Université de Nice – Sophia Antipolis
UFR Sciences

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

Thèse de Doctorat

Préparée pour obtenir le titre de
Docteur en Sciences de l'Université de Nice-Sophia Antipolis

Spécialité : Informatique

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

par

Eric VECCHIÉ

Calcul des états atteignables de programmes ESTEREL partitionné selon la syntaxe

Directeur de thèse : Robert DE SIMONE

Soutenue publiquement le 9 juillet 2004
à l'Ecole Supérieure en Sciences Informatiques
de Sophia Antipolis

devant le jury composé de :

MM.	Jean-Paul RIGAULT	Président	UNSA / ESSI
	Jean-Michel COUVREUR	Rapporteur	CNRS / ENS Cachan
	Nicolas HALBWACHS	Rapporteur	CNRS / Vérimag
Mme.	Dominique BORRIONE	Examinatrice	TIMA / UJF Grenoble
MM.	Pascal RAYMOND	Examinateur	CNRS / Vérimag
	Robert DE SIMONE	Directeur de thèse	INRIA Sophia Antipolis

Table des matières

Remerciements

Je remercie le monde entier pour son soutien.

Thèse

Chapitre 1

Introduction

La raison de vivre d'une porte automatique est de s'ouvrir à l'approche d'un passant. De ce fait, on comprendra aisément le désarroi du chaland qui, probablement très absorbé par la relecture des dernières corrections apportées à son manuscrit de thèse par son directeur qui écrit aussi petit que mal, et si possible dans une langue que seuls les autres directeurs de thèse comprennent, entre en relation plus qu'amicale avec quatre mètres carrés de plexiglas en tentant de franchir une porte caractérielle qui n'a pas souhaité s'ouvrir à son passage parce-que... parce-que c'est son circuit de contrôle qui lui a pas dit. Ce sentiment hostile envers la technologie est d'autant plus justifié si une porte d'ascenseur s'ouvre sur le vide, si l'airbag d'une voiture se déclenche sans raison sur l'autoroute ou si le train d'un avion refuse de sortir de sa trappe au moment de l'atterrissage.
Le premier point commun entre tous ces exemples est que ce sont tous des systèmes où le contrôle (en opposition aux données) tient une place prépondérante. Le langage ESTEREL [15, 12] a été conçu pour modéliser et programmer ce type d'application. Le second point commun est que tous ces exemples sont des applications critiques qui peuvent mettre des vies en danger en cas de dysfonctionnement. Le dysfonctionnement en question peut provenir soit d'une panne de l'un des composants soit d'une erreur de conception. Le contenu de cette thèse se révélera impuissant devant le premier cas. Dans le second cas, des méthodes automatiques permettent de vérifier la correction d'une application vis à vis de critères formels. Par exemple, un critère de fonctionnement correct d'un avion pourrait être de ne jamais rentrer le train d'atterrissage lorsque l'avion est au sol. Dans certains cas, le processus de vérification consiste à calculer l'espace des états atteignables du programme, c'est à dire toutes les configurations possibles du programme et à vérifier que chacun de ces états est correct par rapport aux spécifications.

Cette thèse parle de calcul d'états atteignables. Ce calcul constitue un élément de base dans la compilation de programmes ESTEREL. La vérification automatique, aussi appelée *Model Checking* [25], est l'une des applications les plus intuitivement naturelle de ce calcul, mais en réalité ce dernier est utilisé dans plusieurs étapes de la compilation comme l'optimisation des programmes [71], la génération de séquences de test [6]. L'espace des états atteignables d'un programme ESTEREL est toujours calculable en théorie car cet ensemble est fini. Cet ensemble peut s'obtenir en énumérant un à un tous les états mais le nombre d'états peut être prohibitivement grand. A l'énumération nous préférerons des méthodes symboliques qui nous permettent de représenter les ensembles d'états par des formules qui les caractérisent. En ESTEREL, l'état du programme est représenté par un vecteur de variables booléennes qui indiquent la position du flot de contrôle (comme dans un réseau de Pétri [67]). Ainsi, les ensembles d'états des pro-

grammes peuvent être représentés par des formules logiques sur ces variables. Comme nous le verrons dans ce document, il existe des méthodes permettant de calculer symboliquement l'espace des états atteignables avec des structures de données qui permettent de représenter efficacement ces formules logiques ; ces structures sont appelées des *Diagrammes de Décision Binaires* ou BDD (*Binary Decision Diagrams*).

1.1 Méthodes symboliques

Depuis les années 90, les BDDs ont été utilisés dans diverses méthodes de vérification pour représenter implicitement des espaces d'états [24, 23]. Ces méthodes partent d'une représentation "circuit" des programmes. Les BDDs ont ainsi permis d'appliquer ces méthodes à la vérification de larges applications synchrones réalistes. Le calcul de base de l'espace des états atteignables d'un programme est un algorithme symbolique de point fixe utilisant des BDDs et permettant de produire les états atteignables par un algorithme *Breadth First Search* : chaque étape de l'algorithme permet de construire tous les successeurs des états atteints à l'étape précédente par l'application d'une fonction de transition. Mais compter sur le seul pouvoir des BDDs ne suffit pas toujours pour faire face à la complexité de la construction de cet espace. Plus précisément, alors que le BDD représentant l'espace d'états atteignables final est souvent très compact, la fonction de transition et les calculs des états successeurs dans les étapes intermédiaires de l'algorithme peuvent nécessiter l'utilisation de BDDs beaucoup plus larges. Des techniques intelligentes ont déjà été proposées pour partitionner l'application de la fonction de transition et permettent de résoudre ce problème en partie [21, 22, 50].

Jusqu'à présent, ESTEREL utilisait ces techniques à travers une traduction en circuits. Cette traduction aplatit la structure mais permet de représenter les programmes dans un format directement exploitable par les outils de calcul. Dans le contexte d'ESTEREL, nous proposons d'utiliser la structure syntaxique naturelle des programmes afin d'appliquer la fonction de transition par morceau. Intuitivement, si P et Q sont deux blocs de programme composés en séquence P ; Q, il semble naturel de calculer d'abord **tous** les états atteignables dans P, et ensuite seulement de calculer tous les états atteignables dans Q. Un algorithme Breadth First Search aurait plutôt combiné des états de P avec le "début" de ceux de Q dès lors que tous les comportements de P ne sont pas de même durée. Un tel découpage est intuitivement une bonne idée puisqu'il découpe linéairement un problème dont la complexité est fonction de P ; Q en deux problèmes dont les complexités sont respectivement fonction de P seulement et fonction de Q seulement. Alors que cette idée semble triviale, les difficultés réelles apparaissent en présence de parallélisme. Dans un programme parallèle P || Q, l'ensemble des états atteignables n'est pas en général le simple produit cartésien des comportements de P et de Q du fait des échanges de signaux internes entre P et Q introduisant de la synchronisation entre les blocs. Ceci rend le calcul des états atteignables difficile en même temps qu'il justifie son utilité. Afin que notre approche retienne quelques avantages d'une approche symbolique, une attention toute particulière doit être prise afin que notre partitionnement ne s'apparente pas à l'énumération de tous les états possibles du programme ; une idée typiquement mauvaise consisterait à partitionner un programme de la forme P_1 ;P_2 || Q_1 ;Q_2 suivant P_1Q_1, P_1Q_2, P_2Q_1 et P_2Q_2.

1.2 Notre approche

En essence, notre algorithme partitionné procède de la manière suivante : initialement, une fonction de transition très réduite est appliquée. Les parties de la fonction de transition qui agissent sur les blocs de programme inactifs ne sont pas représentées. Au départ, nous "bloquons" explicitement les endroits du programme où un signal externe ou interne est reçu, faisant progresser le contrôle dans des sous-blocs suivants. Ensuite, ces réceptions sont progressivement "débloquées" dans un ordre adéquat de telle sorte que la fonction de transition ne fait que croître en explorant les sous-blocs de contrôle successifs. Parallèlement à cela, comme les nouvelles extensions de la fonction de transition ne sont appliquées qu'aux états les "plus récents" dans de nouvelles parties du programme, les anciennes parties de la fonction de transition qui ont déjà servi sur les blocs "saturés" sont automatiquement simplifiées grâce à quelques propriétés de simplification des BDDs (proposées par la librairie `TiGeR` [31]). Ces simplifications permettent de ne représenter que la partie utile de la fonction de transition en fonction de son domaine d'application. Cette opération permet en pratique de simplifier radicalement le support de la fonction de transition, c'est à dire l'ensemble des variables dont dépend effectivement la fonction. Ces simplifications ont un effet bénéfique immédiat sur les calculs.
Les heuristiques permettant d'ordonner le "débloquage" des réceptions de signaux sont basées sur une structure de graphe extraite de la syntaxe des programmes. De cette manière, cet ordre est conforme avec l'ordre naturel de propagation du contrôle qui peut exister dans le programme source (en omettant les retours de boucle). Par exemple, quand la réception d'un signal `S` provoque l'émission d'un signal `T`, si `T` est lui même attendu par une instruction réceptrice alors il est évidemment sensé de débloquer la réception de `S` avant celle de `T`.

1.3 Un exemple

La montre à quartz [10] est l'une des applications classiques d'Esterel. Cette application, illustrée par la figure 1.1, consiste en plusieurs modules et une interface composée de quatre boutons en entrée et d'un écran LCD en sortie (avec également une sonnerie). Un des boutons permet de faire basculer la montre d'un mode à l'autre, les trois autres boutons permettent de contrôler la montre en fonction du mode courant. Les modules de ce programme sont les suivants :

- **Un module** Alarm calcule la date et l'heure en fonction d'un signal d'entrée régulier envoyé par le quartz. Lorsqu'un changement intervient, ce module retransmet l'information au module d'affichage Display.
- **Un module** Time_set permet de régler la date et l'heure en utilisant les boutons de la montre.
- **Un module** Alarm_set permet de régler l'heure de l'alarme et d'activer ou de désactiver cette alarme.
- **Un module** Stopwatch permet d'utiliser la montre comme un chronomètre. Le chronomètre peut continuer à tourner même si il n'est pas affiché à l'écran.
- **Un module** Display permet d'afficher les bonnes informations sur l'écran LCD et de gérer la sonnerie de la montre en fonction du mode sélectionné.
- **Un module** Button_decoder permet de faire la liaison entre les boutons de la montre et les signaux d'entrée spécifiques de chacun des sous-modules en fonction du mode sélectionné. En particulier, le rôle du bouton situé en haut et à droite du schéma et appelé "*Mode_Select*" sera de passer d'un mode actif à un autre, c'est à dire d'alterner

entre le mode du simple affichage, le mode de réglage de l'heure (TIME_SET), le mode de réglage de l'alarme (ALARM_SET) et le mode chronomètre (STOPWATCH).

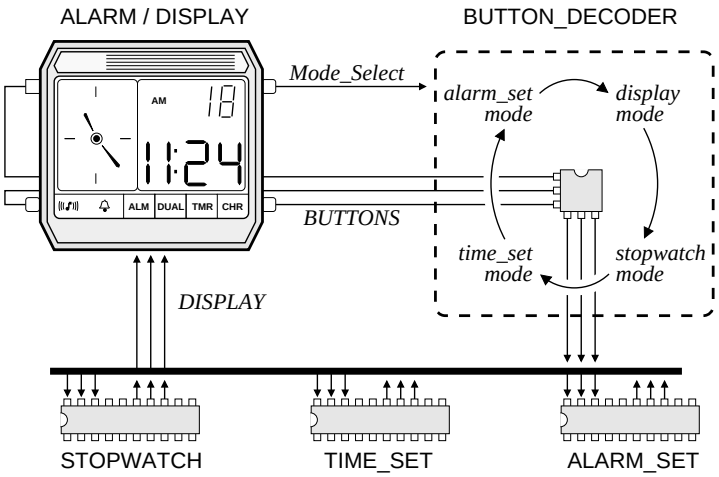


FIG. 1.1 – *La montre à quartz.*

Dans cette application, les modules TIME_SET, ALARM_SET et STOPWATCH sont lancés en parallèle mais leurs comportements sont largement exclusifs en ce qui concerne leur réaction aux événements d'entrée, c'est à dire aux boutons et en ce qui concerne l'affichage. En effet, les boutons ne permettent de contrôler qu'un seul module à la fois et l'affichage varie selon le mode sélectionné. L'analyse Breadth First Search basique d'un tel programme ne tire aucun avantage du fait que tous ces sous-modules sont exclusifs et calcule l'espace des états atteignables sur le programme tout entier.
L'analyse de ce programme pourrait être divisée en quatre partie : La première permettant de calculer les états atteignables pour le mode d'affichage de l'heure et les trois autres permettant de calculer les états atteignables dans les modules TIME_SET puis ALARM_SET et enfin STOPWATCH. Ainsi, la recherche des états de chaque mode pourrait être réalisée indépendamment des trois autres modes. Le gain en espace d'une telle approche est évident puisque l'analyse du programme original peut alors être assimilée à l'analyse de quatre programmes, tous de taille inférieur au programme original. Dans chacune de ces quatre phases, des fonctions de transitions locales sont utilisées à la place d'une seule fonction de transition globale.

1.4 Travaux reliés

Utiliser la structure algorithmique des programmes ESTEREL afin de guider la construction symbolique et exhaustive de l'espace d'états atteignables est une idée simple et claire qui n'a jamais été expérimentée à notre connaissance. D'autres travaux en rapport avec les nôtres utilisent des techniques Depth First Search pour la recherche explicite d'états atteignables afin d'identifier un squelette de configurations initiales "en avance". Une fois ces configurations calculées, des méthodes symboliques Breadth First Search sont appliquées [66] en utilisant une relation de transition partitionnée. Le but de ces travaux est plus d'ordonnancer le processus de génération des états atteignables que de minimiser la représentation de la relation de transition. D'autres travaux présentés dans [2] exploitent la structure hiérarchique de programmes asynchrones afin d'optimiser le calcul des états atteignables. Ces travaux reposent entre autre sur

l'hypothèse d'un opérateur parallèle asynchrone. Cette hypothèse ne permet malheureusement pas d'appliquer ces résultats au cas d'ESTEREL où l'exécution de tâches en parallèle est synchrone.
Certains travaux visent à optimiser le calcul des états atteignables afin de converger le plus rapidement possible vers les états directement concernés par une propriété donnée [13, 79]. Le but est de déterminer le plus rapidement possible si cette propriété n'est pas vérifiée par le programme analysé.
Autour du vérificateur formel Murφ [32, 45], de nombreux travaux visant à améliorer le calcul des états atteignables ont été menés. L'idée d'utiliser des dépendances fonctionnelles entre les variables du programmes a été introduite dans [46, 47] et améliorée dans [48]. La méthode consiste à définir la valeur de certaines variables en fonctions d'autres variables indépendantes. Dès lors, le calcul des états atteignables peut être réalisé à partir des variables indépendantes seulement. Dans [49], la méthode consiste à exploiter les symétries dans la description des programmes. L'introduction d'un type particulier permet au programmeur de définir des équivalences entre les états du programme. Par la suite, lors de l'analyse du programme, il suffit d'explorer un seul état par classe d'équivalence. Cette méthode permet de réduire la taille des BDDs utilisés par le vérificateur Murφ.

Certaines études visent à approcher le calcul des états atteignables. Dans [69] Ravi et Somenzi calculent une sous-approximation de l'espace des états atteignables. La densité d'un BDD se mesure en divisant le nombre d'états encodés par le nombre de noeuds de ce BDD. A chaque étape de calcul, lorsque les BDDs atteignent une taille trop importante, les branches les moins denses sont supprimées. Ceci permet d'obtenir des BDDs plus petits en conservant un maximum d'états atteignables. D'autres travaux s'appuient sur une surapproximation du calcul des états atteignables [37, 38, 63]. L'intérêt de ces méthodes est de garantir certaines propriétés des programmes. Si tous les états calculés vérifient ces propriétés alors tous les états atteignables aussi. Dans le cas contraire, une analyse plus fine est nécessaire pour déterminer si les états qui invalident ces propriétés sont réellement atteignables.

Le calcul des états atteignables à base de BDDs a également été adapté pour permettre l'analyse de programmes C. Dans [33], Edwards et al. ont réalisé l'analyse de petits programmes même si le passage à l'échelle demeure incertain. Dans [8, 9], Ball et Rajamani présentent un vérificateur capable d'analyser des programmes C booléens. Alors que le calcul des états atteignables s'effectue sur des programmes finis, ce vérificateur autorise des appels de fonction récursifs et non bornés.

Les travaux les plus proches techniquement sont ceux de Yannis Bres. La thèse [17] présente des méthodes d'abstraction permettant de simplifier le calcul des états atteignables. Plus précisément, ces techniques visent à remplacer certaines variables d'états par de simples variables d'entrée ou bien à utiliser une logique de Scott trivaluée. Ceci permet de supprimer des blocs entiers de programme lors de son analyse. Cette méthode conduit à une surapproximation de l'espace des états atteignables, mais grâce à cela, Yannis Bres à réussi à valider un certain nombre de propriétés. Concrètement, ces travaux et les nôtres ont été intégrés au sein d'un même logiciel.

1.5 Présentation du document

Cette thèse présente un algorithme de partitionnement permettant de calculer plus efficacement l'espace des états atteignables d'un programme ESTEREL. Cette efficacité se mesure en espace plus qu'en temps. Ce document est organisé de la manière suivante :

Dans le chapitre 2, nous présentons le langage ESTEREL ainsi que les méthodes symboliques à base de BDDs que nous utilisons. Nous présentons également l'algorithme Breadth First Search de base que nous souhaitons améliorer.

Le chapitre 3 motive et présente notre algorithme de manière intuitive. Nous décrivons les difficultés que peut représenter le parallélisme par rapport à notre approche et nous proposons une solution à ces problèmes.

Le chapitre 4 définit formellement les opérateurs utilisés dans notre algorithme de partitionnement. La construction d'un graphe permettant de guider ce partitionnement est donnée.

Le chapitre 5 présente formellement notre algorithme. Une démonstration de la validité de cet algorithme est donnée.

Le chapitre 6 décrit les points clés de l'implémentation de notre méthode par rapport à la chaîne de compilation des programmes ESTEREL et par rapport aux outils d'analyse déjà existants.

Le chapitre 7 présente des résultats expérimentaux montrant l'utilité de notre approche.

Le chapitre 8 conclut cette thèse et donne quelques perspectives.

Chapitre 2

Contexte de l'Etude

Ce chapitre présente le contexte de nos travaux. La section 2.1 présente le langage ESTEREL ainsi que ses modèles sémantiques. La section 2.2 décrit plus particulièrement le modèle à partir duquel est calculé l'espace des états atteignables des programmes ESTEREL (section 2.3). Enfin, la section 2.4 introduit les notions théoriques qui permettent de modéliser et de calculer cet espace de manière symbolique et la section 2.5 présente une solution permettant d'implémenter ce calcul à l'aide de BDDs.

2.1 ESTEREL

A l'origine, le langage ESTEREL [12] a été conçu par deux chercheurs (Jean-Paul Marmorat et Jean-Paul Rigault) dans le but de programmer le comportement d'une voiture automatique pour un concours organisé par un journal d'électronique. Né d'un besoin anecdotique, ce langage a su trouver sa place dans le monde des systèmes temps-réels en réalisant son dessein initial : permettre l'expression d'algorithmes de contrôle au moyen d'un langage clair et intuitif. Ainsi le langage ESTEREL est-il particulièrement adapté aux applications dans lesquelles le contrôle tient une place prépondérante comme les systèmes temps-réel, les systèmes embarqués, les protocoles de communication ou les interfaces homme-machine. ESTEREL est un langage impératif, déterministe, réactif [41] et synchrone [40] :

- **Réactif :** Par opposition à un programme *transformationnel* qui traite un ensemble de données pour produire un résultat, un programme ESTEREL à pour vocation de réagir continuellement à des événements provenant de son environnement. Un programme réactif produit sans cesse de nouvelles sorties en réponse à des événements extérieurs. Ces programmes se composent souvent de sous-programmes s'exécutant en parallèle et communicant entre eux.
- **Synchrone :** Par rapport aux programmes réactifs, l'hypothèse synchrone stipule que chaque réaction est synchrone avec l'événement qui l'a provoquée : la durée d'une réaction est supposée nulle. De cette manière, chaque événement divise naturellement le temps en instants. En ESTEREL, les instants sont cadencés par les émissions d'un signal particulier appelé "`tick`" et les autres événements sont supposés synchrones avec les émissions de ce signal.
- **Déterministe :** Une même séquence d'événements produit toujours la même séquence de réactions.

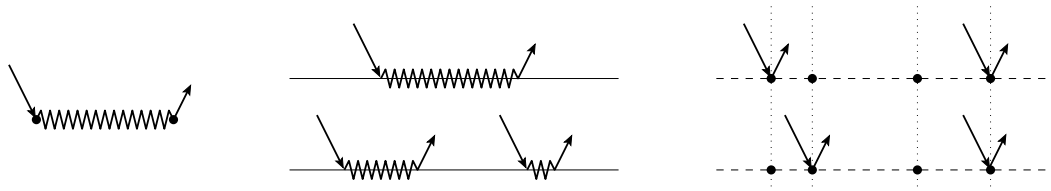


FIG. 2.1 – *Différents types de programmes : transformationnel (a), réactif (b), synchrone (c).*

En réalité, la réaction de durée nulle n'est qu'une abstraction. Une hypothèse équivalente mais plus réaliste consiste à dire que le temps est divisé en une séquence d'instants logiques. Les instants logiques sont vus comme des intervalles de temps communs à tous les composants et sont deux à deux disjoints. A l'intérieur de ces instants, chaque réaction est non interruptible et termine forcément avant le début de l'instant suivant.
L'hypothèse synchrone permet de définir clairement les notions de simultanéité entre occurrences d'événements ainsi que la notion d'absence d'un événement.

En plus des constructions classiques des langages impératifs (séquence, boucle, if-then-else), le noyau du langage ESTEREL procure un opérateur de parallélisme ainsi qu'un mécanisme d'exception et de suspension. La communication entre les différentes parties actives du programme est réalisée par la diffusion de signaux purs (un signal est soit présent soit absent). Cette communication est instantanée, ce qui signifie qu'un signal est reçu à l'instant précis où il est émis. Une réaction consiste en une propagation des entrées vers les sorties. Cette propagation passe par l'utilisation de variables et de signaux locaux. A chaque instant, un signal peut être émis, donc présent ou bien ne plus être émissible, donc absent. L'absence d'un signal est donc une notion effective et non une notion implicite. Les instructions du langage noyau sont :

- **nothing** : l'instruction vide du langage.
- **pause** : cette instruction marque un délai, elle se met en attente de l'instant suivant.
- **emit** S : instruction de durée nulle qui provoque l'émission du signal S.
- **present** S **then** p **else** q **end** : test de présence d'un signal.
- **suspend** p **when** S : suspend l'exécution du bloc p chaque fois que S est reçu.
- $p;q$: opérateur de séquencement. L'exécution de q suit immédiatement l'exécution de p. La notion de séquence est orthogonale à la notion d'instant. Ainsi, la fin du bloc p et le début du bloc q peuvent s'exécuter au sein d'une unique réaction (de durée nulle d'après l'hypothèse synchrone).
- **loop** p **end** : exécute le bloc p en boucle. p ne doit pas être instantané.
- p || q : opérateur parallèle. L'exécution termine lorsque p et q ont tous les deux terminé.
- **trap** T **in** p **end** : exécute le bloc p jusqu'à la levée d'une exception T ou bien jusqu'à-ce que p termine.
- **exit** T : lève l'exception T.
- **signal** S **in** p **end** : déclaration d'un signal local dans p.

A partir du langage noyau, d'autres instructions d'usage fréquent ont été introduites. Parmi elles :

- **halt** : équivalent de **loop pause end**.
- **await** S : attend la réception du signal S.
- **sustain** S : émet S à chaque instant.
- **every** S **do** p **end** : démarre l'exécution de p à chaque réception du signal S.

- **abort** p **when** S : préemption forte. Interrompt l'exécution de p dès la réception de S.
- **weak abort** p **when** S : préemption faible. Dès la réception de S, exécute les dernières réactions instantanées du bloc p avant d'interrompre son exécution.

Par défaut, les instructions `suspend`, `await`, `every` et `abort`/`weak abort` ne réagissent à la présence du signal S qu'à partir de l'instant suivant leur première activation. Le langage étendu définit le mot-clé `immediate` qui, dans un exemple comme "**abort** p **when immediate** S", permet de réagir à la présence de S dès le premier instant.

Un simple exemple. Le programme ESTEREL suivant modélise une porte automatique dont le comportement consiste à s'ouvrir dès qu'un utilisateur appuie sur un bouton. La porte reste ouverte pendant un délai de trois secondes après la dernière demande d'ouverture, puis se referme. Le temps et le bouton d'ouverture sont les entrées du programme (portées par des signaux purs, SECONDES et BOUTON). Comble de modernité, un système de sécurité empêche la fermeture de la porte quand une présence est détectée (signal PRESENCE). Le signal OUVERTURE en sortie indique si la porte est ouverte :

```
module porte:
   input BOUTON, SECONDES, PRESENCE;
   output OUVERTURE;

   every BOUTON do
      trap FERMER in
         await 3 SECONDES;
         suspend exit FERMER when immediate PRESENCE
       ||
         sustain OUVERTURE
      end trap;
   end every
end module
```

Les données en ESTEREL**.** Dans sa version complète, le langage permet aussi de manipuler des données : booléens, entiers, flottants et chaînes de caractères. Nous avons présenté la partie contrôle auparavant car elle constitue la partie novatrice du langage ESTEREL. Le traitement des données est plus classique et souvent délégué à un langage hôte plus généraliste comme C ou JAVA. Par ailleurs, ESTEREL procure également un mécanisme capable de créer et de manipuler ses propres structures de données par une interface avec le langage C. Ces données peuvent être manipulées par le biais de simples variables ou bien être transmises par des signaux. Dans ce second cas, un signal est caractérisé à la fois par son statut de présence ou d'absence et par la valeur qu'il transporte, ces deux notions étant orthogonales. Par exemple, un signal de type booléen peut être à la fois présent et faux ou bien absent et vrai. La valeur portée par un signal demeure inchangée tant que le signal n'est pas émis. La valeur d'un signal demeure donc indéfinie ou bien initialisée à une valeur par défaut jusqu'à ce que le signal soit émis pour la première fois. A la différence d'une variable, un signal ne peut pas prendre plusieurs valeurs successives au cours d'un même instant.
Le langage propose évidemment quelques opérations de base : opérateurs logiques et arithmétique, manipulation de chaînes de caractères, instruction de test sur les variables, etc... En

ESTEREL, toutes ces opérations (ou blocs d'opérations) sont appelées des *actions*. Toute action est supposée être exécutée instantanément. Dans la version noyau du langage, chaque action peut être abstraite par un signal pur : une opération est abstraite par l'émission d'un signal de sortie et le résultat d'un test sur une variable est abstrait par un test de présence d'un signal d'entrée. Pour notre étude, nous nous limiterons à cette version noyau du langage ESTEREL.

Les langages de la famille d'ESTEREL**.** Alors que le langage ESTEREL est particulièrement adapté à la programmation du contrôle, il existe d'autres langages réactifs synchrones spécialisés dans la programmation du flot de données. Ces langages incluent LUSTRE [39] et SIGNAL [53]. Dans ces langages, la valeur de chaque variable est recalculée à chaque instant en fonction de la valeur des variables aux instants précédents. Le traitement des données est beaucoup plus simple qu'en ESTEREL mais la programmation d'une simple séquence devient rapidement très complexe.
D'autres langages graphiques ont également été proposés. Le formalisme des SYNCCHARTS [4, 5] possède la même expressivité que le langage ESTEREL. Son formalisme graphique est inspiré d'ARGOS [57], la version synchrone des STATECHARTS [42].

2.1.1 Aspects sémantiques

Le langage ESTEREL possède une sémantique opérationnelle structurelle [68] (SOS). Cette sémantique se présente sous la forme de règles de réécriture de la forme de l'exemple suivant :

$$\frac{p \xrightarrow[E]{E',0} p' \quad q \xrightarrow[E]{F',l} q'}{p\,;\, q \xrightarrow[E]{E' \cup F',l} q'}$$

La règle sémantique présentée ci-dessus est celle de la séquence $p;q$, pour le cas où p termine dans l'instant. La totalité des règles de la sémantique opérationnelle est détaillée dans [11]. Cette sémantique se décline en deux nuances : la sémantique comportementale logique et la sémantique comportementale constructive qui est un raffinement de la sémantique logique. La différence entre ces deux sémantiques réside dans les règles qui régissent la présence ou l'absence d'un signal. Dans la sémantique logique, le statut d'un signal est déterminé en supposant successivement que le signal est présent, puis absent. Un programme est correct si une et une seule de ces deux suppositions permet d'aboutir a une solution. La sémantique constructive interdit de présumer le statut d'un signal. Un signal n'est présent que si il est forcément émis et il n'est absent que si il ne peut pas être émis.
Les sémantiques opérationnelles d'ESTEREL permettent une interprétation des programmes sous forme de machine de Mealy à états finis. Autrement dit, tout programme ESTEREL correct peut être compilé sous la forme d'un automate et exécuté symboliquement.

La possibilité de traduire les programmes ESTEREL sous forme de circuits logiques séquentiels procure à ESTEREL une sémantique dénotationnelle. Le modèle des circuits séquentiels possède également une sémantique constructive équivalente à celle d'ESTEREL : la sémantique dénotationnelle qui consiste à propager les constantes booléennes 0 et 1 dans la traduction circuit est équivalente à la sémantique constructive qui consiste à propager l'information selon laquelle un signal doit être présent ou bien ne peut être présent.

2.1.1.1 Réincarnation

A cause des boucles instantanées, les signaux peuvent avoir plusieurs instances simultanées appelées *réincarnations*.

```
loop
   signal S in
      present S then emit O1 else emit O2 end;
      pause;
      emit S
   end signal
end loop
```

Au premier instant, `S` n'est pas émis. Au deuxième instant, le corps de la boucle termine en émettant `S` et se relance immédiatement. Un nouveau signal `S` distinct de l'ancien est déclaré. Dans cette seconde incarnation, le signal `S` n'est pas émis. Dans cet exemple, le signal `O2` est émis à chaque instant.
En réalité, la réincarnation existe dans tous les langages de programmation. Dans les langages séquentiels classiques (C, JAVA...) les réincarnations ne sont pas suscitées par les boucles mais par les appels récursifs de fonctions. Le modèle d'exécution de ces programmes étant dynamique, le problème de réincarnation est résolu de manière transparente puisque chaque instance de variable est allouée dans la pile d'exécution. Le modèle d'exécution des programmes ESTEREL est un modèle statique dans lequel chaque instance de variable doit être allouée de manière statique à chaque instant. Les restrictions imposées par le langage ESTEREL par rapport aux langages plus généralistes permettent de garantir que le nombre de ces instances est fini.

2.1.1.2 Correction logique

Programmes non-réactifs. L'instantanéité des réactions et de la diffusion des signaux permet d'écrire des programmes syntaxiquement corrects mais insensés comme le programme suivant :

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

Dans ce programme, le signal `S` ne peut être présent car il n'est émis nulle part. Le signal ne peut pas non plus être absent car dans ce cas, il est immédiatement émis. Ce programme est donc incorrect.

Programmes non-déterministes. ESTEREL est un langage déterministe par conséquent, tout programme non déterministe est incorrect. Considérons à présent l'exemple suivant :

```
present S then emit S end
```

Ici, le fait que `S` soit présent ou absent n'entre pas en contradiction avec la sémantique du langage. L'existence de ces deux interprétations rend ce programme non-déterministe. Ce programme est donc également incorrect.

2.1.1.3 Constructivité

Un programme est logiquement correct si il est à la fois réactif et déterministe. Toutefois, ce critère n'est pas suffisant. Pour un langage impératif comme ESTEREL, l'évaluation d'un test doit toujours précéder l'évaluation de ses branches. Considérons l'exemple suivant :

```
present S then emit S else emit S end
```

Dans ce programme, la seule solution consiste à considérer que le signal `S` est présent. Le programme est donc logiquement correct, mais pour savoir si `S` est présent, la seule solution consiste à évaluer le contenu des branches `then` et `else` avant de pouvoir évaluer le test. Ceci rentre en contradiction avec la propagation naturelle du contrôle dans le programme.

L'idée de la sémantique constructive consiste à interdire tout résonnement spéculatif. Pour déterminer le statut des signaux, on utilise une logique tri-valuée : *présent*, *absent* ou bien *inconnu*. A chaque instant, le statut des signaux d'entrée est donné par l'environnement. Initialement, le statut des autres signaux est inconnu. Les seules déductions autorisées sont les suivantes :

1. Un signal est présent si il est émis.
2. Un signal est absent si il ne peut être émis par aucune instruction.
3. On ne peut exécuter les branches d'un test que si le statut du signal testé est connu.

On dit qu'un programme est *constructif* si le statut de chaque signal peut être déterminé en utilisant les règles précédentes. Les règles mathématiques précises de la sémantique constructive du langage ESTEREL sont données dans [11].

2.1.2 Machines de Mealy

Le comportement d'un programme ESTEREL peut être modélisé par un automate déterministe à états finis appelé *machine de Mealy* [59]. Cette représentation est issue de la sémantique opérationnelle (SOS) du langage par exploration de tous les états de contrôle. Dans cette représentation, chaque transition de l'automate porte une étiquette exprimant les entrées et les sorties du programme : un signal d'entrés peut être précédé par le symbole '`?`' indiquant que le signal est présent ou bien par le symbole '`#`' indiquant que le signal est absent. Un signal de sortie précédé du symbole '`!`' indique que le signal est émis. Par exemple, une transition étiquetée par "`?I1.#I2. !O`" est empruntée si `I1` est présent et si `I2` est absent, quel que soit le statut des autres signaux d'entrée. Cette transition provoque l'émission du signal `O`.
La figure 2.2 représente le programme de la porte automatique sous forme de machine de Mealy. Dans le graphe, les noms des signaux sont représentés par leurs initiales.

Cette représentation sous forme de machine de Mealy rend explicite l'ensemble des états et des transitions d'un programme. Toutefois, l'automate peut avoir un nombre d'état exponentiel par rapport à la taille du programme ESTEREL source. Il n'est donc pas souvent raisonnable de représenter un programme réel par un tel automate ; ceci est vrai aussi bien pour la compilation des programmes que pour leur vérification. Cette représentation n'en demeure pas moins un excellent outils de référence pour définir nos techniques d'analyse de programme et donc un modèle sémantique sous-jacent.

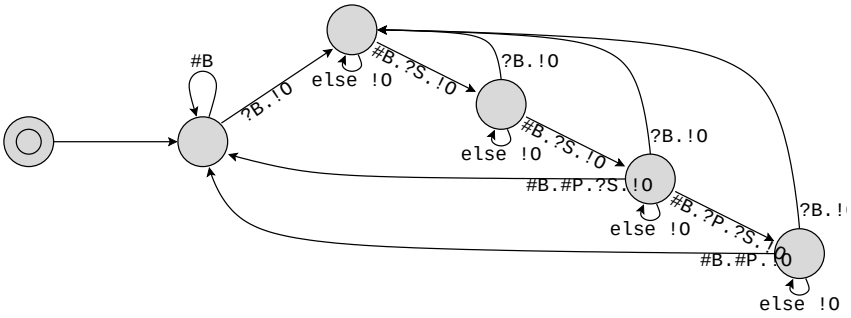
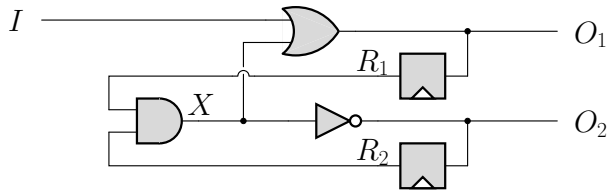


FIG. 2.2 – *Machine de Mealy de la porte automatique.*

2.1.3 Circuits séquentiels

Les machines à états finis de Mealy (Mealy FSM) peuvent aussi servir de modèle opérationnel aux *circuits séquentiels*. Un circuit séquentiel se présente sous forme d'un système d'équations booléennes. Tout circuit séquentiel admet une traduction sous forme d'automate de Mealy.
Le langage ESTEREL bénéficie également d'une traduction sous forme de circuit. La traduction des langages de la famille d'ESTEREL comme les automates hiérarchiques ou les SYNCCHARTS [4] vers les circuits booléens de contrôle est comparable à la traduction d'un calcul de processus vers un réseau de Pétri.
Les circuits séquentiels sont issus d'une interprétation dénotationnelle du langage. Cette traduction est compositionnelle et préserve la sémantique du langage. On parle ici de format circuit par analogie aux circuits électroniques composés de portes logiques.



$$\begin{aligned} O_1 &= I \vee X \\ O_2 &= \neg X \\ R_1 &= reg(O_1) \\ R_2 &= reg(O_2) \\ X &= R_1 \wedge R_2 \end{aligned}$$

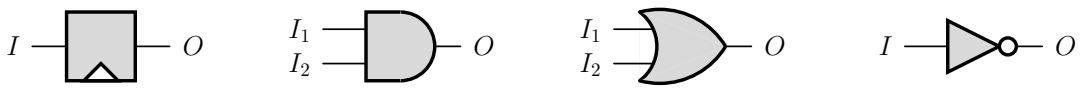
FIG. 2.3 – *Circuit séquentiel.*

Parmi les portes logiques se distinguent les *registres booléens* qui constituent les délais élémentaires du programme. Ils permettent de conserver des données de manière persistante et donc de mémoriser l'état du programme :

$$Y = reg(X)$$

signifie que la valeur Y du registre à l'instant $n+1$ est égale à la valeur de X à l'instant n. Le reste des portes ($\wedge$, $\vee$, $\neg$) constitue la partie *combinatoire* du système. Les signaux d'entrée et de sortie du programmes sont représentés par de simples fils, porteurs d'une valeur booléenne : 1 lorsque le signal est présent et 0 lorsque le signal est absent.

Modèle d'exécution des circuits séquentiels. Comme en ESTEREL, un circuit séquentiel réagit à chaque instant. La notion d'instant est ici réalisée par l'utilisation des registres booléens. Au premier instant, tous les registres sont positionnés à 0. A chaque instant, le statut de chaque signal de sortie est calculé en fonction des signaux d'entrée et de la valeur des registres. Ce calcul est réalisé en suivant les règles de propagation du courant électrique. Ainsi, la valeur



(a) $O = reg(I)$ (b) $O = I_1 \wedge I_2$ (c) $O = I_1 \vee I_2$ (d) $O = \neg I$

FIG. 2.4 – *Portes logiques*

d'une porte "et" peut être déterminée dès que l'une de ses branches porte la valeur 0 ou bien dès que toutes ses branches portent la valeur 1. Cette opération est réalisée instantanément par la partie combinatoire du circuit. De la même manière, la prochaine valeur des registres est calculée en fonction des valeurs courantes et des entrées.

Génération des circuits à partir des programmes ESTEREL**.** La sémantique constructive d'ESTEREL permet de traduire récursivement chaque instruction du langage en un circuit séquentiel. Cette traduction produit un registre booléen pour chaque instruction `pause` du langage source. En d'autres termes, le codage de l'état d'un bloc de programme repose sur un ensemble précis de registres. Le reste de la traduction consiste à câbler les équations combinatoires autour de ces registres.

Circuits cycliques. La traduction des programmes en circuits génère parfois des cycles dans les équations combinatoires. Cela signifie que, au cours d'un même instant, la valeur d'un fil x dépend de la valeur d'un second fil y et que la valeur de y dépend de x. Comme dans la section 2.1.1 traitant des aspects sémantiques du langage, il se peut que le programme soit logiquement incorrect ou bien que le cycle ne pose pas de problème dans la résolution constructive de l'état du circuit. Il existe une théorie de la causalité constructive qui identifie des circuits cycliques corrects. Pour ces derniers, il existe des algorithmes, parfois coûteux en expansion, permettant de les transformer en circuits acycliques et sémantiquement équivalents [73]. Pour notre étude, nous pourrons simplement faire l'hypothèse que les circuits que nous manipulons sont acycliques.

2.1.4 Compilation des programmes ESTEREL en circuit

Cette section présente les principes de base de la traduction des programmes ESTEREL en circuit. La traduction présentée ici est purement structurelle. En réalité, les phénomènes de réincarnation présentés à la section 2.1.1.1 nécessitent la duplication de certaines parties du circuit. Pour plus de détails, on pourra se référer à [11]. Chaque instruction du langage produit un circuit dont l'interface est représentée dans la figure 2.5.

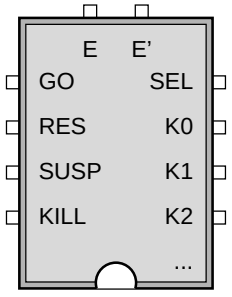


FIG. 2.5 – *Interface circuit des instructions d'*ESTEREL.

2.1.4.1 Interface des circuits

Sur le schéma 2.5, les broches de gauche représentent les entrées de l'interface et les broches de droite sont les sorties. La signification de chaque broche est la suivante :

- L'entrée `GO` est utilisée pour lancer l'exécution d'une instruction. Une instruction s'exécute dès que `GO` vaut 1.

- L'entrée `RES` est utilisée pour reprendre l'exécution d'une instruction après son démarrage. L'instruction continue son exécution après une pause tant que ce fil est à 1. Cette broche est plus particulièrement utilisée par les instructions `abort` et `suspend`.

- L'entrée `SUSP` est utilisée pour suspendre l'exécution d'une instruction. Lorsque ce fil est à 1, tous les registres de l'instruction conservent leur valeur à moins que le signal d'arrêt `KILL` ne soit reçu (voir ci-dessous).

- L'entrée `KILL` permet de tuer l'exécution d'une instruction. Cette broche permet de mettre la valeur de tous les registres de l'instruction à 0. Ce signal passe à 1 lorsqu'une exception est lancée. Dans ce cas, le signal `KILL` est propagé par toutes les instructions vers les instructions `pause`.

- La sortie `SEL` permet d'indiquer que l'instruction est toujours active après une pause et doit être relancée au moyen du signal `RES`. Ce signal vaut 1 dès qu'une `pause` de l'instruction est active. Ce signal est donc la disjonction de tous les registres de l'instruction.

- Les sorties `K0`, `K1`, etc. correspondent aux codes de complétion (terminaison, pause, levée d'exception, voir [11]). Si n représente le nombre d'instructions `trap` qui entourent l'instruction, alors ces broches sont au nombre de $n + 2$. Lorsqu'une instruction est démarrée ou bien relancée, la sortie correspondant au code de complétion de l'instruction est mise à 1. Si l'instruction n'est pas exécutée, toutes ces sorties sont à 0.

- Les broches `E` et `E'` correspondent à l'interface des signaux de l'instruction. `E` et `E'` ne sont pas de simples broches. Ce sont en réalités des vecteurs de signaux contenant un fil par signal. Les broches `E` et `E'` correspondent respectivement aux signaux d'entrée et de sortie.

2.1.4.2 Exécution des circuits

Le schéma d'exécution des circuits consiste tout d'abord à émettre le signal `GO` afin de démarrer l'exécution. Ensuite, à chaque cycle d'horloge, le signal `RES` est émis. A chaque cycle, le contrôle se propage par toutes les portes combinatoires du circuit. Les fils correspondants aux codes de complétion sont calculés et les registres correspondants aux instructions `pause` actives sont recalculés à chaque instant en fonction de leur valeur à l'instant précédent. Les signaux sont reçus et émis par les broches `E` et `E'`.

2.1.4.3 Traduction circuit

La traduction structurelle complète des instructions ESTEREL est donné dans [11]. Cette traduction produit exactement un registre booléen par instruction `pause`. La figure 2.6 montre

la traduction d'une instruction `pause` sous forme de circuit.

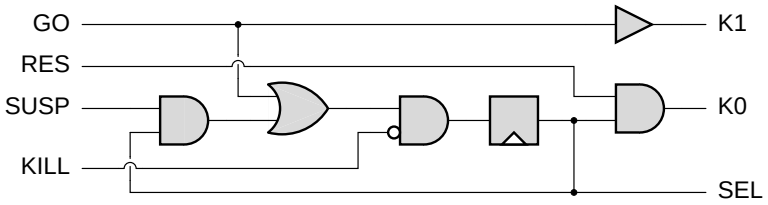


FIG. 2.6 – *Circuit d'une instruction* `pause`.

Les autres instructions du langage ne produisent que des portes combinatoires. La figure 2.7 illustre par exemple la traduction de l'instruction `present s then` p `else` q. Les broches de l'interface du `present` sont reliées aux broches de l'interface du bloc p par des portes logiques.

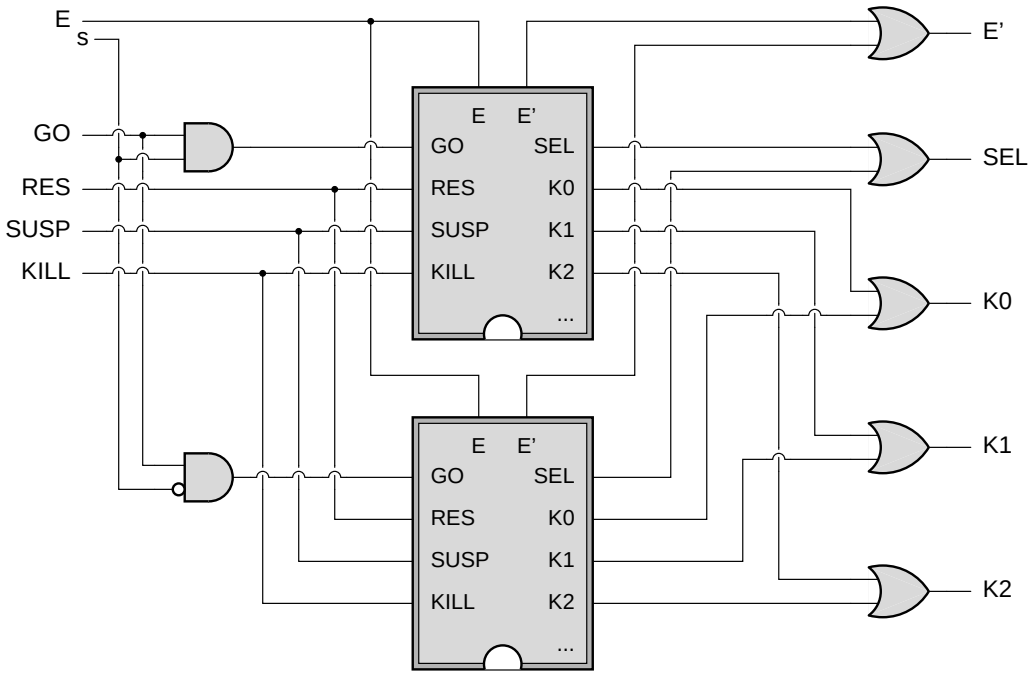


FIG. 2.7 – *Circuit d'une instruction* `present`.

Au niveau global, la traduction d'un programme ESTEREL produit un registre supplémentaire appelé *registre de boot*. Ce registre est relié à la broche `GO` et permet de lancer l'exécution du programme au premier instant.

2.1.5 Interprétation des circuits en machines de Mealy

Les modèles de circuit et de Mealy FSM permettent de représenter les programmes ESTEREL à des niveaux différents :

– La représentation circuit permet de conserver la causalité en rendant explicite les dépendances entre les différentes instructions du programme. En revanche, l'espace des états atteignables n'est pas calculé explicitement.

– La construction d'une Mealy FSM rend explicite l'espace des états atteignables mais la causalité n'est pas préservée.

Partant de ce constat, nous introduisons la notion de *machine séquentielle* (également appelée "*automate symbolique*"). Ce modèle est utilisé par des outils de vérification comme `SMV` [58] ou `TiGeR` [31]. Le modèle de machine séquentielle apporte une vision opérationnelle à la sémantique d'exécution des circuits ; les fonctions de transition de l'automate décrit implicitement sont représentées par des formules booléennes.

2.2 La machine séquentielle

Le calcul symbolique des états atteignables s'obtient traditionnellement sur le modèle de la *machine séquentielle*. A chaque réaction, ce modèle consomme un ensemble de données I en entrée et produit un ensemble de données O en sortie, calculé en fonction de I et de l'état R de la machine. De manière formelle, une machine séquentielle se définit par le triplet :

$$fsm \quad = \quad (\iota, \Upsilon, \Delta, \Gamma)$$

où ι désigne l'état initial, Δ désigne la fonction de transition et Γ désigne la fonction de sortie de la machine. Υ désigne l'ensemble des entrées valides de la machine. Il est possible d'abstraire Υ en supposant que cet ensemble désigne l'univers des entrées tout entier. Ceci revient à ne poser aucune restriction sur les entrées.

Les fonctions Δ et Γ calculent respectivement le prochain état et les sorties de la machine en fonction des entrées courantes et de l'état courant. Si nous notons I_n, O_n et R_n les entrées, les sorties et l'état de la machine à la $n^{\text{ème}}$ réaction, alors :

$$O_n \quad = \quad \Gamma(I_n, R_n) \quad \text{pour tout} \quad n \geq 0 \tag{2.1}$$

$$\begin{aligned} R_0 &= \iota \\ R_n &= \Delta(I_{n-1}, R_{n-1}) \quad \text{si} \quad n > 0 \end{aligned} \tag{2.2}$$

Dans la traduction des programmes ESTEREL sous forme de circuit, les entrées et les sorties se composent d'un vecteur de signaux booléens. L'état du circuit est codé par un ensemble de registres booléens. Soit $\mathbb{B} = \{0, 1\}$ l'ensemble des booléens. Nous avons alors $\Upsilon \in \mathbb{B}^m$, $I \in \mathbb{B}^m$, $\iota \in \mathbb{B}^p$, $R \in \mathbb{B}^p$ et $O \in \mathbb{B}^q$ où m, p et q sont le nombre de signaux d'entrée, le nombre de registres et le nombre de signaux de sortie. La fonction de transition globale $\Delta : \mathbb{B}^m \times \mathbb{B}^p \rightarrow \mathbb{B}^p$ et la fonction de sortie globale $\Gamma : \mathbb{B}^m \times \mathbb{B}^p \rightarrow \mathbb{B}^q$ sont naturellement modélisées par un système d'équations booléennes, comme le montre la figure 2.8.

$$\Delta : \quad (I, R) \quad \rightarrow \quad R' = \Delta(I, R) \tag{2.3}$$

$$\Gamma : \quad (I, R) \quad \rightarrow \quad O' = \Gamma(I, R) \tag{2.4}$$

En réalité, chaque registre et chaque signal de sortie possède sa propre fonction de transition [26]. Δ et Γ se décomposent ainsi en vecteurs de fonctions δ_i et γ_i qui ne dépendent chacune que de certains registres et de certains signaux d'entrée :

$$\begin{aligned} \delta_i : \quad \mathbb{B}^{m_i} \times \mathbb{B}^{p_i} &\rightarrow \mathbb{B} \\ (I_i, R_i) &\rightarrow r'_i = \delta_i(I_i, R_i) \end{aligned} \tag{2.5}$$

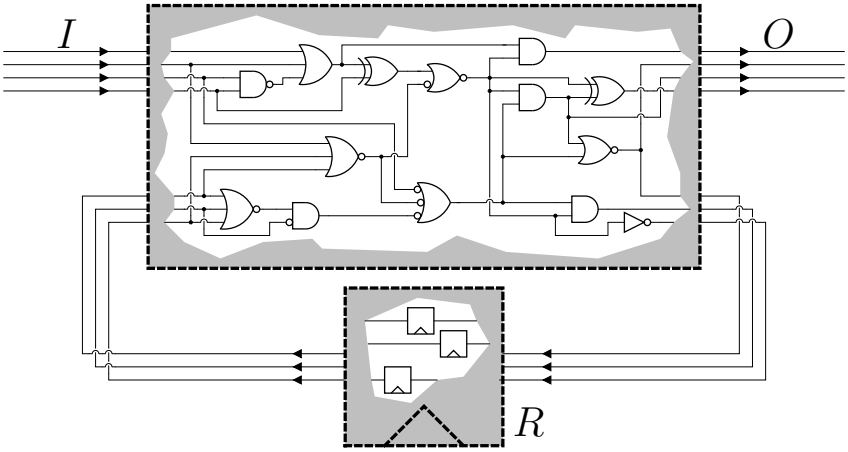


FIG. 2.8 – *Machine séquentielle.*

De même :

$$\begin{array}{rrcl} \gamma_i : & \mathbb{B}^{m_i} \times \mathbb{B}^{p_i} & \rightarrow & \mathbb{B} \\ & (I_i, R_i) & \rightarrow & o'_i = \gamma_i(I_i, R_i) \end{array} \tag{2.6}$$

Les vecteurs I_i et R_i, sous-vecteurs respectifs de I et R, constituent le *support* de ces fonctions. m_i et p_i sont respectivement le nombre de signaux d'entrée et le nombre de registres de ce support. Si R' désigne le vecteur $\langle r'_1 \dots r'_p \rangle$ et O' le vecteur $\langle o'_1 \dots o'_q \rangle$ alors les applications partitionnées des fonctions de transition et de sortie s'écrivent de la manière suivante :

$$R' = \Delta(I, R) \iff \bigwedge_{i=1}^{p} r'_i = \delta_i(I_i, R_i) \tag{2.7}$$

et

$$O' = \Gamma(I, R) \iff \bigwedge_{i=1}^{q} o'_i = \gamma_i(I_i, R_i) \tag{2.8}$$

2.3 Calcul des états atteignables d'une machine séquentielle

L'espace des états atteignables se calcule sur la représentation circuit du programme ESTEREL par un algorithme de recherche en largeur (ou *Breadth First Search*) qui traite donc d'ensembles d'états. La fonction de transition est appliquée successivement à tous les ensembles d'états atteignables à une certaine profondeur, en partant du singleton formé par l'état initial jusqu'à ce qu'un point fixe soit atteint quand plus aucun nouvel état n'est découvert. L'algorithme de base est le suivant :

```
1 reachable ← ι
2 new ← ι
3 tantque ( new ≠ ∅ ) faire
4     new ← Image_Δ(ϒ, new) ∖ reachable
5     reachable ← reachable ∪ new
6 fin tantque
```

Algorithme 2.1 – Algorithme Breadth First Search

A chaque itération, l'algorithme produit les successeurs de l'ensemble "new" des nouveaux états découverts à l'itération précédente. Ce travail est réalisé par un calcul de l'image de l'ensemble new par la fonction de transition Δ (ligne 4). Parmi tous les successeurs, seuls les nouveaux états sont conservés (ligne 4). Au début de l'algorithme, new contient l'état initial ι du circuit (ligne 2). Le point fixe est atteint lorsque l'ensemble des nouveaux états est vide (ligne 3).

Cet algorithme explore les états atteignables par couche successive comme l'illustre la figure

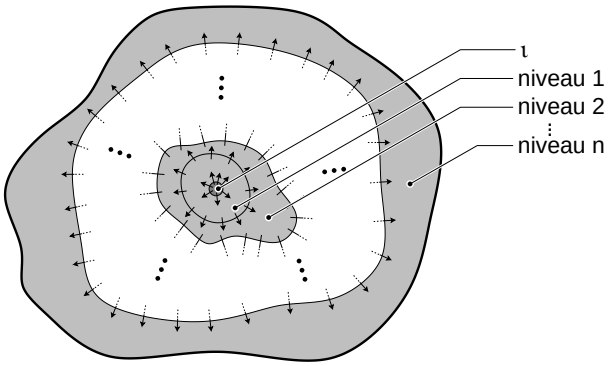


FIG. 2.9 – *Algorithme de base du calcul des états atteignables.*

2.9. Par construction, la couche numéro k contient l'ensemble des états accessibles dès la $k^{\text{ème}}$ réaction du circuit.

2.4 Calcul symbolique des états atteignables

L'implantation de l'algorithme de calcul des états atteignables requiert l'usage de structures de données capable de modéliser des ensembles, des fonctions de transitions et permettant de calculer l'image d'un ensemble par une fonction.

Représentation symbolique des ensembles. Le calcul symbolique des états atteignables repose sur l'utilisation de fonctions booléennes :

$$\begin{array}{rrcl} f: & \mathbb{B}^K & \rightarrow & \mathbb{B} \\ & X & \rightarrow & f(X) \end{array}$$

où X est le vecteur de dimension K des paramètres de f. De telles fonctions permettent de représenter symboliquement des ensembles dans $\mathbb{B}^K$: implicitement, f est la fonction caractéristique de l'ensemble $S_f = \left\{x \in \mathbb{B}^K \ / \ f(x)\right\}$. L'ensemble des états atteignables d'un circuit, autrement dit l'ensemble des valuations valides de ses registres est ainsi représenté.

Les opérations propres aux ensembles comme l'union, l'intersection, la soustraction et le calcul de l'ensemble complémentaire peuvent également être définies à l'aide d'opérations booléennes. Ainsi, si F et G représentent symboliquement des ensembles, alors :

$$F \cup G = \lambda X \rightarrow F(X) \vee G(X) \tag{2.9}$$
$$F \cap G = \lambda X \rightarrow F(X) \wedge G(X) \tag{2.10}$$
$$F \setminus G = \lambda X \rightarrow F(X) \wedge \neg G(X) \tag{2.11}$$
$$\overline{F} = \lambda X \rightarrow \neg F(X) \tag{2.12}$$

L'ensemble vide est représenté par la fonction $f_{\varnothing} = \lambda X \rightarrow 0$ et son complémentaire, l'ensemble $\mathbb{B}^K$ est représenté par la fonction $f_{\mathbb{B}^K} = \lambda X \rightarrow 1$.

2.4.1 Calcul d'image

Notons Φ un ensemble d'états atteignables. Si R appartient à l'ensemble Φ (c'est à dire si $\Phi(R)$ est vrai) et si R' est l'image de R par la fonction de transition Δ pour une certaine valuation valide des entrées (c'est à dire si $\Upsilon(I)$ est vrai et si $\exists I \ / \ R' = \Delta(I, R)$), alors R' appartient à l'image de Φ par la fonction Δ. L'intégralité de l'image de Φ par Δ est l'ensemble Ψ des états R' tels que R' est l'image d'une certaine valuation des entrées et d'un certain état de Φ. L'ensemble Ψ s'écrit formellement de la manière suivante :

$$\Psi = \lambda R' \rightarrow \Big[\exists I, R \ / \ \Upsilon(I) \ \wedge \ \Phi(R) \ \wedge \ R' = \Delta(I, R)\Big] \tag{2.13}$$

Pour que Ψ soit réellement l'expression symbolique d'un ensemble d'états, la dernière opération consiste à remplacer le nom des variables auxiliaires du vecteur $R' = \langle r'_1 \dots r'_p \rangle$ par le nom des variables de registre du vecteur $R = \langle r_1 \dots r_p \rangle$ dans l'expression de Ψ.

Quantification existentielle. La complexité du calcul symbolique de l'image réside dans la résolution de l'opérateur existentiel. Si $f[x \rightarrow v]$ désigne l'expression booléenne obtenue en remplaçant x par v dans f alors, d'après la décomposition de Shannon, nous avons :

$$\exists x \ / \ f = f[x \rightarrow 0] \vee f[x \rightarrow 1] \tag{2.14}$$

On parle alors d'*élimination* existentielle car les variables quantifiées sont éliminées et remplacées par des constantes. La complexité de cette décomposition est d'ordre exponentielle par rapport au nombre de variables quantifiées comme l'illustre l'exemple suivant :

$$\begin{array}{lcc} & \exists x, y \ / & f(x, y) \\ \Longleftrightarrow & \exists y \ / & f(0, y) \quad \vee \quad f(1, y) \\ \Longleftrightarrow & & f(0,0) \vee f(0,1) \vee f(1,0) \vee f(1,1) \end{array}$$

Quantification existentielle partitionnée. Dans la formule du calcul de l'image (équation 2.13), nous pouvons remplacer l'application de la fonction de transition globale par l'application partitionnées des fonctions suivant les registres [20, 21, 22, 35] (voir l'équation 2.7 à la section 2.2). Ce calcul devient alors :

$$\exists I, R \ / \ \Upsilon(I) \ \wedge \ \Phi(R) \ \wedge \ \bigwedge_{i=1}^{p} r'_i = \delta_i(I_i, R_i) \tag{2.15}$$

Chaque fonction de transition δ_i possède son propre support de variable. L'opérateur existentiel n'est **pas** distributif par rapport à l'opérateur "$\wedge$" ce qui nous interdit de simplifier l'équation précédente par une formule de la forme $\bigwedge_{i=1}^{p} \exists I_i, R_i \,/\, r'_i = \delta_i(I_i, R_i)$. En évitant de rentrer dans les détails, la solution consiste à fractionner l'opération existentielle en cherchant les fonctions ayant des supports communs. Par exemple, une formule comme :

$$\exists x, y, z \,/\, f(x) \wedge g(x, y) \wedge h(x, z)$$

peut se réécrire de la manière suivante :

$$\exists x \,/\, f(x) \wedge \left[\exists y \,/\, g(x, y)\right] \wedge \left[\exists z \,/\, h(x, z)\right]$$

Dans le calcul de l'image, chaque quantification est ainsi appliquée sur un nombre minimal de fonctions de transitions. Par rapport à une élimination existentielle globale, cette technique permet souvent en pratique de briser la complexité exponentielle originale.

L'ordre dans lequel est appliqué l'opérateur existentiel a une influence sur les performances de ce partitionnement. Dans une formule comme :

$$\exists x, y, z \,/\, f(x, y) \wedge g(x, z) \wedge h(y, z)$$

nous pouvons partitionner le calcul de trois manières différentes selon que l'on souhaite quantifier en priorité x, y ou z :

$$\begin{array}{c} \exists x, y \,/\, f(x, y) \wedge \left[\exists z \,/\, g(x, z) \wedge h(y, z)\right] \\ \exists x, z \,/\, g(x, z) \wedge \left[\exists y \,/\, f(x, y) \wedge h(y, z)\right] \\ \exists y, z \,/\, h(y, z) \wedge \left[\exists x \,/\, f(x, y) \wedge g(x, z)\right] \end{array}$$

Des travaux permettent d'améliorer le partitionnement proposé dans [21] afin de déterminer un ordre intéressant. Dans [62, 60, 61], cet ordre est déterminé directement à partir des formules booléennes. Dans [74], des informations de haut niveau permettent de guider et d'améliorer ce calcul.

2.4.2 Cofacteur

Le calcul symbolique de l'image d'un ensemble par une fonction utilise largement des techniques de simplification connues sous le nom de techniques de *cofactoring* [28, 29, 30]. Contrairement à d'autres méthodes de simplification, cette technique s'accompagne de perte (contrôlée) d'information. Le principe est que si la valeur d'une fonction f n'est pertinente que sur un domaine de définition restreint S, alors S peut être utilisé pour simplifier l'expression de f (éventuellement en changeant la valeur de f en dehors de S). Nous notons $f_{\downarrow S}$ le cofacteur de f par l'ensemble S :

$$f_{\downarrow S}(X) \quad = \quad \lambda X \rightarrow \begin{cases} f(X) & \text{si } X \in S \\ ? & \text{si } X \notin S \end{cases} \tag{2.16}$$

La valeur de $f_{\downarrow S}$ en dehors de S n'est pas utilisée et peut valoir n'importe quoi. En pratique, des techniques existent pour simplifier la taille de la représentation de la fonction $f_{\downarrow S}$. La figure 2.10 illustre l'effet du cofacteur sur une fonction.

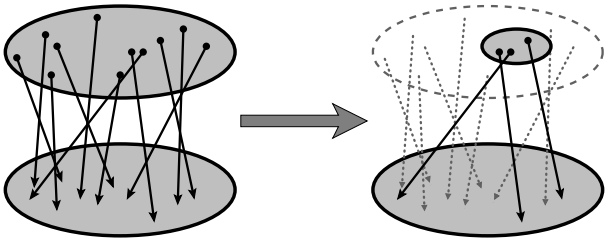


FIG. 2.10 – *Action du cofacteur sur la représentation d'une fonction.*

Le cofacteur dans le calcul de l'image. Dans le calcul de l'image, un opérateur de cofacteur approprié est systématiquement utilisé pour réduire les fonctions de transitions en utilisant l'ensemble des nouveaux états comme domaine de définition. Le calcul de l'image de l'équation 2.15 devient :

$$\exists I, R \,/\, \Upsilon(I) \;\wedge\; \Phi(R) \;\wedge\; \bigwedge_{i=1}^{p} r'_i = \delta_{i\downarrow\Phi}(I_i, R_i) \tag{2.17}$$

En réalité, les fonctions de transition sont construites à la volée à partir de la machine à états finis. De cette manière, les fonctions de transition ne sont jamais complètement représentées afin de simplifier leur représentation. Plus précisément, soit r un registre, si la condition d'activation de r (l'ensemble des états pour lesquels $r = 1$) et le domaine de la fonction de transition sont disjoints, alors la fonction de transition de r peut se réduire à une très simple expression : $\lambda r \rightarrow \neg\, r$. Autrement dit, les fonctions de transition construites à partir de registres inactifs sont très simples.

2.5 Les Diagrammes de Décision Binaires

Afin d'implanter l'algorithme de calcul des états atteignables de la section 2.3, il est nécessaire de représenter les fonctions booléennes $f : \mathbb{B}^K \rightarrow \mathbb{B}$ par des structures de données efficaces, incluant du partage d'information : les *Diagrammes de Décision Binaires* ou *BDDs* [3]. Les BDDs ont été originellement introduits par Lee [54] et Akers [1]. La forme actuelle des BDDs est due à Bryant [18, 19].

2.5.1 Notions de base

Arbres de décision. Un arbre de décision se compose de noeuds de la forme $(v\,?\,t, e)$ où v est une variable booléenne et t et e sont des arbres de décision. Les feuilles d'un arbre de décision sont des constantes booléennes (1 ou 0). Les arbres de décision sont utilisés pour représenter des formules booléennes : tout noeud $(v\,?\,t, e)$ s'interprète comme un opérateur *if-then-else*, c'est à dire "if v then t else e" formellement défini par la formule :

$$(v\,?\,t, e) \;\; = \;\; (v \wedge t) \vee (\neg\, v \wedge e) \tag{2.18}$$

Cette représentation est aussi appelée *If-then-else Normal Form (INF)*. Il est facilement démontrable que toute expression booléenne peut se traduire en expression INF. En effet, un arbre de décision n'est intuitivement rien de plus que la traduction arborescente d'une table de vérité.

Diagrammes de Décision Binaires. Conceptuellement, un Diagramme de Décision Binaire est un arbre de décision dans lequel on aurait supprimé toute information redondante. Ainsi, un BDD n'est plus un arbre d'expression booléennes mais un graphe acyclique orienté (DAG). La figure 2.11 présente un exemple de représentation INF et BDD de la même expression booléenne.

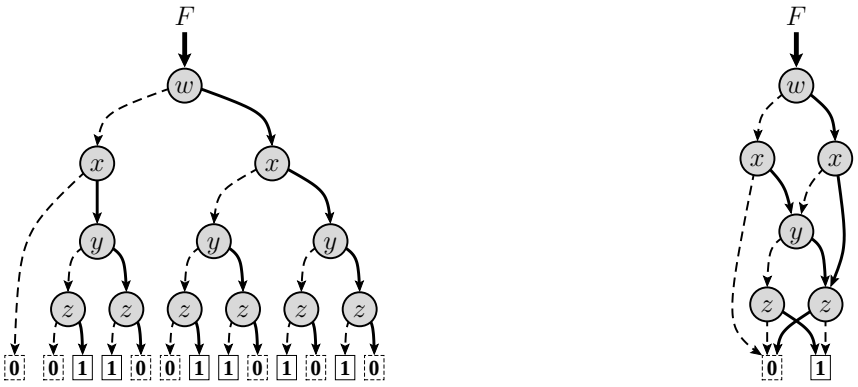


FIG. 2.11 – *Arbre de décision (à gauche) et Diagramme de Décision Binaire (à droite) représentant la même expression booléenne F. Les lignes pleines correspondent aux instances positives des variables, les lignes discontinues aux instances négatives.*

Ordre des variables. Dans un BDD, les variables sont strictement ordonnées de telle sorte que, en parcourant un BDD depuis sa racine :

1. les variables sont toujours placées dans le même ordre, quel que soit le chemin parcouru,
2. chaque variable n'est rencontrée qu'une seule fois.

Le choix de l'ordre des variables est très important dans la construction d'un BDD. Il peut faire la différence entre un BDD de taille polynomiale et un BDD de taille exponentielle. La figure 2.12 présente deux exemples d'ordre de variables pour représenter une même fonction. Trouver le meilleur ordre de variable est un problème NP-complet [75]. Toutefois, il existe des heuristiques qui permettent de trouver un ordre convenable [56, 77, 7].

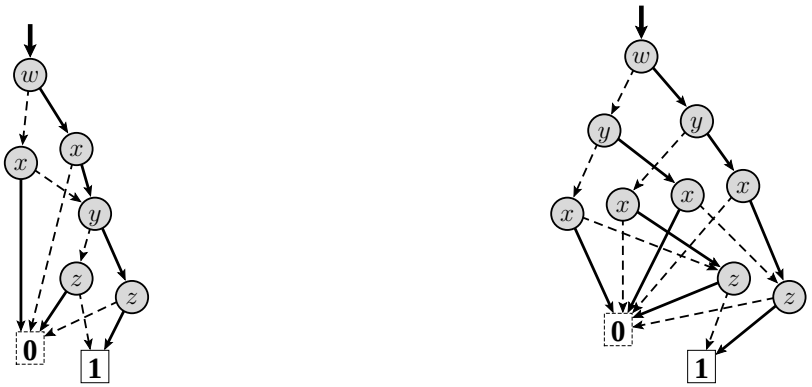


FIG. 2.12 – *Deux BDDs représentant l'expression booléenne $(w = x) \wedge (y = z)$. Le choix de l'ordre des variables a une influence directe sur la taille des BDD.*

Règles de réduction. Conceptuellement, un BDD se construit à partir d'un arbre de décision en appliquant, autant que possible, les règles de réduction suivantes :

1. **Unicité :** Les noeuds identiques sont fusionnés. Deux noeuds distincts ne doivent pas être composés de la même variable et des mêmes sous-graphes :

$$\begin{array}{rl} \text{si} & \mathcal{U} = (v\,?\,t, e) \quad \text{et} \quad \mathcal{V} = (v'\,?\,t', e') \\ \text{alors} & (v = v') \wedge (t = t') \wedge (e = e') \quad \Longrightarrow \quad \mathcal{U} = \mathcal{V} \end{array} \tag{2.19}$$

2. **Non-redondance :** Les noeuds dont les deux sous-graphes sont identiques sont supprimés. Pour tout noeud de BDD de la forme $(v\,?\,t, e)$ on a :

$$t \neq e \tag{2.20}$$

En pratique, les BDDs sont directement construits sous leur forme réduite sans générer l'arbre de décision complet.

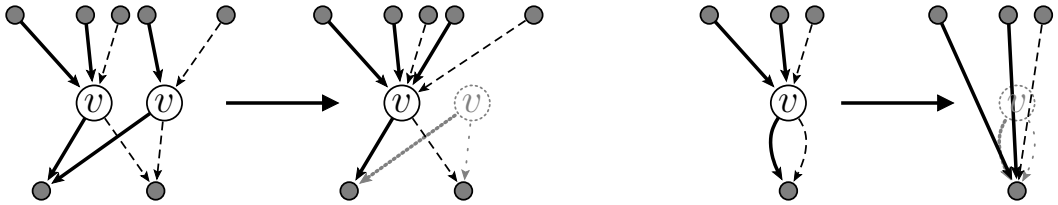


FIG. 2.13 – *Règles de réduction des noeuds de BDDs.*

Canonicité. Pour un ordre de variable donné, la représentation BDD des fonctions booléennes est canonique [18]. Les règles de réduction nous garantissent que deux expressions booléennes équivalentes sont représentées par un graphe unique. Ainsi, la comparaison de deux expressions booléennes est une opération en temps constant qui consiste à vérifier si les racines des graphes sont égales. En particulier, pour vérifier si une expression est une tautologie, il suffit de vérifier si son BDD se réduit à la constante 1.

2.5.2 Raffinements

Marquage des arcs. Une optimisation supplémentaire permet de représenter en même temps une expression et sa négation avec un unique BDD [55, 16]. Pour cela, certains arcs sont marqués d'une négation : si $\mathcal{U}$ est un noeud de BDD, alors nous notons $\neg\mathcal{U}$ le marquage négatif de $\mathcal{U}$. Cette expression représente naturellement la négation de l'expression $\mathcal{U}$. Dans ce formalisme, la constante 0 n'est plus nécessaire car elle se traduit par $\neg\,1$.
Afin de conserver une représentation canonique, il est nécessaire d'imposer une restriction pour le marquage des arcs. Si $(v\,?\,t, e)$ est un noeud de BDD non marqué alors il est interdit de marquer le noeud t. Autrement dit, seuls $(v\,?\,t, e)$ et $(v\,?\,t, \neg\,e)$ sont des noeuds marqués valides. La figure 2.14(a) présente les règles qui permettent de conserver des noeuds marqués sous forme canonique.
Ce raffinement permet d'éliminer quelques noeuds supplémentaires dans les BDDs comme le montre la figure 2.14(b) représentant l'expression $(w = x) \wedge (y = z)$. Il permet en outre de calculer la négation d'une expression en temps constant.

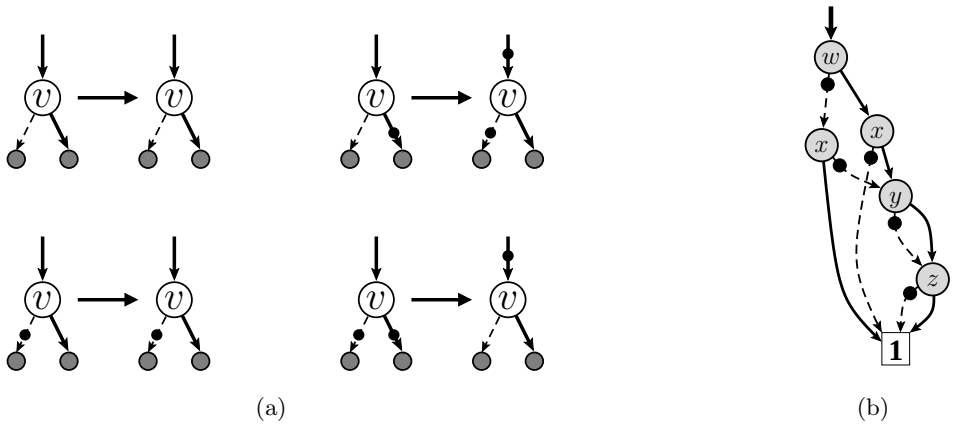


FIG. 2.14 – *Marquage canonique des arcs (a). BDD de* $(w = x) \wedge (y = z)$ *(b).*

Fonctions Booléennes Compactées. Le dernier raffinement concernant le codage des BDD est apporté par l'introduction des *Fonctions Booléennes Compactées* ou *CBF*. Le principe consiste à détecter dans un BDD toutes les équivalences et les non-équivalences entre les variables. La CBF d'un BDD $\mathcal{U}$ se compose :

- d'un BDD $\mathcal{V}$ épuré de toute redondance, de taille inférieure à celle de $\mathcal{U}$,
- d'un ensemble $\mathcal{R} = \{x_1 \sim y_1 \dots x_n \sim y_n\}$ de relations entre variables de la forme $x_i = y_i$ ou bien $x_i \neq y_i$.

Une CBF $(\mathcal{V}, \{x_1 \sim y_1 \dots x_n \sim y_n\})$ représente ainsi l'expression :

$$\mathcal{U} \quad = \quad \mathcal{V} \wedge \left(\bigwedge_{i=1}^{n} x_i \sim y_i \right) \tag{2.21}$$

Sur un exemple précis, l'expression $(w = x) \wedge (y = z)$ se traduit par la CBF : $(1, \{w = x, y = z\})$. Lors d'un calcul entre deux CBFs, cette représentation permet de factoriser l'ensemble des relations communes sans nécessairement reconstruire le BDD original. Cette factorisation n'est pas possible pour n'importe quelle opération logique. Si $\star$ désigne l'un des opérateurs booléens "$\vee$", "$\wedge$" ou "$\wedge\neg$" et si $(\mathcal{U}_1, \mathcal{R}_1)$ et $(\mathcal{U}_2, \mathcal{R}_2)$ sont deux CBFs tels que $r = \{x \sim y\} \in \mathcal{R}_1$ et $r \in \mathcal{R}_2$, alors :

$$\left[\mathcal{U}_1 \wedge \mathcal{R}_1\right] \star \left[\mathcal{U}_2 \wedge \mathcal{R}_2\right] \quad = \quad r \wedge \Big(\left[\mathcal{U}_1 \wedge \mathcal{R}_1 \diagdown r\right] \star \left[\mathcal{U}_2 \wedge \mathcal{R}_2 \diagdown r\right] \Big) \tag{2.22}$$

Les opérateurs désignés par $\star$ permettent de calculer l'union, l'intersection et la soustraction de deux ensembles comme nous l'avons vu à la section 2.4. Les CBFs sont ainsi particulièrement adaptées à la représentation et à la manipulation des ensembles.

Travaux autour des BDDs. Des travaux un peu éloignés des nôtres utilisent des BDD partitionnés [51] pour calculer l'espace des états atteignables [50, 64]. L'idée consiste à partitionner l'espace booléen afin de pouvoir utiliser un ordre des variables différent dans chaque partition. D'autres travaux décrits dans [43, 52, 70] choisissent de modifier localement et dynamiquement l'ordre des variables afin de réduire la taille des BDDs selon les besoins des calculs en évolution.

2.5.3 Calculs symboliques et BDDs

Afin de construire les BDDs dans leur forme réduite, il est nécessaire de s'assurer que chaque noeud $\mathcal{U} = (v\,?\,t, e)$ est unique. Pour cela, nous supposons l'existence d'une table $\mathcal{T}$ qui à chaque triplet $(v\,?\,t, e)$ associe un unique noeud $\mathcal{U}$:

$$\mathcal{T} : \quad (v\,?\,t, e) \quad \rightarrow \quad \mathcal{U}$$

Nous supposons également que l'opération consistant à associer un triplet à un noeud est effectuée en temps constant. En pratique cette complexité peut être obtenue en représentant $\mathcal{T}$ par une table de hachage.

2.5.3.1 Formules propositionnelles

La décomposition de Shannon nous permet de transformer une expression booléenne f en une représentation sous forme de BDD notée $bdd(f)$. Si un ordre sur les variables $v_1, \ldots v_n$ est fixé alors la construction de $bdd(f)$ s'effectue récursivement de la manière suivante :

$$bdd(f) \quad = \quad (v_1\,?\,bdd(f[v_1 \rightarrow 1]), bdd(f[v_1 \rightarrow 0]))$$

La complexité en temps de cette construction est mauvaise car le nombre d'appels récursifs est exponentiel par rapport au nombre n de variables dans f et comme nous l'avons déjà dit, le BDD généré peut être de taille exponentielle.
Il est malheureusement difficile de faire mieux car la construction d'un BDD à partir d'une expression booléenne est un problème NP-complet. Rappelons que tester si une expression booléenne est satisfiable est un problème NP-complet. Dans un BDD, ce test est réalisé en temps constant. Il est donc aussi difficile de trouver un algorithme polynomial capable de transformer une expression booléenne en BDD que de démontrer que P est égal à NP.

2.5.3.2 Opérations basiques

La négation d'un BDD peut se calculer en temps constant (voir section 2.5.2). Si $\star$ désigne un opérateur booléen binaire quelconque, et si $\mathcal{U} = (v\,?\,t_1, e_1)$ et $\mathcal{V} = (v\,?\,t_2, e_2)$ désignent deux noeuds de BDD portant sur la même variable v alors l'application de $\star$ à $\mathcal{U}$ et $\mathcal{V}$ s'effectue de la manière suivante :

$$\mathcal{U} \star \mathcal{V} \quad = \quad (v\,?\,t_1 \star t_2, e_1 \star e_2)$$

On pourra facilement se convaincre de la véracité de cette formule en utilisant la décomposition de Shannon sur $\mathcal{U}$ et $\mathcal{V}$. Pour éviter l'explosion exponentielle du nombre d'appels récursifs, il est possible d'utiliser une table permettant de mémoriser chaque résultat intermédiaire $\mathcal{U}_i \star \mathcal{V}_j$. Nous supposons que l'accès à cette table en fonction de $\mathcal{U}_i$ et $\mathcal{V}_j$ peut être réalisé en temps constant. Si nous appelons m le nombre de noeuds de BDD de $\mathcal{U}$ et n le nombre de noeuds de BDD de $\mathcal{V}$ alors la complexité en temps et en espace du calcul de $\mathcal{U} \star \mathcal{V}$ dans le pire des cas est de l'ordre de $m \times n$.

2.5.3.3 Quantification

Restriction. Les calculs de quantification d'un BDD utilisent un opérateur de restriction. La restriction d'un BDD consiste à remplacer une variable par une constante (0 ou bien 1) dans le

BDD d'origine. Cette opération est réalisée d'après les relations suivantes :

$$\begin{aligned}(v\,?\,t,e)[v\rightarrow 1] &= t\\ (v\,?\,t,e)[v\rightarrow 0] &= e\\ (v'\,?\,t,e)[v\rightarrow b] &= (v'\,?\,t[v\rightarrow b],e[v\rightarrow b])\end{aligned}$$

Cette opération a un coût linéaire en temps et en espace par rapport au nombre de noeuds dans le BDD original.

Quantification existentielle. La quantification existentielle est issue de la décomposition de Shannon comme dans les expressions booléennes de la section 2.4.1 :

$$\exists v\ /\ \mathcal{U} \;=\; \mathcal{U}[v\rightarrow 0]\vee\mathcal{U}[v\rightarrow 1]$$

Nous ne dirons rien sur la quantification universelle qui se déduit facilement de la quantification existentielle. Le coût théorique de cette quantification est égal au coût de deux restrictions ajouté au coût de l'application de l'opérateur "$\wedge$", c'est à dire quadratique par rapport au nombre de noeuds. En pratique, cette opération tend à diminuer le nombre de noeuds dans le BDD quantifié car elle permet de supprimer des variables et donc de réduire le support du BDD.

Quantification dans le calcul de l'image. La complexité en espace du calcul de l'image par les BDDs réside dans le fait que l'expression de l'image repose simultanément sur les variables décrivant les anciennes et les nouvelles valeurs des registres. La relation de transition désigne l'expression :

$$R' \;=\; \Delta(I,R)$$

qui, dans l'expression du calcul de l'image, possède le plus grand support. Les techniques de partitionnement du calcul de l'image présentées à la section 2.4.1 et appliquées aux BDDs n'ont pas pour but principal de simplifier le calcul de la quantification existentielle mais avant tout d'éviter la représentation de la relation de transition complète. En revanche, les moyens mis en place sont les mêmes que ceux décrits dans cette section.

2.5.3.4 Substitution

La substitution de la variable v par le BDD $\mathcal{V}$ dans le BDD $\mathcal{U}$ est l'opération consistant à remplacer chaque occurrence libre de la variable v par l'expression décrite par $\mathcal{V}$ dans l'expression de $\mathcal{U}$. Cette opération est notée :

$$\mathcal{U}[v\rightarrow\mathcal{V}]$$

Dans un BDD, autrement dit une expression INF ne contenant aucun quantificateur, toutes les variables sont libres. Si l'expression $\mathcal{V}$ s'évalue à 1 alors $\mathcal{U}[v\rightarrow\mathcal{V}]$ se réduit en $\mathcal{U}[v\rightarrow 1]$. Sinon, l'expression se réduit en $\mathcal{U}[v\rightarrow 0]$. L'opération de substitution est donc une opération if-then-else dans laquelle le test est conditionné par la valeur de l'expression $\mathcal{V}$. La définition de l'opérateur if-then-else est donnée à la section 2.5.1. Ainsi, nous avons :

$$\mathcal{U}[v\rightarrow\mathcal{V}] \;=\; (\mathcal{V}\wedge\mathcal{U}[v\rightarrow 1])\vee(\neg\,\mathcal{V}\wedge\mathcal{U}[v\rightarrow 0])$$

Substitution dans le calcul de l'image. Le calcul de l'image nécessite un renommage des variables afin d'exprimer l'ensemble des nouveaux états découverts à partir des variables de registres (voir section 2.4.1). L'opérateur de substitution permet de réaliser cette opération.

2.5.3.5 Cofacteur et BDDs

Les opérateurs de cofacteur peuvent s'appliquer aux BDDs. Le cofacteur d'un BDD $\mathcal{U}$ par un BDD $\mathcal{V}$ noté $\mathcal{U}_{\downarrow\mathcal{V}}$ permet de supprimer des noeuds dans $\mathcal{U}$ en restreignant le domaine de $\mathcal{U}$ à l'ensemble $\mathcal{V}$ (voir section 2.4.2). Un algorithme pour cet opérateur a été proposé par Olivier Coudert et al. dans [27]. Cet algorithme permet de supprimer des noeuds dans trois cas. Si le domaine est vide, c'est à dire représenté par le BDD 0, alors le résultat est un BDD constant (0 ou bien 1, nous avons choisi 0) :

$$(v\,?\,t, e)_{\downarrow 0} \quad = \quad 0$$

Les deux autres simplifications s'appliquent si les variables à la racine des BDDs sont identiques et si l'une des deux branches de $\mathcal{V}$ est 0. Supposons par exemple que la branche "then" de $\mathcal{V}$ soit 0. Dans ce cas, la branche "then" de $\mathcal{U}$ peut être remplacée par n'importe quel BDD. Le choix le plus efficace consiste à lui attribuer le même BDD que sa branche "else" et donc à supprimer la racine de $\mathcal{U}$ (voir les règles de simplification des BDDs à la section 2.5.1). Ces simplifications sont formellement données par les relations suivantes :

$$\begin{aligned}(v\,?\,t, e)_{\downarrow(v\,?\,t',0)} &= t_{\downarrow t'} \\ (v\,?\,t, e)_{\downarrow(v\,?\,0,e')} &= e_{\downarrow e'}\end{aligned}$$

Dans le cas général, l'opérateur de cofacteur s'applique comme n'importe quel opérateur booléen binaire. Ainsi, si $\mathcal{U} = (v\,?\,t, e)$ et $\mathcal{V} = (v\,?\,t', e')$ désignent deux noeuds de BDD portant sur la même variable v alors :

$$\mathcal{U}_{\downarrow\mathcal{V}} \quad = \quad (v\,?\,t_{\downarrow t'}, e_{\downarrow e'})$$

Le coût théorique en temps de cet opérateur est proportionnel au produit du nombre de noeuds dans $\mathcal{U}$ et dans $\mathcal{V}$. Le BDD résultant de cette opération est souvent plus simple que $\mathcal{U}$. Dans le pire des cas, l'algorithme proposé par Olivier Coudert peut toutefois produire un BDD plus large que $\mathcal{U}$. Des méthodes permettent d'améliorer cet opérateur et de garantir de ne pas augmenter la taille de $\mathcal{U}$ [44].

Chapitre 3

Présentation Intuitive

L'économie de la consommation mémoire est un enjeu majeur dans l'implémentation des calculs symboliques d'espaces d'états. La consommation mémoire est liée à la taille des BDDs nécessaires aux calculs. Les ressources mémoire sollicitées par les BDDs dans l'algorithme de base rendent l'analyse de certains programmes impossible (à cause du dépassement de la capacité mémoire). Plus précisément, on constate en pratique que les plus gros besoins en mémoire sont transitoires et induits par l'application de la fonction de transition sur un ensemble "provisoire" d'états, lors du calcul de son image. En particulier, les itérations intermédiaires de l'algorithme de base sur des représentations d'ensembles d'états "non saturés" produisent les plus gros BDDs comme le montre la figure 3.1. Ce phénomène peut s'expliquer par le fait que la représentation symbolique d'un ensemble vide est aussi simple que la représentation de l'ensemble de tous les états. De ce fait, l'exploration des états atteignables tend en pratique à simplifier les BDDs dans les dernières étapes de calcul.

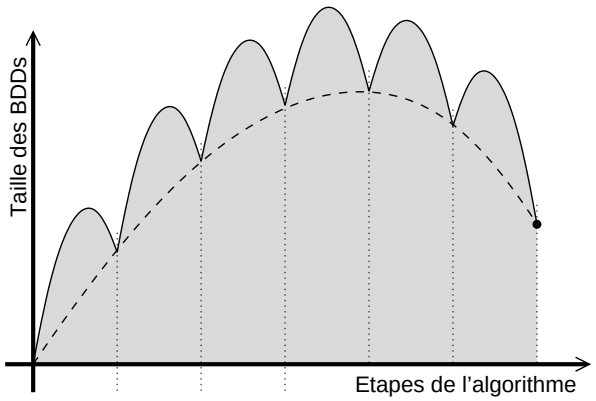


FIG. 3.1 – *Evolution typique de la taille des BDDs dans l'algorithme Breadth First Search. La ligne discontinue représente l'évolution de la taille du BDD des états atteints au cours du calcul. La ligne pleine représente la taille des BDDs nécessaires au calcul de l'image. La consommation tend à diminuer sur la fin des calculs car la représentation des états atteints tend à se régulariser en se saturant.*

Nos travaux visent à réduire ces besoins en mémoire. Notre stratégie a pour but de partitionner le domaine d'application de chaque fonction de transition et de saturer les BDDs intermédiaires au plus tôt afin de :

– diminuer la taille des BDDs nécessaires au calcul de l'image,
– diminuer la taille des BDDs intermédiaires nécessaires à la représentation des états atteints.

Nous nous attendons à ce que cette stratégie nécessite un temps de calcul plus important que par l'algorithme Breadth First Search car la fonction de transition est appliquée un plus grand nombre de fois. Notre stratégie propose donc un compromis entre le temps et l'espace nécessaire aux calculs. Toutefois, il est possible qu'un plus grand nombre d'opérations appliquées à des structures plus petites soit plus rapide qu'un nombre réduit d'opérations appliquées à des objets plus gros. L'évolution espérée de la taille des BDDs au cours des calculs est illustrée par la figure 3.2.

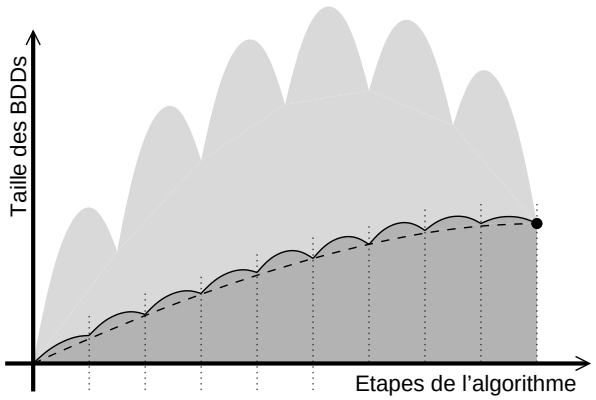


FIG. 3.2 – *Evolution attendue de la taille des BDDs dans l'algorithme partitionné. Les états intermédiaires sont idéalement encodés par des BDDs dont la taille est inférieure à celle du BDD final (ligne discontinue). La courbe représentant la taille attendue des BDDs nécessaires au calcul de l'image (ligne pleine) présente également moins de fluctuations que dans l'algorithme Breadth First Search.*

Pour y parvenir, nous partons de l'observation que toutes les instructions du programme ne sont **pas** actives simultanément à chaque instant dans l'exécution d'un programme. Dans le simple exemple suivant :

present S **then** P **else** Q **end**

l'exécution de P ou Q est conditionnée par la présence d'un signal S. Les états correspondant à l'activation du bloc P sont distincts des états correspondant à l'activation du bloc Q. Les registres des deux blocs ne sont jamais actifs simultanément. Ainsi, le calcul des états atteignables dans P ne nécessite pas l'intégralité de la fonction de transition. En particulier, le bloc Q étant inactif, la représentation de ses comportements dans la fonction de transition n'est pas nécessaire.
Plus généralement, la structure des programmes ESTEREL décrit souvent un enchaînement de blocs (ou de macros-états) dans lesquels seule une partie des registres (et donc des comportements) est active. Ces blocs sont naturellement décrits par la syntaxe des programmes.
Plutôt qu'appliquer une fonction de transition globale avec une politique Breadth First Search, nos travaux visent à appliquer successivement des parties de la fonction de transition. Le but est de savoir combiner les applications de ces parties de fonction de transition afin d'être équivalent

avec l'application de la fonction de transition globale. Pour cela, nous proposons de suivre autant que possible la syntaxe et donc la logique de l'enchaînement du contrôle entre les états décrits par cette syntaxe, comme dans une exécution symbolique. D'une manière générale, nous cherchons à saturer l'exploration de chaque bloc avant d'aller prospecter les comportements activés par la terminaison de ce bloc. Lorsqu'aucun nouvel état ne peut être découvert, le calcul se poursuit avec une autre partie de la fonction qui concerne un autre macro-état du programme. Ainsi, en explorant les états valides bloc après bloc, nous espérons réduire les besoins en mémoire du calcul de l'image.

Notre partitionnement du calcul des états atteignables consiste à construire ces parties de la fonction de transition et à réorganiser les calculs d'image afin de prendre en compte séparément plusieurs fonctions de transition disjointes contre une seule monolithique dans l'algorithme de base. Pour cela, nous nous appuyons sur la structure des programmes source : nous voulons construire progressivement l'espace des états accessibles en suivant la structure algorithmique naturelle du programme.
Alors que le calcul des états atteignables s'effectue sur une représentation circuit, le partitionnement que nous proposons puise son inspiration au niveau du code ESTEREL source. Nous verrons dans un chapitre ultérieur comment nous effectuons la liaison entre ces deux représentations. Pour le moment, nous admettrons que les raisonnements que nous faisons ici au niveau du code source peuvent être reportés au niveau circuit pour partitionner la description de la fonction de transition, puis le calcul des états atteignables.

3.1 Description générale de la méthode

Notre méthode de partitionnement repose sur l'idée que le corps d'un programme peut se diviser suivant des blocs (ou macro-états) de granularité appropriée. Dans des composants dits "séquentiels", les blocs sont combinés en séquence, en boucle ou bien en alternative par le biais d'une instruction *if-then-else*. La recherche partitionnée des états atteignables s'effectue à l'intérieur de chacun de ces blocs individuellement. On ne passe d'un bloc à un autre qu'une fois que l'algorithme s'est stabilisé pour le premier. L'exploration de chaque nouveau bloc démarre à partir d'un nouvel ensemble d'états initiaux, états obtenus à partir de l'exploration des blocs précédents : les états situés exactement sur une frontière entre deux blocs sont en même temps les états finaux du premier bloc et les états initiaux du second (voir figure 3.3). En réalité,

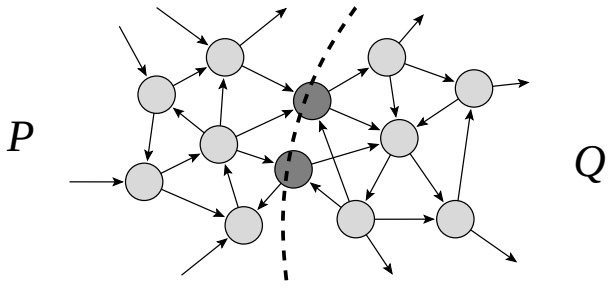


FIG. 3.3 – *Frontière entre deux blocs P et Q. Les états finaux de P servent d'états initiaux à Q.*

les états situés sur une frontière appartiennent clairement au second bloc mais les transitions qui mènent à eux proviennent du premier bloc. Pour interdire l'exploration de certains blocs,

il suffit d'interdire l'activation des registres qui les constituent, ce qui simplifie drastiquement l'expression BDD des fonctions de transition.

Cette méthode soulève un problème en présence de parallélisme, dans le cas où deux frontières peuvent être traversées de manière concurrente dans des composants s'exécutant en parallèle. Le cas est illustré par l'exemple suivant qui contient une frontière entre P_1 et P_2 d'une part et Q_1 et Q_2 d'autre part :

$$\begin{array}{l} \quad P_1\,;\ P_2 \\ \texttt{||} \\ \quad Q_1\,;\ Q_2 \end{array}$$

Prendre alors en compte toutes les combinaisons de blocs possibles, autrement dit le produit cartésien, mènerait à une explosion du nombre de cas possibles : ici $\texttt{[}P_1\ \texttt{||}\ Q_1\texttt{]}$, $\texttt{[}P_1\ \texttt{||}\ Q_2\texttt{]}$, $\texttt{[}P_2\ \texttt{||}\ Q_1\texttt{]}$ ou bien $\texttt{[}P_2\ \texttt{||}\ Q_2\texttt{]}$.

Schéma général. Nous choisissons donc de mettre en place la stratégie suivante : tout d'abord, il est important de trouver un "bon" ordonnancement des frontières afin de suivre la progression naturelle de l'exploration des états. Ensuite, nous commençons avec un nombre minimal de blocs actifs et nous activons progressivement les blocs un à un, à chaque fois que nous décidons d'ouvrir une nouvelle frontière. Quand une frontière a été ouverte, elle n'est plus refermée. Ainsi, la fonction de transition initialement très réduite, ne fait que croître vers la fonction de transition globale complète. Dans le même temps, les parties déjà atteintes par les fonctions de transition antérieures n'auront plus à être considérées. Les fonctions de transition étendues poursuivent l'exploration des états atteignables à partir des dernières frontières ouvertes.

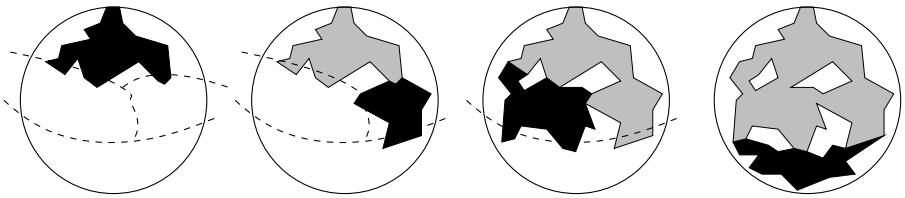


FIG. 3.4 – *Méthode de partitionnement selon quatre blocs de programme. Les frontières entre les blocs (représentées par des lignes discontinues) sont ouvertes une à une et ne sont jamais refermées.*

Nous choisissons de n'ouvrir une frontière que lorsque l'exploration des blocs présentement actifs ne permet plus de découvrir de nouveaux états. De cette manière, les états atteints lors des étapes précédentes du calcul le sont à partir d'une fonction de transition réduite. Dans le même temps, nous n'appliquons les fonctions de transition grandissantes qu'à l'ensemble des états situés à l'extérieur des blocs précédents, car ce sont les seuls capables d'engendrer de nouveaux états. Là-dessus, l'utilisation d'opérateurs de cofacteur nous permet d'espérer alléger la représentation de la fonction de transition en ne conservant que la description des nouveaux comportements, c'est à dire les transitions portant sur des blocs nouvellement activés. L'action combinée des frontières et du cofacteur nous permet d'espérer réduire le calcul des états atteignables en se focalisant à chaque fois sur un nombre de blocs actifs réduit. Cette progression par "vague" est illustrée par la figure 3.4 et la figure 3.5 montre un détail des comportements à la frontière.

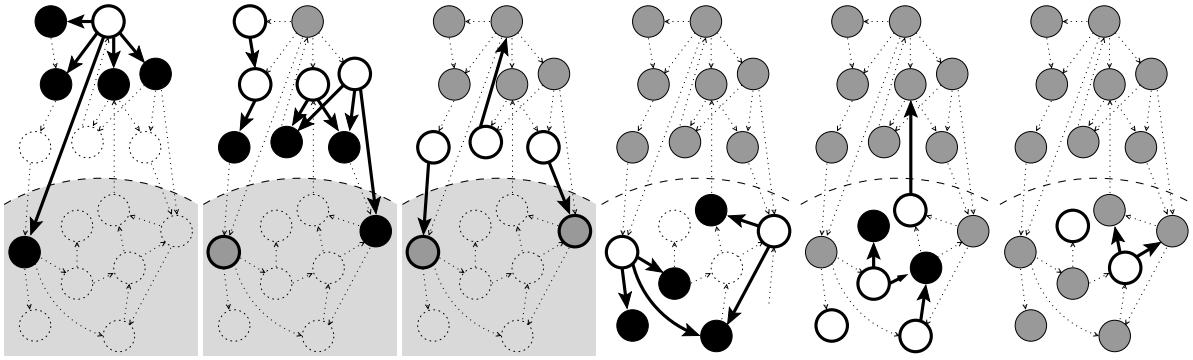


FIG. 3.5 – *Détail de notre méthode de partitionnement sur une frontière entre deux blocs P et Q. Au cours des trois premières étapes, P est entièrement exploré. Les états qui "débordent" hors de P ne sont pas utilisés dans le calcul de l'image. Dans les trois dernières étapes, l'exploration de P et de Q est effectuée à partir des états "en attente" obtenus précédemment. Comme P a été entièrement exploré, cette seconde phase d'exploration ne concerne plus que les états de Q. Dans cette figure, les ronds blancs désignent les états source et les ronds noirs désignent les nouveaux états cible à une étape donnée. Les ronds gris sans cercle représentent les états précédemment atteints et les ronds gris avec cercle représentent les états en attente.*

Support syntaxique. La division du programme en blocs ainsi que la définition de frontières pertinentes dépend fortement de la structure et donc de la syntaxe des programmes. Plus particulièrement, cette division repose principalement sur la réception des signaux comme dans l'exemple :

abort P **when** S

Nous utilisons un graphe de flot de contrôle pour nous aider dans cette tâche. Le graphe est construit directement au dessus de l'arbre syntaxique du programme de telle sorte que l'arbre syntaxique et le graphe de contrôle partagent les mêmes noeuds. Le graphe décrit tous les chemins possibles suivis par le contrôle entre chacune des instructions du programme ESTEREL et en particulier entre les registres qui dans le programme correspondent aux points d'arrêt possibles du contrôle entre les instants (et qui correspondent à l'opérateur `pause` dans le programme source). La frontière entre les blocs actifs et inactifs est alors décrite en sélectionnant un sous-ensemble des arcs du graphe. Au cours des calculs, ce sous-ensemble est amené à varier au fur et à mesure que des frontières sont ouvertes, que des blocs sont activés et que la fonction de transition est étendue, comme nous l'avons décrit précédemment. A partir du graphe courant contenant des arcs ouverts et fermés, chaque macro-étape de l'algorithme itératif consiste à :

- calculer l'ensemble des registres inactifs afin de construire une description BDD de l'ensemble des blocs considérés,
- sélectionner parmi les états dits "en attente" un nouvel ensemble d'états initiaux pour la prochaine étape,
- achever l'exploration à l'intérieur des blocs actifs courants. Les états découverts à l'extérieur des blocs actifs sont placés "en attente",
- une fois l'exploration des blocs actifs courants terminée, d'autres frontières sont ouvertes et de nouveaux blocs sont activés.

L'algorithme se termine une fois que l'ensemble des états en attente est vide. Dans les sections suivantes, nous décrivons les critères du choix de nos frontières en fonction de la syntaxe des

programmes.

3.2 Partitionnement des blocs séquentiels

Dans cette section, nous nous intéressons au découpage des blocs séquentiels dont le traitement est simple. Ce découpage peut s'opérer de manière récursive. De plus, il existe de nombreuses similitudes dans le traitement des opérateurs de séquence, de choix et de préemption. Le partitionnement que nous proposons permet de suivre fidèlement la structure de contrôle du programme source. Nous présentons ici le partitionnement de chacune des constructions séquentielles.

3.2.1 L'opérateur de séquencement

Considérons un programme formé de deux blocs d'instructions composés en séquence :

P ; Q

Si l'ensemble des états atteignables est calculé par un algorithme de recherche Breadth First Search et si la progression à chaque niveau de profondeur est réalisée par l'application d'une fonction de transition globale, alors il se peut que des états de Q soient découverts avant d'avoir épuisé la recherche des états dans P. Le problème concerne plus particulièrement la terminaison de P.

3.2.1.1 Terminaison des blocs de programme

Il y a deux cas où partitionner une séquence ne constitue qu'une dépense d'énergie inutile. Le premier est le cas évident où P ou bien Q ne contient aucune instruction `pause`. Le second est en rapport avec la durée de P.

Blocs de durée fixe. Si le comportement de P est de longueur fixe, alors toutes les terminaisons possibles de P se produisent "à la même date". En d'autre terme, le calcul en largeur des états de P ; Q ne peut exhiber des états de Q que lorsque tous les états de P sont explorés. Par exemple, si P est de la forme :

```
present S then
   pause1
else
   pause2
end;
pause3
```

alors les deux exécutions possibles de P se déroulent toujours en deux instants. Dans ce cas, l'algorithme Breadth First Search se comporte de la manière suivante :

- La première itération permet de découvrir deux états de P, un état dans lequel seule `pause`$_1$ est active et un état dans lequel seule `pause`$_2$ est active.
- La deuxième itération permet de découvrir un état supplémentaire de P dans lequel seule `pause`$_3$ est active.

– A partir de la troisième itération, tous les états de P ont été explorés et l'algorithme passe automatiquement à l'exploration de Q.

Pour le cas où P est un bloc de durée fixe, il n'y a donc aucun intérêt particulier à partitionner explicitement la séquence P ; Q suivant sa syntaxe (mais il n'y a également aucun inconvénient).

Blocs de durée variable. Si le comportement de P est de longueur variable, alors l'algorithme Breadth First Search peut commencer à explorer Q sans que P soit "saturé". On obtient alors des représentations mélangeant "des bouts" de P avec "des bouts" de Q (voir figure 3.6). Dans un exemple plus précis, si P est de la forme :

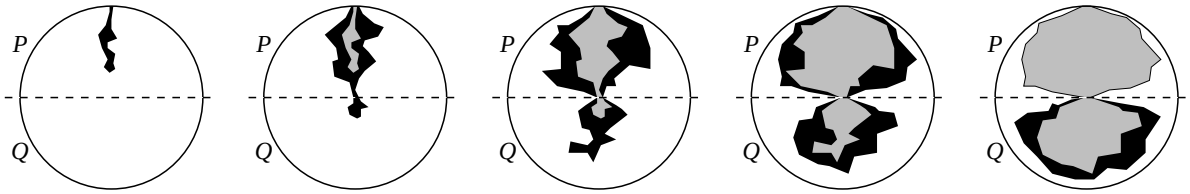


FIG. 3.6 – *Exploration des états atteignables dans une séquence P ; Q. L'exploration de Q commence avant que celle de P soit achevée.*

abort R **when** S

où S est un signal externe, alors Q est potentiellement actif dès le deuxième instant ; à la n-ème itération, l'algorithme de calcul des états atteignables est capable de produire de nouveaux états dans P de profondeur n et de nouveaux états dans Q de profondeur $n-1$.

3.2.1.2 Points faibles de l'algorithme Breadth First Search

L'algorithme de recherche en largeur présente les points faibles suivants :

– La représentation des états atteignables dans les calculs intermédiaires encode en même temps des états de P et de Q. Or, c'est précisément dans les étapes de calcul intermédiaires que la représentation des états atteignables suscite les plus gros BDDs, plus que dans les étapes initiales ou finales de l'algorithme. Ainsi, le fonctionnement de l'algorithme fait que les gros BDDs nécessaires à la représentation de l'ensemble des états provisoirement atteints de P sont automatiquement combinés avec les gros BDDs représentants les états provisoirement atteints de Q. De plus, comme les BDDs représentant les états de P et de Q reposent sur des supports disjoints, la combinaison de ces deux ensembles d'état ne permet pas de bénéficier pleinement du partage de l'information propre aux BDDs ; par conséquent, le résultat d'une telle combinaison est souvent un très gros BDD.

– La représentation de la fonction de transition combine les transitions de P et de Q. Ce point est particulièrement préjudiciable car, dans l'algorithme Breadth First Search, la représentation de la fonction de transition constitue, avec le calcul de l'image qui en découle, l'opération nécessitant les plus gros BDDs.
  La syntaxe du programme nous indique que P et Q s'exécutent en séquence. En d'autre termes, cela signifie que lorsque le bloc P est actif, alors le bloc Q est inactif et réciproquement ; P et Q sont mutuellement exclusifs. Or, cette propriété n'est pas exploitée dans l'algorithme de recherche en largeur.

3.2.1.3 Partitionnement

Partitionner séquentiellement la recherche des états atteignables en traitant intégralement P, puis Q peut nous permettre d'alléger la représentation des ensembles d'états provisoires mais aussi et surtout la représentation de la fonction de transition et le calcul de l'image en se focalisant sur une partie du programme à chaque étape du calcul comme l'illustre la figure 3.7.

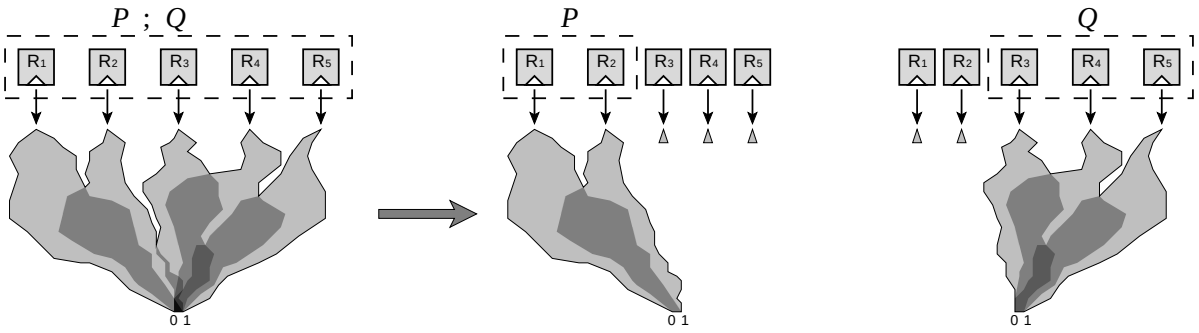


FIG. 3.7 – *Représentation de la fonction de transition de P ; Q dans l'algorithme Breadth First Search (à gauche) et dans l'algorithme partitionné (à droite). P et Q étant indépendants, le partage de l'information dans les BDDs (représenté par les zone sombres) ne s'opère pas bien entre P et Q. Le partitionnement permet de simplifier l'expression de la fonction de transition.*

Dans cet exemple, le partitionnement que nous proposons consiste à découper le processus en deux phases. La première phase consiste à saturer l'exploration des états accessibles à l'intérieur de P uniquement. Il s'agit de bloquer explicitement les transitions correspondant à la terminaison normale du bloc P et interdire ainsi l'exploration des états de Q. De cette manière, seuls les ensembles d'états intermédiaires de P sont encodés et seule la partie de la fonction de transition correspondant au bloc P est utilisée. De manière analogue, la deuxième phase parachève le calcul par la saturation des états de Q. Dans cette deuxième phase, la fonction de transition que nous utilisons encode les comportements de P et de Q, mais comme tous les états de P ont déjà été découverts dans la première phase, l'opérateur de cofacteur permet de ne conserver que les transitions issues du bloc Q.

3.2.2 L'opérateur de choix

Nous reconsidérons à présent l'exemple de l'introduction :

present S **then** P **else** Q **end**

Ici, la situation est très similaire à celle de l'opérateur de séquencement. Si nous supposons qu'aucune des deux branches ne termine instantanément alors les états accessibles dans P et Q peuvent être construits indépendamment alors que l'algorithme de recherche en largeur s'accomplit en parallèle dans les deux blocs. Ici encore, nous proposons de partitionner la recherche des états, d'abord dans P puis dans Q afin de réduire la taille des BDDs nécessaires aux calculs.

Plus concrètement, les calculs se déroulent comme pour la séquence. Dans un premier temps, nous supposons que la fonction de transition a été appliquée de telle sorte que les états initiaux de P et Q ont été construits. Dans un second temps il s'agit de bloquer explicitement l'exploration des états de Q. De cette manière nous saturons P. Dans un troisième temps, nous utilisons l'opérateur de cofacteur sur la fonction de transition globale (qui encode la totalité

des comportements de P et Q) de telle manière que seules les transitions de Q sont réellement encodées.

3.2.3 Le mécanisme de préemption ou d'exception

Notre calcul partitionné des états atteignables consiste à saturer l'exploration d'un bloc avant d'aller prospecter les comportements activés par la terminaison de ce bloc. Considérons l'exemple suivant :

abort P **when** S

dans lequel un programme P est exécuté jusqu'à réception d'un signal `S`. En plus des terminaisons normales liées à l'achèvement du bloc P, l'opérateur de préemption permet d'ajouter des moyens de terminaisons "prématurées". De ce point de vue, ce mécanisme est similaire à celui des exceptions. Pour notre partitionnement du calcul des états atteignables, nous souhaitons bloquer explicitement les transitions correspondant à toutes les terminaisons de P, aussi bien les terminaisons normales que celles engendrées par l'instruction `abort` et ce tant que tous les états de P ne sont pas explorés. Nous voulons donc considérer toutes les transitions sortant de P comme des frontières.
Le fait de bloquer explicitement toutes les terminaisons prématurées de P aura bien évidemment pour effet de bloquer aussi toutes les émissions du signal `S` dans le reste du programme qui pourraient la causer car les émissions et les réceptions de `S` sont en relation directe dans la fonction de transition globale : "si `S` n'est pas reçu c'est qu'il n'a pas été émis".

3.2.4 Découpage de programme séquentiel : un exemple

Considérons le programme séquentiel suivant :

abort P **when** S1;
present S2 **then** Q_1 **else** Q_2 **end**;
R

Ce programme est constitué de trois blocs en séquence P, Q et R. Le bloc Q se décline en deux blocs Q_1 et Q_2 articulés par l'opérateur de choix. L'exploration Breadth First Search de ce programme est illustrée par la figure 3.8. Le partitionnement de la séquence ne produit en

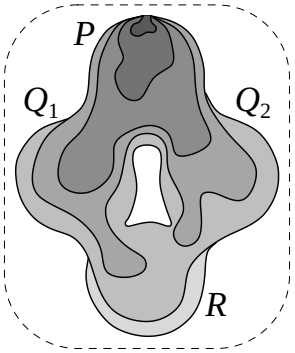


FIG. 3.8 – *Exploration Breadth First Search d'un programme séquentiel.*

lui-même aucune frontière. L'instruction `abort` produit automatiquement des frontières à la sortie de P. Le partitionnement de l'instruction `present` produit des frontières autour de chacune des branches Q_1 et Q_2 (avant et après chaque branche). Certaines des frontières produites automatiquement sont donc redondantes (notamment à la sortie de P et à l'entrée de Q_1 et Q_2) mais cela ne pose pas de problème particulier à notre algorithmique.
L'algorithme partitionné consiste à saturer successivement l'exploration des blocs P, Q_1 (resp. Q_2), Q_2 (resp. Q_1) et R (voir figure 3.9). Précisons que le partitionnement des programmes

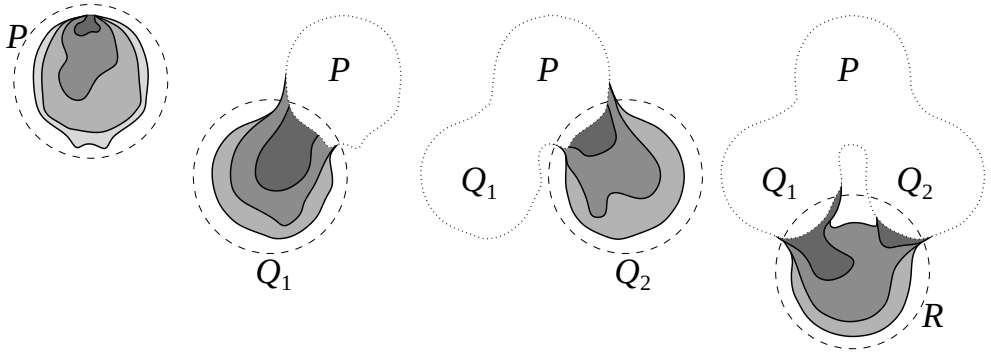


FIG. 3.9 – *Exploration partitionnée d'un programme séquentiel.*

séquentiels est complètement récursif. Chacun des blocs P, Q_1, Q_2 et R peut ainsi être partitionné récursivement. L'ordre de l'exploration des blocs est évident. Pour une séquence, il s'agit de l'ordre des blocs dans la séquence et pour un *if-then-else*, l'ordre n'a pas d'importance. Malheureusement, la prise en compte du parallélisme dans le partitionnement remet en cause tous ces avantages.

3.3 Partitionnement des boucles

Dans un contexte purement séquentiel, sans parallélisme, l'exploration d'un programme de la forme :

loop P **end**

se résume à l'exploration de P puisque la boucle mènera soit aux mêmes états initiaux de P déjà explorés, soit vers des blocs de P complètement inexplorés comme dans le programme suivant :

```
signal S in
   loop
      present S then [ Q1 ; emit S ] else P1 end;
      present S then Q2 else [ P2 ; emit S ] end
   end
end
```

Dans cet exemple, l'exploration du corps de la boucle consiste à explorer successivement P_1, P_2, Q_1 puis Q_2 sans qu'aucune nouvelle frontière ne soit ajoutée par l'instruction `loop`. D'autre part, comme une boucle ne termine jamais spontanément, il est inutile de chercher à partitionner de tels programmes, sauf à l'intérieur de P. Une boucle peut être terminée par un mécanisme

de préemption ou d'exception. Dans ce cas, le partitionnement doit être géré au niveau de l'instruction `abort` (ou `trap`) et non pas au niveau de l'instruction `loop`.

Dans le contexte plus fréquent d'un programme parallèle, le partitionnement des boucles pose quelques difficultés. Le problème vient du fait qu'il est très difficile de savoir quels blocs sont susceptibles d'être actifs simultanément. A cause des synchronisations successives entre les différents blocs, cette information est souvent impossible à obtenir statiquement (ceci explique en grande partie pourquoi le calcul de l'espace des états atteignables peut être aussi complexe). Comme nous l'avons décrit en section 3.1, notre solution actuelle consiste uniquement à élargir la fonction de transition en autorisant à chaque étape l'exploration de nouveaux blocs de programme. Nous comptons sur l'opérateur de cofacteur pour éviter d'encoder le comportement des blocs déjà explorés dans la fonction de transition. Toutefois, cette méthode présente une faiblesse. Si nous considérons l'exemple suivant :

```
   loop P end
||
   Q
```

alors à chaque tour de boucle de P, le bloc Q peut se retrouver dans un état différent. Plus précisément, supposons que P soit de la forme :

P_1 ; P_2 ; **emit** S

et que Q soit de la forme :

Q_1 ; **await** S; Q_2

Le partitionnement de P et Q produit deux frontières. La première se situe entre P_1 et P_2 et la deuxième entre Q_1 et Q_2. Après avoir exploré P_1 et Q_1, nous supprimons la frontière entre P_1 et P_2. Le calcul se poursuit par l'exploration de [P_2 ; `emit S`] et Q_1. La branche P boucle et finit par activer le bloc Q_2. Dans l'exploration de P, il peut arriver que `S` soit émis alors que Q_1 n'est pas terminé mais au bout du compte, tous les états initiaux de Q_2 finissent par être atteints. La seconde frontière est alors ouverte. A partir du deuxième tour de boucle, la frontière entre P_1 et P_2 n'existe plus et la branche P n'est donc plus partitionnée. Ce phénomène est illustré par la figure 3.10. Ce phénomène est un cas bien particulier et nous pouvons espérer que, dans

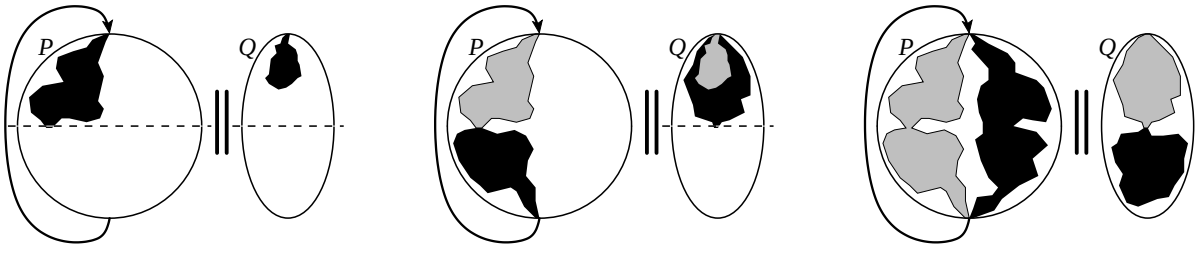


FIG. 3.10 – *Application du partitionnement à une boucle dans un contexte parallèle. Au premier tour de boucle, la frontière de P est ouverte. Ensuite, la frontière de Q est ouverte. Au deuxième tour de boucle de P, la frontière n'existe plus et P est exploré d'un seul coup.*

le cas général, la plupart des états de P sont explorés lors du premier tour de boucle. Cette

supposition pourrait alors nous affranchir du besoin de partitionner P à partir du deuxième tour de boucle. Dans le futur, de plus amples études pourraient nous aider à corriger cette lacune, par exemple en autorisant de refermer certaines frontières après ouverture afin que les frontières soient préservées à chaque tour de boucle.

3.4 L'opérateur parallèle et les signaux

Le rôle de l'opérateur parallèle d'Esterel est d'exécuter ses différentes branches de façon synchrone. Considérons un exemple de programme parallèle à deux branches séquentielles :

```
signal S1, ... Sn in
   P || Q
end
```

Dans ce programme, les branches P et Q sont supposées purement séquentielles. Supposons que le partitionnement de ces branches décompose P en un ensemble de blocs $\{P_1, \ldots P_m\}$ et Q en un ensemble de blocs $\{Q_1, \ldots Q_n\}$. L'absence d'information supplémentaire sur ce programme nous laisse supposer que chaque bloc P_i est susceptible de s'exécuter en même temps que chaque bloc Q_j. Un partitionnement purement structurel de ce programme parallèle consisterait naïvement à découper le calcul des états atteignables suivant le produit cartésien de $\{P_1, \ldots P_m\}$ et de $\{Q_1, \ldots Q_n\}$. Ainsi, chaque partie de la fonction de transition encoderait en même temps les comportements d'un bloc P_i et les comportements d'un bloc Q_j. Par cette technique, le nombre de ces parties de fonction de transition serait de l'ordre de $m \times n$.
Pour un programme Esterel quelconque contenant des instructions parallèles à divers niveaux de profondeur dans le code, cette technique de partitionnement n'est pas satisfaisante : elle engendre rapidement un très grand nombre de morceaux de fonction de transition qu'il s'agit de savoir ordonner et combiner. Nous cherchons ici à exhiber des solutions moins complexes exploitant des relations d'exclusion mutuelle entre les blocs parallèles, comme nous l'avons fait pour les opérateurs séquentiels.

L'opérateur parallèle en lui-même ne nous donne aucune information d'exclusion. Il est toutefois rare de trouver des programmes Esterel dans lesquels de grands composants parallèles s'exécutent de manière purement indépendante. Le programmeur Esterel est souvent amené à faire dialoguer et à synchroniser à l'aide de signaux des tâches lancées en parallèle. De manière plus générale, en Esterel, tout le style de programmation vise à démarrer, tuer ou bien cadencer des modules. Nous proposons donc d'utiliser les signaux comme autant d'éléments synchronisants entre les différentes branches de l'opérateur.

3.4.1 Un programme parallèle au comportement séquentiel

Le partitionnement des programmes parallèles est fondé sur l'idée que l'utilisation de l'opérateur "$\|$" d'Esterel, combiné aux synchronisations et aux préemptions dûes aux signaux internes, peut autoriser certains composants à démarrer, suspendre ou stopper certains autres. Considérons l'exemple suivant, une abstraction de la montre à quartz présentée en introduction :

```
input BUTTON;
signal START_1, START_2, START_3 in

    every START_1           every START_2           every START_3
      abort                   abort                   abort
        run TASK_1    ||        run TASK_2    ||        run TASK_3
      when START_2            when START_3            when START_1
    end                     end                     end

    ||

    pause;
    loop
      emit START_1; await BUTTON;
      emit START_2; await BUTTON;
      emit START_3; await BUTTON;
    end loop

end signal
```

Dans cet exemple illustré par la figure 3.11, quatre programmes apparaissent en parallèle. Trois d'entre eux possèdent la même structure, une tâche principale (TASK$_{-1,2,3}$) qui est démarrée et stoppée en fonction de la réception de signaux START$_{-1,2,3}$. Le quatrième module séquentiel que nous qualifierons de "principal" permet de piloter les trois autres en émettant des ordres de lancement et d'interruption spécifiques à chacun des autres modules. Les tâches TASK_1, TASK_2 et TASK_3 s'exécutent de manière séquentielle malgré la forme "parallèle" apparente du programme global. L'utilisation des signaux permet de faire transiter le programme d'un mode à un autre, chaque mode n'activant que certains blocs du programme. Dans cet exemple simple, les TASKs sont cycliquement exécutés en séquence, mais le scheduler pourrait être un sélecteur plus sophistiqué.

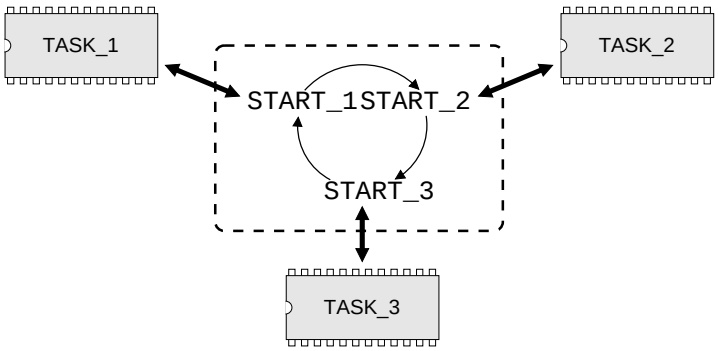


FIG. 3.11 – *Programme parallèle au comportement séquentiel. Au centre, le module principal permet de passer d'un mode actif à un autre par l'envoi de signaux.*

Nous pouvons appliquer à cet exemple le même type de partitionnement que celui décrit pour l'opérateur de séquencement. Le module principal est toujours actif. Par conséquent, le

comportement de ce bloc de programme sera encodé dans chacune des parties de la fonction de transition. Parallèlement à cela, nous proposons de partitionner les calculs en saturant successivement la recherche des états atteignables dans les modules `TASK_1`, `TASK_2` puis `TASK_3`.
Dans un premier temps, nous saturons l'exploration de `TASK_1` en bloquant explicitement les transitions menant aux deux autres modules. En quelque sorte, cela revient à interdire l'émission du signal `START_2`.
Dans un second temps, nous interdisons l'émission du signal `START_3`. A cette étape du calcul, la fonction de transition que nous utilisons encode les comportements de `TASK_1`, `TASK_2` et du module principal. Tous les états de `TASK_1` étant explorés, l'opérateur de cofacteur nous permet de n'appliquer qu'un morceau de notre fonction de transition dépourvu des comportements de `TASK_1`.
Dans un troisième et dernier temps, l'autorisation du signal `START_3`, l'application de la transition globale et l'utilisation de l'opérateur de cofacteur permettent d'achever les calculs par l'exploration du bloc `TASK_3`.

Dans cet exemple, nous voyons comment l'utilisation de signaux peut induire un comportement séquentiel dans un programme écrit comme parallèle. D'une manière plus générale, nous pensons que l'analyse de l'utilisation des signaux nous permet de partitionner le calcul des états atteignables dans les opérateurs parallèles, sans toutefois toujours tomber dans des situations aussi favorables que celle présentée ici (pur séquencement).

3.4.2 Partitionnement des blocs parallèles

Pour partitionner les programmes parallèles nous considérons chaque programme ESTEREL dans sa globalité. Afin d'introduire progressivement les difficultés, cette section présente une collection de petits exemples. Chacun de ces exemples est un cas particulier permettant de mettre l'accent sur une difficulté précise.
Etant donné un programme ESTEREL quelconque, nous nous focalisons sur les couples d'instructions formés d'une instruction émettrice et d'une instruction réceptrice d'un même signal local. Les instructions émettrices sont de la forme "`emit S`" ou bien "`sustain S`". Les instructions de la forme "`present S ...`", "`await S`" ou bien "`abort ... when S`" sont qualifiées de réceptrices. Pour simplifier l'exposition du problème, nous supposerons que chaque signal local n'est émis et reçu qu'à un seul endroit dans tout le programme. De cette manière, à chaque signal local correspond une instruction émettrice et une instruction réceptrice placées dans deux branches parallèles distinctes, comme dans l'exemple ci-dessous :

P_1; || Q_1
emit S; || **await** S
P_2 || Q_2

Pour cet exemple, nous supposons aussi que chaque émission du signal `S` est interceptée par l'instruction `await`. Ce programme est donc équivalent à :

[P_1 || Q_1];
[P_2 || Q_2]

Le partitionnement des programmes parallèles est basé sur l'idée que l'utilisation de chaque signal permet de diviser le programme en deux parties : la première partie concerne tous les

comportements du programme avant l'émission du signal et la deuxième concerne les comportements du programme après l'émission. Il s'agit ici de l'ordre des réactions et non pas de l'ordre des instructions instantanées dans l'instant. Avant l'émission de `S`, seuls les blocs P_1 et Q_1 sont actifs. Lorsque `S` est émis, seuls les blocs P_2 et Q_2 sont actifs.

En partitionnant les programmes parallèles de cette manière, le nombre de partitions obtenues est linéaire par rapport au nombre de signaux locaux. L'exemple suivant présente un parallèle à trois branches synchronisées par deux signaux locaux :

```
P1;              Q1;
emit S1;   ||    await S1;
P2               Q2;              R1;
                 emit S2;   ||    await S2;
                 Q3               R2
```

Ici encore, nous supposons que chaque signal émis est intercepté. L'utilisation des signaux `S1` et `S2` permet de partitionner ce programme en trois parties. Dans la première partie, aucun signal n'est émis. Ainsi, seuls les blocs P_1, Q_1 et R_1 sont actifs. Dans un deuxième temps, `S1` est émis. A la suite de cet événement, seuls les blocs P_2, Q_2 et R_1 sont actifs. Enfin, le signal `S2` est émis et seuls les blocs P_2, Q_3 et R_2 sont actifs.

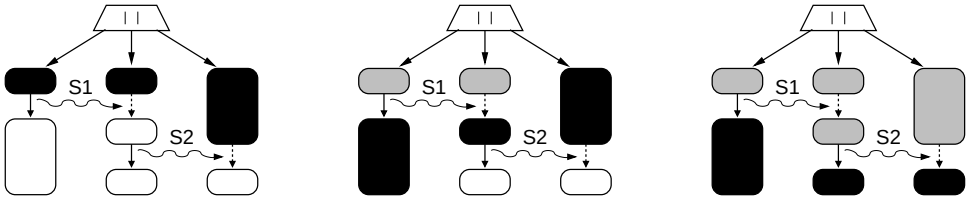


FIG. 3.12 – *Méthode de partitionnement d'un bloc parallèle. Trois blocs en parallèle sont synchronisés par deux signaux. Notre technique vise à partitionner suivant les blocs en noir. Les blocs en gris sont censés être supprimées automatiquement par les méthodes de cofactoring.*

Comme pour le partitionnement des programmes séquentiels, nous faisons grossir progressivement la fonction de transition. Le partitionnement que nous proposons est illustré par la figure 3.12. La première étape consiste à saturer l'exploration du bloc $[P_1||Q_1||R_1]$ avec une fonction de transition dans laquelle nous avons enlevé toutes les transitions consécutives à l'émission et à la réception de `S1` et de `S2`. En pratique, cela revient à dire que nous construisons une fonction de transition qui n'encode que les comportements de P_1, Q_1 et R_1, soit les blocs situés en amont de toute émission et de toute réception de `S1` et `S2`. Dans la deuxième étape, nous rajoutons à la fonction de transition les transitions consécutives à l'émission et la réception de `S1` (nous rajoutons les comportements de P_2 et Q_2 à la fonction de transition). L'application de l'opérateur de cofacteur sur cette nouvelle partie de fonction de transition nous permet ainsi de ne saturer que l'exploration du bloc $[P_2||Q_2||R_1]$. Dans la dernière étape, nous rajoutons les transitions consécutives à l'émission et la réception de `S2`. Nous recouvrons ainsi la fonction de transition globale qui, après utilisation de l'opérateur de cofacteur, nous permet d'achever les calculs par l'exploration du seul bloc $[P_2||Q_3||R_2]$.

3.4.2.1 Partitionnement sur les couples émetteur/récepteur

Pour partitionner les branches parallèles synchronisées par les signaux locaux, il n'est pas nécessaire de partitionner en même temps la branche contenant l'instruction émettrice et la branche contenant l'instruction réceptrice.

Partitionnement suivant l'émission des signaux Partitionner l'exemple précédent suivant l'émission des signaux consiste à découper le programme de la manière suivante :

1. Dans la première étape, nous ne considérons que les blocs situés en amont de toute émission de `S1` et `S2`. Ainsi, la fonction de transition que nous utilisons encode a priori les comportements des blocs P_1, Q_1, Q_2, R_1 et R_2. Or, il est impossible d'exécuter le bloc Q_2 sans exécuter le bloc P_2 et il est impossible d'exécuter le bloc R_2 sans exécuter le bloc Q_3. Ainsi, les blocs Q_2 et R_2 sont automatiquement supprimés de la fonction de transition par l'application de l'opérateur de cofacteur. La fonction de transition n'encode ainsi que les blocs P_1, Q_1 et R_1.
2. Dans la seconde étape, nous débloquons l'émission du signal `S1`. La fonction de transition encode donc a priori les comportements des blocs P_1, P_2, Q_1, Q_2, R_1 et R_2. Pour les mêmes raisons que précédemment, le bloc R_2 n'est pas réellement encodé. Comme nous avons déjà saturé l'exploration des états avant l'émission de `S1`, l'opérateur de cofacteur permet de n'encoder que les blocs P_2, Q_2 et R_1.
3. Dans la troisième étape, nous utilisons l'opérateur de cofacteur sur la fonction de transition globale qui permet de n'encoder que les blocs P_2, Q_3 et R_2.

Partitionnement suivant la réception des signaux De manière similaire, partitionner les programmes parallèles suivant la réception des signaux seulement permet d'obtenir finalement le même résultat. La seule différence se situe dans les blocs qui sont a priori encodés et qui sont automatiquement supprimés par l'opérateur de cofacteur.

1. Dans la première étape, nous choisissons de saturer l'exploration des états atteignables dans les seuls blocs P_1, P_2, Q_1 et R_1. Comme toute émission de `S1` est interceptée par hypothèse, il est impossible que le bloc P_2 soit actif alors que Q_2 est inactif. Ainsi, seuls les blocs P_1, Q_1 et R_1 sont encodés.
2. De la même manière, dans la seconde étape, des blocs P_1, P_2, Q_1, Q_2, Q_3 et R_1 a priori encodés, seuls subsistent les blocs P_2, Q_2 et R_1.
3. Dans la troisième étape, comme toujours, seuls les comportements des blocs P_2, Q_3 et R_2 sont encodés.

De cette manière, nous partitionnons les branches parallèles suivant un point unique du programme. Le fait de partitionner seulement sur l'instruction émettrice ou l'instruction réceptrice nous permet de prendre en compte plus facilement les synchronisations mettant en jeu plus de deux instructions dans un programme. Par exemple, dans le cas où un seul signal (que nous supposons toujours intercepté) permet de synchroniser un programme en deux points, comme c'est le cas dans :

```
loop                    Q1;
   P1;                  await S;
   emit S      ||       Q2
   P2                   await S;
end loop                Q3;
```

Ici, le signal S émis en un seul point dans la première branche est reçu en deux point dans la deuxième branche. Nous proposons de partitionner le calcul des états atteignables suivant les instructions réceptrices. Dans la deuxième branche, cela se traduit par saturer Q_1 seulement, puis saturer Q_2 seulement et enfin saturer Q_3 seulement. Ceci nous permet de partitionner les calculs sans avoir besoin de savoir partitionner la première branche en fonction de la deuxième.

De manière générale, nous choisissons de partitionner les branches parallèles suivant les instructions réceptrices, comme dans l'exemple précédent. Nous considérons toutes les instructions réceptrices y compris les réceptions de signaux externes.

3.4.2.2 Vraies et fausses synchronisations

Dans un programme ESTEREL réel, un signal émis n'est pas forcément intercepté. Avant de chercher à savoir si un signal interne est toujours reçu par une quelconque analyse du programme, nous constatons qu'il n'est pas obligatoirement nécessaire de savoir répondre à cette question pour appliquer notre technique de partitionnement. Dans l'exemple suivant, supposons que le signal S est potentiellement émis trop tôt, avant que le contrôle ne passe dans l'instruction await :

```
P1;                     Q1
emit S;         ||      await S
P2                      Q2
```

Nous partitionnons suivant l'instruction réceptrice. Dans un premier temps, le calcul des états atteignables s'effectue dans les blocs P_1, P_2 et Q_1, avant réception de S. Comme le signal S peut être émis sans être intercepté, nous ne savons pas dire si le bloc P_2 sera ou non encodé dans la fonction de transition après usage de l'opérateur de cofacteur. Néanmoins, cette étape permet de saturer l'exploration des états dans lesquels le bloc Q_1 est actif. Dans un deuxième temps, nous explorons les états atteignables après réception de S. A cette étape des calculs, nous sommes certains que S a été émis. Nous savons alors que l'opérateur de cofacteur permettra de ne pas encoder le bloc P_1 dans la fonction de transition. Le calcul des états atteignables s'effectue donc uniquement dans les blocs P_2 et Q_2. Si le signal S n'est jamais reçu, cette deuxième étape ne produit aucun nouvel état.
Il peut arriver aussi qu'un signal soit émis trop tard, comme dans l'exemple suivant où le bloc Q_1 peut terminer avant le bloc P_1 :

```
P1;                     abort Q1
emit S;         ||      when immediate S;
P2                      Q2
```

Encore une fois, nous partitionnons suivant l'émission réceptrice. Dans la première étape du calcul, l'exploration des états atteignables s'effectue a priori dans les blocs P_1, P_2 et Q_1. Ici,

nous savons qu'il est impossible d'émettre `S` sans activer le bloc Q_2. Autrement dit, cela signifie que le bloc P_2 est forcément inactif. Dans cette première étape, seuls les blocs P_1 et Q_1 seront encodés dans la fonction de transition. Nous saturons ainsi l'exploration des états dans lesquels le bloc Q_1 est actif. Dans la deuxième étape, l'opérateur de cofacteur permet de ne pas encoder le bloc Q_1 dans la fonction de transition. Dans cet exemple, le fait que le bloc Q_2 soit actif ne signifie pas que le signal `S` ait été émis. La fonction de transition est donc susceptible d'encoder le bloc P_1 en plus des blocs P_2 et Q_2.

Pour partitionner un programme parallèle suivant un couple d'instructions formé d'une instruction émettrice et d'une instruction réceptrice d'un même signal, il n'est donc pas nécessaire de savoir si toute émission du signal est interceptée. Si le programme est conçu de telle sorte que toute émission du signal `S` est reçue, alors le partitionnement que nous proposons permet de suivre fidèlement le comportement du programme à l'exécution. Dans le cas contraire, le partitionnement peut paraître plus artificiel, car partitionner selon Q_1 et Q_2 n'entraînera pas de conséquence précise sur la branche P. Dans tous les cas, il est bénéfique de partitionner de cette manière. Dans le meilleur des cas, le calcul des états atteignables concernera tout d'abord P_1 et Q_1 et ensuite, P_2 et Q_2. Dans le pire des cas, cela concernera P_1, P_2 et Q_1 et ensuite P_2 et Q_2. Nous proposons donc de partitionner le calcul des états atteignables suivant toutes les réceptions de signaux internes sans discrimination. Comme pour le partitionnement des programmes séquentiels, cela revient donc à placer des frontières autour des branches des instructions `present` et sur toutes les terminaisons des instructions `abort` ou `trap`.

3.5 Exploration partitionnée des programmes ESTEREL

Le calcul partitionné des états atteignables suivant les blocs de programme s'appuie sur un graphe de flot de contrôle dans lequel les frontières entre les blocs sont représentées par un sous-ensemble des transitions du graphe. Les frontières qui sont ouvertes progressivement permettent de guider l'exploration des états atteignables afin de suivre autant que possible la structure du programme source.
Les frontières du graphe sont construites à partir du programme ESTEREL source en suivant la syntaxe du programme. Chaque frontière correspond à une réception de signal. Ainsi, les frontières sont générées par les instructions de choix (`present`) ainsi que par les instructions de préemption ou d'exception (`abort` ou `trap`) bien qu'en réalité, seules les frontières situées entre les instructions `pause` nous permettent de partitionner le programme.
Au début de ce chapitre, nous avions évoqué le besoin de savoir ouvrir les frontières dans le "bon" ordre en demeurant évasif sur la question. La syntaxe du programme peut nous aider à définir un ordre a priori idéal qui suit strictement la syntaxe du programme. Toutefois, le parallélisme des programmes, les boucles et les diverses synchronisations réalisées par les envois de signaux impliquent que cet ordre défini statiquement ne peut être que partiel. L'ordre total de l'algorithme sera donc défini en grande partie de manière dynamique, au vu des résultats en évolution.

3.5.1 Ordonnancement statique des frontières

Dans un programme de la forme :

abort P **when** S

nous souhaitons saturer P avant d'explorer les états accessibles après la réception de `S`. Pour cela, nous introduisons une relation d'ordre statique et partielle sur les frontières. Ici, toutes les frontières construites dans le bloc P doivent être antérieures aux frontières relatives à la réception de `S`. Cette relation d'ordre est utilisée pour ordonner les frontières a priori, avant de décider pour chacune d'elle si elle doit ou non être ouverte. De la même manière, nous définissons un ordre partiel pour toutes les constructions du langage. Ainsi, dans :

P ; Q

toutes les frontières dans P sont antérieures aux frontières dans Q. Cet ordonnancement se définit à partir du programme ESTEREL source. Notre but étant l'exploration exhaustive des états d'un programme, cet ordre est quasiment l'ordre d'apparition des instructions dans le texte du programme, excepté pour les instructions parallèles et `present`. Pour ces deux constructions en effet, nous ne souhaitons pas imposer d'ordre a priori entre les différentes branches.
On aurait tout aussi bien pu imaginer un ordre dont le but serait de converger le plus rapidement possible vers une zone particulière du programme. Si le but du calcul des états atteignables était de confirmer ou d'infirmer une propriété particulière, comme par exemple l'émission d'un signal particulier, alors nous aurions tout intérêt à ouvrir en priorité les frontières menant le plus rapidement à la partie du programme directement concernée.
L'ordre que nous avons choisi est donc celui qui privilégie une exploration exhaustive des états. Il est formellement défini à la section 4.3.

3.5.2 Ordonnancement du déblocage des frontières

Contrairement au partitionnement des opérateurs séquentiels qui permet d'ouvrir les frontières en suivant fidèlement la syntaxe du programme, l'ordre de l'ouverture des frontières dans le partitionnement des programmes parallèles peut s'avérer complexe ou arbitraire. Le problème provient des synchronisations. La difficulté consiste à débloquer les frontières dans un ordre intéressant. En particulier, nous cherchons à ne pas débloquer des frontières trop tôt. Il n'est pas forcément facile de définir cet ordre statiquement même si des indications partielles importantes sont souvent déductibles. Dans l'exemple ci-dessous, les émissions et les réceptions des signaux `S1` et `S2` sont croisées.

```
loop                    loop
   P1;                     emit S2;
   emit S1;                await S1;
   P2;            ||       Q
   await S2;               end loop
   P3
end loop
```

Ici, la première émission de `S2` n'est pas interceptée. Le signal `S2` n'est reçu qu'à partir du deuxième tour de boucle. Si nous débloquons `S2` avant `S1`, alors nous libérons prématurément l'accès au bloc P_3 et de ce fait, le partitionnement suivant la réception du signal `S2` ne s'effectue pas. Pour partitionner correctement le calcul des états atteignables de ce programme il faut donc débloquer la réception du signal `S1` avant la réception du signal `S2`.
Plutôt que de chercher un critère permettant d'ordonner statiquement le déblocage des frontières, nous choisissons de résoudre le problème de manière dynamique, c'est à dire en s'appuyant

sur le calcul partitionné des états atteignables. La solution consiste à débloquer uniquement les frontières qui permettent de faire progresser les calculs. Ces frontières sont de fait assez facilement identifiables car nous aurons stocké les états en attente dans notre algorithme (voir figure 3.5 ou 3.13). Dans l'exemple précédent, le fait de débloquer la réception de `S2` avant la réception de `S1` ne permet pas de faire progresser les calculs. Pour débloquer les bonnes frontières, nous nous appuyons sur l'ensemble des états situés à la frontière entre les blocs actifs et les blocs inactifs. Il s'agit en réalité d'états accessibles depuis un bloc actif en une réaction instantanée, mais qui "débordent" dans un bloc inactif.

3.5.3 Débordement des états atteignables

L'état d'un programme ESTEREL est caractérisé par l'état de tous les registres booléens qui le constituent. Plus particulièrement, l'ensemble des registres actifs permet de déterminer l'ensemble des blocs actifs du programme. Lorsque l'algorithme partitionné produit des états qui débordent à l'extérieur d'une frontière, nous savons déterminer précisément l'ensemble des blocs activés par ce débordement. A partir de cette information et à l'aide du graphe de flot de contrôle il est alors facile de déterminer les frontières qui ont été franchies. Cette technique est illustrée par la figure 3.13.

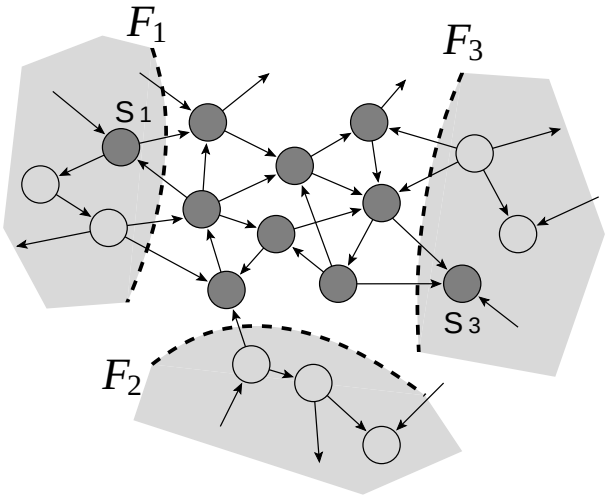


FIG. 3.13 – *Débordement des états atteignables. Les frontières F_1, F_2 et F_3 sont situées autour des blocs actifs courants. La frontière F_2 n'a pas été franchie et doit demeurer fermée. La découverte des états S_1 et S_3 en dehors des blocs actifs indique que les frontières F_1 et F_3 peuvent être ouvertes.*

Ainsi, le débordement des états permet de guider l'algorithme partitionné et de n'ouvrir que les frontières qui permettent de faire progresser l'exploration des états atteignables. De cette manière, nous évitons d'ouvrir des frontières prématurément.

Chapitre 4

Notations

Ce chapitre à pour but de formaliser la mise en oeuvre algorithmique des notions décrites au chapitre précédent. Dans la section 4.1, nous introduisons une notation permettant de représenter les programmes ESTEREL sous forme d'arbre syntaxique. A partir de cet arbre, la section 4.2 décrit la construction d'un graphe de flot de contrôle contenant des frontières. Enfin, la section 4.3 définit une relation d'ordre entre les frontières de ce graphe.

4.1 L'arbre de syntaxe abstraite

Pour notre étude, nous nous restreindrons à un sous-ensemble du langage ESTEREL dans lequel nous n'avons conservé qu'un noyau caractéristique des instructions du langage original. Les variables et la gestion des données ont été supprimées par abstraction comme il est d'usage pour les techniques liées au model-checking vu comme exploration des états de contrôle. Un programme complet est composé d'une en-tête de déclarations définissant l'ensemble des signaux d'entrée et de sortie, suivie d'un corps dont la syntaxe est définie par la grammaire suivante :

$$
\begin{array}{rcl}
P & ::= & \texttt{nothing} \\
 & | & \texttt{pause} \\
 & | & P \texttt{ ; } P \\
 & | & \texttt{loop } P \texttt{ end} \\
 & | & P \parallel P \\
 & | & \texttt{emit } S \\
 & | & \texttt{signal } S \texttt{ in } P \texttt{ end} \\
 & | & \texttt{present } S \texttt{ then } P \texttt{ else } P \texttt{ end} \\
 & | & \texttt{abort } P \texttt{ when } S
\end{array}
$$

Les autres instructions du langage n'introduisent pas de difficultés particulières et ne sont pas nécessaires à la compréhension de notre méthode de partitionnement. Par exemple, **dans le contexte de nos travaux**, l'instruction `trap` peut être traitée de la même manière que l'instruction `abort`. Ces instructions ne font par conséquent pas partie du langage noyau μ-ESTEREL. En revanche, ces instructions ont été prises en compte pour l'implémentation de notre logiciel présenté au chapitre 6.

Avant de commencer la description du graphe de flot de contrôle que nous allons utiliser, nous supposerons que nous possédons une représentation du programme ESTEREL sous forme d'arbre syntaxique. Chaque noeud de l'arbre est typé en fonction de l'instruction qu'il représente mais

nous nous intéressons en fait aux occurrences des instructions dans le programme source. Ainsi, dans l'arbre syntaxique, chaque instance d'instruction sera identifiée de manière unique par un label unique. Au besoin, ces derniers seront explicitement mentionnés en position d'exposant de l'instance d'instruction considérée. Le noeud d'une instruction de type **instruction** portant le label L s'écrit alors de la manière suivante :

$$(\textbf{instruction}^{L}\ \mathit{argument}_1\ \ldots\ \mathit{argument}_n)$$

Les pauses. Le calcul des états atteignables s'obtient à partir d'une représentation des programmes ESTEREL sous forme de circuit alors que le partitionnement que nous proposons s'appuie sur la syntaxe du langage source. Il est donc nécessaire de savoir relier un programme source à sa représentation sous forme de circuit. Dans la traduction d'ESTEREL en circuits que nous utilisons, chaque instruction `pause` produit exactement un registre booléen. Dans le format circuit, un registre est identifié de manière unique par son nom. Ce nom apparaît explicitement dans chaque instruction `pause` de l'arbre syntaxique ce qui nous procure l'association nécessaire entre le code source et le format circuit. Ainsi, une instruction `pause` est représentée par un noeud de la forme :

$$(\textbf{pause}^{L}\ \mathit{register\_id})$$

où *register_id* représente le nom du registre généré par la traduction de l'instruction `pause`. Dans un arbre syntaxique, chaque `pause` possède évidemment un *register_id* distinct.

Les instructions atomiques. Dans l'arbre syntaxique, les instructions `nothing` et `emit` sont naturellement représentées par des feuilles de la forme :

$$(\textbf{nothing}^{L})$$

$$(\textbf{emit}^{L}\ \mathit{signal\_id})$$

où *signal_id* représente le nom du signal dans la traduction en circuit. Le nom des signaux et le nom des registres constituent les seuls liens entre l'arbre syntaxique et la traduction sous forme de circuit mais contrairement aux registres, le nom des signaux n'est pas utilisé par notre technique de partitionnement.

Les constructions hiérarchiques. Une déclaration de signal local est représentée par un noeud de la forme :

$$(\textbf{signal}^{L}\ \mathit{signal\_id}\ \mathit{instruction}^{l}\ \textbf{end}^{L'})$$

où *instruction* est un sous-arbre représentant le corps de l'instruction `signal`. La fin de la portée est marquée par le délimiteur **end**. Par la suite, nous verrons que ce type de délimiteur est utilisé dans la construction du graphe de flot de contrôle.
De la même manière, une boucle est représentée par un noeud de la forme :

$$(\textbf{loop}^{L}\ \mathit{instruction}^{l}\ \textbf{end}^{L'})$$

Une séquence est un noeud de la forme :

$$(\textbf{seq}^{L}\ \mathit{instruction}^{l_1}\ \mathit{instruction}^{l_2}\ \textbf{end}^{L'})$$

Deux blocs s'exécutant en parallèles se notent :

$$(\mathbf{par}^{L}\ \mathit{instruction}^{l_1}\ \mathit{instruction}^{l_2}\ \mathbf{end}^{L'})$$

Les deux instructions permettant de réagir à la présence d'un signal sont `present` et `abort`. Elles sont représentées par des noeuds de la forme :

$$(\mathbf{present}^{L}\ \mathit{signal\_id}\ \mathit{instruction}^{l_{then}}\ \mathit{instruction}^{l_{else}}\ \mathbf{end}^{L'})$$

$$(\mathbf{abort}^{L}\ \mathit{signal\_id}\ \mathit{instruction}^{l}\ \mathbf{end}^{L'})$$

4.2 Le graphe de contrôle

Notre graphe de flot de contrôle est un graphe orienté construit au dessus de l'arbre syntaxique des programmes ESTEREL. Etant donné un arbre T, le graphe de flot de contrôle est défini de la manière suivante :

$$\mathsf{G}(\mathsf{T}) \quad = \quad (\mathsf{I}\ ,\ \mathsf{O}\ ,\ \mathsf{N}\ ,\ \mathsf{E}\ ,\ \mathsf{F})$$

dans lequel :

- N représente l'ensemble des noeuds du graphe. Ces noeuds sont communs avec ceux de l'arbre syntaxique et sont par conséquent typés.
- I, sous-ensemble de N, représente l'ensemble des noeuds initiaux du graphe.
- O qui est aussi un sous-ensemble de N, représente l'ensemble des noeuds finaux du graphe.

Les arcs du graphe se divisent en deux catégories :

- E désigne l'ensemble des arcs "normaux".
- F désigne l'ensemble des arcs considérés comme frontières dans le graphe. Par construction, l'ensemble $\mathsf{E} \cap \mathsf{F}$ est vide.

Ainsi, les arcs correspondant aux transitions des instructions `present` ou `abort` sont placés dans F. Ces arcs sont naturellement appelés "frontières". Les autres arcs sont placés dans E.
Dans E et F, les arcs orientés représentent le chemin du contrôle entre les instances d'instructions et sont de la forme :

$$\mathit{instruction}^{l_1} \mapsto \mathit{instruction}^{l_2}$$

En réalité, la description du graphe utilise les labels qui permettent d'identifier les noeuds de manière plus légère. L'arc précédent est donc noté :

$$l_1 \mapsto l_2$$

Le noeud source d'un arc x est noté $\mathit{Src}(\mathsf{x})$. Le noeud de destination est noté $\mathit{Dest}(\mathsf{x})$:

$$\mathit{Src}(\mathsf{u} \mapsto \mathsf{v}) \quad = \quad \mathsf{u} \tag{4.1}$$

$$\mathit{Dest}(\mathsf{u} \mapsto \mathsf{v}) \quad = \quad \mathsf{v} \tag{4.2}$$

4.2.1 Construction du graphe

Construire le graphe de flot de contrôle consiste à construire les arcs entre les noeuds de l'arbre syntaxique. Ce travail s'effectue de manière structurelle à partir de l'arbre syntaxique et en s'appuyant sur l'ensemble des noeuds initiaux et finaux du graphe. L'opérateur traditionnel

"×" permet de joindre chaque élément d'un ensemble $\mathsf{U} = \{\mathsf{u}_1, \ldots \mathsf{u}_m\}$ à chaque élément d'un ensemble $\mathsf{V} = \{\mathsf{v}_1, \ldots \mathsf{v}_n\}$:

$$\mathsf{U} \times \mathsf{V} \;=\; \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \{\mathsf{u}_i \mapsto \mathsf{v}_j\} \tag{4.3}$$

Les instructions atomiques produisent des graphes formés d'un unique noeud et ne contenant aucun arc :

$$\mathsf{G}(\mathbf{pause}^L\ r) \;=\; (\{L\}\ ,\ \{L\}\ ,\ \{L\}\ ,\ \varnothing\ ,\ \varnothing) \tag{4.4}$$
$$\mathsf{G}(\mathbf{nothing}^L) \;=\; (\{L\}\ ,\ \{L\}\ ,\ \{L\}\ ,\ \varnothing\ ,\ \varnothing) \tag{4.5}$$
$$\mathsf{G}(\mathbf{emit}^L\ s) \;=\; (\{L\}\ ,\ \{L\}\ ,\ \{L\}\ ,\ \varnothing\ ,\ \varnothing) \tag{4.6}$$

La déclaration de signaux locaux. Dans le graphe, nous pouvons abstraire le début et la fin des déclarations de signaux. En effet, notre technique de partitionnement ne repose pas sur une analyse fine des émissions et des réceptions des signaux : seules les instructions réceptrices génèrent des frontières. Le graphe d'une déclaration de signal local est donc identique au graphe de son corps P :

$$\mathsf{G}(\mathbf{signal}^L\ s\ P\ \mathbf{end}^{L'}) \;=\; \mathsf{G}(P) \tag{4.7}$$

La séquence binaire. Dans une séquence binaire, les noeuds finaux du premier graphe sont reliés aux noeuds initiaux du second graphe :

$$\begin{aligned} \text{si}\quad \mathsf{G}(P_i) &= (\mathsf{I}_i\ ,\ \mathsf{O}_i\ ,\ \mathsf{N}_i\ ,\ \mathsf{E}_i\ ,\ \mathsf{F}_i) \quad \text{pour}\quad i \in [1,2] \\ \text{alors}\quad \mathsf{G}(\mathbf{seq}^L\ P_1\ P_2\ \mathbf{end}^{L'}) &= \big(\mathsf{I}_1\ ,\ \mathsf{O}_2\ ,\ \mathsf{N}'\ ,\ \mathsf{E}'\ ,\ \mathsf{F}'\big) \\ \text{où}\quad \mathsf{N}' &= \mathsf{N}_1 \cup \mathsf{N}_2 \\ \mathsf{E}' &= \mathsf{E}_1 \cup \mathsf{E}_2 \cup (\mathsf{O}_1 \times \mathsf{I}_2) \\ \mathsf{F}' &= \mathsf{F}_1 \cup \mathsf{F}_2 \end{aligned} \tag{4.8}$$

Les boucles. Une boucle ne termine jamais. L'ensemble des noeuds finaux du graphe est donc vide. Les noeuds finaux du graphe du corps de la boucle sont reliés aux noeuds initiaux :

$$\begin{aligned} \text{si}\quad \mathsf{G}(P) &= (\mathsf{I}\ ,\ \mathsf{O}\ ,\ \mathsf{N}\ ,\ \mathsf{E}\ ,\ \mathsf{F}) \\ \text{alors}\quad \mathsf{G}(\mathbf{loop}^L\ P\ \mathbf{end}^{L'}) &= \big(\mathsf{I}\ ,\ \varnothing\ ,\ \mathsf{N}\ ,\ \mathsf{E}'\ ,\ \mathsf{F}\big) \\ \text{où}\quad \mathsf{E}' &= \mathsf{E} \cup (\mathsf{O} \times \mathsf{I}) \end{aligned} \tag{4.9}$$

L'opérateur parallèle binaire. Les deux branches d'un parallèle sont démarrées au même instant. Par conséquent, le point de départ d'un parallèle est un unique noeud relié aux noeuds

initiaux de chacune de ses branches :

$$\begin{array}{rrcl} & \text{si} \quad \mathsf{G}(P_i) & = & (\mathsf{I}_i\ ,\ \mathsf{O}_i\ ,\ \mathsf{N}_i\ ,\ \mathsf{E}_i\ ,\ \mathsf{F}_i) \quad \text{pour} \quad i \in [1,2] \\ \text{alors} & \mathsf{G}(\mathbf{par}^L\ P_1\ P_2\ \mathbf{end}^{L'}) & = & \left(\{L\}\ ,\ \mathsf{O}'\ ,\ \mathsf{N}'\ ,\ \mathsf{E}'\ ,\ \mathsf{F}'\right) \\ & \text{où} \quad \mathsf{O}' & = & \mathsf{O}_1 \cup \mathsf{O}_2 \\ & \mathsf{N}' & = & \mathsf{N}_1 \cup \mathsf{N}_2 \cup \{L\} \\ & \mathsf{E}' & = & \mathsf{E}_1 \cup \mathsf{E}_2 \\ & & \cup & \{L\} \times (\mathsf{I}_i \cup \mathsf{I}_2) \\ & \mathsf{F}' & = & \mathsf{F}_1 \cup \mathsf{F}_2 \end{array} \tag{4.10}$$

Test de présence d'un signal. Dans une instruction `present`, nous souhaitons placer nos frontières afin d'explorer P_1, puis P_2, puis tous les blocs qui sont exécutés après cette instruction dans le programme. Par conséquent, les frontières sont placées avant et après la branche "then" et la branche "else" :

$$\begin{array}{rrcl} & \text{si} \quad \mathsf{G}(P_i) & = & (\mathsf{I}_i\ ,\ \mathsf{O}_i\ ,\ \mathsf{N}_i\ ,\ \mathsf{E}_i\ ,\ \mathsf{F}_i) \quad \text{pour} \quad i \in [1,2] \\ \text{alors} & \mathsf{G}(\mathbf{present}^L\ s\ P_1\ P_2\ \mathbf{end}^{L'}) & = & \left(\{L\}\ ,\ \left\{L'\right\}\ ,\ \mathsf{N}'\ ,\ \mathsf{E}'\ ,\ \mathsf{F}'\right) \\ & \text{où} \quad \mathsf{N}' & = & \mathsf{N}_1 \cup \mathsf{N}_2 \cup \left\{L, L'\right\} \\ & \mathsf{E}' & = & \mathsf{E}_1 \cup \mathsf{E}_2 \\ & \mathsf{F}' & = & \mathsf{F}_1 \cup \mathsf{F}_2 \\ & & \cup & \{L\} \times (\mathsf{I}_1 \cup \mathsf{I}_2) \\ & & \cup & (\mathsf{O}_1 \cup \mathsf{O}_2) \times \left\{L'\right\} \end{array} \tag{4.11}$$

La préemption. Une instruction `abort` est susceptible de se terminer après chaque instruction `pause` qui la constitue. De telles transitions sont des frontières qui nous aideront à partitionner le calcul des états atteignables et sont donc placées dans l'ensemble F. Pour réaliser notre partitionnement, il est nécessaire que le corps de l'instruction `abort` soit parfaitement isolé par des frontières. Ainsi, les terminaisons naturelles du corps du `abort` sont également considérées comme des frontières :

$$\begin{array}{rrcl} & \text{si} \quad \mathsf{G}(P) & = & (\mathsf{I}\ ,\ \mathsf{O}\ ,\ \mathsf{N}\ ,\ \mathsf{E}\ ,\ \mathsf{F}) \\ \text{alors} & \mathsf{G}(\mathbf{abort}^L\ s\ P\ \mathbf{end}^{L'}) & = & \left(\mathsf{I}\ ,\ \left\{L'\right\}\ ,\ \mathsf{N}'\ ,\ \mathsf{E}\ ,\ \mathsf{F}'\right) \\ & \text{où} \quad \mathsf{N}' & = & \mathsf{N} \cup \left\{L'\right\} \\ & \mathsf{F}' & = & \mathsf{F} \\ & & \cup & \mathsf{O} \times \left\{L'\right\} \\ & & \cup & \left\{l\ /\ (\mathbf{pause}^l\ r) \in \mathsf{N}\right\} \times \left\{L'\right\} \end{array} \tag{4.12}$$

4.2.2 Exemple

Nous présentons ici un exemple de programme ESTEREL avec sa représentation sous forme de graphe de flot de contrôle correspondante (figure 4.1).

```
abort
   loop pause1 end
 ||
   pause2; pause3
when S;
present T then
   pause4;
   [ pause5 || pause6 ]
else
   pause7; pause8
end;
pause9
```

La construction des frontières permet de diviser le programme en quatre blocs. Le premier constitue le corps de l'instruction `abort`, le second et le troisième correspondent à la branche "then" et à la branche "else" du `present` et le dernier bloc est constitué de la dernière `pause` située après l'instruction `present`.

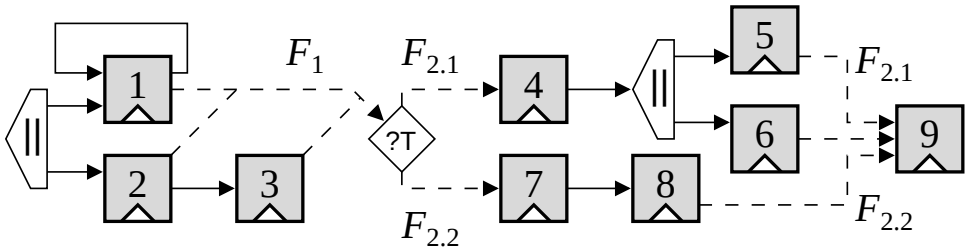


FIG. 4.1 – *Exemple de graphe de flot de contrôle de programme* ESTEREL. *Les frontières* F_1, $F_{2.1}$ *et* $F_{2.2}$ *représentées par les lignes discontinues ont été produites par les instructions* `abort` *et* `present`.

4.2.3 Graphe de contrôle et partitionnement

Dans notre graphe de flot de contrôle, les arcs de type "frontière" permettent de diviser le programme en deux parties : à l'intérieur de la frontière se trouve l'ensemble des blocs actifs que nous souhaitons explorer. A l'extérieur de la frontière se trouve l'ensemble des blocs que nous ne souhaitons pas explorer et qui doivent demeurer inactifs.
La traduction des programmes ESTEREL en circuit produit un registre de contrôle pour chaque instruction `pause`. Cette traduction nous permet de caractériser très précisément l'ensemble des états du programme situés à l'intérieur de la frontière. Si P désigne les blocs de programme situés à l'intérieur de la frontière et si Q désigne les blocs de programme situés à l'extérieur de la frontière, alors les états situés à l'intérieur de la frontière sont les états dans lesquels aucun registre de Q n'est actif. L'intérieur de la frontière n'est donc pas décrit à partir de l'ensemble des registres potentiellement actifs mais à partir des registres que nous forçons à être inactifs.

Codage des blocs actifs. Etant donné un ensemble de variables de BDD $\mathcal{R} = \{r_1, \ldots r_n\}$, nous introduisons l'opérateur $NOr(\mathcal{R})$ défini de la manière suivante :

$$NOr(\mathcal{R}) \quad = \quad \lambda X \rightarrow \neg\, r_1 \wedge \ldots \wedge \neg\, r_n \tag{4.13}$$

Si $r_1, \ldots r_n$ sont des variables représentant les registres booléens $R_1, \ldots R_n$ alors $\mathcal{NO}r(\mathcal{R})$ représente l'ensemble des états dans lesquels tous les registres R_i sont inactifs pour tout $i \in [1..n]$. Nous remarquons que l'ensemble $Or(\mathcal{R}) = \overline{\mathcal{NO}r(\mathcal{R})}$ défini par :

$$Or(\mathcal{R}) \quad = \quad \lambda X \rightarrow r_1 \vee \ldots \vee r_n \tag{4.14}$$

représente l'ensemble des états dans lesquels au moins un registre R_i est actif pour $i \in [1..n]$. L'ensemble des variables de registres inactifs est déterminé à partir du graphe de flot de contrôle, plus particulièrement en considérant les noeuds de type `pause`. Etant donné un ensemble X de noeuds du graphe, nous introduisons l'opérateur $Register\langle \mathsf{X} \rangle$ qui retourne l'ensemble des variables de registres contenues dans les instructions de X :

$$Register\langle \mathsf{X} \rangle \quad = \quad \{r \ / \ (\mathbf{pause}\ r) \in \mathsf{X}\} \tag{4.15}$$

Cet opérateur nous permet d'établir le lien entre le graphe de flot de contrôle et les calculs symboliques à base de BDDs.

Opérations de base dans le graphe de flot de contrôle. Soit $(\mathsf{N}\ ,\ \mathsf{E})$ un graphe "traditionnel" où N représente l'ensemble des noeuds du graphe et E représente l'ensemble des arcs du graphe. Nous notons $Succ_{(\mathsf{N},\mathsf{E})}(\mathsf{X})$ l'ensemble des noeuds successeurs de X, c'est à dire les destinations des arcs de E dont l'origine appartient à X :

$$Succ_{(\mathsf{N},\mathsf{E})}(\mathsf{X}) \quad = \quad \{j \in \mathsf{N} \ / \ i \in \mathsf{X} \ \wedge \ i \mapsto j \in \mathsf{E}\} \tag{4.16}$$

Nous introduisons également l'opérateur $Trans_{(\mathsf{N},\mathsf{E})}(\mathsf{X})$ qui retourne l'ensemble des arcs du graphe dont l'origine appartient à l'ensemble de noeuds X :

$$Trans_{(\mathsf{N},\mathsf{E})}(\mathsf{X}) \quad = \quad \{i \mapsto j \in \mathsf{E} \ / \ i \in \mathsf{X}\} \tag{4.17}$$

Calcul des blocs actifs. L'ensemble des registres situés à l'intérieur de la frontière se calcule à l'aide du graphe de flot de contrôle par fermeture transitive en excluant l'ensemble des arcs de type frontière. L'opérateur $Closure_{(\mathsf{N},\mathsf{E})}(\mathsf{Y})$ permet de calculer l'ensemble des noeuds atteignables à partir d'un ensemble de noeuds initiaux Y en passant par les arcs de l'ensemble E. N représente l'ensemble des noeuds du graphe. La fermeture transitive de Y se calcule par point fixe de la manière suivante :

$$Closure_{(\mathsf{N},\mathsf{E})}(\mathsf{Y}) \quad = \quad \mu(\lambda \mathsf{X} \rightarrow \mathsf{Y} \cup Succ_{(\mathsf{N},\mathsf{E})}(\mathsf{X})) \tag{4.18}$$

où $\mu(f)$ désigne l'opérateur calculant le plus petit point fixe de f.

Calcul de la surface. La fonction suivante calcule la "surface" d'un bloc de programme. A partir d'un ensemble de noeuds $\mathsf{Y} \subseteq \mathsf{N}$ correspondant à un ensemble d'instructions de type `pause`, la surface $Surface_{(\mathsf{N},\mathsf{E})}(\mathsf{Y})$ de Y est l'ensemble des noeuds du graphe qui peuvent être atteints dans le même instant que l'une des instructions contenue dans Y. La surface se calcule selon le même principe que la fermeture transitive, en calculant les successeurs de Y qui ne sont pas de type `pause`. Si P représente l'ensemble de tous les noeuds de type `pause` contenus dans N :

$$\mathsf{P} \quad = \quad \{i \in \mathsf{N} \ / \ i = (\mathbf{pause}\ r)\} \tag{4.19}$$

alors $Surface_{(\mathsf{N},\mathsf{E})}(\mathsf{Y})$ est définie de la manière suivante :

$$Surface_{(\mathsf{N},\mathsf{E})}(\mathsf{Y}) \quad = \quad \mu(\lambda \mathsf{X} \rightarrow \mathsf{Y} \cup (Succ_{(\mathsf{N},\mathsf{E})}(\mathsf{X}) \setminus \mathsf{P})) \tag{4.20}$$

La définition de cette fonction demeure correcte même si tous les éléments de Y ne sont pas de type `pause`.

4.3 Une relation d'ordre pour les frontières du graphe

Cette section a pour but de formaliser les idées présentées à la section 3.5.1. Il s'agit de donner un ordre statique, a priori idéal pour une exploration exhaustive des états atteignables, et partiel sur l'ensemble des frontières du graphe.

4.3.1 Notations

Nous cherchons ici à définir formellement une relation d'ordre stricte notée "$\prec$" entre les frontières du graphe. Cette relation nous permet de définir un ordre pour l'ouverture des frontières. Ainsi, si x et y désignent deux frontières, alors le prédicat : «la frontière x doit être ouverte avant la frontière y» s'écrit :

$$\mathsf{x} \prec \mathsf{y}$$

Cette relation d'ordre se construit avec l'aide de l'arbre syntaxique et complète l'information donnée par le graphe de flot de contrôle. L'arbre syntaxique ne permet pas de définir directement une relation entre les frontières puisque l'arbre ne se compose que d'instructions, autrement dit de noeuds. Ainsi, si u et ν sont deux instructions, alors nous cherchons avant tout à définir un prédicat : «l'accès à l'instruction u doit être ouvert avant l'accès à l'instruction ν». Le prédicat s'écrit de la même manière que précédemment :

$$\mathsf{u} \prec \nu$$

Dès lors, dire que l'accès à l'instruction u doit être ouvert avant l'accès à l'instruction ν est équivalent à dire que tout arc-frontière menant à u doit être ouvert avant tout arc-frontière menant à ν, quelle que soit l'origine des arcs. Ceci s'écrit :

$$\mathsf{u} \prec \nu \iff (\mathsf{x} \mapsto \mathsf{u}) \prec (\mathsf{y} \mapsto \nu) \quad \forall \mathsf{x}, \forall \mathsf{y} \tag{4.21}$$

De cette manière, décrire une relation entre les noeuds de l'arbre est strictement équivalent à décrire implicitement une relation entre les frontières du graphe. Par commodité, nous définissons également l'opérateur "$\ll$" qui permet de définir des relations d'ordre à partir d'ensembles de noeuds. Ainsi, si U et V sont deux ensembles de noeuds, alors l'opérateur "$\ll$" est défini de la manière suivante :

$$\mathsf{U} \ll \mathsf{V} \iff \mathsf{u} \prec \nu \quad \forall \mathsf{u} \in \mathsf{U}, \forall \nu \in \mathsf{V} \tag{4.22}$$

4.3.2 Définition structurelle

La séquence. Comme il est écrit dans la section 3.5.1, dans un programme séquentiel de la forme P ; Q, toutes les frontières de P doivent être ouvertes avant n'importe quelle frontière de Q.

Si nous notons $\mathsf{G}(P_i) = (\mathsf{I}_i\ ,\ \mathsf{O}_i\ ,\ \mathsf{N}_i\ ,\ \mathsf{E}_i\ ,\ \mathsf{F}_i)$ pour $i \in [1,2]$, alors un noeud de séquence dans l'arbre syntaxique de la forme :

$$(\mathbf{seq}^L\ P_1\ P_2\ \mathbf{end}^{L'})$$

induit les relations d'ordre suivantes :

$$\mathsf{N}_1 \ \lessdot\ \mathsf{N}_2 \tag{4.23}$$

La préemption. De la même manière, dans une instruction `abort`, toutes les frontières situées à l'intérieur du bloc doivent être ouvertes avant n'importe quelle frontière menant à la fin du bloc.
Si nous notons $\mathsf{G}(P) = (\mathsf{I}\ ,\ \mathsf{O}\ ,\ \mathsf{N}\ ,\ \mathsf{E}\ ,\ \mathsf{F})$ alors un noeud de l'arbre de la forme :

$$(\mathbf{abort}^L\ s\ P\ \mathbf{end}^{L'})$$

induit les relations d'ordre suivantes :

$$\mathsf{N} \ \lessdot\ \{L'\} \tag{4.24}$$

L'opérateur parallèle. L'opérateur parallèle n'induit aucune relation d'ordre. Il se peut en revanche que le point d'entrée de l'instruction parallèle soit aussi le noeud de destination d'une frontière.
Par conséquent, si nous notons $\mathsf{G}(P_i) = (\mathsf{I}_i\ ,\ \mathsf{O}_i\ ,\ \mathsf{N}_i\ ,\ \mathsf{E}_i\ ,\ \mathsf{F}_i)$ pour $i \in [1,2]$, alors un noeud parallèle de la forme :

$$(\mathbf{par}^L\ P_1\ P_2\ \mathbf{end}^{L'})$$

induit les relations d'ordre suivantes :

$$\{L\} \ \lessdot\ \mathsf{N}_1 \cup \mathsf{N}_2 \tag{4.25}$$

L'opérateur de choix. L'instruction `present` n'induit aucune relation d'ordre entre ses deux branches. Pour les mêmes raisons que précédemment, le point d'entrée et le point de sortie doivent être pris en compte par notre relation d'ordre.
Par conséquent, si nous notons $\mathsf{G}(P_i) = (\mathsf{I}_i\ ,\ \mathsf{O}_i\ ,\ \mathsf{N}_i\ ,\ \mathsf{E}_i\ ,\ \mathsf{F}_i)$ pour $i \in [1,2]$, alors un noeud `present` de la forme :

$$(\mathbf{present}^L\ s\ P_1\ P_2\ \mathbf{end}^{L'})$$

induit les relations d'ordre suivantes :

$$\begin{aligned} \{L\} \ &\lessdot\ \mathsf{N}_1 \cup \mathsf{N}_2 \\ \mathsf{N}_1 \cup \mathsf{N}_2 \ &\lessdot\ \{L'\} \end{aligned} \tag{4.26}$$

En réalité, les relations d'ordre induites par l'opérateur de choix sont plus compliquées qu'il n'y parait. En effet, elles n'interdisent pas d'alterner l'ouverture de frontières dans P_1 et dans P_2. Ainsi, dans un programme de la forme :

present S **then** P **else** Q **end**

rien ne permet d'espérer que le bloc P (resp. Q) sera intégralement exploré avant le bloc Q (resp. P). Pour corriger cela, nous pouvons tout simplement choisir d'ouvrir toutes les frontières de la branche `then` avant toutes les frontières de la branche `else` :

$$\mathsf{N}_1 \lessdot \mathsf{N}_2 \tag{4.27}$$

Ce choix est purement arbitraire car nous aurions très bien pu choisir l'ordre inverse mais il permet d'établir un ordre.

Chapitre 5

Calcul des Etats Atteignables Partitionné

Nous présentons ici notre algorithme partitionné du calcul des états atteignables introduit au chapitre 3 et basé sur le graphe de flot de contrôle des programmes ESTEREL défini au chapitre 4. Notre technique repose sur certaines propriétés des diagrammes de décision binaires (voir chapitre 2), déjà existantes dans l'algorithme originel. Dans la section 5.1, nous présentons notre algorithme partitionné et nous démontrons sa correction dans la section 5.2. Enfin, nous terminons ce chapitre par une analyse de quelques propriétés de notre approche à la section 5.3.

5.1 Algorithme partitionné

L'algorithme présenté dans cette section permet de calculer l'espace des états atteignables d'une machine séquentielle $fsm = (\iota, \Upsilon, \Delta, \Gamma)$. Notre algorithme partitionné est guidé par le graphe de flot de contrôle dans lequel les arcs de type "frontière" sont progressivement débloqués. Dans un premier temps, nous choisissons de présenter une version simplifiée de l'algorithme partitionné dans laquelle toutes les opérations s'appuyant le graphe de flot de contrôle ont été abstraites. L'algorithme complet est donné à la page 66. Ce premier algorithme ne décrit pas la façon dont le graphe est utilisé.

Dans l'algorithme suivant, toutes les variables sont des BDDs ou bien des structures équivalentes comme les CBFs (voir section 2.5.2). Le BDD pending contient les états atteignables dont l'image par la fonction de transition n'a pas encore été calculée. Le BDD area représente l'ensemble de tous les états (atteignables ou non) situés à l'intérieur de la frontière définie par le graphe. Ainsi, à chaque itération de l'algorithme, le calcul de l'image est réalisé uniquement à partir des états de pending qui sont situés à l'intérieur de area (ligne 10). A la fin de chaque itération, les nouveaux états découverts sont placés dans l'ensemble pending (ligne 12).

```
1  reachable ← ι
2  pending ← ι
3  −− Calcul de area_0 voir algorithme 5.2
4  area ← area_0
5  tantque ( pending ≠ ∅ ) faire
6      si ( (pending ∩ area) = ∅ ) alors
7          −− Calcul de area' tel que area' ⊃ area voir algorithme 5.3
8          area ← area'
9      fin si
10     domain ← pending ∩ area
11     new ← Image_Δ(ϒ, domain) ∖ reachable
12     pending ← (pending ∖ area) ∪ new
13     reachable ← reachable ∪ new
14 fin tantque
```

Algorithme 5.1 – Algorithme partitionné

La valeur initiale area_0 de l'ensemble area (ligne 4) est calculée à partir du graphe de flot de contrôle comme nous le verrons par la suite. Nous pouvons remarquer que si area_0 désigne l'ensemble $\mathbb{B}^K$ alors cet algorithme est identique à l'algorithme 2.1 présenté à la section 2.3. Tant que des nouveaux états sont découverts à l'intérieur de l'ensemble area, aucune frontière n'a besoin d'être ouverte (ligne 6). Dans le cas contraire, certaines frontières du graphe sont ouvertes. Un nouvel ensemble area' contenant strictement l'ancien ensemble area est alors calculé à partir du graphe (ligne 8).

5.1.1 Initialisations dans le graphe de contrôle (calcul de area_0)

Nous supposons que l'arbre syntaxique du programme est donné dans T. La phase d'initialisation consiste à construire le graphe de flot de contrôle afin d'obtenir un ensemble d'arcs-frontière fermés initial. A partir de ces conditions initiales, l'ensemble area_0 est initialisé.

```
1  (I , O , N , E , F) ← G(T)
2  ℜ ← Register⟨N⟩
3  inner ← Closure_(N,E) (I)
4  ℜ+ ← Register⟨inner⟩
5  area_0 ← NOr(ℜ ∖ ℜ+)
```

Algorithme 5.2 – Calcul de area_0

La première étape consiste à construire le graphe (ligne 1). A partir du graphe, nous construisons l'ensemble inner contenant l'ensemble des noeuds situés à l'intérieur de la frontière. Cet ensemble est construit par fermeture transitive en partant des noeuds initiaux I et en passant par les arcs de E (ligne 3). L'ensemble $\mathfrak{R}$ contient la totalité des registres du graphe (ligne 2). Seuls les registres situés à l'intérieur de la frontière peuvent être actifs. L'ensemble $\mathfrak{R}^+$ des registres actifs est construit à partir de l'ensemble inner à la ligne 4. Finalement, area_0 est défini comme l'ensemble des états tels qu'aucun registre exceptés ceux de $\mathfrak{R}^+$ n'est actif (ligne 5).

5.1.2 Elargissement des blocs actifs (calcul de area′)

Lorsque l'ensemble area des blocs actifs a besoin d'être élargi, nous voulons ouvrir les "bonnes" frontières. Nous ne voulons ouvrir que les frontières qui nous permettent d'inclure des états en attente de l'ensemble pending à l'intérieur de l'ensemble area des blocs actifs. Les frontières permettant de faire progresser l'algorithme ne peuvent se trouver qu'à la surface des blocs actifs, c'est à dire parmi les arcs de l'ensemble F dont l'origine appartient à l'ensemble surface des noeuds situés à la surface de inner. Par ailleurs, il peut être nécessaire d'ouvrir plus d'une frontière avant que area ne soit suffisamment élargi pour inclure des états en attente. Le cas typique est celui de deux branches parallèles qui attendent la réception d'un même signal :

```
    P1 ; await S ; P2
||
    Q1 ; await S ; Q2
```

Si nous supposons que S est toujours reçu dans chacune des branches de ce programme, alors les deux frontières générées par les deux instructions `await S` doivent être ouverte en même temps. De ce fait, tant qu'aucun état en attente ne se situe à l'intérieur de l'ensemble area, une nouvelle frontière est analysée afin de décider si elle doit être ouverte ou pas. Dans cet algorithmne, la fonction $\mathbf{Sort}_{\prec}(\mathsf{E})$ permet de trier topologiquement un ensemble d'arcs E suivant la relation "$\prec$" :

```
1  surface ← Surface_(N,E∪F) (inner)
2  frontier ← Sort_≺(Trans_(N,F)(surface))
3  i ← 1
4  tantque ( (pending ∩ area) = ∅ ) faire
5      f ← frontier[i]
6      -- Vérifier si f doit être ouverte (variable open?) voir algorithme 5.4
7      si ( open? ) alors
8          -- Ouvrir f voir algorithme 5.6
9      fin si
10     i ← i + 1
11 fin tantque
```

Algorithme 5.3 – Calcul de area′

L'ensemble de noeuds surface représente la surface des blocs situés à l'intérieur de la frontière (ensemble inner à la ligne 1). Les arcs-frontière dont l'origine appartient à surface sont les seuls que nous pouvons ouvrir. Ces arcs-frontière sont triés d'après la relation d'ordre "$\prec$" décrite à la section 4.3 et placés dans l'ensemble frontier (ligne 2). Chaque arc-frontière est ensuite analysé un à un dans l'ordre jusqu'à ce que l'ensemble area soit suffisamment élargi (lignes 4 à 11). Cet ordonnancement nous permet d'analyser et d'ouvrir chaque frontière dans l'ordre que nous jugeons a priori le meilleur.

5.1.2.1 Franchissement des frontières

Pour déterminer si un arc-frontière doit être ouvert, nous nous focalisons sur les nouveaux registres actifs de l'ensemble pending. Les états situés dans l'ensemble pending sont des états

atteignables qui activent des registres situés à l'extérieur de la frontière. Si l'ouverture d'un arc-frontière permet d'inclure un de ces registres à l'intérieur de la frontière, alors l'arc-frontière doit être ouvert :

1 $\mathtt{inner}^{new} \leftarrow Closure_{(\mathsf{N},\mathsf{E})}\,(Dest(\mathsf{f}))$
2 $\mathfrak{R}^{new} \leftarrow Register\langle \mathtt{inner}^{new} \rangle \setminus \mathfrak{R}^{+}$
3 **si** ($\mathfrak{R}^{new} = \varnothing$) **alors** *open*? $\leftarrow$ *true*
4 **si** (pending $\cap\, Or(\mathfrak{R}^{new}) \neq \varnothing$) **alors** *open*? $\leftarrow$ *true*
5 **sinon** *open*? $\leftarrow$ *false*
6 **fin si**

Algorithme 5.4 – Test de franchissement d'une frontière

Dans un premier temps, nous calculons l'ensemble $\mathtt{inner}^{new}$ des noeuds atteignables par l'ouverture de l' arc-frontière courant f. Cet ensemble se calcule par fermeture transitive depuis le noeud destination de f en passant par les arcs de E (ligne 1). En réalité, nous ne nous intéressons qu'aux nouveaux registres découverts dans $\mathtt{inner}^{new}$. Ces nouveaux registres sont placés dans l'ensemble $\mathfrak{R}^{new}$ (ligne 2). Trois cas peuvent alors se présenter :

1. Si l'arc-frontière f ne débouche sur aucun nouveau registre, alors il peut être ouvert mais ceci n'aura aucune influence sur l'élargissement de l'ensemble area (ligne 3).
2. Si l'ensemble $\mathfrak{R}^{new}$ des nouveaux registres n'est pas vide, nous vérifions si l'ensemble pending contient des états ayant activé un ou plusieurs registres de l'ensemble $\mathfrak{R}^{new}$. Dans ce cas, l'arc-frontière f peut être ouvert, ce qui permettra d'élargir l'ensemble area (ligne 4).
3. Si nous ne sommes dans aucun des cas précédents, cela signifie que l'arc-frontière courant débouche sur des registres non activés et qui doivent par conséquent demeurer inactifs (ligne 5). f ne doit pas être ouvert.

5.1.2.2 Sélection des frontières compatibles

L'algorithme précédent est susceptible d'ouvrir des arcs-frontière qui ne sont pas "compatibles" entre eux.

Un exemple. Supposons que r_1, r_2 et r_3 sont trois registres inactifs dont l'accès est maintenu fermé par trois arcs-frontière distincts. L'ensemble pending contient deux états : le premier dans lequel seuls r_1 et r_3 sont actifs et le second dans lequel seuls r_2 et r_3 sont actifs. Nous ouvrons une première frontière qui nous permet d'activer r_1. A cette étape de l'algorithme, rien ne nous interdit alors d'activer r_2 avant r_3 alors que nous préférerions activer seulement r_3.

La solution consiste à effectuer une copie de l'ensemble pending appelée pending$'$ avant de commencer à analyser et à ouvrir nos arcs-frontière (ligne 2). Chaque fois qu'un arc est ouvert, nous réduisons l'ensemble pending$'$ afin de ne conserver que les états "compatibles" c'est à dire les états dont les registres actifs font aussi partie de l'ensemble $\mathfrak{R}^{new}$ des registres que nous sommes sur le point d'activer (ligne 6) :

```
1  ...
2  pending' ← pending
3  tantque ( (pending ∩ area) = ∅ ) faire
4      ...
5      si ( pending' ∩ Or(ℜ^new) ≠ ∅ ) alors
6          pending' ← pending' ∩ Or(ℜ^new)
7          open? ← true
8      ...
```

Algorithme 5.5 – Test de franchissement d'une frontière "compatible"

Dans notre exemple précédent, une fois que r_1 a été activé, il est impossible d'activer r_2 avant r_3 par cette technique.

5.1.2.3 Ouverture d'une frontière

A partir du moment où nous avons décidé l'ouverture d'un arc-frontière, nous devons simplement effectuer quelques mises à jour :

```
1  E ← E ∪ {f}
2  F ← F ∖ {f}
3  inner ← inner ∪ inner^new
4  ℜ+ ← ℜ+ ∪ ℜ^new
5  area ← NOr(ℜ ∖ ℜ+)
```

Algorithme 5.6 – Ouverture d'une frontière

Tout d'abord, l'arc f est déplacé de l'ensemble des frontières F vers l'ensemble des noeuds normaux E (lignes 1 et 2). L'intérieur de la frontière est élargi en conséquence (ligne 3) et les nouveaux registres actifs de $\mathfrak{R}^{new}$ sont ajoutés à l'ensemble des registres actifs de $\mathfrak{R}^+$ (ligne 4). Enfin, l'ensemble area est élargi (ligne 5).

5.2 Correction des algorithmes

Nous définissons formellement l'ensemble des états atteignables d'une machine séquentielle $fsm = (\iota, \Upsilon, \Delta, \Gamma)$ comme le plus petit point fixe d'une fonction Θ définie de la manière suivante :

$$\Theta(X) \quad = \quad \iota \cup \Delta(X) \tag{5.1}$$

Ce point fixe est forcément défini puisque Θ est une fonction croissante et que nous travaillons dans l'ensemble fini $\mathbb{B}^K$. Dans cette section, nous choisissons de ne pas tenir compte de l'ensemble des entrées valides Υ de la machine séquentielle qui ne pose aucun problème particulier sinon rendre la lecture plus difficile.
Si nous appelons R_{bfs} l'ensemble des états atteignables calculé par l'algorithme traditionnel et R_{part} l'ensemble des états atteignables calculé par notre algorithme partitionné, alors le but de cette section est de montrer que :

$$R_{bfs} \quad = \quad \mu(\Theta) \tag{5.2}$$

$$\text{et} \qquad R_{part} \quad = \quad \mu(\Theta) \tag{5.3}$$

```
1  reachable ← ι
2  pending ← ι
3  (I , O , N , E , F) ← G(T)
4  ℜ ← Register⟨N⟩
5  inner ← Closure_(N,E) (I)
6  ℜ+ ← Register⟨inner⟩
7  area ← NOr(ℜ ∖ ℜ+)
8  tantque ( pending ≠ ∅ ) faire
9      surface ← Surface_(N,E∪F) (inner)
10     frontier ← Sort_≺(Trans_(N,F)(surface))
11     i ← 1
12     pending′ ← pending
13     tantque ( (pending ∩ area) = ∅ ) faire
14         f ← frontier[i]
15         inner^new ← Closure_(N,E) (Dest(f))
16         ℜ^new ← Register⟨inner^new⟩ ∖ ℜ+
17         si ( ℜ^new = ∅ ) alors
18             E ← E ∪ {f}
19             F ← F ∖ {f}
20             inner ← inner ∪ inner^new
21         sinon si ( pending′ ∩ Or(ℜ^new) ≠ ∅ ) alors
22             pending′ ← pending′ ∩ Or(ℜ^new)
23             E ← E ∪ {f}
24             F ← F ∖ {f}
25             inner ← inner ∪ inner^new
26             ℜ+ ← ℜ+ ∪ ℜ^new
27             area ← NOr(ℜ ∖ ℜ+)
28         fin si
29         i ← i + 1
30     fin tantque
31     domain ← pending ∩ area
32     new ← Image_Δ(ϒ, domain) ∖ reachable
33     pending ← (pending ∖ area) ∪ new
34     reachable ← reachable ∪ new
35 fin tantque
```

FIG. 5.1 – *Algorithme partitionné complet*

Afin de démontrer la correction de notre algorithme, nous traduisons les algorithmes 2.1 et 5.1 précédents sous forme de fonctions mathématiques récursives.

5.2.1 Rappels et hypothèses

Propriétés du point fixe. Le théorème de Tarski nous permet d'expliciter la valeur de notre point fixe $\mu(\Theta)$:

$$\mu(\Theta) \quad = \quad \lim_{n\to\infty} \Theta^n(\varnothing) \tag{5.4}$$

Comme nous travaillons dans l'ensemble fini $\mathbb{B}^K$, il existe un entier m tel que :

$$\mu(\Theta) \quad = \quad \Theta^m(\varnothing) \tag{5.5}$$

Calcul de $\Theta^n(\varnothing)$. Par commodité, nous définissons la suite Θ_n de la manière suivante :

$$\Theta_n \quad = \quad \Theta^n(\varnothing) \quad \forall n \geq 0 \tag{5.6}$$

Nous cherchons à traduire l'expression de Θ_n sous la forme d'une équation récursive. Nous voulons montrer que :

$$\Theta_n \quad = \quad \Theta_{n-1} \cup \Delta^{n-1}(\iota) \quad \forall n > 0 \tag{5.7}$$

Vrai pour $n = 1$:

$$\begin{aligned} \Theta_1 &= \iota \cup \Delta(\varnothing) \\ &= \iota \cup \varnothing \\ &= \Theta_0 \cup \Delta^0(\iota) \end{aligned}$$

Si vrai pour $n-1$ alors vrai pour n :

$$\begin{aligned} \Theta_n &= \Theta(\Theta_{n-1}) \quad \text{par définition} \\ &= \Theta\left(\Theta_{n-2} \cup \Delta^{n-2}(\iota)\right) \quad \text{par hypothèse de récurrence} \\ &= \iota \cup \Delta\left(\Theta_{n-2} \cup \Delta^{n-2}(\iota)\right) \quad \text{par définition de } \Theta \\ &= (\iota \cup \Delta(\Theta_{n-2})) \cup \Delta^{n-1}(\iota) \quad \text{d'après 5.8} \\ &= \Theta(\Theta_{n-2}) \cup \Delta^{n-1}(\iota) \quad \text{par définition de } \Theta \\ &= \Theta_{n-1} \cup \Delta^{n-1}(\iota) \end{aligned}$$

Vrai pour tout $n > 0$.

Propriétés de la fonction de transition. La fonction de transition Δ est une fonction croissante. Ainsi :

$$\Delta(F \cup G) \quad = \quad \Delta(F) \cup \Delta(G) \tag{5.8}$$

Il est également facilement démontrable que :

$$\Delta(F \smallsetminus G) \quad \supseteq \quad \Delta(F) \smallsetminus \Delta(G) \tag{5.9}$$

Algorithme Breadth First Search. L'algorithme traditionnel se traduit par la définition d'une fonction $bfs\,(R,\ N)$ où la variable R représente l'ensemble des états atteints et N représente l'ensemble des nouveaux états. Le calcul de la fonction se décompose en deux cas selon que l'ensemble des nouveaux états est vide ou non. Ainsi, l'algorithme 2.1 page 21 se traduit de la manière suivante :

$$bfs\,(R,\ N) \quad = \quad \begin{cases} R & \text{si} \quad N = \varnothing \\ bfs\,(R',\ N') & \text{sinon} \end{cases} \tag{5.10}$$

$$\begin{aligned} \text{avec} \quad R' &= (\Delta(N) \cup R) \\ N' &= (\Delta(N) \setminus R) \end{aligned}$$

Cette fonction nous permet de définir R_{bfs} mathématiquement, en calculant l'ensemble des états atteignables à partir de l'état initial ι :

$$R_{bfs} \quad = \quad bfs\,(\iota,\ \iota) \tag{5.11}$$

Algorithme partitionné Notre algorithme partitionné (algorithme 5.1 page 62) se traduit par la définition d'une fonction $part\,(R,\ P,\ A)$ où R représente l'ensemble des états atteints, P représente l'ensemble des états en attente et A représente l'intérieur de notre frontière. L'ensemble A' désigne ici un ensemble quelconque contenant A.

$$part\,(R,\ P,\ A) \quad = \quad \begin{cases} R & \text{si} \quad P = \varnothing \\ part\,(R,\ P,\ A' \supset A) & \text{si} \quad P \neq \varnothing \quad \text{et} \quad P \cap A = \varnothing \\ part\,(R',\ P',\ A) & \text{sinon} \end{cases} \tag{5.12}$$

$$\begin{aligned} \text{avec} \quad R' &= (\Delta(P \cap A) \cup R) \\ P' &= (\Delta(P \cap A) \setminus R) \ \cup \ (P \setminus A) \end{aligned}$$

Fondamentalement, l'algorithme partitionné se décompose en trois cas. Si l'ensemble des états en attente est vide alors la fonction retourne R. Si l'ensemble des états en attente situés à l'intérieur de notre frontière est vide ($P \cap A = \varnothing$), alors nous faisons croître A en A'. Sinon, nous calculons l'image des états en attente situés à l'intérieur de la frontière.
Cette fonction permet de définir R_{part} de la manière suivante, à partir de l'état initial ι et d'un ensemble A initial que nous pouvons supposer vide dans le pire des cas :

$$R_{part} \quad = \quad part\,(\iota,\ \iota,\ \varnothing) \tag{5.13}$$

Nous pouvons par ailleurs remarquer que :

$$part\left(\iota,\ \iota,\ \mathbb{B}^K\right) \quad = \quad bfs\,(\iota,\ \iota) \tag{5.14}$$

5.2.2 Correction de l'algorithme traditionnel

5.2.2.1 Calcul de $bfs\,(R,\ N)$

Soient R_n et N_n les suites décrivant l'évolution de R et N dans les appels successifs de la fonction bfs :

– si $n = 1$

$$\begin{aligned} R_1 &= \iota \\ N_1 &= \iota \end{aligned} \tag{5.15}$$

– si $n > 1$

$$\begin{aligned} R_n &= \Delta(N_{n-1}) \cup R_{n-1} \\ N_n &= \Delta(N_{n-1}) \setminus R_{n-1} \end{aligned} \tag{5.16}$$

5.2.2.2 Convergence de *bfs* (R, N)

Par définition de Θ, $\mu(\Theta) \neq \Theta_0$. $\mu(\Theta)$ converge vers Θ_n dès que $\Theta_n = \Theta_{n+1}$. L'algorithme traditionnel converge vers $R_{bfs} = R_n$ lorsque $N_n = \varnothing$, c'est à dire lorsque $\Delta(N_{n-1}) \subseteq R_{n-1}$, c'est à dire lorsque $R_n = R_{n-1}$. Les suites R_n et Θ_n convergent donc vers un même point fixe $\mu(\Theta)$.

5.2.2.3 $R_n = \Theta_n$

Nous allons montrer que pour tout $n > 0$ nous avons :

$$R_n = \Theta_n \tag{5.17}$$
$$N_n = \Theta_n \setminus \Theta_{n-1} \tag{5.18}$$

Vrai pour $n = 1$:

$$\begin{aligned} R_1 &= \iota \\ &= \Theta_1 \\ N_1 &= \iota \\ &= \Theta_1 \setminus \varnothing \end{aligned}$$

Si vrai pour $n-1$ alors vrai pour n :

$$\begin{aligned} R_n &= \Delta(\Theta_{n-1} \setminus \Theta_{n-2}) \cup \Theta_{n-1} \\ N_n &= \Delta(\Theta_{n-1} \setminus \Theta_{n-2}) \setminus \Theta_{n-1} \end{aligned}$$

$$\begin{array}{ccccc} \Delta(\Theta_{n-1}) \cup \Theta_{n-1} & \supseteq & R_n & \supseteq & (\Delta(\Theta_{n-1}) \setminus \Delta(\Theta_{n-2})) \cup \Theta_{n-1} \\ \Delta(\Theta_{n-1}) \setminus \Theta_{n-1} & \supseteq & N_n & \supseteq & (\Delta(\Theta_{n-1}) \setminus \Delta(\Theta_{n-2})) \setminus \Theta_{n-1} \end{array}$$

Nous pouvons également écrire :

$$\begin{array}{ccccc} (\iota \cup \Delta(\Theta_{n-1})) \cup \Theta_{n-1} & \supseteq & R_n & \supseteq & ((\iota \cup \Delta(\Theta_{n-1})) \setminus (\iota \cup \Delta(\Theta_{n-2}))) \cup \Theta_{n-1} \\ (\iota \cup \Delta(\Theta_{n-1})) \setminus \Theta_{n-1} & \supseteq & N_n & \supseteq & ((\iota \cup \Delta(\Theta_{n-1})) \setminus (\iota \cup \Delta(\Theta_{n-2}))) \setminus \Theta_{n-1} \end{array}$$

Ceci se réécrit en :

$$\begin{array}{ccccc} \Theta_n \cup \Theta_{n-1} & \supseteq & R_n & \supseteq & (\Theta_n \setminus \Theta_{n-1}) \cup \Theta_{n-1} \\ \Theta_n \setminus \Theta_{n-1} & \supseteq & N_n & \supseteq & (\Theta_n \setminus \Theta_{n-1}) \setminus \Theta_{n-1} \end{array}$$

donc :

$$\begin{array}{rcccl} \Theta_n & \supseteq & R_n & \supseteq & \Theta_n \\ \Theta_n \setminus \Theta_{n-1} & \supseteq & N_n & \supseteq & \Theta_n \setminus \Theta_{n-1} \end{array}$$

Donc vrai pour tout n.
Les suites R_n et Θ_n sont égales pour tout $n > 0$.

5.2.3 Correction de l'algorithme partitionné

5.2.3.1 Calcul de *part* (*R*, *P*, *A*)

Soient R_n, P_n et A_n les suites décrivant l'évolution de R, P et A dans les appels successifs de la fonction *part* :

– si $n = 1$

$$\begin{array}{rcl} R_1 & = & \iota \\ P_1 & = & \iota \\ A_1 & = & \varnothing \end{array} \tag{5.19}$$

– si $n > 1$
si $P_{n-1} \neq \varnothing \quad \wedge \quad P_{n-1} \cap A_{n-1} = \varnothing$

$$\begin{array}{rcl} R_n & = & R_{n-1} \\ P_n & = & P_{n-1} \\ A_n & = & A' \supset A_{n-1} \end{array} \tag{5.20}$$

– si $n > 1$
si $P_{n-1} = \varnothing \quad \vee \quad P_{n-1} \cap A_{n-1} \neq \varnothing$

$$\begin{array}{rcl} R_n & = & \Delta(P_{n-1} \cap A_{n-1}) \cup R_{n-1} \\ P_n & = & (\Delta(P_{n-1} \cap A_{n-1}) \setminus R_{n-1}) \cup (P_{n-1} \setminus A_{n-1}) \\ A_n & = & A_{n-1} \end{array} \tag{5.21}$$

5.2.3.2 Convergence de *part* (*R*, *P*, *A*)

Par définition, l'algorithme termine lorsque $P = \varnothing$. Nous allons montrer que la la suite P_n converge vers $\varnothing$.
Supposons qu'il existe un $n > 1$ tel que :

$$\begin{array}{rcccl} R_{n+1} & = & R_n & = & R_{n-1} \\ A_{n+1} & = & A_n & = & A_{n-1} \end{array}$$

Comme $A_n = A_{n-1}$, nous pouvons déduire que R_n vérifie l'équation 5.21. Comme $R_n = R_{n-1}$ nous pouvons déduire que :

$$\Delta(P_{n-1} \cap A_{n-1}) \quad \subseteq \quad R_{n-1}$$

Par simplification de l'équation 5.21, nous avons donc :

$$P_n \quad = \quad P_{n-1} \setminus A_{n-1}$$

et donc :

$$P_n \cap A_{n-1} \quad = \quad P_n \cap A_n \quad = \quad \varnothing$$

Comme $A_{n+1} = A_n$ nous avons :

$$P_n = \varnothing \quad \vee \quad P_n \cap A_n \neq \varnothing$$

Nous savons par ailleurs que $P_n \cap A_n = \varnothing$, nous avons donc :

$$P_n \quad = \quad \varnothing$$

Les suites R_n et A_n sont croissantes et majorées par $\mathbb{B}^K$, donc on démontrerait facilement qu'il existe un entier n tel que $R_{n+1} = R_n = R_{n-1}$ et que $A_{n+1} = A_n = A_{n-1}$. Donc la suite P_n converge vers $\varnothing$.

5.2.3.3 $R_{part} \supseteq \Theta_n$

$\boldsymbol{\forall n > 1, \forall x \in P_{n-1} \quad x \in P_n \vee \Delta(x) \in R_n.}$ Nous voulons montrer que tout élément appartenant à P ne peut sortir de P que lorsque son image est calculée. Si P_n vérifie l'équation 5.20, alors la propriété est évidente. Sinon, deux cas se présentent selon que x appartient ou non à A_{n-1} :

- si $x \in A_{n-1}$ alors :

$$\Delta(x) \in \Delta(P_{n-1} \cap A_{n-1}) \subseteq R_n$$

- si $x \notin A_{n-1}$ alors :

$$x \in P_{n-1} \setminus A_{n-1} \subseteq P_n$$

$\boldsymbol{\forall n > 0 \quad \exists m \ / \ \Delta(P_n) \subseteq R_m.}$ L'image de tout ensemble P_n est contenu dans R_{part}. Démontrons cette propriété par l'absurde et supposons que :

$$\exists x \in P_n \ / \ \forall m \quad \Delta(x) \notin R_m$$

Nous déduisons de la propriété précédente que :

$$\exists x \in P_n \ / \ \forall m > n \quad x \in P_m$$

or cette propriété est fausse puisque nous avons démontré que P_n converge vers $\varnothing$.

$\boldsymbol{\forall n > 0, \forall x \in R_n \quad x \in P_n \vee \Delta(x) \in R_n.}$ Nous voulons montrer que pour tout élément x de R_n, si l'image de x par Δ n'est pas dans R_n, alors x appartient à P_n.
Cette propriété est vraie pour $n = 1$:

$$P_1 = R_1 = \iota$$

Si la propriété est vraie pour $n-1$ alors elle est vraie pour n. Si P_n vérifie l'équation 5.20 alors cette propriété est évidente. Si :

$$\forall x \in R_{n-1} \quad x \in P_{n-1} \vee \Delta(x) \in R_{n-1}$$

alors :

$$\forall y \in R_n \quad y \in P_n \vee \Delta(y) \in R_n$$

Si P_n vérifie l'équation 5.21 alors deux cas se présentent :

- si $y \in R_{n-1}$ alors la propriété est démontrée par hypothèse de récurrence.
- si $y \notin R_{n-1}$ alors $y \in \Delta(P_{n-1} \cap A_{n-1}) \setminus R_{n-1}$ et donc $y \in P_n$.

La propriété est donc vraie pour tout n.

$\forall \boldsymbol{x} \in \boldsymbol{R_{part}} \quad \boldsymbol{\Delta(x)} \in \boldsymbol{R_{part}}$**.** Des deux propriétés précédentes, nous déduisons que l'image de tout élément de R_{part} est aussi contenue dans R_{part}.

$\boldsymbol{R_{part} \supseteq \Theta_n}$**.** Nous montrons que les applications successives de la fonction de transition sont contenues dans l'ensemble R_{part} :

$$\forall n \geq 0 \quad \Delta^n(\iota) \subseteq R_{part}$$

Nous savons que $R_{part} \supseteq \iota$ et par la propriété précédente, nous savons également que si $\Delta^{n-1}(\iota) \subseteq R_{part}$ alors $\Delta^n(\iota) \subseteq R_{part}$ pour tout $n > 0$.
La décomposition de Θ_n donnée à la section 5.2.1 nous permet donc de dire que :

$$R_{part} \quad \supseteq \quad \Theta_n \tag{5.22}$$

5.2.3.4 $\boldsymbol{R_n \subseteq \Theta_n}$

On démontrerait facilement que $R_n \subseteq \Theta_n$ et ainsi que $R_{part} \subseteq \mu(\Theta)$: l'algorithme partitionné ne génère spontanément aucun nouvel état et à chaque étape de l'algorithme, les nouveaux états sont calculés à partir d'anciens états eux-mêmes inclus dans $\mu(\Theta)$.

Pour conclure cette section, nous pouvons affirmer que notre algorithme partitionné est correct et calcule bien le point fixe $\mu(\Theta)$.

5.3 Analyse des caractéristiques

Nous terminons ce chapitre par une brève analyse des caractéristiques de notre algorithme ce qui amène une réflexion sur la forme des programmes pour lesquels cet algorithme est particulièrement adapté, compte tenu de sa complexité empirique. Nous terminons par des commentaires sur la possibilité d'adapter notre technique à un autre type de codage des programmes que celui que nous utilisons (voir 2.1.4).

5.3.1 Complexité

Théoriquement, il est impossible de démontrer que notre algorithme est globalement meilleur que l'algorithme Breadth First Search : dans le pire des cas, ces deux algorithmes sont exponentiels en espace par rapport au nombre de registres, et la complexité en temps de notre algorithme est possiblement beaucoup moins bonne puisqu'elle divise les étapes. Les qualités de notre algorithme ne peuvent être mises en évidence qu'empiriquement (voir chapitre 7).

Soit p la "*profondeur*" du programme exploré, c'est à dire le nombre d'itérations nécessaires pour calculer l'espace des états atteignables par l'algorithme Breadth First Search. Soit r le nombre de registres de ce programme. Dans le pire des cas, l'algorithme partitionné effectue de l'ordre de $p \times r$ itérations (la libération de chacun des r registres peut déclencher à chaque fois p itérations).
Chaque ouverture de frontière nécessite l'appel à des fonctions de manipulation de graphe (*Closure*, *Surface*). Dans le pire des cas, chacune de ces fonctions est polynomiale par rapport au nombre de noeuds du graphe et en pratique très rapide et particulièrement par rapport aux fonctions de manipulation des BDDs.

Le partitionnement mobilise aussi des opérations d'intersection ou de soustraction avec l'ensemble area représentant l'intérieur de notre frontière. L'ensemble area possède une représentation BDD très simple puisqu'il s'agit d'une conjonction de négation de registres. Le BDD de area est donc linéaire par rapport au nombre de registres inactifs et peut se traduire en une CBF de taille constante. En pratique, les opérations combinant un BDD $\mathcal{U}$ avec area produisent des BDDs plus petits que $\mathcal{U}$ mais le temps de calcul qui dépend principalement de la taille de $\mathcal{U}$ n'est en pratique pas négligeable. Notre méthode de partitionnement est donc assez coûteuse en temps mais peu coûteuse en mémoire.

5.3.2 Performances

Les performances de notre algorithme dépendent du programme ESTEREL source. Etant donné que notre technique de partitionnement repose sur la réception des signaux, le partitionnement sera d'autant plus performant que le programme contiendra de nombreuses constructions `present` ou `abort` et de taille suffisamment grande. A l'inverse, un programme contenant de nombreuses boucles combinées en parallèle présentera plus de risque de donner des résultats médiocres. Plus précisément, le problème apparait lorsqu'une instruction `loop` produit de nouveaux états à chaque tour de boucle et lorsque ces nouveaux ensembles d'états ont une représentation BDD irrégulière.
Il est important de remarquer que notre partitionnement ne permet pas de simplifier l'expression des BDDs des registres situés juste derrière la frontière. En effet, la frontière nous permet de restreindre le domaine de définition des fonctions booléennes mais en aucun cas leur image. Seuls les registres situés à une profondeur supérieure ou égale à 2 au delà de la frontière voient leur expression simplifiée.

5.3.3 Encodage des programmes

Nous pensons que notre technique de partitionnement est particulièrement adaptée au cas du langage ESTEREL : il s'agit d'un langage impératif dans lequel les zones actives et inactives du programme sont clairement définies par la valeur des registres. Le codage des programmes sous forme de circuit qui produit un registre booléen par instruction `pause` est également très adapté à notre problème. Cela nous permet de représenter simplement l'ensemble des blocs actifs par une simple conjonction de registres.
On peut imaginer adapter notre technique pour le cas où le codage des `pause`s serait plus compliqué comme par exemple un codage hiérarchique : dans un programme $P\,;Q$ où les exécutions de P et de Q sont mutuellement exclusives, il est possible d'encoder les états de P et de Q par un même vecteur de registre. La séquence serait alors encodée par un registre booléen supplémentaire valant 0 lorsque P est actif et 1 lorsque Q est actif. Notre technique de partitionnement pourrait être appliquée à de tels circuits mais le codage des blocs actifs, c'est à dire de l'ansemble area perdrait en simplicité. Ceci permettrait d'adapter notre technique au calcul de l'espace d'états de circuits optimisés [71].

Chapitre 6

Mise en Œuvre

Pour implémenter notre technique de partitionnement, nous devons prendre en compte quelques contraintes techniques. Tout d'abord, le calcul des états atteignables est réalisé à partir du format circuit des programmes qui permet aisément de reconstruire selon ses besoins des fragments de la fonction de transition, basés sur les registres correspondants aux points de contrôle actifs. Les fonctions de transition manipulées lors de ce calcul reposent donc sur un ensemble fini de variables booléennes : l'ensemble des registres et l'ensemble des signaux d'entrée du circuit.
D'un autre côté, notre technique de partitionnement utilise un graphe de flot de contrôle, construit à partir d'un arbre syntaxique, pour piloter l'extraction des parties utiles des fonctions de transition à chaque étape de l'algorithme. L'arbre syntaxique est censé représenter fidèlement le programme ESTEREL source et chaque instruction `pause` qui le constitue est censée être identifiée de manière unique par le nom du registre qu'elle génère. De cette association entre les instructions `pause` du programme source et les noms des registres dans le format circuit dépend notre partitionnement. La solution la plus simple consistant à réécrire intégralement un compilateur ESTEREL afin de créer et conserver l'information nécessaire à chaque étape de la compilation n'a pas été retenue.
Dans ce chapitre, nous présentons la manière dont a été implémentée notre technique de partitionnement, de la construction de l'arbre syntaxique et du graphe de flot contrôle jusqu'à son intégration dans un outil de vérification existant. Avant cela, nous faisons un bref rappel de la chaîne de compilation des programmes ESTEREL.

6.1 Chaîne de compilation des programmes ESTEREL

La figure 6.1 illustre les étapes de la compilation d'un programme ESTEREL vers divers format. La version du compilateur utilisée est la v5_9x. Les fichiers sources ESTEREL sont d'abord compilés dans un format intermédiaire [36, 65] à l'aide de l'outil `strlic`. Le but du format intermédiaire est à la fois de préparer la traduction des programmes vers le format circuit en minimisant le nombre de primitives et en introduisant des continuations dans le flot de contrôle. Ce format permet également de faciliter l'édition de liens entre les différents modules. Le format intermédiaire encode donc chaque module ESTEREL sous forme de graphe contenant plusieurs sortes de noeuds et d'arcs. La structure du programme original est en partie conservée. Ce format contient également une table de signaux et une table de registres : `strlic` alloue un registre booléen pour chaque instruction `pause`. L'édition de liens est réalisée par l'outil

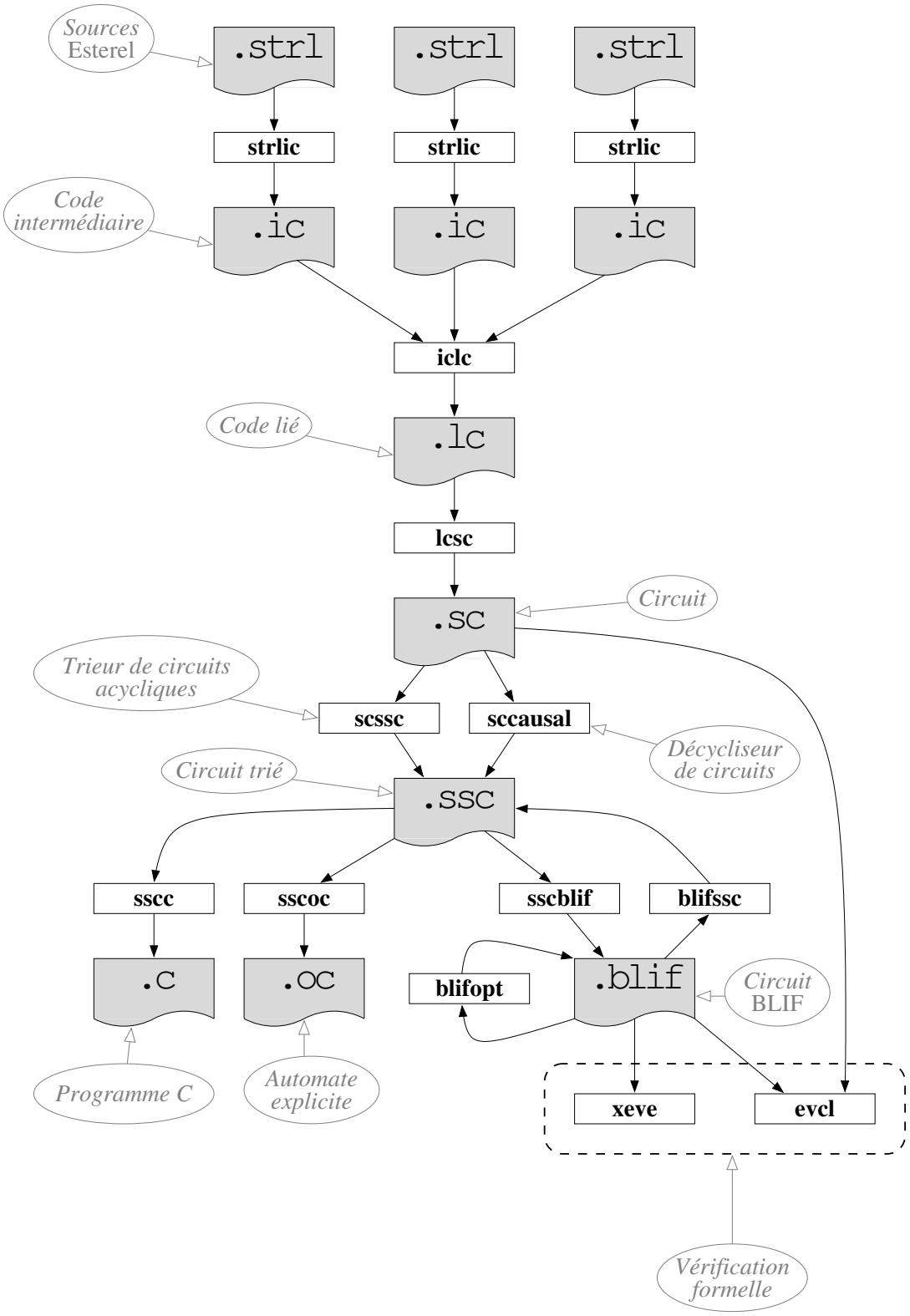


FIG. 6.1 – *La chaîne de compilation d'*ESTEREL.

`iclc`. Le format généré est identique au format intermédiaire à la différence que le programme ESTEREL est constitué d'un module unique avec une unique table de registres. A partir de cette étape, le programme peut être traduit en format circuit [11] par l'outil `lcsc`. La transformation du format intermédiaire vers le format circuit présente l'avantage (pour nous) de conserver les tables de variables et en particulier la table des registres c'est à dire la correspondance entre les pauses et les registres.
Le format circuit peut se traduire en divers formats exécutables (sous forme d'automate ou bien de programme C compilable). Il peut également être traduit en circuit `BLIF` [78], optimisé par `blifopt` [34, 72, 71] et vérifié à l'aide de l'outil `Xeve` [14].
Un autre outil de vérification appelé `evcl` a été développé par Yannis Bres [17]. Cet outil reconnaît aussi bien le format `BLIF` que le format circuit `sc` propre à ESTEREL. Ces deux outils utilisent la librairie `TiGeR` pour la manipulation des BDDs.

6.2 Représentation des programmes ESTEREL

La construction du graphe de flot de contrôle a été réalisée selon deux approches différentes. La première consiste à construire l'arbre syntaxique des programmes lors de la transformation du code source vers le format intermédiaire `ic`. La seconde approche consiste à construire directement le graphe de contrôle à partir du format intermédiaire après l'édition de liens. Chacune de ces approches présente ses avantages et ses inconvénients, comme nous allons le voir dans la suite de cette section.

6.2.1 Construction de l'arbre syntaxique dans `strlic`

Construire l'arbre syntaxique lors de la transformation du programme ESTEREL vers le format intermédiaire nous permet d'obtenir un arbre strictement fidèle au code source. Cette traduction génère une variable de registre pour chaque instruction `pause` et nous permet de construire un arbre dans lequel chaque instruction `pause` est associée au nom de son registre de contrôle.
L'inconvénient de cette approche réside dans le fait que les programmes ESTEREL sont très souvent constitués de plusieurs modules. Cette approche ne produit donc pas un arbre syntaxique mais une forêt d'arbres syntaxiques que nous devons lier entre eux. La phase d'édition de lien consiste à combiner l'ensemble des modules d'un programme ESTEREL au sein d'un module unique. Cette phase s'accompagne également du renommage des variables en général et des registres en particulier. Pour être viable, l'édition de lien dans les arbres syntaxiques doit être réalisée de la même manière que dans les modules ESTEREL, avec les mêmes renommages afin de conserver l'information nécessaire à l'identification des registres de contrôle.
Si le programme ESTEREL source ne contient qu'un seul module, alors la phase d'édition de liens réalisée par l'outil `iclc` ne modifie pas la table des registres. Nous avons choisi de n'accepter que les programmes ESTEREL ne contenant qu'un unique module.

Pour construire l'arbre syntaxique de programmes réels, nous avons donc besoin d'un outil capable d'effectuer l'édition de lien des programmes ESTEREL au niveau du code source. Un tel outil a été développé par Olivier Tardieu au cours de ses travaux sur le compilateur ESTEREL [76]. Cet outil prototype n'accepte pas aujourd'hui l'intégralité de la syntaxe du langage. Par conséquent, notre approche visant à construire notre graphe de flot de contrôle à partir du langage source est encore aujourd'hui incomplète par manque pratique de réalisation logicielle.

Instrumentation de `strlic`. Pour construire notre arbre syntaxique, nous avons modifié le programme `strlic` afin qu'il construise l'arbre syntaxique en même temps que la traduction vers le format intermédiaire. Le résultat est un arbre syntaxique représentant le programme sous forme parenthésée conforme à celui présenté à la section 4.1.

6.2.2 Construction du graphe à partir du format intermédiaire

Après l'édition de lien, un programme ESTEREL est codé par un unique module au format `lc`. A partir de cette étape, les noms des signaux et des registres demeurent inchangés jusqu'à leur traduction dans le format circuit. La construction du graphe de flot de contrôle à partir du format intermédiaire lié nous permet d'accepter n'importe quel programme ESTEREL compatible avec la version v5_9x du compilateur, à l'exception des programmes cycliques.
En contrepartie de cet avantage, le format intermédiaire ne décrit pas complètement la structure du programme ESTEREL original. Le format intermédiaire décrit un graphe de flot de contrôle dans lequel le contrôle peut se propager de trois manières différentes :

- Dans les instructions comme la séquence, le test de présence ou l'opérateur parallèle, le contrôle se propage par *continuation*. Le contrôle se propage séquentiellement d'une instruction à une autre. Par exemple, dans une instruction `present`, le test de présence d'un signal précède l'activation de l'une de ses deux branches.
- La deuxième catégorie est celle des *exceptions* dont nous avons choisi de ne pas parler dans la construction du graphe. De notre point de vue, nous pouvons considérer que les exceptions se comportent comme les instructions précédentes.
- Dans les instructions de suspension ou de préemption comme `suspend` ou `abort` (voir section 2.1), le contrôle se propage par *sélection*. Le format intermédiaire décrit également un arbre appelé *arbre de sélection* dont les noeuds sont des instructions du programme. Dans l'arbre de sélection, le contrôle se propage instantanément des feuilles vers la racine de l'arbre. Par exemple, une instruction `abort` est codée comme un noeud de l'arbre de sélection dont les feuilles sont des instructions appartenant au corps du `abort`. L'instruction `abort` est active dès qu'une de ces instructions est active.

A partir d'une telle représentation des programmes, il est parfois difficile de construire notre graphe de flot de contrôle tel que nous l'avons défini. Par exemple, la construction de notre graphe autour des instructions `present` génère des frontières à la fin de chaque branche. Dans le format intermédiaire, la fin de chaque branche n'est pas mentionnée explicitement. Pour contourner cette difficulté, nous avons tout simplement considéré que chaque noeud ayant plusieurs prédécesseurs était un noeud marquant la fin d'un test. Les relations d'ordre sur les frontières (voir section 4.3) sont également plus difficiles à obtenir. L'ordre induit par la séquence est donc préservé mais nous n'avons pas été en mesure d'implémenter complètement l'ordre permettant d'ordonner les frontières à l'intérieur des instructions `present`.

Nous avons implémenté la construction du graphe à partir du format intermédiaire en respectant autant que possible les définitions du chapitre 4. Les détails de cette construction ne sont pas d'un grand intérêt et ne sont pas donnés dans ce document. Le graphe ainsi construit respecte ces définitions pour toutes les instruction du langage excepté l'instruction `present` : l'ordre que nous imposons ne permet pas d'ouvrir toutes les frontières dans une branche avant les frontières dans la deuxième. Cette simplification est néanmoins assez satisfaisante pour nous procurer de bons résultats expérimentaux.

6.3 Calcul partitionné de l'espace des états atteignables

Nous avons utilisé la librairie `TiGeR` [31] pour la manipulation des BDDs. Notre calcul partitionné des états atteignables a été intégré à l'outil `evcl` développé par Yannis Bres au cours de sa thèse [17]. Cet outil repose sur une extension de la librairie `TiGeR` appelée `TiGeREnh` également développée par Yannis Bres.

6.3.1 `TiGeR` et `TiGeREnh`

La librairie `TiGeR` implémente efficacement les BDDs et les CBFs décrits à la section 2.5. Elle implémente toutes les fonctions de calcul symbolique décrites à la section 2.4. Elle possède également la particularité de simplifier la représentation de la fonction de transition en fonction du domaine auquel elle est appliquée. En réalité, les fonctions de transition de chaque registre sont reconstruites à chaque étape de l'algorithme en fonction de ce domaine. Ceci permet de ne jamais représenter la fonction de transition complète.
En plus des BDDs, la librairie `TiGeR` permet de construire des circuits séquentiels. Elle propose également une routine monolithique permettant de calculer l'espace des états atteignables d'un circuit par l'algorithme Breadth First Search présenté à la section 2.3.
La librairie `TiGeREnh` a été développée dans le but d'expérimenter les méthodes d'abstraction décrites dans [17]. Yannis Bres s'est également appliqué à rendre la routine de calcul des états atteignables plus souple et beaucoup plus interactive que celle de `TiGeR`. Ceci nous a permis d'intégrer très facilement notre technique de partitionnement au reste de la librairie. Précisons également que l'approche par abstraction de Yannis Bres est orthogonale à la notre. Par conséquent, il est possible d'utiliser notre technique de partitionnement dans le calcul des états atteignables avec abstraction.
La librairie `TiGeREnh` permet de redéfinir un module dont le but à chaque étape est d'enregistrer l'ensemble des nouveaux états atteints à l'étape courante et de retourner un ensemble d'états utilisé comme domaine de définition pour le calcul d'image à l'étape suivante. Le module par défaut se contente d'enregistrer les nouveaux états atteints et de renvoyer cet ensemble lors de l'étape suivante. Nous avons donc redéfini ce module afin d'y intégrer notre méthode de partitionnement. Ce module fonctionne avec un graphe de flot de contrôle et permet à chaque étape de choisir un nouveau domaine pour la fonction de transition.

6.3.2 `evcl`

`evcl` est l'outil de vérification fondé sur la librairie `TiGeREnh`. Il constitue un outil très souple permettant de calculer l'ensemble des états atteignables d'un circuit séquentiel. Il procure également une information très riche sur la quantité mémoire utilisée, la taille des BDDs et les temps de calcul à chaque étape de l'algorithme. Nous avons intégré notre méthode de partitionnement à cet outil, ce qui nous a permis d'obtenir facilement les résultats expérimentaux donnés au chapitre 7.

Chapitre 7

Expérimentations

Nous avons testé notre méthode sur de nombreux exemples. La liste de ces exemples est donnée dans les tableaux 7.1 et 7.2. Nous avons mentionné pour chaque programme le nombre de registres et le nombre d'états lorsque ce dernier est connu. Les tests présentés dans ce chapitre ont été réalisés sur un Bi-Pentium III cadencé à 550 Mhz avec 1 Giga octet de mémoire.

Programme	registres	# d'états
`transcad`	7	7
`runner`	9	7
`jeu`	10	7
`tcp`	18	28
`tcpServer`	20	21
`abcd`	21	32
`gsm`	25	18
`rnis`	33	1 213
`pds`	47	65
`mmip`	48	355

Programme	registres	# d'états
`symbologie`	50	62
`wristwatch`	53	41
`mca200`	64	1 921
`xmem`	66	395 403
`aa`	75	116
`tcint`	82	286
`seqLec`	82	90 114
`control`	87	137
`main`	104	10 241
`atds`	124	151

FIG. 7.1 – *Tableaux récapitulatifs des petits programmes* ESTEREL. *La deuxième colonne indique le nombre de registres et la dernière colonne indique le nombre d'états du programme.*

Malheureusement, bon nombre de ces exemples sont de petits programmes (figure 7.1). Ces exemples nous ont permis de vérifier expérimentalement que notre implémentation permettait de calculer le même espace d'états que la méthode de base, mais les résultats sur ces exemples ne sont pas significatifs car les phénomènes d'explosion intermédiaire y sont très limités, et gommés par le petit nombre d'itérations pour atteindre le point fixe des états atteignables. Le calcul partitionné des états atteignables des petits programmes ne permet donc pas de réduire la consommation mémoire, déjà très basse dans l'algorithme de base.

Notre méthode a été conçue pour traiter les exemples plus gros comme ceux présentés dans le tableau 7.2. Ces exemples seront détaillés dans les prochaines sections.

Pour ces expériences, nous avons limité la mémoire utilisée par la librairie de BDDs `TiGeR` à 900Mo afin de n'utiliser que la mémoire vive de la machine. Cette condition est nécessaire afin de garantir que les temps d'exécution sont corrects et non ralentis par une utilisation excessive de la mémoire *swap*. Pour chacun des programmes, nous avons appliqué l'algorithme de base

Programme	registres	# d'états
`chorusBin`	92	136 329 824
`mmid`	111	10 308 357
`steam`	128	41 774 141 026
`sequenceur`	154	122 597
`sat`	192	35 740 420 392 968
`cdtmica`	208	23 384 736 769
`site`	308	$> 2\,380\,837\,289$
`trainsTrappes`	538	> 1
`globalopt`	598	$> 705\,085\,932\,547$
`fuel`	686	$> 8\,749$
`cabine`	919	$> 719\,031\,955$

FIG. 7.2 – *Tableau récapitulatif des gros programmes* ESTEREL.

et notre algorithme partitionné. Dans les résultats expérimentaux que nous présentons nous indiquons :

- le nombre d'itérations réalisées avec succès,
- le nombre d'états découverts,
- le nombre d'états complètement analysés c'est à dire le nombre d'états dont l'image a été calculée,
- la mémoire nécessaire aux calculs,
- le temps total utilisé pour les calculs d'image,
- le temps de calcul total.

7.1 Analyse de programmes coriaces

Les résultats présentés dans cette section concernent les programmes pour lesquels aucun des deux algorithmes n'est parvenu à calculer complètement l'espace des états atteignables. Pour le programme `fuel`, chacun des deux algorithmes échoue dès la seconde itération en ne produisant que 8 749 états. Pour le programme `trainsTrappes`, les 900Mo de mémoire sont consommés avant même d'achever la première itération. A part l'état initial, aucun des deux algorithmes n'a été capable de produire le moindre état.

Pour les programmes `globalopt`, `site` et `cabine` notre algorithme partitionné a pu produire un nombre plus important d'états que l'algorithme de base comme le montrent les tableaux 7.3, 7.4 et 7.5. Toutefois, comme les deux algorithmes ne produisent pas les états dans le même ordre, nous ne sommes pas en mesure de garantir que l'ensemble des états découverts par l'algorithme de base est inclus dans l'ensemble des états découverts par l'algorithme partitionné.

7.1.1 `globalopt`

L'analyse du programme `globalopt` produit les résultats donnés dans le tableau 7.3. Nous pouvons remarquer que l'algorithme partitionné permet de découvrir 2 fois plus d'états atteignables et permet de calculer l'image de 10 fois plus d'états que l'algorithme de base.

Algorithme	défaut	partitionné
Nombre d'itérations	3	80
Nombre d'états découverts	342 858 276 099	**705 085 932 547**
Nombre d'états analysés	583 065 603	**5 542 740 483**
Mémoire nécessaire	$> 900Mo$	$> 900Mo$
Durée totale des calculs d'image	$17m06s$	$3h57m04s$
Temps d'exécution	$34m40s$	$26h45m32s$

FIG. 7.3 – *`globalopt` (598 registres)*

7.1.2 `site`

Le tableau 7.4 présente les résultats de l'analyse du programme `site`. Pour cet exemple, l'algorithme partitionné permet de découvrir 10 fois plus d'états atteignables et permet de calculer l'image 400 fois plus d'états que l'algorithme de base.

Algorithme	défaut	partitionné
Nombre d'itérations	3	91
Nombre d'états découverts	232 705 179	**2 380 837 289**
Nombre d'états analysés	1 049 601	**452 110 875**
Mémoire nécessaire	$> 900Mo$	$> 900Mo$
Durée totale des calculs d'image	$20m$	$9h21m12s$
Temps d'exécution	$22m51s$	$9h58m45s$

FIG. 7.4 – *`site` (308 registres)*

7.1.3 `cabine`

Le tableau 7.5 présente les résultats de l'analyse du programme `cabine`. Dans cet exemple, l'algorithme partitionné permet d'aller beaucoup plus loin que l'algorithme de base en explorant 50 000 fois plus d'états. L'image de 900 000 fois plus d'états a également pu être calculée.

Algorithme	défaut	partitionné
Nombre d'itérations	3	147
Nombre d'états découverts	13 321	**719 031 955**
Nombre d'états analysés	534	**484 744 348**
Mémoire nécessaire	$> 900Mo$	$> 900Mo$
Durée totale des calculs d'image	$12m58s$	$3h38m50s$
Temps d'exécution	$14m22s$	$18h54m29s$

FIG. 7.5 – *`cabine` (919 registres)*

7.2 Réduction de la consommation mémoire

Les résultats présentés dans cette section concernent les gros programmes qui peuvent être complètement explorés par l'algorithme de base et par l'algorithme partitionné. Ces expériences

sont particulièrement intéressantes car les données obtenues permettent de comparer complètement les deux approches. Les résultats expérimentaux montrent que le calcul partitionné des états atteignables des programmes `sequenceur` et `mmid` utilise moins de mémoire que l'algorithme de base.

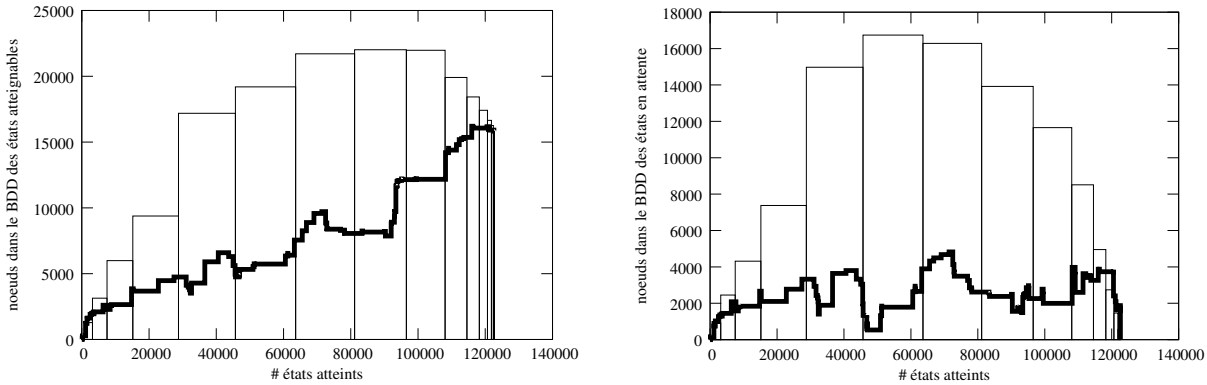
7.2.1 `sequenceur`

Le tableau 7.6 présente les résultats de l'analyse du programme `sequenceur`. Nous pouvons constater que l'exploration exhaustive partitionnée de ce programme utilise 60% de mémoire de moins que l'algorithme de base. En contrepartie, la durée du calcul a été multipliée par près de 2,5. Nous pouvons remarquer que le temps passé à calculer l'image de la fonction de transition est plus court dans l'algorithme partitionné que dans l'algorithme de base.

Algorithme	défaut	partitionné
Nombre d'itérations	18	145
Nombre d'états découverts	122 597	122 597
Nombre d'états analysés	tous	tous
Mémoire nécessaire	40 359Ko	**17 022*Ko***
Durée totale des calculs d'image	$1m44s$	$51,12s$
Temps d'exécution	$3m47,22s$	$8m56,59s$

FIG. 7.6 – *`sequenceur` (154 registres)*

Les graphes de la figure 7.7 représentent l'évolution de la taille des BDDs au cours des calculs. Nos expériences ne permettent pas de représenter l'évolution de la taille des BDDs utilisés pour les calculs d'image car cette information est volatile et donc difficile à obtenir. Toutefois, cette information peut être estimée indirectement d'après la courbe du pic de la consommation mémoire donnée par le graphe 7.8.



(a) (b)

FIG. 7.7 – *Graphes représentant la taille des BDDs en fonction du nombre des états atteints dans le programme `sequenceur`. Les rectangles correspondent à l'algorithme de base et les traits pleins épais correspondent à l'algorithme partitionné. Le graphe (a) représente l'évolution de la taille du BDD des états atteignables. Le graphe (b) représente l'évolution de la taille du BDD des états en attente (les nouveaux états pour l'algorithme de base).*

Nous pouvons remarquer que l'algorithme partitionné permet de réduire la taille des BDDs

dans les étapes intermédiaires. La forme de la courbe 7.7(a) est conforme à la courbe 3.2 de la page 32.

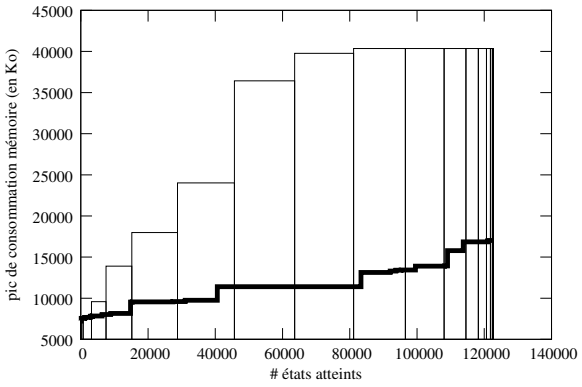


FIG. 7.8 – *Evolution du pic de la consommation mémoire en fonction des états atteints lors de l'analyse du programme* `sequenceur`*. Les rectangles correspondent à l'algorithme de base et les traits pleins épais correspondent à l'algorithme partitionné.*

7.2.2 `mmid`

Le tableau 7.9 présente les résultats de l'analyse du programme `mmid`. L'analyse de ce programme par l'algorithme partitionné nécessite environ 5 fois moins de mémoire que l'algorithme de base. Dans cet exemple, les temps de calcul ont également été raccourcis puisque l'algorithme partitionné est environ 2 fois plus rapide.

Algorithme	défaut	partitionné
Nombre d'itérations	13	113
Nombre d'états découverts	10 308 357	10 308 357
Nombre d'états analysés	tous	tous
Mémoire nécessaire	205 214Ko	**42 368*Ko***
Durée totale des calculs d'image	$42m52s$	$8m25s$
Temps d'exécution	$45m59s$	**$19m38$**

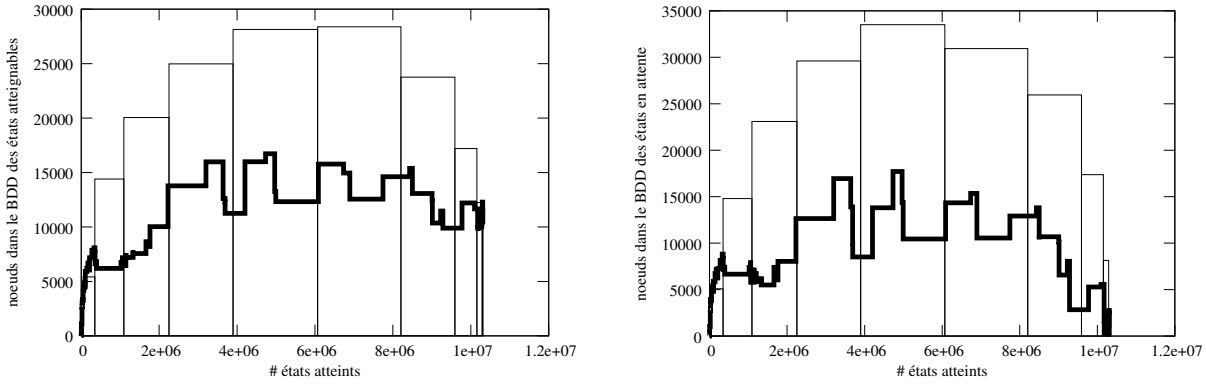
FIG. 7.9 – *`mmid` (111 registres)*

Les graphes de la figure 7.10 représentent l'évolution de la taille des BDDs au cours des calculs. La figure 7.11 représente l'évolution du pic de consommation mémoire au cours des calculs.

Comme pour le programme `sequenceur`, l'algorithme partitionné permet de réduire la taille des BDDs dans les étapes intermédiaires.

7.3 Exploration exhaustive

Les exemples présentés dans cette section ne peuvent pas être entièrement explorés par l'algorithme de base avec moins 900Mo de mémoire. Notre algorithme partitionné a permis d'explorer entièrement ces programmes.



(a) (b)

FIG. 7.10 – *Graphes représentant la taille des BDDs en fonction du nombre des états atteints dans le programme* `mmid`*. Les rectangles correspondent à l'algorithme de base et les traits pleins épais correspondent à l'algorithme partitionné. Le graphe (a) représente l'évolution de la taille du BDD des états atteignables. Le graphe (b) représente l'évolution de la taille du BDD des états en attente (les nouveaux états pour l'algorithme de base).*

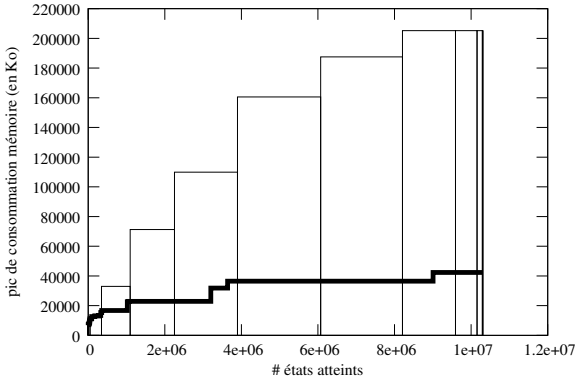


FIG. 7.11 – *Evolution du pic de la consommation mémoire en fonction des états atteints lors de l'analyse du programme* `mmid`*. Les rectangles correspondent à l'algorithme de base et les traits pleins épais correspondent à l'algorithme partitionné.*

7.3.1 `chorusBin`

Le tableau 7.12 présente les résultats de l'analyse du programme `chorusBin`. L'algorithme de base n'est capable de produire que 12% de l'espace des états atteignables total. Dans cet exemple, l'analyse complète du programme est très coûteuse en temps puisque les calculs ont duré 238 heures. Nous ne pouvons pas savoir quelle aurait été la durée des calculs de l'algorithme de base si la mémoire avait été suffisante. Toutefois, nous pouvons remarquer que l'algorithme partitionné passe 99% de son temps dans les calculs d'image. Nous avons donc de bonnes raisons de croire que la durée des calculs provient de la nature du programme `chorusBin` et non pas des calculs supplémentaires nécessaires au partitionnement.

Algorithme	défaut	partitionné
Nombre d'itérations	6	79
Nombre d'états découverts	16 928 480	**136 329 824**
Nombre d'états analysés	441 417	**tous**
Mémoire nécessaire	$> 900Mo$	$851\,369Ko$
Durée totale des calculs d'image	$5h27m44s$	$237h01m40s$
Temps d'exécution	$5h39m35s$	$238h10m45s$

FIG. 7.12 – *`chorusBin` (92 registres)*

7.3.2 `cdtmica`

Le tableau 7.13 présente les résultats de l'analyse du programme `cdtmica`. L'algorithme de base ne permet de produire que 54% de l'espace des états atteignables total. Dans cet exemple, le temps de calcul de l'algorithme partitionné semble raisonnable : l'algorithme partitionné met 2 fois plus de temps à converger que l'algorithme de base à échouer. D'autre part, la courbe 7.14 représentant le nombre d'états atteints au cours du temps semble indiquer que l'algorithme partitionné est un peu plus rapide que l'algorithme de base.

Algorithme	défaut	partitionné
Nombre d'itérations	10	185
Nombre d'états découverts	12 538 388 785	**23 384 736 769**
Nombre d'états analysés	10 651 674 353	**tous**
Mémoire nécessaire	$> 900Mo$	$748\,971Ko$
Durée totale des calculs d'image	$15h17m50s$	$35h38m10s$
Temps d'exécution	$15h24m46s$	$36h31m23s$

FIG. 7.13 – *`cdtmica` (208 registres)*

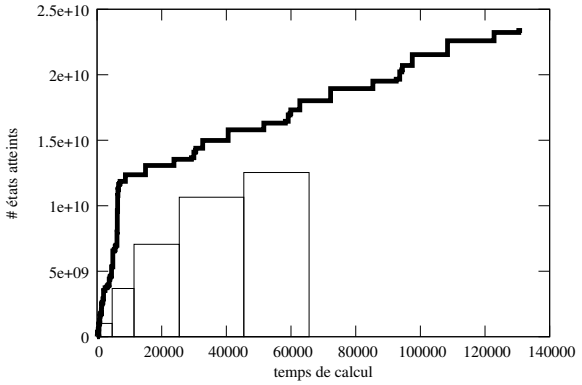


FIG. 7.14 – *Nombre d'états découverts au cours du temps lors de l'analyse du programme `cdtmica`. Le temps en abscisse est exprimé en secondes. Les rectangles correspondent à l'algorithme de base et les traits pleins épais correspondent à l'algorithme partitionné.*

7.3.3 `steam`

Le tableau 7.15 présente les résultats de l'analyse du programme `steam`. L'algorithme de base ne permet de produire que 9% de l'espace des états atteignables total. Ici encore, le temps de calcul de l'algorithme partitionné semble raisonnable (25 heures) par rapport au temps de calcul inachevé de l'algorithme de base.

Algorithme	défaut	partitionné
Nombre d'itérations	3	101
Nombre d'états découverts	3 865 747 524	**41 774 141 026**
Nombre d'états analysés	396 566 399	**tous**
Mémoire nécessaire	$> 900Mo$	$762\,153Ko$
Durée totale des calculs d'image	$47m29s$	$24h09m21s$
Temps d'exécution	$48m36s$	$25h30m21s$

FIG. 7.15 – *`steam` (128 registres)*

7.3.4 `sat`

Pour finir, le tableau 7.16 présente les résultats de l'analyse du programme `sat`. L'algorithme de base ne permet de produire que 0,12% de l'espace des états atteignables total. Dans cet exemple, nous pouvons constater que la consommation mémoire est exceptionnellement basse avec au plus 76Mo. Les temps de calculs sont également remarquables puisque l'algorithme partitionné s'achève avec succès au bout de 3 heures alors que l'algorithme de base échoue au bout de plus de 6 heures.

Algorithme	défaut	partitionné
Nombre d'itérations	17	339
Nombre d'états découverts	43 487 202 056	**35 740 420 392 968**
Nombre d'états analysés	17 566 150 006	**tous**
Mémoire nécessaire	$> 900Mo$	$\mathbf{77\,797Ko}$
Durée totale des calculs d'image	$6h28m29s$	$2h14m40s$
Temps d'exécution	$6h42m50s$	$\mathbf{3h00m56s}$

FIG. 7.16 – *`sat` (192 registres)*

7.4 Conclusion des résultats

Les résultats expérimentaux tendent à montrer que notre algorithme partitionné permet de réduire la taille des BDDs utilisés lors des calculs. Dans chacun des cas présentés ici, notre méthode a permis d'obtenir soit des résultats plus complets en terme de nombre d'états atteints soit les mêmes résultats avec une consommation mémoire diminuée. L'objectif de notre approche a donc été atteint.

En ce qui concerne les temps de calculs, les résultats sont moins évidents. Sur les 11 gros exemples présentés dans ce chapitre, seuls 3 exemples permettent de comparer les deux approches : `sequenceur`, `mmid` et `sat`. Dans le premier exemple (le plus petit des trois), l'algo-

rithme partitionné est nettement plus lent. Dans les deux derniers, l'algorithme partitionné est nettement plus rapide. Dans tous les cas, la durée des calculs d'image a été réduite. Le phénomène peut s'expliquer par le fait que le calcul de l'image prend de plus en plus d'importance sur les gros exemples, autrement dit, le calcul de l'image dans l'algorithme de base est comparativement très rapide dans `sequenceur` (46% du temps total) et très lent dans `mmid` (93% du temps) et `sat` (96% du temps). Ces maigres indices tendent à laisser penser que notre méthode de partitionnement permettrait également de réduire les temps de calcul dans l'exploration des programmes les plus gros où globalement, le temps perdu à partitionner est rattrapé par des calculs d'image plus rapides. D'autres expériences sont nécessaires afin de clarifier ce point.

Chapitre 8

Conclusion et Perspectives

Nous avons présenté une méthode de partitionnement du calcul des états atteignables guidé par la syntaxe des programmes. Ce partitionnement purement automatique est basé sur l'information donnée par les signaux. Nous avons démontré formellement la correction de notre algorithme. Si la complexité théorique de notre algorithme est moins bonne que l'algorithme de base, les résultats expérimentaux sont très encourageants et montrent l'utilité de notre approche. Nous pensons que cette méthode mérite d'être expérimentée sur un plus grand nombre d'exemples tirés d'applications réelles afin d'être complètement validé. Nous souhaiterions également confronter notre méthode avec d'autres méthodes concurrentes comme par exemple [66].
Notre méthode est compatible avec les travaux réalisés par Yannis Bres sur la vérification de programmes par des techniques d'abstraction. Ces deux travaux ont d'ailleurs donné naissance à des prototypes intégrés au sein d'un même logiciel. Il serait intéressant d'expérimenter les apports de notre partitionnement sur ces techniques. Ces expériences n'ont pu être menées par manque de temps ; alors que notre méthode de partitionnement est complètement automatique, les techniques d'abstraction nécessitent une bonne connaissance des applications traitées.

Notre approche présente une faiblesse concernant les boucles dans un contexte parallèle. Le problème vient du fait que notre méthode consiste à ouvrir des frontières sans les refermer. Pour nous, "refermer une frontière" signifie interdire l'activation de certains registres. De ce fait, nous ne faisons que grossir de domaine d'application de la fonction de transition. Souvent, tous les états sont atteignables à la première itération et le fait de refermer les frontières devient inutile. Pour les autres cas, nous pouvons penser que la synchronisation entre les boucles en parallèle fait que nous devrions pouvoir refermer les frontières sous certaines conditions. Dans le futur, nous souhaiterions combler cette lacune. Il s'agirait alors de savoir refermer certaines frontières de manière intelligente. A l'heure actuelle, nous ne savons pas précisément quelles sont ces frontières ni comment, ni à quel moment, ces frontières doivent être refermées dans l'algorithme. De plus, le fait de refermer des frontières nécessite de redéfinir dynamiquement l'ordre de (ré)ouverture a priori des frontières donné par la relation $\prec$ (voir les sections 3.5.1 et 4.3).

Dans sa forme actuelle, notre algorithme de partitionnement calcule un seul ensemble contenant tous les états atteignables. Nous souhaiterions améliorer ce calcul afin qu'il produise la trace des états atteignables, c'est à dire une liste d'ensembles où chaque cellule contiendrait les états atteignables à une profondeur donnée. Ceci pourrait aussi nous permettre de partitionner un peu plus les calculs suivant chaque élément de la liste.

A la section 2.5.1, nous avons vu que l'ordre des variables de BDD avait une influence sur la complexité des BDDs. Dans nos expériences, cet ordre est choisi par la librairie `TiGeR` en fonction de la forme globale du circuit analysé. Dans le futur, nous souhaiterions également trouver, si il existe, un ordre sur les variables de BDDs adapté à notre partitionnement. L'intérêt serait alors de tirer profit du fait que chaque étape de l'algorithme permet de n'appliquer la relation de transition que "localement".

Bibliographie

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6), June 1978.

[2] Rajeev Alur, Radu Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *Computer Aided Verification*, 2000.

[3] Henrik Reif Andersen. An Introduction to Binary Decision Diagrams, October 1997. Lecture notes for 49285 Advanced Algorithms E97. Available from : `http ://www.it.dtu.dk/~hra`.

[4] Charles André. SyncCharts : A visual representation of reactive behaviors. RR 95-52, I3S, Sophia-Antipolis, France, 1996.

[5] Charles André, Hedi Boufaïed, and Sylvan Dissoubray. SyncCharts : un modèle graphique synchrone pour systèmes réactifs complexes. In *Real-Time and Embedded Systems, RTS'98*, pages 175–194. Teknea, January 1998.

[6] L. Arditi, A. Bouali, H. Boufaied, G. Clavé, M. Hadj-Chaib, and R. de Simone. Using Esterel and formal methods to increase the confidence in the functional validation of a commercial dsp, 1999.

[7] Adnan Aziz, Serdar Tasiran, and Robert K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *Proceedings of the 31st Design Automation Conference, DAC'94*, pages 283–288. ACM Press, June 1994.

[8] T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. In *Proceedings of the Workshop on Advances in Verification*, July 2000.

[9] Thomas Ball and Sriram K. Rajamani. Bebop : A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[10] G. Berry. Programming a digital watch in Esterel v3. Technical Report RR-1032, Inria, Institut National de Recherche en Informatique et en Automatique, May 1989.

[11] Gérard Berry. The constructive semantics of pure Esterel. Draft book available at `http ://www-sop.inria.fr/esterel.org/`, July 1999.

[12] Gérard Berry. *The Esterel v5 Language Primer*. CMA, Ecole des Mines and INRIA and Esterel Technologies, July 2000. Available at `http ://www-sop.inria.fr/esterel.org/`.

[13] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Symbolic guided search for CTL model checking. In *Design Automation Conference*, pages 29–34, 2000.

[14] A. Bouali. Xeve, an Esterel verification environment. In *Proc. 10th International Computer Aided Verification Conference*, LNCS, pages 500–504, UBC, Vancouver, Canada, June 1998.

[15] F. Boussinot and R. de Simone. The Esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79 :1293–1304, 1991.

[16] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[17] Yannis Bres. *Exploration implicite et explicite de l'espace d'états atteignables de circuits logiques Esterel.* PhD thesis, Ecole des Mines de Paris, December 2002.

[18] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8) :677–691, August 1986.

[19] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3) :293–318, September 1992.

[20] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In ACM-SIGDA ; IEEE, editor, *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 403–407, San Francisco, CA, June 1991. ACM Press.

[21] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[22] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4) :401–424, April 1994.

[23] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, Los Alamitos, CA, June 1990. ACM/IEEE, IEEE Society Press.

[24] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[25] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, April 1986.

[26] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L.J.M. Claesen, editor, *Proceedings of the IFIP International Workshop Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989. North-Holland.

[27] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[28] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In Satoshi Sangiovanni-Vincentelli, Alberto ; Goto, editor, *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 126–129, Santa Clara, CA, November 1990. IEEE Computer Society Press.

[29] O. Coudert and J. C. Madre. Symbolic computation of the valid states of a sequential machine : algorithms and discussion. In *ACM Workshop on Formal Methods in VLSI Design*, 1991. Miami.

[30] Olivier Coudert. *SIAM : Une Boite à Outils Pour la Preuve Formelle de Systèmes Séquentiels*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Octobre 1991.

[31] Olivier Coudert, Jean-Christophe Madre, and Hervé Touati. TiGeR *Version 1.0 User Guide*. Digital Paris Research Lab, December 1993.

[32] David L. Dill. The Murφ Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification, CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer Verlag, July 1996.

[33] Stephen A. Edwards, Tony Ma, and Robert Damiano. Using a hardware model checker to verify software. In *Proceedings of the 4th International Conference on ASIC (ASICON) (2001)*. IEEE Press, 2001.

[34] Xavier Fornari. *Optimisation du contrôle et implantation en circuits de programmes Esterel*. PhD thesis, Ecole des Mines de Paris, CMA, Sophia Antipolis, France, March 1995.

[35] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Proc. 6th International Computer Aided Verification Conference*, volume 818, pages 299–310, 1994.

[36] Georges Gonthier. *Sémantique et modèles d'exécution des langages réactifs synchrones : application à Esterel*. PhD thesis, Université d'Orsay, Paris, France, March 1988.

[37] Shankar G. Govindaraju and David L. Dill. Verification by approximate forward and backward reachability. In *International Conference on Computer Aided Design (ICCAD-98)*, pages 366–370, N. Y., November 8–12 1998. ACM Press.

[38] Shankar G. Govindaraju, David L. Dill, Alan J. Hu, and Mark A. Horowitz. Approximate reachability with BDDs using overlapping projections. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 451–456, Los Alamitos, CA, June 15–19 1998. ACM/IEEE.

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

[40] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[41] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.

[42] David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[43] Hiroyuki Higuchi and Fabio Somenzi. Lazy group sifting for efficient symbolic state traversal of FSMs. In *International Conference on Computer-Aided Design (ICCAD '99)*, pages 45–49, Washington - Brussels - Tokyo, November 1999. IEEE.

[44] Youpyo Hong, Peter A. Beerel, Jerry R. Burch, and Kenneth L. McMillan. Safe BDD minimization using don't cares. In *Proceedings of the 34th Conference on Design Automation (DAC-97)*, pages 208–213, NY, June 9–13 1997. ACM Press.

[45] A. J. Hu, D. L. Dill, A. J. Drexler, and C. H. Yang. Higher-level specification and verification with BDDs. In *Proc. 4th International Computer Aided Verification Conference*, pages 82–95, 1992.

[46] Alan J. Hu and David L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *Computer Aided Verification*, pages 3–14, 1993.

[47] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *Design Automation Conference*, pages 266–271, 1993.

[48] Alan J. Hu, Gary York, and David L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In Michael Lorenzetti, editor, *Proceedings of the 31st Conference on Design Automation*, pages 276–282, New York, NY, USA, June 1994. ACM Press.

[49] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design : An International Journal*, 9(1/2) :41–75, August 1996.

[50] S. Iyer, D. Sahoo, Ch. Stangier, A. Narayan, and J. Jain. Improved symbolic verification using partitioned techniques. pages 410–424. LNCS 2860, 2003.

[51] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham. Functional partitioning for verification and related problems. In *Advanced Research in VLSI and Parallel Systems : Proceedings of the 1992 Brown/MIT Conference*, pages 210–226, 1992.

[52] Gila Kamhi and Limor Fix. Adaptive Variable Reordering for Symbolic Model Checking. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'98*. ACM Press, November 1998.

[53] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. In *Proceedings of the IEEE*, volume 79(9), pages 1321–1336, 1991.

[54] C. Y. Lee. Representation of switching functions by binary decision programs. *Bell Systems Technical Journal*, 38 :985–999, 1959.

[55] J. C. Madre and J. P. Billon. Proving circuit correctness by formally comparing their expected and extracted behavior. In *25th Design Automation Conference*, Anaheim, June 1988.

[56] Sharad Malik, Albert Wang, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the International Conference on Computer Aided Design, ICCAD'88*, November 1988.

[57] Florence Maraninchi. The Argos Language : Graphical Representation of Automata and Description of Reactive Systems. In *Proceedings of the IEEE Workshop on Visual Languages*, October 1991.

[58] K. L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[59] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5) :1045–1079, 1955.

[60] Christoph Meinel and Christian Stangier. Speeding Up Image Computation by Using RTL Information. In *Formal Methods in Computer-Aided Design, FMCAD'00*, volume 1954 of *Lecture Notes in Computer Science*, pages 443–454. Springer, November 2000.

[61] Christoph Meinel and Christian Stangier. A New Partitioning Scheme for Improvement of Image Computation. In *Proceedings of Asia South Pacific Design Automation Conference (ASPDAC'01)*, pages 97–102. ACM Press, January 2001.

[62] In-Ho Moon, Gary D. Hachtel, and Fabio Somenzi. Border-Block Triangular Form and Conjunction Schedule in Image Computation. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD'00*, volume 1954 of *Lecture Notes in Computer Science*, pages 79–90. Springer, November 2000.

[63] In-Ho Moon, James Kakula, Tom Shiple, and Fabio Somenzi. Least fixpoint approximations for reachability analysis. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design (ICCAD-99)*, pages 41–44, San Jose, CA, November 7–11 1999. ACM/IEEE.

[64] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-ROBDDs. In *IEEE/ACM International Conference on Computer Aided Design; Digest of Technical Papers (ICCAD '97)*, pages 388–393, Washington - Brussels - Tokyo, November 1997. IEEE Computer Society Press.

[65] Jean-Pierre Paris. *Exécution de tâches asynchrones depuis Esterel*. PhD thesis, Université de Nice, France, July 1992.

[66] E. Pastor and M.A. Peña. Combining Simulation and Guided Traversal for the Verification of Concurrent Systems. In *Proceedings of DATE'03*. IEEE publisher, 2003.

[67] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Darmstadt, West Germany, 1962.

[68] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[69] K. Ravi and F. Somenzi. High-density reachability analysis. In *International Conference on Computer Aided Design (ICCAD '95)*, pages 154–158, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[70] Richard Rudell. Dynamic Variable Ordering For Binary Decision Diagrams. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'93*. IEEE Computer Society Press, November 1993.

[71] E. M. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 428–435, Washington, November 10–14 1996. IEEE Computer Society Press.

[72] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, and Robert K. Brayton et al. SIS, A System for Sequential Circuit Synthesis. Technical report, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[73] Thomas Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *Proccedings of International Design and Testing Conference IDTC'96*, Paris, France, 1996.

[74] Christian Stangier, Ulrich Holtmann, and Christoph Meinel. Optimizing Partitioning of Transition Relations by Using High-Level Information. In *Proceedings of the International Workshop on Logic Synthesis, IWLS'2000*, May 2000.

[75] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima. The Complexity of the Optimal Variable Ordering Problems of Shared Binary Decision Diagrams. In *Proceedings of the 4th International Symposium on Algorithms and Computation, ISAAC'93*, volume 762 of *Lecture Notes in Computer Science*. Springer, December 1993.

[76] Olivier Tardieu. Goto and Concurrency : Introducing Safe Jumps in Esterel, March 2004. Synchronous Languages, Applications, and Programming.

[77] Herve J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'90*. IEEE Computer Society Press, November 1990.

[78] University of California at Berkeley. *Berkeley Logic Interchange Format*, December 1998. Available at `http ://www.bdd-portal.org/docu/blif/`.

[79] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 599–604, Los Alamitos, CA, June 15–19 1998. ACM/IEEE.

Résumé

Le calcul symbolique des états atteignables d'un programme constitue un élément de base dans la compilation des programmes réactifs synchrones. Nous proposons d'améliorer la complexité parfois prohibitive de ce calcul en exploitant la structure des programmes. L'idée de base parait extrêmement simple : pour le cas $P\,;Q$ où deux blocs de programme sont combinés en séquence, nous cherchons à construire entièrement les états atteignables de P et de ne s'occuper de Q que lorsque P est complètement exploré. L'intérêt est de n'utiliser à chaque étape que la partie pertinente de la fonction de transition. Si les comportements de P étaient de durée variable, le calcul symbolique Breadth First Search aurait combiné l'exploration de P et de Q dans un même mouvement. De ceci aurait résulté une irrégularité dans les représentations intermédiaires des états atteints, ce qui constitue l'une des plus grandes causes de la complexité du model checking utilisant des techniques symboliques.

Les difficultés de notre approche apparaissent en présence de parallélisme et d'échange de signaux locaux où les blocs de programme peuvent se synchroniser de multiples façons en raison du comportement dynamique du programme. Considérer toutes ces possibilités mènerait à une forte complexité. Le but ici est de trouver un compromis satisfaisant entre l'approche globale Breadth First Search et l'approche compositionnelle partitionnée. Concrètement, nous nous appuyons sur des caractéristiques intéressantes de notre librairie de BDD pour développer une approche efficace. Nous employons des heuristiques permettant de partitionner notre programme et d'ordonnancer la construction des états atteignables afin de calculer exactement les mêmes résultats que par la méthode de base mais en appliquant des fonctions de transition plus localisées. Les premiers résultats expérimentaux montrent la pertinence de notre approche.

Abstract

We consider the issue of exploiting the structural form of Esterel programs to partition the algorithmic RSS (reachable state space) fix-point construction used in model-checking techniques. The basic idea sounds utterly simple, as seen on the case of sequential composition : in $P\,;Q$, first compute entirely the states reached in P, and then only carry on to Q, each time using only the relevant transition relation part. Here a brute-force symbolic breadth-first search would have mixed the exploration of P and Q instead, in case P had different behaviors of various lengths, and that would result in irregular BBD representation of temporary state spaces, a major cause of complexity in symbolic model-checking.

Difficulties appear in our decomposition approach when scheduling the different transition parts in presence of parallelism and local signal exchanges. Program blocks (or "Macro-states") put in parallel can be synchronized in various ways, due to dynamic behaviors, and considering all possibilities may lead to an excessive division complexity. The goal is here to find a satisfactory trade-off between compositional and global approaches. Concretely we use some of the features of the BDD library, and heuristic orderings between internal signals, to have the transition relation progress through the program behaviors to get the same effect as a global RSS computation, but with much more localized transition applications. We provide concrete benchmarks showing the usefulness of the approach.