
end process

MSB: process(clock) is begin
  if clock = '1' then
    s2 <= j xor l;
    cout <= (j and k) or (j and
          cout1) or (j and cout1);
  end if
end process
end architecture behav;
```

Le composant $add2bits$ d´eclare dans son interface en entr´ees les deux bits du premier nombre $a = i + 2.j$ et les deux bits du second $b = k + 2.l$. Apr`es calcul, les sorties sont s_1 (bit de poids faible), s_2 (bit de poids fort) et c_{out} (la retenue). Le signal d'entr´ee $clock$ permet de d´ebuter un cycle.

Nous associons `a l'entit´e add2bits une architecture $behav$ d´ecrivant l'additionneur. Un signal interne $c_{out_1}$ est d´eclar´e de sorte `a propager la retenue obtenue apr`es addition des bits les moins significatifs. Ce signal est de type $bit$ et initialis´e `a la valeur la plus basse du type ´enum´er´e bit soit '0'.

Le premier processus nomm´e LSB calcule s_1 et c_{out_1}. Il est *sensible* au signal $clock$ et ne commencera `a s'ex´ecuter qu'`a son arriv´ee. La *présence* d'un signal ne d´epend pas de sa valeur mais du changement de sa valeur. Ce changement implique un *évènement* sur le signal qui avertit le moteur de simulation que le processus doit être r´e´evalu´e.

Le premier processus termine syntaxiquement et la d´eclaration du second suit. Un parall`ele implicite d´eclare ces deux processus en concurrence.

Le second processus MSB est lui aussi synchronis´e sur le signal $react$: il d´emarre au même moment que le premier sur l'arriv´ee de $react$.
Les deux instructions d'affectation des signaux s_2 et c_{out} sont r´ealis´ees en s´equence, sur le front montant du signal $clock$, `a l'int´erieur du corps du processus d´clar´e entre *begin–end*.

TAB. 2.1 –. $add2bits$ en VHDL comportemental

Nous remarquons dans l'exemple 2.1 que le r´esultat de l'´evaluation du processus MSB d´epend de celui de LSB. Un lien de *causalité* existe entre les deux processus: le signal c_{out_1} est calcul´e dans LSB et utilis´e dans MSB pour positionner c_{out}.

2.2 Moteur de simulation

Le moteur de simulation orient´e*évènements* est bas´e sur une s´emantique de simulation [13] permettant d'aboutir `a des logiciels de synth`ese. Le moteur de simulation VHDL refl`ete des constructions syntaxiques du langage. Il permet de simuler des programmes tr`es complexes ayant un tr`es fort pouvoir d'expressivit´e: pointeurs, boucles, flux d'entr´ees/sorties, ... Un grand nombre d'´el´ements du langage ne peut pas s'exprimer en termes de portes: une distinction doit être faite entre *tout VHDL* et *VHDL synthétisable*[14], sous-ensemble restreint de *tout VHDL* accept´e par les outils de synth`ese industriels.

Le moteur VHDL est dirig´e par les ´ev`enements. A chaque signal est associ´e un *pilote* ou *driver* qui m´emorise les valeurs du signal au cours du temps continu. Un algorithme permet ensuite de calculer et de manipuler ces valeurs.

2.2.1 Temps continu et delais

Delai physique

VHDL permet de pr´eciser des d´elais temporels simulant le temps de travers´ee des portes, ou le temps de calcul d'un composant.

```
s2 <= j xor l after 10 ns;
```

Cette instruction affectera au signal s_2 le r´esultat du calcul de $j\ xor\ l$ apr`es un delai de 10 ns apr`es le temps physique courant de simulation. Si un *évènement* doit se produire sur le signal s_2 — changement effectif de valeur entre sa valeur au temps de simulation courant et celle apr`es ´evaluation de l'expression de droite — alors il se produira apr`es un delai de 10 ns apr`es le temps physique courant de simulation. Cette sp´ecification non synth´etisable permet de simuler le temps de travers´ee de la porte $xor$ d'une librairie de composants particuli`ere.

Delai implicite ou logique

Le moteur de simulation de VHDL ´evalue les expressions au temps courant de simulation avec les valeurs des signaux `a ce temps. Les nouvelles valeurs des signaux seront positionn´ees dans les *pilotes* `a la date courante plus un delai fourni en unit´e de temps. Cependant, l'exemple 2.1 ne pr´ecise aucun d´elai donc comment la simulation respecte-elle la causalit´e qu'il existe entre LSB et MSB via le signal c_{out_1} ?

Un delai implicite est introduit: le δ-*delai*. Il symbolise un pas de calcul au temps courant de simulation. Pour ´evaluer la sp´ecification comportementale 2.1 au temps $t \in \mathbb{R}$, nous aurons besoin d'un δ-delai pour que c_{out_1} soit calcul´e et d'un second pour calculer c_{out}.

2.2.2 Les pilotes

Chaque signal a son pilote ou *driver* que l'on peut repr´esenter comme un tableau `a deux lignes. La premi`ere positionne les temps de simulation et la seconde la valeur du signal pris `a cette date. La simulation d´ebute en initialisant les pilotes au temps 0 avec les valeurs d'initialisation d´etermin´ees par l'utilisateur ou par d´efaut le premier ´el´ement du type — ´enum´er´e — du signal. Tous les processus sont ex´ecut´es une fois et se *suspendent* sur la premi`ere *primitive de suspension*. Si le signal a ´et´e affect´e alors une nouvelle entr´ee dans son pilote est cr´e´ee pr´ecisant la date `a laquelle le signal sera modifi´e et la valeur qu'il prendra.

Au temps de simulation t qui correspond `a la date d'au moins un pilote, un ´ev`enement se produit sur le signal qui prend la valeur associ´ee `a ce temps. Apr`es un calcul requ´erant ce signal, les nouvelles valeurs des signaux sont *projet*´*es* dans leur pilote en ajoutant une nouvelle entr´ee pr´ecisant la date de modification et la valeur associ´ee. Cette entr´ee dans le futur est appell´ee *futur projet* du signal. Pour l'additionneur $add2bits$ sp´ecifi´e en 2.1 nous obtenons en fin de simulation les pilotes 2.2 associ´es au sc´enario VHDL ou *test-bench* suivant[1]

```
entity testbench is
end entity testbench;
```

1. Architecture mixte: structurelle et fbt de donn´ees.

```
architecture dataflow of testbench is
   -- Declaration de l'interface du composant (entite) add2bits
   component add2bits port (
      i_c, j_c, k_c, l_c  : in bit;
      s1_c, s2_c, cout_c  : out bit;
      clock_c             : in bit);
   end component;

   -- Association de l'architecture behav au composant add2bits
   for Add:add2bits use entity Work.add2bits(behav) port map (
      i => i_c; j => j_c; k => k_c; l => l_c;
      s1 => s1_c; s2 => s2_c; cout => cout_c;
      clock => clock_c);

   -- Declaration de signaux locaux
   signal i, j, k, l, s1, s2, cout, clock : bit;

begin
   Add: add2bits port map (
      clock_c => clock;
      i_c => i; j_c => j; k_c => k; l_c => l;
      s1_c => s1; s2_c => s2; cout_c => cout);

   i     <= '1' after 10 ns, '0' after 35 ns;
   j     <= '1' after 10 ns, '0' after 35 ns;
   k     <= '1' after 12 ns;
   l     <= '1' after 12 ns;
   clock <= '1' after 28 ns;
end architecture dataflow;
```

Les valeurs affect´ees aux signaux sont *persistantes*: elles sont modifi´ees par l'environnement ext´erieur. Dans les pilotes des signaux 2.2, "$-$" au temps t repr´esente la valeur inchang´ee des signaux positionn´es dans le pass´es. Au temps $28ns$, le signal $clock$ initie l'´evaluation des processus et calcule les signaux de sorties en fonctions des valeurs des entr´ees se trouvant dans les pilotes. Ces ´evaluations ne sont pas stables car c_{out_1} a ´et´e modifi´e: MSB doit être recalcul´e. Aucun temps physique n'est pr´ecis´e, nous ajoutons donc des δ-delais. Au bout de 2δ-delais au temps $28ns$, tous les signaux sont stables: les valeurs affect´ees aux signaux sont coh´erentes. Par exemple, le signal s_2 a oscill´e au même temps physique entre '0' et '1', se stabilisant `a '1'.

temps	0	10 ns	12 ns	28 ns	28 ns $+\delta$	28 ns $+2\delta$	35 ns
i	'0'	'1'	–	–	–	–	'0'
j	'0'	'1'	–	–	–	–	'0'
k	'0'	–	'1'	–	–	–	–
l	'0'	–	'1'	–	–	–	–
c_{out}	'0'	–	–	–	'1'	'1'	–
s_1	'0'	–	–	–	'0'	–	–
s_2	'0'	–	–	–	'0'	'1'	–
c_{out_1}	'0'	–	–	–	'1'	–	–
$react$	'0'	–	–	'1'	–	–	–

TAB. 2.2 –. Pilotes des signaux de $add2bits$

Pilotes multi-sources

Le tableau 2.2 montre le comportement logique de l'`etat des fils au cours du temps physique et logique. Chaque signal est affect´e dans un seul processus. Cette section traite de la mani`ere dont un même signal peut-être affect´e dans plusieurs processus: un tel signal multi-sources est appel´e *signal résolu*.

Consid´erons le cas d'´ecole suivant o`u *S*, *A* et *B* sont des signaux de type *bit*.

```
P_1 : process (A) is
```

```
begin
   S <= A;
end process P_1;

P_2 : process (B) is
begin
   S <= B;
end process P_2;

process is
begin
   A <= '1' after 10 ns;
   B <= '1' after 12 ns;
end process;
```

Les pilotes des ces signaux sont:

temps	0	10 ns	10 ns + δ	12 ns	12 ns + δ
A	'0'	'1'	–	–	–
B	'0'	–	–	'1'	–
$P_1 :: S$	'0'	–	'1'	–	–
$P_2 :: S$	'0'	–	–	–	'1'
S	'0'	–	'0'	–	'1'

TAB. 2.3 –. Pilotes

Le tableau 2.3 d´ecrit les pilotes des signaux. Nous remarquons que trois pilotes sont associ´es au signal *S*:

- le pilote $P_1 :: S$ associ´e `a la source du signal *S* du processus P_1
- le pilote $P_2 :: S$ associ´e `a la source du signal *S* du processus P_2
- le pilote du signal *S* calcul´e en fonction des valeurs contenus dans $P_1 :: S$ et $P_2 :: S$ et d'une *fonction de résolution*

Le signal r´esolu *S* est d´sclar´e de la mani`ere suivante:

```
function ResolveS ( param : in bit_vector ) return bit is
   variable back  : bit := '1';
   variable index : natural := 0;
begin
   for index in param'range loop
      back := aux and param(index);
   end loop;

   return back;
end ResolveS;

signal S : ResolveS bit := 0;
```

Au temps $10ns + \delta$, la valeur du signal S est le and entre $'1'$ provenant de la source $P_1 :: S$ et $'0'$ de $P_2 :: S$ i.e. $'0'$. Au temps $12ns + \delta$, la valeur de la source $P_1 :: S$ est inchang´ee et vaut toujours $'1'$, alors que la source $P_2 :: S$ a ´et´e modifi´ee valant actuellement elle aussi $'1'$. Le signal S porte `a cette date la valeur $'1'$.

2.2.3 Attente d'évènements

Plusieurs primitives implicites ou explicites permettent l'attente de signaux dans les processus. Les signaux des listes de sensitivit´es sont une attente implicite des signaux pr´ecis´es. Les processus ex´ecuteront leur corps lorsque un ´ev`enement sur un signal de leur liste de sensitivit´e se sera produit dans le pass´e et inscrit dans le pilote au temps courant de simulation.

Une primitive explicite **wait** pr´esente dans le corps du processus permet de suspendre l'ex´ecution du processus jusqu'`a ce que le signal attendu soit pr´esent.

```
wait until clk'event and clk = '1';
```

Cette instruction "consacr´ee" repr´esente un processus synchrone sur le front montant du signal clk: il attend qu'un ´ev`enement se produise sur clk et que la valeur port´ee soit '1'.

2.2.4 Algorithme d'évaluation

Les pilotes repr´esentent une ´evaluation d'un programme VHDL pour une instance de suites d'entr´ees indic´ees sur le temps continu. Comment construire ces pilotes et d´ecider en l'occurence de la stabilit´e des signaux? L'algorithme de simulation est le suivant:

ex´ecuter une fois le corps des processus et les suspendre sur les primitives d'attente d'´ev`enements
Pour toujours, faire:
  Calculer le nouveau temps comme ´etant le plus petit temps suivant le temps courant de simulation pr´esent dans les pilotes
  Mettre `a jour les valeurs des signaux
  R´eactiver tous les processus suspendus sensibles `a ces signaux v´erifiant la condition de r´eveil
  Calculer les valeurs des signaux affect´es par le r´eveil des processus:
    Ajouter une nouvelle ent´ree dans les pilotes des signaux affect´es
    Pr´eciser le temps auquel les ´ev`enements sur ces signaux seront pris en compte (temps physique ou avec δ-delais)
      et la valeur qu'ils porteront
  Fin du calcul des valeurs des futurs projett´es des signaux
Fin de la boucle de simulation infinie

2.3 Esterel vs VHDL

L'ex´ecution d'un programme Esterel est dirig´ee par les r´eactions marquant les cycles d'´evaluation – lectures des entr´ees, traitement et positionnement des sorties – des instants. Le programme est constitu´e d'instructions mises indiff´eremment en s´equence ou en parall`ele. L'instant est commun et de dur´ee nulle `a toutes les branches du parall`ele qui communiquent entre elles via des signaux instantan´emant diffus´es. La r´eaction `a l'absence permet `a un signal absent dans l'instant de donner le contrôle `a du code. Le programme est compil´e en syst`eme d'´equations ordonnanc´ees par un tri topologique de sorte `a toujours utiliser des variables affect´ees: respect de la causalit´e. A la fin de chaque instant, le programme se trouve dans un ´etat d´ependant de l'historique des entr´ees.

La simulation d'un programme VHDL est dirig´ee par les ´ev`enements. Une collection de processus parall`eles constitu´es d'instructions s´equentielles communiquent entre eux via des signaux avec un δ-delai de latence: les δ-delais permettent de simuler la causalit´e entre l'´emission d'un signal et sa r´eception. Les processus sont activ´es par une condition de r´eveil sur des signaux et se suspendent sur une instruction *wait*.

Traduire un programme Esterel en VHDL Comportemental demande l'analyse des points suivants:

1. Un même probl`eme sp´ecifi´e en Esterel et VHDL est-il d´eterministe, en fonction de leur s´emantiques de simulation respectives?
2. Comment simuler la r´eaction Esterel – et donc l'instant commun de dur´ee nulle – en VHDL orient´e ´ev`enements?
3. Comment traduire les instructions indiff´eremment mises en s´equence ou en parall`ele en un ensemble de processus VHDL parall`eles constitu´es d'instructions s´equentielles?
4. Comment r´eagir `a l'absence d'un signal dans l'instant?
5. Comment repr´esenter les signaux ´eph´em`eres Esterel diff´erents des signaux persistants VHDL?
6. Comment traiter les registres d'´etat du programme Esterel diff´erents des registres VHDL?
7. Comment diffuser instantan´ement l'´emission d'un signal?
8. Comment conserver la causalit´e du programme Esterel initial?
9. Comment g´erer les probl`emes de schizophr´enie ou de r´eincarnation li´es `a Esterel?
10. En g´en´eral, comment traduire les instructions Esterel en VHDL?

Déterminisme

Esterel comme VHDL sont des langages dont les mod`eles d'ex´ecution sont d´eterministes. Nous allons montrer ici de mani`ere intuitive que l'*évaluation séquentielle* et l'*évaluation parallèle* d' une même sp´ecification donne toujours les mêmes

sorties en la comparant avec l'*évaluation symbolique* des r´esultats attendus. Ces mod`eles d'´evaluation sont pr´esent´es dans [9].

L'exemple du double additionneur 2-bits, simple et concis, d´ecrit deux additionneurs 1-bit mis en parall`ele. La repr´esentation syntaxique que nous lui donnons est une interpr´etation de sa sp´ecification Esterel 1.2 et VHDL 2.1.

```
add2bits(i, j, k, l) = (s1, s2, cout) is
   add(0, i, j, cout1, s1)
||
   add(cout1, k, l, cout, s2)
end
```

Le signal de synchronisation qui apparaissait jusqu'`a pr´esent de façon explicite est d´esormais implicite dans cette repr´esentation. L'additionneur 2-bits est $add2bits$, d´eclar´e avec son interface: $i$, $j$, $k$ et $l$ sont les bits d'entr´ees et $s_1$, $s_2$ et $c_{out}$ ceux de sorties. Le corps du programme est compos´e de deux appels parall`eles `a la fonction $add$ d´efinie par ailleur. Le premier appel (au sens de l'´ecriture dans le corps) `a add calcule s_1 comme la somme de trois bits i, j et $c_{in} = 0$, et c_{out_1} comme le calcul de la retenue obtenue par i, j et $c_{in} = 0$. Nous remarquons que c_{out_1} est un signal interne implicite car il n'est pas d´eclar´e dans l'interface de $add2bits$. Sa valeur servira `a l'´evaluation de $c_{out}$ et $s_2$ r´esultant du second appel (toujours au sens syntaxique) `a add, prenant en param`etres c_{out_1}, k et l.

Evaluation parallèle

L'´evaluation parall`ele — assimilable `a celle op´er´ee par le moteur de simulation VHDL — est vue comme un syst`eme de composants ´evoluant en parall`ele tel que le proposerait un circuit ´electronique. Le tableau suivant pr´esente les diff´erents potentiels des signaux jusqu'`a l'atteinte de leur stabilit´e. La simulation suivante est r´ealis´ee pour une instance d'entr´ees toutes `a 1. Le symbole * repr´esente une valeur arbitraire 0 ou 1. Lors de l'activation du syst`eme par un signal implicite de synchronisation, repr´esent´e par $t = 0$, les entr´ees sont fig´ees aux valeurs donn´ees (toutes `a 1). Les signaux internes et de sorties ont une valeur arbitraire. Par la suite, le passage d'une ligne `a une autre est symbolis´e par un laps de temps $\epsilon$ permettant d'obtenir le calcul de signaux. Au bout de $2\epsilon$, le syst`eme est stable et donne comme r´esultat sur les sorties $s_1 = 0$, $s_2 = 1$, $c_{out} = 1$. R´ealiser la même simulation pour les 15 autres s´equences d'entr´ees possibles permettra de trouver toutes les s´equences de sorties possibles.

t	i	j	k	l	s_1	s_2	c_{out_1}	c_{out}
0	1	1	1	1	*	*	*	*
ϵ	1	1	1	1	0	*	1	*
2ϵ	1	1	1	1	0	1	1	1
3ϵ	1	1	1	1	0	1	1	1
4ϵ	1	1	1	1	0	1	1	1
5ϵ	1	1	1	1	0	1	1	1

Evaluation séquentielle

L'´evaluation s´equentielle repr´esente la mani`ere dont un programme exprimant le parall´elisme est simul´e en s´equence, assimilable au syst`eme d'´equations tri´ees d'Esterel. Le but est de commencer par d´eterminer un ordre topologique sur l'´evaluation des branches de sorte `a ´evaluer l'expression qu'une seule fois. Ici, l'´evaluation de la seconde instruction $add$ d´epend de $c_{out_1}$, calcul´ee par la premi`ere instruction add. Nous allons donc commencer par initialiser tous les signaux `a 0 puis ´evaluer la premi`ere instruction add et ensuite la seconde. A $t = 3$, nous obtenons les mêmes sorties que pour la simulation parall`ele. Bien entendu, l'´evaluation doit être op´er´ee pour touts les autres tuples.

t	i	j	k	l	s_1	s_2	c_{out_1}	c_{out}
0	1	1	1	1	0	0	0	0
1	1	1	1	1	0	0	1	0
3	1	1	1	1	0	1	1	1

Evaluation symbolique

L'´evaluation symbolique repr´esente le syst`eme d'un point de vue math´ematique et propose une solution unique. Posons:

- $I_1(n)$ la premi`ere entr´ee au temps $n$ o`u $I_1(n) = i(n) + 2k(n)$
- $I_2(n)$ la seconde entr´ee au temps n o`u $I_2(n) = j(n) + 2l(n)$
- $S(n)$ la somme de $I_1(n)$ et $I_2(n)$ apr`es calcul au temps n

Pour la s´equence d'entr´ees -tout `a 1-, $S(n) = 1 + 2.1 + 1 + 2.1 = 6 = 1_2 10$ le r´esultat binaire est $s_1 = 0$, $s_2 = 1$ et $c_{out} = 1$. En appliquant cette formule aux autres tuples, nous obtenons les mêmes r´esultats que pour les deux autres styles d'´evaluation.

Ces trois mod`eles d'´evaluation [9] permettent de montrer qu'un programme synchrone est d´eterministe i.e. pour les mêmes s´equences d'entr´ees, les sorties seront toujours identiques, quelque soit le mod`ele de simulation.

2.4 Synthèse

VHDL est un langage de *description de matériel* dont le *moteur de simulation* permet un grand pouvoir d'expressivit´e. Il permet de sp´ecifier un syst`eme selon trois styles d'*architectures*: *structurelle*, *flot de données* et *comportementale*. Cette troisi`eme architecture se base sur un niveau de programmation algorithmique permettant l'utilisation de constructions de contrôle dans un style imp´eratif s´equentiel. En Esterel, nous pouvions indif´eremment combiner des instructions parall`eles et s´equentielles alors que VHDL d´ecrit une collection de *processus parallèles* qui communiquent entre eux via des *signaux* mod´elise le syst`eme.

Un sous-ensemble de VHDL permet la *synthèse* de processus *synchrones*, activ´es p´eriodiquement par un signal commun. Les δ-delais permettent de simuler la *causalité* qu'il existe entre deux processus au moyen d'un signal commun. Les attentes sur ces signaux sont implicites (*liste de sensitivité*) ou explicites (instruction *wait* du langage).

Nous nous sommes assur´es en premier lieu que la simulation d'un même programme sp´ecifi´e d'une part en Esterel et d'autre part en VHDL est *déterministe*.

Chapitre 3

Format interne

Utiliser une forme interm´ediaire d'Esterel permet de s'abstraire de plusieurs probl`emes de compilation li´es `a l'interpr´etation des constructions syntaxiques en fonction de leur s´emantique constructive. Par exemple, la *r´eincarnation* de signaux dans les boucles ou la *synchronisation des branches parallèles* pourront être pr´ecompil´ees en une repr´esentation interne fournie par un format interm´ediaire.

Cette section a pour but de montrer les diff´erentes mani`eres de transformer un programme Esterel en un CCFG ( **C**oncurrent **C**ontrol-**F**low **G**raph ) qui pourra entrer facilement dans un processus de compilation s´equentielle traditionnelle. Ces techniques sont en effet d´ej`a largement utilis´ees par les compilateurs Esterel existants (EC de Synopsys, SAXO du CNET et v5 de l'INRIA).

Nous traiterons ainsi en premier lieu du format IC/LC actuellement utilis´e dans la version V5 du compilateur Esterel INRIA/Esterel Technologies[1, 15]. Ce format est directement obtenu `a partir du code source Esterel.

En second lieu, nous pr´esenterons un format CCFG[2] hi´erarchique aidant `a la visualisation de la structure de contrôle. Il conserve la structure d'un graphe de contrôle IC/LC mais rend sa potentielle synth`ese en un code s´equentiel plus facile. En effet, le code source n'augmentera pas exponentiellement en synth´etisant une net-list mais conservera une taille proche de celle du source IC/LC. Il est en l'occurence utilis´e comme format interm´ediaire par le compilateur propri´etaire EC[3] de Synopsys.

Nous terminerons par l'analyse du format GRC[11] d´evelopp´e `a l'INRIA prenant en compte, toujours `a un niveau structurel, certains probl`emes de duplication de code inh´erents `a la s´emantique d'Esterel. Ces portions de code peuvent potentiellement être ex´ecut´ees en plusieurs instances dans diff´erents contextes lors d'une même r´eaction.

Ce document `a pour but de pr´esenter les diff´erents CCFG propos´es pour un mod`ele Esterel et non de montrer comment transformer un fichier source IC en un autre format. Sur ce sujet, de plus amples renseignements se trouvent en [2, 3].

3.1 Exemple en Pure Esterel v5

L'exemple suivant `a pour but d'illustrer dans la suite des transformations en diff´erents formats, la traduction de cet exemple Esterel dans ces formats. L'exemple choisi est ABO, certes r´eduit en instructions du noyau mais suffisament simple pour concevoir les approches dans les diff´erents formats.

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

Nous remarquons l'instruction **await** qui n'est pas une instruction du noyau. **await I** est une macro expans´ee en

```
trap T in
```

```
   loop
      pause;
      present I then
         exit T
      else
         nothing
      end present
   end loop
end trap
```

3.2 Terminologie

Une terminologie r´ecurrente li´ee au domaine des langages *synchrones* est commune aux formats pr´esent´es ici.

Le *tick* repr´esente une r´eaction. Les entr´ees sont fig´ees et les sorties sont positionn´ees en une dur´ee de r´eaction nulle.

Le *start* repr´esente le premier appel d'un programme `a la fonction de r´eaction.

Le *restart* repr´esente tous les autres appels `a la fonction de r´eaction jusqu'`a la terminaison du programme.

Le statut d'un signal peut être *présent* si il a ´et´e ´emis, *absent* si il n'est pas ´emis ou *inconnu* lors de sa d´eclaration.

Le statut d'un nœud dans un graphe repr´esentant un programme Esterel peut être *actif* ou *inactif*. Ce statut est d´etermin´e en fonction de ceux des nœuds fils qui lui propagent l'information.

Le contrôle permet de pr´eciser l'´etat d'*activ*ou d'*inactivité* de l'instruction. Une instruction s'ex´ecute d`es qu'elle acquiert le contrôle. D`es qu'elle termine son ex´ecution elle passe le contrôle `a l'instruction suivante.

3.3 Format IC/LC

3.3.1 Instructions

Le format IC/LC d´efinit 12 constructions pr´esent´ees ci-dessous. Leurs ´equivalents graphiques sont d´ecrits dans la figure 3.1.

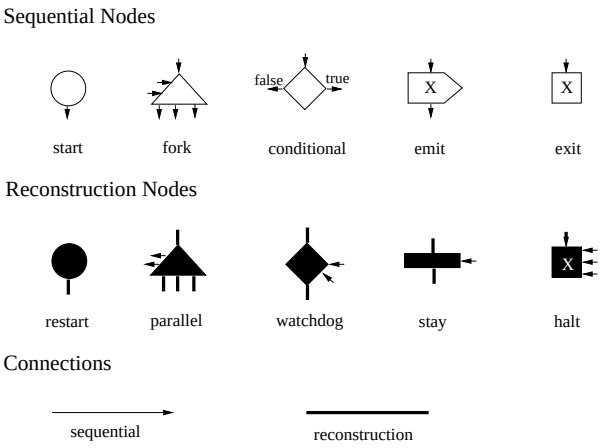


FIG. 3.1 –. Constructions graphiques IC/LC

IC/LC fait apparaitre deux arbres en un:

- L'**arbre de sélection** au travers des **nœuds de reconstruction** indique, lors de chaque *restart*, quels registres (*halt*) sont actifs (*watchdog*) et permet donc une s´election des branches par lesquelles le flot de contrôle doit passer dans L'**arbre de flot de contrôle**
- L'**arbre de flot de contrôle** au travers des **nœuds séquentiels** donnera le contrôle, en s´equence, `a toutes les instructions atteignables, apr`es l'avoir lui-même reçu de l'**arbre de sélection**

Première exécution: *start* Le nœud est associ´e `a la premi`ere ex´ecution de l'application.

Exécutions suivantes: *restart* L'action est r´ealis´ee lors des ex´ecutions suivantes.

Le parallélisme: *fork* Ce nœud correspond `a donner imm´ediatement le contrôle `a toutes les branches qui s'ex´ecutent en parall`ele. Le statut du nœud d´epend de celui des branches parall`eles qui l'informent de leur terminaison.

Instructions séquentielles dans les branches: *parallel* Chaque branche du nœud — appel´e *thread* — regroupe un ensemble d'instructions purement s´equentielles termin´ees par un **registre**. Il connait le statut de terminaison de ses branches, et peut d´ecider en fontion de celui-ci, dans le même instant, de donner le contrôle `a de nouvelles instructions.

Comment clarifier en d'autres termes la diff´erence fondamentale entre *parallel* et *fork* qui peuvent paraitre ´equivalents dans le lexique informatique? L'instruction *parallel* donne le contrôle `a toutes ses branches filles alors que le *fork* donnera certes lui aussi le contrôle `a toutes ses branches d´ecrites par:

- une branche est appel´ee *thread*
- `a chaque *thread* est associ´e un *registre* et un *watchdog*
- le *watchdog* permet, lors de l'acquisition du contrôle de v´erifier si le *registre* associ´e est actif ou pas
- dans le cas o`u il est actif, alors la s´equence d'instructions associ´e reçoit `a son tour le contrôle
- dans le cas o`u il est inactif, alors le *thread* est consid´er´e comme termin´e

Condition d'activation d'une branche: *watchdog* Une branche du graphe de contrôle peut être activ´ee si et seulement si la branche de graphe de s´election associ´ee est active. Ce nœud permet de d´eterminer si la branche du graphe de s´election portant un registre est active ou pas.

Condition de statut: *conditional* Le nœud teste le statut du signal X. Si celui-ci est *présent* alors la branche *true* obtiendra le contrôle, si il est *absent* alors la branche *false* l'obtiendra.

Emission d'un signal: *emit* Le signal X est instantan´ement ´emis, donnant le contrôle `a l'instruction suivante.

Terminaison: *exit* La branche termine avec le statut X, repr´esent´e par un entier naturel. Les statuts sont les suivants:

- 0 indique que la branche s'est termin´ee *normalement*
- 1 indique que la branche est suspendue jusqu'au prochain *restart*. Il est associ´e aux registres (*halt*), indiquant un statut de *non terminaison*
- $x, x \in [\![2, +\infty[\![$ indique que la branche termine par la lev´ee d'une *exception*. L'augmentation des num´eros correspond au niveau d'imbrication des pr´eemptions

Ces codes de terminaison sont transmis au *parallel* associ´e aux *threads* les ayant produits puis interpr´et´es par celui-ci. A titre d'exemple, lorsque toutes les branches ont retourn´e 0 alors l'instruction *parallel* associ´ee est termin´ee et passe le contrôle `a l'instruction suivante.

Suspension: *stay* Ce nœud gêle le contrôle des branches situ´ees en-dessous de celui-ci, jusqu'`a la prochaine r´eactivation. N´ecessairement pr´ec´ed´e d'un *watchdog*, celui-ci v´erifiera son ´etat `a chaque *restart*. Dans le cas o`u la condition de r´eactivation est v´erifi´ee par le *watchdog* alors l'ex´ecution reprendra l`a o`u elle a ´et´e gel´ee.

Registres: *halt* Le nœud repr´esente les registres d´enot´es par un identifiant. Il est *actif* d`es qu'il obtient le contrôle. N´ecessairement pr´ec´ed´e d'un *watchdog*, celui-ci v´erifiera son ´etat `a chaque *restart*.

Flux de contrôle La fl`eche *sequential* donne instantan´ement le contrôle obtenu par le nœud ou l'instruction ne portant pas la fl`eche `a l'instruction ou au nœud cible.

Flux de reconstruction Le trait *reconstruction* permet de combiner les diff´erents *threads* au *restart*.

3.3.2 Exemple

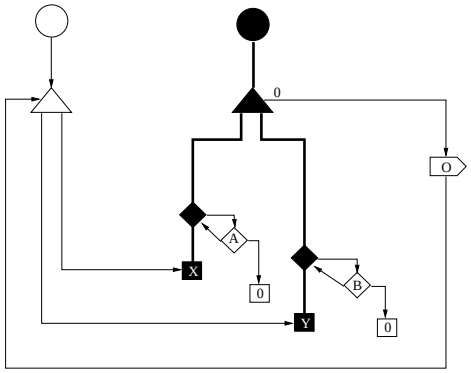


FIG. 3.2 –. ABO en IC/LC

La figure 3.2 d´ecrit la traduction de ABO en IC/LC. La traduction actuelle produit n´eanmoins un code bien plus complexe car elle est structurelle. Au *start*, la seule action est d'activer les registres X et Y. Une branche *fork* r´ealise cette op´eration en cr´eant deux branches d'ex´ecution en même temps. L'application ne compte, outre le registre d´edi´e de *start*, que deux registres. Ainsi, deux *threads* sont cr´e´es portant chacun un registre respectivement X et Y. Lors de chaque *restart* le *watchdog* des *threads* d´etermine si le *thread* est actif. Si c'est le cas, alors le contrôle est donn´e aux instructions associ´ees. D`es qu'une instruction termine sur un test de pr´esence des signaux A ou B, le code de terminaison normale est transmis `a l'instruction *thread*. Dans le cas contraire, le test est falsifi´e et le registre associ´e reste toujours actif. Lorsque cette instruction a reçu les statut de terminaison normale de ses deux *threads* alors elle donne le contrôle `a l'instruction charg´ee de l'´emission du signal O. Elle boucle ensuite instantan´ement sur le *fork* charg´e de rendre le contrôle aux registres X et Y (instruction *loop*).

3.4 Format CCFG de EC

3.4.1 Compilateur EC

Le compilateur EC a ´et´e d´efini par S. Edwards [3] dans les laboratoires de Synopsys, Inc. Etant de ce fait un format propri´etaire, nous ne pourrons pas l'utiliser. Cependant, une description en est faite ci-dessous de sorte `a pr´esenter son potentiel d'expressivit´e. Il est g´en´er´e `a partir du IC/LC de sorte `a interpr´eter le flot de contrôle du graphe IC beaucoup mieux que le feraient les compilateurs bas´es sur un syst`eme de portes trop bas niveau. Le but est d'´eliminer le parall´elisme via des nœuds de d´ecision portant un ´etat, modifi´e tout au long du flot de contrôle. Ainsi, une notion de *contexte* apparait de sorte `a reprendre l'ex´ecution en tenant compte d'une variable d'´etat.

Ce format r´esout de plus le probl`eme des d´ependances de donn´ees par des *fils en pointillés* mettant en relation un signal ´emis et le test de pr´esence sur ce même signal effectu´e dans une autre branche.

3.4.2 Instructions

Le format CCFG de EC d´efinit 11 constructions pr´esent´ees ci-dessous. Leurs ´equivalents graphiques sont d´ecrits dans la figure 3.3.

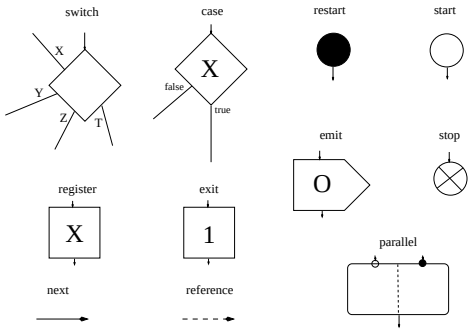


FIG. 3.3 –. Constructions graphiques CCFG

Les capsules: *parallel* Les capsules permettent de d´eclarer des branches parall`eles en y incluant de mani`ere transparente les d´eclarations de signaux locaux. Ces capsules sont hi´erarchiques c'est `a dire qu'elles peuvent contenir d'autres capsules. On appellera *toplevel* l'environnement initial qui n'est contenu dans aucune capsule. Dans chaque branche de la capsule, seront au moins pr´esents trois ´el´ements:

- *start* qui indique comment initialiser la branche lors de la premi`ere ex´ecution lorsque le contrôle lui est transmis
- *restart* qui indique comment ex´ecuter la branche lors des ex´ecutions suivantes lorsque le contrôle lui est transmis
- *exit* qui indique que l'ex´ecution de la branche est termin´ee

Lorsque le contrôle est donn´e `a la capsule alors il est transmis instantan´ement `a toutes les branches. Lorsque toutes les branches ont termin´e alors le contrôle est instantan´ement rendu `a l'instruction suivante pour la relation *next* de la capsule.

Première exécution: *start* L'action est r´ealis´ee lors de la premi`ere ex´ecution , `a l'int´erieur et `a l'ext´erieur des blocs hi´erarchiques. Le contrôle peut n´eanmoins être redonn´e, même apr`es la premi`ere ex´ecution, `a un *start* via le *start* repr´esent´e sur les capsules.

Exécutions suivantes: *restart* L'action est r´ealis´ee lors des ex´ecutions suivantes, `a l'int´erieur et `a l'ext´erieur des blocs hi´erarchiques.

Choix de registre actif: *switch* L'action correspond `a un choix exclusif entre un nombre quelconque de variables. Le contrôle sera donn´e `a la branche portant le nom du registre actif.

Test sur statut: *case* L'action correspond `a un test *if* sur **X**. Si la pr´esence de **X** est ´evalu´ee `a vrai alors la branche *true* sera activ´ee, sinon, celle portant *false*, le sera

Les registres: *register* Les registres portent une lettre les repr´esentant univoquement. Ils sont activ´es si et seulement si dans le cycle courant — *start* ou *restart* — une fl`eche *next* pointe dessus. Lors du prochain cycle un *switch* portera sur une de ses branches la lettre associ´ee `a ce registre.

Un registre `a une port´ee strictement locale i.e. il est visible uniquement dans la branche de d´eclaration. La d´eclaration d'un registre correspond `a lui attribuer une lettre diff´erente de celles associ´ees aux autres registres de la même branche. Ainsi, deux registres de même nom d´eclar´es dans deux branches distinctes sont deux registres diff´erents.

Statut de terminaison: *exit* L'´el´ement graphique repr´esentant une terminaison ressemble aux registres. Cependant, au lieu de porter une lettre, il porte un chiffre entier naturel qui repr´esente son statut de terminaison dans le cycle courant. Par convention, le statut de terminaison normale portera l'entier 0. Son interpr´etation sera faite via un *switch* dont une branche porte l'entier associ´e. Elle pourra être ´evalu´ee lors du prochain *restart* ou bien dans le cycle courant d`es que le contrôle lui parvient.

Emission d'un signal: *emit* Les signaux sont ´emis via une instruction *emit* portant le nom de ce signal. Sous les hypoth`eses du *broadcasting* imm´ediat des ´emissions et de la dur´ee nulle de l'instant, les signaux sont ´emis dans l'environnement et accessibles `a tous ceux susceptibles de les capter. La port´ee d'un signal *interne* est donn´e par une capsule *parallel*. Les signaux *globaux* i.e. d´eclar´es dans l'interface du *module* sont visibles par tous.

Arrêt du contrôle: *stop* Le contrôle s'ach`eve d`es qu'il rencontre un *stop*. S'il se trouve dans une capsule hi´erarchique alors la branche courante est termin´ee. D`es que toutes les branches sont achev´ees en aboutissant sur cette instruction *stop* alors le contrôle est rendu au niveau hi´erarchique directement sup´erieur. Dans le cas o`u il n'y ait plus de niveau hi´erarchique sup´erieur alors l'ex´ecution du cycle courant s'ach`eve.

Flux de contrôle: *next* Les instructions consid´er´ees jusqu'`a pr´esent doivent inter-agir entre elles en se donnant le contrôle. Les fl`eches *next* permet de structurer ces instructions entre elles. L'extr´emit´e qui ne porte pas la fl`eche signifie que l'instruction associ´ee a ´et´e ex´ecut´ee et que, de ce fait, la prochaine point´ee par la fl`eche peut l'être `a son tour.

Dépendances: *reference* Nous avons vu que lors de l'´emission d'un signal, cette ´emission ´etait perçue dans tout le sous-programme inclus dans la port´ee de sa d´eclaration. Cependant, comment attacher `a l'´emission d'un signal le fait qu'une autre branche attend sa pr´esence via un *case*. Les d´ependances servent donc a repr´esenter ces liens ´emission–attente. Elles ne sont pas indispensables mais permettent de visualiser instantan´ement les relations existantes entre ´emetteurs/r´ecepteurs.

3.4.3 Exemple

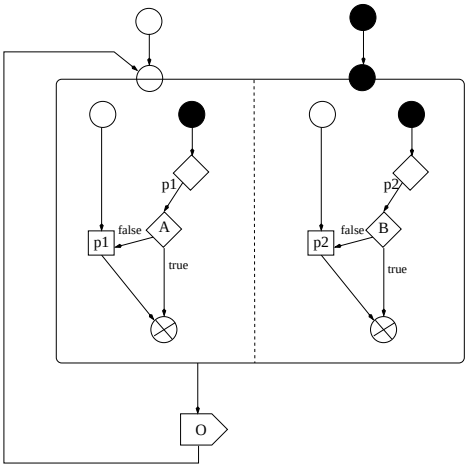


FIG. 3.4 –. ABO en CCFG de EC

La figure 3.4 montre la meilleure mani`ere de traduire le programme Esterel ABO en graphe CCFG d'EC. Une capsule met en parall`ele deux automates de structure identique. L'un d´eclare un registre p_1 dont l'activit´e d´epend du signal A, et l'autre un registre p_2 dont l'activit´e d´epend de B. Lors du *start*, le contrôle est transmis aux deux automates qui activent p_1 et p_2 puis terminent le cycle d'ex´ecution. Lors de *restart* les *switch* des deux branches d´etermineront respectivement si le registre est actif, dans ce cas la pr´esence du signal est ´evalu´ee. S'il est absent alors le registre associ´e est r´eactiv´e puis l'instruction termine. S'il est pr´esent, la terminaison est imm´ediate et le registre associ´e n'est pas r´eactiv´e. Lorsque les deux branches sont termin´ees i.e. p_1 et p_2 sont d´esactiv´es alors l'instruction parall`ele est termin´ee et le contrôle est rendu pour ´emettre le signal O puis reboucler sur l'instruction de *start* symbolisant la boucle.

3.5 Format GRC

GRC, propos´e en [4], distingue *arbre de sélection* et *arbre de flot de contrôle*.

3.5.1 Arbre de sélection

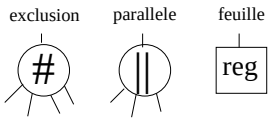


FIG. 3.5 –. Graphe de sélection

L'arbre de s´election hi´erarchise les registres — les feuilles — entre eux. La figure 3.5 pr´esente les types de nœuds utilis´es. Les nœuds et les feuilles sont num´erot´es sur $\mathbb{N}$ avec par convention 0 pour le premier nœud et 1 pour le *boot register*[1]. Ces indices sont utilis´es dans l'*arbre de flot de contrôle* pour pr´eciser sur quel(s) nœud(s) les actions sont r´ealis´ees.

Les nœuds de séquence

Les branches sortantes sont en exclusion i.e. une seule `a la fois poss`ede le contrôle. Le contrôle est donn´e aux branches de la gauche vers la droite. Lorsque la branche la plus `a droite devient inactive via le *graphe de flot de contrôle* alors le contrôle du nœud de s´equence est rendu `a son nœud p`ere. Si le nœud se fait pr´eempter, alors, `a la charge du *graphe de flot* de g´erer la lev´ee de l'exception.

Les nœuds d'exclusion

Les branches sortantes sont en exclusion. Le contrôle peut être donn´e `a n'importe quelle branche i.e. aucun ordre sur celles-ci ne peut être pr´edit. Ainsi, rendre le contrôle au nœud p`ere du nœud d'exclusion revient enti`erement `a la charge du *graphe de flot de contrôle*.

Les nœuds de parallélisation

Toutes les branches sortantes sont activ´ees d`es lors que le contrôle atteint le nœud. Lorsque toutes les branches sont d´esactiv´ees via le *graphe de flot de contrôle* alors le contrôle est rendu au nœud p`ere du nœud de parall´elisation. Si le nœud se fait pr´eempter, alors, `a la charge du *graphe de flot* de g´erer la lev´ee de l'exception.

Les nœuds feuilles

Les feuilles de l'arbre portent n´ecessairement un *registre*, tel que **pause**, ou r´ecemment **pre**. Toutes les *macros* mettant en jeu l'une de ces primitives noyau contiennent aussi des feuilles: **await**, **halt**, ... en l'occurence.

3.5.2 Arbre de flot de contrôle

La figure 3.6 pr´esente les constructions utilis´ees dans un graphe de contrôle.

Initialisation: *tick*

Ce nœud est utilis´ee une seule et unique fois dans un graphe de flot de contrôle. Il repr´esente l'initialisation du syst`eme lors du premier appel `a la fonction de r´eaction, puis le d´ebut d'´evaluation du syst`eme pour les prochains instants.

Actions: *action*

Lorsque le contrôle est transmis `a ce nœud une s´erie d'actions associ´ees est imm´ediatement r´ealis´ee. Dans le cas o`u des signaux — locaux ou d´eclar´es en *output* du syst`eme — sont ´emis alors ils sont repr´esent´es comme ´emis dans l'environnement (par exemple S_1 et S_2 sur ce sch´ema). Le contrôle est instantan´ement rendu et le flot propag´e au prochain nœud.

1. registre associ´e au *start*

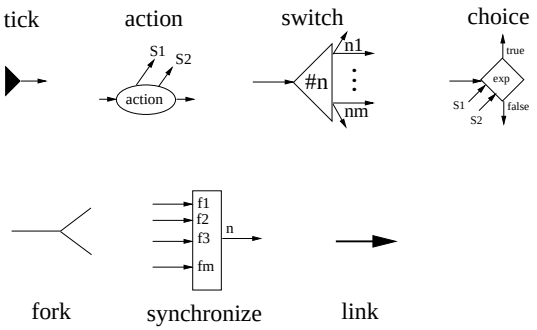


FIG. 3.6 –. Graphe de flot de contrôle

Des actions particuli`eres ont ´et´e d´efinies. Elles sont repr´esent´ees dans la figure 3.6 par le nœud *action* portant le nom de celles-ci.

- *emit* : permet l'´emission dans l'environnement de signaux
- *enter*: permet de rendre actif un nœud du graphe de s´election en pr´ecisant son ´etiquette
- *exit* : permet de rendre inactif un nœud du graphe de s´election en pr´ecisant son ´etiquette

Parallélisme: *fork*

L'expression du parall´elisme est incluse au format de description, il s'agit d'un lien [Lien: *link* 3.5.2] qui en engendre deux autres donnant ainsi le contrôle `a deux branches. Il est `a relier au nœud de parall´elisation de l'arbre de s´election.

Choix multiple: *switch*

D`es que le contrôle arrive `a ce nœud, un choix doit être fait de sorte `a le propager dans les branches appropri´ees. Ce nœud est ´etiquet´e par un num´ero de nœud du graphe de s´election lui ´etant associ´e. Le type du nœud du graphe de s´election indiquera qu'une ou plusieurs branches seront actives. Cependant, quelle(s) branche(s) activer? Le graphe de s´election d´etermine justement via les num´eros des nœuds, les branches `a activer. Ainsi, chaque branche sortant du *switch* est explicitement li´ee `a un nœud du graphe de s´election via son ´etiquette.

Choix simple: *choice*

Quand ce nœud reçoit le contrôle, il op`ere un test (son ´etiquette) sur des signaux de l'environnement (ils sont repr´esent´es par $S_1$ et $S_2$ sur le sch´ema) ou sur l'activit´e d'un nœud du graphe de s´election. Le contrôle est imm´ediatement transmis `a la branche *true* si l'expression est ´evalu´ee `a *vrai* ou `a la branche *false* si elle est ´evalu´ee `a *faux*.

La synchronisation: *synchronize*

Ce nœud permet de synchroniser diff´erentes branches parall`eles. A chaque terminaison de branche ($f_1 .. f_2$) est associ´e un statut. Le nœud combinera ces statuts de sorte `a d´ecider s'il peut ou pas donner le contrôle aux instructions suivantes.

Lien: *link*

Les nœuds de l'arbre de flot de contrôle sont li´es entre eux via des flêches indiquant le sens de transmission du flot de contrôle.

3.5.3 Exemple

La figure 3.7 montre la traduction de l'exemple ABO dans la repr´esentation du format GRC.

L'arbre de s´election (en bas `a gauche) pr´ecise les d´ependances entre les registres associ´es `a $boot$, A et B. Le nœud ´etiquet´e 0 pr´ecise qu'un seule des deux branches `a la fois pourra être active. Celui ´etiquet´e 2 indique que les deux branches sont mises

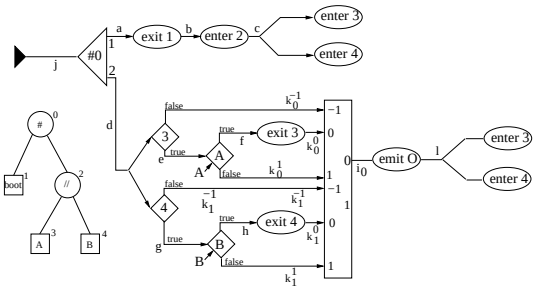


FIG. 3.7 –. ABO en GRC

en parall`ele. L'arbre de contrôle montre comment le contrôle se propage dans le graphe lors de chaque *start* ou *restart* en fonction des nœuds actifs donn´es par l'arbre de s´election.

3.6 Synthèse et choix du format intermédiaire

La pr´esentation de ces formats: IC/LC, CCFG de EC et GRC a propos´ee une mani`ere de traduire un programme ´ecrit en Exterel Pur v5 en graphes. En effet, une telle repr´esentation am´eliore la compr´ehension et le travail des compilateurs, optimiseurs, ... Notre objectif est de transformer de l'Esterel en VHDL. Ainsi, une alternative possible est de partir d'un langage exprimant le sens d'un programme ´ecrit en Esterel Pur v5, de sorte `a faciliter la production d'un prototype. Plusieurs param`etres sont `a prendre en consid´eration.

L'analyse de ces formats interm´ediaires nous a amen´e `a choisir le format interm´ediaire GRC. Pourquoi?

3.6.1 Coût d'accés au format intermédiaire

En fonction de ces langages mis `a notre disposition nous devons en choisir un qui n´ecessite peu d'efforts d'interfaçage avec la future application. De ce fait le format EC ´etant propri´et´e de Synopsys, Inc nous ne pouvons pas l'utiliser.

3.6.2 Graphes

Le format interm´ediaire d'un programme Esterel Pur v5 est scind´e en deux parties:

a- un premier graphe qui permet lors des activations de donner les branches (et registres) actives au graphe g´erant l'enchaînement des actions

b- un second graphe qui ordonnance les actions entre elles

GRC permet d'acc´eder `a ces deux graphes appel´es arbre de s´election et graphe de flot de contrôle.

3.6.3 Réincarnation

La r´eincarnation traite de la duplication d'un signal dans un même instant. Le format devra si possible, pour des raisons de facilit´e, g´erer ce ph´enom`ene baptis´e *schizophrénie* des signaux. En l'occurence, IC/LC ne le g`ere pas alors que GRC le propose dans ses sp´ecifications.

La section de code suivante montre un cas de r´eincarnation en Esterel o`u le signal A ne sera jamais ´emis.

```
...
loop
  signal S, A in
    present S then emit A end;
    pause;
    emit S;
  end signal
```

```
end loop
...
```

Le signal S test´e est toujours absent. Il est ´emis au tick suivant, le corps de la boucle se termine et reboucle instantan´ement cr´eant une instance fraiche de S. C'est ce nouveau signal qui est test´e et non celui ´emis. Pour ´eviter ce ph´enom`ene, le corps de la boucle est dupliqu´e:

```
...
loop
  signal S, A in
    present S then emit A end;
    pause;
    emit S;
  end signal;
  signal S, A in
    present S then emit A end;
    pause;
    emit S;
  end signal
end loop
...
```

3.7 Simulation GRC

La s´emantique de simulation du format [11] est ´equivalante `a la s´emantique *contructive* [12]. L'arbre de s´election permet de coder l'´etat du syst`eme `a la fin de chaque instant pour l'instant suivant. Le graphe de flot de contrôle, initi´e par l'instruction d'activation, va propager le contrôle de nœuds en nœuds sur un chemin dirig´e par l'´etat courant du syst`eme et les statuts des signaux. Tout au long du chemin des actions sur les signaux permettent leur ´emission, la modification de l'arbre de s´election ou l'initiation d'effets de bords qui d´epassent notre sujet [4].

Un codage judicieux exploitant la structure hi´erarchique de l'arbre de s´election [5] permet de r´eduire efficacement le nombre de registres codant l'´etat du syst`eme.

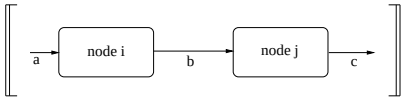
Chapitre 4

Génération du source VHDL

Cette partie montre l'interpretation des constructions de GRC dans l'optique de la traduction vers VHDL. Les instructions GRC sont repr´esent´ees par un syst`eme de processus parall`eles activ´es par une condition de r´eveil.

4.1 Système de processus

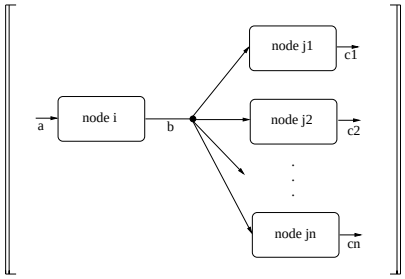
4.1.1 Séquence



$$\equiv\ [\triangleright_a]\mathcal{P}([\![node\ i]\!];\ !\blacktriangleright_b)\ |\ [\triangleright_b]\mathcal{P}([\![node\ j]\!];\ !\blacktriangleright_c)$$

L'op´erateur | d´eclare en parall`ele deux processus $\mathcal{P}$ guard´es par une condition d'activation sur des signaux repr´esent´e par une liste de signaux mis entre crochets []. L'instruction $node\ i$ sera ´evalu´ee d`es que le signal a aura activ´e le processus. La convention $\triangleright_a$ signifie que le signal a a ´et´e ´emis dans l'environnement par un autre processus par $!_a\blacktriangleright$D`es que $node\ i$ a termin´e alors le signal de continuit´e $b$ est ´emis permettant le r´eveil du processus ex´ecutant $node\ j$. La s´equence est marqu´ee par un $delta$-delai au sens VHDL, repr´esentant la causalit´e due `a la s´equence entre la fin de l'ex´ecution de $node\ i$ et le d´ebut de l'ex´ecution de $node\ j$. Le signal b a volontairement ´et´e rajout´e pour obliger les deux processus `a se s´equentialiser.

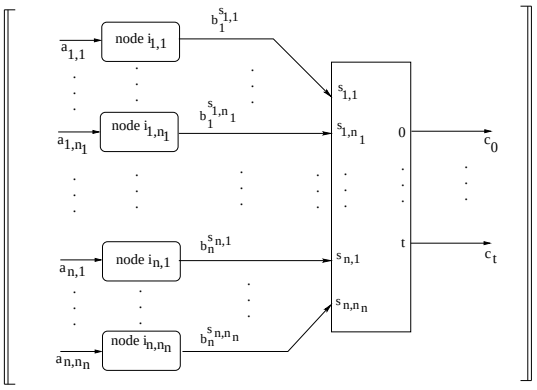
4.1.2 Parallélisme et synchronisation



$$\begin{array}{l}\equiv\ [\triangleright_a]\ \mathcal{P}([\![node\ i]\!];\ !\blacktriangleright_b)\\ |\ [\triangleright_b]\ \mathcal{P}([\![node\ j_1]\!];\ !\blacktriangleright_{c_1})\\ |\ [\triangleright_b]\ \mathcal{P}([\![node\ j_2]\!];\ !\blacktriangleright_{c_2})\\ |\ \vdots\\ |\ [\triangleright_b]\ \mathcal{P}([\![node\ j_n]\!];\ !\blacktriangleright_{c_n})\end{array}$$

Le parall´elisme entre les nœuds $j, \ldots, j_n$ est exprim´e par un ensemble de processus parall`eles associ´es `a l'´evaluation de ces nœuds. Ils sont tous sensibles au même signal b ´emis apr`es la terminaison de l'´evaluation de $node\ i$.

Esterel implante un *parallèle synchrone*, la construction de synchronisation suivante permet d'attendre la fin de l'ex´ecution de toutes ses branches.



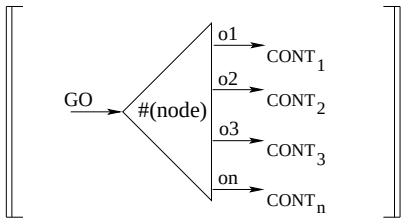
$$
\begin{array}{l}
\equiv\ [\triangleright_{a_{1,1}}]\,\mathcal{P}([\![node\ i_{1,1}]\!];\ !\triangleleft_{b_1}^{s_{1,1}}) \\
\vdots \\
\mid [\triangleright_{a_{1,n_1}}]\,\mathcal{P}([\![node\ i_{1,n_1}]\!];\ !\triangleleft_{b_1}^{s_{1,n_1}}) \\
\vdots \\
\mid [\triangleright_{a_{n,1}}]\,\mathcal{P}([\![node\ i_{n,1}]\!];\ !\triangleleft_{b_n}^{s_{n,1}}) \\
\vdots \\
\mid [\triangleright_{a_{n,n_n}}]\,\mathcal{P}([\![node\ i_{n,n_n}]\!];\ \triangleleft_{b_n}^{s_{n,n_n}}) \\
\mid [\triangleleft_{b_1},\ \ldots,\ \triangleleft_{b_n}]\mathcal{P}(!\ \blacktriangleright_{c_{max(Val(b_i),i\in[\![1,n]\!])}})
\end{array}
$$

Nous introduisons les signaux d'exception $\triangleleft_{b_{i,j}}^{s}$ lev´es par une branche $b_i, i \in [\![1,n]\!]$. A chaque instant, une exception est lev´ee par toutes les branches $b_i, i \in [\![1,n]\!]$, portant un statut s de terminaison. $Val(signal)$ permet de r´ecup´erer la valeur associ´ee au signal $signal$. La fonction max calcule le maximum des valeurs port´ees par $b_1, \ldots, b_n$. Cet indice est n´ecessairement compris entre 0 et t qui sont les seules continuit´es possible apr`es synchronisation des branches.

Les branches ne terminent pas n´ecessairement dans le même instant: celles ayant termin´e dans le pass´e ´emettent une exception portant le status -1.

L'attente du prochain cycle s'exprime en bloquant le contrôle dans le synchroniseur par la continuit´e sp´ecifique c.

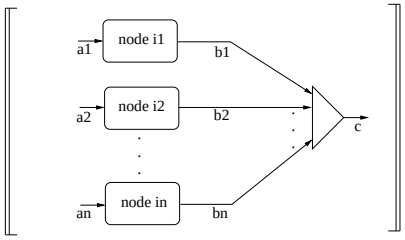
4.1.3 Exclusion



$$\equiv\ [\triangleright_{GO}]\,\mathcal{P}_{\sharp}([o_1]!\ \blacktriangleright_{\widehat{CONT_1}}, \ldots, [o_n]!\ \blacktriangleright_{CONT_n})$$

Il s'agit d'un processus de s´election d'une seule et unique branche `a chaque instant: c'est un automate avec une variable interne. Le signal de continuit´e GO permet d'activer le s´electionneur qui ´emettra dans l'environnement le signal de continuit´e marqu´e sous le chapeau. Le chapeau repr´esente via la variable interne, l'´etat courant de l'automate. Il est d´eplac´e dynamiquement en fonction de la r´eception d'un des signaux de guarde $o_1, \ldots, o_n$ exclusifs entre eux ´emis dans l'environnement.

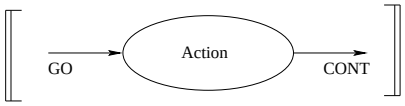
Toutes ces branches exclusives peuvent se rejoindre de sorte `a donner le contrôle `a une partie de code commune.



$$
\begin{array}{l}
\equiv\ [\triangleright_{a_1}]\ \mathcal{P}([\![node\ i_1]\!];\ !\blacktriangleright_{b_1}) \\
|\ [\triangleright_{a_2}]\ \mathcal{P}([\![node\ i_2]\!];\ !\blacktriangleright_{b_2}) \\
\vdots \\
|\ [\triangleright_{a_n}]\ \mathcal{P}([\![node\ i_n]\!];\ !\blacktriangleright_{b_n}) \\
|\ [\triangleright_{b_1}]\ \mathcal{P}(!\blacktriangleright_c) \\
\vdots \\
|\ [\triangleright_{b_n}]\ \mathcal{P}(!\blacktriangleright_c)
\end{array}
$$

Le signal de continuit´e c est ´emis par au moins un processus activ´e par sa condition de guarde sensible `a un des signaux $b_1,\ldots,b_n$.

4.1.4 Appel à une action

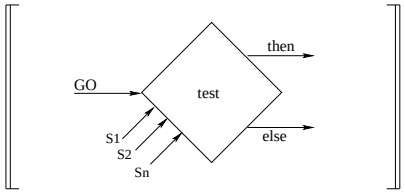


$$
\begin{array}{l}
\equiv\ [\triangleright_{GO}]\ \mathcal{P}(![\![Action]\!];\ !\blacktriangleright_{CONT}) \\
[\![Action]\!] \equiv
\end{array}
$$

$!\blacktriangleright_i, i \in \mathbb{N}$	le signal de selection i est positionn´e `a *enter*.
$!\blacksquare_i, i \in \mathbb{N}$	le signal de selection i est positionn´e `a *exit*.
$!\triangleleft_b^s$	le signal d'exception b porte la valeur s.
$!S$	le signal local ou d´eclar´e dans l'interface en signal de sortie est ´emis

Lorsque le processus est activ´e par le signal GO, il r´ealise une des actions pr´ec´edentes puis ´emet le signal de continuit´e $CONT$.

4.1.5 Test



$$\equiv\ [\triangleright_{GO}]\ \mathcal{P}(\{test(S_1,\ldots,S_n)\}\ ?\ !\blacktriangleright_{then} :!\blacktriangleright_{else})$$

Lorsque le processus acquiert le contrôle par le signal GO, le test $test$ est r´ealis´e. Si la condition est ´evalu´ee `a *vrai* alors le signal $then$ est ´emis, sinon $else$ est ´emis.

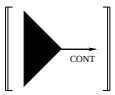
Un test peut ´evaluer le statut d'un *signal de sélection* positionn´e soit `a *enter* ´emettant $then$, soit `a *exit* ´emettant $else$. La condition de test -$\{\blacktriangleright_i\}$?- ´evalue l'´etat du signal de s´election$_i$ ▶

Il peut ´evaluer le statut d'un *signal de l'interface* positionn´e soit `a *present* ´emettant $then$, soit `a *absent* ´emettant $else$. La condition de test -$\{A\}$?- ´evalue le statut du signal S d´eclar´e dans l'interface du module.

La combinaison logique de l'´evaluation de ces signaux emploie les op´erateurs logiques $\wedge$ (et), $\vee$ (ou) et $\neg$ (non).

Le test peut aussi s'appliquer aux *signaux internes* utilisant les mêmes op´erateurs et r`egles d'´evaluation que pour les signaux pr´ec´edents. Les hypoth`eses d'*réaction à l'absence* et de *causalité* ne nous permettent pas de prendre la d´ecision d'´evaluer les statuts des signaux internes. Un point de synchronisation $\bullet$ permet de suspendre l'´evaluation du test jusqu`a ce que le statut de tous les signaux internes requis par le test aient un statut *present* — ´emis par un processus concurrent — ou *absent* — d´ecid´e par un m´ecanisme d'ordonnancement —. La condition de test -$\bullet\{S\}$?- ´evalue le statut du signal interne $S$ d`es que son statut dans l'instant est d´etermin´e.

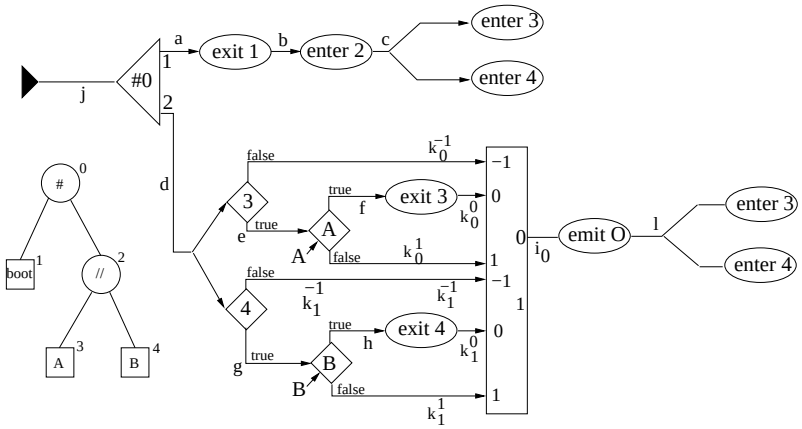
4.1.6 Réaction



$$\equiv\ [tick]\,\mathcal{P}(!\,\blacktriangleright_0;\ !\,\blacktriangleright_1;\ !\,\blacktriangleright_{CONT})$$

Sur le signal de d´ebut d'instant $tick$, le processus est activ´e, positionnant pour l'instant courant les signaux de s´election particuliers 0 et 1 `a *enter*. Le contrôle est ensuite donn´e `a la prochaine instruction via le signal de continuit´e $CONT$. Le s´eparateur ";" permet de mettre en s´equence une s´erie d'instructions `a l'int´erieur d'un processus. La s´equence repr´esente un *bloc de base* qui induit l'atomicit´e de l'´evaluation de ces instructions.

4.1.7 Exemple



L'exemple ci-dessus repr´esente ABO traduit intuitivement en GRC. Les lettres sur les transitions sont les noms des signaux de continuit´e. Il se traduit en un syst`eme de processus parall`eles.

$[tick]\mathcal{P}(!\blacktriangleright_0;\ !\blacktriangleright_1;\ !\blacktriangleright_j)$	Sur la pr´esence de $tick$, les signaux de s´election 0 et 1 sont positionn´es d`es l'instant courant `a *enter* et le signal de continuit´e j est ´emis.
$\mid [\triangleright_j]\ \mathcal{P}_\sharp(\widehat{[\blacktriangleright_1]!\ \blacktriangleright_a},\ [\blacktriangleright_2]!\ \blacktriangleright_d)$	La variable d'´etat de l'automate est remise `a jour d`es qu'un ´ev`enement se produit sur les signaux de s´election 1 ou 2. Sur la pr´esence de j, le signal de continuit´e a ou d est ´emis selon l'´etat de la variable interne d'´etat.
$\mid [\triangleright_a]\mathcal{P}(!\blacksquare_1;\blacktriangleright_b)$	Sur la pr´esence de a, le signal de s´election 1 est positionn´e `a *exit* au prochain instant et le signal de continuit´e b est ´emis.
$\mid [\triangleright_b]\mathcal{P}(!\ \blacktriangleright_2;\blacktriangleright_c)$	Sur la pr´sence de b, le signal de s´election 2 est positionn´e `a *enter* au prochain instant et le signal de continuit´e c est ´emis.
$\mid [\triangleright_c]\mathcal{P}(!\ \blacktriangleright_3)$ $\mid [\triangleright_c]\mathcal{P}(!\ \blacktriangleright_4)$	Sur la pr´sence de c, les signaux de s´election 3 et 4 sont positionn´es `a *enter* au prochain instant, en parall`ele. Aucune continuit´e n'est ´emise: l'instant est termin´e.
$\mid [\triangleright_d]\mathcal{P}(\{\blacktriangleright_3\}\ ?\ !\ \blacktriangleright_e : !\triangleleft^{-1}_{k_1})$	Sur la pr´esence de d, l'´etat du signal de s´election 3 est test´e permettant d'´emettre e si son statut est *enter*. Sinon, un signal d'exception k_1 est ´emis portant la valeur -1: la branche a dans le pass´e termin´e son ´evaluation.
$\mid [\triangleright_d]\mathcal{P}(\{\blacktriangleright_4\}\ ?\ !\ \blacktriangleright_g : !\triangleleft^{-1}_{k_2})$ $\mid [\triangleright_e]\mathcal{P}(\{A\}\ ?\ !\ \blacktriangleright_f : !\triangleleft^{1}_{k_1})$	Sur la pr´esence de e, le signal d'interface A *présent* assure l'´emission du signal f. Si il est *absent* alors le signal d'exception k_1 prend le statut 1 marquant la fin de l'instant.
$\mid [\triangleright_g]\mathcal{P}(\{B\}\ ?\ !\ \blacktriangleright_h : !\triangleleft^{1}_{k_2})$ $\mid [\triangleright_f]\mathcal{P}(!\blacksquare_3;\ !\triangleleft^{0}_{k_1})$	Sur la pr´esence de f, le signal de s´election 3 est positionn´e `a *exit* au prochain instant et le signal d'exception k_1 est ´emis avec 0 en statut marquant la fin de l'´evaluation de la branche.
$\mid [\triangleright_h]\mathcal{P}(!\blacksquare_4;\ !\triangleleft^{0}_{k_2})$ $\mid [\triangleleft_{k_1},\triangleleft_{k_2}]\mathcal{P}(!\ \blacktriangleright_{i_{max(Val(a),Val(b))}})$	Lorsque les deux signaux k_1 et k_2 sont pr´esents alors la valeur maximum des valeurs port´ees est calcul´ee. Si elle vaut 1 l'instant est termin´e, sinon le signal de continuit´e i_0 est ´emis. Le cas -1 ne peut pas se produire.
$\mid [\triangleright_{i_0}]\mathcal{P}(!O;\ !\blacktriangleright_l)$ $\mid [\triangleright_l]\mathcal{P}(!\ \blacktriangleright_3)$ $\mid [\triangleright_l]\mathcal{P}(!\ \blacktriangleright_4)$	

4.2 Réduction du système

Nous avons introduit avec la traduction du nœud associ´e au $tick$ la notion de bloc de base. Nous proposons de r´eduire le syst`eme pr´ec´edent en construisant des blocs de base bas´es sur les crit`eres suivants.

1- Les actions qui s'enchainent par l'interm´ediaire de signaux de continuit´e comprenant aussi les actions s'ex´ecutant en parall`ele au même $\delta$-delai. Par construction, le sens donn´e `a un ensemble de processus parall`eles $\{P_i\}$ r´ealisant les actions $\{a_i\}$ s´equentialis´es par des signaux est ´equivalant `a celui donn´e au bloc de base constitu´e par la s´equence d'actions $\{a_i\}$. De plus, par l'hypoth`ese de la dur´ee nulle de l'instant, r´ealiser deux actions en parall`ele revient `a les s´equentialiser.

2- Les tests sur les signaux de l'interface ou les signaux de s´election.

Les processus suivant ne peuvent pas être r´eduits.

1- Les tests sur les signaux locaux dus `a la causalit´e d'une ´emission se produisant dans l'instant mais pouvant arriver au moins un $\delta$-delai apr`es le test de pr´esence, et la r´eaction `a l'absence qui implique un m´ecanisme d'ordonnancement.

2- Les jointures de branches d'un s´electionneur qui permettent de factoriser le code commun. Nous pouvons cependant r´eduire le nombre de p processus engendr´es par les p branches `a un seul en le rendant sensible `a la pr´esence d'au moins un signal de continuit´e des branches. Cette optimisation d'implantation n'affecte en rien le sens donn´e mais r´eduit le nombre de processus de $p-1$.

3- Les s´electionneurs qui sont des automates distincts. Ils influencent en grande partie le parcours du flot de contrôle et pourraient faire l'objet lors de la synth`ese de la d´efinition d'un composant de base.

4- Les synchroniseurs qui permettent de rendre synchrone les branches d'un parall`ele Esterel.

Exemple

L'ensemble de processus d´ecrit pr´ec´edemment se r´eduit en

$$
\begin{array}{l}
[tick]\mathcal{P}(!\blacktriangleright_0;\ !\blacktriangleright_1;\ !\blacktriangleright_j) \\
|\ [\triangleright_j]\ \mathcal{P}_{\sharp}(\widehat{[\blacktriangleright_1]!\blacktriangleright_a}, [\blacktriangleright_2]!\blacktriangleright_d) \\
|\ [\triangleright_a]\ \mathcal{P}(!\blacksquare_1;\ !\blacktriangleright_2;\ !\blacktriangleright_3;\ !\blacktriangleright_4) \\
|\ [\triangleright_a]\ \mathcal{P}(\{\blacktriangleright_3\}\ ?\ \{A\}\ ?\ !\blacksquare_3, !\triangleleft^0_{k_1}\ :!\triangleleft^1_{k_1}\ :!\triangleleft^{-1}_{k_1};\ \{\blacktriangleright_4\}\ ?\ \{B\}\ ?\ !\blacksquare_4, !\triangleleft^0_{k_2}\ :!\triangleleft^1_{k_2}\ :!\triangleleft^{-1}_{k_2}) \\
|\ [\triangleleft_{k_1}, \triangleleft_{k_2}]\ \mathcal{P}(!\blacktriangleright_{i_{max(Val(k_1),\ Val(k_2))}}) \\
|\ [\triangleright_{i_0}]\ \mathcal{P}(!O;\ !\blacktriangleright_3;\ !\blacktriangleright_4)
\end{array}
$$

Le nombre de processus a ´et´e r´eduit de 16 `a 6 et le nombre de signaux de continuit´e de 11 `a 3. Un passage `a l'´echelle s'impose afin de v´erifier l'efficacit´e de ces gains en termes de taille et de simulation.

4.3 Ordonnancement

Le syst`eme de processus suivant $\{P_1,P_2,P_3,P_4,P_5,P_6,P_7\}$ pr´esente un probl`eme de causalit´e entre le processus P_7 ´emettant le signal local S et le test de ce signal dans P_1. Nous montrons aussi le probl`eme li´e `a la r´eaction `a l'absence du signal local S' pour le test dans P_3.

Nous supposons que S et S' n'ont jamais ´et´e ´emis jusqu'au δ-delai courant et qu'un ´ev`enement s'est produit sur a.

$P_1 = [\triangleright_{a_0}]\mathcal{P}(\bullet_{sync_1}\{S\}\ ?\ !\blacktriangleright_{a_1} : !\blacktriangleright_{a_2})$	S est de statut ind´etermin´e, le processus se suspend.
$\|P_2 = [\triangleright_{a_0}]\mathcal{P}(!action_3;\ !\blacktriangleright_{a_5})$	L'action $action_3$ est r´ealis´ee et a modifi´e.
$\|P_3 = [\triangleright_{a_5}]\mathcal{P}(\bullet_{sync_2}\{S'\}\ ?\ !\blacktriangleright_{a_6} : !\blacktriangleright_{a_7})$	S' est de statut ind´etermin´e, le processus se suspend.
$\|P_4 = [\triangleright_{a_1}]\mathcal{P}(!action_1;\ !\blacktriangleright_{a_3})$	Le processus n'est pas r´eveill´e alors qu'il devrait l'être
$\|P_5 = [\triangleright_{a_2}]\mathcal{P}(!action_2;\ !\blacktriangleright_{a_4})$	
$\|P_6 = [\triangleright_{a_6}]\mathcal{P}(!action_4;\ !\blacktriangleright_{a_8})$	
$\|P_7 = [\triangleright_{a_7}]\mathcal{P}(!S;\ !\blacktriangleright_{a_9})$	Le processus n'est pas r´eveill´e alors qu'il devrait l'être, ´emettant S.

L'´etat du syst`eme se fige dans un ´etat non coh´erent: il ne respecte pas les hypoth`eses d'*instant communément* et d'*action à l'absence*.

Nous proposons de r´esoudre ce probl`eme par des signaux de synchronisation $sync_1$,$sync_2$ requ´erant le d´eblocage d'au moins un processus sensible `a un de ces signaux. Un processus *superviseur* joue le rôle d'arbitre: l'ordre d'´evaluation des processus activ´es par un signal de synchronisation a ´et´e fix´e de mani`ere statique suivant la relation $\prec$ appliqu´ee, $\forall i \in [\![1,7]\!]\}$. Les programmes trait´es ´etant *acycliques*, nous pouvons d´efinir un tri topologique.

4.3.1 Ordre

Soit $\{P^{U_i}_{i_{S_i}}\}_i$ un ensemble de processus. L'ensemble S_i contient les signaux locaux ´emis dans la fermeture des branches contrôl´ees par P_i. L'ensemble U_i contient les signaux locaux `a $P_i$ devant être positionn´es `a un statut *present* ou *absent*. Nous associons `a $\{P^{U_i}_{i_{S_i}}\}_i$ la relation $\prec$ telle que

$$\forall p^{B_p}_{A_p}, q^{B_q}_{A_q} \in \{P^{U_i}_{i_{S_i}}\}_i, (p^{B_p}_{A_p} \prec q^{B_q}_{A_q} \iff A_q \cap B_p = \varnothing)$$

Dans le syst`eme pr´ec´edent, nous devons ordonner $P^{\{S\}}_{1\varnothing}$ et $P^{\{S'\}}_{3_{\{S\}}}$. D'apr`es la d´efinition pr´ec´edente,

$$P^{\{S'\}}_{3_{\{S\}}} \prec P^{\{S\}}_{1\varnothing}$$

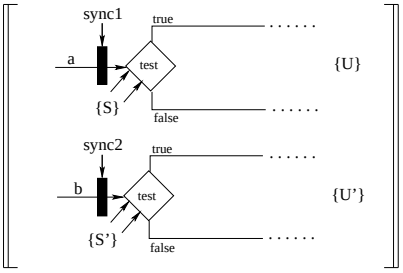
4.3.2 Absence des signaux

L'ordonnancement d'un syst`eme $\mathfrak{S}$ de processus avec la relation $\prec$ d´efinie au-dessus permet de r´esoudre les signaux *absents* de l'ensemble $\mathfrak{Absent}$.

$$\forall P_A^B, Q_C^D \in \mathfrak{S},\ P_A^B \prec Q_C^D \implies B \subseteq \mathfrak{Absent} \wedge D \backslash A \subseteq \mathfrak{Absent}$$

Dans l'exemple, $S' \in \mathfrak{Absent}$.

4.3.3 Processus d'ordonnancement



$$\begin{aligned}
&\equiv\ P_1 \ =\ [\triangleright_a]\,\mathcal{P}(\bullet_{sync_1}\{test(\{S\})\}\ ?\ [\![then]\!]\ :\ [\![else]\!])\\
&|\ P_2 \ =\ [\triangleright_b]\,\mathcal{P}(\bullet_{sync_2}\{test(\{S'\})\}\ ?\ [\![then]\!]\ :\ [\![else]\!])\\
&|\ P_3 \ =\ \mathcal{P}([\triangleright_a][\![S \cap U' \ =\ \varnothing \Longrightarrow S \subseteq \mathfrak{Absent}]\!];\ !\blacktriangleright_{sync_1}\ ,[\triangleright_b][\![S' \backslash U \subseteq \mathfrak{Absent}]\!];\ !\blacktriangleright_{sync_2})
\end{aligned}$$

Le corps du processus P_3 est calcul´e statiquement. Soit $\{S\}$ et $\{S'\}$ les ensembles de signaux locaux test´es dans les deux processus P_1, P_2 respectifs. $\{U\}$ et $\{U'\}$ sont les ensembles des signaux locaux ´emis par la fermeture des branches des deux processus P_1, P_2 respectifs. Nous admettrons que $S \cap U' = \varnothing$ apr`es calcul statique.

D`es que le signal $a$ est ´emis alors nous pouvons positionner tous les signaux de $\{S\}$ avec le statut *absent* et ´emettre le signal de synchronisation $sync_1$ de sorte `a d´ebloquer le test du processus P_1. Lorsque b est ´emis alors les signaux $\{S'\}\backslash\{U\}$ sont positionn´es `a *absent* puis le signal de synchronisation $sync_2$ d´ebloque le test du processus P_2.

Exemple

Le syst`eme de processus $\{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$ pr´esente le besoin d'ordonnancer $P_{1\varnothing}^{S}$ et $P_{3S}^{S'}$. Le processus $P_8$ y r´epond en ´emetant le signal local $'S$ avec le statut *absent*: $\overline{S'}$ permet de d´ebloquer le syst`eme entier.

$$P_8 \ =\ \mathcal{P}([\triangleright_{a_5}]!\overline{S'};\ !\blacktriangleright_{sync_2}\ ,\ [\triangleright_{a_1}]!\blacktriangleright_{sync_1})$$

4.4 Interprétation VHDL

4.4.1 Signaux

Les identificateurs mentionn´es dans cette section sont du type discut´e. Un tableau montre ensuite leur traduction lors de leurs utilisations.

Signaux de l'interface

Si le signal S de type primitif std_logic est pr´esent alors il vaut $'1'$, sinon $'0'$.

$\{S\}\ ?\ [\![then]\!]\ :\ [\![else]\!]$ $!S$	if S = '1' then $[\![then]\!]$ else $[\![else]\!]$ end if S <= '1'

Signaux de sélection

Le signal S du type $SelectionSignal$ suivant peut prendre soit le statut *enter* soit *exit*. Le champ *go* pr´ecise l'´etat du signal dans l'instant courant en fonction du statut positionn´e dans l'instant pr´ec´edent.

```
type SelectionSignal is record
   enter : boolean;
   exit  : boolean;
   go    : boolean;
end record;
```

$!\blacksquare_S$	S.enter <= $false$; S.exit <= true;
$!\blacktriangleright_S$	S.enter <= $true$; S.exit <= false;
$[\triangleright_S]$	*voir la traduction des processus*
$\{\blacktriangleright_S\}$? $[\![then]\!]$: $[\![else]\!]$	if S.enter and S.go then $[\![then]\!]$ else $[\![else]\!]$ end if

Signaux de continuité

Le signal S du type $GoSignal$ suivant est *activé* ou pas.

```
type GoSignal is record
   go : boolean;
end record;
```

$!\blacktriangleright_S$	S.go <= true
$[\triangleright_S]$	*voir la traduction des processus*

Signaux d'exception

Le signal S du type $ExitSignal$ suivant a un statut d'´emission et porte la valeur lev´ee par la branche.

```
type ExitSignal is record
   go     : boolean;
   status : integer;
end record;
```

$!\triangleleft^s_S$	S.go <= true; S.status <= s;
$[\triangleleft^s_S]$	*voir la traduction des processus*

Signaux locaux

Le signal S du type $InnerSignal$ suivant peut avoir comme statut *présent*, *absent* ou *indéterminé*. Un signal de potentiel lui est associ´e de sorte `a faciliter les synchronisations dues `a leur ordonnancement.

```
type InnerSignal is record
   present   : boolean;
   absent    : boolean;
   potential : boolean;
end record;
```

$!S$	S.present <= true; S.absent <= false;
$!\overline{S}$	S.present <= false; S.absent <= true;
$!\pi_S$	S.potential <= true;
$\bullet\{S\}$? $[\![then]\!]$: $[\![else]\!]$	if S.absent or S.potential then $[\![else]\!]$ else if S.present then $[\![then]\!]$ end if; end if

Signaux multi-sources

Les δ-delais sont fortement utilis´es dans la traduction de sorte `a respecter la causalit´e. Cette contrainte ne nous permet pas d'utiliser directement les fonctions de r´esolution associ´ees aux signaux apparaissant en partie gauche d'une affectation dans plusieurs processus. Ainsi, un syst`eme du type:

$[\triangleright_a]\mathcal{P}^1(!\blacktriangleright_c)$
$[\triangleright_b]\mathcal{P}^2(!\blacktriangleright_c)$

est transform´e, par l'interm´ediaire de substitutions lexicales σ, en:

$[\triangleright_a]\mathcal{P}^1(!\blacktriangleright_{\sigma(c/c_1)})$
$[\triangleright_b]\mathcal{P}^2(!\blacktriangleright_{\sigma(c/c_2)})$
$[\triangleright_{c_1}|\triangleright_{c_2}]\mathcal{P}^3_\sharp([c_1]!\blacktriangleright_{\sigma(c_1/c)};\ [c_2]!\blacktriangleright_{\sigma(c_2/c)})$

Remarque: les *patterns* de traduction suivants sont pr´esent´es pour des signaux poss´edant une source unique. Dans la r´ealit´e, la plupart des signaux (signaux de continuit´e, signaux d'exception et signaux de s´election) sont tous a priori multi-sources. Ces traductions devront ainsi n´ecessairement être modifi´ees de sorte a prendre en consid´eration les substitutions σ et le processus de r´esolution suppl´ementaire.

4.4.2 Processus

$\mathcal{P}([\![body]\!])$

```
P: process is
begin
   [| body |]
end process;
```

Processus et listes de sensitivités

L'activation d'un processus est guard´ee par une condition.

1. $[\triangleright_a]\ \mathcal{P}_\sharp([\blacktriangleright_1]\ \widehat{[\![seq_1]\!]},\ \ldots,\ [\blacktriangleright_n]\ [\![seq_n]\!])$

```
SelSwitch_i: process(a) is
   variable variableSelSwitch_i : integer := 0;
begin
   if selection_1.enter then variableSelSwitch_i := 0; end if;
   ...
   if selection_n.enter then variableSelSwitch_i := n; end if;

   if a.go = '1' then
      case variableSelSwitch_i is
         when 0      => [| seq_1  |]
         ...
         when n      => [| seq_n  |]
         when others => null;
      end case;
      a.go <= false;
   end if;
end process P;
```

2. $[\triangleright_S]\ \mathcal{P}([\![body]\!])$

```
P: process (S) is
begin
   if S.go then
      [| body |]
   end if;
end process P;
```

3. $[\triangleright_{S_1},\ldots,\triangleright_{S_n}]\ \mathcal{P}([\![body]\!])$

```
P: process (S1, ..., Sn) is
begin
   if S1.go and ... and Sn.go then
      [| body |]
   end if;
end process P;
```

4. $[\triangleleft_{p_1}, \ldots, \triangleleft_{p_m}]\ \mathcal{P}(!\ \blacktriangleright_{o_{max_{i=1}^{m}(Val(p_i))}})$

```
Synchronizer_i: process (p1, ..., pm) is
   variable max : natural := -1;
begin
   if p1.go and ... and pm.go then
      if p1.status > max then
         max := p1.status;
      end if;
      ...
      if pm.status > max then
         max := pn.status;
      end if;

      case max is
         when 0      => o_0.go <= true;
         when 2      => o_2.go <= true;
         ...
         when others => null;
      end case
   end if;
end process;
```

5. $[\triangleright_T]\mathcal{P}(\bullet\{S\}\ ?\ [\![then]\!]\ :\ [\![else]\!])$

```
Test_i: process (T, S) is
begin
   if T.go then
      if S.absent or S.potential then
         [| else |]
      else
         if S.present then
            [| then |]
         end if;
      end if;
   end if;
end process;
```

4.4.3 Processus de début de réaction

Lorsque le signal $tick$ est ´emis, le processus $[tick]\ \mathcal{P}(!\ \blacktriangleright_0; !\ \blacktriangleright_1; !\ \blacktriangleright_{start})$ r´ealise une s´erie de r´einitialisations de signaux avant d'´emettre le signal de continuit´e.

- Les signaux internes S sont positionn´es `a la valeur *ind´eterminée*.
- Les signaux de s´election T deviennent actifs si il sont entr´es dans le mode *enter* lors de la r´eaction pr´ec´edente.
- Les signaux de continuit´e G sont r´einitialis´es.
- Les signaux d'exception E sont r´einitialis´es.

```
Start: process is
begin
   wait until tick'event and tick = '1';
   -- inner signals
   S.present   <= false;
   S.absent    <= false;
   S.potential <= false;

   -- selection signal
   T.go        <= T.enter

   -- exit signals
   E.go        <= false;

   -- control passing signal
   G.go        <= false;
   start.go    <= false;
```

```
   start.go    <= true;

end process;
```

4.4.4 Exemple

```
entity ABO_entity is
   port (
      clk   : in std_logic;  % signal d'horloge
      rst   : in std_logic;  % signal de reset
      A     : in std_logic;
      B     : in std_logic;
      R     : in std_logic;
      O     : out std_logic
   );

begin
   type SelectionSignal is record
      enter    : boolean;
      exit     : boolean;
      go       : boolean;
   end record;
   type GoSignal is record
      go     : boolean;
   end record;
   type ExitSignal is record
      go       : boolean;
      status   : integer;
   end record;
   type InnerSignal is record
      present  : boolean;
      absent   : boolean;
      potential: boolean;
   end record;
end entity ABO_entity;


architecture ABO_architecture of ABO_entity is
   signal selection0  : SelectionSignal := SelectionSignal'(enter => true, exit => false, go => true);
   signal selection1  : SelectionSignal := SelectionSignal'(enter => true, exit => false, go => true);
   signal selection2  : SelectionSignal;
   signal selection3  : SelectionSignal;
   signal selection4  : SelectionSignal;
   signal selection5  : SelectionSignal;
   signal selection6  : SelectionSignal;
   signal selection7  : SelectionSignal;
   signal selection8  : SelectionSignal;
   signal selection9  : SelectionSignal;
   signal selection10 : SelectionSignal;
   signal K_0         : ExitSignal;
   signal K_1         : ExitSignal;
   signal K_2         : ExitSignal;
   signal K_3         : ExitSignal;
   signal K_4         : ExitSignal;
   signal else9       : GoSignal;
   signal K_5         : ExitSignal;
   signal else14      : GoSignal;
   signal cont0       : GoSignal;

   Start: process is
   begin
      wait until tick'event and tick = '1';
      selection0.go  <= selection0.enter;
      selection1.go  <= selection1.enter;
      selection2.go  <= selection2.enter;
```

```
   selection3.go  <= selection3.enter;
   selection4.go  <= selection4.enter;
   selection5.go  <= selection5.enter;
   selection6.go  <= selection6.enter;
   selection7.go  <= selection7.enter;
   selection8.go  <= selection8.enter;
   selection9.go  <= selection9.enter;
   selection10.go <= selection10.enter;
   K_0.go         <= false;
   K_1.go         <= false;
   K_2.go         <= false;
   K_3.go         <= false;
   K_4.go         <= false;
   else9.go       <= false;
   K_5.go         <= false;
   else14.go      <= false;
   cont0.go       <= false;
   cont0.go       <= true;
end process;

SelSwitch0: process(cont0, selection1, selection2) is
   variable SelSwitch0variable : natural := 0;
begin
   if selection1.enter then SelSwitch0variable := 0; end if;
   if selection2.enter then SelSwitch0variable := 1; end if;

   if cont0.go  then
      case SelSwitch0variable is
         when 0 =>
            selection1  <= SelectionSignal'(false, true, selection1.go);
            selection2  <= SelectionSignal'(true, false, false);
            selection3  <= SelectionSignal'(true, false, false);
            selection4  <= SelectionSignal'(true, false, false);
            selection5  <= SelectionSignal'(true, false, false);
            selection6  <= SelectionSignal'(true, false, false);
            selection7  <= SelectionSignal'(true, false, false);
            K_0         <= ExitSignal'(status => 1, go => true);
            selection8  <= SelectionSignal'(true, false, false);
            selection9  <= SelectionSignal'(true, false, false);
            selection10 <= SelectionSignal'(true, false, false);
            K_1         <= ExitSignal'(status => 1, go => true);
         when 1 =>
            if selection5.enter and selection5.go then
               if A = '1' then
                  selection6 <= SelectionSignal'(false, true, selection6.go);
                  K_2        <= ExitSignal'(status => 0, go => true);
               else
                  else9.go <= true;
               end if;
            else
               K_2 <= ExitSignal'(status => -1, go => true);
            end if;
            if selection8.enter and selection8.go then
               if B = '1' then
                  selection9 <= SelectionSignal'(false, true, selection9.go);
                  K_5        <= ExitSignal'(status => 0, go => true);
               else
                  else14.go <= true;
               end if;
            else
               K_5 <= ExitSignal'(status => -1, go => true);
            end if;
         when others => null;
      end case;
      cont0.go <= false;
   end if;
end process;
```

```
Synchronizer0: process(K_0, K_1) is
   variable max := integer := -1;
begin
   if K_0.go and K_1.go  then
      if K_0.status > max then
         max := K_0.status;
      end if;
      if K_1.status > max then
         max := K_1.status;
      end if;
      case max is
         when 1 => null;
         when others => null;
      end case;
   end if;
end process;

Synchronizer1: process(K_2, K_5) is
   variable max := integer := -1;
begin
   if K_2.go and K_5.go  then
      if K_2.status > max then
         max := K_2.status;
      end if;
      if K_5.status > max then
         max := K_5.status;
      end if;
      case max is
         when 0 =>
            selection4  <= SelectionSignal'(false, true, selection4.go);
            O           <= '1';
            selection4  <= SelectionSignal'(true, false, false);
            selection5  <= SelectionSignal'(true, false, false);
            selection6  <= SelectionSignal'(true, false, false);
            selection7  <= SelectionSignal'(true, false, false);
            K_3         <= ExitSignal'(status => 1, go => true);
            selection8  <= SelectionSignal'(true, false, false);
            selection9  <= SelectionSignal'(true, false, false);
            selection10 <= SelectionSignal'(true, false, false);
            K_4         <= ExitSignal'(status => 1, go => true);
         when 1 => null;
         when others => null;
      end case;
   end if;
end process;

Synchronizer2: process(K_3, K_4) is
   variable max := integer := -1;
begin
   if K_3.go and K_4.go  then
      if K_3.status > max then
         max := K_3.status;
      end if;
      if K_4.status > max then
         max := K_4.status;
      end if;
      case max is
         when 1 => null;
         when others => null;
      end case;
   end if;
end process;

Join0: process(else9) is
begin
   if else9.go  then
```

```
         selection7 <= SelectionSignal’(false, true, selection7.go);
         selection6 <= SelectionSignal’(true, false, false);
         selection7 <= SelectionSignal’(true, false, false);
         K_2        <= ExitSignal’(status => 1, go => true);
      end if;
   end process;

   Join1: process(else14) is
   begin
      if else14.go  then
         selection10 <= SelectionSignal’(false, true, selection10.go);
         selection9  <= SelectionSignal’(true, false, false);
         selection10 <= SelectionSignal’(true, false, false);
         K_5         <= ExitSignal’(status => 1, go => true);
      end if;
   end process;
end architecture ABO_architecture;
```

Conclusion et Perspectives

Le travail effectu´e a permis de proposer un mod`ele de traduction d'un programme initial Esterel en VHDL respectant la s´emantique de simulation. Le format interm´ediaire d'Esterel, GRC, pr´esentait l'avantage de r´esoudre des probl`emes li´es aux hypoth`eses des langages synchrones et de la s´emantique constructive. L'attention s'est donc principalement port´ee sur la mise en relation de la s´emantique de simulation des constructions Esterel — ou GRC — avec celle orient´ee ´ev`enements de VHDL.

L'expression des nœuds GRC a ´et´e abstraite par un syst`eme de processus concis orient´es simulation VHDL, communiquant via des signaux. L'architecture d'un programme Esterel est refl´et´ee par une collection de processus parall`eles dont l'activation d´epend d'une condition sur des signaux ´emis par d'autres processus. Un ordonnanceur de processus, calcul´e statiquement, est cependant n´ecessaire de sorte `a r´esoudre les statuts des signaux locaux, le moteur de simulation de VHDL ne pouvant pas r´esoudre les signaux de statut *absent* `a un instant physique. Nous avons ensuite apport´e des optimisations — en nombre de processus et de signaux de continuit´e — en regroupant les instructions *simples* en groupes de base s´equentiels.

Finalement, produire du code VHDL `a partir d'une mod´elisation `a base de tels processus a largement contribu´e `a acc´el´erer le processus de compilation. Nous obtenons un ensemble de processus VHDL synth´etisables communiquant via des signaux. Des registres permettent de conserver l'´etat du syst`eme `a chaque instant.

Le code VHDL g´en´er´e pour l'exemple ABO est compar´e au syst`eme de portes actuel produit, en nombre d'instructions par crit`ere.

Crit`ere	VHDL comportemental	VHDL RTL[16]	Interpr´etation
Registres	11	5	Chaque nœud de l'arbre de s´election GRC est un registre. En utilisant les propri´et´es hi´erarchiques [5], de l'arbre nous pouvons en r´eduire leur nombre.
Signaux/Variables	10	47	Une description RTL n´ecessite plus de connexions qu'une description comportementale haut niveau.
Composants	8	2	VHDL RTL est compos´e d'un processus combinatoire ´evalu´e `a tout instant et d'un processus synchrone de mise `a jour des registres. En VHDL comportemental, seules les parties actives dans l'instant sont ´evalu´ees.
Lignes de code	214	138	VHDL RTL est une ´enum´eration d'´equations bool´eennes. Ce crit`ere est ici peu significatif car la verbosit´e de VHDL comportemental ne refl`ete pas le code *utile*.
Temps de simulation	–	–	Les outils de simulation industriels ne sont pas disponibles.

Actuellement, le travail se focalise sur l'implantation de l'ordonnanceur de signaux internes. Une fois achev´e et les outils op´erationnels, les phases de tests commenceront avec des co-simulations de programmes Esterel compil´es en VHDL RTL et VHDL Comportemental. Un passage `a l'´echelle sur des programmes Esterel de r´ef´erence permettront d'estimer les coûts des crit`eres du tableau pr´ec´edent. Une derni`ere phase consistera `a estimer les coûts en composants, fils et registres des deux descriptions VHDL apr`es synth`ese.

La continuit´e du projet suppose l'extension du syst`eme de processus avec la prise en compte de *tout* Esterel (signaux valu´es, effets de bord, . . .).

De plus, combiner des mod`eles de simulation Esterel et VHDL — le syst`eme de processus que je propose n'est qu'un mod`ele de simulation d'Esterel en VHDL — permettrait de sp´ecifier un syst`eme de contrôle synchrone indiff´eremment en Esterel ou VHDL, passant de l'un `a l'autre par des outils de traduction. Les outils existant pour les deux langages pourraient indiff´eremment être employ´es et se compl´eter.

Actuellement, un mod`ele commun existe `a un trop bas niveau (*net-list*) occultant la structure d'un programme Esterel ou VHDL d'origine. L'objectif est de combiner les styles de sorte `a accroitre les gains d'efficacit´e. L'´etude r´ealis´ee durant le stage de DEA a permis l'´evaluation orient´ee VHDL (simulation *event-driven* de programme Esterel (simulation *time-driven*).

Une ouverture vers d'autres langages de sp´ecifications — Verilog, Lustre, CowareC, SystemC, SpecC, VHDL-AMS, . . . — permettrait la proposition d'un mod`ele commun de simulation attendu par les industriels dans un but d'inter-op´erabilit´e entre les diff´erentes sp´ecifications.

Bibliographie

[1] G. Berry & G. Gonthier, The Esterel synchronous programming language: design, semantics, implementation. *Science of computer programming*, 19(2):87–152, Nov. 1992

[2] S. Edwards, Synopsys Inc, Compiling Esterel into Sequential Code. *Proceedings of the 7th International Workshop on Hardwarw/Software Codesign* (CODES'99), Rome, Italy, May 3–5, 1999

[3] S. Edwards, Columbia University, An Esterel Compiler for Large Control-Dominated Systems. *Department of Computer Sciences*

[4] D. Potop–Butucaru Fast Redundancy Elimination Using High-Level Structural Information from Esterel RR 4330 (C) 2002 INRIA Sophia-Antipolis

[5] D. Potop–Butucaru, R. de Simone Optimizing for Faster Simulation of Esterel Programs ENSMP/CMA/INRIA Sophia-Antipolis

[6] F. Boussinot, Objets r´eactifs en JAVA Presses Polytechniques et Universitaires Romandes

[7] Action MIMOSA, Projet MEIJE, http://www.inria.fr/rapportsactivite/RA2000/mimosa/logic_reactif-dynami.html INRIA Sophia-Antipolis

[8] G. Berry, The Esterel v5 Language Primer Version v5_91, CMA/ENSMP/INRIA Sophia-Antipolis

[9] J. Vuillemin, Th´eorie des technologies de l'information Magist`ere de Math´ematiques Fondamentales & Appliqu´ees et d'Informatique, ENS

[10] E. Closse, . . . SAXO-RT: Interpreting Esterel Semantic on a Sequential Execution Structure France Telecom R&D

[11] D. Potop-Butucaru, The Graph Intermediate Format, document interne

[12] G. Berry, The Constructive Semantics of Pure Esterel CMA/ENSMP/INRIA

[13] IEEE, IEEE Standard VHDL Language Reference Manual

[14] IEEE, IEEE P1076.6/D2.0 Draft Standard For VHDL Register Transfer Level Synthesis

[15] G. Gonthier, S´emantique et mod`eles d'ex´ecution des langages r´eactifs synchrones; application `a Esterel Universit´e de Paris-Sud

[16] A. Ressouche, Compiling synchronous reactive languages into Verilog/VHDL INRIA Sophia-Antipolis