Rapport de stage — Mpri M2
Compilation de ReactiveML

Louis Jachiet dans l'équipe PARKAS (ENS) sous la direction de Louis Mandel

18 mars - 22 août 2013

Le contexte général

Les langages synchrones ont été introduits dans les années 80 pour programmer les systèmes temps réel critiques avec une structure de contrôle complexe. Ces langages proposent des constructions de composition parallèle où le temps y est vu comme une suite d'instants. À chaque instant, le programme réagit aux stimuli de l'environnement. On y fait l'hypothèse que les communications et le calcul se font de manière instantanée. En effet, les différents processus ne peuvent communiquer que par signaux (une construction ad-hoc du langage).

Dans un langage synchrone comme Esterel [BC84], pendant un instant, chaque processus peut émettre des signaux ; pendant cet instant tous les autres processus voient que ce signal a été émis. Le statut d'un signal (*présent* ou *absent*) n'est connu que si un processus l'a émis ou si l'on est sûr qu'aucun processus ne peut plus l'émettre pendant l'instant.

Comme les processus ne communiquent que par signaux et que la lecture du statut d'un signal ne se fait qu'après son éventuelle émission, l'ordre d'exécution des différents processus pendant un instant ne peut donc pas changer le résultat : le parallélisme donc déterministe !

Les langages synchrones étant dédiés à la programmation de systèmes critiques, ils avaient une expressivité limitée pour que l'analyse de dépendance entre les signaux puisse être réalisée à la compilation. Cela permet d'ordonnancer statiquement le parallélisme et ainsi faire des analyses de pire temps d'exécution (WCET). Dans les années 90, Frédéric Boussinot a montré avec le langage ReactiveC [Bou91] qu'en supprimant l'analyse de dépendance à la compilation, on pouvait introduire du parallélisme synchrone dans des langages généralistes. Ainsi, le modèle de concurrence synchrone peut être utilisée pour la programmation d'applications non temps-réels. ReactiveML [MP08] est un tel langage composite : OCaml doté de constructions Esterel.

Le problème étudié

```
let process cascade s1 s2 =
  present s1 then (present s2 then print_int 42)
  ||
  present s1 then emit s2
  ||
  emit s1
```

Figure 1: Exemple de programme Esterel à optimiser

En ReactiveML, il n'y a pas d'analyse sur l'ordonnancement. Celui ci est dynamique. Dans un programme comme celui de la figure 1, on va commencer par lancer les deux premières branches qui vont se mettre en attente de *s1*. Ensuite, on exécute la troisième branche qui va émettre *s1* et donc réactiver les deux premières. La première va se mettre en attente de *s2* puis la seconde va émettre *s2* ; cette émission va alors activer une troisième fois la première branche qui affichera enfin 42. En ReactiveML chacune de ces mises en attente est coûteuse car on crée à chaque fois une clôture tandis qu'une analyse statique du programme aurait pu déterminer l'ordre dans lequel exécuter les différentes branches.

L'objectif de ce stage était de programmer un compilateur effectuant une telle analyse et utilisant son résultat pour ordonnancer statiquement les différents processus. Une analyse précise de la totalité des programmes ReactiveML valides étant un objectif hors de portée, on essaie de couvrir le fragment le plus grand possible en identifiant les difficultés soulevées par les constructions du langage.

La contribution proposée

Différentes analyses et différentes méthodes de compilation pour le langage Esterel ont déjà été largement étudiées. En particulier pour de la compilation vers du logiciel efficace, il y a : le compilateur *grc2c* [PB02] de chez Inria, le compilateur *CEC* [Edw94] de l'université de Columbia et *Saxo-RT* développé à France-Télécom pour [CPP+02].

Ma contribution à l'étude de la compilation de ReactiveML était de commencer par étudier le langage Esterel (présenté dans la section 1) et s'en inspirer pour compiler ReactiveML (faire de l'ordonnancement statique).

Pour cela un premier compilateur d'un fragment d'Esterel vers OCaml fortement inspiré par les méthodes classiques de compilation a été développé. Celui est présenté dans la section 2. Cependant, les techniques classiques de compilation pour Esterel sont adaptées pour le code embarqué et produisent donc du code statique (pas d'allocation dynamique, pas de pile d'appel de fonction) et donc ces techniques s'étendent difficilement à la compilation de ReactiveML. En ReactiveML tous les objets sont dans des processus qui peuvent se lancer plusieurs fois simultanément et donc tous les objets sont crées dynamiquement.

La seconde partie du stage a donc consisté en le développement d'un compilateur utilisant une technique originale de compilation pour Esterel plus facilement extensible à la compilation de tout ReactiveML.

La fin du stage a porté sur l'amélioration de ce compilateur : ajouter les fonctionnalités de ReactiveML manquantes ou chercher à comprendre ce qui les rendait intrinsèquement difficiles à ajouter.

Les arguments en faveur de sa validité

Cette méthode de compilation produit en moyenne du code plus rapide sur certains types de programmes (avec beaucoup de communication instantanée). Par exemple, le code généré pour le programme de la figure 1 s'exécute plus de 2 fois plus rapidement qu'avec le compilateur natif. Sur les exemples qui multiplient les communications instantanées les gains peuvent être beaucoup plus importants. La section A détaille plus précisément ces questions de performance.

Le bilan et les perspectives

Le développement fait pendant le stage a été réalisé dans le compilateur ReactiveML mais il n'est capable pour l'instant que de compiler des programmes qui font un usage relativement simples des signaux. Si le compilateur rejette des programmes c'est parce qu'il n'arrive pas à les analyser. La prochaine grande étape est donc naturellement d'améliorer cette analyse.

Quelques améliorations peuvent aussi être apportées à la phase de génération de code. Le code produit pour le moment demande des optimisations faites à la main pour obtenir de bonnes performances.

Enfin, le compilateur n'est pour l'instant que partiellement intégré au compilateur ReactiveML. On peut imaginer dans un futur proche que le compilateur ReactiveML appelle le compilateur présenté dans ce rapport pour optimiser des parties du code (et non la totalité). Cette approche permettrait de combiner les performances de notre méthode de compilation avec le support complet du langage fourni par le compilateur actuel.

Remerciements

Je remercie mon encadrant, Louis Mandel, pour avoir su cadrer mes efforts, parfois trop dispersés, et pour son aide, même pendant les vacances. Je remercie toute l'équipe Parkas pour les conversations qui ont su approfondir ma culture, informatique ou non, et répondre à mes questions ; enfin, je suis reconnaissant à l'ENS et à son département d'informatique pour son cadre convivial et pour tous les gens que j'ai pu rencontrer (*Parkas*, *Grotas*, *Ulminfo*).

1 ESTEREL

Le langage de programmation ESTEREL est utilisé pour programmer des systèmes réactifs avec une structure de contrôle complexe. L'exécution d'un programme ESTEREL est vue comme une suite d'instants. Un système réactif est un système où, à chaque instant, le programme réagit à des événements en entrée et produit des événements en sortie.

1.1 Parallélisme

Le langage ESTEREL permet de lancer plusieurs processus en parallèle qui peuvent interagir. Contrairement à d'autres modèles de parallélismes, comme le parallélisme par thread, où un ordonnanceur décide comment répartir le calcul entre les différents threads selon une logique qui lui est propre, dans le modèle synchrone, le parallélisme est coopératif : à chaque instant, tous les processus seront activés et l'instant se finit après que tous les processus se sont mis en pause. Les instants se terminant quand tous les processus sont mis en pause, ils ne durent pas nécessairement tous aussi longtemps.

1.2 Signaux

Pour communiquer les processus utilisent des signaux. À chaque instant, un signal a un statut : *présent* ou *absent*. Un signal est considéré absent sauf si un (ou plusieurs) processus l'émet pendant l'instant.

On impose une règle de cohérence aux signaux : le statut d'un signal ne peut pas changer pendant un instant. Pour respecter cette règle, un compilateur n'acceptera donc que les programmes causaux : les programmes causaux sont les programmes où la lecture du statut d'un signal se fait nécessairement après ses éventuelles émissions (après en temps réel et non en temps synchrone).

Par exemple, le programme 2A n'est pas causal et aucun statut pour **s** n'est cohérent (si **s** est présent alors **s** n'est pas émis et si **s** est absent alors **s** est émis) . Le programme 2B n'est pas causal non plus mais les deux statuts pour **s** sont possibles, le programme serait donc non-déterministe. Enfin, dans le programme 2C, la présence de **s** est cohérente mais le programme est refusé car il est n'est pas causal : l'émission de **s** se fait après la lecture de son statut.

```
signal s in
present s else emit s
```

(A) Absence implique présence

```
signal s in
present s then emit s
```

(B) Émission après le test présence

```
signal s in
present s then emit s
          else emit s
```

(C) Émission après le test présence

FIGURE 2: Exemples de programmes non causaux

1.3 Le langage ESTEREL

On ne considère pas tout ESTEREL mais seulement PURE ESTEREL (au sens de [Ber99]) et sans les instructions d'échappement (qui n'existent pas en REACTIVEML) ; les autres constructions manquantes se dérivent de celles de PURE ESTEREL. La grammaire du langage est la suivante :

$$
\begin{array}{lll}
\langle stmt \rangle & ::= & \text{`}\texttt{pause}\text{'} \\
 & | & \text{`}\texttt{nothing}\text{'} \\
 & | & \text{`}\texttt{print}\text{'}\ \langle string \rangle \\
 & | & \text{`}\texttt{signal}\text{'}\ \langle ident \rangle\ \text{`}\texttt{in}\text{'}\ \langle stmt \rangle\ \text{`}\texttt{end}\text{'} \\
 & | & \text{`}\texttt{emit}\text{'}\ \langle ident \rangle \\
 & | & \text{`}\texttt{present}\text{'}\ \langle ident \rangle\ \text{`}\texttt{then}\text{'}\ \langle stmt \rangle\ \text{`}\texttt{else}\text{'}\ \langle stmt \rangle\ \text{`}\texttt{end}\text{'} \\
 & | & \langle stmt \rangle\ \text{`}\texttt{;}\text{'}\ \langle stmt \rangle
\end{array}
$$

| ⟨*stmt*⟩ '||' ⟨*stmt*⟩
| 'suspend' ⟨*stmt*⟩ 'when' ⟨*ident*⟩
| 'loop' ⟨*stmt*⟩ 'end'

— *nothing* ne fait rien ;
— *pause* met le processus en attente de l'instant d'après ;
— *print s* affiche la chaîne de caractère *s* ;
— *signal S in p* déclare le signal *s* local à *p* ;
— *emit S* passe le statut de *S* à présent ;
— *present S then A else B* exécute *A* si *S* est présent, *B* sinon. On peut omettre la branche *then* ou la branche *else* ;
— *q ; p* effectue *q* et quand *q* a terminé, effectue *p* ;
— *q // p* lance immédiatement en parallèle l'exécution de *q* et de *p*. Cette instruction termine quand les *q* et *p* ont tous deux terminé ;
— *suspend p when S* termine quand p a terminé mais suspend ; temporairement l'exécution de *p* si *S* est présent ;
— *loop p end* répète *p* indéfiniment.

1.4 Quelques exemples de programmes en Esterel

```
signal Tick2 in
signal Tick3 in
  loop
    pause ;
    pause ;
    emit Tick2
  end
||
  loop
    pause ;
    pause ;
    pause ;
    emit Tick3
  end
||
  loop
    present Tick3
    then present Tick2
         then print "Tick6" ;
    pause
  end
end
end
end
```

(A) Affiche Tick6 tous les six instants

```
signal s1 in
  signal s2 in
    emit s2 ;
    present s1 then print "42"
   ||
    present s2 then emit s1
  end
end
```

(B) Exemple de processus à couper

```
loop
  signal S in
    present S then print "42"
    pause ;
    emit S ;
  end
end
```

(C) Exemple de schizophrénie

Figure 3: Exemples de programmes Esterel

Le programme 3A est composé de trois branches parallèles. La première branche émet *tick2* tous les deux instants, la deuxième émet *tick3* tous les trois instants et la troisième branche affiche "Tick6" dès que *tick3* et *tick2* sont présents simultanément. Ainsi le programme 3A affiche "Tick6" tous les 6 instants.

Le programme 3B est composé de deux branches parallèles. La première émet *s2* puis teste la présence de *s1* tandis que la seconde commence par tester la présence de *s2* puis émet *s1*. Ce programme s'exécute instantanément mais on peut remarquer qu'au cours de l'instant, il y a un dialogue entre les deux branches.

Il faut donc que le compilateur prévoit d'exécuter le *emit s2* de la première branche, qu'ensuite il exécute toute la seconde branche puis qu'il finisse la première branche. Le programme 3C boucle sur un test de signal et une émission. À partir du deuxième instant, on commence l'instant par la fin de la boucle et on émet s puis on recommence la boucle et on fait le test *present s*. Cependant, cette émission et ce test ne portent pas sur le même s car s a été redéfini au début de la boucle. Le programme n'affiche donc jamais "42". On parle alors de schizophrénie, en effet les programmes ESTEREL sont souvent utilisés dans des logiciels embarqués critiques, on n'utilise donc pas d'allocation dynamique et on souhaite partager au maximum les données entre les instants.

2 Le compilateur ESTEREL vers OCAML

Dans cette partie, nous présentons un premier compilateur de notre fragment d'ESTEREL vers OCAML. Ce compilateur est très fortement inspiré par les compilateurs CEC et INRIA. Le compilateur utilise un format intermédiaire appelé GRC (GRaph Code). Nous présentons donc d'abord le format GRC, puis la compilation vers ce format, et enfin la compilation vers le langage cible. La section 2.5 explique le besoin d'effectuer des changements à la méthode de compilation pour que cette méthode puisse s'appliquer à REACTIVEML.

2.1 Le format GRC

Le format GRC est composé de deux graphes : l'arbre de sélection, et un graphe de transition.

L'*arbre de sélection*, associe à chaque instruction q, une variable booléenne, notée $[q]$. À la fin de chaque instant, cette variable est vraie si et seulement si l'instruction q a été mise en pause sans avoir terminé.

Le graphe de transition représente les changements à appliquer sur l'état des nœuds de l'arbre de sélection pendant un instant. Au début de l'instant, on commence par noter les nœuds comme inactifs sauf le nœud d'entrée du graphe que l'on marque comme activé. Chaque nœud activé activera alors certains de ses fils (les nœuds activés étant traités dans un ordre topologique). On distingue les différents types de nœuds donnés figure 4. Lorsqu'ils sont activés, leur comportement est le suivant :

— un nœud de type 4A affiche une chaîne de caractères et active ses fils ;
— un nœud de type 4B modifie la valeur d'une variable et active ses fils ;
— un nœud de type 4C lit la valeur de la variable *var*. Si *var* alors le nœud active les fils sur sa sortie $\top$ sinon il active ceux sur $\bot$;
— un nœud de type 4D active les nœuds sur la sortie $\mathbf{A}_s$ s'il est activé par $\mathbf{A}_1$ et $\mathbf{A}_2$, le nœud active $\mathbf{B}_s$ s'il est activé par $\mathbf{B}_1$ ou $\mathbf{B}_2$;
— un nœud de type 4E ne peut pas être activé et n'active pas d'autres nœuds. Ces nœuds ne servent qu'à marquer des contraintes d'ordonnancement (symbolisées par des arcs en pointillés). On les appellera des "nœuds de dépendances".

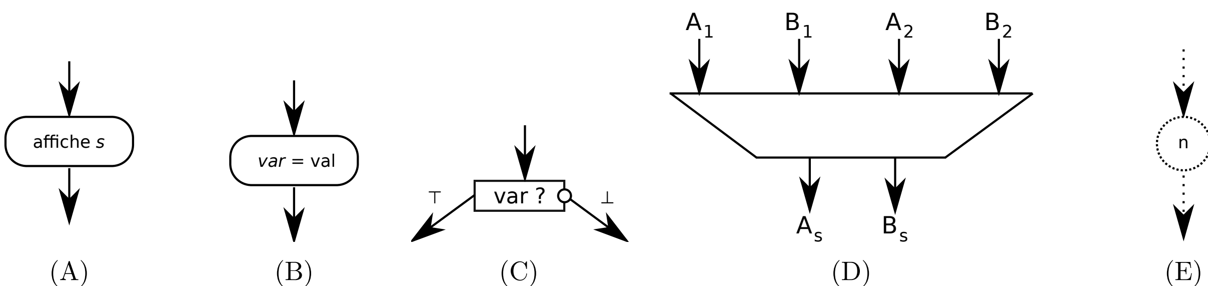


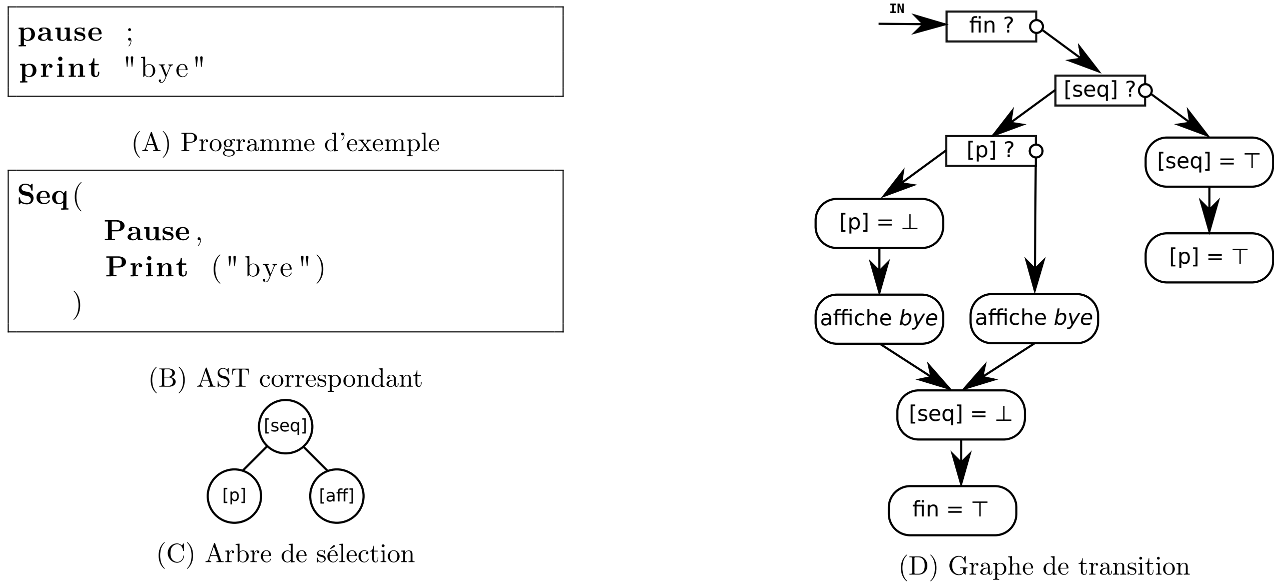
FIGURE 4: Les différents types de nœuds du format GRC

2.2 Un premier exemple

Nous considérons ici le programme 5A. Ce programme attend un instant puis affiche "bye". L'arbre de syntaxe abstraite est représenté sur la figure 5B.

De cet AST, on déduit l'arbre de sélection de la figure 5C. L'arbre de sélection est isomorphique à l'AST : le nœud *[seq]* pour l'instruction **Seq**, la feuille *[p]* (resp. *[aff]*) pour l'instruction **Pause** (resp. **Print**) dans

l'AST. En plus des variables portées par l'arbre de sélection on rajoute la variable *fin* pour distinguer l'état d'un programme qui n'a pas commencé et celui d'un programme qui a terminé son exécution.



(A) Programme d'exemple

(B) AST correspondant

(C) Arbre de sélection

(D) Graphe de transition

FIGURE 5: Exemple d'une transformation au format GRC

La figure 5D représente le graphe de transition du programme 5A tel qu'obtenu par l'algorithme de construction présenté dans cette section. Une trace d'interprétation de ce graphe est la suivante :

Premier instant

Le test *fin ?* renvoie faux et l'on active le nœud *[seq] ?* qui est faux aussi. On met donc *[seq]* et *[p]* à $\top$ dans l'arbre de sélection.

Deuxième instant

Le test *fin ?* renvoie toujours faux mais le nœud *[seq] ?* renvoie vrai et active le noeud *[p] ?*. *[p] ?* est vrai [1], on met donc *[p]* à $\bot$ et on affiche *bye*. La séquence est finie donc on met *[seq]* à $\bot$ puis *fin* à $\top$ car le programme termine.

2.3 Calcul du graphe de transition et de l'arbre de sélection

2.3.1 Graphe de profondeur et de surface

Le graphe de transition se calcule à l'aide de deux fonctions qui vont chacune calculer un sous-graphe à partir d'une instruction. La première fonction calcule le graphe de *surface* et la seconde calcule le graphe de *profondeur*.

Chacune des ces fonctions calcule un graphe comme celui de la figure 6 : un point d'entrée pour activer le bloc et deux sorties. Si le bloc n'est pas activé alors aucune des sorties ne le sera. Sinon, une unique sortie sera activée : la sortie **PAUSE** si l'instruction a été mise en pause, la sortie **FIN** si l'instruction a terminé.

Les seuls arcs entrants ou sortants depuis ou vers un nœud du bloc (autre que l'entrée et les deux sorties) sont ceux qui relient un nœud de dépendance à l'extérieur et un nœud interne. Ces nœuds n'activent donc pas de noeud interne. Ces arcs servent à représenter les contraintes d'ordonnancement entre les émissions et les réceptions d'un même signal.

Dans le *graphe de surface*, l'entrée est nommée **BOOT** et signifie que l'instruction démarre à l'instant courant. Dans le *graphe de profondeur* l'entrée est nommée **CONT** et signifie que l'instruction avait été mise en pause et qu'il faut la réactiver. Quand on réactive une instruction mise en pause, il est possible qu'on doive activer de nouvelles instructions (par exemple, si l'on réactive $a; b$ et que a termine, alors il faut activer b), le calcul du graphe de *profondeur* fait donc parfois appel au calcul du graphe de *surface*.

1. Tester $[p]$ sert à savoir dans quelle branche du **Seq** nous sommes. C'est inutile ici car la deuxième branche du **Seq** est instantanée et donc nous sommes forcément dans la première branche, au début d'un instant.

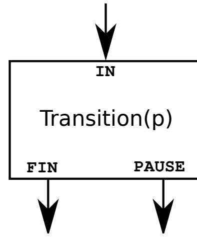


FIGURE 6: Bloc de profondeur ou de surface

2.3.2 Environnement

Pour représenter dans le graphe les dépendances entre les émissions et les réceptions de signaux, on crée un nœud de dépendance pour chaque signal et on ajoute un arc de toutes les émissions du signal vers ce nœud et de ce nœud de dépendance vers toutes les lectures du signal.

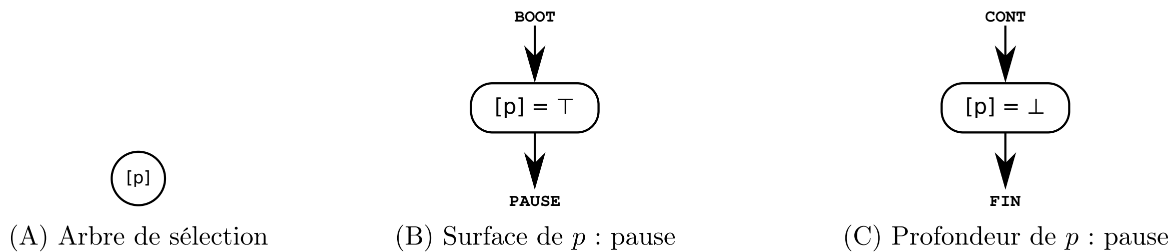
Pour cela, on calcule les graphes de surface et de profondeur avec un environnement qui associe à chaque signal un nœud de dépendance. Cet environnement est transmis et modifié dans le calcul récursif des graphes de surface et de profondeur (modifié car on ajoute un nœud à l'environnement lors d'une déclaration *signal s in q* pour le calcul du sous-graphe de q que l'on enlève ensuite).

On n'utilise pas de mémoization, si l'on a besoin à deux endroits différents du graphe de surface d'une même instruction q, la fonction graphe de surface sera appelée deux fois et les nœuds correspondants aux signaux définis à l'intérieur de q ne seront pas les mêmes. En effet, si l'on utilisait de la mémoization, dans le cas d'un programme avec schizophrénie, les nœuds représentant les signaux déclarés à l'intérieur d'une boucle seraient les mêmes dans le graphe de surface et dans le graphe de profondeur.

2.3.3 Pause

L'arbre de sélection d'une pause est une feuille qui contient une variable booléenne (l'exécution s'est-elle arrêtée sur ce *pause* ?). Pour le graphe de surface, donc lors de la première activation, on marque le nœud associé dans l'arbre de sélection ($[p]$) à $\top$ et on active les nœuds sur la sortie **PAUSE** (voir 7B).

Pour le graphe de profondeur, lors de la réactivation, la pause finit et donc on marque le nœud dans l'arbre de sélection à $\bot$ et on active les nœuds sur la sortie **FIN** (voir 7C).



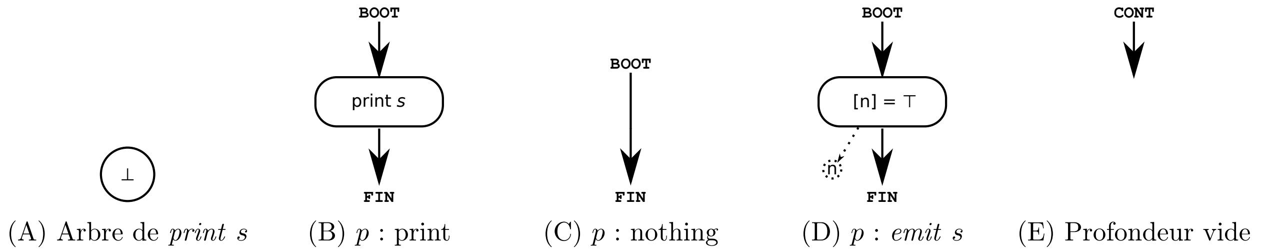
(A) Arbre de sélection (B) Surface de p : pause (C) Profondeur de p : pause

FIGURE 7: Graphes associés à *p : pause*

2.3.4 Instructions instantanées

Pour les instructions instantanées (print, nothing, emit), l'arbre de sélection associé est une feuille (comme sur la figure 8A) qui contient la valeur $\bot$ (l'exécution ne peut pas s'arrêter sur cette instruction car elle est instantanée). Le graphe de profondeur ne devrait jamais être activé car cela voudrait dire que l'instruction a été mise en pause. On génère donc un graphe vide (voir 8E). Pour les graphes de surface, voir 8C, 8D, 8B.

Pour l'instruction *emit s*, on rajoute un lien de dépendance entre le nœud de *emit* et le nœud n associé à s dans l'environnement (voir figure 8D).



(A) Arbre de *print s* (B) p : print (C) p : nothing (D) p : *emit s* (E) Profondeur vide

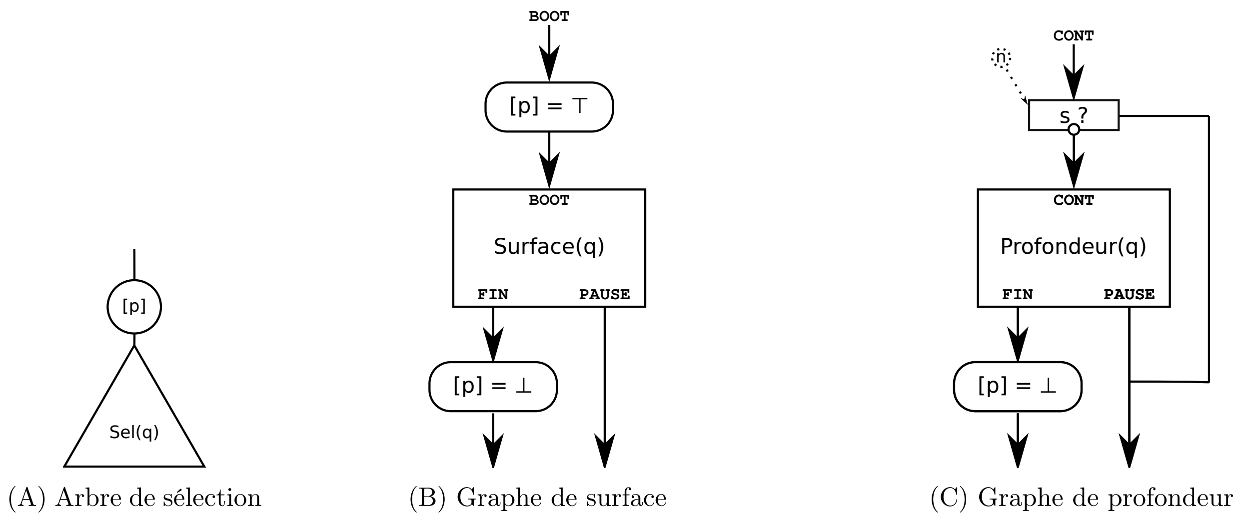
FIGURE 8: Graphes associés aux instructions instantanées

2.3.5 La suspension

La suspension *suspend q when s* est une instruction composée. Pour obtenir ses graphes (arbre de sélection, graphe de surface et de profondeur), il faut compiler son corps (en transmettant l'environnement courant). Pour l'arbre de sélection on ajoute une variable pour marquer si l'instruction est active ou non.

Le comportement d'une suspension est de toujours exécuter son corps à la première activation puis de le réactiver aux instants suivants selon la présence d'un signal de contrôle. On peut voir que le graphe de surface de q (voir 9B) est toujours activé tandis que dans le graphe de profondeur, q est activé après un test sur le signal s (ce qui impose une dépendance entre le nœud n associé à s dans l'environnement courant et ce nœud de test).

On peut remarquer que lors de la première activation l'instruction est marquée active ($[p] = \top$) et qu'elle ne sera marquée inactive ($[p] = \bot$) que lorsque le signal **FIN** sera activé.



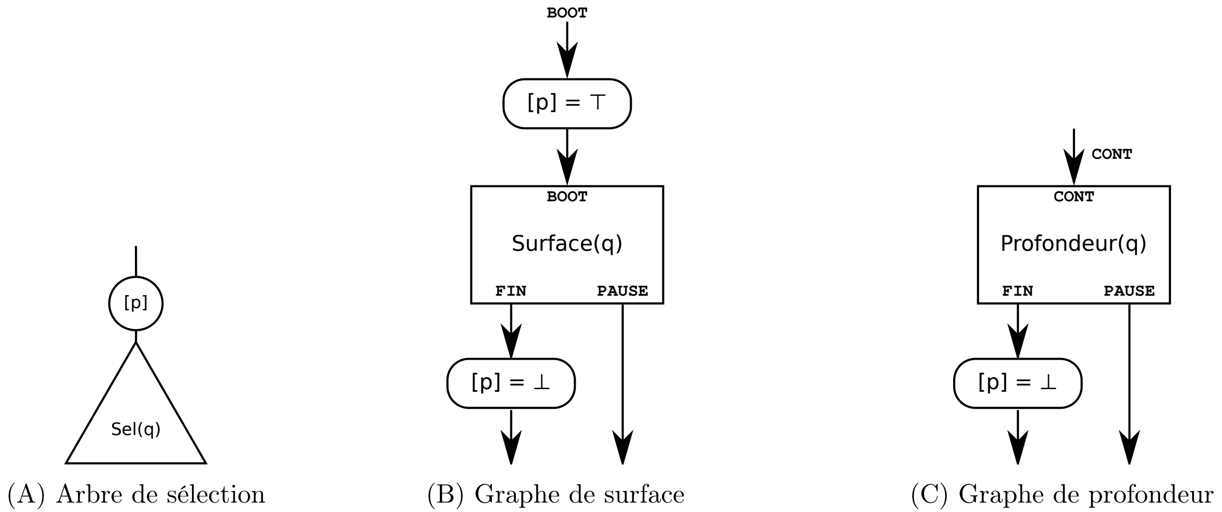
(A) Arbre de sélection (B) Graphe de surface (C) Graphe de profondeur

FIGURE 9: Graphes associés à *p : suspend q when s*

2.3.6 Déclaration de signaux

Pour la déclaration d'un signal, l'arbre de sélection est similaire à celui de la suspension.

Quand on calcule le graphe de surface de *signal s in q*, on crée un nœud de dépendance n, et l'association entre le signal s et n ne sera visible que pour le calcul du graphe de surface de q (ce sera un nœud différent pour le graphe de profondeur).



(A) Arbre de sélection (B) Graphe de surface (C) Graphe de profondeur

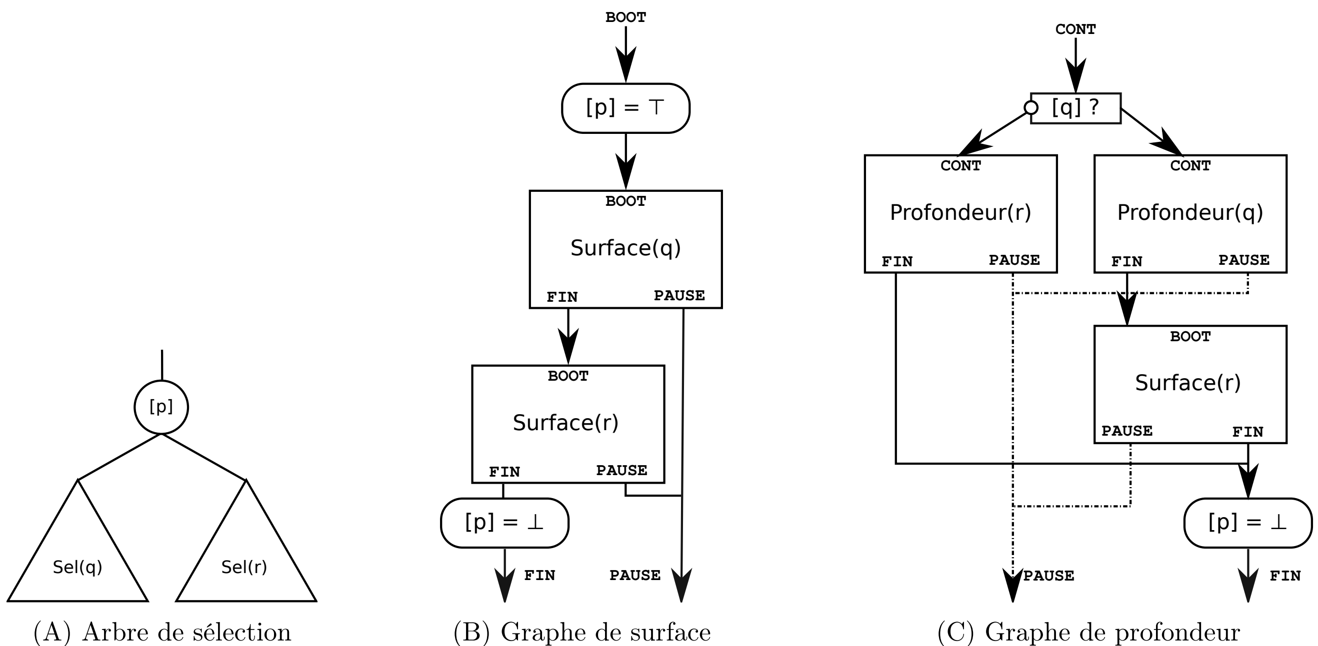
FIGURE 10: Graphes associés à *signal s in q*

2.3.7 Séquence

La séquence est un nœud qui a deux fils. Son arbre de sélection a donc aussi deux fils comme sur la figure 12A.

Dans la séquence $q; r$, dès que l'on finit l'exécution de q, on commence immédiatement celle de r. Cela se reflète dans les graphes de surface et de profondeur par le fait que la terminaison de q est immédiatement suivie par l'exécution du premier instant de r (sa surface).

Dès que l'on réactive la séquence un test sur le statut du nœud $[q]$ dans l'arbre de sélection est nécessaire pour savoir s'il faut reprendre l'exécution de q ou celle de r.

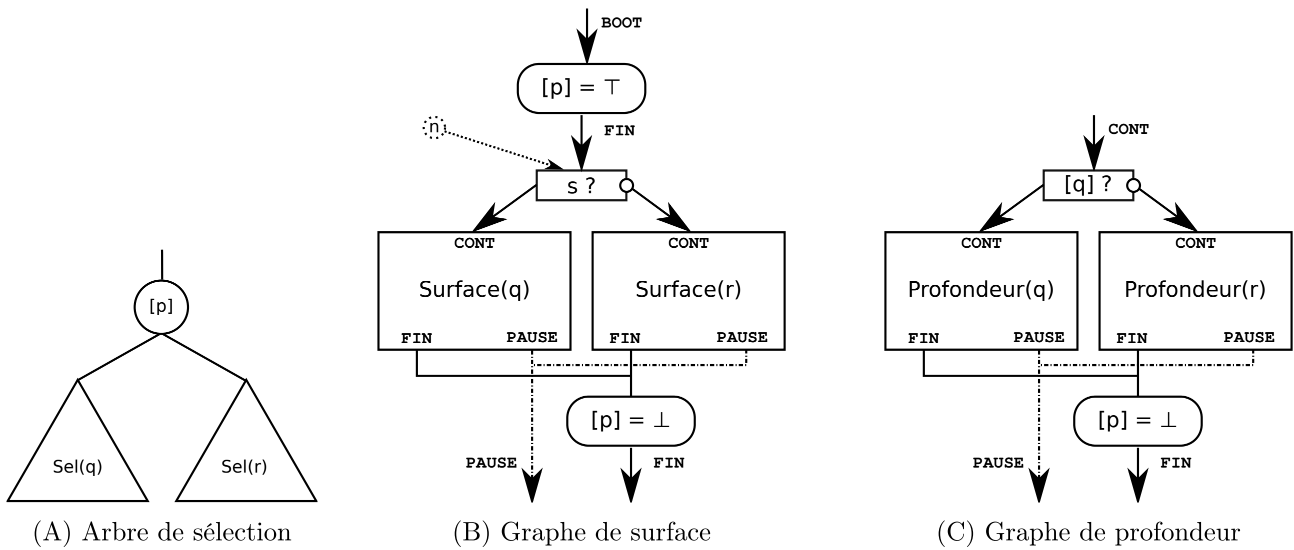


(A) Arbre de sélection (B) Graphe de surface (C) Graphe de profondeur

FIGURE 11: Graphes associés à la séquence $p : q; r$

2.3.8 Les tests de présence

Pour les tests de présence on utilise le nœud de test qui active une branche selon la valeur d'une variable. Ce nœud sert aussi ici à savoir si la première branche a été mise en pause (et donc la variable dans l'arbre de sélection est à $\top$) et donc si c'est la branche *then* qu'il faut réactiver. À l'instar du *emit*, on impose une dépendance entre le nœud associé au signal et le test mais dans l'autre sens cette fois-ci.



(A) Arbre de sélection (B) Graphe de surface (C) Graphe de profondeur

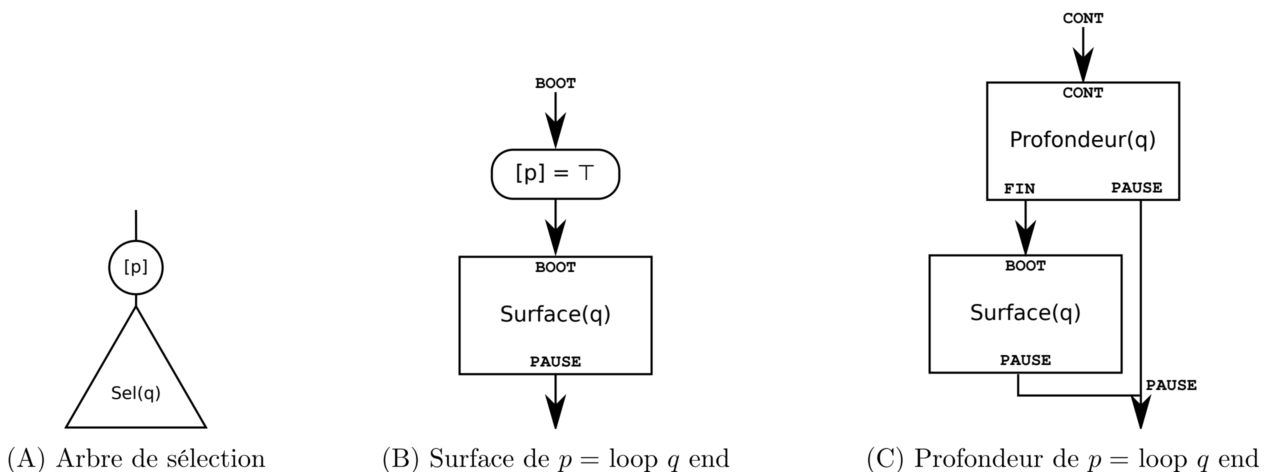
FIGURE 12: Graphes associés à p : present s then q else r

2.3.9 Boucles

Dans la compilation de *loop p end*, on peut remarquer dans le graphe de surface de q qu'il n'y a pas de sortie FIN. En effet, dans la sémantique d'ESTEREL, il est interdit que le corps d'une boucle soit instantané. Ainsi la même instruction ne peut pas commencer plusieurs fois pendant un instant et il n'est pas possible d'avoir de cette manière une boucle dans le graphe associé à un programme.

Pour la compilation des boucles, il faut faire attention à la schizophrénie. Lorsque l'on compile un programme comme celui de la présentation d'ESTEREL (le programme 3C), à la fin de la boucle, le test *present S* est effectué au même instant que le *emit s* du début de la boucle mais ils ne portent pas sur le même s, s ayant été redéfini avant le *emit*.

Ici, le problème est résolu pour le calcul de la profondeur parce que les signaux définis à l'intérieur q ne seront pas associés au même nœud dans le calcul de la surface et dans celui de la profondeur et donc ne partageront pas la même valeur de signal.

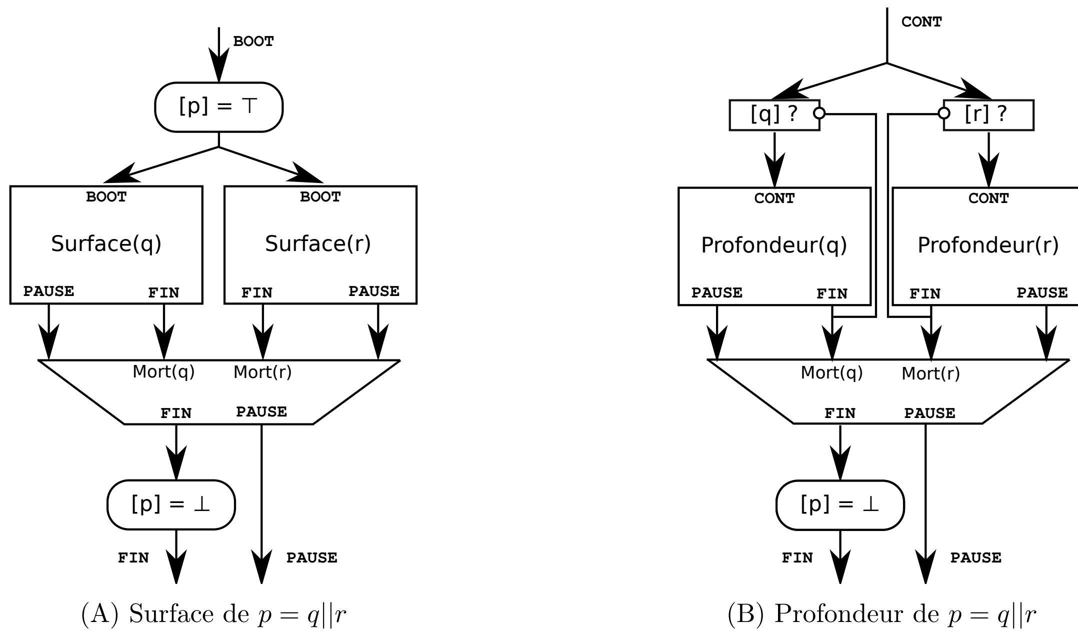


(A) Arbre de sélection (B) Surface de p = loop q end (C) Profondeur de p = loop q end

FIGURE 13: Graphes associés aux boucles

2.3.10 Le parallélisme

Pour le parallélisme, il faut utiliser le nœud *sync* (voir 4D) dont la sémantique est : si ses deux entrées *Mort* sont actives alors *sync* active FIN. Si une de ses deux autres entrées est activée, il active PAUSE. Cela correspond à la sémantique du parallèle : $a||b$ termine quand a et b ont terminé.



(A) Surface de $p = q||r$

(B) Profondeur de $p = q||r$

FIGURE 14: Graphes associés aux parallèles

2.3.11 Calcul du graphe de transition à partir des graphes de surface et de profondeur

Une fois que l'on a construit les deux graphes de surface et de profondeur, on peut les associer en un grand comme sur la figure 15. Ce graphe doit d'abord vérifier si le programme n'a pas terminé et, le cas échéant, il doit tester si l'on a déjà commencé pour savoir s'il faut activer le graphe de surface ou celui de profondeur.

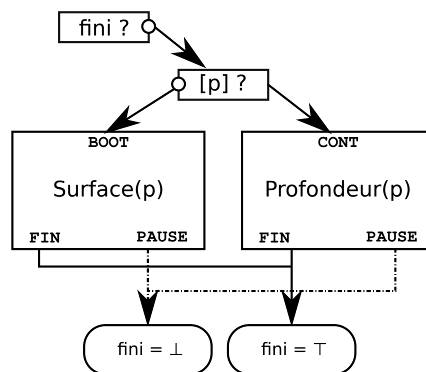


FIGURE 15: Schéma général

2.4 Compilation vers le langage cible

2.4.1 Un premier compilateur

À partir du format GRC, on veut produire un code séquentiel qui simule le graphe. Le graphe de transition étant connu à la compilation, on peut statiquement trier topologiquement les nœuds du graphe de transition et les numéroter de 0 à n. On numérote aussi les variables de signaux de 0 à s.

Au lancement du programme, le code généré marque toutes les variables (les variables de l'arbre de sélection et $fini$) à faux. À chaque instant, il commence par marquer les signaux comme absents et les nœuds du graphe de transition comme désactivés sauf le nœud d'entrée qui est activé. Puis, on parcourt les nœuds dans l'ordre topologique et à chaque nœud activé on exécute le code correspondant au type du nœud (tester une variable, modifier une variable, afficher une chaîne de caractères, etc.) puis on active certains nœuds fils.

2.4.2 Structure de données utilisée pour parcourir les nœuds activés

À chaque instant, le compilateur doit parcourir la liste des nœuds activés du graphe de transition. La structure de données utilisée est importante parce qu'à chaque instant seule une faible portion du graphe est activée.

Pour ne parcourir que les nœuds activés, tout en respectant l'ordre, on découpe le graphe en un nombre minimal m de niveaux de sorte qu'un nœud de niveau n ait tous ses ancêtres de niveau k avec $k < n$. Au début de chaque instant on crée m files d'attente ; à chaque fois que l'on active un nœud de niveau n, on l'ajoute à la file d'attente n s'il n'avait pas déjà été activé. Il suffit donc de parcourir les files d'attente dans l'ordre et pour chaque file d'attente parcourir tous les nœuds qu'elle contient.

Le découpage en niveau nous assure que si les nœuds ne sont pas parcourus dans l'ordre de la numérotation, ils sont bien parcourus dans un ordre topologique.

2.4.3 Quelques optimisations

Dans cette méthode de compilation de nombreuses astuces permettent de gagner beaucoup en temps et en mémoire, par exemple ou peut :

- retirer les nœuds qui ne sont pas accessibles dans le graphe ;
- supprimer les variables qui ne sont pas testées par le graphe de transition ainsi que les instructions qui modifient ces variables ;
- partager la mémoire des variables entre deux branches exclusives (comme dans une séquence ou dans un test de présence) ;
- éliminer des graphes de surface ou de profondeur après une analyse qui détermine si une instruction est nécessairement instantanée ou prend nécessairement plusieurs instants ;
- éliminer certaines variables de l'arbre de sélection qui s'expriment facilement en fonction d'autres (par exemple l'instruction *suspend q when s* est active si et seulement si q l'est).

2.4.4 Activer de nombreux nœuds en même temps

Dans le graphe de transition, de nombreux nœuds n'ont qu'un seul parent. Pour ces nœuds, il est inefficace que le parent marque le nœud en *activé*, et qu'il s'en occupe seulement plus tard. Il aurait été plus efficace de directement traiter ce nœud. La transformation que l'on veut appliquer c'est de remplacer, dans le code généré pour le nœud parent, l'activation du fils par directement le code du fils (et d'enlever ensuite le code du fils du reste du programme).

Pour appliquer cette méthode à tout le programme et ainsi profiter des autres optimisations, on commence par transformer le graphe en un graphe où les nœuds sont ceux qui n'ont pas qu'un seul arc entrant (dans le graphe original) et où les arcs relient a à b, si un nœud n aggloméré avec a a pour fils b.

Par l'exemple le sous-graphe de la figure 16A sera transformé en celui de la figure 16B. Le code correspondant généré sera transformé de 17A vers 17B.

(A) Graphe avant

(B) Graphe après

FIGURE 16: Optimisation du graphe

2.5 Limite de la méthode

La méthode présentée ici maintient entre deux instants un ensemble de points du programme activés. Cet ensemble est représenté par l'arbre de sélection. Par ailleurs, la duplication du code en graphe de surface

```
if actif.(42) then
begin
  print_string "a";
  activate 54
end

...

if actif.(54) then
begin
  print_string "b";
  activate 63
end
```

(A) code avant

```
if actif.(42) then
begin
  print_string "a";
  print_string "b";
  activate 63
end
```

(B) Code après

FIGURE 17: Optimisation du code

et graphe de profondeur permet de transformer les mémoires locales en mémoires globales [2].

Il est possible dans notre technique de simuler la modularité en remplaçant les appels de fonctions par le corps de celles-ci. Pour avoir de la vraie modularité, il faut être capable d'avoir un nombre arbitraire de processus qui s'exécutent concurremment et pour chaque processus de maintenir une mémoire dédiée qui stocke les variables locales. Par ailleurs le graphe de transition ne doit pas contenir autant de copies de chaque fonction qu'il y a d'appels à cette fonction dans le code.

Pour maintenir la mémoire utilisée pour chaque processus ainsi que le code qui doit être exécuté, la section 3 présente une méthode qui utilise une fonctionnalité du langage cible : les continuations.

3 Compilation avec continuations

3.1 Principe

Dans le compilateur vu à la section 2, le programme est transformé en un graphe qui est ensuite interprété. Au cours d'un instant, les nœuds de ce graphe ne peuvent pas être traités dans n'importe quel ordre : si un nœud a peut activer un nœud b alors on doit traiter a avant b. Si b lit la valeur d'un signal que a peut émettre alors a doit être traité avant b. On fait donc une analyse du graphe à la compilation, qui associe à chaque nœud un niveau, et on traite les nœuds par niveau croissant.

Dans cette version du compilateur, on reprend cette idée de niveau : les instants sont divisés en n niveaux. On voit ces niveaux comme des sous-instants et on introduit une forme de pause qui interrompt un processus pendant quelques sous-instants. Si une instruction a lit un signal qui est émis par b alors le niveau de b est inférieur strictement à celui de a réglant ainsi le problème des dépendances.

La compilation se déroule en plusieurs étapes : on construit un graphe qui représente les dépendances entre les différentes instructions (voir 3.2.1) ; on fait une analyse sur ce graphe pour attribuer un niveau à chaque instruction (voir 3.2.4), on compile le programme vers une fonction OCaml (voir 3.3) que l'on exécute dans un environnement qui définit plusieurs fonctions (voir 3.5).

3.2 Phase d'annotation

3.2.1 Graphe d'ordonnancement

La première étape de la phase d'annotation c'est de construire un graphe à partir de l'AST que l'on pourra analyser pour déduire les dépendances et attribuer un niveau à chaque instruction. La construction du graphe d'ordonnancement est inspirée de celle de [CPP$^+$02], cependant la méthode de construction est

2. Dans la version présentée, les seules mémoires sont les signaux booléens mais dans Esterel les signaux peuvent porter des valeurs et il y a aussi des variables.

similaire à celle du format GRC : on construit récursivement le graphe, on maintient un environnement de signaux pour imposer les dépendances entre émissions et réceptions de signaux, on compile chaque instruction vers un sous-graphe (où les seuls arcs entrants ou sortants sont pour imposer des dépendances de signaux) mais, à la différence de la construction du format GRC, on associe à chaque instruction un nœud d'entrée et un nœud de sortie qui sont aussi les nœuds d'entrée et de sortie du sous-graphe. Par ailleurs, on ne différencie pas entre surface et profondeur.

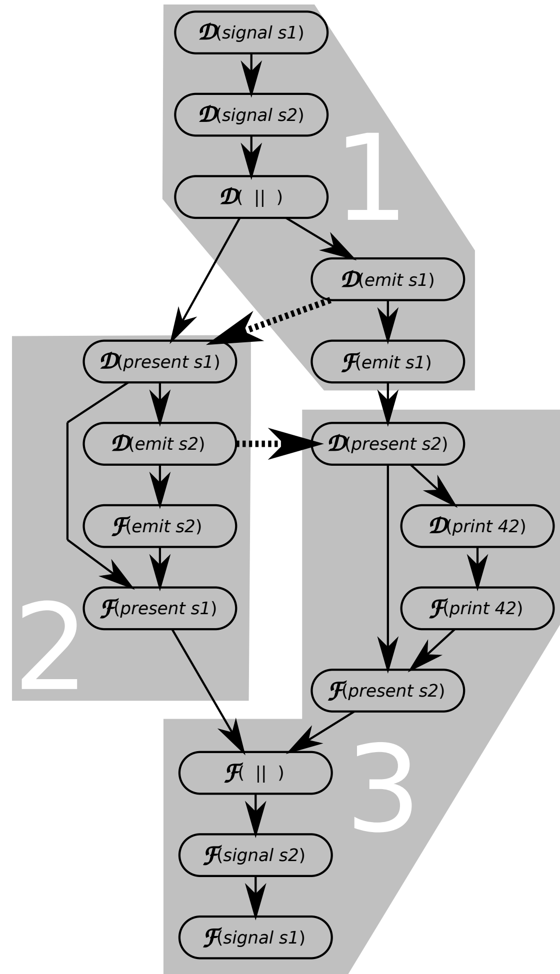
Le graphe est donc composé des nœuds de début et de fin des instructions. On note $\mathcal{D}(p)$ le nœud correspondant au début de l'instruction p et $\mathcal{F}(p)$ le nœud correspondant à sa fin. Entre deux nœuds a et b, il y a trois types d'arcs possibles : pour marquer la dépendance entre l'émission et la réception d'un signal (noté $a \overset{dep}{\to} b$), pour marquer qu'un nœud peut éventuellement activer un autre à l'instant courant (noté $a \overset{act}{\to} b$), pour marquer qu'un nœud peut activer un autre pour l'instant d'après (noté $a \overset{pau}{\to} b$).

3.2.2 Exemple

L'exemple est celui présenté en introduction (figure 3B) et est reproduit ci-dessous (figure 18A). Le graphe généré par la construction présentée ci-après est figure 18B.
Comme le programme comporte un couple émission / réception sur *s1* et un sur *s2*, il y a deux arcs de dépendances (les arcs $\overset{dep}{\to}$ sont en gros pointillés sur la figure, les arcs d'activation $\overset{act}{\to}$ sont représentés en petites flèches pleines).
Le graphe peut se décomposer en trois sous-graphes de niveau 1, 2 et 3. Cette décomposition en niveaux correspond à exécuter jusqu'au *emit* de la première branche (*emit* inclus) puis d'exécuter la seconde branche et de reprendre l'exécution de la première.

```
signal s1 in
signal s2 in
  present s1 then emit s2
||
  emit s1 ;
  present s2 then print "42"
end
end
```

(A) Programme de référence



(B) Graphe de dépendance

FIGURE 18: Compilation d'un programme simple

La construction du graphe se trouve en annexe **??**.

3.2.3 Contraintes que les arcs imposent sur les niveaux

Pendant un instant, le statut doit être lu nécessairement après la dernière émission. Il faut donc que, pour tous les arcs $a \overset{dep}{\to} b$, le niveau de a soit inférieur strictement au niveau de b.

Pendant l'exécution d'un instant, on traite les instructions par niveau croissant. Donc, si on a un arc $a \overset{act}{\to} b$ entre a et b, le niveau de a doit être inférieur au niveau de b.

Les arcs $a \overset{pau}{\rightarrow} b$ n'imposent aucune contrainte car a et b s'exécuteront sur deux instants différents.

3.2.4 Algorithme d'annotation

Quand les contraintes de niveaux entre les différents nœuds ne sont pas incohérentes, on peut trouver plusieurs solutions. Comme on le verra dans 3.5, le code généré fait des calculs pour chaque niveau et il fait une continuation quand il exécute les instructions qui correspondent à des arcs $a \overset{act}{\rightarrow} b$ avec a et b de niveaux différents. On essaye donc de trouver une annotation qui minimise le nombre de niveaux et le nombre d'arcs $\overset{act}{\rightarrow}$ reliant deux nœuds de niveaux différents.

Le problème que l'on essaye de résoudre c'est de trouver une solution à un ensemble d'inégalités linéaires et on souhaite en plus maximiser les cas d'égalité dans certaines inégalités. Comme nous n'avons pas connaissance d'un algorithme exact pour résoudre ce problème (et qu'une solution qui maximise les cas d'égalité n'est pas nécessairement plus efficace qu'une autre) la solution implémentée s'appuie sur des heuristiques pour maximiser les cas d'égalité.

On commence par appliquer un algorithme qui nous donne une première solution minimisant le nombre de niveaux (ici un Bellman-Ford).

Ensuite, on applique tant que possible une technique pour augmenter le nombre d'égalités : on partitionne le graphe en ensembles connexes pour les arcs $a \overset{act}{\rightarrow} b$ où tous les nœuds sont de même niveau. On cherche alors une composante pour laquelle on peut monter le niveau de ses éléments et atteindre ainsi une égalité dans une contrainte $\overset{act}{\rightarrow}$ sans l'atteindre pour une contrainte $\overset{dep}{\rightarrow}$.

3.3 Compilation des instructions annotées

3.3.1 Le langage cible

Les instructions sont compilées vers sous-ensemble d'OCaml où l'on suppose que plusieurs fonctions sont définies. Nous verrons comment définir les fonctions du langage dans la section 3.5. Ce sous-ensemble peut être vu comme un langage intermédiaire défini de la façon suivante :

⟨*expr*⟩ ::= '`IfThenElse`' (present ⟨*ident*⟩) ⟨*expr*⟩ ⟨*expr*⟩
  | ⟨*expr*⟩ '`;`' ⟨*expr*⟩
  | '`let`' ('`rec`'?) ⟨*ident*⟩ '`=`' ⟨*val*⟩ '`in`' ⟨*expr*⟩
  | '`emit`' ⟨*ident*⟩
  | '`print_string`' ⟨*string*⟩
  | '`instant_schedule`' ⟨*level*⟩ ⟨*expr*⟩
  | '`pause_schedule`' ⟨*level*⟩ ⟨*expr*⟩
  | ⟨*val*⟩ ()

⟨*val*⟩ ::= '`fresh_signal`'
  | '`fresh_join`' ⟨*expr*⟩
  | '`fun () ->`' ⟨*expr*⟩

Les ⟨*expr*⟩ sont des expressions OCaml de type `unit` : l'expression `instant_schedule l s` met s en pause jusqu'au niveau `l`. L'expression `pause_schedule l s` met `s` en pause jusqu'au niveau `l` de l'instant d'après ; le `present` dans `IfThenElse (present` s`) i1 i2` renvoie `i1` a été émis pendant l'instant, `i2` sinon, et donc `IfThenElse (present` s`) i1 i2` exécute `i1` si s a été émis et `i2` sinon). Enfin `emit` s marque le signal s comme émis.

Les ⟨*val*⟩ sont des fonctions, fresh_signal sert à déclarer une valeur signal qui sera donnée en paramètre des `emit` et des `present` ; `fun () ->` ⟨*expr*⟩ sert à définir une continuation ; `fresh_join` c renvoie une fonction qui ne fait rien la première fois et appelle c la seconde. 3.3.11.

3.3.2 Principe de la compilation

Le code généré pour une expression `p` doit terminer sans changer d'instant ou de niveau même si l'expression `p` dure plusieurs instants ou plusieurs niveaux pendant un instant. Par conséquent, la compilation d'une séquence de deux expressions $q; r$ ne peut pas être la séquence des codes générés $\mathcal{C}(q); \mathcal{C}(r)$ car si q dure plusieurs instants, r ne doit pas être démarré immédiatement.

Les expressions sont donc compilées en s'inspirant de la transformation CPS (*Continuation passing style*) : on compile récursivement les expressions de l'AST en leur passant en paramètre le code (*la continuation*) à exécuter une fois qu'elles terminent.

On note $\mathcal{C}(s, c)$ la fonction qui associe, à l'expression s et à la continuation c, le code généré. La continuation est déjà en langage intermédiaire et est de type $\langle expr \rangle$.

On maintient l'invariant que le code généré pour une expression p est exécuté au niveau du nœud correspond au début de cette expression p (le nœud $\mathcal{D}(p)$ dans le graphe, noté ici p_d) et que cette expression doit terminer au niveau du nœud de fin de cette expression (le niveau du nœud $\mathcal{F}(p)$ dans le graphe, noté p_f) et qui doit aussi être le niveau du début de la continuation c.

Si l'instruction fait référence à d'autres instructions comme q (resp. r) on note q_d et q_f le niveau dans lequel on commence q et finit q (resp. r_d et r_f).

3.3.3 nothing

L'instruction *nothing* commence au niveau n_d et finit au niveau n_f, ne fait rien puis appelle sa continuation. Il suffit donc de faire un pause de quelques niveaux (éventuellement un nombre nul de niveaux) :

$\mathcal{C}(nothing, c) =$ `fun () -> instant_schedule` n_f c

3.3.4 print s

L'instruction *print s* affiche la chaîne s puis, comme dans le cas de *nothing*, fait une pause de quelques niveaux avant d'appeler sa continuation :

$\mathcal{C}(print\ s, c) =$ `fun () -> (print_string s ; instant_schedule` n_f c`)`

3.3.5 emit s

De la même manière que pour les instructions précédentes, pour l'instruction *emit s*, on a :

$\mathcal{C}(emit\ s, c) =$ `fun () -> (emit sig_`s `; instant_schedule` n_f c`)`

3.3.6 pause

La *pause* ne fait qu'appeler sa continuation au niveau n_f de l'instant d'après, on a donc :

$\mathcal{C}(pause, c) =$ `fun () -> (pause_schedule` n_f c`)`

3.3.7 signal s in q

Dans la déclaration de signal, on déclare le signal immédiatement puis on attend un certain nombre de niveaux avant de commencer l'exécution de q. Comme on recrée un nouveau signal à chaque fois que l'on rencontre une instruction `signal` s `in` q, il n'y a pas de problème de schizophrénie. On a :

$\mathcal{C}(signal\ s\ in\ q, c) =$ `(fun () ->`
```
    let sig_s = fresh_signal () in
        instant_schedule q_d C(q, fun () -> instant_schedule n_f c))
```

3.3.8 a ; b

Pour la séquence, on commence par définir le code pour exécuter b qui se termine en exécutant c (après une éventuelle pause de quelques niveaux). Ensuite on compile le terme a qui appelle la fonction de b quand il finit.

$\mathcal{C}((a; b), c) =$ `(fun () ->`
```
    let instr_n = C(b,instant_schedule n_f c) in
        instant_schedule a_d C(a,fun () -> instant_schedule b_d instr_n)
```

Le code pour b est encapsulé dans un `let` car le code de la continuation est parfois dupliqué. Le nom `instr_n` doit être un nom frais pour ne pas masquer un autre identifiant.

3.3.9 present s then q else r

Par construction des niveaux, l'instruction *present* est appelée à un niveau où le statut de s est déjà déterminé. Il faut donc éventuellement attendre quelques niveaux pour les fils mais le test `present` s peut être fait immédiatement :

$\mathcal{C}(present\ s\ then\ q\ else\ r, c) =$

```
IfThenElse(present sig_s,
    instant_schedule q_d (fun () -> instant_schedule n_f C(q,c)),
    instant_schedule r_d (fun () -> instant_schedule n_f C(r,c)))
```

3.3.10 loop q end

Pour la boucle, il faut attendre du niveau n_d à q_d puis exécuter le corps de q puis attendre du niveau q_f à q_d et recommencer à l'étape *exécuter le corps de la boucle.* Grâce à la présence de la récursion, on peut écrire cela directement (le `n` dans `instr_n` est encore à remplacer par un nom frais) :

$\mathcal{C}(loop\ q\ end, c) =$

```
let rec instr_n = fun () ->
    C(q,instant_schedule q_d instr_n)
in instant_schedule q_d instr_n
```

3.3.11 $q \mid\mid r$

Il faut exécuter la continuation c du parallèle quand $q||r$ a terminé, c'est à dire quand les deux sous expressions q et r ont terminé. Pour cela, `fresh_join` c renvoie une fonction qui termine immédiatement la première fois qu'elle est appelée et appelle c la seconde fois (le `instr_n` est encore à remplacer par un nom frais).

$\mathcal{C}(q||r, c) =$

```
let join_n = fresh_join c in
    instant_schedule q_d C(q,fun () -> instant_schedule n_f join_n);
    instant_schedule r_d C(r,fun () -> instant_schedule n_f join_n)
```

3.4 Exemple

On reprend le code d'exemple de la section 2 (répété figure 19A). On suppose que la phase d'annotation a été faite et qu'il n'y a qu'un seul niveau, nommé 0. Le code compilé associé est présenté dans la figure 19B. Le code de la figure 19B est obtenu à partir du code `Seq(Pause,print "bye")` traduit par la fonction $\mathcal{C}$ avec la continuation `exit` (une fonction que l'on suppose définie).

```
pause ;
print "bye"
```

(A) Programme de référence

```
fun () ->
  let instr_1 = fun () ->
    print_string "bye" ;
    instant_schedule 0 exit
  in
    instant_schedule 0
              (fun () -> pause_schedule 0 instr_1)
```

(B) Compilation en langage intermédiaire

FIGURE 19: Compilation d'un programme simple

On commence par compiler `Seq`. `Seq` compile `print "bye"` en `fun () -> print "bye"` puis on compile `Pause`, en disant qu'une fois qu'il a terminé il appelle `instr_1`.

3.5 Environnement d'exécution

L'environnement d'exécution consiste en plusieurs variables globales, en un mécanisme qui va à chaque instant activer toutes les fonctions de tous les niveaux et enfin en plusieurs fonctions présentées plus haut (section 3.3). Par ailleurs, pour que le code généré s'arrête quand le programme ESTEREL termine, on compile celui-ci avec comme continuation la fonction `exit` définie figure 20B.

3.5.1 Variables globales

L'environnement maintient plusieurs variables : `nbNiveaux` qui stocke le nombre de niveaux, on numérote les instants avec la variable `curInstant`, `nowDo` est un tableau de taille `nbNiveaux` qui stocke à la i-ème case la liste des continuations à exécuter pendant le niveau i de l'instant courant. De la même manière, la i-ème case de `pauseDo` stocke la liste des continuations à exécuter pendant le i-ème niveau de l'instant d'après.

3.5.2 Boucle principale

La fonction `run_all` correspond à la boucle principale d'appel. Elle boucle sur les instants et à chaque fois appelle toutes les continuations de chaque niveau ; elle prépare ensuite le niveau suivant en mettant à jour `curInstant` et en déplaçant les continuations en attente de `pauseDo` vers `nowDo`. `run_all` est présentée figure 20A.

```
let run_all () =
  while (true) do
    for i = 0 to nbNiveaux-1 do
      while nowDo.(i) <> [] do
        let head::tail = nowDo.(i) in
        nowDo.(i) <- tail ;
        head ()
      done
    done
    for i = 0 to pred nbNiveaux do
      nowDo.(i) <- pauseDo.(i) ;
      pauseDo.(i) <- ()
    done ;
    incr curInstant
  done
```

(A) Fonction `run_all`

```
let exit = fun () -> exit 0
```

(B) Fonction `exit`

```
let pause_schedule l f =
  pauseDo.(l) <- f::pauseDo.(l)
```

(C) Fonction `pause_schedule`

```
let instant_schedule l f =
  nowDo.(l) <- f::nowDo.(l)
```

(D) Fonction `instant_schedule`

```
let fresh_join f =
  let wait = ref true in
  fun () ->
    if !wait
      then wait := false
      else f ()
```

(E) Fonction `fresh_join`

```
let fresh_signal () = ref (-1)
```

(F) Fonction `fresh_signal`

```
let present signal =
  (!signal) == !curInstant
```

(G) Fonction `present`

```
let emit signal =
  signal := !curInstant
```

(H) Fonction `emit`

FIGURE 20: Environnement d'exécution

3.5.3 Pauses

Pour les pauses, il suffit d'ajouter la continuation dans la bonne liste. Les fonctions `instant_schedule` et `pause_schedule` sont présentées figures 20D et 20C. Pendant l'exécution d'un programme, on perd beaucoup de temps à exécuter des `instant_schedule l f` : il faut créer une continuation, la mettre dans

un tableau, la rappeler ensuite et glaner la mémoire correspondante. À la compilation on sait déjà à quel niveau chaque instruction va être exécutée. Ainsi, quand un `instant_schedule l f` est exécuté pendant le niveau `l`, on le remplace par un appel direct (`f ()`). La continuation est toujours crée mais on évite de la stocker dans un tableau et le compilateur OCaml pourra ensuite optimiser ces appels. Il est donc très intéressant d'optimiser le nombre de ces simplifications, c'est pourquoi on maximise dans l'algorithme d'annotation le nombre d'arcs $a \overset{act}{\to} b$ avec a et b de même niveau.

3.5.4 Gestion des signaux

Les signaux sont codés par une référence qui contient le numéro du dernier instant où le signal a été émis. Émettre un signal revient à modifier cette référence pour qu'elle contienne le numéro de l'instant courant et le test de présence est simplement une comparaison avec l'instant courant. Les fonctions `fresh_signal`, `present` et `emit` sont présentées aux figures 20F, 20G et 20H.

4 Compilation de ReactiveML

Jusqu'à présent nous avons présenté la compilation d'Esterel. L'objectif de ce stage étant de pouvoir compiler le langage ReactiveML cette section présente comment intégrer certaines des constructions de ReactiveML et quelles sont les pistes ou les difficultés pour intégrer les autres.

4.1 Expression OCaml pures

On compile toutes les expressions OCaml pures — celles qui ne contiennent pas de `emit`, de `present` et qui ne renvoient pas de signaux — en les traduisant d'un seul tenant car la transformation CPS peut produire du code moins efficace. Si e est une expression OCaml pure on a :
$\mathcal{C}(e,c)$ `= fun () ->` e `() ; instant_schedule` e_f c.

4.2 Valeurs (`let` a `=` v `in` b, expressions)

Pour récupérer les valeurs de façon à ce que celles-ci puissent être calculées sur plusieurs instants, ou plusieurs niveaux, il faut transmettre les valeurs à la continuation (comme dans la transformation CPS normale). On a :
$\mathcal{C}($`let` a `=` v `in` $b,c) = \mathcal{C}(v,$`fun` a `->` $\mathcal{C}(b,c))$

Pour compiler l'expression `let` v `= present s then "present" else "absent" in print` v avec la continuation c, on commence par calculer $\mathcal{C}($`present` s `then "present" else "absent", fun` v `->` $\mathcal{C}($`print` v`,`$c))$. Cela donne, en éliminant les indications de niveaux, le programme :

```
let instr_0 =
  fun v -> c (print_string v)
in
if(present s)
then (instr_0 "present")
else (instr_0 "absent")
```

4.3 Structures de contrôle (`if`, `for`, ...)

La compilation de chaque structure de contrôle nécessite un traitement particulier mais ne pose pas de difficulté majeure. Par exemple, pour compiler `if` c `then` a `else` b, on ajoute dans le graphe d'annotation les arcs : $\mathcal{D}(if) \overset{act}{\to} \mathcal{D}(c)$, $\mathcal{F}(c) \overset{act}{\to} \mathcal{D}(a)$, $\mathcal{F}(c) \overset{act}{\to} \mathcal{D}(b)$, $\mathcal{F}(a) \overset{act}{\to} \mathcal{F}(if)$, $\mathcal{F}(b) \overset{act}{\to} \mathcal{F}(if)$.

Pour la partie compilation, on a :
$\mathcal{C}($`if` $cond$ `then` a `else` $b,c) =$
$\mathcal{C}(cond,$`fun` val_{cond} `-> ifThenElse(`val_{cond}`,`$\mathcal{C}(a,c)$`,`$\mathcal{C}(b,c)$`))`

4.4 Gestion de la suspension

En ReactiveML, chaque processus peut être préempté par différents signaux. Contrairement à la compilation par le format GRC où l'on reparcourait tout l'arbre de syntaxe pour aller au processus à exécuter, ici, les processus sont activés directement.

Pour ajouter la suspension, il faut donc maintenir dynamiquement pour chaque processus la liste des signaux qui peuvent le préempter. On maintient une variable globale `preemption`, à chaque fois que l'on dépile une continuation de `nowDo`, on dépile la liste S des signaux qui la préempte et on met cette liste dans la variable `preemption`. Si un des signaux n'est pas émis [3], on remet la continuation dans `pauseDo`, sinon on exécute normalement. Quand on commence l'exécution d'un `suspend` q `when` s, on ajoute s à la cette liste et on le retire s quand q. On modifie `instant_schedule` et `pause_schedule`, pour stocker la liste `preemption` en même temps que la continuation.

Enfin, pour gérer la suspension, il faut faire en sorte qu'un processus ne soit activé qu'après que le statut des signaux qui peuvent le préempter ont été fixés. Aussi il faut maintenir dans le graphe d'annotation les dépendances entre les émissions de s et les instructions qui peuvent être potentiellement préemptées par s.

4.5 Ordre supérieur et signaux comme valeurs de première classe

Dans toute la méthode présentée plus haut, on supposait que les signaux étaient définis par des `signal` s `in` et qu'ils n'étaient jamais copiés. Comment traiter des signaux qui deviendraient des éléments de première classe ou qui peuvent être cachés des fermetures (comme dans la figure 22) sans qu'une telle analyse ne refuse trop de programmes ?

```
signal s in
let f () = emit s in
...
```

Figure 22: Exemple de masquage d'un signal

Sur cette question, plusieurs pistes ont été explorées en s'inspirant principalement des analyses sur les exceptions mais aucune n'a abouti. En effet, si les signaux peuvent être copiés, ils peuvent stockés par des références, encapsulés dans des fermetures, ...

Cette tâche est similaire à l'analyse d'exception (qui est un problème déjà bien étudié) mais, ici, nous avons une notion d'instant et du parallélisme ce qui complique l'analyse, nous avons aussi des besoins plus fins en analyse (les faux positifs bloquent la compilation), et nous avons aussi besoin de connaître le point d'émission des signaux.

4.6 Compilation hybride

Pour profiter de l'exhaustivité de l'ordonnancement dynamique du compilateur actuel de ReactiveML et de la performance de notre ordonnancement statique, il est possible de mélanger les deux techniques. En effet, si l'on arrive à isoler des parties que l'on peut ordonnancer statiquement, il est possible de les compiler de cette manière et de laisser le reste compilé de manière dynamique. Par exemple, si une fonction ne fait qu'émettre, elle peut être statiquement ordonnancée au début. Inversement, si elle ne fait que lire des signaux (et pas en émettre), elle peut être ordonnancée à la fin.

Conclusion

En raison des contraintes de longueur, on ne répète pas ici les conclusions du rapport mais le lecteur est invité à consulter le bilan de la fiche de synthèse et, pour plus de détails, l'annexe A sur les performances du compilateur.

3. En ReactiveML, la suspension est inversée par rapport à Esterel, l'expression `q` dans `do q when s` s'exécute quand `s` est présent.

Références

[BC84] G. Berry and L. Cosserat. The synchronous programming language Esterel and its mathematical semantics. *Lecture Notes in Computer Science*, 1984.

[Ber99] G. Berry. The constructive semantics of pure esterel, 1999.

[Ber13] G. Berry. Chaire algorithme, machines et langages du collège de france : « l'informatique du temps et des événements », 2013.

[Bou91] Frédéric Boussinot. Reactive C : An extension of C to program reactive systems. *Software Practice and Experience*, 21(4) :401–428, April 1991.

[CPP+02] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. Saxo–rt : Interpreting esterel semantic on a sequential execution structure. *Electronic Notes in Theoretical Computer Science*, 65(5) :80 – 94, 2002. SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of {ETAPS} 2002).

[Edw94] Stephen Edwards. An Esterel Compiler for a Synchronous/Reactive Development System. 1994.

[GR02] Vinod Ganapathy and S. Ramesh. Slicing synchronous reactive programs, 2002.

[MP08] Louis Mandel and Marc Pouzet. ReactiveML : un langage fonctionnel pour la programmation réactive. *Technique et Science Informatiques (TSI)*, 27(9–10/2008) :1097–1128, 2008.

[PB02] Dumitru Potop-Butucaru. *Generation of Fast C Code from Esterel Programs*. Thèse de doctorat, ENSMP/CMA, 2002.

[PBEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. 2007.

A Performances

Les tests ont été effectués sur une machine Linux avec deux cœurs cadencés à 2.00GHz. Les codes produits par le compilateur natif et notre compilateur ont été compilés vers du code machine avec `ocamlopt` (sans option particulière) et tous les tests ont été moyennés sur 5 exécutions.

Le premier programme testé est celui de l'introduction mais modifié pour se répéter dix millions de fois. Le code testé est reproduit figure 23A et le code généré pour ce programme est reproduit figure 26.

Le code généré par notre compilateur (évidemment sans optimisation manuelle) a mis 4.65 secondes à terminer tandis que le compilateur natif a pris 11.52 secondes, le code produit par notre compilateur tourne donc 2.5 fois plus rapidement !

```
let aug =
  let c = ref 0 in
(fun () -> incr c ;
 if !c=10000000 then (exit 0))

let process main =
  signal s1 in
  signal s2 in
  loop
    pause;
   (present s1 then emit s2
  ||
    emit s1 ;
    present s2 then aug());
  end
```

(A) Exemple d'introduction

Ce résultat (et ceux présentés ci-dessous) est très encourageant mais il est à nuancer : les performances seront négativement impactées par l'ajout de certaines fonctionnalités (par exemple l'ajout de la suspension décrit dans 4.4) même si, inversement, l'amélioration de la phase de génération de code et l'optimisation de l'environnement peuvent compenser en partie et permettre de générer du code plus rapide.

Ce gain dépend aussi énormément du type de programme à compiler : le gain de vitesse sur un programme déjà bien ordonnancé sera sensiblement moindre que sur un programme mal ordonnancé. Le tableau 24 compare les performances des codes générés par le compilateur ReactiveML sur les programmes 23A, 25A et 25B et les performances de notre compilateur.

Programme	bien ordonnancé (25A)	mal ordonnancé (25B)	programme 23A
Avec ordonnancement statique	4.42	4.37	4.65
Compilateur natif	7.1 (+60%)	9.56 (+120%)	11.52 (+150%)

Figure 24: Tableau de performance

```
let process main =
  let c = ref 0 in
  loop
    incr c;
    if !c = 10000000
    then exit 0 ;
    signal s in
      present s then pause
    ||
      (emit s ; pause)
  end
```

(A) Exemple avec schizophrénie, ordre inversé

```
let process main =
  let c = ref 0 in
  loop
    incr c;
    if !c = 10000000
    then exit 0 ;
    signal s in
      (emit s ; pause)
    ||
      present s then pause
  end
```

(B) Exemple avec schizophrénie, dans le bon ordre

```
let aug = (*OCaml pure*)
  let c = ref 0 in
 fun () -> (incr c ;if !c = 10000000 then (exit 0))
let main =
  let s1 = new_signal (()) in (* end of s1*)
  let s2 = new_signal (()) in (* end of s2*)
  let rec code_19 = function |() ->
    let code_5 = function |() ->
      let code_6 = function |() ->
        let code_7 = function |() ->
          let par_count7 = Pervasives.ref (2)  in (* end of par_count7*)
          let join7 = function |() ->
            if (Pervasives.(==) (Pervasives.(!) (par_count7) ) (1) )
            then (code_19 () )
            else (Pervasives.(:=) (par_count7)
                    (Pervasives.(-) (Pervasives.(!) (par_count7) ) (1) ))
          in (* end of join7*)
          let code_9 = function |() ->
            let code_10 = function |() ->
              stack2 join7 (()) in (* end of code_10*)
            let code_11 = function
              |() -> (emit (s2) ; code_10() )
            in (* end of code_11*)
            let code_12 = function
              |() ->  code_10()
            in (* end of code_12*)
            if (is_present (s1) )
            then (code_11 (()) )
            else (pause_stack0 (code_12))
          in (* end of code_9*)
          let code_18 = function
            |() -> let code_13 = function
              |() -> (emit (s1) ;
                      let code_14 = function |() ->
                        let code_15 = function |() ->
                          join7 (())
                        in (* end of code_15*)
                        let code_16 = function |() ->
                          (aug ()) ; code_15()
                        in (* end of code_16*)
                        let code_17 = function |() ->
                          code_15 ()
                        in (* end of code_17*)
                        if (is_present (s2) )
                        then (code_16 (()) )
                        else (pause_stack2 (code_17))
                      in (* end of code_14*)stack2 (code_14) )
                   in (* end of code_13*)code_13 (())
          in (* end of code_18*)(stack1 (code_9) ;code_18 (()) )
        in (* end of code_7*)code_7 (())
      in (* end of code_6*)pause_stack0 (code_6)
    in (* end of code_5*)code_5 (())
  in (* end of code_19*)stack2 (code_19)
```

Figure 26: Code généré par notre compilateur pour l'exemple

B Construction

Voici les arcs créés pour chaque instruction :

— p : *nothing* on crée un arc $\mathcal{D}(p) \overset{act}{\to} \mathcal{F}(p)$, l'instruction étant immédiate, elle finit à l'instant où elle commence.

— p : *print* s on crée un arc $\mathcal{D}(p) \overset{act}{\to} \mathcal{F}(p)$.

— p : *emit* s on crée un arc $\mathcal{D}(p) \overset{act}{\to} \mathcal{F}(p)$ et un arc $\mathcal{F}(p) \overset{dep}{\to} \mathcal{D}(b)$ pour toutes les instructions b qui lisent le signal s.

— p : *pause* p termine un instant après que p commence donc on crée un arc $\mathcal{D}(p) \overset{pau}{\to} \mathcal{F}(p)$

— p : *signal* s *in* q on crée un arc $\mathcal{D}(p) \overset{act}{\to} \mathcal{D}(q)$ et un arc $\mathcal{F}(q) \overset{act}{\to} \mathcal{F}(p)$, à l'instant où p commence alors q commence et quand q termine, alors p.

— p : *present* s *then* q *else* r on crée les arcs $\mathcal{D}(p) \overset{act}{\to} \mathcal{D}(q)$ et $\mathcal{D}(p) \overset{pau}{\to} \mathcal{D}(r)$ pour quand p commence les arcs $\mathcal{F}(q) \overset{act}{\to} \mathcal{F}(p)$ et $\mathcal{F}(r) \overset{act}{\to} \mathcal{F}(p)$ pour quand p termine. On a aussi un arc $\mathcal{F}(b) \overset{dep}{\to} \mathcal{D}(p)$ pour tous les b qui lisent s.

— p : q; r, q commence quand p commence ; r commence quand q termine, et p termine quand r termine. On a donc les arcs : $\mathcal{D}(p) \overset{act}{\to} \mathcal{D}(q)$, $\mathcal{F}(q) \overset{act}{\to} \mathcal{D}(r)$ et $\mathcal{F}(r) \overset{act}{\to} \mathcal{F}(p)$.

— p : q // r, quand p commence, r et q commencent aussi, et p peut terminer après que q ou r termine, les arcs sont donc : $\mathcal{D}(p) \overset{act}{\to} \mathcal{D}(q)$, $\mathcal{D}(p) \overset{act}{\to} \mathcal{D}(r)$, $\mathcal{F}(q) \overset{act}{\to} \mathcal{F}(p)$ et $\mathcal{F}(r) \overset{act}{\to} \mathcal{F}(p)$.

— p : *loop* q *end*, q commence quand p commence ou quand q termine donc les arcs sont : $\mathcal{D}(p) \overset{act}{\to} \mathcal{D}(q)$, $\mathcal{F}(q) \overset{act}{\to} \mathcal{D}(q)$