N° d'ordre :

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS
École Doctorale Sciences Pour l'Ingénieur

# Thèse

présentée pour obtenir le titre de

**Docteur en Sciences**

mention

**Sciences de l'Ingénieur**

par

Daniel Gaffé

# Le modèle Grafcet : réflexion et intégration dans une plate-forme multiformalisme synchrone

Soutenue le 11 Janvier 1996 devant le jury composé de :

| | | |
|---|---|---|
| Messieurs | Frédéric Boussinot | Président |
| | René David | Rapporteur |
| | Nicolas Halbwachs | Rapporteur |
| | Fernand Boéri | Examinateur |
| | Jean-Claude Gentina | Examinateur |
| | Charles André | Directeur de thèse |

# Table des matières

# Chapitre 1

# Introduction

Les applications temps-réel couvrent un très grand nombre de domaines. Nous les rencontrons couramment dans l'industrie manufacturière par l'intermédiaire des Systèmes Automatisés de Production (S.A.P). Dans le secteur militaire ou aéronautique, elles caractérisent souvent les systèmes embarquées. Enfin, elles apparaissent aussi dans le domaine des télécommunications qui doivent faire face à un nombre toujours croissant d'informations à router.

Ces systèmes doivent respecter des contraintes temporelles fortes où le délais d'exécution est prépondérant voire impératif. Ce critère est retenu par le groupe TEMPS-RÉEL [Ell88] et Levi et Agrawala [LA90] pour caractériser les applications temps-réel. De nombreux ouvrages [HS91, HS92, Lap93], mettent également en avant, les qualités de sûreté, de fiabilité et de déterminisme de fonctionnement, comme exigences essentielles.

Dans ce mémoire, nous nous intéressons particulièrement à la conception des systèmes temps-réel réactifs. Ils se distinguent des dispositifs transformationnels, par leur interaction permanente avec l'environnement et leur capacité à réagir en temps borné à des situations critiques. Proposé initialement par D.Harel et A.Pnuelli [HP85], le terme "réactif" est désormais retenu pour qualifier cette classe d'application.

La complexité croissante des applications oblige le concepteur à s'éloigner du matériel et à s'orienter vers une approche de haut-niveau. Cette évolution n'est pas sans problème : la connaissance de chaque détail est perdue au profit d'une vision globale du système. Il devient donc nécessaire, de prouver que le système est conforme au cahier des charges à chaque phase de sa conception. Pour cela, il faut disposer de modèles adaptés, de générateurs efficaces et fiables ainsi que d'outils conviviaux de simulation ou de preuves.

Des méthodologies d'aide à la conception sont donc apparues, elle mettent en œuvre outils et modèles. Par exemple, dans le domaine des S.A.P, l'outil

SPEX [Pan91] propose une assistance à la *spécification* et permet de manipuler des *objets génériques et réutilisables.* Son grand intérêt réside dans son aspect *multiformalisme* qui permet d'interconnecter des boîtes fonctionnelles exprimées dans différents langages “métier”. De même, la méthode CASPAIM (Conception Assistée de Systèmes de Production Automatisée en Industrie Manufacturière) [Cas87, Elk93] permet *d'automatiser et de valider* la démarche de conception d'une unité de production flexible. L'approche retenue est hiérarchique. Elle permet de prendre en compte les aspects fonctionnels et matériels.

Une approche différente est apparue avec les langages synchrones tels que ESTEREL [BG92], LUSTRE [CHPR91] ou SIGNAL [LGLL91]. Leur capacité d'abstraction provient des hypothèses de synchronicité forte retenues. Elle permet de modéliser le comportement du système de manière déterministe, puis de d'implanter la partie contrôle sous forme d'automates à états finis ou d'équations dynamiques. Ces langages bénéficient de sémantiques mathématiques sur lesquelles s'appuient leurs compilateurs. Le code généré est donc fiable et peut être prouvé par des outils communs AUTO [Ver87] ou MEC [ABC94]. Grâce à l'approche synchrone, de nombreuses applications ont vu leur conception grandement simplifiée [Ben89, BB91].

Pour mieux partager les différents résultats sur l'approche synchrone, un format commun de représentation des automates a été défini (OC). Il est devenu un des points d'entrée d'une plate-forme synchrone commune de simulation, de preuve et de génération de code cible. De nouveaux outils sont apparus pour intégrer l'approche par *objets synchrones* de F.Boulanger [Bou93].

Bien avant l'émergence des langages synchrones, l'approche synchrone était présente également en automatique par le modèle de comportement GRAFCET. Ce modèle est largement utilisé pour spécifier la partie commande des S.A.P. Son aspect graphique facilite l'expression du parallélisme et de la séquentialité des actions.

Connaissant les préoccupations des informaticiens et des automaticiens, nous préconisons une *approche multiformalisme synchrone* pour modéliser puis implanter les systèmes de commande. Celle-ci a aussi été adoptée par d'autres auteurs. Nous avons déjà présenté SPEX qui permet d'utiliser le formalisme le mieux adapté à l'expression des comportements souhaités. Cette idée est également à la base des travaux sur les objets synchrones de F.Boulanger. De notre côté, nous avons montré dans [AG94] que la coopération GRAFCET-ESTEREL permettait de modéliser facilement la commutation des modes de marches et d'arrêts.

Dès lors, il nous est apparu intéressant d'intégrer le GRAFCET au sein d'une plate-forme synchrone commune basée sur OC et de concilier les points de vues “automatisme industriel” et “programmation réactive”. De manière analogue, P.Leparc [Lep94], propose aussi de rapprocher langages synchrones et GRAFCET via une traduction systématique de GRAFCET vers SIGNAL.

Toutefois, l'intégration du modèle GRAFCET induit un certain nombre de problèmes :

- *Aspects sémantiques liés au multiformalisme.* Il faut assurer une *cohérence au niveau sémantique.* Les langages synchrones ont des sémantiques mathématiques bien établies. Pour sa part, le GRAFCET a donné lieu à des interprétations divergentes : il faut clairement délimiter le modèle et préciser les choix retenus. D'ailleurs, l'activité de recherche est toujours forte dans ce domaine [DA89, Rou94, FC92]. Elle est menée en particulier par le groupe "GRAFCET" de l'AFCET pour affermir les bases du modèle.

- *Exigence de qualité.* Un des objectifs de la programmation synchrone est la production de code "fiable". Les compilateurs doivent s'appuyer sur des spécifications formelles. L'intégration du GRAFCET nécessite donc une formalisation complète du modèle.

- *Validation.* Le code produit doit être validé. La validation peut être partielle et menée par des simulations, ou être formelle et utiliser des prouveurs. Notons que les validations formelles se font actuellement sur des représentations intermédiaires plus générales (automates, systèmes d'équations... ) [Lep94, Rou94, RR94] ou sur des réseaux de Petri [Moa85, AD93].

- *Spécificités des modèles.* Les outils généraux de validation devraient être capables de prendre en compte les spécificités de chaque formalisme et domaines d'applications.

Ces divers points sont repris dans ce mémoire. Nous plaçons résolument notre étude dans le contexte synchrone pour apporter de réponses nouvelles au problème des évolutions GRAFCET.

### Plan du mémoire

Le chapitre 2 est une introduction à "l'univers synchrone". La spécificité et les avantages de cette approche y sont présentés. L'argumentation s'appuie sur la présentation de plusieurs langages synchrones que sont ESTEREL, LUSTRE, et les STATECHARTS.

Le chapitre 3 est entièrement consacré au modèle GRAFCET. Nous présentons aussi bien les aspects syntaxiques que sémantiques. Nous reviendrons ensuite sur les ambiguïtés du modèle. Pour respecter les axiomes de base des langages synchrones, nous sommes amené à définir une nouvelle interprétation des règles d'évolution. De ce fait, certaines classes de grafcets n'ont plus forcément un mode d'évolution compatible avec la norme. Ces classes sont identifiées et peu utilisées dans la pratique. Pour éviter toute équivoque, nous nous sommes rallié à la suggestion faite par R.David de nommer ce modèle différemment. Le terme S-GRAFCET a été retenu.

Nous proposons dans le chapitre 4, une caractérisation par point fixe de la sémantique comportementale du S-GRAFCET. Celle-ci a été conçue de telle sorte que les conditions soient réunies pour assurer l'existence et l'unicité de la solution. Mais les conditions trouvées ne sont en fait suffisantes que pour une sous-classe de S-GRAFCET. De plus cette spécification ne sait pas prendre en compte la dynamique complète du modèle. Par exemple, elle ne permet pas de caractériser la notion *d'instabilité* GRAFCET.

C'est pourquoi, deux autres sémantiques de comportement du S-GRAFCET sont étudiées dans le chapitre 5. La première est basée sur une réécriture en ESTEREL des comportements et permet d'associer les deux langages au sein d'une même application. Les potentialités d'instabilité et d'indéterminisme sont elles-mêmes traduites en ESTEREL. Ceci permet de traiter ces problèmes comme des *causalités ou des boucles instabilités* ESTEREL. Ainsi, tout compilateur respectant scrupuleusement la sémantique de ce langage synchrone, sera capable de rejeter les grafcets incorrects et de compiler les autres.

La seconde approche présentée, est issue du modèle de point fixe et utilise une notion de micro-pas. Elle donne une caractérisation des grafcets sémantiquement corrects et fournit une algorithme pour trouver *toutes les situations stables accessibles*. Elle permet ainsi de compiler tout grafcet en automate d'états finis déterministe ou en système d'équations d'états.

Dans le chapitre 6, nous nous intéressons à la validation formelle des systèmes modélisés. Les classes de preuves recherchées en GRAFCET ou en S-GRAFCET, sont identifiées et présentées. L'intégration du GRAFCET dans la plate-forme synchrone commune, facilite l'utilisation d'outils généraux de validation d'automates ou de systèmes de transitions. Dans ce mémoire, nous présentons donc quelques preuves réalisables. Mais ces outils ne sont pas particulièrement efficaces pour exprimer simplement les propriétés recherchées en GRAFCET. Pour être pertinente et avoir des chances d'être largement adoptée, une méthode de preuves doit manipuler des entités proches du domaine d'application. La maîtrise complète de la chaîne de compilation GRAFCET, nous permet d'apporter une solution originale pour les applications développées en GRAFCET et d'exprimer les propriétés de sûreté directement dans ce formalisme. L'outil sous-jacent se nomme BAC (Boolean Automaton Checker) [Hal94]. Il est dédié à la vérification symbolique de systèmes décrits sous formes d'équations booléennes.

# Chapitre 2

# Une Méthodologie de Conception des Systèmes de Commande: l'Approche Synchrone

## 2.1 Présentation de l'approche synchrone

### 2.1.1 Spécificité des systèmes réactifs temps-réel

Dans les systèmes temps-réel, la durée *d'exécution* ou de *prise en compte* de l'environnement est une contrainte primordiale. Indépendamment de cette distinction, les systèmes industriels peuvent se différencier en deux classes d'application nommée respectivement *transformationnelle* et *réactive*. La première concerne les opérateurs de transformation ou de calcul sur des données et réalise principalement des fonctions de filtrage. La seconde classe d'application a été qualifiée de "réactive" par D.Harel et A.Pnuelli [HP85] car elle regroupe les systèmes devant réagir de *manière réflexe* au stimuli de l'environnement.

Dans ce mémoire, nous nous intéresserons à la conception des systèmes réactifs temps-réel. Ils se rencontrent dans un très grand nombre de domaines répertoriés [Ell88, Ben89]: dans l'industrie manufacturière, ils interviennent dans la conduite de procédés, les systèmes de régulation, la gestion des robots et des automates industriels. Dans l'aéronautique et les domaines militaires, ces applications se retrouvent dans les systèmes embarqués. enfin dans les télécommunications, les caractères réactifs et temps-réels sont tout à fait adaptés pour qualifier les protocoles de communication. Tous ces systèmes ont en commun les caractéristiques suivantes:

- **Contraintes temporelles fortes**: Les systèmes temps-réel possèdent souvent des temps d'exécution ou de réaction *impératifs*. Il est même fréquent que les dépassements temporels soient purement interdits dans certaines applications. Pour les autres, le non-respect d'un délai doit être systématiquement détecté et conduire à un traitement d'exception. Rappelons que ces mêmes délais sont uniquement vu comme facteur d'optimisation dans les systèmes de gestion classique.

- **Prévisibilité et sûreté** : En fonctionnement normal, toute les évolutions (ou leurs conséquences) doivent être prévisibles. Un tel système est souvent qualifié de *déterministe*. Par exemple, il ne doit jamais être responsable (en fonctionnement normal) de dégradation matériel, d'incident de fabrication ou d'accident sur le personnel.

- **Fiabilité** : Le risque de panne doit être le plus faible possible. Ceci se concrétise souvent par le doublement voire le triplement des organes critiques.

- **Robustesse** : Lorsque l'environnement n'est plus celui prévu ou que l'un des organes tombe en panne, le système global doit être capable de prendre un mode de fonctionnement dégradé sécurisé, soit en terminant proprement les tâches en cours, soit en continuant d'assurer ses fonctions essentielles. Cette caractéristique est souvent liée à la prévisibilité.

- **Parallélisme et répartition** : Le processus à commander est souvent constitué de plusieurs tâches concurrentes. Ceci impose la mise en place de mécanismes de synchronisation et d'ordonnancement spécifiques aux contraintes temps-réel. De plus le système peut lui-même être réparti géographiquement, ce qui complique encore sa conception.

- **Hiérarchie** : Les systèmes temps-réel constitués "d'un bloc" sont rares. Par soucis de simplification, il est courant de définir un système de contrôle comme le chef d'orchestre de sous-systèmes dont chacun ne gère qu'une partie de l'application. À leur tour, ces sous-systèmes peuvent se décomposer en des systèmes plus élémentaires. Définir une hiérarchie stricte n'est cependant pas toujours possible (en particulier dans les systèmes flexibles). Le concepteur est alors obligé de tenir compte de la dynamique d'exécution.

Différentes définitions ont été données. Pour notre part, nous retiendrons celle du groupe TEMPS-RÉEL où *une application peut être qualifié de temps-réel, si elle met en œuvre un système dont le fonctionnement est assujetti à l'évolution dynamique d'un environnement et dont il contrôle le comportement* ... Nous retiendrons aussi l'approche de Levi et Agrawala [LA90] qui met en avant les notions de contraintes temporelles pour modéliser et implanter ces systèmes. Enfin nous prendrons en compte l'approche de Halang et Stoyenko [HS91] qui met plutôt en avant la *prévisibilité* et la *sûreté de fonctionnement* comme caractéristiques fondamentales de ces systèmes.

### 2.1.2 Intérêt de l'approche synchrone

Le temps physique est une grandeur continue. La première approche qui vient naturellement à l'esprit pour modéliser puis implanter les systèmes temps-réel est de nature asynchrone. Mais les modèles classiques comme les réseaux de Petri prennent mal en compte les contraintes temporelles. Au niveau implantation, les langages doivent accepter le parallélisme de l'application et mettre en place des mécanismes de synchronisation et de partage de données (sémaphores, sections critiques ...). Si ces aspects sont maintenant assez bien pris en compte, le respect des délais d'exécution reste cependant difficile à maîtriser. De plus tous ces langages doivent s'appuyer sur un système d'exploitation temps-réel (norme Sceptre [Bur84] par exemple) pour que l'application soit réellement opérationnelle. Au niveau preuve, il est difficile d'analyser les évolutions possibles de toutes les tâches concurrentes et de prouver telle ou telle propriété. Ce problème est lié notamment à la simultanéïté des événements d'entrée, qui est responsable de bon nombre d'indéterminismes.

L'approche synchrone permet de lever ces problèmes. Elle s'intéresse au système uniquement lorsque celui-ci est susceptible de changer d'état. Ces évolutions sont *synchronisées* avec les variations des entrées (appelés souvent *événements déclencheurs*) ou la validité des sorties. Dans le premier cas, chaque occurrence d'événement détermine un instant d'activation. La simultanéïté des événements prend un sens et peut conduire à un comportement spécifique du système. Dans le second cas, la prise en compte de nouvelles sorties induit un ordre partiel d'évaluation des équations qui définit un changement de *temps logique*.

Par contre l'hypothèse synchrone forte impose que toute évolution du système commence et terminent dans le même instant. En théorie, cela suppose que la *durée d'exécution de la réaction* est nulle. En pratique, il faut vérifier que toutes réactions du système terminent avant l'occurrence de l'événement suivant. Un moyen fréquemment utilisé, est de maintenir *atomique* toute réaction du système.

Le GRAFCET a été le premier modèle synchrone réellement utilisé. De nos jours, il est largement employé dans le milieu industriel[1] pour modéliser puis implanter les automatismes logiques. Depuis sont apparus d'autres formalismes graphiques comme les STATECHARTS ou les BHRS [Per93] qui ont un pouvoir descriptif supérieur vis à vis de l'encapsulation des sous-systèmes.

Une certaine révolution est apparue avec l'arrivée des langages synchrones comme ESTEREL, LUSTRE ou SIGNAL. Ils ont en effet permis de définir des sémantiques de comportement rigoureuses et assurent au niveau implantation que le code généré sur la machine cible aura rigoureusement le même comportement que le modèle de départ. Tous les langages synchrones sont capables de générer du code efficace sous forme d'automate fini ou de système d'équations. L'apport

1. au moins en Europe.

d'une sémantique formelle permet non seulement de lever toute ambiguïté d'interprétation comportementale, mais de plus elle permet la construction d'outils de preuves formelles de comportement. Tous ces avantages ont conduit à préconiser l'approche synchrone [BB91] (dont ESTEREL [BdS91]) et à développer une plate-forme commune de simulation, de preuves et de génération de code cible. De notre coté, nous nous sommes intéressé plus spécifiquement au langage ESTEREL car il possède un style impératif, des opérateurs explicites de séquencement et de parallélisme et des primitives puissantes de préemption. De manière plus circonstancielle, nous avons pu accéder à toutes les pré-versions du compilateur et discuter de nombreuses fois de sémantique et d'implémentation avec toute l'équipe du CMA [2]. C'est pourquoi, dans ce mémoire, nous ferons référence à ce langage pour tout problème sémantique et nous intégrerons cette plate-forme synchrone commune au chapitre 5.

Pourtant les langages synchrones ne font pas encore partie de la culture industrielle, même si des résultats probants ont été obtenus sur des systèmes "grandeur nature" [BGP93, Ghe92]. Parmi les raisons, deux d'entres-elles sont certainement liées à l'implémentation. D'abord un langage synchrone n'est pas autonome : il nécessite la présence d'une machine d'exécution dédiée à l'application (chapitre 2.3). Ensuite les automates finis issus de la compilation, ont généralement une taille importante. Mais ce phénomène est peut-être aussi lié à la réticence des industriels à intégrer le formalisme synchrone.

## 2.2 Principaux modèles synchrones de représentation des systèmes de commande

Parmi les modèles synchrones que nous allons maintenant présenter, ne figurent ni le GRAFCET, ni SIGNAL. Le premier modèle n'est pas présenté ici car il sera largement étudié lors du chapitre 3. Le cas de SIGNAL [LGLL91] est différent : ce langage important, n'est pour l'instant pas compatible avec la plate-forme synchrone présentée au chapitre 2.1.2. Il possède en fait son propre environnement de simulation et de preuves SIGALI.

### 2.2.1 ESTEREL

Le langage synchrone ESTEREL [BG92, BdS91, BC84] a été créé pour modéliser puis implanter les systèmes réactifs. L'idée essentielle est de réduire la frontière entre ces deux phases en assurant la conservation des propriétés comportementales. Pour cela le modèle ESTEREL a été muni d'une sémantique ma-

2. Centre de Mathématiques Appliquées

thématique de comportement précise qui doit être suivie par tout compilateur. Cette approche permet de conserver les propriétés logiques et temporelles jusqu'à la génération de code objet (automate fini). Ce code est indépendant de l'implémentation est peut faire l'objet de preuves formelles de comportement. Comme les automates peuvent également être implantés de manière automatique dans un langage de programmation impératif, toute preuve logique établie se retrouve réellement dans l'application finale ("WYPIWYE")[3] [Ber89]. Les propriétés sont d'ailleurs très probantes dès que l'application est modélisée en ESTEREL pur [4].

Pour exprimer facilement le *parallélisme* et la *séquentialité* des actions, le langage possède une syntaxe textuelle impérative. Il se démarque ainsi des autres langages synchrones que nous allons présenter, qui mettent en avant leur vision fonctionnelle des systèmes.

Pour qualifier le langage ESTEREL, F.Boussinot et R. de simone l'ont exprimé par une équation [BdS91]:

| | | |
|---|---|---|
| ESTEREL | = | *Réactivité* |
| | + | *Atomicité des réactions* |
| | + | *Diffusion instantanée des signaux* |
| | + | *Déterminisme* |

La *réactivité* caractérise ESTEREL. Elle est liée à *l'hypothèse de synchronisme fort* intrinsèque au modèle. Tout contrôle écrit en ESTEREL réagit à une histoire des entrées et émet à son tour une histoire déterministe des sorties. Nous verrons dans le chapitre 3 entièrement dédié au modèle GRAFCET que cette approche a également été retenue. Pour cela ESTEREL fournit des instructions réactives sensibles aux occurrences d'événements.

L'*atomicité des réactions* signifie que toute évolution du système se réalise avec les entrées figées. Théoriquement ceci n'est acceptable que si tout système modélisé évolue infiniment vite. L'atomicité induit donc *l'instantanéité des réactions.* En pratique, cet axiome reste valable si le concepteur peut assurer a posteriori que le temps de réaction du système est inférieur à celui de l'environnement.

Le signal est une des entités de base d'ESTEREL. Ce vecteur d'informations regroupe les variables et les événements. Ces derniers se distinguent par leur rôle de *déclencheur de réactions.* C'est pourquoi ils sont essentiels pour toute synchronisation. La *diffusion instantanée des signaux* concentre deux notions: diffusion à tous et transmission instantanée. La première indique que toute partie d'un programme entend et peut réagir à une quelconque émission d'un signal. Ce type de liaison est appelé dans la littérature anglo-saxonne "OnetoAll". La seconde notion est fortement liée à l'instantanéité des réactions. Comme un programme évolue si l'une de ses parties évolue et que la réaction globale est atomique, toute

3. what you prove is what you execute.
4. sous-ensemble du langage qui ignore la notion de valeur.

conséquence locale doit pouvoir être prise en compte par le reste du programme *instantanément*.

Le *déterminisme* du modèle est assuré par la sémantique d'ESTEREL. En particulier l'opérateur "||" n'induit aucun indéterminisme de comportement. Le *déterminisme* du langage provient des compilateurs qui respectent scrupuleusement le modèle et sa sémantique. Si le programme est accepté, ils peuvent toujours engendrer un automate d'état fini déterministe de comportement équivalent.

#### 2.2.1.1 Sémantique d'Esterel

La sémantique d'un langage est l'expression du sens de chacune des entités et des opérateurs du langage. Cette sémantique ne donne pas forcément la *manière* d'implanter le langage. Dans [Gon88], l'auteur propose trois sémantiques pour ESTEREL.

- La **sémantique dénotationnelle** regroupe un ensemble d'équations conditionnelles qui caractérise les évolutions du programme. Vus les axiomes de base, la solution des équations doit être unique. Cette dernière est assimilée à un point fixe sur les ensembles de signaux internes émis à chaque instant.
- La **sémantique comportementale** est formellement compatible avec la précédente. Elle décrit le comportement de chaque instruction par un ou plusieurs systèmes de réécritures. Chacune traduit l'instruction et l'environnement en une nouvelle instruction et un nouvel environnement. Cette écriture récursive s'arrête avec l'instruction de changement d'instant "halt" ou la terminaison instantanée.
- La **sémantique d'exécution** donne une manière d'implanter le langage et le compilateur. Dans [Gon88], elle est basée sur la notion de *potentiel* et impose l'étude exhaustive de tous les états accessibles. Là encore, cette sémantique est compatible avec la précédente : tout programme *compilable* est sémantiquement juste. Par contre le contraire n'est pas vrai. Depuis, d'autres sémantiques d'exécution ont été proposées [Mig94, For95]. Elles sont basées sur les notions de précédences de signaux et permettent d'éviter l'étude exhaustive de tous les états. Nous verrons pourtant au § 5.3.5 que le choix d'une sémantique d'exécution ou d'une autre, n'est pas du tout anodin et peut avoir des conséquences fâcheuses sur le pouvoir d'expression du langage ESTEREL.

#### 2.2.1.2 Primitives d'Esterel

Le langage possède plusieurs primitives ou instructions de base qui sont à leur tour utilisées pour en décrire d'autres. La sémantique comportementale de toutes ces primitives est décrite formellement dans le manuel de référence du langage [CIS91]. Pour notre part, nous allons présenter quelques instructions et opérateurs qui nous semblent caractériser le langage. L'ensemble des primitives peut être regroupées en quatre familles :

- les instructions classiques,
- les opérateurs de manipulation de données,
- les opérateurs de synchronisation,
- les instructions réactives et temporelles.

**Les instructions classiques**

Le langage ESTEREL est doté d'un ensemble d'instructions classiques :

- de communication de signaux (`emit`) à l'ensemble du programme,
- de structures de contrôle (`if ...then ...else ...end` ou `present ...then ...else ...end`),
- d'itération (`loop ...end loop`),
- d'abstraction (`module`) permettant de regrouper des blocs d'instructions dédiés à un comportement spécifique.

Le langage possède aussi une instruction d'arrêt (`halt`) qui bloque l'évolution de la branche dans lequel il se trouve. Cette instruction qui est mineure dans les langages asynchrones, est ici très importante. En effet un programme ESTEREL évolue jusqu'à ce que toutes ses branches soient finies ou qu'elles soient bloquées par l'instruction `halt`. À l'instant d'après, certaines branches seront préemptées et l'évolution recommencera. Dans sa thèse, Gonthier considère d'ailleurs qu'un programme ESTEREL évolue d'un ensemble de `halt` vers un autre ensemble de `halt`.

ESTEREL a de plus introduit depuis la version 3.30, l'instruction `exec`[Par92]. Cette primitive assure la gestion logique de tâches externes et permet une ouverture du langage vers l'univers asynchrone. Elle est utilisée lorsque certains traitement ont une durée non négligeable par rapport au temps de réaction du système. De ce fait, un `exec` est *bloquant* et ne peut intervenir au niveau des évolutions, qu'à partir de l'instant suivant.

### Les opérateurs de manipulation de données

Ces opérateurs classiques dans les langages asynchrones (":=", "+","*", ...), permettent de construire des expressions sur les variables.

```
X := ( X + 3) *5
```

Il faut cependant remarquer que tous ces opérateurs s'exécutent en temps nul. Comme l'opérateur de séquencement ";" s'exécute lui-même en temps nul, un nombre illimité de manipulations sont susceptibles de s'effectuer en temps nul. Notons ici, que ESTEREL manipule uniquement un temps *logique discret* basé sur la succession des réactions du système. Ce temps est indépendant du temps *physique*. Le concepteur doit donc vérifier a posteriori que les ralentissements occasionnés par les manipulations de données, restent compatibles avec l'hypothèse *de synchronicité forte* ou utiliser l'instruction `exec`.

### Les opérateurs de synchronisation

Pour synchroniser au plus bas niveau deux instructions ou deux blocs d'instructions, ESTEREL fournit un opérateur de *séquencement* ";" et un opérateur de *parallélisme* "||". L'opérateur de séquencement permet de débuter le bloc aval à l'instant où le bloc amont termine. En ce sens l'opérateur ";" s'exécute en temps nul. L'opérateur de parallélisme est du type "fork-join" : Les différentes branches sont lancées et s'exécutent en même temps. Par contre l'opérateur "||" ne se termine que lorsque toutes les branches sont terminées. Au niveau des variables, la sémantique de l'opérateur "||" a été restreinte pour éviter tout indéterminisme. En particulier, il est interdit de partager des variables entre plusieurs branches.

### Les instructions réactives et temporelles

Cette classe d'instructions donne au langage, une richesse d'expression pour modéliser le comportement des systèmes. Elle permet la préemption instantanée d'une exécution. ESTEREL distingue d'ailleurs la *préemption forte* où l'exécution n'a pas lieu, de la préemption faible où l'exécution et ces conséquences restent effectives. La première est caractérisée par la puissante instruction "`do ...watching ...timeout`".

```
do
    await debut;
    emit Consequences
watching Arret
timeout
    emit Termine
end
```

Cet exemple montre comment l'émission du signal `Conséquences` peut être préemptée si `Arret` survient *avant* ou en *même* temps que `Debut`. Dans ce dernier cas la clause `timeout` sera tout de même exécutée et pourra donner lieu à un traitement d'exception.

De même, la préemption faible est exprimée grâce à l'instruction `trap ... in ... handle ... end trap`

```
trap Arret_demande in
    [
        await Debut;
        emit Consequences
    ]
    ||
    [
        await Arret;
        exit Arret_demande
    ]
end trap
```

Dans cet exemple, si `Arret` arrive avant `Debut`, `Consequences` ne sera pas émis comme dans l'exemple précédent. Par contre si les deux événements arrivent simultanément, `Consequences` sera tout de même émis avec la terminaison du trap.

Avec ces instructions de préemption, il existe maintenant une instruction de suspension (`suspend ... when`). Celle-ci permet de bloquer l'exécution d'un processus avec l'occurrence d'un signal.

Ces trois instructions de préemption permettent de construire les autres primitives temporelles connues. Ces dernières sont souvent utilisées pour gérer l'*aspect multiforme du temps*. Nous pouvons citer en particulier les instructions `await` ou `await immediate` qui sont dérivées du `do ... watching` et qui attendent l'occurrence d'un signal.

Curieusement, le langage ne fournit pas de primitive `pre` qui donnerait l'image d'un signal à l'instant d'avant. Cette primitive serait utile pour garder une trace du passé événementiel. Elle intervient d'ailleurs dans la construction du module étape qui sera présenté au § 5.3.2.1. Cette primitive peut quand même s'écrire directement en ESTEREL :

```
loop
    await immediate S;
    await tick;
    emit pre_S
end loop
```

#### 2.2.1.3 Caractérisation des programmes incorrects

Il existe deux cas principaux de rejet de programme par le compilateur ESTEREL que sont les *boucles instantanées* et les *cycles de causalité*.

La première est souvent liée à l'instruction "`loop ... end` qui se réécrit d'après la sémantique comportementale, comme un nombre infini d'exécution de son corps. Ceci impose que le corps dure au moins un instant. Ainsi le programme suivant sera rejeté :

```
loop
    emit S
end loop
```

Le second type d'erreur regroupe en fait beaucoup de cas. Par exemple, Gonthier [Gon88] a montré que le programme qui suit, ne pouvait pas se réécrire. Si S est absent, la règle de réécriture de `present` conduira à son émission. Inversement, si S est présent il sera impossible a posteriori de le confirmer.

```
signal S in
    present S else emit S end
end
```

Un autre cas d'erreur concerne les réécritures aboutissant à deux conclusions distinctes. Sur le programme suivant, Le programme suivant, doit être rejeté pour indéterminisme car l'émission ou la non-émission de S forment deux solutions viables.

```
signal S in
    present S then emit S end
end
```

Le dernier cas de causalité présenté, regroupe tout programme sémantiquement juste mais *non causal*. Ici apparaît la différence entre *sémantique comportementale* et *sémantique d'exécution*. La sémantique comportementale est surtout une *spécification* des solutions. Elle accepte toute hypothèse pourvu que celle-ci soit a posteriori vérifiée. Ainsi elle acceptera (exemple suivant) le test de présence d'un signal qui sera effectivement émis ensuite. La sémantique d'exécution devant donner une manière de trouver la solution, ne sait pas forcément prendre en compte ce phénomène.

```
signal S in
    present S then emit O end
    ;
    emit S
end
```

#### 2.2.1.4 La chaîne de compilation d'ESTEREL

La chaîne de compilation d'ESTEREL s'est légèrement compliquée depuis la mise en place du compilateur V4 qui génère un système d'équations tout en restant compatible avec son prédécesseur. Pour simplifier cette présentation, nous indiquons sur la figure 2.1 les spécificités de chaque compilateur.

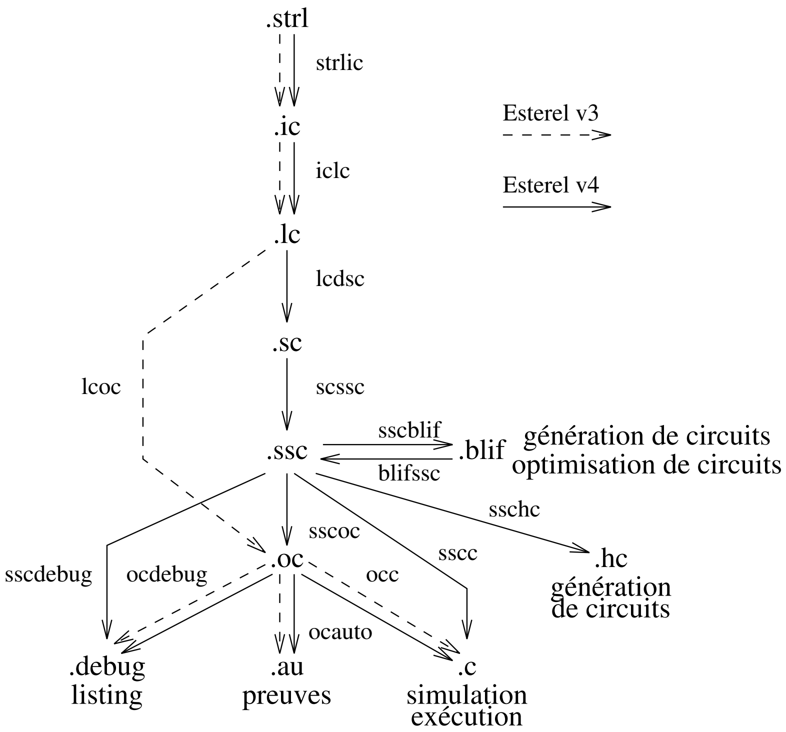



FIG. 2.1 - *Chaîne de compilation d'* ESTEREL

Le compilateur V3 fournit un automate à états finis dont le comportement est équivalent au programme initial. Ceci est réalisé par les phases suivantes.

- Une vérification syntaxique des programmes ESTEREL est effectuée. Elle est suivie de la réécriture du programme en ESTEREL noyau (code IC). Ce format intermédiaire conserve toutes les structures du programme initial.

- Les instructions `run` sont instanciées avec leur module pour donner le code LC.

- La compilation effective aboutit à la mise à plat du parallélisme et à la résolution de toutes les communications internes. Le code obtenu est un automate d'états finis au format OC.

- Cet automate peut être traduit en langages classiques (C, Lisp ...). Il peut aussi intégrer des outils de simulation [CIS88, Ghe92] ou de preuves formelles de comportement [Ver87, Roy90, Fer88, Arn89].

Le compilateur V4 génère un système d'équations au format SSC. Le code généré peut ensuite être transcrit au format OC (automate monoboucle) pour rester compatible avec le compilateur V3. La chaîne de compilation peut se résumer ainsi :

- Les générations de code IC et LC sont effectuées comme précédemment.

- Chaque instruction est réécrite sous forme d'équations. Cette première phase réelle de la compilation aboutit au code SC.

- Un tri est effectué sur les équations suivant les précédences de signaux et aboutit à la génération de code SSC. À ce niveau les programmes incorrects sont rejetés.

- Pour bénéficier de tous les outils spécifiques de la version 3, un automate à un état peut être engendré au format OC. Le code SSC peut aussi être transcrit au format BLIF ou au format HC pour intégrer une chaîne de génération de circuits électroniques. Dans ce cas le code ESTEREL initial doit normalement être pur. Cette limitation est en partie levée [For95] pour certaines classes de variables dont les compteurs. Le format BLIF permet également d'optimiser le code produit [Ell92], avant de le retranscrire en SSC.

### 2.2.2 LUSTRE

Le langage synchrone LUSTRE [CHPR91, HCRP91] a été développé pour modéliser les systèmes temps-réel utilisés en automatique ou en électronique. Les automaticiens ont souvent l'habitude de modéliser cette classe de systèmes par des formalismes mathématiques (équations) ou par des formalismes graphiques (réseaux d'opérateurs, schémas blocs). Dans la culture informatique, ces formalismes sont regroupés sous le vocable de "*flot de données*" et s'opposent à la vision impérative classique. Ils ont cependant bon nombre d'avantages.

Tout d'abord, le parallélisme de l'application est explicite. Les opérateurs sont autonomes et ils se synchronisent entre-eux par les données. Ensuite les représentations graphiques sont très explicites, chaque opérateur pouvant être vu comme une "boîte noire" dont le comportement est connu. Ces boîtes peuvent être ensuite décomposées en sous-réseaux pour faire apparaître la hiérarchie fonctionnelle. Enfin les langages *flot de données* et plus généralement les langages fonctionnels ont adoptés des sémantiques mathématiques qui permettent l'élaboration de preuves ou de réécritures formelles.

C'est pourquoi LUSTRE est un langage *flot de données* qui a adopté un style de programmation *fonctionnel*. Son caractère synchrone provient de la capacité de ces opérateurs à répondre *instantanément* à ces entrées.

#### 2.2.2.1 Les flots

Un flot est formé d'une suite de valeurs d'un type donné et d'une horloge. La suite peut éventuellement être infinie. Les flots ne sont pas synchronisés entre eux si les horloges sont indépendantes. Il est par contre possible d'utiliser des flots booléens pour définir l'horloge de flots plus lents. Il suffira pour cela de considérer la suite des instants où la valeur booléenne est *vraie*. Comme ESTEREL, les horloges ne sont pas liées au temps physique. Les auteurs les voient plutôt comme *définissant le grain minimal de temps discernable par le programme.*

#### 2.2.2.2 Les nœuds

Un programme ou un sous-programme LUSTRE est appelé "Nœuds". Il possède une interface d'entrées/sorties et réalise un traitement particulier. Chaque nœud peut dont être considéré comme une fonction (ou un opérateur) travaillant sur des n-uplets. À chaque nœud est associée une horloge de base qui est celle de ses entrées: si les horloges sont différentes, la plus rapide sera utilisée. Par contre les sorties peuvent être associées à des horloges différentes de l'horloge de base et être utilisées comme entrées d'autres nœuds.

#### 2.2.2.3 Les opérateurs temporels

Pour manipuler les flots de donnés, LUSTRE dispose d'opérateurs classiques sur les types et d'opérateurs temporels :

- L'opérateur *pre* permet de mémoriser une expression à l'instant précédent. Si $E$ est une expression définie par la suite $(e_1, e_2, \ldots, e_n, \ldots)$ alors $pre(E)$ aura la même horloge et sera défini par la suite $(nil, e_1, e_2, \ldots, e_{n-1}, \ldots)$ où $nil$ est une valeur indéfinie.

- L'opérateur $\rightarrow$ ("suivi de") définit la valeur initiale. Si $E$ et $F$ définis respectivement par $(e_1, e_2, \ldots, e_n, \ldots)$ et $(f_1, f_2, \ldots, f_n, \ldots)$, ont la même horloge, $E \rightarrow F$ sera défini par $(e_1, f_2, \ldots, f_n, \ldots)$. $E \rightarrow F$ est donc toujours égal à $F$ sauf au premier instant où il prend la valeur $e1$.

- L'opérateur *when* échantillonne les signaux sur une horloge plus lente. Si $E$ défini par $(e_1, e_2, e_3, \ldots, e_n, \ldots)$ et $F$ une expression booléenne définie par $(true, false, false, \ldots, true_n, \ldots)$ ont la même horloge, $X = E \ when \ F$ sera défini sur une horloge plus lente (synchronisée sur celle de $E$) et aura comme séquence les valeurs de $E$ lorsque $F$ est vraie, soit $(e_1, \ldots, e_n, \ldots)$.

- L'opérateur *current* permet de maintenir la valeur d'une expression par rapport à une horloge plus rapide. Sur l'expression $X$ précédente, $current\ X$ a la même horloge que $F$ et se définit par $(e_1, e_1, e_1, \ldots, e_n, e_n, \ldots)$.

#### 2.2.2.4 Les assertions

Outre les systèmes d'équations, un programme LUSTRE est composé d'assertions. Ceux sont des équations booléennes toujours vraies. Elles permettent d'optimiser le code généré. Elles caractérisent des propriétés de l'environnement.

Considérons par exemple un vérin géré par deux capteurs de fin de course $f1$ et $f2$, en fonctionnement normal nous pouvons affirmer que ces deux capteurs ne rendent jamais la valeur `vraie` en même temps. Ceci s'écrira :

```
assert not (f1 and f2);
```

Les assertions peuvent utiliser tous les opérateurs (dont *pre*). Il est ainsi possible de créer des expressions assez complexes pour spécifier les histoires d'entrées possibles. Ces assertions peuvent donc jouer un grand rôle dans la vérification des programmes.

#### 2.2.2.5 La chaîne de compilation de lustre

La compilation d'un nœud LUSTRE engendre un automate à états finis au format OC (commun avec ESTEREL) [PS90]. Ceci permet à tout programme LUSTRE de bénéficier des mêmes outils de preuves qu'ESTEREL. Comme tout langage synchrone, LUSTRE est doté d'un compilateur capable de vérifier la cohérence sémantique des programmes :

- absence de récursion dans les nœuds,
- absence de cycle et de blocage dans les programmes,
- cohérence des horloges,
- initialisations des variables.

Il possède ses propres outils de preuves `lesar` [Ber92], `bac` [Hal94] . La vérification d'une propriété se fait en compilant celle-ci avec le code de l'application. La machine d'états finis résultante est ensuite étudiée de manière symbolique par les outils de preuves. Notons que ces derniers n'ont pas besoin de construire explicitement l'automate d'états correspondant pour effectuer leur vérification.

### 2.2.3 STATECHARTS

Les STATECHARTS [Har87] ont été créés à partir d'une extension du modèle d'automate d'états, pour décrire des systèmes réactifs complexes. Comme leur prédécesseur, ce modèle graphique s'appuie sur la notion d'état et de transition. Il apporte par contre les notions de *hiérarchie* et de *parallélisme* qui ne figure pas dans le modèle d'états classique.

Ce modèle est synchrone : les évolutions sont déclenchées sur événement et les actions atomiques associées s'exécutent en temps nul. De plus les communications sont instantanées et le franchissement d'une transition peut conduire par implications successives à une modification complète de l'état global.

Dans le modèle STATECHARTS, chaque état peut se décomposer en plusieurs sous-états activé simultanément lorsque l'état "père" devient actif. L'expression du parallélisme s'exprime par des lignes pointillées qui séparent l'état en plusieurs morceaux. Ce modèle facilite donc l'expression des *exclusions* entre les états ou les *activations multiples* d'états.

Grâce à leur lisibilité, les STATECHARTS sont adaptés pour spécifier les systèmes de commande. Il permettent surtout d'éviter le phénomène classique d'explosion des états que l'on rencontre dès que le système est composé de multiples organes indépendants. En effet les automates ne supportent pas l'expression du parallélisme et mettent tous les comportements à plat. En particulier lorsque les sous-systèmes sont totalement indépendants, l'automate global est le produit de tous les automates locaux.

La hiérarchie est une autre caractéristique des STATECHARTS. Elle évite aussi l'explosion des états. Elle décrit le système par *raffinements successifs* via une analyse descendante.

La figure 2.2 montre clairement les ressemblances et les différences entre le modèle STATECHARTS et le modèle d'automate de comportement équivalent. Nous pouvons remarquer d'emblée le nombre d'états et le nombre de transitions nécessaires pour exprimer le même comportement avec un automate d'états fini classique. Sur le statechart, l'état initial nommé S0, se distingue par sa flèche entrante. Les événements qui déclenchent le passage d'un état à un autre, sont eux-même portés par des flèches.

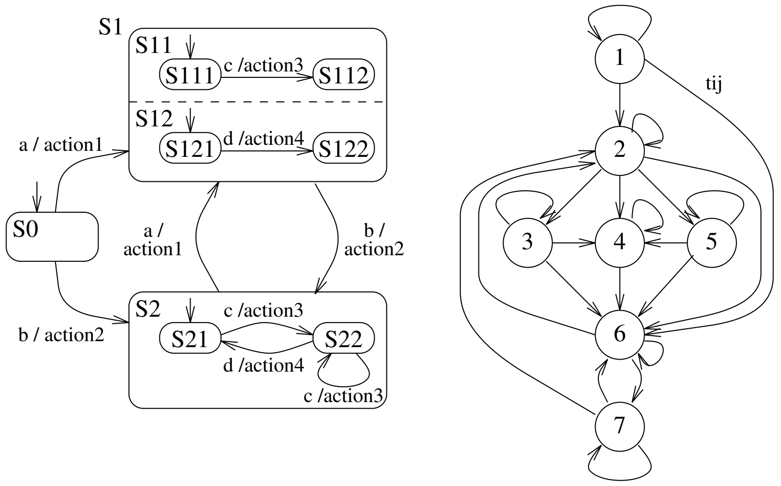



FIG. 2.2 - *Équivalence de comportement entre un statechart et un automate*

Les transitions $t_{ij}$ partant de l'état $i$ vers l'état $j$, prennent la forme :

$t_{11} : \neg a \wedge \neg b/$
$t_{12} : a/action1$
$t_{16} : \neg a \wedge b/action2$
$t_{22} : \neg b \wedge \neg c \wedge \neg d/$
$t_{23} : \neg b \wedge c \wedge \neg d/action3$
$t_{24} : \neg b \wedge c \wedge d/action3, action4$
$t_{25} : \neg b \wedge \neg c \wedge d/action4$
$t_{26} : b/action2$
$t_{33} : \neg b \wedge \neg d/$
$t_{34} : \neg b \wedge d/action4$
$t_{36} : b/action2$
$t_{44} : \neg b/$
$t_{46} : b/action2$
$t_{54} : \neg b \wedge c/action3$
$t_{55} : \neg b \wedge \neg c/$
$t_{56} : b/action2$
$t_{62} : a/action1$
$t_{66} : \neg a \wedge \neg c$
$t_{67} : \neg a \wedge c/action3$
$t_{72} : a/action1$
$t_{76} : \neg a \wedge d/action4$
$t_{77} : \neg a \wedge c/action3$

Si les avantages graphiques des STATECHARTS sont nombreux, ils ont cependant de nombreux inconvénients sémantiques. En premier, les évolutions peuvent être indéterministes. La figure 2.3 montre un tel cas : lorsque l'événement $a$ est présent, soit le système évolue normalement vers le sous-l'état $S112$, soit la préemption conduit à l'état $S12$. Ici l'existence de transitions non disjonctives issues d'un même état, est interprétée comme un choix non déterministe.

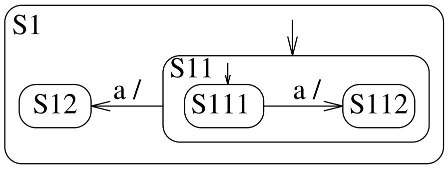



FIG. 2.3 - *Comportement non déterministe*

Le second problème est lié aux multiples interprétations que l'on peut donner aux symboles graphiques. De nombreuses variantes des STATECHARTS sont apparues. Dans [von94], l'auteur a ainsi dénombré 21 modèles de STATECHARTS différents!

Notons que ARGOS [Mar90] a été inclut dans cet ensemble, malgré l'opposition des ses auteurs. Il se distingue en effet des autres STATECHARTS par ses mécanismes de communication par *événements*, son caractère déterministe et la possibilité de traduire tout comportement en ESTEREL. De plus, ce langage est doté d'un environnement de validation ARGONAUTE qui permet *la description des systèmes, l'obtention des modèles et leur validation.*

Parallèlement aux STATECHARTS, a été développé l'outil d'aide à la spécification STATEMATE [HLN+90]. Il permet de spécifier tout système réactif en l'étudiant selon trois vue:

- **Vue architecturale** : Elle est basée sur l'utilisation de modules (appelés **modulescharts**), qui expriment de manière hiérarchique la structure physique du système et son environnement.

- **Vue fonctionnelle** : Elle décrit les unités fonctionnelles (**activity-chart**) sous forme de boîtes noires reliées par des flots de données et des flots de contrôle. Ces derniers expriment les relations de causalité entre activités.

- **Vue comportementale** : Elle permet de décrire chaque activité de contrôle sous forme de STATECHARTS.

À partir de ces trois représentations, STATEMATE permet de réaliser des simulations, d'éditer des rapports et de générer du code (C ou ADA).

## 2.3 Nécessité d'une machine d'exécution

La compilation d'un programme synchrone génère une *machine d'états* sous forme d'un automate fini ou d'un système d'équations. Cette machine d'états voit l'univers de manière synchrone. En particulier, elle considère le temps comme discret et ne s'intéresse qu'aux instants d'activation. Ainsi, il est nécessaire de définir un exécutif de langage synchrone (aussi appelé machine d'exécution) pour lier l'univers synchrone des machines d'états à l'univers asynchrone réel.

Des études ont déjà été réalisée sur la définition formelle des exécutifs. Citons par exemple [AMP91], qui s'est intéressé à l'implantation des automates issus de la compilation par ESTEREL. Les auteurs ont découpé cette machine d'exécution suivant trois grandes fonctionnalités :

- Mémorisation de l'environnement,
- Gestion des réactions,
- Traitements asynchrones.

La **mémorisation des événements** consistent à prendre en compte les *signaux physiques d'entrée* du système. Ceci est réalisé soit par des gestionnaires d'interruptions dédiés, soit par scrutation périodique des ports physiques. Le résultat peut ensuite être mémorisé dans des tables.

Les sorties sont de même préalablement mémorisées pour générer un vecteur de sortie unique qui sera ensuite transmis vers l'extérieur (émission synchrone).

Le **gestionnaire de réactions** permet de figer les entrées à un instant donné pour les présenter à l'automate sous forme d'événement. Il provoque ensuite la transition de l'automate, puis met à jour la table des sorties. Notons que le figeage des entrées est ici très important pour respecter l'hypothèse *d'atomicité des réactions* spécifique à l'approche synchrone.

Les **traitements asynchrones** regroupent toutes les tâches asynchrones de l'application. Ces tâches ne communiquent entre-elles que par l'intermédiaire de l'automate et forment *la partie transformationnelle* du système. Les politiques d'ordonnancement choisies doivent respecter les contraintes temps-réel précédemment exposées. Dans un système informatique classique, la gestion des tâches auraient été effectuées par l'ordonnanceur de tâches interne au système d'exploitation. Dans cet exécutif, elles sont créées, preemptées et détruites par un gestionnaire spécifique, qui reçoit ses ordres directement de l'automate.

D'autres travaux [Par92, Per93] ont affiné ces principes. Ils visent à définir des machines très génériques peu dépendantes de la machine cible et générées automatiquement.

Plusieurs implantations sur des systèmes réels ont été effectuées [Gaf91, AP92b]. Elles incluent aussi les fonctionnalités précédentes et montrent la faisabilité de ce type d'application.

**Remarque** : Choisir l'instant de déclenchement de l'automate en fonction des entrées, est un problème très délicat. La politique la plus simple consiste à déclencher *l'instant* à chaque fois qu'une nouvelle donnée asynchrone arrive. Mais cette solution n'est pas viable si les données peuvent arriver de manière sporadique. D'autres politiques plus intéressantes sont possibles. Dans [Per93], l'auteur préconise une approche par nécessité.

## 2.4 Objets synchrones

### 2.4.1 Introduction

Parmi les méthodes de conception asynchrones, l'approche orientée objet a connu un réel essor ces dernières années. Que ceux soient OMT [RBP+91] (pour Object oriented Modeling Technique), ou ROOM [SGW94] (pour Real-Time Object Oriented Modeling), toutes préconisent le regroupement des fonctions des comportements communs au sein d'entités autonomes *les objets* et permettent de raisonner sur ces objets sans se soucier de leur implémentation (abstraction des données).

Les modèles orientés objet s'appuient sur les propriétés d'encapsulation et d'héritage.

- **Encapsulation**: L'encapsulation permet de regrouper dans l'objet des structures de données spécifiques et les fonctions qui manipulent ces données. Ces données et ces fonctions peuvent alors devenir totalement invisible de l'extérieur. L'objet ainsi constitué, peut être modifié ou interrogé par des fonctions normalisées (appelées *méthodes*). Enfin, pour dupliquer facilement cet objet, les structures de données et les méthodes sont regroupées dans un prototype unique appelé: *classe*.
- **Héritage**: Il est fréquent que deux objets (ou plutôt deux classes d'objet) soit très proches, soit par leur structure interne, soit par leur comportement. Le principe d'héritage consiste à définir la seconde classe comme une copie de la première. Chaque structure ou méthode héritée peut ensuite être enrichie ou modifiée suivant la spécificité de la classe.

### 2.4.2 Présentation et Intérêts des objets synchrones

Les objets synchrones que nous présentons maintenant, reposent sur les travaux de Boulanger [Bou93]. Écrits en C++, ils ont permis d'une part, de concilier les approches synchrone et objet. D'autres part, ils ont apporté une réponse intéressante aux problèmes de génération de machines d'exécution (chapitre 2.3).

Pour l'auteur, un "objet synchrone est un objet ayant l'interface suffisante pour intégrer les modules synchrones[5] dans le langage à objet". L'adjectif synchrone rappelle que ces objets communiquent entre-eux dans le même *instant logique* pour conserver l'hypothèse *d'atomicité des réactions*. Pour cela Boulanger a défini dans sa thèse un ensemble de méthodes spécifiques aux objets synchrones qui définissent un protocole de communication.

Pour faciliter et accélérer le passage de la modélisation (en langage synchrone) à la réalisation effective, l'auteur a écrit un générateur automatique de code. OCC++ permet de traduire tout comportement décrit en OC par un automate fini déterministe ou un système d'équations, en une classe synchrone C++.

Le choix de OC est ici très judicieux. En effet, nous avons vu au chapitre 2.2.1 que ce format était commun à plusieurs langages synchrones. Les travaux sur les objets synchrones sont a priori *applicables sur n'importe quel formalisme synchrone* dès que celui-ci dispose d'un compilateur vers OC. De même la propriété d'encapsulation des objets, cache la provenance de ces objets. Elle permet donc l'utilisation de plusieurs formalismes synchrones en même temps pour modéliser, puis réaliser l'application.

L'apport des objets synchrones est significatif lorsqu'un système réactif peut de décomposer en plusieurs systèmes autonomes. Pour que le comportement d'un ensemble d'entités synchrones soit synchrone, il faut que toutes réagissent dans le *même instant*. Une première solution consiste à compiler toutes ces entités ensemble, ce qui induit souvent une explosion du nombre d'états de l'automate final. L'approche de Boulanger est différente. Il préconise de compiler ces entités séparément pour aboutir à des objets synchrones, puis de trouver un ordre correct de réaction des objets. Cet ordre est donné par les relations de dépendance entre objet. La figure 3.8 (extraite de [Bou93]) montre clairement que seul l'ordonnancement (`A`,`B`) respecte l'hypothèse synchrone. En effet `B` a besoin de connaître la valeur de $\beta$ pour transiter. Or cette valeur ne sera remise à jour (dans l'instant logique courant) uniquement lorsque `A` aura transité.

Pour que cet ordonnancement existe, il suffit que le graphe de dépendance soit acyclique. Cette condition peut être vérifiée automatiquement dès l'établissement des connexions. L'auteur propose donc un outil de *spécification d'objets*

5. Ces modules sont décrits dans un langage synchrone ou directement en C++.

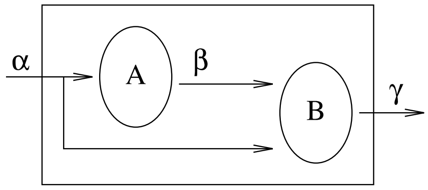



FIG. 2.4 - *Ordonnancement des réactions internes*

*synchrones* (MDL), construits à partir d'autres objets synchrones. De l'extérieur la réaction du nouvel objet est atomique. En interne, elle se décompose en la cascade des réactions de ces constituants.

MDLC permet aussi la génération de l'application (exécutif). Pour gérer l'interface entre le code synchrone et l'univers asynchrone extérieur, il inclut des objets synchrones spécialisés responsables de l'acquisition des entrées asynchrones et de l'émission des sorties asynchrones. Pour respecter l'hypothèse synchrone, l'ordonnanceur activera à chaque instant, les gestionnaires d'entrées, puis les objets synchrones, puis les gestionnaires de sortie. Comme les entrées-sorties sont très dépendantes de la machine cible, les gestionnaires associés doivent être écrits par le concepteur.

En partant de la modélisation du comportement par des langages synchrones, l'approche de Boulanger permet de générer *automatiquement* une très grande partie de l'exécutif. Elle réduit ainsi les sources d'erreurs humaines liées au programmeur.

## 2.5 Conclusion

Dans ce chapitre, nous avons présenté des avantages de l'approche synchrone pour modéliser puis implanter les systèmes réactifs temps-réels. Cette étude s'est appuyée sur une présentation des langages synchrones classiques. En particulier ESTEREL possède un *support mathématique puissant.* Sa sémantique non ambiguë permet d'appréhender de manière déterministe le parallélisme de l'application et d'établir formellement des preuves de comportement. Dans les chapitres suivants, nous ferons souvent recours à ce langage pour clarifier certains comportements.

Au niveau implantation, l'approche synchrone nécessite tout de même la mise en place d'une *machine d'exécution* qui permet de coupler l'univers synchrone à l'univers asynchrone réel. Pour que toutes les propriétés et les preuves logiques de comportement soient effectives, l'exécutif doit respecter toutes les hypothèses du synchrone en particulier la propriété *d'atomicité des réactions.* Sa réalisation est en fait guidée par deux critères que sont : la *généricité* et *l'optimisation tem-*

*porelle* sur la machine cible. Une réponse partielle a été apportée par les *objets synchrones* qui permettent de générer automatiquement une très grande partie de l'application à partir d'une spécification synchrone des comportements. Par contre l'optimisation du code pose encore problème et rejoint d'ailleurs celui de la compilation efficace des langages à objets.

Parmi les langages et formalismes synchrones importants, nous n'avons pas introduit le GRAFCET. Nous allons maintenant lui consacrer les chapitres 3 et 4 pour présenter complètement le modèle et ses ambiguïtés, puis formaliser la recherche de *l'état suivant*. Nous bénéficierons pour cela de l'expérience acquise avec l'approche synchrone. Notre pensons en effet que les langages synchrones comme ESTEREL ont fait progresser la compréhension des modèles synchrones en utilisant des concepts mathématiques nouveaux pour le domaine.

# Chapitre 3

# Le modèle GRAFCET

## 3.1 Introduction

Le GRAFCET (acronyme de "GRaphe de Commande Etape Transition") est largement utilisé dans les applications de systèmes automatisés industriels. Il permet de spécifier le comportement de la partie commande des systèmes automatisés de production (SAP). Son aspect graphique permet facilement d'expliciter le parallélisme et la séquentialité des actions. Pour R.David [Dav95], ce modèle constitue une bonne alternative au réseaux Ladder.

Des groupes de réflexion, notamment dans le cadre de l'AFCET, mènent des études amont sur le modèle et ses extensions. Lors du congrès "GRAFCET 92", des études sur la sémantique du GRAFCET [LPR92, FC92] et ses relations avec les *langages synchrones* [AP93, ML92] ont été présentées. Ces recherches traduisent des convergences d'intérêts entre les automaticiens qui utilisent le GRAFCET depuis de nombreuses années et les informaticiens jusqu'ici plutôt concernés par les langages.

Les références sur le GRAFCET sont de deux types : les normes et les ouvrages. Les *normes* [AFN82, IEC88, AFN93] ont un caractère officiel mais restent vagues sur certains points. Les *ouvrages* sont généralement didactiques et peuvent servir de guide à l'utilisateur. Ils vont au-delà de la norme dans un souci de clarification et d'efficacité (citons [DA89, BBC+92] parmi les plus récents). Les auteurs sont amenés à exprimer leur vision ou celle d'un groupe. Des divergences d'interprétation peuvent donc exister.

# 3.2 Présentation du modèle

## 3.2.1 Formalisme graphique de base associé au modèle

La syntaxe GRAFCET est constituée d'un ensemble réduit de symboles graphiques.

### 3.2.1.1 Etapes et macro-étapes

Une étape caractérise un *état local* du système à un instant donné de son évolution. Par définition, une étape est soit **active**, soit **inactive**. L'ensemble des étapes actives d'un grafcet à un instant donné définit donc la *situation* de ce GRAFCET à cet instant. Une étape se représente par un carré, elle est identifiée par un repère alpha-numérique qui est classiquement un numéro (fig. 3.1). Les étapes initiales caractérisent l'état initial du GRAFCET, elles se distinguent graphiquement par leur représentation en double carré.

Pour faciliter la lecture d'un grafcet conséquent, des sous-ensembles d'étapes peuvent être regroupés en étapes simples. Ces étapes sont alors appelées "macro-étapes". La norme autorise ce type de regroupement à condition que le sous-ensemble n'ait qu'une étape d'entrée et qu'une étape de sortie. Comme cette simplification est purement syntaxique, elle sera ignorée dans la suite du mémoire.

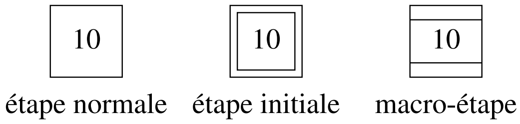



FIG. 3.1 - *Différents types d'étapes*

### 3.2.1.2 Transitions et réceptivités

Les transitions indiquent la possibilité d'évolution entre certaines étapes. Cette évolution est accomplie par le **franchissement** de la transition et provoque un changement d'activité des étapes. Une transition se représente par un trait perpendiculaire aux liaisons joignant deux étapes (fig. 3.2). Il ne peut y avoir qu'une seule transition entre deux étapes. Par définition, une transition est dite "*validée*" si les étapes directement amont sont actives (même transitoirement).

À chaque transition, est associée une condition logique appelée **réceptivité**. Ces conditions peuvent prendre n'importe quelle forme (expression booléenne, comparaison, test de temporisation) pourvu qu'elles puissent être évaluées à vrai ou faux. Cette évaluation permet de déterminer si la transition préalablement *validée* est *franchissable* (cf. 3.2.2.2).

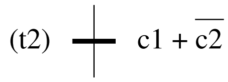



FIG. 3.2 - *Transition munie de sa réceptivité*

#### 3.2.1.3 Liaisons orientées

Elles servent à indiquer les voies d'évolution du GRAFCET en reliant les étapes aux transitions et les transitions aux étapes. Ces liaisons sont représentées par des lignes horizontales ou verticales. Par convention, le sens des évolutions s'effectue du haut vers le bas. Une flèche doit être disposée sur les traits verticaux lorsque cette convention n'est pas respectée.

Ces liaisons permettent d'exprimer des structures classiques comme la séquentialité, la condition ou le parallélisme. Le GRAFCET distingue en particulier :

- Les liaisons simples :

  La sortie d'une étape (resp. d'une transition) est reliée à l'entrée d'une transition (resp. d'une étape). Ces liaisons expriment la séquentialité des opérations associées aux étapes.

- Les divergences et convergences en OU :

  Dans le cas d'une divergence (resp. convergence) en OU, la sortie (resp. l'entrée) d'une étape est reliée à $n$ entrées (resp. $n$ sorties) de transition comme le montre la figure 3.3. La convergence en OU permet de regrouper plusieurs branches d'évolution vers un traitement commun. Les divergences en OU sont utilisées pour choisir une branche d'évolution particulière suivant la valeur des entrées. Remarquons à ce niveau que des réceptivités non exclusives peuvent conduire à l'activation de plusieurs branches (parallélisme interprété).

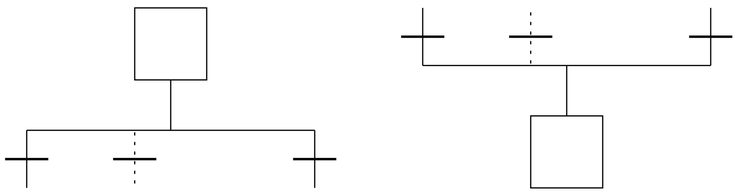


FIG. 3.3 - *Divergence et convergence en OU*

- Les divergences et convergences en ET :

Dans le cas d'une divergence (resp. convergence) en ET, la sortie (resp. l'entrée) d'une transition est reliée à $m$ entrées (resp. $m$ sorties) d'étape comme

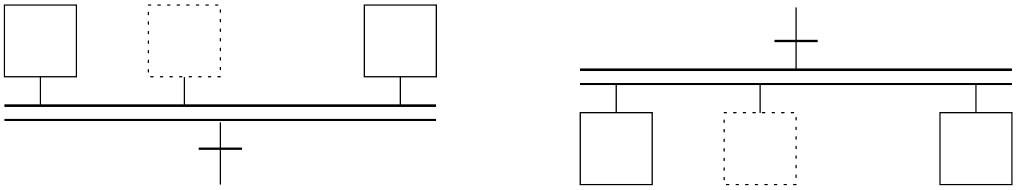


FIG. 3.4 - *Divergence et convergence en ET*

le montre la figure 3.4. La convergence en ET permet de synchroniser plusieurs branches entre elles. La divergence en ET est utilisée pour exprimer explicitement le parallélisme de traitement.

#### 3.2.1.4 Actions

Les actions associées à une étape caractérisent les ordres ou les commandes émis vers la partie opérative lorsque l'étape considérée est activée. La norme définie deux grandes familles que sont les actions continues (ou à niveau) et les actions impulsionnelles. Leur sémantique sera étudiée au chapitre 3.3.3.

Une action est représentée par un rectangle relié par un tiret à l'étape concernée. Ce rectangle est lui-même séparé en trois parties normalisée [IEC88] [BBC+92] (fig. 3.5). La section "a" est facultative : elle contient le type d'action sous forme d'une combinaison de lettres[1] qui sont répertoriées dans le tableau 3.2.1.4. La section "b" contient la déclaration de l'action. La section "c" est facultative : elle définit une variable permettant de gérer la fin de l'exécution [IEC93].

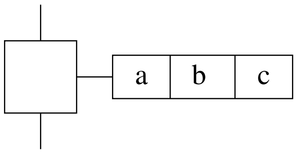



FIG. 3.5 - *Représentation normalisée d'une action*

**Remarques :**

- Si plusieurs actions sont associées à une étape, leur ordre de déclaration n'implique aucune priorité entre ces actions.
- Les actions de type D et L seront de nouveau développées dans le chapitre 3.3.4.2 car elles posent des problèmes de frontière du modèle GRAFCET

1. Par exemple CD pour conditionnée, puis limitée en temps

| lettre | définition |
|---|---|
| N | Action normale ou continue |
| P | Action impulsionnelle |
| C | Action conditionnelle |
| D | Action retardée |
| L | Action limitée dans le temps |
| S | Activation d'une action mémorisée |
| R | Désactivation d'une action mémorisée |

TAB. 3.1 - *Table des types d'actions*

- Les actions de type S et R peuvent être vues comme des cas particuliers d'actions impulsionnelles

#### 3.2.1.5 Hiérarchie de grafcets

Un grafcet peut être partitionné en sous-ensembles connexes (au sens de la théorie des graphes). Chaque entité prend alors le nom de "grafcet connexe". La réunion de deux (ou plus) grafcets connexes est appelé "grafcet partiel". La réunion de *tous* les grafcets intervenant dans une application est appelée "grafcet global".

Lorsque qu'un grafcet global est composé d'une multitude de grafcets connexes, chacun d'eux évoluent indépendamment des autres (parallélisme d'évolution). Il est cependant possible de hiérarchiser les grafcets connexes entre-eux grâce aux ordres de forçage et de figeage. Ces ordres permettent à un grafcet de forcer l'état d'un ou plusieurs autres. Le premier grafcet peut alors être vu comme un contrôleur de grafcets de niveau inférieur. Cette hiérarchie permet de mieux structurer les applications de grande taille, de ne pas surcharger l'écriture d'un grafcet en remontant au niveau supérieur la gestion de certains événements (classiquement arrêts d'urgence). Par contre la cohérence de la hiérarchie impose un forçage arborescent des grafcets connexes entre-eux. La norme définit deux types de forçage :

- Le grafcet $G$ est forcé dans une situation donnée caractérisée par l'ensemble des étapes actives $\{Xi, Xj...\}$ :

  $F/G : \{Xi, Xj...\}$

  Cet ensemble peut être vide. $G$ est alors désactivé.

- Le grafcet $G$ est forcé dans la situation courante (figeage) :

  $F/G : *$

### 3.2.2 Sémantique de comportement

Nous avons exposé dans la section 3.2.1 la syntaxe GRAFCET en apportant quelques éléments de comportement. Nous allons maintenant présenter les postulats et les règles d'évolutions du modèle ainsi que leurs différentes interprétations.

#### 3.2.2.1 Postulats fondamentaux relatifs aux évolutions

Les évolutions et l'interprétation du temps sont guidées par trois postulats de base [BBC+92] [AFN93].

– *"À l'échelle du temps externe*[2]*, tout changement d'état des entrées est pris en compte par le modèle, dès son apparition."*

– *"La totalité des conséquences de cet événement sur le modèle est déterminée à temps nul."*

– *"Depuis l'extérieur du modèle, les événements d'entrée et les états des sorties qui en résultent sont vus à la même date."*

À ces trois postulats, les concepteurs du GRAFCET ont émis dès le début, l'hypothèse que deux événements non-corrélés ne pouvaient arriver en même temps.

**Remarque :** Ces trois postulats se rencontrent aussi dans les langages synchrones : ils sont d'ailleurs fondamentaux. Leur absence dans le modèle GRAFCET aurait fortement gêné les tentatives d'intégration dans une plate-forme synchrone commune.

#### 3.2.2.2 Règles d'évolution

Ces règles ne figurent pas dans la norme mais sont généralement adoptées.

– Règle 1 : situation initiale.
*La situation initiale* du GRAFCET caractérise le comportement initial de la partie commande vis-à-vis de la partie opérative et correspond aux étapes actives au début du fonctionnement.

– Règle 2 : franchissement d'une transition.
L'évolution de la situation du GRAFCET correspondant au *franchissement* d'une transition. Il ne peut se produire :
- que lorsque cette *transition* est validée

2. Nous reviendrons au chapitre 3.2.2.4 sur les notions de temps internes et de temps externes.

- et que la réceptivité associée à cette transition est vraie.
Lorsque ces deux conditions sont réunies, la transition devient *franchissable*. Elle est alors *obligatoirement* franchie.

– Règle 3 : évolution des étapes actives.
Le franchissement d'une transition entraîne simultanément l'activation de *toutes* les étapes immédiatement suivantes (aval) et la désactivation de *toutes* les étapes immédiatement précédentes (amont).

– Règle 4 : évolutions simultanées.
Plusieurs transitions simultanément franchissables sont *simultanément* franchies.

– Règle 5 : activation et désactivation des étapes.
Si lors d'une évolution, une même étape est désactivée et activée simultanément, elle reste active.

Ces règles possèdent deux interprétations principales que nous allons maintenant étudier : SRS et ARS.

#### 3.2.2.3 Interprétation Sans Recherche de Stabilité

Cette interprétation[3] (notée SRS) est classiquement utilisée dans le milieu industriel car elle est facile à mettre en œuvre et correspond à une première idée intuitive des évolutions. Elle consiste à :

- lire les entrées
- franchir toutes les transitions franchissables
- mettre à jour l'état de chaque étape
- mettre à jour les actions

Dans ce mode d'évolution, le problème de la stabilité des étapes ne se pose pas : une étape activée reste au moins active jusqu'à la prochaine acquisition des entrées. En effet l'évaluation des transitions franchissables est réalisée *une et une seule fois* avant l'activation des étapes. Ces activations peuvent avoir des conséquences sur la franchissabilité de certaines transitions, mais il est *trop tard* pour les prendre en compte.

La figure 3.6 montre un exemple d'évolution. Dès que $a$ est vraie, les transitions $t1$ et $t2$ sont franchies simultanément. Par contre la transition $t3$ n'est pas

3. Sémantique d'évolution.

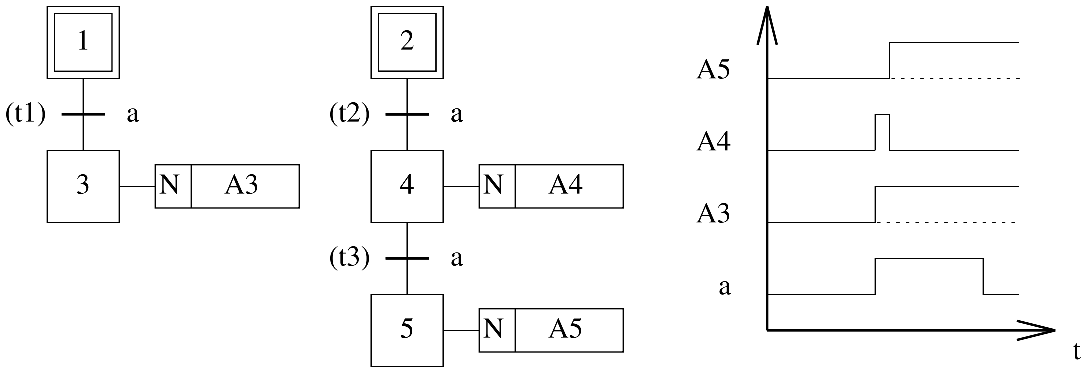



FIG. 3.6 - *Interprétation sans recherche de stabilité*

franchie car non validée. Ce franchissement conduit au nouvel état $\{X3, X4\}$ et à l'activation des actions $A3$ et $A4$. La transition $t3$ est à son tour franchie, ce qui conduit au nouvel état $\{X3, X5\}$ et à l'émission des actions $A3$ et $A5$. Cette évolution est résumée dans le chronogramme, on observe en particulier l'émission fugitive de l'action $A4$.

De nombreux ouvrages [GRE85] [BBC+92] encouragent les utilisateurs à abandonner l'interprétation SRS car elle respecte mal les postulats de base du modèle et n'est pas conforme à la sémantique des actions continues. De plus la mise en action intempestive de la partie opérative risque de provoquer des accidents corporels graves ou la destruction du système. Pour ces raisons, nous abandonnerons complètement l'interprétation SRS dans la suite de ce mémoire.

#### 3.2.2.4 Interprétation Avec Recherche de Stabilité

Cette interprétation (notée ARS) diffère de la précédente par la recherche d'un nouvel état stable pour toutes variations des entrées. Elle peut se résumer ainsi [Lep94] :

- lire les entrées
- franchissement et recalcul des transitions franchissables tant que la situation est instable
- mettre à jour l'état de chaque étape
- mettre à jour les actions

Contrairement à l'interprétation SRS, l'activation des nouvelles étapes aboutit à une situation temporaire qui peut rendre franchissable de nouvelles transitions. Ces dernières sont alors franchies et activent à leur tour de nouvelles étapes.

Lorsque le système est stabilisé, l'état des étapes est réellement mis à jour. Pour respecter les postulats et les règles du modèle, toutes ces évolutions doivent être instantanées pour un observateur externe. En particulier les entrées doivent être figées et les sorties émises lorsque le système s'est stabilisé.

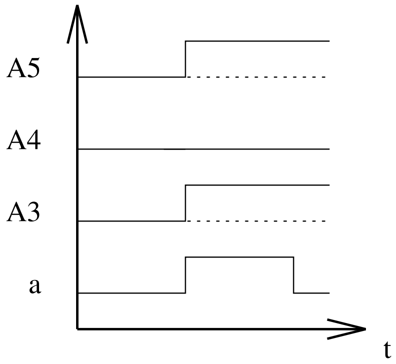



FIG. 3.7 - *Interprétation avec recherche de stabilité*

La figure 3.7 montre le chronogramme obtenu sur l'exemple précédent : l'action $A4$ n'est plus du tout émise et l'action $A5$ est émise dès que $a$ est vrai.

**Remarques :**

La situation est stable si *plus aucune transition n'est franchissable.* Ce critère de stabilité est classiquement utilisé en GRAFCET.

Si le grafcet ne respecte pas ce premier critère, il serait à la limite possible, de définir un second critère de stabilité basé sur *l'identité de deux situations successives.* Comme la classe de grafcets concernés est tres restreinte et spécifique, nous ne conserverons dans la suite de ce mémoire que le premier critère.

Le caractère instantané des réactions et la recherche de stabilité ont conduit les concepteurs du modèle à définir une double échelle de temps (temps externe, temps interne) [AFN93] [BBC+92]. Ainsi chaque évolution du système a une *durée nulle* en temps externe et une *durée non nulle* en temps interne pour permettre un mode d'évolution interne SRS.

### 3.2.3 Prise en compte des ordres de forçage

Les ordres de forçage présentés au chapitre 3.2.1.5 sont prioritaires devant les règles d'évolution du modèle. Ils sont vus comme des *actions continues standards* par la norme et sont donc rattachés à l'activité d'une étape. En particulier ils sont maintenus tant que les étapes associées sont actives. Or l'activité de cette étape est elle-même dépendante des règles d'évolution. Pour respecter les interprétations ARS et SRS précédentes, il est nécessaire de trouver à chaque évolution du système, un *ordonnancement* des grafcets connexes. Il faut ensuite faire transiter tous les grafcets dans l'ordre établi. La figure 3.8 montre un exemple de forçage où seuls les ordonnancements $(G2, G1, G3)$ et $(G2, G3, G1)$ respectent la dynamique du modèle.

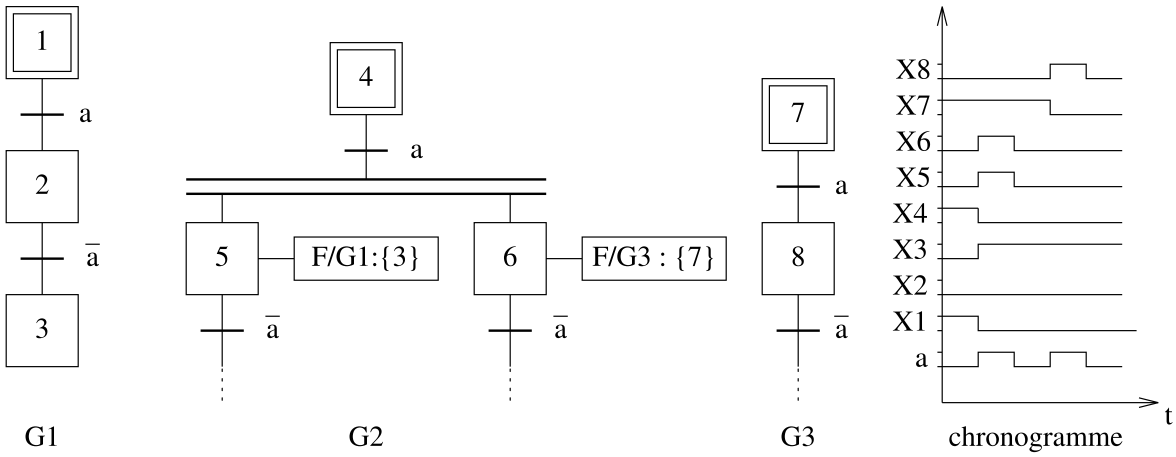



FIG. 3.8 - *Forçage de grafcets*

Ce thème fait actuellement l'objet de recherches. Il s'agit aussi de détecter statiquement tout grafcet incorrect (boucle de forçage ou instabilité liée au forçage). Citons en particulier J.J. Lesage et J.M. Roussel [LR92] qui utilisent la théorie des graphes pour aborder ces problèmes. Notons que ces questions ont beaucoup de similitudes avec des problèmes spécifiques aux langages synchrones : *répartition* au sein des *machines d'exécution* et *causalité*.

Malgré cela, il existe plusieurs interprétations sémantiques pour certains cas très spécifiques concernant en particulier la libération des forçages et les forçages simultanés de grafcets. La notion de forçage est donc susceptible d'évoluer encore dans les prochaines normes du GRAFCET. Il serait par exemple plus judicieux de considérer ces ordres comme des *actions impulsionnelles internes* [LPR92]. C'est pourquoi nous avons préféré pour l'instant, ne pas intégrer cette notion dans notre modèle.

## 3.3 Ambiguïtés du modèle

Le modèle GRAFCET possède un certain nombre d'ambiguïtés que nous allons maintenant présenter. Lorsque plusieurs choix sémantiques seront possibles, nous prendrons celui qui facilite l'intégration du GRAFCET parmi les autres modèles synchrones. Nous serons également amené à ne pas retenir la notion de *double échelle de temps* qui est maintenant officialisée dans la nouvelle norme [AFN93]. Pour éviter toute équivoque entre la norme et notre interprétation des règles d'évolution, R. David nous a suggéré de nommer différemment notre modèle. Nous avons opté pour S-GRAFCET (Synchronous Grafcet). Cette discussion servira donc de base à sa formalisation complète, présentée au chapitre 4).

## 3.3.1 Problèmes liés aux réceptivités

Considérons le *Système à Evénements Discrets (S.E.D.)* réactif ayant une entrée booléenne `A`, deux sorties booléennes `ON` et `OFF` et dont le comportement est défini par le grafcet $G1$ (Fig. 3.9).

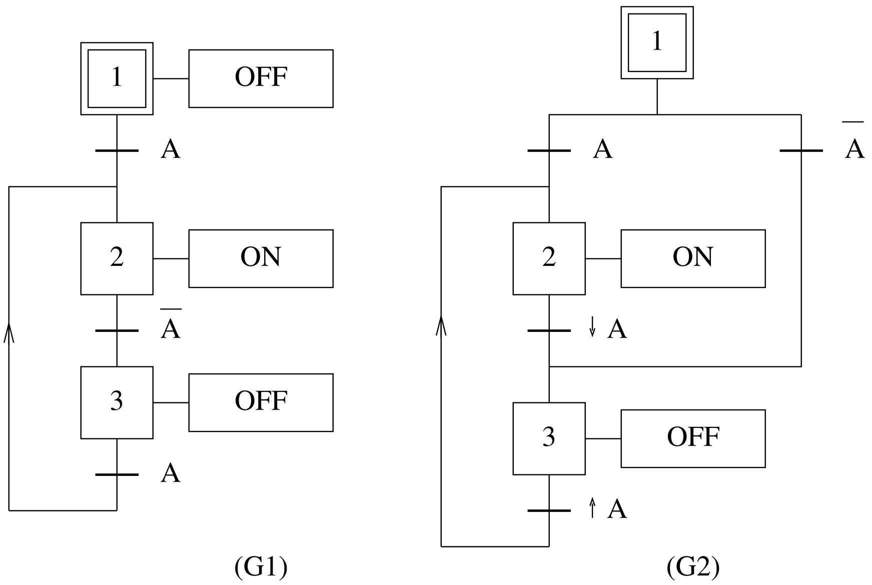



FIG. 3.9 - *Grafcets de marche - arrêt*

Pour analyser le comportement de ce système, nous utilisons (sauf indication contraire) les règles classiques d'évolution avec *recherche de stabilité*.

### 3.3.1.1 Distinction entre événements et conditions

L'action `ON` est lancée dès que la réceptivité `A` est vraie, alors que l'action `OFF` est réalisée lorsque la réceptivité `A` repasse à faux. En fait, ce sont les occurrences d'événements externes (changement d'état de `A`) qui provoquent l'évolution du système. Pour certains auteurs, comme C.A. Petri, un événement se définit par, et uniquement par, l'ensemble des conditions qu'il modifie (principe d'extensionalité

[Pet80]). Dans les "Systèmes Evénement/Condition", les concepts d'événement et de condition sont donc nécessairement distincts. La dualité événement/condition se retrouve dans de nombreux modèles de systèmes réactifs. Calvez [Cal90], par exemple, attribue aux événements un caractère *fugace*, contrairement aux conditions qui expriment une certaine *permanence*. Moalla [Moa85] définit les événements comme des "faits" déclencheur d'évolution. Pour le GRAFCET, la norme française ne connaît que les *réceptivités* qui sont des expressions booléennes qui doivent être évaluées à `vrai` ou `faux`. La norme internationale intègre les événements mais reste floue sur leur signification. La plupart des ouvrages didactiques sur le GRAFCET autorisent la distinction entre événement et condition. Nous allons préciser cette idée, tout en allant au-delà du point de vue classique (voir [DA89]).

Dans l'exemple présenté, les événements sont les fronts montants ($\uparrow A$) et descendants ($\downarrow A$). Ils interviennent en tant que *garde* pour le déclenchement des évolutions. Les conditions ($A$ et $\neg A$) jouent plutôt un rôle de *sélection*. Le grafcet $G2$ (Fig. 3.9) de comportement équivalent au précédent, met en évidence ces différences de fonctionnalité. Le grafcet $G2$ diffère essentiellement de $G1$ par ses réceptivités qui ont été étendues à la notion de *front*: les deux conditions booléennes sur $A$, sont donc remplacées par des fronts montants et descendants de ce signal.

La seconde version nous paraît préférable, car les différences syntaxiques traduisent des différences sémantiques. En effet l'émission de `ON` est bloquée par l'attente de l'événement "front montant de $A$". On peut dire également que le passage à l'état `ON` est *gardé* par l'événement "front montant de $A$". De même le passage à l'état `OFF` est *gardé* par l'événement "front descendant de $A$". Par la garde qu'il impose, l'événement devient donc un déclencheur d'évolution contrairement à la condition qui joue un rôle de sélecteur ou de contrainte supplémentaire au franchissement d'une transition.

#### 3.3.1.2 Nécessité des événements

Généralement, dans le monde GRAFCET, les événements ne sont considérés que par l'intermédiaire des fronts. Il s'agit alors d'un *concept dérivé* de celui de condition (ou d'état). Les fronts sont donc des abréviations comme le montre la traduction proposée figure 3.10 [GRE85, p35].

Cette interprétation soulève des problèmes que nous allons détailler par la suite. Au préalable, signalons l'usage des événements et conditions dans d'autres modèles. Certains utilisent à la fois événements et conditions: e.g. les STATECHARTS. D'autres n'utilisent que les événements: Réseaux de Petri Synchrones (Hilal). Pour ESTEREL, l'événement est une notion fondamentale: seules les occurrences d'événements peuvent provoquer des changements. ESTEREL peut accéder

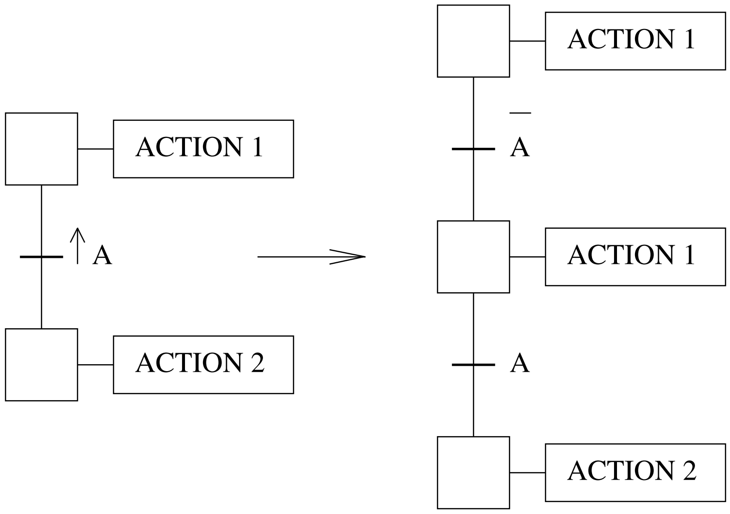



FIG. 3.10 - *Décomposition du front montant*

à des conditions issues de l'environnement par l'intermédiaire des "sensors", mais ces derniers ne sont jamais déclencheur d'évolution.

Lorsque la notion de front est définie indirectement à partir des niveaux logiques, on peut utiliser la formule de récurrence suivante:

$$\uparrow A(k) = \neg A(k-1) \wedge A(k)$$

Cette approche ne pose aucun problème dans les langages flots de données synchrones comme LUSTRE et SIGNAL car un signal suppose, pour eux, l'existence d'une horloge associée au signal. En l'absence de précision complémentaire l'horloge de base est utilisée, comme dans l'exemple LUSTRE suivant.

```
node rising_edge (A:bool) returns (upA:bool);
let
  upA = false -> (not pre(A) and A);
tel.
```

En GRAFCET l'existence d'une telle horloge n'est *nulle part mentionnée*, donc les équations de récurrence n'ont pas de sens. Pour pallier cette insuffisance théorique, des travaux de recherche récents ont permis de définir une algèbre de Boole qui inclut événement et condition [Rou94] [RL93]. Les "événements-conditions" sont considérés comme des fonctions booléennes continues à droite par morceaux qui admettent des doubles discontinuités. Certes intéressante, cette approche masque l'importance des événements, dont le rôle est de déclencher les évolutions de manière asynchrone[4].

4. Le terme "asynchrone" est pris ici au sens de l'Automatique et de la Logique : "non synchronisé sur une horloge de base"

Nous préférons donc envisager un *temps logique* et une sémantique “ à la ESTEREL” dans laquelle les événements jouent un rôle fondamental. Dans la suite nous considérons qu'un événement est une entité primitive dans le modèle, pas nécessairement liée à la variation d'un niveau logique. Si $\uparrow A$ est un événement, le dépassement d'une température donnée est également un événement (point de vue partagé par Calvez [Cal90] et Moalla [Moa85]). Nous considérons même qu'un événement peut être utilisé sans faire allusion à un quelconque changement d'état. Ce qui est important, c'est *le caractère fugace de l'occurrence d'un événement*, en cela un événement ne peut être confondu avec une condition.

#### 3.3.1.3 Ambiguïté des réceptivités

Revenons aux réceptivités telles qu'elles apparaissent dans le GRAFCET. Suivant le contexte (situation courante), une réceptivité est interprétée différemment. Considérons l'exemple de la figure 3.11. Si au cours d'une évolution l'étape k est activée et $A$ est vrai, alors l'étape k+1 est activée et l'étape k est désactivée. La réceptivité $A$ a été interprétée comme une *condition* imposant la poursuite de l'évolution. Par contre si $A$ est faux, l'évolution se bloque avec l'étape k active. Dans ce cas, l'étape k sera désactivée lorsque $A$ redeviendra vrai, ce qui s'exprime par une occurrence de $\uparrow A$. Cette fois $A$ sert de *garde* pour déclencher une évolution. D'où deux interprétations différentes pour la même notation (*condition* et *garde*); ceci peut être source d'ambiguïtés qu'il vaudrait mieux éviter.

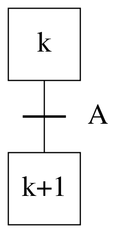



FIG. 3.11 - *Différence d'interprétation d'une réceptivité*

#### 3.3.1.4 Proposition pour les réceptivités

Dans notre approche une réceptivité peut faire intervenir des conditions, des événements ou les deux. Cette proposition est assez généralement adoptée par les utilisateurs et explicitée dans [DA89]. Nous insistons simplement sur le fait que événements et conditions sont deux entités indépendantes qui doivent être *des primitives du modèle.*

**Définition 1** *À chaque transition $t$ on associe une et une seule réceptivité $R(t)$ formée d'un* ***couple événement-condition*** $< G\,,\, C >$. *La partie événementielle est formée d'une conjonction (qui peut être vide) d'événements d'entrée. La partie conditionnelle est une expression booléenne classique qui fait intervenir les entrées booléennes et les variables d'état.*

**Définition 2** *Une transition $t$ est dite* ***gardée*** *ssi $R(t).G \neq \emptyset$.*

Généralement en GRAFCET on se limite à l'occurrence d'un seul événement par réceptivité ($R(t).G$ est alors un singleton).

## 3.3.2 Problèmes liés aux évolutions internes

### 3.3.2.1 Généralités

Lorsqu'on étudie le comportement d'un GRAFCET, dans le mode dit "avec recherche de stabilité", le problème essentiel est celui de déterminer "*LA* situation stable suivante, pour une situation et pour un état de l'environnement donnés".

- Comme il a été vu, cette évolution ne peut se faire que sur l'occurrence d'un événement.
- On cherche *LA* situation suivante car le déterminisme est une propriété imposée au GRAFCET.

**Notation:** Nous appelons *réaction* le changement d'une situation stable vers la situation stable suivante.

**Remarque:** Le problème posé est complexe car on exige l'unicité de la solution, alors que l'existence d'une solution n'est même pas garantie. Le grafcet Fig. 3.12 est un cas bien connu d'absence de situation suivante stable lorsque $A$ et $B$ sont simultanément vrais.

### 3.3.2.2 Recherche de LA situation suivante

Pour trouver la situation stable suivante, différents algorithmes ont été proposés dans la littérature. Beaucoup gèrent le changement d'état en faisant évoluer le GRAFCET jusqu'à la stabilité par une succession de "micro-pas" qui s'exécutent lors de "micro-instants". L'environnement observe un changement d'état résultant du cumul de tous ces micro-instants, sans avoir connaissance du détail des évolutions. Notons qu'une sémantique basée sur les micro-pas a été également

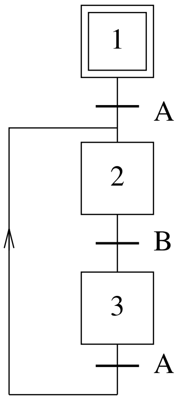



FIG. 3.12 - *Instabilité structurelle*

proposée pour les STATECHARTS [PS91]. En GRAFCET, la synthèse de ces travaux a conduit à la notion de *double échelle de temps* clairement définie dans la nouvelle norme [AFN93, BBC+92, FC92].

Nous qualifierons de “Maximal Parallèle Itéré” (noté MPI dans la suite) l'algorithme le plus utilisé (appelé “franchissement itéré” dans [DA89], par exemple). “Maximal Parallèle” fait référence à un mode de franchissement des Réseaux de Petri dans lequel on franchit à partir d'un marquage donné, un sous-ensemble maximal de transitions validées en tenant compte des conflits éventuels. Le terme “itéré” indique que ce type de franchissement est répété jusqu'à la stabilité. Cette propriété dérive directement des règles 2–4 du paragraphe 3.2.2.2. L'algorithme est rappelé ci-dessous.

### Algorithme de franchissement Maximal Parallèle Itéré (M.P.I)

```
début
  /* nouvel instant */
  figer les entrées;
  effectuer les actions impulsionnelles associées aux étapes actives;
  itérer
    évaluer les réceptivités des transitions validées;
    Tf ← ensemble des transitions franchissables;
    si Tf est non vide alors
      franchir toutes les transitions de Tf;
      effectuer les actions impulsionnelles associées aux étapes activées
    finsi
  jusqu'à Tf vide;
  effectuer les actions à niveau associées aux étapes actives
fin
```

Les *actions à niveau* ne sont effectuées que dans la situation stable suivante, elles ne présentent pas de difficultés particulières. Les actions impulsionnelles sont émises au fur et à mesure de la réaction pour que leurs effets puissent être pris en compte dans les réceptivités dès l'itération suivante (cas des compteurs internes).

**Remarques:**

- Cet algorithme n'est jamais mentionné dans la norme, qui d'ailleurs ne fait pas clairement allusion à la recherche de stabilité.
- Cet algorithme respecte les cinq règles de base.
- Une itération correspond à un micro-instant, la proposition faite dans [FC92] de passer à l'analyse non standard justifie que les itérations ne sont pas visibles de l'extérieur, mais n'apportent pas d'arguments sémantiques autres.
- Il est tout à fait acceptable de prendre en compte les actions impulsionnelles car celles-ci durent un temps nul et *l'ordre* des micro-instants caractérise la réaction globale.
- Il n'y a aucune garantie de sortir de la boucle (cas d'instabilité).
- Dans le cas où le calcul termine, le résultat est unique, ce qui assure le déterminisme.

Cette dernière remarque rassure la majorité des utilisateurs, mais elle est peut-être illusoire, comme nous le montrons dans le paragraphe suivant.

#### 3.3.2.3 Elément neutre

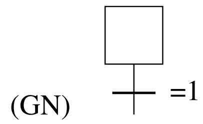



FIG. 3.13 - *Élément neutre*

Considérons le grafcet $(GN)$ (Fig. 3.13). Aucune action n'est associée à l'étape unique de $(GN)$. La réceptivité de l'unique transition est la condition toujours vraie $(= 1)$. Ce grafcet $(GN)$ devrait être *un élément neutre* pour la séquence. En d'autres termes, le comportement d'une séquence d'étapes et transitions ne devrait pas être modifié par l'ajout de $(GN)$ au sein de la séquence[5]. Avec cette définition, $(GN)$ correspondrait par exemple au "nothing" d'ESTEREL.

5. Ceci n'est bien sûr vérifié que pour le mode avec recherche de stabilité.

Montrons que l'algorithme MPI ne respecte pas l'élément neutre. Étudions le grafcet $G3$ de la figure 3.14 avec les réceptivités suivantes:

$$R1 = \texttt{vrai}, R10 = \texttt{vrai}, R21 = \texttt{faux}, R30 = \texttt{vrai}.$$

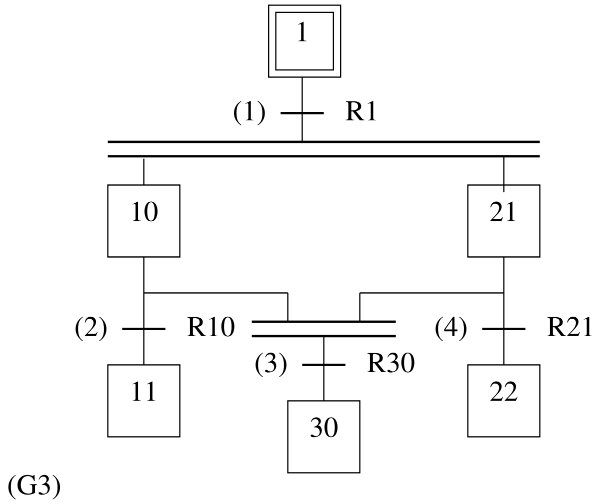



FIG. 3.14 - *Recherche de stabilité (1)*

L'application de l'algorithme MPI donne les situations suivantes : {1}, puis {10,21} par franchissement de {t1} au premier micro-instant, puis {11,30} par franchissements de {t2,t3} au second micro-instant. L'état stable suivant est donc {11,30}.

Ajoutons maintenant entre la transition t1 et l'étape 21 une instance du grafcet ($GN$) (étape 20, transition t5). Nous obtenons le grafcet $G4$ Fig. 3.15. Avec les mêmes réceptivités que précédemment, l'algorithme conduit à : {1}, puis {10,20} par franchissement de {t1} au premier micro-instant, puis {11,21} par franchissements de {t2,t5} au second micro-instant. Le nouvel état stable solution est donc {11,21}. Cette situation est différente de celle obtenue pour le grafcet sans l'élément neutre.

Face à cette situation deux attitudes sont possibles:

1. Conserver l'algorithme MPI et abandonner la notion d'élément "neutre" de comportement.

2. Renoncer à l'algorithme MPI pour ne pas hypothéquer la possibilité de définir une algèbre de processus sous-jacente au modèle.

Notre objectif étant de donner au GRAFCET une sémantique plus formelle, c'est la seconde solution qui a notre préférence. La non-prise en compte d'un

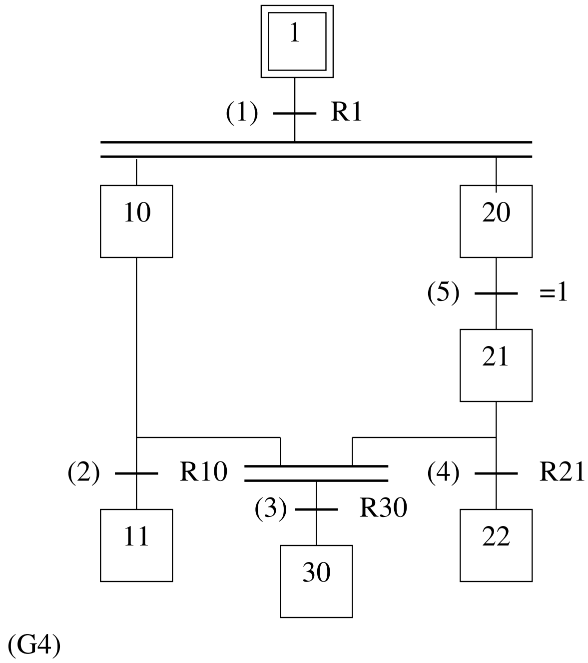



FIG. 3.15 - *Recherche de stabilité (2)*

élément neutre comportemental nous paraît rédhibitoire. Ce choix induit par contre deux inconvénients majeurs. En premier, il va falloir trouver une nouvelle façon de déterminer les évolutions. En second, nous risquons a priori de ne plus accepter des évolutions "classiquement admises".

#### 3.3.2.4 Variables et événements internes

Dans un grafcet, on associe à chaque étape "$k$" une variable booléenne $X_k$ dont la valeur est vraie si et seulement si l'étape $k$ est active, dans la situation considérée. Il s'agit d'une *variable interne*, que nous appellerons *variable d'étape* dans la suite. Certains auteurs [DA89] considèrent que "des variables en provenance de la partie commande sont des variables internes". Il s'agit là d'une utilisation du GRAFCET dans un contexte particulier (spécification du contrôle exercé sur une partie opérative, elle-même incluse dans la partie commande d'un système). Ceci sort de la définition stricte du modèle. Nous limitons donc les variables internes aux seules variables d'étapes.

Une variable interne peut être utilisée dans les réceptivités exactement comme une variable d'entrée. Or, dans les évolutions avec recherche de stabilité les variables d'entrée et d'étape ont des comportements différents. En effet, une variable d'entrée a une valeur figée pendant une réaction, alors qu'une variable d'étape peut être modifiée. Il est clair qu'utiliser une variable interne dans une réceptivité pose le problème délicat de connaître de façon non ambiguë la valeur de cette variable au cours de la réaction. L'algorithme MPI peut lever l'ambiguïté, mais nous avons déjà signalé les inconvénients de cet algorithme. Nous pensons qu'il est *incorrect de faire intervenir la valeur courante d'une variable d'étape dans une condition*. De façon générale, on s'interdit, dans les approches à états, d'observer les variables d'état pendant une transition.

Par contre, il est tout à fait acceptable de faire référence à la situation stable à partir de laquelle on effectue la réaction. Lorsqu'un $X_k$ est utilisé dans une réceptivité, nous considérons qu'il a la valeur "avant la réaction".

Cette dernière prise de position est lourde de conséquence : notre sémantique du $Xk$ est incompatible avec la *double échelle de temps* définie dans la norme. D'après celle-ci, le grafcet évolue en temps interne par une succession d'états *stables* dont seul le dernier sera connu en temps externe. Pour chaque micro-état, la valeur des variables internes $Xk$ est recalculée. Ces dernières peuvent alors intervenir dans les réceptivités pour la détermination du micro-état suivant. Notons que cette notion de temps interne s'oppose formellement à l'hypothèse de synchronicité forte d'ESTEREL ce qui entrave tout rapprochement avec ce langage synchrone. Pour ces raisons, nous *abandonnons* dans notre modèle ce principe de double échelle de temps.

Étant donné que les variables d'étape ne fournissent pas une information fiable au cours d'une réaction, comment prendre en compte les évolutions internes de façon dynamique? Nous suggérons d'avoir recours aux *événements internes*. En tant qu'événements ils peuvent exprimer le caractère fugace des "transitoires" lors d'une réaction, et contrairement aux variables qui peuvent prendre plusieurs valeurs dans un même instant, un événement est *soit présent, soit absent*. Ce point de vue est celui adopté dans la sémantique du langage ESTEREL; il est tout à fait fondamental dans la définition des sémantiques mathématiques du langage.

Nous introduisons pour chaque étape $k$, deux événements internes $\uparrow X_k$ et $\downarrow X_k$. La présence de l'événement $\uparrow X_k$ ($\downarrow X_k$, resp.) signifie que durant la réaction l'étape $k$ a été activée (désactivée, resp.). Notons que $\uparrow X_k$ et $\downarrow X_k$ sont indépendants : on peut avoir dans la même réaction : présence simultanée (activation puis désactivation ou bien désactivation puis activation de l'étape $k$), absence simultanée (l'étape $k$ n'est pas concernée par la réaction), présence d'un seul (changement d'état simple de l'étape $k$).

Nous arrivons ainsi à une conclusion diamétralement opposée à celle défendue habituellement. Il est formellement déconseillé [DA89, Recommandation 1.4] d'utiliser des événements internes dans les réceptivités. Ce point de vue est cohérent à partir du moment où les transitoires sont déterminées par l'algorithme MPI. Notre point de vue découle de l'hypothèse qu'un événement ne peut être que présent ou absent dans un instant. Il est tout aussi défendable. En fait les approches divergent car elles partent d'axiomes différents. De même le refus de la double échelle de temps pour formaliser les évolutions instantanées du système nous empêche a priori d'appeler notre modèle "*Grafcet*". C'est pourquoi, nous nous sommes ralliés à la suggestion de R. David de nommer différemment notre modèle. Le terme S-GRAFCET (Synchronous GRAFCET a été retenu.

Dans la suite, nous développons les conséquences de notre choix, non pas dans un souci d'originalité, mais parce qu'il nous permet, fort de l'expérience acquise avec les langages synchrones, d'aller plus loin dans la compréhension des comportements complexes. Les évolutions internes sont souvent source de tels comportements; les synchronisations en présence d'instabilité également.

#### 3.3.2.5 La non-simultanéité d'événements externes

Les normes du GRAFCET spécifient que deux signaux d'entrée non corrélés ne peuvent jamais changer d'état en même temps. Ceci découle de l'hypothèse d'un *temps continu* : la probabilité d'occurence simultanée de deux événements indépendants est nulle. La figure 3.16 qui suit, est donc un exemple de divergence en OU disjonctive.

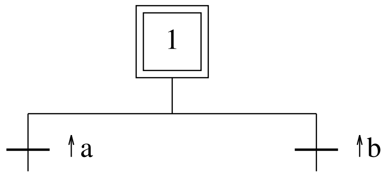



FIG. 3.16 - *Divergence en* OU *disjonctive*

Rien n'empêche les utilisateurs de ne considérer que des singletons en provenance de l'extérieur. Mais pour notre part, nous ne retiendrons pas cette hypothèse à cause des points suivants :

- Elle empèche la composition de grafcets telle que des événements internes de l'un soient utilisés simultanément comme événements d'entrée de l'autre.

- L'implantation effective doit assurer cette hypothèse même si le système de commande est physiquement trop lent pour séparer deux événements temporellement très proches. Sur l'exemple précédent, il serait nécessaire de rendre prioritaire un des deux événements. Si cette implantation est écrite en *langage* GRAFCET, ceci revient à modifier les réceptivités. De ce fait, le grafcet "implanté" risque d'être différent du grafcet de spécification. Notons enfin que le choix arbitraire d'une priorité sur les événements peut conduire au non-respect de la spécification initiale.

### 3.3.3 Ambiguïté liée aux actions impulsionnelles

La norme GRAFCET définit clairement la sémantique des actions continues : ces dernières sont émises dès que l'étape associée est *active* et *stable*[6] et durent le temps d'activation de l'étape. La stabilité de l'étape est essentielle pour empêcher l'émission d'actions continues infiniment courtes. La figure 3.17 montre un exemple de chronogramme d'émission d'actions continues.

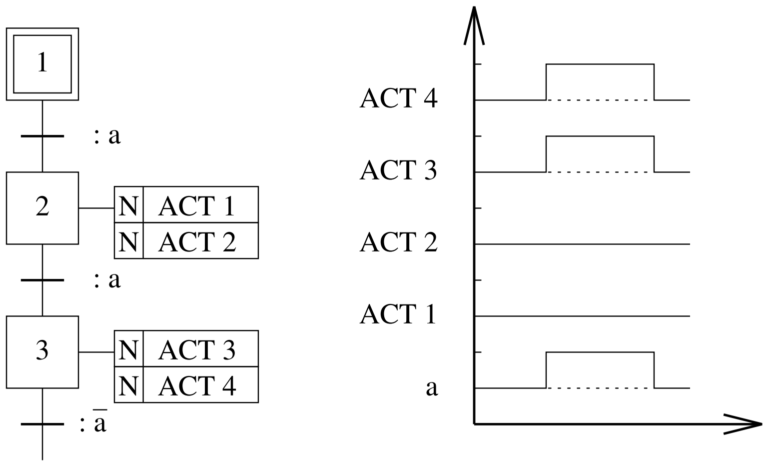



FIG. 3.17 - *Chronogramme d'actions à niveaux*

Par contre la norme et les ouvrages restent assez flous sur la sémantique des actions impulsionnelles et donnent naissance à deux interprétations possibles :

- La première considère une action impulsionnelle comme une *action Grafcet classique (temporisée avec une durée très petite)* [BBC+92]. Dans ce cas l'action n'est émise que si l'étape associée est active et stable comme le montre la figure 3.18.

- La seconde interprétation consiste à mettre en avant le côté impulsionnel de ce type d'action et à l'émettre dès que l'étape associée est activée (même transitoirement). En effet la norme considèrent ces actions comme des *ordres* de la partie commande. Il suffit d'associer l'action impulsionnelle à la présence de l'ordre d'activation $\uparrow X_k$ de l'étape $X_k$. La figure 3.19 montre comment les actions évoluent avec cette nouvelle sémantique. De même, des actions impulsionnelles pourraient être associées à la présence de l'ordre de désactivation comme le remarquent certains auteurs [Lep94].

Au niveau de ce mémoire, il a été décidé de retenir la seconde sémantique car elle a un pouvoir descriptif plus important. De plus cette sémantique est conforme

6. Avec l'interprétation ARS.

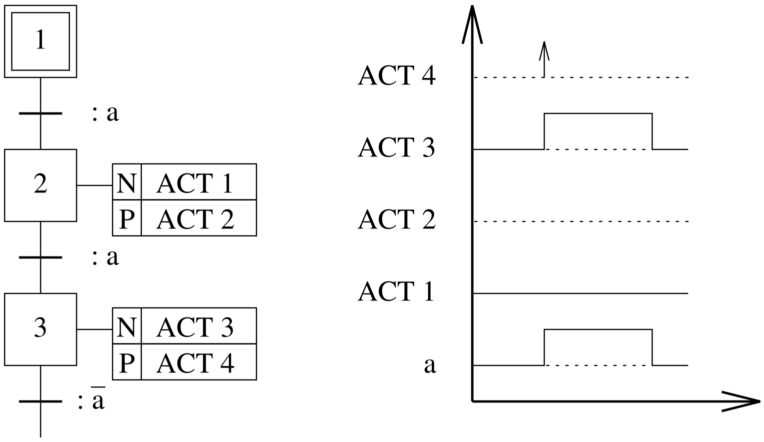



FIG. 3.18 - *Première sémantique des actions impulsionnelles*

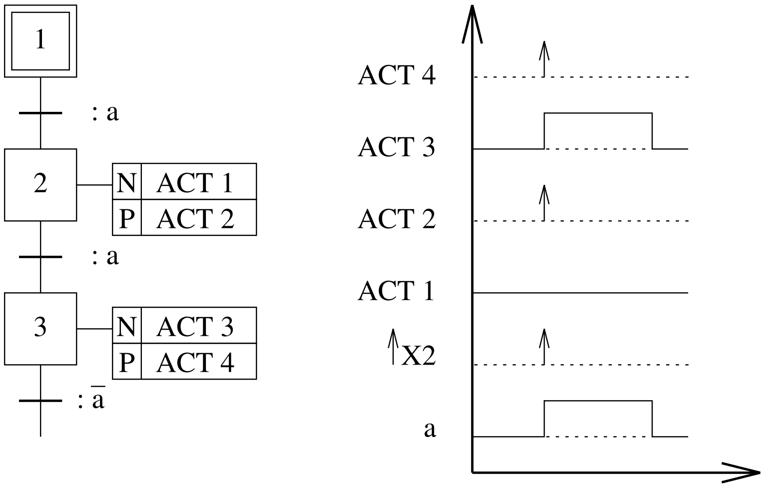



FIG. 3.19 - *Seconde sémantique des actions impulsionnelles*

au *output* ESTEREL ce qui facilite l'intégration du GRAFCET parmi les autres langages synchrones.

### 3.3.4 Frontières du modèle

Le GRAFCET possède deux frontières de description distinctes. La première appelée "modèle formel" formalise le comportement des sorties continues et événementielles en fonction des entrées également continues et événementielles. Il est important de noter que le temps physique n'intervient nullement et que le système réagit seulement sur occurrence d'événements (temps logique). Il sera donc qualifié de "réactif".

Cette première frontière est englobée par une seconde nommée "description normalisée". Cette dernière intègre en plus des "fonctions opératives associées" nécessaires pour prendre en compte des mémorisations et le temps physique 3.20. Ces fonctions opératives interviennent donc fortement dans le comportement global du système.

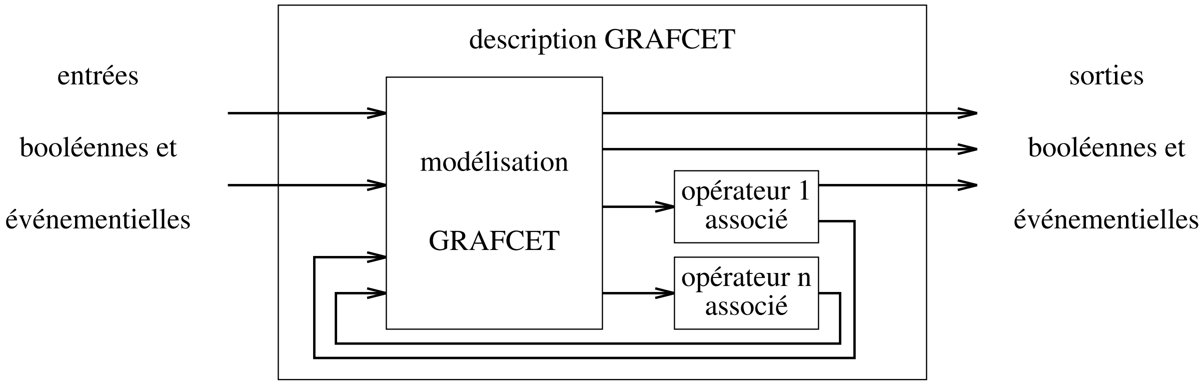



FIG. 3.20 - *Frontières du modèle GRAFCET*

Cette double frontière est très gênante pour intégrer le grafcet dans une plateforme synchrone commune. En effet les langages synchrones ne font pas cette distinction et savent implémenter complètement toutes descriptions comportementales. Il est donc nécessaire de définir une frontière unique pour le modèle GRAFCET qui soit la plus large possible et qui permettra l'intégration complète du modèle.

#### 3.3.4.1 Actions intervenant dans les évolutions et mémorisation

Le grafcet de la figure 3.21 est un exemple classique d'évolution conditionnée par une action. À chaque fois que l'événement $\uparrow A$ est présent, la variable $X$ est incrémentée. Lorsque cette variable atteint le seuil fixé, la transition $(t1)$ devient franchissable et conduit à l'activation de l'étape 2.

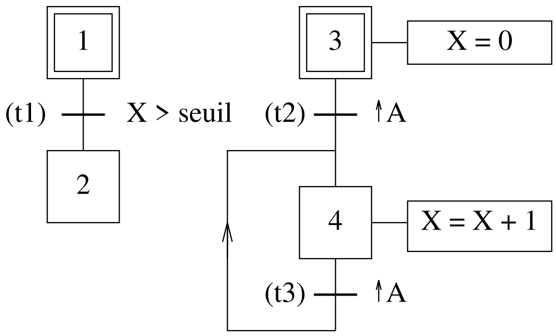



FIG. 3.21 - *Évolution conditionnée par une action*

Nous avons vu dans le chapitre 3.3.2.4 qu'il était incorrect d'observer les variables d'états pendant une transition. Dans notre exemple, $X$ est tout à fait assimilable à une variable d'état puisqu'elle représente le nombre de boucles réalisées. Sa valeur peut uniquement être prise en compte *avant* la réaction. De plus cette variable ne peut pas intervenir dans les évolutions instantanées car elle est associée à une action. Celle-ci est uniquement gérée quand le système s'est stabilisé. Pour ces deux raisons la gestion de la variable $X$ doit donc être laissée à un opérateur externe indépendant du grafcet (fig. 3.22) dont l'évolution sera prise en compte pour l'instant suivant.

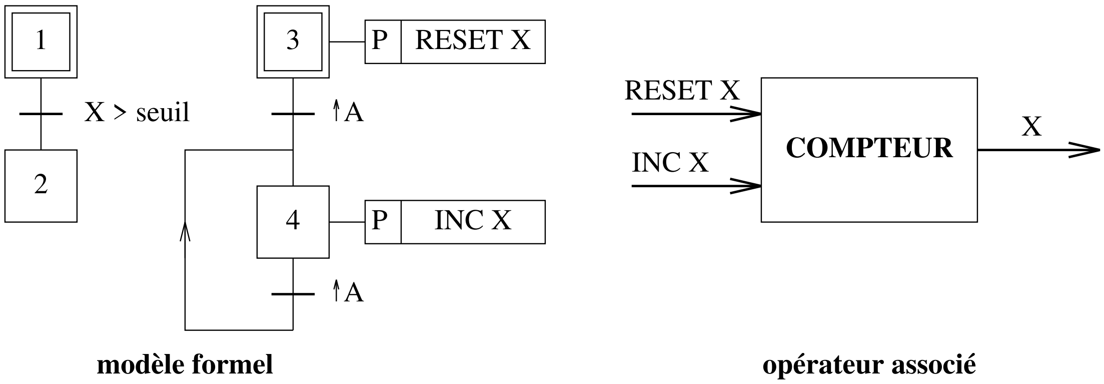



FIG. 3.22 - *Écriture formelle*

#### 3.3.4.2 Intégration des temporisations

Pour exprimer des contraintes de temps, le modèle GRAFCET intègre des fonctions opératives particulières appelées "opérateurs à retard". Ceux-ci ont été empruntés à la norme CEI 617-12 [IEC84] et expriment par une variable booléenne $S$, le retard apporté à une variable booléenne $E$ (fig. 3.23).

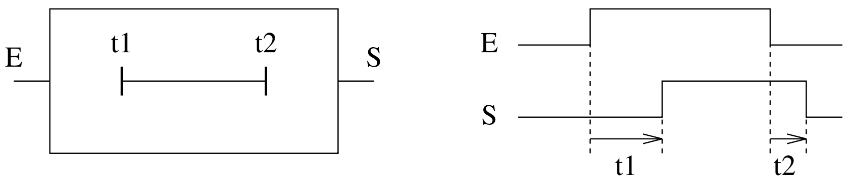



FIG. 3.23 - *Représentation et chronogramme d'un opérateur à retard*

Cet opérateur est noté "$t1/E/t2$" tel que :

- t1 soit le retard apporté au front montant de la variable E

- t2 soit le retard apporté au front descendant de la variable E

- E doit être présente pendant un temps supérieur ou égal à t1 pour que S prenne[7] l'état logique 1

Le modèle GRAFCET utilise cet opérateur au niveau des réceptivités et des actions sous la forme "$t/Xk$" qui signifie classiquement "$t/Xk/0$". En particulier les actions `D` et `L` présentées au paragraphe 3.2.1.4 sont considérées comme des actions conditionnelles (fig. 3.24).

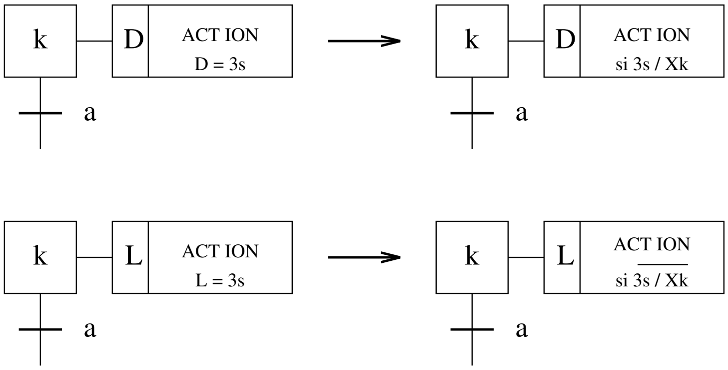



FIG. 3.24 - *Équivalence comportementale des actions "Delayed" et "Limited"*

L'usage de ces opérateurs induit certaines ambiguïtés:

- La notation $t/Xk$ est reconnue par la norme internationale CEI 848 [IEC88] qui cite explicitement la norme CEI 617-12. Par contre la norme française plus ancienne NF C 03 190 [AFN82] préconise la notation "t/Xk/q" qui a d'ailleurs un sens différent: la temporisation $t$ émet la valeur 1 dès qu'un délai $q$ s'est écoulé depuis la *dernière* activation de $k$. Chaque événement $\uparrow Xk$ provoque donc la réinitialisation de la temporisation même si cette dernière n'est pas terminée. Cette notion de préemption n'existe pas dans la norme internationale et peut changer totalement le comportement d'un grafcet. Notons de plus, que la nouvelle norme du GRAFCET (NF C 03 191 [AFN93]) laisse ce problème d'interprétation en suspend.

- La seconde ambiguïté est liée aux actions temporisées. Comment doit réagir une action $D$ maintenue un temps $t1$ si l'étape associée n'est active qu'un temps $t'$ inférieur à $t1$? Une réponse est de considérer l'équivalence comportementale de la figure 3.24. L'action doit être émise si l'étape est *stable* et si la condition est vraie (ici la temporisation). Le comportement de l'étape est donc prioritaire et l'action sera *préemptée* même si elle n'est pas terminée.

7. Curieusement la norme ne spécifie pas que E soit absente pendant un temps supérieur ou égal à t2 pour que S *reprenne* l'état logique 0!

– Au niveau des réceptivités, l'utilisation d'un opérateur à retard pose aussi des problèmes. En effet il transmet un résultat *booléen* calculé à partir de la valeur d'une *variable d'étape* $Xk$. Nous tombons ici encore dans les problèmes de validité de variables internes durant les évolutions présentées au 3.3.2.4. Il n'est donc pas souhaitable d'utiliser les opérateurs à retard pendant les évolutions.

#### 3.3.4.3 Conclusion relative aux frontières du modèle

Les fonctions opératives associées posent des problèmes d'intégration au sein d'un modèle comportemental unique. Seul le *modèle formel* GRAFCET sera intégré dans une plate-forme synchrone commune. Par contre les opérateurs associés peuvent quand même être considérés comme des tâches asynchrones *déclencheur d'instants*. Ils pourront donc être utilisés pour réaliser des preuves formelles de comportement.

## 3.4 Conclusion

L'association du GRAFCET et des langages synchrones tels que LUSTRE ou ESTEREL pour modéliser et implanter des systèmes de commande temps réel suppose que le modèle GRAFCET respecte l'hypothèse d'*atomicité* des réactions propres aux langages synchrones. Bien que synchrone, le GRAFCET ne repose pas sur les mêmes modèles mathématiques, si bien que des divergences d'interprétation peuvent exister. La plus importante porte sur la sémantique des variables internes.

Après avoir présenté les aspects syntaxiques du modèle et les grands concepts de base, nous sommes revenu dans ce chapitre sur la notion de réceptivité. Nous avons en particulier insisté sur la nécessité de distinguer sémantiquement et syntaxiquement *événements* et *conditions*. Les événements sont fugaces par nature et sont caractérisés par leur présence ou absence dans un instant. Leur occurrence provoque les évolutions du modèle alors que les conditions apparaissent comme des contraintes supplémentaires.

La *formalisation des évolutions internes* du GRAFCET induit une autre difficulté. Les évolutions sont généralement calculées par un algorithme de détermination de la situation suivante avec recherche de stabilité qui s'appuie sur la notion de *double échelle de temps* du modèle. Nous avons d'une part, illustré une faiblesse de ce type d'algorithme et d'autre part, réfuté la notion de *temps interne* qui est incompatible avec l'hypothèse de synchronicité forte d'ESTEREL.

Le dernier point susceptible d'empêcher une collaboration étroite avec les langages synchrones concerne la double frontière du modèle. Nous avons remarqué que le GRAFCET devait être restreint à son *modèle formel* pour permettre son intégration dans une plate-forme synchrone commune.

Ces difficultés et les convergences conceptuelles vont maintenant servir de base de travail pour formaliser complètement le modèle S-GRAFCET.

# Chapitre 4

# Formalisation des évolutions S-grafcet

## 4.1 Présentation

À partir de la syntaxe normalisée du GRAFCET et conformément aux problèmes soulevés au chapitre 3.3, nous avons étendu la notion de réceptivité. Nous faisons une distinction claire entre "événements" et "conditions". Les premiers sont utilisés pour *garder* la transition ou déclencher une nouvelle évolution. Les seconds expriment la condition de franchissement des transitions. Nous choisissons donc la notation "`<événement> : <condition>`" pour exprimer une réceptivité.

Lors d'une évolution, la partie conditionnelle est *figée* pour que le modèle reste compatible avec l'hypothèse de synchronicité forte des langages synchrones. Une telle condition peut représenter l'activité d'une étape $k$ (notation $X_k$). Dans ce cas elle sera valuée *avant* l'évolution. De même, un prédicat d'événement peut servir de réceptivité conditionnelle. Si l'événement est interne, la franchissabilité de la transition ne sera pas connue au début de l'instant.

### 4.1.1 Structure d'un S-grafcet

Un S-GRAFCET est composé d'un graphe (structure) et d'annotations textuelles (environnement et interprétation). La dynamique du modèle est définie par des règles d'évolution.

#### 4.1.1.1 Composantes statiques

**Structure :** La figure 4.1 montre un exemple de structure S-GRAFCET. Les règles syntaxiques sont identiques à celles du GRAFCET classique et largement commentées dans les normes [IEC88, AFN82, AFN93]. En fait, cette structure est un réseau [BRR87]. Les S-éléments sont appelés *étapes*, les T-éléments sont appelés *transitions*. La relation de flot est exprimée par un graphe orienté (par défaut du haut vers le bas). Dans notre exemple $G1$, nous obtenons les ensembles suivants :

$$\begin{aligned} S &= \{0, 1, 11, 12, 21, 22, 23\} \\ T &= \{t_1, t_2, t_3, t_{11}, t_{12}, t_{21}, t_{22}\} \\ F &= \{< 0\,,\, t_1 >, < t_1\,,\, 11 >, < t_1\,,\, 21 >, \cdots\} \end{aligned}$$

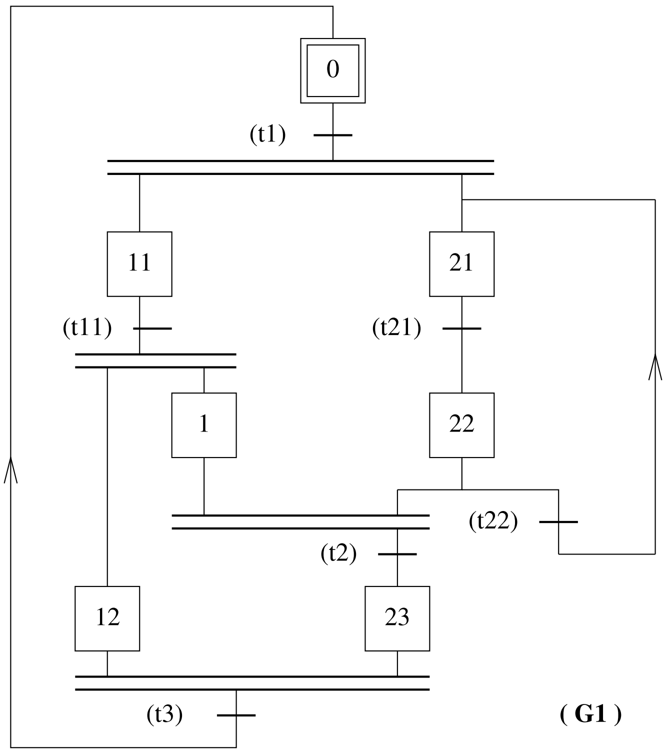



FIG. 4.1 - *Structure d'un sgrafcet.*

**Environnement :** Un S-grafcet reçoit des stimuli (événements d'entrées et données booléennes) et produit une réaction (événements de sortie, niveaux booléens). L'environnement est caractérisé par les ensembles : $\mathcal{E}_{\mathcal{I}}$ (resp. $\mathcal{E}_{\mathcal{O}}$) l'ensemble des entrées (resp. sorties) événementielles, $\mathcal{I}$ ($\mathcal{O}$) l'ensemble des entrées (resp. sorties) booléennes. Parmi toutes les actions normalisées et conformément aux remarques du chapitre 3.3.4, nous retiendrons les actions (continues (N), impulsionnelles (P), conditionnées continues (C) et conditionnées impulsionnelles (CP)). Les actions N sont considérées comme des actions C dont la condition est toujours vérifiée. La même remarque peut être faite pour les actions P et CP. Nous pouvons ainsi associer à $G1$ l'environnement suivant :

$$\begin{aligned} \mathcal{E}_{\mathcal{I}} &= \{lancement, fin\} \\ \mathcal{I} &= \{a, b, c\} \\ \mathcal{O} &= \{MONTER, TOURNER\} \\ \mathcal{E}_{\mathcal{O}} &= \emptyset \end{aligned}$$

**Interprétation** En GRAFCET, l'interprétation exprime le lien entre le modèle et son environnement. Ce terme est assez mal choisi et peut être sources d'ambiguïtés pour les informaticiens. En S-GRAFCET, nous utiliserons plutôt le terme "*Interface*".

Cette interface associe un ensemble d'actions à chaque étape et une réceptivité à chaque transition. Chaque action $A$ est elle-même associée à une ou plusieurs étapes $s$ et conditionnée par une expression booléenne $c$ qui peut être une tautologie. Pour les représenter, nous adoptons la convention "$A(s, c)$. La table 4.1 est un exemple d'interface pour $G1$.

TAB. 4.1 - *Interface de G1*

| Transition | Réceptivité | Étape | Ens. d'actions |
|---|---|---|---|
| $t_1$ | $lancement$ : | 0 | $\emptyset$ |
| $t_2$ | $: \neg c$ | 1 | $\emptyset$ |
| $t_3$ | $: 1$ | 11 | $\{MONTER(11, 1)\}$ |
| $t_{11}$ | $: a$ | 12 | $\{MONTER(12, 1)\}$ |
| $t_{21}$ | $fin$ : | 21 | $\{TOURNER(21, b)\}$ |
| $t_{22}$ | $: c$ | 22 | $\emptyset$ |
| | | 23 | $\emptyset$ |

Le s-grafcet se représente alors par la figure 4.2.

#### 4.1.1.2 Évolutions

Une étape peut être soit *active* soit *inactive*. L'état global d'un s-grafcet est défini par sa situation (ensemble des étapes actives). La situation est similaire au "cas" dans les systèmes Événement-Condition [Rei85]. Les actions continues associées aux étapes actives sont forcées (variable associée positionnée à 1). Lors d'une évolution, les actions impulsionnelles sont émises avec l'occurrence du front

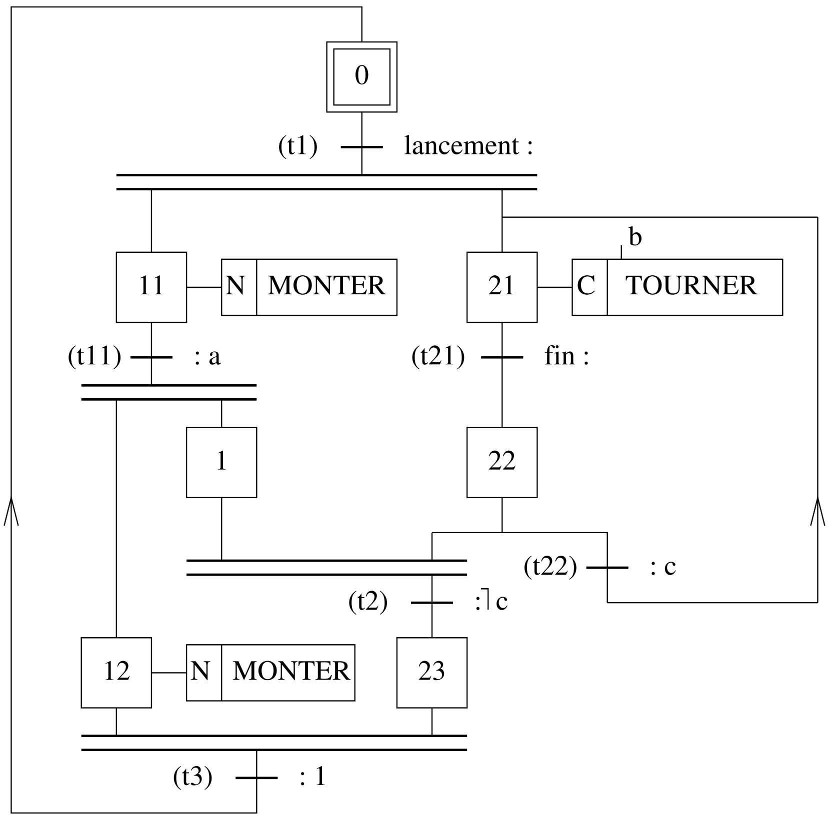



FIG. 4.2 - *S-grafcet muni de son interface.*

d'activation de l'étape. Tout événement externe déclenche une réaction du S-GRAFCET qui peut alors changer sa situation par franchissement de plusieurs transitions. Pour une situation $X$, une transition $t$ est *validée* ssi toutes les étapes amont sont actives.

La transition $t$ est franchissable ssi elle est validée, sa réceptivité est satisfaite et l'événement qui garde la transition est présent. Cet événement peut être formé de n'importe quelle combinaison d'événements élémentaires. Toute transition franchissable DOIT être franchie. Comme le franchissement d'un ensemble de transitions peut rendre instantanément franchissable un autre ensemble de transitions a priori, un nombre quelconque de transitions peut être franchi dans l'instant. Dans la suite, nous ne retenons que les réactions finies.

Une évolution d'une situation $X$ à une situation $X'$, pour un environnement $< \epsilon_{\mathcal{I}} \, , \, v_{\mathcal{I}} >$ où $\epsilon_{\mathcal{I}}$ est un sous-ensemble d'événements élémentaires et $v_{\mathcal{I}}$ une valuation des variables booléennes, induit le franchissement d'un sous-ensemble de transitions $T_f \subseteq T$ :

$$X \xrightarrow[T_f]{< \epsilon_{\mathcal{I}} \, , \, v_{\mathcal{I}} >} X'.$$

Considérons maintenant un exemple d'évolution de $G1$ (Table. 4.2). L'occurrence de *lancement* conduit au franchissement de $t_1$ et à l'activation de $\{11, 21\}$. Dans le même instant $t_{11}$ devient franchissable. Les deux franchissements conduisent donc à l'activation de $\{1, 12, 21\}$. Parmi les actions, seule $MONTER$ est émise car $TOURNER$ est conditionnée par $b$ dont la valuation est fausse et l'étape 12 est stable. Le second instant est déclenché par l'événement $\uparrow b$. Il ne conduit pas à une évolution de la situation, mais permet l'émission de $TOURNER$. Le troisième instant est déclenché par $fin$. Il conduit successivement aux situations $\{1, 12, 22\}$, $\{12, 23\}$, puis se stabilise en $\{0\}$. Remarquons la transition $t_3$ dont la réceptivité est toujours satisfaite. Elle permet la synchronisation de 12 et 23.

| instant | $X$ | $\epsilon_{\mathcal{I}}$ | valuation | $T_f$ | $X'$ | émission |
|---|---|---|---|---|---|---|
| 1 | $\{0\}$ | $\{lancement\}$ | $v(a) = 1$<br>$v(b) = 0$<br>$v(c) = 0$ | $\{t_1, t_{11}\}$ | $\{1, 12, 21\}$ | $\{MONTER\}$ |
| 2 | $\{1, 12, 21\}$ | $\emptyset$ | $v(a) = 1$<br>$v(b) = 1$<br>$v(c) = 0$ | $\emptyset$ | $\{1, 12, 21\}$ | $\{MONTER,$<br>$TOURNER\}$ |
| 3 | $\{1, 12, 21\}$ | $\{fin\}$ | $v(a) = 1$<br>$v(b) = 1$<br>$v(c) = 0$ | $\{t_2, t_{21}, t_3\}$ | $\{0\}$ | $\emptyset$ |

TAB. 4.2 - *Évolutions de $G1$*

## 4.2 Définitions préalables à la formalisation du S-Grafcet

L'ouvrage [Zah80] peut être consulté sur les expressions booléennes.

### 4.2.1 Expressions booléennes

**Notation 1** $\mathbb{B}$
*Posons $\mathbb{B} = \{0, 1\}$ et $\mathbb{B}_n$ le produit cartésien $\mathbb{B} \times \cdots \times \mathbb{B}$ (n facteurs).*

**Définition 3** ***Expressions booléennes***

Soient $x_1, \cdots, x_n : \mathbb{B}$ des variables booléennes. On appelle *expressions booléennes* sur $x_1, \cdots, x_n$, les expressions construites sur les symboles

$$0, 1, x_1, \cdots, x_n, \vee, \wedge, \neg, (, )$$

**Notation 2** *On note $EB(x_1, \cdots, x_n)$ une expression booléenne sur $x_1, \cdots, x_n$.*

**Définition inductive**

Les expressions sont définies inductivement par:

- $0, 1, x_1, \cdots, x_n$ sont des EB$(x_1, \cdots, x_n)$,
- Si $F$ et $G$ sont des EB$(x_1, \cdots, x_n)$, l'expression $(F \vee G)$ est une EB$(x_1, \cdots, x_n)$,
- Si $F$ et $G$ sont des EB$(x_1, \cdots, x_n)$, l'expression $(F \wedge G)$ est une EB$(x_1, \cdots, x_n)$,
- Si $F$ est une EB$(x_1, \cdots, x_n)$, l'expression $\neg F$ est une EB$(x_1, \cdots, x_n)$.

**Définition 4** ***Sous-ensemble de $\mathbb{B}_n$ représenté par une expression booléenne***

À tout $F$ : EB$(x_1, \cdots, x_n)$, on associe un sous-ensemble de $\mathbb{B}_n$, noté $[\![F]\!]_{(x_1,\cdots,x_n)}$ et appelé le sous-ensemble de $\mathbb{B}_n$ représenté par $F$ selon la liste $x_1, \cdots, x_n$. Généralement, l'indice $(x_1, \cdots, x_n)$ est sous-entendu, on écrit simplement $[\![F]\!]$. Si on considère l'espace de $n$ dimensions associé à $\mathbb{B}_n$, $[\![F]\!]$ est l'ensemble des points de cet espace qui rendent EB$(x_1, \cdots, x_n)$ vraie.

**Calcul inductif :** $[\![F]\!]_{(x_1,\cdots,x_n)}$ est calculé de manière inductive par :

- $[\![0]\!] = \emptyset$
- $[\![1]\!] = \mathbb{B}_n$
- $[\![x_k]\!] = \{\xi \in \mathbb{B}_n | \xi_k = 1\}$
- $[\![(F \vee G)]\!] = [\![F]\!] \cup [\![G]\!]$
- $[\![(F \wedge G)]\!] = [\![F]\!] \cap [\![G]\!]$
- $[\![\neg F]\!] = \mathbb{B}_n \setminus [\![F]\!]$

## 4.3 Définition formelle du modèle S-grafcet

Nous allons maintenant définir la sémantique du modèle S-GRAFCET de manière plus formelle.

**Définition 5** *Un* S-GRAFCET $S_G$ *est défini par un triplet* $< N , Env , Int >$ *où*

- $N = < S\,,\, T\,,\, F >$ est un réseau:
  - $S$ ensemble d'étapes,
  - $T$ ensemble de transitions,
  - $F \subseteq S \times T \cup T \times S$ relation de flot,
  - $S \cap T = \emptyset$
- $Env = \langle \mathcal{I}, \mathcal{E}_I, \mathcal{O}, \mathcal{E}_O \rangle$ est un environnement:
  - $\mathcal{I}$ ensemble de variables booléennes d'entrée,
  - $\mathcal{O}$ ensemble de variables booléennes de sortie,
  - $\mathcal{E}_\mathcal{I}$ ensemble d'événements élémentaires d'entrée,
  - $\mathcal{E}_\mathcal{O}$ ensemble d'événements élémentaires de sortie

  Ces ensembles sont disjoints deux à deux.
- $Int = < R\,,\, A >$ est une interface:
  - $R$ est une fonction qui permet d'associer une réceptivité à chaque transition de $T$.
    $\forall t \in T : R(t)$ est donc la réceptivité associée à $t$, elle est constituée d'une paire $< G\,,\, C >$
    - $R(t).G$ est la garde,
    - $R(t).C$ est la condition.
  - $A$ est une fonction qui permet d'associer zero, une ou plusieurs actions à chaque étape.
    $\forall s \in S$, $\eta \in \{cont, impuls\}$, $c$ une expression booléenne:
    $A(s, \eta, c)$ définit les actions de type continue ou impulsionnelle, conditionnées par $c$ associées à $s$.

  Pour préciser $R$ et $A$, nous avons besoin d'ensembles auxiliaires:
  - $\mathcal{X} = \{X_s |\ s \in S\}$ ensemble des variables d'étape,
  - $\hat{\mathcal{X}} = \{\uparrow X_s |\ s \in S\}$ ensemble des événements élémentaires d'activation,
  - $\check{\mathcal{X}} = \{\downarrow X_s |\ s \in S\}$ ensemble des événements élémentaires de désactivation,
  - $\mathcal{E} = \mathcal{E}_\mathcal{I} \cup \hat{\mathcal{X}} \cup \check{\mathcal{X}}$ ensemble des événements élémentaires significatifs[1],
  - $\mathcal{P} = \{P_e |\ e \in \mathcal{E}\}$ ensemble des variables propositionnelles de présence.

1. L'adjectif significatif rappelle que ces événements sont significatifs pour le calcul des évolutions. Par opposition aux événements élémentaires de sortie qui dans notre modèle ne participent pas au calcul d'une réaction.

On ordonne arbitrairement les éléments de $\mathcal{I} \cup \mathcal{X} \cup \mathcal{P}$. Soit $\mathcal{L}$ la liste correspondante, $L$ le nombre d'éléments. On a alors :

- $R : T \rightarrow 2^{\mathcal{E}} \times \mathrm{EB}(\mathcal{L})$,
- $A : S \rightarrow (2^{\mathcal{O}} \times \{cont, impuls\} \times \mathrm{EB}(\mathcal{L}))$

où $\mathrm{EB}(\mathcal{L})$ désigne une expression booléenne d'éléments de $\mathcal{L}$.

**Remarque :** Par rapport aux définitions classiques, nous distinguons clairement événements et conditions. Au niveau des réceptivités, nous utilisons la syntaxe concrète, présentée au début de ce chapitre (§ 4.1).

Certains ensembles ne sont pas classiques ($\check{\mathcal{X}}, \hat{\mathcal{X}}$), ils ont été introduits car ils jouent un rôle central dans l'étude des évolutions transitoires. Quant à l'ensemble $\mathcal{P}$ il résulte de la nécessité de transformer des présences d'événements en valeurs booléennes.

**Notation 3** *Nous utilisons la notion classique des réseaux pour désigner les pre et post-sets:*

$$
\begin{aligned}
\forall x \in S \cup T \quad : \quad {}^{\circ}x &= \{y \in S \cup T \mid \langle y\,,\, x \rangle \in F\} && (4.1)\\
x^{\circ} &= \{y \in S \cup T \mid \langle x\,,\, y \rangle \in F\} && (4.2)\\
\forall X \subseteq S \cup T \quad : \quad {}^{\circ}X &= \bigcup\{{}^{\circ}x \mid x \in X\} && (4.3)\\
X^{\circ} &= \bigcup\{x^{\circ} \mid x \in X\} && (4.4)
\end{aligned}
$$

**Notation 4** *Nous introduisons également des fonctions de* $2^T \rightarrow 2^S$*:*

$$
\begin{aligned}
\forall T' \subseteq T \quad : \quad & \mathrm{Pre}\ T' = {}^{\circ}T' && (4.5)\\
& \mathrm{Post}\ T' = T'^{\circ} && (4.6)\\
& \widetilde{\mathrm{Pre}}\ T' = S \setminus \mathrm{Pre}\ (T \setminus T') && (4.7)\\
& \widetilde{\mathrm{Post}}\ T' = S \setminus \mathrm{Post}\ (T \setminus T') && (4.8)
\end{aligned}
$$

**Remarque:** Ces sous-ensembles peuvent également s'écrire:

$$
\begin{aligned}
\mathrm{Pre}\ T' &= \{s \in S \mid s^{\circ} \cap T' \neq \emptyset\} && (4.9)\\
\mathrm{Post}\ T' &= \{s \in S \mid {}^{\circ}s \cap T' \neq \emptyset\} && (4.10)\\
\widetilde{\mathrm{Pre}}\ T' &= \{s \in S \mid s^{\circ} \subseteq T'\} && (4.11)\\
\widetilde{\mathrm{Post}}\ T' &= \{s \in S \mid {}^{\circ}s \subseteq T'\} && (4.12)
\end{aligned}
$$

$\widetilde{\mathrm{Pre}}\ T'$ est l'ensemble des étapes puits ou telles que tous leurs successeurs sont dans $T'$.

## 4.4 Règles d'évolution du modèle

### 4.4.1 Généralités

Dans les définitions qui suivent on suppose donné un s-grafcet $\mathcal{S}_G$ tel que:

$$\mathcal{S}_G = << S\,, T\,, F >, \langle \mathcal{I}, \mathcal{E}_I, \mathcal{O}, \mathcal{E}_O \rangle\,, < R\,, A >>.$$

**Définition 6** *Une **situation** est définie par un sous-ensemble d'étapes. La situation courante $X$ est telle que:*

$$X \;=\; \{s \in S|\ s \text{ est active}\} \tag{4.13}$$

**Définition 7** *Un* événement *est un sous-ensemble de $\mathcal{E}$ (ensemble des événements élémentaires). L'événement courant appliqué à $\mathcal{S}_G$ est noté $e$.*

$e_{\mathcal{I}}$ est l'événement courant d'entrée, c'est la restriction de $e$ à $\mathcal{E}_{\mathcal{I}}$

**Définition 8** *On considère des* valuations *de $\mathcal{I} \cup \mathcal{X} \cup \mathcal{P} \rightarrow \{0, 1\}$. La valuation courante appliquée à $\mathcal{S}_G$ est notée $v$.*

$v_{\mathcal{I}}$ est la valuation courante des entrées, c'est la restriction de $v$ à $\mathcal{I}$. $v_{\mathcal{X}}$ est la valuation courante des variables d'étapes, c'est la restriction de $v$ à $\mathcal{X}$, on a $v(X_s) = 1 \Longleftrightarrow s \in X$. $v_{\mathcal{P}}$ est la valuation courante des variables propositionnelles de présence, c'est la restriction de $v$ à $\mathcal{P}$, on a $\forall x \in \mathcal{E} : v(P_x) = (x \in e)$.

**Définition 9** *On appelle **instant** toute date logique où $\mathcal{S}_G$ doit tenter d'évoluer. Cet instant est déclenché par n'importe quelle occurrence d'événements élémentaires externes (y compris sur une demande implicite de l'environnement qui est elle-même considérée comme un événement élémentaire).*

Nous associons à chaque *instant*, l'occurrence d'un événement élémentaire appelé tick et noté $\top$.

**Problème des évolutions d'un s-grafcet:**

> "Étant donnés $\mathcal{S}_G$, une situation $X$, un événement d'entrée $e_{\mathcal{I}}$ et une valuation des entrées $v_{\mathcal{I}}$, trouver LA situation $X'$ suivante de $\mathcal{S}_G$".

Pour résoudre ce problème on introduit les notions de validation et de franchissabilité des transitions. La validation d'une transition $t$ ne fait intervenir que la structure du s-grafcet ( $^{\circ}t$) et la situation courante ($X$). Le franchissement fait intervenir, en plus, l'interprétation du s-grafcet ($R(t)$) et l'état de son environnement ($< e_{\mathcal{I}}\,,\, v_{\mathcal{I}} >$).

**Définition 10** *Une transition $t$ de $\mathcal{S}_G$ est* validée *pour une situation $X$* **ssi** $^{\circ}t \subseteq X$.

**Définition 11** *Une transition $t$ de $\mathcal{S}_G$ est* franchissable *pour une situation $X \subseteq S$, un événement $e \subseteq \mathcal{E}$ et une valuation $v : (\mathcal{I} \cup \mathcal{X} \cup \mathcal{P}) \rightarrow \{0, 1\}$* **ssi**

1. *$t$ est validée pour $X$,*
2. *$R(t).G \subseteq e$,*
3. *$R(t).C(v) = 1$*

**Commentaires :** Cette définition n'est pas directement applicable puisqu'elle utilise $e$ et $v$ qui sont dérivés de $X, e_{\mathcal{I}}, v_{\mathcal{I}}$. La façon de traiter les transitoires va être déterminante pour le calcul de $e$ et $v$. Ce point sera étudié ci-après.

L'algorithme d'évolution retenu devra, bien entendu, évoluer de la même façon qu'un grafcet classique dans les cas simples (en particulier en l'absence d'évolutions internes).

**Notation :** Une transition $t \in T$ est gardée ssi $R(t).G \neq \emptyset$. On partitionne $T$ en $\{T_g, T_{ng}\}$ où $T_g$ est l'ensemble des transitions gardées et $T_{ng}$ celui des transitions non gardées.

La condition (2) montre qu'un événement (expression d'événements élémentaires) est nécessaire pour déclencher le franchissement d'une transition. Pour les transitions non gardées, cette condition est trivialement satisfaite.

Dans le cas où la transition est gardée, nous isolons dans la réceptivité événementielle, le côté déclencheur du prédicat de présence. Nous pouvons écrire :

$< G, C > = < \top, C \wedge P_G > = < \top, C' >$

où $\top$ est l'événement toujours présent (tick) précédemment défini.

Dans ce cas la condition (2) revient à attendre l'occurrence de n'importe quel événement. La condition (3) est par contre étendue aux prédicats des événements. Pour simplifier les équations d'évolution, nous considérons dans la suite de ce chapitre que les réceptivités gardées sont systématiquement réécrites en syntaxe abstraite sous la forme $< \top, C >$ qui sera abusivement noté : $< C >$ avec $t \in T_g$.

### 4.4.2 Formalisation du problème des évolutions

L'idée est de trouver $T_f$ le plus grand ensemble de transitions franchissables dans l'instant, pour $X, e_{\mathcal{I}}, v_{\mathcal{I}}$ donnés. Pour cela, on est amené à construire un ensemble auxiliaire $\mathcal{E}_p$: ensemble des événements présents dans l'instant.

**Définition 12** *Soit $\theta$ un instant. $X^\theta$ est la situation d'un s-grafcet $\mathcal{S}_G$ à cet instant. $< e_{\mathcal{I}}\,,\, v_{\mathcal{I}} >$ est l'état de l'environnement au même instant. La **réaction** de $\mathcal{S}_G$ à l'instant $\theta$ conduit à la situation $X^{\theta+1}$ par le franchissement des transitions $T_f \subseteq T$.*

On note cette réaction:

$$X^\theta \xrightarrow[T_f]{< e_{\mathcal{I}}\,,\, v_{\mathcal{I}} >} X^{\theta+1}$$

Initialement on ne connaît avec certitude que $e_{\mathcal{I}} \subseteq \mathcal{E}_p$ pour les événements, et $v_{\mathcal{I}}, v_{\mathcal{X}}$ pour les valuations. Les fonctions de conditions $R(t).C$ peuvent donc être partiellement connues. Les non-déterminations sont levées au fur et à mesure qu'augmente notre connaissance sur $\mathcal{P}$ par l'intermédiaire de $\mathcal{E}_p$:

$$\forall e \in \mathcal{E} \quad : \quad e \in \mathcal{E}_p \Longrightarrow v(P_e) = 1 \tag{4.14}$$

Ceci permet de préciser $v_{\mathcal{P}}$ (restriction de $v$ à $\mathcal{P}$) et, par là même, de calculer les conditions. De façon analogue, notre connaissance sur la présence ou l'absence d'événements élémentaires s'enrichit avec celle de $T_f$ :

$$\forall s \in \mathcal{S} \quad : \quad s \in T_f{}^{\circ} \Longrightarrow \uparrow\! X_s \in \mathcal{E}_p, \tag{4.15}$$

$$\forall s \in \mathcal{S} \quad : \quad s \in {}^{\circ}T_f \Longrightarrow \downarrow\! X_s \in \mathcal{E}_p. \tag{4.16}$$

**Remarque :** En GRAFCET, une étape $s$ activée et désactivé reste active. Si cette conjonction a lieu au même instant interne, il est tout à fait acceptable de ne pas émettre $\uparrow\! X_s$ et $\downarrow\! X_s$. En S-GRAFCET ces émissions d'événement sont obligatoires car ils sont susceptibles de participer à la réaction globale.

### 4.4.3 Formalisation des sorties

La connaissance complète de la réaction est suffisante pour définir les ensembles $o \subseteq \mathcal{O}$ et $e_o \subseteq \mathcal{E}_{\mathcal{O}}$. En effet $o$ regroupe les variables booléennes associées aux actions continues `N` et `C`, il ne dépend que de $X^{\theta+1}$ et de $v$. L'ensemble $e_o$ regroupe les événements associés aux actions impulsionnelles `P` et `CP`, il est déduit de $\mathcal{E}_p$ et dépend de $v$.

C'est pourquoi, nous nous focaliserons principalement sur le problème des évolutions dans la suite de ce chapitre.

**Définition 13** *On définit les ensembles* $o \subseteq \mathcal{O}$ *et* $e_o \subseteq \mathcal{E}_{\mathcal{O}}$ *tels que :*

$$e_o = \bigcup_{s \in S | \uparrow \mathcal{X}_s \in \mathcal{E}_p} A(s, impuls, EB_s(\mathcal{L})) \tag{4.17}$$

$$o = \bigcup_{s \in X_{\theta+1}} A(s, cont, EB_s(\mathcal{L})) \tag{4.18}$$

### 4.4.4 Caractérisation de la solution par point fixe

#### 4.4.4.1 Équations d'évolution

Les transitions franchissables sont caractérisées par une conjonction de conditions à satisfaire. Une transition est franchissable ssi :

1. elle est validée,
2. il y a occurrence d'au moins un événement,
3. la partie condition de sa réceptivité est vraie. Elle intègre en particulier les prédicats des événements qui interviennent dans la garde.

La définition de la validation dépend de l'interprétation retenue pour les évolutions:

- *Sans recherche de stabilité :* il faut que toutes les étapes amont soient actives,
- *Avec recherche de stabilité :* on peut avoir soit toutes les étapes amont actives, soit elles le deviennent lors des évolutions internes. Toutefois dans ce dernier cas la transition doit être non gardée.

Ces conditions peuvent être formulées ainsi:

**Définition 14** *On définit les fonctions* $f, f_1, f_2, f_3, g, h$ *paramétrées par la situation* $X$ *telles que:*

$$\begin{aligned} f, f_1, f_2, f_3 &: 2^T \times 2^{\mathcal{E}} \times 2^{\mathbb{B}_L} \to 2^T \\ g &: 2^T \times 2^{\mathcal{E}} \times 2^{\mathbb{B}_L} \to 2^{\mathcal{E}} \\ h &: 2^T \times 2^{\mathcal{E}} \times 2^{\mathbb{B}_L} \to 2^{\mathbb{B}_L} \end{aligned}$$

$\forall T' \subseteq T, \forall \mathcal{E}' \subseteq \mathcal{E}, \forall [\![V]\!] \subseteq \mathbb{B}_L$ :

$$f \quad = \quad (f_1 \cup f_2) \cap f_3 \quad \text{tel que :} \tag{4.19}$$

$f_1$ est l'ensemble des transitions validées pour la situation $X$ courante :

$$f_1(T', \mathcal{E}', [\![V]\!]) \quad = \quad \{t \in T |\ (t \in T_g) \wedge (\ ^{\circ}t \subseteq X)\} \tag{4.20}$$

$f_2$ est l'ensemble des transitions non gardées qui deviennent validées lors des évolutions internes :

$$f_2(T', \mathcal{E}', [\![V]\!]) \quad = \quad \{t \in T |\ (t \notin T_g) \wedge (\ ^{\circ}t \setminus X \subseteq \{s | \uparrow X_s \in \mathcal{E}'\})\} \tag{4.21}$$

$f_3$ est l'ensemble des transitions dont la condition est vraie :

$$f_3(T', \mathcal{E}', [\![V]\!]) \quad = \quad \{t \in T |\ [\![V]\!] \subseteq [\![R(t).C]\!]\} \tag{4.22}$$

$g$ est l'ensemble des événements présents :

$$g(T', \mathcal{E}', [\![V]\!]) \quad = \quad e_{\mathcal{I}} \cup \{\downarrow X_s |\ s \in \ ^{\circ}T'\} \cup \{\uparrow X_s |\ s \in T'^{\circ}\} \tag{4.23}$$

$h$ définit le singleton de $\mathbb{B}_L$ qui représente l'état de toutes les variables booléennes :

$$\begin{aligned} h(T', \mathcal{E}', [\![V]\!]) \quad = \quad & \left( \bigcap_{(i \in \mathcal{I} \cup \mathcal{X}) \wedge (v(i)=1)} [\![i]\!] \right) \cap \left( \bigcap_{(i \in \mathcal{I} \cup \mathcal{X}) \wedge (v(i)=0)} [\![\neg i]\!] \right) \cdots \\ & \cap \left( \bigcap_{(P_e \in \mathcal{P}) \wedge (e \in \mathcal{E}')} [\![P_e]\!] \right) \cap \left( \bigcap_{(P_e \in \mathcal{P}) \wedge (e \notin \mathcal{E}')} [\![\neg P_e]\!] \right) \end{aligned} \tag{4.24}$$

**Commentaires :** L'ensemble $f_1$ est constant car il ne dépend que de la topologie du graphe et de $X$. $f_2$ ne dépend explicitement que de $\mathcal{E}'$. L'ensemble de transitions $f_3$ (condition vraie) ne dépend explicitement que de $[\![V]\!]$. L'ensemble $g$ des événements présents ne dépend explicitement que de $T'$.

L'ensemble des transitions franchissables est défini par $f(T', \mathcal{E}', [\![V]\!])$. En fait, nous cherchons à résoudre le système d'équations simultanées $\Sigma_1$:

$$T' \quad = \quad f(T', \mathcal{E}', [\![V]\!]) \tag{4.25}$$
$$\mathcal{E}' \quad = \quad g(T', \mathcal{E}', [\![V]\!]) \tag{4.26}$$
$$[\![V]\!] \quad = \quad h(T', \mathcal{E}', [\![V]\!]) \tag{4.27}$$

Nous allons considérer deux cas selon que les conditions intervenant dans les réceptivités font intervenir ou non les événements internes. Lorsque les conditions sont indépendantes des événements internes, les conditions peuvent être évaluées statiquement au début de l'instant. Il en résulte une simplification de la détermination de la situation suivante.

#### 4.4.4.2 Cas des conditions statiques

Dans de nombreux s-grafcets, les conditions dans les réceptivités ne dépendent pas des prédicats de présence des événements internes:

**Lemme 1** *Si les événements internes n'interviennent pas au niveau des réceptivités, alors $[\![V]\!]$ ne dépend que de l'environnement et de $X$.*

En effet, $\mathcal{P} \setminus \{P_z | \; z \in (\hat{\mathcal{X}} \cup \check{\mathcal{X}})\} = \{P_e | \; e \in \mathcal{E_I}\}$. Soit Varin $= \mathcal{I} \cup \mathcal{X} \cup \{P_e | e \in \mathcal{E_I}\}$. Lorsque la condition est satisfaite seules les variables de Varin agissent sur $[\![V]\!]$. Notons $\text{Varin}_1$ ($\text{Varin}_0$, resp.) l'ensemble des variables booléennes (utiles) à 1 (à 0, resp.).

$$\begin{aligned}
\text{Varin}_0 &= \{i \in \mathcal{I} | \; v(i) = 0\} \cup \{X_s \in \mathcal{X} | \; s \text{ est inactive}\} \\
&\quad \cup \{P_e \in \mathcal{P} | \; e \in \mathcal{E_I} \wedge e \text{ est absent}\} \qquad (4.28) \\
\text{Varin}_1 &= \{i \in \mathcal{I} | \; v(i) = 1\} \cup \{X_s \in \mathcal{X} | \; s \text{ est active}\} \\
&\quad \cup \{P_e \in \mathcal{P} | \; e \in \mathcal{E_I} \wedge e \text{ est présent}\} \qquad (4.29)
\end{aligned}$$

On a alors:

$$[\![V]\!] = \left( \bigcap_{z \in \text{Varin}_1} [\![z]\!] \right) \cap \left( \bigcap_{z \in \text{Varin}_0} [\![\neg z]\!] \right) \qquad (4.30)$$

**Solution :** Considérons chaque équation de $\Sigma_1$:

- $f_1$ est constante,
- $f_2$ est monotone[2] par rapport à $\mathcal{E}'$,
- $f_3$ est anti-monotone[3] par rapport à $[\![V]\!]$,
- $g$ est monotone par rapport à $T'$,
- $h$ est constante sous l'hypothèse de conditions statiques

2. $f : 2^E \to 2^E$ est monotone si $\forall X, X' \subseteq E : X \subseteq X' \Longrightarrow f(X) \subseteq f(X')$. Cette propriété est trivialement vérifiée si $f$ est indépendante de X.

3. $f : 2^E \to 2^E$ est anti-monotone si $\forall X, X' \subseteq E : X \subseteq X' \Longrightarrow f(X') \subseteq f(X)$.

On en déduit que $f_3$ est constante et $f$ est monotone. Le système $\Sigma_1$ admet alors un plus grand et un plus petit point fixe vectoriel (voir [Arn92, Chap.6.4]). De plus les ensembles considérés sont finis. On peut obtenir le plus petit point fixe en itérant à partir de l'ensemble vide et le plus grand point fixe en itérant depuis l'ensemble lui-même.

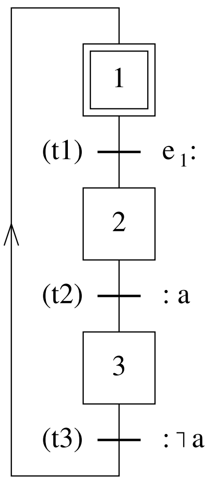



FIG. 4.3 - *S-grafcet G2*

**Exemple G2:** Soit le s-grafcet $G2$ (Fig. 4.3).

$$\begin{array}{rcl} S &=& \{1,2,3\} \\ T &=& \{t_1,t_2,t_3\} \\ \mathcal{I} &=& \{a\} \\ \mathcal{E}_{\mathcal{I}} &=& \{e_1\} \end{array}$$

Nous considérons l'interprétation :

$R(t_1).G = \{e_1\}, R(t_1).C = 1; R(t_2).G = \emptyset, R(t_2).C = a;$
$R(t_3).G = \emptyset, R(t_3).C = \neg a.$

La transition $t1$ se réécrit en syntaxe abstraite : $R(t_1).G = \{\top\}, R(t_1).C = P_{e_1};$

Supposons $e_{\mathcal{I}} = \{e_1\}$ et $v(a) = 1$, les fonctions constantes sont:

$$\begin{array}{rcl} f_1 &=& \{t_1\} \\ f_3 &=& \{t_1,t_2\} \\ [\![V]\!] &=& [\![a \wedge X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge P_{e_1}]\!] \end{array}$$

**plus petit point fixe :** Initialement $T' = \mathcal{E}' = \emptyset$. La Table 4.3 résume les calculs. À chaque pas de calcul, $f1$, $f2$, $f3$, $f4$ et $\mathcal{E}'$ sont recalculés avec les valeurs trouvées à l'itération précédente (point fixe vectoriel).

| itération | $(f_1 \cup f_2) \cap f_3 \to T'$ | $\mathcal{E}'$ |
|---|---|---|
| 1 | $(\{t_1\} \cup \emptyset) \cap \{t_1, t_2\} \to \{t_1\}$ | $\{e_1\}$ |
| 2 | $(\{t_1\} \cup \emptyset) \cap \{t_1, t_2\} \to \{t_1\}$ | $\{e_1, \downarrow X_1, \uparrow X_2\}$ |
| 3 | $(\{t_1\} \cup \{t_2\}) \cap \{t_1, t_2\} \to \{t_1, t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_2\}$ |
| 4 | $(\{t_1\} \cup \{t_2\}) \cap \{t_1, t_2\} \to \{t_1, t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3\}$ |
| 5 | $(\{t_1\} \cup \{t_2, t_3\}) \cap \{t_1, t_2\} \to \{t_1, t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3\}$ |

TAB. 4.3 - *Plus petite solution pour G2.*

**plus grand point fixe :** Initialement $\mathcal{E}' = \{e_1, \uparrow X_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3, \downarrow X_3\}$ et $T' = \{t_1, t_2, t_3\}$. La Table 4.4 donne le résultat.

| itération | $(f_1 \cup f_2) \cap f_3 \to T'$ | $\mathcal{E}'$ |
|---|---|---|
| 1 | $(\{t_1\} \cup \{t_2, t_3\}) \cap \{t_1, t_2\} \to \{t_1, t_2\}$ | $\{e_1, \uparrow X_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3, \downarrow X_3\}$ |
| 2 | $(\{t_1\} \cup \{t_2, t_3\}) \cap \{t_1, t_2\} \to \{t_1, t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3\}$ |
| 3 | $(\{t_1\} \cup \{t_2, t_3\}) \cap \{t_1, t_2\} \to \{t_1, t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3\}$ |

TAB. 4.4 - *Plus grande solution pour G2.*

$G2$ illustre une situation normale. La plus petite et la plus grande solution sont égales. Il n'y a qu'une évolution possible : celle qui franchit dans l'instant $\{t_1, t_2\}$ et conduit à la situation $\{3\}$.

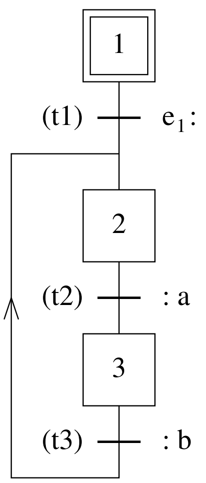



FIG. 4.4 - *S-grafcet G3*

**Exemple G3 :** Soit le s-grafcet $G3$ (Fig. 4.4).

$$\begin{aligned} S &= \{1,2,3\} \\ T &= \{t_1,t_2,t_3\} \\ \mathcal{I} &= \{a,b\} \\ \mathcal{E}_\mathcal{I} &= \{e_1\} \end{aligned}$$

Nous considérons l'interprétation :

$R(t_1).G = \{e_1\}$, $R(t_1).C = 1$; $R(t_2).G = \emptyset$, $R(t_2).C = a$;
$R(t_3).G = \emptyset$, $R(t_3).C = b$.

En syntaxe abstraite, la réceptivité associée à $t_1$ est réécrite sous la forme :

$R(t_1).G = \{\top\}$, $R(t_1).C = P_{e_1}$;

Supposons $e_\mathcal{I} = \{e_1\}$ et $v(a) = v(b) = 1$, les fonctions constantes sont:

$$\begin{aligned} f_1 &= \{t_1\} \\ f_3 &= \{t_1,t_2,t_3\} \\ [\![V]\!] &= [\![a \wedge b \wedge X_1 \wedge \neg X_2 \wedge \neg X_3 \wedge P_{e_1}]\!] \end{aligned}$$

| # | $(f_1 \cup f_2) \cap f_3 \to T'$ | $\mathcal{E}'$ |
|---|---|---|
| 1 | $(\{t_1\} \cup \emptyset) \cap \{t_1,t_2,t_3\} \to \{t_1\}$ | $\{e_1\}$ |
| 2 | $(\{t_1\} \cup \emptyset) \cap \{t_1,t_2,t_3\} \to \{t_1\}$ | $\{e_1, \downarrow X_1, \uparrow X_2\}$ |
| 3 | $(\{t_1\} \cup \{t_2\}) \cap \{t_1,t_2,t_3\} \to \{t_1,t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_2\}$ |
| 4 | $(\{t_1\} \cup \{t_2\}) \cap \{t_1,t_2,t_3\} \to \{t_1,t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3\}$ |
| 5 | $(\{t_1\} \cup \{t_2,t_3\}) \cap \{t_1,t_2,t_3\} \to \{t_1,t_2,t_3\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3\}$ |
| 6 | $(\{t_1\} \cup \{t_2,t_3\}) \cap \{t_1,t_2,t_3\} \to \{t_1,t_2,t_3\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3, \downarrow X_3\}$ |
| 7 | $(\{t_1\} \cup \{t_2,t_3\}) \cap \{t_1,t_2,t_3\} \to \{t_1,t_2,t_3\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3, \downarrow X_3\}$ |

TAB. 4.5 - *Plus petite solution pour G3.*

| # | $(f_1 \cup f_2) \cap f_3 \to T'$ | $\mathcal{E}'$ |
|---|---|---|
| 1 | $(\{t_1\} \cup \{t_2,t_3\}) \cap \{t_1,t_2,t_3\} \to \{t_1,t_2,t_3\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3, \downarrow X_3\}$ |
| 2 | $(\{t_1\} \cup \{t_2,t_3\}) \cap \{t_1,t_2,t_3\} \to \{t_1,t_2,t_3\}$ | $\{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3, \downarrow X_3\}$ |

TAB. 4.6 - *Plus grande solution pour G3.*

Le calcul du plus petit point fixe (Initialement $T' = \mathcal{E}' = \emptyset$) donne $T_f = \{t_1,t_2,t_3\}$ (Table 4.5). Le plus grand point fixe (Initialement $T' = \{t_1,t_2,t_3\}$ et

$\mathcal{E}' = \{e_1, \uparrow X_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3, \downarrow X_3\}$) conduit à la même conclusion (Table 4.6). On franchit donc $T' = \{t_1, t_2, t_3\}$ avec $\mathcal{E}' = \{e_1, \downarrow X_1, \uparrow X_2, \downarrow X_2, \uparrow X_3, \downarrow X_3\}$. Ce qui devrait conduire à la situation $\{2\}$.

Ici apparaît la première limitation de la caractérisation par point fixe. Elle donne l'ensemble des transitions franchies, mais n'indique pas le nombre de franchissements. Sur cet exemple, les transitions $t_2$ et $t_3$ peuvent être franchies un nombre infini de fois durant l'instant. On se trouve en présence d'une instabilité de grafcet. Ce problème est connu en ESTEREL sous le nom de *cycle instantané*. Une solution trouvée par point fixe, doit donc faire l'objet d'une étude de stabilité ultérieure.

#### 4.4.4.3 Cas général

**Insuffisance du système $\Sigma_1$:** Lorsque les réceptivités dépendent aussi des prédicats de présence des événements internes, la fonction $h$ (Eq. 4.24) n'est ni monotone, ni anti-monotone. Ce qui ne garantit plus que $f_3$ (Eq. 4.22) soit antimonotone et $f$ (Eq. 4.19) soit monotone. En conséquence l'existence de points fixes n'est plus assurée. À titre d'illustration, prenons le s-grafcet $G4$ (Fig. 4.5).

$$\begin{aligned} S &= \{1, 10, 20\} \\ T &= \{t_1, t_2\} \\ \mathcal{I} &= \emptyset \\ \mathcal{E}_{\mathcal{I}} &= \{e_1\} \end{aligned}$$

Avec l'interprétation suivante :

$$\begin{aligned} R(t_1).G &= R(t_2).G = \{e_1\} \\ R(t_1).C &= \neg P_{\uparrow X_{20}} \\ R(t_2).C &= \neg P_{\uparrow X_{10}} \end{aligned}$$

Cette interprétation s'écrit en syntaxe abstraite :

$$\begin{aligned} R(t_1).G &= R(t_2).G = \{\top\} \\ R(t_1).C &= \neg P_{\uparrow X_{20}} \wedge P_{e_1} \\ R(t_2).C &= \neg P_{\uparrow X_{10}} \wedge P_{e_1} \end{aligned}$$

Supposons $e_{\mathcal{I}} = \{e_1\}$. Les fonctions constantes sont dans cet exemple

$$\begin{aligned} f_1 &= \{t_1, t_2\} \\ f_2 &= \emptyset \end{aligned}$$

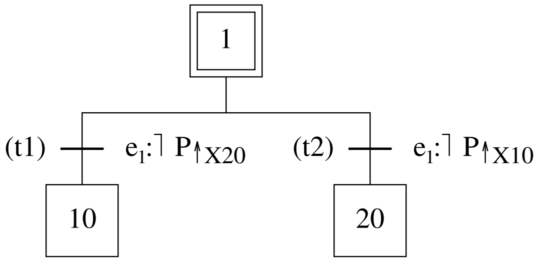



FIG. 4.5 - *S-grafcet G4*

Si on essaie de calculer le plus grand point fixe, on prend initialement :

$$\begin{aligned} T' &= \{t_1, t_2\} \\ \mathcal{E}' &= \{e_1, \uparrow X_1, \downarrow X_1, \uparrow X_{10}, \downarrow X_{10}, \uparrow X_{20}, \downarrow X_{20}\} \\ [\![V]\!] &= [\![X_1 \wedge P_{e_1}]\!] \end{aligned}$$

On vérifie que $f_3$ reste constant et égal à $\{t_1, t_2\}$. La Table 4.7 montre qu'on n'a pas de plus grand point fixe: $T'$, $\mathcal{E}'$ ont des variations non monotones. De même, $[\![V]\!]$ n'est pas monotone car il dépend de $\mathcal{E}'$. Ceci ne signifie pas qu'il n'y ait pas de solution au système $\Sigma_1$. En effet il existe deux solutions, mais leur union n'est pas solution.

$T' = \{t_1\}$ $\qquad$ $T' = \{t_2\}$
$\mathcal{E}' = \{e_1, \downarrow X_1, \uparrow X_{10}\}$ $\qquad$ $\mathcal{E}' = \{e_1, \downarrow X_1, \uparrow X_{20}\}$

| # | $(f_1 \cup f_2) \cap f_3 \to T'$ | $\mathcal{E}'$ |
|---|---|---|
| 1 | $(\{t_1, t_2\} \cup \emptyset) \cap \emptyset \to \emptyset$ | $\{e_1, \downarrow X_1, \uparrow X_1 0, \uparrow X_2 0\}$ |
| 2 | $(\{t_1, t_2\} \cup \emptyset) \cap \emptyset \to \emptyset$ | $\{e_1\}$ |
| 3 | $(\{t_1, t_2\} \cup \emptyset) \cap \{t_1, t_2\} \to \{t_1, t_2\}$ | $\{e_1\}$ |
| 4 | $(\{t_1, t_2\} \cup \emptyset) \cap \{t_1, t_2\} \to \{t_1, t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_1 0, \uparrow X_2 0\}$ |
| 5 | $(\{t_1, t_2\} \cup \emptyset) \cap \emptyset \to \emptyset$ | $\{e_1, \downarrow X_1, \uparrow X_1 0, \uparrow X_2 0\}$ |

TAB. 4.7 - *Non existence de plus grand point fixe*

**Système $\Sigma_2$ :** Afin de contrôler les variations de $[\![V]\!]$ on augmente le nombre d'équations. Aux équations définissant l'ensemble des transitions franchissables, on ajoute des équations caractérisant les transitions non franchissables. Les nouvelles équations constituent le système $\Sigma_2$.

**Définition 15** *On définit les fonctions $f, f', f_1, f_1{}', f_2, f_2{}', f_3, f_3{}', g, g', h$ telles que:*

$$\begin{array}{rcl} f, f_1, f_2, f_3 & : & 2^T \times 2^{\mathcal{E}} \times 2^T \times 2^{\mathcal{E}} \times 2^{\mathbb{B}_L} \to 2^T \\ f', f_1', f_2', f_3' & : & 2^T \times 2^{\mathcal{E}} \times 2^T \times 2^{\mathcal{E}} \times 2^{\mathbb{B}_L} \to 2^T \\ g, g' & : & 2^T \times 2^{\mathcal{E}} \times 2^T \times 2^{\mathcal{E}} \times 2^{\mathbb{B}_L} \to 2^{\mathcal{E}} \\ h & : & 2^T \times 2^{\mathcal{E}} \times 2^T \times 2^{\mathcal{E}} \times 2^{\mathbb{B}_L} \to 2^{\mathbb{B}_L} \end{array}$$

$$f = (f_1 \cup f_2) \cap f_3 \tag{4.31}$$

$$f' = f_1' \cup f_2' \cup f_3' \tag{4.32}$$

$\forall T_1, T_2 \subseteq T; \forall \mathcal{E}_1, \mathcal{E}_2 \subseteq \mathcal{E}; \forall [\![V]\!] \subseteq \mathbb{B}_L$ :

$f_1$ est l'ensemble des transitions gardées validées pour la situation courante $X$ :

$$f_1(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) = \{t \in T |\ (t \in T_g) \wedge ({}^{\circ}t \subseteq X)\} \tag{4.33}$$

$f_2$ est l'ensemble des transitions non gardées qui deviennent validées lors des évolutions internes :

$$\begin{array}{rcl} f_2(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) & = & \{t \in T |\ (t \notin T_g) \wedge \cdots \\ & & \cdots (({}^{\circ}t \setminus X) \subseteq \{s | {\uparrow} X_s \in \mathcal{E}_1\})\} \end{array} \tag{4.34}$$

$f_3$ est l'ensemble des transitions dont la condition est vraie :

$$f_3(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) = \{t \in T |\ [\![V]\!] \subseteq [\![R(t).C]\!]\} \tag{4.35}$$

$g$ est l'ensemble des événements présents :

$$g(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) = e_{\mathcal{I}} \cup \{{\downarrow} X_s |\ s \in {}^{\circ}T_1\} \cup \{{\uparrow} X_s |\ s \in T_1{}^{\circ}\} \tag{4.36}$$

$f_1'$ est l'ensemble des transitions gardées non validées pour la situation courante $X$ :

$$f_1'(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) = \{t \in T |\ (t \in T_g) \wedge ({}^{\circ}t \not\subseteq X)\} \tag{4.37}$$

$f_2'$ est l'ensemble de transitions non gardées qui ne seront jamais validées lors des évolutions internes :

$$\begin{array}{rcl} f_2'(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) & = & \{t \in T |\ (t \notin T_g) \wedge \cdots \\ & & \cdots (({}^{\circ}t \setminus X) \cap \{s | {\uparrow} X_s \in \mathcal{E}_2\} \neq \emptyset)\} \end{array} \tag{4.38}$$

$f_3'$ est l'ensemble de transitions dont la condition est fausse :

$$f_3'(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) = \{t \in T |\ [\![V]\!] \subseteq [\![\neg R(t).C]\!]\} \tag{4.39}$$

$g'$ est l'ensemble des événements absents :

$$\begin{aligned} g'(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) &= (\mathcal{E}_{\mathcal{I}} \setminus e_{\mathcal{I}}) \cup \{\downarrow X_s |\ s \in \widetilde{\text{Pre}}\ T_2\} \cup \cdots \\ &\cdots \{\uparrow X_s |\ s \in \widetilde{\text{Post}}\ T_2\} \end{aligned} \tag{4.40}$$

$h$ définit le singleton de $\mathbb{B}_L$ qui représente l'état de toutes les variables booléennes connuees au début de l'instant, les prédicats de présence des événements de $\mathcal{E}_1$ et les prédicats d'absence des événements de $\mathcal{E}_2$ :

$$\begin{aligned} h(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) &= \left( \bigcap_{(i \in \mathcal{I} \cup \mathcal{X}) \wedge (v(i)=1)} [\![i]\!] \right) \cap \left( \bigcap_{(i \in \mathcal{I} \cup \mathcal{X}) \wedge (v(i)=0)} [\![\neg i]\!] \right) \cdots \\ &\cap \left( \bigcap_{(P_e \in \mathcal{P}) \wedge (e \in \mathcal{E}_1)} [\![P_e]\!] \right) \cap \left( \bigcap_{(P_e \in \mathcal{P}) \wedge (e \in \mathcal{E}_2)} [\![\neg P_e]\!] \right) \end{aligned} \tag{4.41}$$

L'ensemble des transitions franchissables est défini par $f(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!])$, l'ensemble des transitions non franchissables par $f'(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!])$. En fait, nous cherchons à résoudre le système d'équations simultanées $\Sigma_2$:

$$T_1 = f(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) \tag{4.42}$$
$$\mathcal{E}_1 = g(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) \tag{4.43}$$
$$T_2 = f'(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) \tag{4.44}$$
$$\mathcal{E}_2 = g'(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) \tag{4.45}$$
$$[\![V]\!] = h(T_1, \mathcal{E}_1, T_2, \mathcal{E}_2, [\![V]\!]) \tag{4.46}$$

**Remarques :** Pour alléger les équations, nous avons exprimé les réceptivités $< G, C >$ en syntaxe abstraite : $< \top, C \wedge P_G >$. Les fonctions $f$ et $g$ sont analogues à celles introduites dans le cas des conditions statiques (paragraphe 4.4.4.2). Notons toutefois que $\mathcal{E}_1$ correspond uniquement à l'ensemble des événements connus comme étant présents. Ainsi $f_2$ ne dépend explicitement que de $\mathcal{E}_1$ et ${f_2}'$ ne dépend que de $\mathcal{E}_2$.

$f_1$ et ${f_1}'$ sont constants et ne dépendent que de $X$. $f_3$ et ${f_3}'$ ne sont déterminés à partir de $[\![V]\!]$. $g$ (resp. $g'$) ne dépend explicitement que de $T_1$ (resp. $T_2$). La fonction $h$ diffère de celle donnée précédemment (eq. 4.24). Il s'agit cette fois d'une fonction partielle qui comprend une partie constante et une partie variable. La partie variable dépend elle-même de $\mathcal{E}_1$ et $\mathcal{E}_2$. La fonction $h$ est anti-monotone par rapport à $\mathcal{E}_1$ et $\mathcal{E}_2$.

Le système $\Sigma_2$ est syntaxiquement monotone en prenant les variables signées: ${T_1}^+$,${\mathcal{E}_1}^+$,${T_2}^+$,${\mathcal{E}_2}^+$,$[\![V]\!]^-$. On résout alors $\Sigma_2$ en cherchant les plus petits points fixes pour $T_1$,$\mathcal{E}_1$,$T_2$,$\mathcal{E}_2$ et le plus grand point fixe pour $[\![V]\!]$.

$$\begin{aligned}
{T_1}^+ &= f({T_1}^+, {\mathcal{E}_1}^+, {T_2}^+, {\mathcal{E}_2}^+, [\![V]\!]^-) && (4.47)\\
{\mathcal{E}_1}^+ &= g({T_1}^+, {\mathcal{E}_1}^+, {T_2}^+, {\mathcal{E}_2}^+, [\![V]\!]^-) && (4.48)\\
{T_2}^+ &= f'({T_1}^+, {\mathcal{E}_1}^+, {T_2}^+, {\mathcal{E}_2}^+, [\![V]\!]^-) && (4.49)\\
{\mathcal{E}_2}^+ &= g'({T_1}^+, {\mathcal{E}_1}^+, {T_2}^+, {\mathcal{E}_2}^+, [\![V]\!]^-) && (4.50)\\
[\![V]\!]^- &= h({T_1}^+, {\mathcal{E}_1}^+, {T_2}^+, {\mathcal{E}_2}^+, [\![V]\!]^-) && (4.51)
\end{aligned}$$

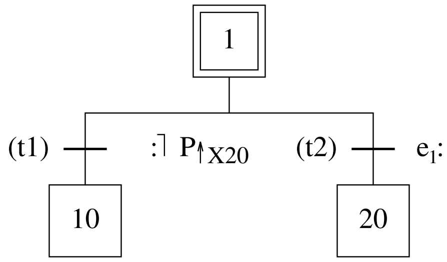



FIG. 4.6 - *S-grafcet G5*

**Exemple G5:** Pour montrer un exemple de résolution du système $\Sigma_2$, considérons le s-grafcet $G5$ (Fig. 4.6).

$$\begin{aligned}
S &= \{1, 10, 20\}\\
T &= \{t_1, t_2\}\\
\mathcal{I} &= \emptyset\\
\mathcal{E}_{\mathcal{I}} &= \{e_1\}
\end{aligned}$$

Avec l'interprétation suivante :

$R(t_1).G = \emptyset$, $R(t_1).C = \neg P_{\uparrow X_{20}}$; $R(t_2).G = \{e_1\}$, $R(t_2).C = 1$;

La transition $t2$ se réécrit en syntaxe abstraite : $R(t_2).G = \{\top\}$, $R(t_2).C = P_{e_1}$;

Supposons $e_{\mathcal{I}} = \{e_1\}$, les fonctions constantes sont:

$$\begin{aligned}
f_1 &= \{t_2\}\\
f_2 &= \{t_1\}\\
{f_1}' &= \emptyset\\
{f_2}' &= \emptyset
\end{aligned}$$

Initialement ${T_1}' = {\mathcal{E}_1}' = {T_2}' = {\mathcal{E}_2}' = \emptyset$. La Table 4.8 résume les calculs.

Le point fixe est atteint avec le franchissement de $t_2$, le "non-franchissement" de $t_1$, l'activation de $\{20\}$ et la désactivation de $\{1\}$. La nouvelle situation stable est donc $\{20\}$.

| # | $(f_1 \cup f_2) \cap f_3 \rightarrow T_1'$ | $\mathcal{E}_1'$ | $f_1' \cup f_2' \cup f_3' \rightarrow T_2'$ | $\mathcal{E}_2'$ |
|---|---|---|---|---|
| 1 | $(\{t_2\} \cup \{t_1\}) \cap \{t_2\} \rightarrow \{t_2\}$ | $\{e_1\}$ | $\emptyset \cup \emptyset \cup \emptyset \rightarrow \emptyset$ | $\{\uparrow X_1, \downarrow X_{10}, \downarrow X_{20}\}$ |
| 2 | $(\{t_2\} \cup \{t_1\}) \cap \{t_2\} \rightarrow \{t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_{20}\}$ | $\emptyset \cup \emptyset \cup \emptyset \rightarrow \emptyset$ | $\{\uparrow X_1, \downarrow X_{10}, \downarrow X_{20}\}$ |
| 3 | $(\{t_2\} \cup \{t_1\}) \cap \{t_2\} \rightarrow \{t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_{20}\}$ | $\emptyset \cup \emptyset \cup \{t_1\} \rightarrow \{t_1\}$ | $\{\uparrow X_1, \downarrow X_{10}, \downarrow X_{20}\}$ |
| 4 | $(\{t_2\} \cup \{t_1\}) \cap \{t_2\} \rightarrow \{t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_{20}\}$ | $\emptyset \cup \emptyset \cup \{t_1\} \rightarrow \{t_1\}$ | $\{\uparrow X_1, \uparrow X_{10}, \downarrow X_{10}, \downarrow X_{20}\}$ |
| 5 | $(\{t_2\} \cup \{t_1\}) \cap \{t_2\} \rightarrow \{t_2\}$ | $\{e_1, \downarrow X_1, \uparrow X_{20}\}$ | $\emptyset \cup \emptyset \cup \{t_1\} \rightarrow \{t_1\}$ | $\{\uparrow X_1, \uparrow X_{10}, \downarrow X_{10}, \downarrow X_{20}\}$ |

TAB. 4.8 - *Solution de G5.*

Le système d'équations $\Sigma_2$ permet également de statuer sur l'évolution du s-grafcet $G4$ (Table. 4.9).

| # | $(f_1 \cup f_2) \cap f_3 \rightarrow T_1'$ | $\mathcal{E}_1'$ | $f_1' \cup f_2' \cup f_3' \rightarrow T_2'$ | $\mathcal{E}_2'$ |
|---|---|---|---|---|
| 1 | $(\{t_1, t_2\} \cup \emptyset) \cap \emptyset \rightarrow \emptyset$ | $\{e_1\}$ | $\emptyset \cup \emptyset \cup \emptyset \rightarrow \emptyset$ | $\{\uparrow X_1, \downarrow X_{10}, \downarrow X_{20}\}$ |
| 2 | $(\{t_1, t_2\} \cup \emptyset) \cap \emptyset \rightarrow \emptyset$ | $\{e_1\}$ | $\emptyset \cup \emptyset \cup \emptyset \rightarrow \emptyset$ | $\{\uparrow X_1, \downarrow X_{10}, \downarrow X_{20}\}$ |

TAB. 4.9 - *Solution de G4.*

Le point fixe est trouvé avec aucun franchissement ($T_1 = \emptyset$) et aucun "non-franchissement" ($T_2 = \emptyset$). Pour le modèle S-GRAFCET, cette solution n'est pas suffisante car le statut de $t_1$ et $t_2$ reste *indéterminé*: $T_1$ et $T_2$ ne forment pas une partition de $T$.

## 4.5 Conclusion sur les points fixes

Nous avions à résoudre un système d'équations simultanées. Toutefois les conditions booléennes associées aux transitions n'ont pas des propriétés de monotonie suffisantes pour assurer l'existence de point fixe pour l'ensemble des transitions franchissables dans l'instant. Si on contraint ces conditions (indépendance vis à vis des prédicats de présence d'événements internes), alors on peut calculer un plus petit et un plus grand point fixe pour l'ensemble des transitions franchissables dans un instant. Si ces deux points fixes coïncident, cette solution unique, bien que satisfaisant aux exigences statiques imposées par les règles d'évolution du S-GRAFCET et du GRAFCET, peut ne pas être satisfaisante d'un point de vue dynamique (interprétation du modèle *avec recherche de stabilité*). Il peut exister une où plusieurs boucles instantanées (connue sous le nom "*d'instabilité*" en GRAFCET),

Dans le cas général (conditions non-statiques), nous devons augmenter le nombre d'équations afin de pouvoir calculer un plus petit point fixe pour l'ensemble des transitions franchissables. Nous définissons en particulier l'ensemble $T_1$ des transitions franchissables et l'ensemble $T_2$ des transitions non-franchissables. Là encore l'existence d'une solution ne garantie pas que nous avons trouvé une "bonne" solution à notre problème. Deux cas peuvent se produire :

1. $(T_1, T_2)$ n'est pas une partition de $T$.

   Pour ce cas, nous pouvons adopter différentes positions :

   - Rejeter la solution. Vu que nous ne connaîssons pas le statut des transitions $T \setminus \{T_1, T_2\}$, le s-grafcet ne peut être considéré comme strictement déterministe.
   - Accepter la solution. Dans ce cas nous devons donner un statut à chaque transitions indéterminée et vérifier que la solution précédemment trouvée est toujours viable. Ceci induit de nouveaux problèmes car la solution trouvée ne sera pas forcément unique (problème de causalité). De plus nous pouvons tomber sur des s-grafcets schizophrenic qui admettent ou non une solution au fur et à mesure que les transitions restantes sont itérativement statuées (absence de convergence vers la solution).

2. $(T_1, T_2)$ est une partition de $T$.

   Même dans ce cas, la solution unique trouvée n'est pas forcément viable. Nous pouvons rencontré un cycle instantanée composé de transitions infiniment franchissables. Ceci conduit à une situation instable. La solution doit donc être rejetée.

Comme un s-grafcet doit avoir un comportement déterminé et stable, une spécification par *point fixe* de la solution, n'est pas suffisante. Elle demande une analyse *a posteriori*. Dans le cas d'une violation de causalité, cette spécification est de plus inadaptée pour garder une trace des franchissements qui ont menés à la violation. Ce même reproche peut aussi être fait pour les cycles instantanés.

Dans le chapitre suivant, nous allons apporter deux réponses à ces différentes questions : la première préconise de décrire les comportements du S-GRAFCET en ESTEREL pour bénéficier de la sémantique de ce langage. Les problèmes de causalité, d'unicité de la solution et de cycle instantané seront alors résolus par ESTEREL lui-même. La seconde solution reprend les définitions vues dans ce chapitre et caractérise les évolutions par une notion de *micro-pas*. Ceci permet la définition d'un algorithme itératif pour quantifier la solution.

# Chapitre 5

# Compilation effective du modèle S-grafcet

## 5.1 Introduction

Le GRAFCET a été conçu au départ comme un formalisme de spécification des systèmes de commande. La commande ainsi modélisée peut ensuite être programmée (microprocesseur, automate industriel) ou câblée avec la technologie la plus appropriée (électrique, hydraulique, pneumatique). La puissance d'expression du Grafcet a poussé rapidement les industriels à développer sur leur matériel des "langages grafcet" de programmation de plus en plus proches de la *syntaxe* grafcet tout en ayant une *sémantique* différente. En particulier ces langages *interprètent* les règles d'évolution et intègrent plus ou moins bien l'hypothèse d'évolution en *temps nul* du modèle. Ceci a de multiples conséquences. D'abord les utilisateurs peuvent confondre *modèle* et *langages industriels*. Ensuite les grafcets programmés ne sont pas portables d'un constructeur à un autre. Enfin tout système *modélisé* en GRAFCET risque d'être implanté d'une manière différente en langage grafcet. Tous ces problèmes sont encore aggravés avec les ambiguïtés sémantiques propres au modèle (§ 3).

Face à ces problèmes, des travaux récents ont porté sur la sémantique et la compilation du GRAFCET. L'idée principale est de traduire de manière systématique le comportement d'un grafcet vers un modèle "plus performant" qui dispose soit de théorèmes mathématiques puissants, soit d'outils de preuves formelles de comportements[1]. En particulier des travaux de traduction du comportement ont été menés vers les `Réseaux de Pétri` [Moa85, AD93], vers les systèmes de transition [Lep94], vers le langage synchrone SIGNAL [Lep94], vers les automates d'état [Rou94]. Ainsi la génération de systèmes de transition ou d'automates

1. ces deux qualités vont d'ailleurs souvent de pair !

d'états permet l'utilisation de l'outil de preuves MEC [ABC94]. En particulier, l'approche par automate, est basée sur la recherche exhaustive des situations accessibles [M.B79] couramment employée par les utilisateurs du GRAFCET. De même la génération d'équations SIGNAL de comportement équivalent, permet de bénéficier de l'environnement de ce langage synchrone. D'ailleurs ceci permet d'intégrer directement un grafcet dans une application écrite en SIGNAL et forme une première tentative d'intégration du modèle dans une plate-forme synchrone commune.

Notre approche se trouve au confluent de ces travaux. Les différentes traductions et compilations que nous allons maintenant présenter de l'interprétation S-GRAFCET, apportent des résultats similaires sur les automates d'états. Elle permet de plus l'intégration du modèle au sein d'une plate-forme synchrone de deux manières différentes. Elle permet enfin l'utilisation d'outils de preuves plus appropriés au modèle GRAFCET.

### 5.1.1 Modélisation du comportement en ESTEREL

Nous allons présenter dans les chapitres 5.2 et 5.3 des méthodes de traduction du GRAFCET en ESTEREL. Comme ESTEREL est aussi un formalisme, cette "traduction" peut aussi se voir comme une *modélisation* par ESTEREL du comportement de systèmes réactifs GRAFCET. Vues les raisons sémantiques exposées au chapitre 3.3 sur les ambiguïtés du modèle, nous ne considèrerons que l'interprétation S-GRAFCET. Ce choix est aussi guidé par la sémantique d'ESTEREL : d'une part, celle-ci refuse toute ambiguïté comportementale, d'autre part, elle repose sur le principe d'*instantanéïté des réactions internes*. En fait, la modélisation des comportements GRAFCET permet de bénéficier des avantages sémantiques du *formalisme* ESTEREL et de l'environnement de programmation associé au *langage* ESTEREL(simulation et preuves). Elle assure de plus l'intégration du modèle GRAFCET qui est vu comme un sous-ensemble d'ESTEREL. En particulier nous pourrons directement utiliser les mécanismes de préemption forte et faible de ce langage synchrone. Nous avons ainsi montré dans [AG94] que l'association GRAFCET-ESTEREL était intéressante pour modéliser les modes de marches et d'arrêt des systèmes.

### 5.1.2 Compilation directe

Parallèlement à ces travaux, nous avons développé une méthode de compilation directe du S-GRAFCET et du GRAFCET standard vers un automate d'états ou vers un automate booléen [Hal94]. Ces travaux peuvent être rapprochés de ceux de J. M. Roussel [Rou94] même si les principes de compilation utilisés sont totalement différents. Comme lui, nous partirons de l'hypothèse de M. Blanchard que "*tout grafcet possède un automate d'états équivalent construit à partir de son graphe des situations accessibles*" [M.B79].

Les différences sémantiques entre S-GRAFCET et GRAFCET portent sur la recherche de l'état suivant. Nous donnerons donc au chapitre 5.4.2, un algorithme de recherche pour l'interprétation S-GRAFCET. Ce dernier est issu de l'approche par point fixe présentée au chapitre § 4.

## 5.2 Modélisation en ESTEREL par une vision structurelle

### 5.2.1 Présentation

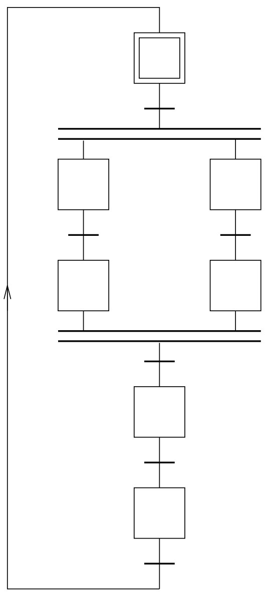
Dans [AP92a], les auteurs ont montré l'intérêt de traduire tout grafcet en programme ESTEREL de comportement équivalent. Outre les intérêts de ce langage vus au § 2.2.1, les auteurs ont mis en avant sa capacité d'exprimer *explicitement* le parallélisme et la séquentialisation de processus synchrones. Comme ces deux notions sont également essentielles en GRAFCET : elles définissent la "forme" du graphe et les précédences sémantiques entre étapes, les auteurs ont eu l'idée de transcrire directement la structure d'un grafcet en ESTEREL.

La modélisation par " vision structurelle" est en droite ligne de ces travaux. Pour mettre en avant la *forme* d'un grafcet, il suffit de regrouper chaque étape et sa transition aval au sein d'un "module" unique `step`" et ensuite d'assembler chaque instance de ces modules à l'aide des opérateurs parallèle "`||`" et séquentiel "`;`" d'ESTEREL. La figure 5.1 montre un exemple d'une telle traduction structurelle.

### 5.2.2 Entités de base

Avec ce mode de traduction, l'entité la plus petite est le module STEP. Celui-ci peut se représenter par une boîte noire réactive qui serait sensible aux réceptivités booléennes et événementielles et qui émettrait dans l'instant les actions. Cette entité doit pouvoir communiquer instantanément avec les autres et *évoluer* en conséquence. C'est pourquoi, elle doit aussi être sensible et capable d'émettre des événements internes.

Suivant le type de réceptivité (purement événementiel, purement conditionnel...), le code ESTEREL du module STEP est susceptible de changer. Pour réduire le nombre de cas possibles, nous associons à chaque réceptivité booléenne un événement interne qui ne *garde* pas l'étape-transition STEP. La transcription des conditions en événement apporte un autre avantage : les programmes sont écrits



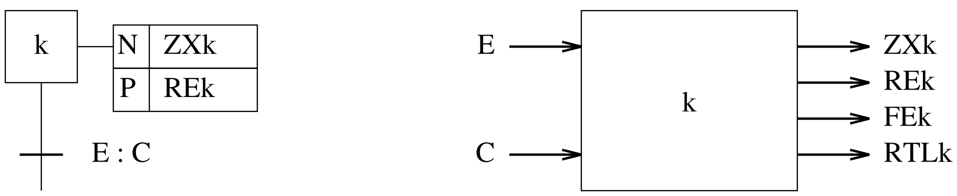
```
loop
  run STEP [...]
  ;
  [
    [
      run STEP [...]
      ;
      run STEP [...]
    ]
    ||
    [
      run STEP [...]
      ;
      run STEP [...]
    ]
  ]
  ;
  run STEP [...]
  ;
  run STEP [...]
end loop
```

FIG. 5.1 - *Traduction structurelle*

en ESTEREL pur, ce qui permet d'utiliser pleinement le compilateur et d'éviter de fausses causalités. Avec cette écriture, les entités STEP se regroupent en deux grandes familles comportementales suivant qu'elles soient gardées ou non.

#### 5.2.2.1 Étape-transitions gardées

Cette famille d'étape-transition que nous nommerons "STEP_G", est au moins gardée par un événement. Son évolution peut aussi être déterminée par une condition ou un prédicat d'événement. Elle peut se représenter en GRAFCET par la figure 5.2a et sous forme d'un système réactif autonome par la figure 5.2b.




( a ) ( b )

FIG. 5.2 - *Représentation d'une étape-transition gardée*

| entrées | sorties |
|---|---|
| $E$ : événement | $ZXk$ : activation de l'action associée |
| $C$ : prédicat d'événement | $REk$ : $\uparrow X_k$ (Raising Edge) |
| ou condition | $FEk$ : $\downarrow X_k$ (Falling Edge) |
| | $RTLk$ : étape prête à sortir (Ready To Leave) |

Les signaux de sortie $ZXk$, $REk$ et $FEk$ sont classiques. Le premier sert à activer l'action continue associée, si l'étape est stable. Les autres peuvent être utilisés soit au niveau des réceptivités, soit pour engendrer des actions impulsionnelles. Par contre le signal $RTLk$ est plus spécifique à notre modélisation. Il est émis par l'étape si cette dernière peut potentiellement *terminer dans l'instant*. Nous verrons au § 5.2.3.2 l'importance de ce signal. Munie de ces signaux, l'étape-transition gardée a un comportement qui peut se décrire par le programme ESTEREL suivant :

```
module STEP_G
input E,C;
output RTLk, ZXk, REk, FEk;

emit REk;
emit ZXk;
trap STEPk in
    every tick do
        emit RTLk;
        present E and C then
            exit STEPk
        else
            emit ZXk
        end present
    end every
end trap;
emit FEk
end module
```

Comme l'étape-transition est *gardée* par un événement, elle ne peut terminer dans l'instant. Cette propriété se voit immédiatement sur l'automate d'états correspondant Fig. 5.3. Pour le premier instant, on remarquera de plus la présence du signal de stabilité $ZXk$ et l'absence du signal $RTLk$.

#### 5.2.2.2 Étape-transitions non gardées

Nous nommerons cette famille d'étape-transition "STEP_NG", Elle n'est pas gardée par un événement et peut donc terminer *dans l'instant* si la réceptivité

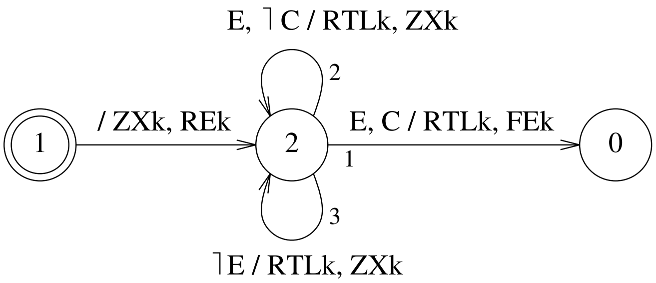



FIG. 5.3 - *Automate d'états correspondant*

conditionnelle est vraie. Elle peut se représenter en GRAFCET par la figure 5.4a et sous forme d'un système réactif autonome par la figure 5.4b.

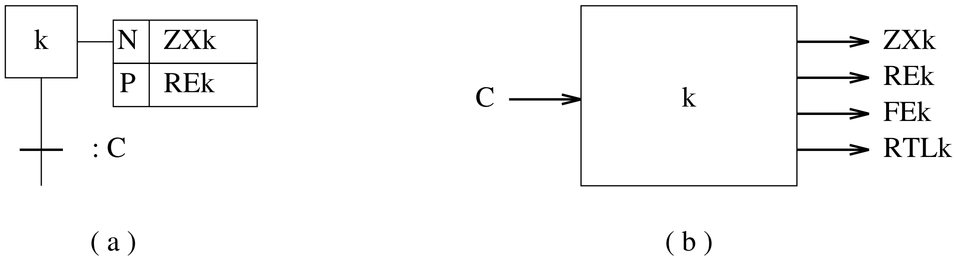



FIG. 5.4 - *Représentations d'une étape-transition non gardée*

Les signaux de sortie $ZXk$, $REk$ et $FEk$ et $RTLk$ ont la même définition que ceux vus précédemment. Par contre le modèle comportemental ESTEREL est différent dans le cas d'une étape-transition non gardée :

```
module STEP_NG
input C;
output RTLk, ZXk, REk, FEk;

emit REk;
trap STEPk in
    loop
        emit RTLk;
        present C then
            exit STEPk
        end present;
        emit ZXk
    each tick
end trap;
emit FEk
end module
```

La propriété de terminaison instantanée se voit sur le code ESTEREL : contrairement au module STEP_G, la boucle peut être préemptée avant la terminaison du premier instant. Cette propriété caractérise d'ailleurs la forme de l'automate d'états Fig. 5.5 pour toute cette famille d'étape-transition.

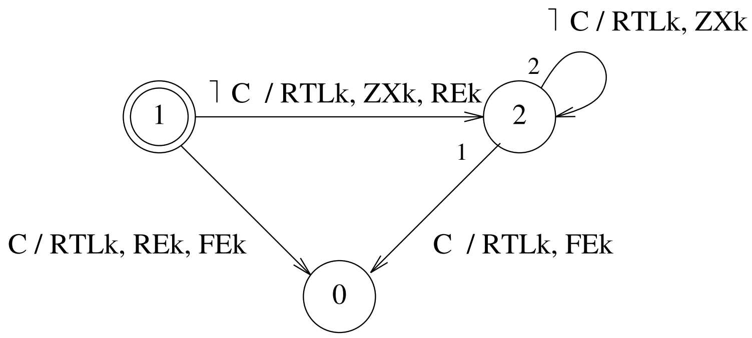



FIG. 5.5 - *Automate d'états correspondant*

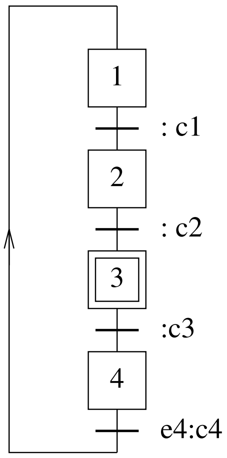
### 5.2.3 Modélisation des structures GRAFCET

Les deux entités STEP_G et STEP_NG sont directement agencées suivant la topologie du grafcet avec les opérateurs "`;`" et "`||`". Pour décrire un grafcet quelconque, il nous faut cependant encore étudier quelques opérateurs spécifiques au GRAFCET.

#### 5.2.3.1 Boucle séquentielle

Pour réaliser des boucles séquentielles instantanées, ESTEREL dispose de la structure "`loop ... end loop`" que nous allons utiliser. La traduction d'une boucle GRAFCET vers une boucle ESTEREL n'est pourtant pas immédiate. En effet le GRAFCET débute la boucle avec l'étape initiale alors que ESTEREL exécute la *première* instruction de la boucle. Il est donc impératif lors de la génération de code ESTEREL de décrire la boucle séquentielle à partir de l'étape initiale (Fig. 5.6).

De même, si des étapes initiales se trouvent dans une sous-structure elle-même incluse dans une boucle, la traduction ESTEREL de cette sous-structure devra être placée en tête de la boucle ESTEREL. Ici apparaît la première limitation de la méthode. Il n'est pas possible de traduire une boucle qui contient plusieurs étapes initiales (fig. 5.7). En effet ces étapes évoluent parallèlement malgré leur séquentialité syntaxique ce qui ne correspond pas du tout à la sémantique de l'opérateur "`;`" ( [BG92]).




```
loop
    run STEP_NG [signal c1 / c,
                      RE3 / REk, FE3 / FEk ...] ;
    run STEP_G    [signal c2 / c,
                      RE4 / REk, FE4 / FEk ...] ;
    run STEP_NG [signal c3 / c,
                      RE1 / REk, FE1 / FEk ...] ;
    run STEP_NG [signal c4 / c, e4 / e,
                      RE2 / REk, FE2 / FEk ...]
end loop
```

FIG. 5.6 - *Code Esterel associé à une boucle Grafcet*

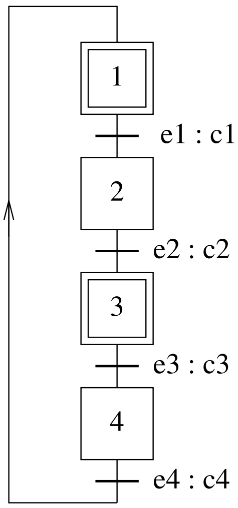



FIG. 5.7 - *Boucle non directement traduisible*

#### 5.2.3.2 Divergence et convergence en ET

Les divergences en ET peuvent facilement se traduire par l'opérateur "||". Elles représentent en effet l'évolution *concurrente* de deux processus synchrones. Par contre, la convergence en ET nécessite plus de travail de conversion. En effet elle lie deux étapes ou plus, à une *unique* transition. Pour modéliser ces convergences, nous ne possédons que des entités étapes-transitions indissociables. Pour expliquer la sémantique des convergences en ET, certains ouvrages didactiques donnent l'équivalence de la figure 5.8.

Cette écriture repose les problèmes d'évolution et de variables internes étudiés au chapitre 3.3.2.4. Nous la refuserons pour les raisons déjà évoquées. De plus en S-GRAFCET, nous considérons que la valeur des variables $Xk$ est celle

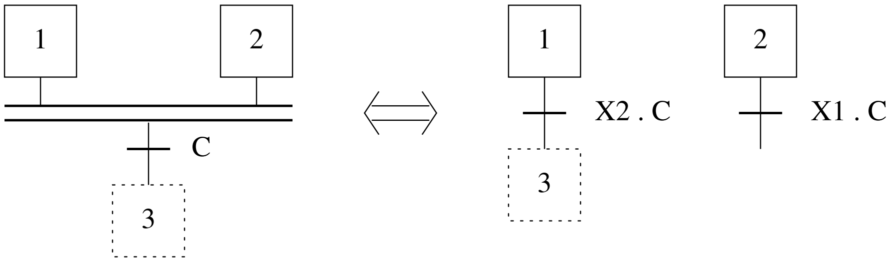



FIG. 5.8 - *Première équivalence d'une convergence en* ET

au *début de l'instant.* Nous devons donc trouver une autre manière d'exprimer le comportement des convergences en ET.

Pour cela revenons à leur définition: pour que la transition soit *validée*, il faut et il suffit que les étapes en convergence soient actives (mais pas forcément stables!). Cette activation a pu arriver lors d'un instant précédent de telle sorte que l'étape soit restée stable (mémorisée par la variable $Xk$). Cette activation peut aussi avoir lieu dans l'instant (pour les transitions non gardées): dans ce cas elle apparaît par l'intermédiaire de l'événement interne $\uparrow Xk$. Cette constatation nous conduit à l'écriture fig. 5.9 où $P$ est la fonction prédicat définie au § 4.

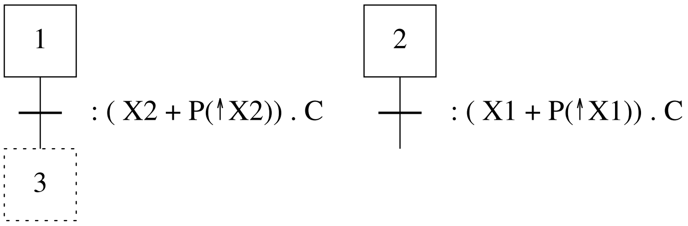



FIG. 5.9 - *Seconde équivalence d'une convergence en* ET

Pour condenser cette écriture, nous avons défini le nouveau signal de sortie RTLk. Comme celui-ci est émis dès que l'étape $k$ est susceptible d'être désactivée, il s'instaure entre les étapes en convergence en ET un *dialogue instantané* qui conduit, soit à la stabilité, soit à la désactivation globale simultanée.

Ainsi chaque étape-transition $k$ non gardée qui arrive sur une convergence en "et", émet à partir de cet instant (et pour tous ceux qui suivent) son signal "RTLk". La dernière étape activée, se désactive instantanément car elle "entend" le signal "RTLk" de toutes ses sœurs. Elle émet tout de même son propre signal "RTLk" pour que les autres étape-transitions puissent se désactiver à leur tour dans le *même* instant.

Par la compilation ESTEREL, ce mécanisme de synchronisation n'apporte aucune lourdeur au code généré car les signaux "RTLk" sont internes et disparaissent entièrement à la compilation. Nous pouvons donc traduire la convergence en ET précédente par l'extrait de code ESTEREL :

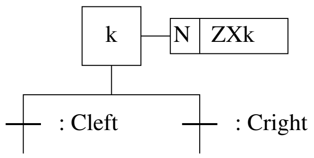
```
module GRAFCET
input C,...;
output ...;
signal RTL1, RTL2, Cfor1, Cfor2, ...in

    loop
        present C and RTL1 then emit Cfor2 end
        ||
        present C and RTL2 then emit Cfor1 end
    each tick
    ||
    ...
    [
        run STEP_NG [signal Cfor1/c, RTL1/RTLk]
        ||
        run STEP_NG [signal Cfor2/c, RTL2/RTLk]
    ]
    ;
    ...
```

#### 5.2.3.3 Divergence et convergence en OU

Comme la divergence en ET, la convergence en OU d'étapes-transitions ne pose pas de problème, elle correspond à une terminaison d'alternative ESTEREL `present ... end`. Par contre la divergence en OU se décrit mal avec nos entités STEP. Le problème est aggravé si les réceptivités ne sont pas disjonctives. Pour ce cas, nous avons vu (§3.2.1.3) que les branches étaient susceptibles de s'exécuter en parallèle.

Même si nous ne considérons que grafcets disjonctifs, nous sommes quand même obligé de définir de nouvelles entités "étape-transition" qui indiquent lors de leur désactivation, quelle branche a été lancée. Ainsi une "étape-transitions" non gardée comportant deux transitions de sortie pourra se modéliser par le programme ESTEREL de la figure 5.10. La transition de sortie sera connue par la présence ou l'absence de l'événement RWk (Right Way k).




```
module STEP2_NG
input Cleft, Cright;
output RTLk, ZXk, REk, FEk, RWk;

emit REk;
trap STEPk in
    loop
        emit RTLk;
        present Cleft or Cright then
            exit STEPk
        end present;
        emit ZXk
    each tick
end trap;
present Cright then emit RWk end present
end module
```

FIG. 5.10 - *Modèle* ESTEREL *associé à une divergence en* OU

### 5.2.4 Problèmes liés à cette approche

La compilation du GRAFCET en ESTEREL par cette approche structurelle, pose de nombreux problèmes théoriques et pratiques :

- Il faut reconnaître les différentes structures classiques du GRAFCET et les hiérarchiser suivant un arbre syntaxique. Ceci est assez facile pour les divergences et les convergences en OU/ET. Par contre identifier les boucles est beaucoup plus délicat. Une solution consisterait à éliminer les autres structures en remplaçant chaque bloc identifié par une macro-étape, puis de rechercher sur chaque sous-grafcet sa fermeture transitive. Une autre solution pourrait mettre en œuvre un algorithme de potentiel sur le graphe complet. La présence de boucle pose également le problème des étapes initiales qui doivent impérativement être placées en première position dans le code ESTEREL.

- Nous avons vu au chapitre 5.2.3.1 que les séquences comportant plusieurs étapes initiales ne pouvaient pas être modélisées en ESTEREL par des séquences d'entités "STEP".

- La gestion des divergences en OU nécessite la création de nouvelles entités STEP qui possèdent à chaque fois le bon nombre de transitions de sortie. Ceci limite la réutilisabilité du code généré. De plus, ceci induit un *ordre implicite* d'évaluation des réceptivités et interdit les divergences en OU non disjonctives.

- Le codage du GRAFCET sous forme d'une hiérarchie arborescente d'opérateurs suppose qu'à chaque divergence en ET est associée *une et une seule* convergence en ET. Cette remarque est aussi vraie pour les divergences et les convergences en OU. Le grafcet doit donc être “bien structuré”. En particulier le grafcet Fig. 3.15 étudié au chapitre 3 est difficilement traduisible. Cette limitation ne serait pas gênante si la norme imposait aux utilisateurs d'écrire des grafcets bien structurés. La réalité est autre : la destructuration intrinsèque du GRAFCET donne un pouvoir d'expression supplémentaire qui est réellement utilisé en pratique. Elle permet par exemple d'exprimer des *synchronisations locales* entre processus.

### 5.2.5 Conclusion

Parmi tous les points précédents, le dernier est suffisamment important pour *remettre complètement en cause cette approche*. En effet l'intégration du GRAFCET dans une plate-forme synchrone est intéressante si elle permet d'exprimer en GRAFCET des comportements difficilement modélisables par les autres modèles synchrones. En particulier de nombreux systèmes de commande se modélisent facilement sous une écriture “destructurée”.

Pour ces raisons, l'approche structurelle est plus difficile à mettre en œuvre que prévu et perd de son intérêt pratique et théorique. Les recherches dans cette voie ont donc été freinées au profit de l'approche “équationnelle” que nous allons maintenant présenter.

L'approche “structurelle” a tout de même été présentée dans ce mémoire car elle a montré l'importance :

- des signaux “RTLk” qui deviennent fondamentaux dans l'approche équationnelle,
- de séparer en deux entités distinctes *étape* et *transition* pour gérer facilement les divergences en OU,
- de considérer que toutes les étapes (même en séquence) sont concurrentes entre elles car elles sont susceptibles d'être actives en *même temps*.

## 5.3 Modélisation en ESTEREL par une vision équationnelle

### 5.3.1 Présentation

Dans l'approche équationnelle, un S-GRAFCET peut être perçu comme un ensemble d'étapes interagissant par dialogues instantanés. Les contraintes imposées aux dialogues résultent de la topologie du GRAFCET et de son interprétation. Les grandes différences avec la vision structurelle portent d'une part sur la séparation complète de l'étape et de sa (ou ses) transitions, d'autre part sur le regroupement de toutes les transitions en un "moteur de transition unique" et enfin sur la transposition des structures GRAFCET.

Les "étapes" et le "moteur de transition" sont des objets réactifs qui communiquent instantanément entre eux (fig. 5.11). À partir d'une situation connue et d'un environnement donné, ils cherchent la nouvelle situation (ensemble des étapes actives) du système. Tous ces comportements sont décrits par des modules ESTEREL mis en parallèle. Le problème de trouver l'état suivant, se ramène donc à la résolution d'un système d'équations (de signaux à émettre dans l'instant).

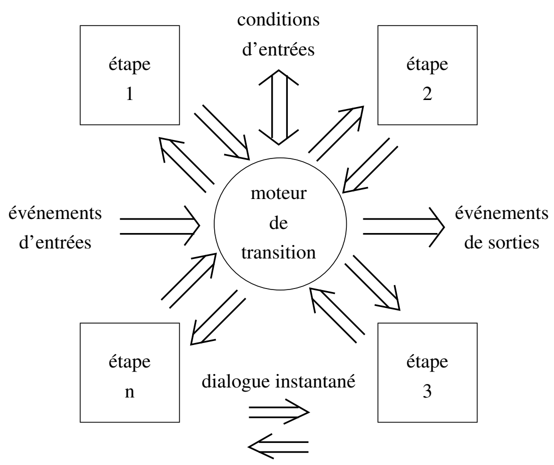



FIG. 5.11 - *Schéma comportemental d'un grafcet*

L'apport de notre approche, est d'exprimer ces équations de telle sorte que le compilateur ESTEREL sache les résoudre pour chaque instant lorsque le grafcet est stable et les rejette sinon (pas de solution unique). Nous nous sommes appuyé pour cela sur la sémantique comportementale d'ESTEREL. Nous montrerons d'ailleurs au § 5.3.3 pourquoi les grafcets instables sont *systématiquement* rejetés.

Toutefois il peut rejeter, à tort, certains programmes corrects par présomption de cycle de causalité. Les grafcets qui utilisent plutôt les conditions booléennes que les événements, induisent a priori de nombreux rejets de la part du compilateur. La traduction que nous proposons au niveau des modules, a sû contourner ce problème. Elle est de plus entièrement écrite en ESTEREL *pur*, ce qui permet d'augmenter le nombre de propriétés logiquement vérifiables sur l'automate d'états équivalent.

## 5.3.2 Entités de base

### 5.3.2.1 Étapes

Une étape peut être considérée comme un objet réactif qui communique avec son environnement par des signaux d'entrée (Xk_Init, Xk_Set, Xk_Reset) et des signaux de sortie (REk, FEk, RTLk, ZXk, Xk). La signification de ces signaux est précisée ci-dessous. Le comportement (réactif) d'une étape peut s'exprimer par l'automate déterministe Fig. 5.12. Le code ESTEREL associé est décrit en annexe. Nous pouvons remarquer qu'il est : ni concis, ni "agréable" à lire. Il a été au contraire développé pour réduire au maximum le nombre de faux rejets par le compilateur V3. En particulier le premier instant est codé à part et le signal RTLk est émis dès que possible, pour "aider" l'algorithme de Gonthier.

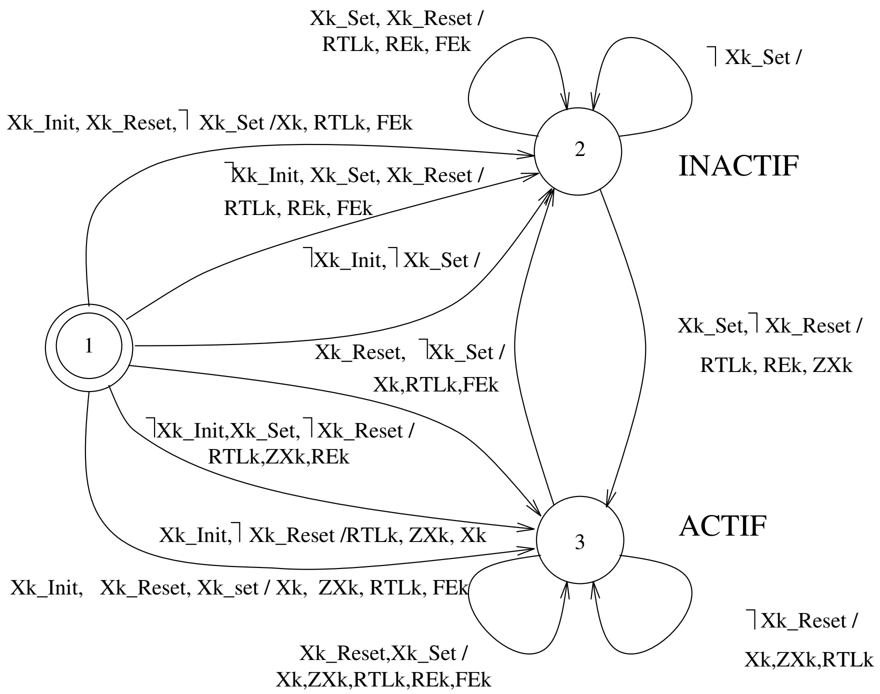



FIG. 5.12 - *Automate d'états associé au comportement d'une étape*

- Xk_Init: initialisation de l'étape,
- Xk_Set: ordre d'activation de l'étape,
- Xk_Reset: ordre de désactivation de l'étape,
- REk: l'étape activée dans l'instant (Rising Edge),
- FEk: l'étape désactivée dans l'instant (Falling Edge),
- RTLk: l'étape prête à être désactivée dans l'instant (Ready To Leave),
- ZXk: l'étape stable à la "fin" de l'instant,
- Xk: l'étape stable au "début" de l'instant (ou à la "fin" de l'instant précédent).

#### 5.3.2.2 Moteur de transition

Le moteur de transition regroupe l'ensemble des transitions du grafcet. Il orchestre toutes les communications entre les étapes suivant l'état des réceptivités. Son comportement est de plus, dépendant de la topologie du réseau. En supposant par exemple qu'une transition "$k$" relie "$n$" étapes amont (convergence en ET) vers "$m$" étapes aval (divergence en *et*) (Fig. 5.13), le moteur aura la forme caractéristique suivante:

```
loop
    [
        ⋮
        % transition K
        present RTLamont1 then
            ⋮
            present RTLamontN then
                present receptK then
                        emit Xamont1_RESET;
                        ...
                        emit XamontN_RESET;
                        emit Xaval1_SET;
                        ...
                        emit XavalM_SET
                end present
            end present
            ⋮
        end present
        ||
        ⋮
        % transition K+1
    ]
each tick
```

Ce moteur de transition s'appuie sur l'utilisation intensive des signaux "RTLk" dont la sémantique est la même que pour l'approche structurelle. Dès que toutes les étapes amont sont actives (même transitoirement), elles émettent leur signal "RTL" (Fig. 5.12). Le moteur de transition réagit instantanément à la présence de tous les signaux "RTL" attendus et émet dans l'instant les ordres d'activation des étapes aval et de désactivation des étapes amont. suivant la valeur des conditions. Si la condition est vraie, toutes les étapes aval émettent à leur tour le signal "RTL", susceptible de faire réagir le moteur une nouvelle fois [2] sur une

2. Rappelons que cette micro-évolution se fait dans le même instant logique que la précédente!

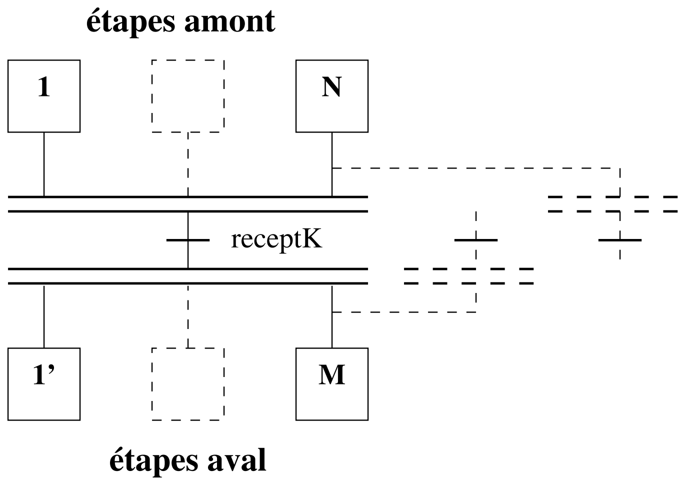



FIG. 5.13 - *Convergence/ divergence en “et”*

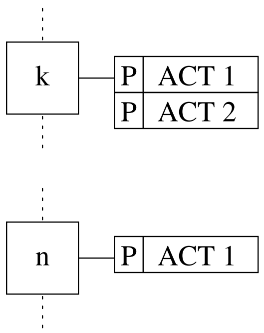
autre transition.

### Remarques

- Lorsque la transition est gardée par un événement, les signaux “RTLk” sont remplacés par les signaux “Xk” qui ne sont émis que si les étapes $k$ considérées sont stables et actives au début de l'instant (interprétation S-GRAFCET).
- La place du “`present receptk`” permet de tester la réceptivité $k$ uniquement lorsque celle-ci est réellement attendue.
- La séquence de deux étapes est un cas particulier où le nombre d'étapes amont et aval est réduit à l'unité.
- La convergence ou la divergence en OU de $n$ étapes s'écrit facilement par la mise en parallèle de $n$ motifs ESTEREL précédents au sein du moteur de transition.
- Les communications instantanées entre le moteur de transition et les étapes permettent le franchissement instantané d'une séquence finie quelconque d'étapes.
- Contrairement à l'approche structurelle, celle-ci traite correctement les divergences en OU non disjonctives. Elle suit complètement l'interprétation S-GRAFCET (comme le respect de l'élément neutre) car toute étape $k$ desactivée dans l'instant par une première transition, continue d'émettre son signal “RTLk” pour toutes transitions validées.

#### 5.3.2.3 Gestionnaires d'Actions

La gestion en ESTEREL des actions impulsionnelles ne pose pas de problème vue la sémantique choisie (§3.3.3). Il suffit de tester l'occurrence du signal `REk` correspondant à chaque instant. Ainsi les actions impulsionnelles du grafcet Fig.5.14a seront gérées suivant le programme 5.14b.




( a )

```
% actions emission

loop
    present REk then
        emit ACT1 ;
        emit ACT2
    end present
    ||
    present REn then
        emit ACT1
    end present
each tick
```

( b )

FIG. 5.14 - *Gestion des actions impulsionnelles*

Par contre les actions continues ne sont pas directement traduisibles en ESTEREL car ce dernier s'appuie sur un modèle discret du temps. Cette continuité peut en fait être simulée de deux manières différentes.

- Soit l'action est émise à chaque instant. Le signal associé prend alors la forme d'un peigne d'événements.
- Soit la gestion effective de l'action est dédiée à un opérateur synchrone externe (bascule S/R). Dans ce cas le module de commande doit seulement émettre l'ordre d'activation "`SET_action`" et l'ordre de désactivation "`RESET_action`".

Ces deux cas sont présentés par le chronogramme de la figure 5.15. Nous remarquerons qu'il ne doit pas y avoir de discontinuité vis à vis des actions commune à plusieurs étapes comme c'est le cas pour `act1`. En particulier les ordres `SET_act1` et `RESET_act1` ne doivent jamais être émis en même temps.

Si l'action doit être émise à chaque instant, le problème est similaire à celui des actions impulsionnelles. Par contre le signal interne déclencheur ne sera plus `REk` (étape activée dans l'instant), mais `ZXk` (étape stable à la fin de l'instant).

Si l'action est gérée par les signaux `SET` et `RESET`, ceux-ci peuvent être émis par des modules spéciaux (Action_N_Manager) instanciés pour chaque action. Le module Action_N_Manager peut s'écrire en ESTEREL sous la forme:

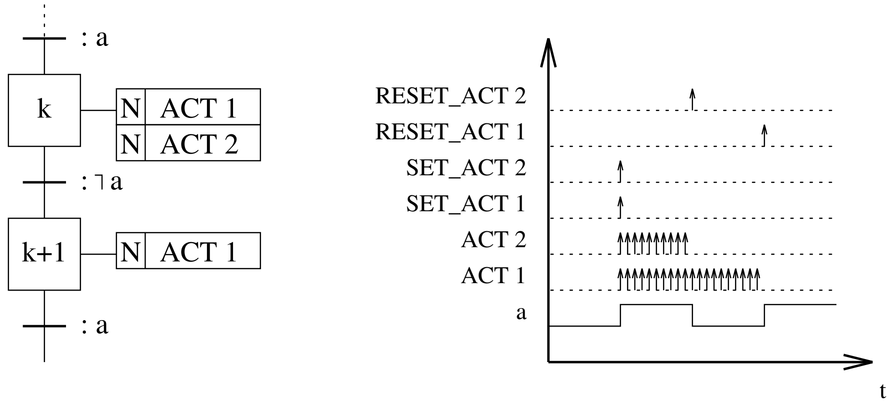



FIG. 5.15 - *Chronogramme des actions continues*

```
module action_N_manager:
input ACTk;
output ACTk_set,
        ACTk_reset;

signal ACTk_pre in
    loop
        % regeneration du ACTk_pre a partir du ACTk
        present ACTk then
            await tick;
            emit ACTk_pre
        else
            await tick
        end present
    end loop
    ||
    loop
        present ACTk then
            present ACTk_pre else
                emit ACTk_set
            end present
        else
            present ACTk_pre then
                emit ACTk_reset
            end present
        end present
    each tick
end signal
end module
```

**Remarque :** La gestion des actions (continues ou événementielles) conditionnées par une entrée "C" est une généralisation de l'extrait de code 5.14b :

```
loop
    % action impulsionnelle ``i'' émis par l'étape ``k''
    % conditionnée par ``C''
    present REk then
        present C then
            emit ACTi
        end present
    end present
    ||
    % action continue ``j'' émis par l'étape ``l''
    % conditionnée par ``C''
    present ZXl then
        present C then
            emit ACTj
        end present
    end present
    ...
each tick
```

### 5.3.3 Détection des grafcets instables par ESTEREL

Nous avons donné dans le chapitre 3.2.2.4 une définition de la stabilité d'un grafcet. Cette définition induit celle de l'instabilité qui peut s'exprimer ainsi :

"Un grafcet est instable ssi il existe au moins une transition dont le franchissement implique de nouveau son propre franchissement dans le même instant."

Or notre modèle comportemental attache au franchissement de chaque transition au moins deux signaux locaux que sont : `Xaval_SET` et `Xamont_RESET`. La notion d'instabilité GRAFCET va donc s'exprimer en ESTEREL par l'existence d'au moins un signal local qui dépend de lui-même à un instant donné. Cette propriété est classiquement appelée "erreur de causalité" [Gon88] en ESTEREL et provoque systématiquement le rejet du programme.

ESTEREL est donc capable de détecter formellement *tout* grafcet instable. Cette propriété offre deux avantages pratiques. En premier un compilateur ESTEREL est capable de trouver la situation et l'environnement qui ont conduit à cette instabilité. En second, les signaux locaux qui interviennent dans cette instabilité se réfèrent à l'ensemble des transitions intervenant dans l'instabilité.

### 5.3.4 Problèmes liés à cette approche

#### 5.3.4.1 Terminaison des Grafcets

Le code ESTEREL décrivant un grafcet par la vision équationnelle est composé de boucles et de modules mis en parallèle. Parmi ces entités, aucune ne *termine*[3]. Vue la sémantique de l'opérateur "`||`", le module ESTEREL complet ne finit jamais. En GRAFCET, la présence de transition "puits" conduit parfois à la désactivation de toutes les étapes. Le grafcet considéré devient totalement inactif et nous pouvons considérer qu'il termine dans l'instant.

Pour prendre en compte ce phénomène en ESTEREL, il est nécessaire de pouvoir préempter le module complet dans l'instant où *toutes* les étapes deviennent inactives sans remettre en cause les signaux émis. Ceci peut être fait avec la puissante instruction ESTEREL `trap`. Le module complet prendra alors la forme :

```
module GRAFCET
...
% emission des signaux Xk_init ...;
trap cycle_END_REQ in
    % moteur de transition
    loop
        ...
        present not ZX1 then
            ⋮
                present not ZXi then
                    exit cycle_END_REQ
                end present
            ⋮
        end present
    each tick
    ||
 %  etapes
    ⋮
    % gestion des actions
    ⋮
end trap
end module
```

3. au sens ESTEREL du terme.

#### 5.3.4.2 Fausses instabilités

Les compilateurs ESTEREL ne garantissent pas l'obtention de la solution pour tout programme ESTEREL viable. Par exemple le module STEP a été réécrit plusieurs fois pour être accepté par le compilateur V3. Ce phénomène est encore amplifié avec l'approche équationnelle puisque les dépendances entre les signaux `RTLk`, `Xk_set` et `Xk_reset` sont très fortes. Les grafcets les plus sensibles sont ceux dont aucune transition n'est gardée et dont la stabilité est uniquement assurée par des réceptivités disjonctives. En effet les gardes événementielles imposent un *changement explicite d'instant*. Parmi ces grafcets, celui de la figure 5.16 est un exemple typique de grafcet stable dont le code ESTEREL associé admet une solution unique mais qui est pourtant rejeté par le compilateur V3.

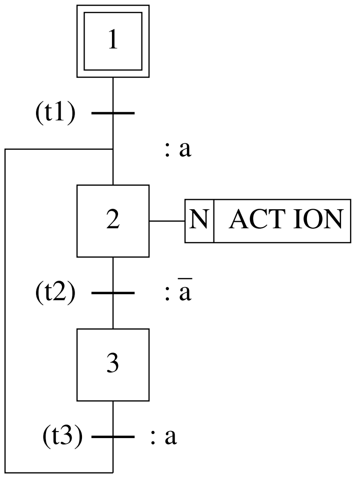



FIG. 5.16 - *Grafcet stable rejeté par le compilateur* ESTEREL

Ce problème intervient souvent quand le compilateur soupçonne des causalités sur des cas "qu'il ne devrait pas étudier". Sur le grafcet précédent, il est inutile d'étudier la franchissabilité de la transition $t2$ lorsque le grafcet est dans l'état initial car la variable booléenne $a$ ne peut pas être `vraie` et `fausse` en même temps. De plus cette non-franchissabilité implique la non-franchissabilité de $t3$ située en aval. Pour les mêmes raisons, l'instant de désactivation de l'étape 1 ne peut pas être celui du franchissement de $t3$. Dans le cas général, il est possible de connaître statiquement pour chaque étape un ensemble de transitions susceptibles[4] d'être franchi lorsque cette étape est désactivée.

D'ou l'idée d'utiliser cette analyse statique pour *aider* le compilateur ESTEREL à trouver la solution. Le principe consiste à *geler* le moteur de transition dans certains états du grafcet. Bien sûr les modifications de code ne doivent pas changer le comportement du grafcet, ni rendre artificiellement stable un grafcet instable.

4. Sur-ensemble des transitions réellement franchissables

Pour celà nous avons développé un algorithme “d'accessibilité” qui étudie statiquement la topologie du grafcet et qui fournit une matrice (étapes,transition) dont chaque élément $acces_{ij}$ est une fonction booléenne. Cette dernière caractérise la combinaison des entrées *nécessaires* mais non suffisantes pour franchir instantanément la transition $j$ lorsque l'étape $i$ est désactivée. Pour que cette fonction booléenne soit tautologiquement fausse il suffit que tout chemin de $i$ vers $j$ contienne des réceptivités disjonctives ou une transition gardée par un événement qui ne soit pas directement en aval de $i$. L'algorithme associé est rendu complexe par la possibilité d'atteindre une transition par plusieurs chemins et par le comportement des transitions gardées.

```
début
    pour toute transition j
      /* premiere etape: initialisation d'une ligne i de access */
      /* avec la receptivite conditionnelle associe a j */
      /* dans le cas ou i precede et j*/
      pour toute etape i
        si i ∈ pre(j) alors
          access[i,j] ← receptivite[j].C
        sinon
          access[i,j] ← FAUX
        fin si
      fin pour

      /* seconde etape: affectation d'une ligne i de access */
      /* avec le produit cumule des receptivites conditionnelles */
      /* dans le cas ou le cumul des transitions */
      /* est franchissable dans l'instant */
      /* recherche de l'etape amont */
      faire
        termine ← VRAI
        pour toute etape i
          si non completement_gardee(i) alors
            pour toute etape j
              si access[i,j] ≠ FAUX alors
                pour toute transition k ∈ pre(i)
                  chemin ← access[i,j] and receptivite(k).C
                  si chemin ≠ FAUX alors
                    si j = k alors
                      INSTABILITE_POTENTIELLE ← VRAI
                    fin si
                    pour toute etape l ∈ pre(k)
                      cond ← chemin or access[l,j]
                      si cond ≠ access[l,j]
                        access[l,j] ← cond
```

```
                    termine ← FAUX
                  fin si
                fin pour
              fin si
            fin pour
          fin si
        fin pour
      fin si
    fin pour
  tant que non termine
fin pour
fin
```

De cette matrice résultat, ne sont finalement gardées que les fonctions tautologiquement fausses. Elles sont utilisées pour surcharger le moteur de transition avec l'intégration des événements `Xk`. Ces derniers sont connus dès le début de l'instant (car résultat de l'instant précédent). Leur présence va activer l'étude des transitions “intéressantes” au niveau du compilateur ESTEREL.

Pour montrer un exemple de construction de la matrice $acces_{ij}$, considérons la figure 5.17.

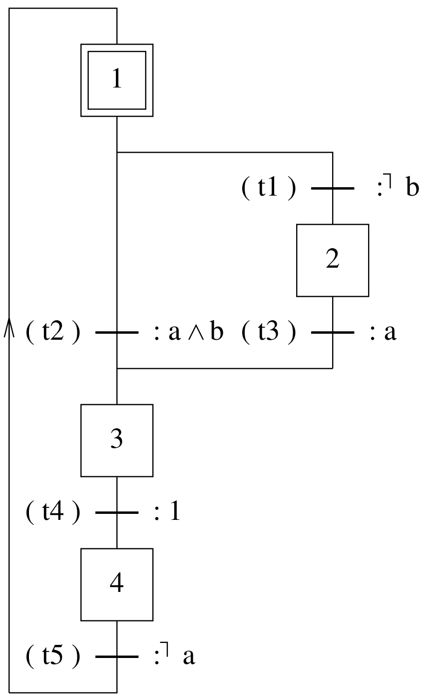



FIG. 5.17 - *Accessibilité statique d'un s-grafcet*

À chaque itération, l'algorithme obtient la matrice d'accessibilité suivante : première étape : franchissement simple

| | $t1$ | $t2$ | $t3$ | $t4$ | $t5$ |
|---|---|---|---|---|---|
| 1 | $\neg b$ | $a \wedge b$ | 0 | 0 | 0 |
| 2 | 0 | 0 | $a$ | 0 | 0 |
| 3 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 0 | $\neg a$ |

seconde étape, première itération : franchissement double

| | $t1$ | $t2$ | $t3$ | $t4$ | $t5$ |
|---|---|---|---|---|---|
| 1 | $\neg b$ | $a \wedge b$ | $a \wedge \neg b$ | $a \wedge b$ | 0 |
| 2 | 0 | 0 | $a$ | $a$ | 0 |
| 3 | 0 | 0 | 0 | 1 | $\neg a$ |
| 4 | $\neg a \wedge \neg b$ | 0 | 0 | 0 | $\neg a$ |

seconde étape, seconde itération : franchissement triple

| | $t1$ | $t2$ | $t3$ | $t4$ | $t5$ |
|---|---|---|---|---|---|
| 1 | $\neg b$ | $a \wedge b$ | $a \wedge \neg b$ | $a$ | 0 |
| 2 | 0 | 0 | $a$ | $a$ | 0 |
| 3 | $\neg a \wedge \neg b$ | 0 | 0 | 1 | $\neg a$ |
| 4 | $\neg a \wedge \neg b$ | 0 | 0 | 0 | $\neg a$ |

seconde étape, troisième itération : franchissement quadruple

| | $t1$ | $t2$ | $t3$ | $t4$ | $t5$ |
|---|---|---|---|---|---|
| 1 | $\neg b$ | $a \wedge b$ | $a \wedge \neg b$ | $a$ | 0 |
| 2 | 0 | 0 | $a$ | $a$ | 0 |
| 3 | $\neg a \wedge \neg b$ | 0 | 0 | 1 | $\neg a$ |
| 4 | $\neg a \wedge \neg b$ | 0 | 0 | 0 | $\neg a$ |

**Remarques :**

- L'algorithme s'arrête à la troisième itération car la matrice n'évolue plus.

- Pour franchir la transition 4 à partir de l'étape 1, il faux que $a$ soit vrai. La valeur de $b$ n'intervient plus car les branches ont reconvergées avant la transition.

- L'algorithme permet également de détecter des boucles instantanées *potentielles*. Celles-ci ne contiennent aucune transition gardée et le produit des réceptivités n'a pas un noyau vide. Cette condition n'est pourtant pas suffisante pour rendre la boucle *instable* car cette dernière peut être synchronisée via les convergences en ET avec d'autres éléments du grafcet.

**Modification du moteur de transition :**

La première colonne de la matrice d'accessibilité indique les conditions nécessaires (mais non suffisantes) pour franchir $t1$ dans l'instant à partir de chaque étape. Celle-ci pourrait être franchie en partant de l'étape 1 (avec la condition $\neg b$), ou de l'étape 3 ($\neg a \wedge \neg b$) ou de l'étape 4 ($\neg a \wedge \neg b$). De même, si l'étape 2 devient active et stable à un instant donné, la réaction qui suivra ne pourra en aucun cas conduire au franchissement de $t1$. L'étude de la franchissabilité de $t1$ est donc nécessaire si les étapes 1 ou 3 ou 4 sont actives au début de l'instant. Ceci se traduit en ESTEREL par :

```
present X1 or X3 or X4 then
    % etude de t1
    ...
end present
```

Le même raisonnement est appliqué aux autres transitions: la franchissabilité de $t2$ (resp. $t3$) dépend de $\{1\}$ (resp. $\{1, 2\}$). De même $t4$ (resp. $t5$) est potentiellement franchissable à partir de $\{1, 2, 3\}$ (resp. $\{3, 4\}$) uniquement.

Sur cet exemple, le moteur de transition peut maintenant prendre la seconde forme et être accepté par le compilateur :

**sans recherche d'accessibilité :**

```
loop
  [

    present RTL1 then
      present not b then
        emit X2_SET;
        emit X1_RESET
      end present
    end present

    ||

    present RTL1 then
      present a and b then
        emit X3_SET;
        emit X1_RESET
      end present
    end present

    ||

    present RTL2 then
      present a then
        emit X3_SET;
        emit X2_RESET
      end present
    end present

    ||

    present RTL3 then
      present tick then
        emit X4_SET;
        emit X3_RESET
      end present
    end present

    ||

    present RTL4 then
      present not a then
        emit X1_SET;
        emit X4_RESET
      end present
    end present

    ||
    ...
  ]
each tick
```

**avec recherche d'accessibilité :**

```
loop
  [
    present X1 or X3 or X4 then
      present RTL1 then
        present not b then
          emit X2_SET;
          emit X1_RESET
        end present
      end present
    end present
    ||
    present X1 then
      present RTL1 then
        present a and b then
          emit X3_SET;
          emit X1_RESET
        end present
      end present
    end present
    ||
    present X1 or X2 then
      present RTL2 then
        present a then
          emit X3_SET;
          emit X2_RESET
        end present
      end present
    end present
    ||
    present X1 or X2 or X3 then
      present RTL3 then
        present tick then
          emit X4_SET;
          emit X3_RESET
        end present
      end present
    end present
    ||
    present X3 or X4 then
      present RTL4 then
        present not a then
          emit X1_SET;
          emit X4_RESET
        end present
      end present
    end present
    ||
    ...
  ]
each tick
```

### 5.3.5 Limites du compilateur ESTEREL V4

Le compilateur V4 est basé sur l'algorithme de Mignard-Fornari [Mig94, For95]. Le grand avantage vis à vis de l'algorithme de Gonthier est de ne plus avoir besoin de générer l'ensemble des états accessibles. Par contre le nouveau compilateur est encore plus susceptible vis à vis des programmes "bizarres". Ainsi l'exemple qui suit [For95], est rejeté par le compilateur V4. Il est pourtant sémantiquement correct.

```
module refuse:
    signal S, P in
        await tick;
        present S then emit T end present;
        await tick;
        present T then emit S end present
    end signal
end module
```

Ce rejet peut facilement s'expliquer. Une des phases de la compilation consiste à trouver les liens de dépendances entre les signaux. Dans cet exemple `T` dépend de `S` (second instant) et `S` dépend de `T` (troisième instant). La phase suivante consiste à trouver un ordre topologique de ces liens de dépendance valable pour *tous* les instants (sauf l'instant initial géré à part). Si cet ordre n'est pas trouvé le programme est rejeté. L'exemple précédent est donc refusé car il présente un cycle (Fig. 5.18) de dépendance.

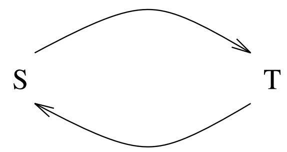



FIG. 5.18 - *Cycle de dépendance entre S et T*

Les auteurs précisent eux-mêmes que ce rejet est tout à fait *arbitraire*. Il a pourtant été maintenu car il n'existe pour l'instant pas de solution et les cycles de dépendance cachent souvent des erreurs de modélisations.

Avec l'approche équationnelle, cette prise de position induit des conséquences fâcheuses. Nous avons en effet *volontairement* couplé les signaux internes, pour que le système admette une solution uniquement lorsque le grafcet est stable. Tout grafcet Fig.5.19a possédant au moins une boucle sans transition gardée mènera à un cycle de dépendance Fig. 5.19b. Il sera irrémédiablement rejeté par le compilateur V4 même s'il est viable. En fait le compilateur ne se rendrait pas compte que le cycle est cassé à chaque instant à un endroit différent.

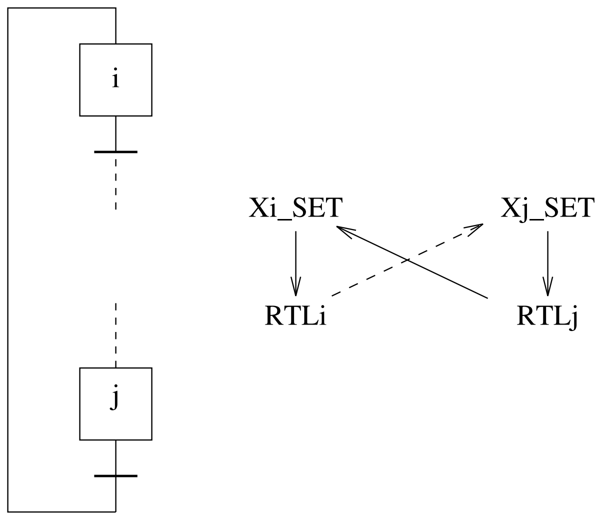



FIG. 5.19 - *Cycle de dépendance sur les boucles grafcet*

Finalement le compilateur V4 ne sait pas faire la distinction entre "cycle sur tous les instants" et "cycle par instant" ce qui est très gênant pour notre approche. Heureusement il intègre *toujours* l'algorithme de Gonthier en option qui nous donne satisfaction. Actuellement des recherches sont menées pour ne plus rejeter les programmes ESTEREL sémantiquement justes qui possèderaient ces dépendances pseudo-cycliques. Une nouvelle version du compilateur a d'ailleurs été annoncée. Elle est basée sur une sémantique dite "constructive" et devrait de nouveau accepter nos programmes sémantiquement corrects.

## 5.3.6 Le compilateur G2E

### 5.3.6.1 Introduction

Pour valider l'approche équationnelle, nous avons développé en langage C un compilateur de GRAFCET vers ESTEREL (G2E). Ce compilateur est actuellement opérationnel. L'approche équationnelle, permet d'intégrer n'importe quelle topologie de grafcet (ce qui aurait été impossible avec l'approche structurelle). De plus la génération du moteur de transition est beaucoup plus aisée et systématique que la construction d'un arbre syntaxique.

La génération automatique de code est intéressante sur de nombreux points :

- Elle évite toutes les erreurs humaines de traduction.

- Elle permet de compiler de plus gros programmes.

- Elle peut être incluse dans des chaînes de compilation.

G2E offre ces trois avantages. Il permet principalement d'intégrer complètement le modèle GRAFCET (interprétation S-GRAFCET) dans la chaîne de compilation d'ESTEREL (Fig. 5.20).

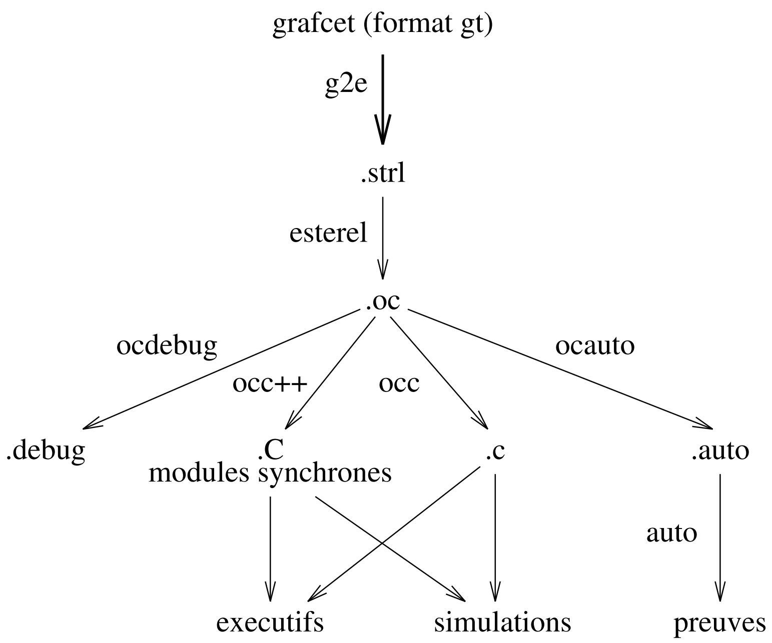



FIG. 5.20 - *Chaîne de compilation de Grafcet*

#### 5.3.6.2 Principe de la compilation

À partir d'un fichier grafcet (format GT) généré par l'éditeur EDGE ou EG7, le compilateur détermine la matrice d'accessibilité. À ce niveau, il manipule les expressions booléennes via leur représentation sous forme de BDDS. Il engendre ensuite le code ESTEREL de comportement équivalent suivant la topologie du grafcet (définition des signaux, moteur de transition, instanciation des étapes, gestion des actions). Il peut générer de plus, une bibliothèque contenant les corps des modules `STEP` et `ACTION_N_MANAGER`. Le code généré peut ensuite intégrer la chaîne de compilation d'ESTEREL vue précédemment. Il peut de plus, être utilisé par EG7 pour simuler le comportement d'un grafcet face à une histoire d'entrée du système.

### 5.3.7 Conclusion

La compilation directe du S-GRAFCET en ESTEREL offre de nombreux avantages. Elle permet de bénéficier de la sémantique mathématique du modèle ESTEREL et de l'environnement de programmation du langage ESTEREL (simulateurs, outils de preuves). Elle permet aussi de modéliser des applications par une approche multiformalisme. En effet certains comportements sont beaucoup plus faciles à exprimer en GRAFCET (synchronisations locales). D'autres sont exprimables plus facilement en ESTEREL (préemptions).

L'approche équationnelle permet une génération ESTEREL beaucoup plus aisée que l'approche structurelle même si cette dernière paraît plus intuitive. Ses principaux avantages sont de pouvoir traduire n'importe quelle structure grafcet et de générer du code ESTEREL non viable si le grafcet de départ possède une boucle instantanée.

En fait, cette approche déplace le problème de la stabilité du modèle GRAFCET vers le modèle ESTEREL. Ceci pose problème au compilateur v4 qui refuse des grafcets stables. Pourtant la version 3 qui implémente l'algorithme de Gonthier, est parfaitement capable de résoudre les systèmes d'équations engendrés par notre approche, pourvu qu'elle soit aidée par une pré-étude d'accessibilité. Ce problème spécifique de la version 4 est connu. Des travaux ont été effectués pour lever cette limitation. Ils ont conduit très récemment à définir une nouvelle sémantique qui accepterait *tous* les programmes corrects en ESTEREL v3 ou v4. En pratique, ce phénomène n'est pas gênant si certaines transitions du grafcet peuvent être gardée par un événement. Le programme est alors compilable quelle que soit la version. Si cette condition n'est pas remplie, le compilateur V4 offre toujours la possibilité d'utiliser l'algorithme de Gonthier via l'option "oldcausality".

## 5.4 Compilation directe du GRAFCET

La compilation directe du GRAFCET avec l'interprétation S-GRAFCET consiste à trouver pour tout grafcet stable, une *machine d'états* entièrement spécifiée et de comportement équivalent. Elle diffère des approches précédentes qui utilisent ESTEREL pour spécifier les comportements, puis utilisent le compilateur associé pour générer cette machine d'états. La sémantique de comportement choisie pour trouver "l'état suivant" est issue du modèle de point fixe présenté au chapitre 4 sans en avoir les inconvénients. Elle apporte de plus une caractérisation pratique de la notion d'*instabilité*, et permet de compiler tout grafcet stable. En cela, elle est supérieure à l'approche équationnelle.

### 5.4.1 Caractérisation de la solution par micro-pas

#### 5.4.1.1 Micro-instants

Une réaction est décomposée en micro-pas. Un micro-pas correspond au franchissement d'un sous-ensemble de transitions ce qui donne des situations intermédiaires. Un micro-pas s'exécute dans un micro-instant. Contrairement à l'algorithme Maximal Parallèle Itéré (MPI), l'ensemble de transitions franchies dans un micro-instant n'est pas l'ensemble des transitions franchissables au sens classique. Ce n'est qu'un sous-ensemble de cet ensemble, le choix est fait de façon à ne pas hypothéquer des franchissements ultérieurs possibles. Bien-sûr le nombre de micro-instants doit être fini. Nous introduisons divers ensembles:

- $T_{cf}$ : ensemble des transitions *certainement franchies* dans l'instant,
- $T_{cnf}$ : ensemble des transitions *certainement non franchies* dans l'instant,
- $\mathcal{E}_{cp}$ : ensemble des événements *certainement présents* dans l'instant,
- $\mathcal{E}_{ca}$ : ensemble des événements *certainement absents* dans l'instant,
- $Ai_{ce}$ : ensemble des actions impulsionnelles *certainement émises* dans l'instant,
- $Ac_{ce}$ : ensemble des actions continues *certainement émises* dans l'instant,
- $V$ : expression booléenne représentant la valuation (partielle).

Une réaction est une séquence finie de micro-pas telle qu'il existe un entier $n$ et

$$\xi^0 \xrightarrow[T^1]{e^1, v^1} \xi^1 \xrightarrow[T^2]{e^2, v^2} \xi^2 \cdots \xi^{n-1} \xrightarrow[T^n]{e^n, v^n} \xi^n$$

avec $\forall k \in \{0..n\} : \xi^k \subseteq S; \xi^0 = X^\theta; \xi^n = X^{\theta+1}$
$\forall k \in \{1..n\} : (T^k \subseteq T) \wedge (\forall l \neq k : T^k \cap T^l = \emptyset);$
et $T_{cf} = \bigcup_{k=1}^n T^k$.

Les $\xi^k$ sont les situations transitoires obtenues au cours de la réaction. Les $T^k, e^k, v^k$ sont respectivement l'ensemble des transitions franchies lors du $i^{ième}$ micro-pas, l'événement et la valuation tels qu'ils sont connus lors de ce pas.

#### 5.4.1.2 Principe du calcul

On va effectuer des micro-pas, jusqu'à ce qu'on atteigne la stabilité ou un cas d'erreur (présence de cycle instantané). Les ensembles précédents sont calculés itérativement. On note en exposant le numéro du micro-pas courant. Au $k^{ième}$ micro-pas, lorsque la situation courante est $\xi^{k-1}$, on calcule $T^k_{pf}$ l'ensemble des *transitions potentiellement franchissables* au cours de ce micro-pas. De ce sous-ensemble, on extrait le sous-ensemble maximal $T^k$ qui est l'ensemble des *transitions effectivement franchies* lors du $k^{ième}$ micro-pas. La connaissance de $T^k$ nous permet de détecter des situations anormales avec arrêt de l'algorithme, ou bien de calculer $T^k_{cf}$,$T^k_{cnf}$,$\mathcal{E}^k_{cp}$,$\mathcal{E}^k_{ca}$,$V^k$ pour l'itération suivante. Les équations doivent être choisies de façon à ne mettre dans ces ensembles que des informations certaines (c'est à dire non susceptibles d'être remises en cause lors de micro-pas ultérieurs). Une autre contrainte est de retrouver la solution classique dans les cas "ordinaires".

**Équations:** Elles sont bien-sûr inspirées de celles trouvées dans l'approche par calcul de points fixes. Nous nous contenterons de commenter celles qui diffèrent.

$$\begin{aligned}
T^k_{pf} &= (f_1 \cup f_2) \cap f_3 && (5.1)\\
f_1 &= \{t \in T|\ (R(t).G \neq \emptyset) \wedge (\ ^{\circ}t \subseteq (X^{\theta} \cap \xi^{k-1}))\} && (5.2)\\
f_2 &= \{t \in T|\ (R(t).G = \emptyset) \wedge (\ ^{\circ}t \subseteq \xi^{k-1})\} && (5.3)\\
f_3 &= \{t \in T|\ [\![V^{k-1}]\!] \subseteq [\![R(t).C]\!]\} && (5.4)
\end{aligned}$$

**Commentaires:** Ces équations sont indépendantes. $T^k_{pf}$ se calcule aisément. $T'^k$, qui est le plus grand sous-ensemble de $T^k_{pf}$ vérifiant la propriété:

$$\forall t \in T'^k : (^{\circ}t)^{\circ} \subseteq (T'^k \cup T^{k-1}_{cnf})$$

est déterminé comme le plus grand point fixe de l'équation (5.5). Sa signification est que toute étape $s$ désactivée par $t$ ($s \in\ ^{\circ}t$) ne sera plus utile pour des franchissements ultérieurs, dans le même instant. $T^{k-1}_{cnf}$ et $T^k_{pf}$ étant donnés, le membre droit de l'équation (5.5) est une fonction monotone de $T'^k$. De plus la solution est recherchée sur un ensemble fini. Le plus grand point fixe s'obtient donc aisément en itérant à partir de $T$.

$$T'^k = T^k_{pf} \cap \mathrm{Post}\ \left(\widetilde{\mathrm{Pre}}\ (T'^k \cup T^{k-1}_{cnf})\right) \tag{5.5}$$

L'ensemble des transitions effectivement franchies dans le micro-pas est $T^k$ obtenu à partir de $T'^k$ en supprimant les transitions gardées déjà franchies dans les micro-pas précédents (Eq. 5.6). La raison de ces suppressions est qu'une transition gardée *ne peut être franchie qu'une fois dans un instant.*

$$T^k = T'^k \setminus \{t \in T|\ (R(t).G \neq \emptyset) \wedge (t \in T^{k-1}_{cf})\} \tag{5.6}$$

Le franchissement de $T^k$ conduit à la situation $\xi^k$ telle que:

$$\xi^k = \left(\xi^{k-1} \setminus\ ^{\circ}(T^k)\right) \cup (T^k)^{\circ} \tag{5.7}$$

Les ensembles $T^k_{cf}$ et $\mathcal{E}^k_{cp}$ se calculent ainsi:

$$\begin{aligned}
T^k_{cf} &= T^{k-1}_{cf} \cup T^k && (5.8)\\
\mathcal{E}^k_{cp} &= e_{\mathcal{I}} \cup \{\downarrow X_s|\ s \in\ ^{\circ}(T^k_{cf})\} \cup \{\uparrow X_s|\ s \in (T^k_{cf})^{\circ}\} && (5.9)
\end{aligned}$$

**Remarque:** L'équation 5.9 peut s'écrire de façon incrémentale grâce à la propriété d'additivité[5] des opérateurs pre et post [Sif82].

$$\mathcal{E}_{cp}^{k} = \mathcal{E}_{cp}^{k-1} \cup \{\downarrow X_s |\ s \in {}^{\circ}(T^k)\} \cup \{\uparrow X_s |\ s \in (T^k)^{\circ}\} \tag{5.10}$$

L'ensemble $T_{cnf}^k$ est caractérisé par des conditions suffisantes de *non franchissabilité*:

$$\begin{aligned} T_{cnf}^k &= f_1' \cup f_2' \cup f_3' && (5.11)\\ f_1' &= \{t \in T |\ (R(t).G \neq \emptyset) \wedge ({}^{\circ}t \not\subset X^{\theta})\} && (5.12)\\ f_2' &= \{t \in T |\ ((R(t).G = \emptyset) \wedge (({}^{\circ}t \setminus X^{\theta}) \cap \{s |\ \uparrow X_s \in \mathcal{E}_{ca}^{k-1}\} \neq \emptyset\})\} && (5.13)\\ f_3' &= \{t \in T |\ [\![V^{k-1}]\!] \subseteq [\![\neg R(t).C]\!]\} && (5.14) \end{aligned}$$

**Commentaires:** $f_1'$ est une constante, elle se calcule à l'initialisation du calcul d'un pas (micro-pas 0). $f_2'$ est monotone vis à vis de $\mathcal{E}_{ca}^{k-1}$; $f_3'$ est antimonotone vis à vis de $[\![V^{k-1}]\!]$. Donc pour une suite croissante de $\left(\mathcal{E}_{ca}^k\right)$ et décroissante de $\left([\![V^k]\!]\right)$, la suite $\left(T_{cnf}^k\right)$ est croissante.

$$\mathcal{E}_{ca}^k = (\mathcal{E}_{\mathcal{I}} \setminus e_{\mathcal{I}}) \cup \{\downarrow X_s |\ s \in \widetilde{\text{Pre}}\ (T_{cnf}^k)\} \cup \{\uparrow X_s |\ s \in \widetilde{\text{Post}}\ (T_{cnf}^k)\} \tag{5.15}$$

**Commentaires:** $\mathcal{E}_{ca}^k$ comprend un terme constant et deux termes monotones par rapport à $T_{cnf}^k$. La suite $\left(\mathcal{E}_{ca}^k\right)$ sera croissante. Toutefois, le fait que $\widetilde{\text{Pre}}$ et $\widetilde{\text{Post}}$ ne soient pas additifs[6] ne facilite pas l'écriture d'une version itérative pour les calculs de $\mathcal{E}_{ca}^k$ et $T_{cnf}^k$. La même remarque peut être faite pour les calculs de $V^k$.

$$\begin{aligned} [\![V^k]\!] \;=\; & \left(\bigcap_{(i \in \mathcal{I} \cup \mathcal{X}) \wedge (v(i)=1)} [\![i]\!]\right) \cap \left(\bigcap_{(i \in \mathcal{I} \cup \mathcal{X}) \wedge (v(i)=0)} [\![\neg i]\!]\right) \cdots \\ & \cdots \cap \left(\bigcap_{e \in \mathcal{E}_{cp}^k} [\![P_e]\!]\right) \cap \left(\bigcap_{e \in \mathcal{E}_{ca}^k} [\![\neg P_e]\!]\right) \end{aligned} \tag{5.16}$$

On peut tout de même écrire de façon plus itérative:

$$\begin{aligned} T_{cnf}^k &= T_{cnf}^{k-1} \cup f_3'(\mathcal{E}_{ca}^{k-1}) \cup f_4'(V^{k-1}) && (5.17)\\ \mathcal{E}_{ca}^k &= \mathcal{E}_{ca}^{k-1} \cup \{\downarrow X_s |\ s \in \widetilde{\text{Pre}}\ (T_{cnf}^k)\} \cup \{\uparrow X_s |\ s \in \widetilde{\text{Post}}\ (T_{cnf}^k)\} && (5.18) \end{aligned}$$

5. $f : 2^E \to 2^E$ est additive si $\forall X, X' \subseteq E : f(X \cup X') = f(X) \cup f(X')$.

6. $\forall X, X' \subseteq E : \widetilde{\text{Pre}}\ (X \cup X') \supseteq \widetilde{\text{Pre}}\ (X) \cup \widetilde{\text{Pre}}\ (X')$

$$\begin{aligned}[\![V^k]\!] &= [\![V^{k-1}]\!] \cap \left( \bigcap_{t \in T^k} \left( \bigcap_{s \in {}^{\circ}t} [\![P_{\downarrow} X_s]\!] \right) \cap \left( \bigcap_{s \in t^{\circ}} [\![P_{\uparrow} X_s]\!] \right) \right) \cdots \\ &\quad \cdots \cap \left( \bigcap_{e \in \mathcal{E}^k_{cp_{|\mathcal{P}}}} [\![\neg P_e]\!] \right) \qquad (5.19)\end{aligned}$$

Les ensembles $Ai_{ce}$ et $Ac_{ce}$ se déterminent à partir de la connaissance *finale* de $\xi^n$ et de $\mathcal{E}^n_{cp}$ :

$$Ac_{ce} = \bigcup_{s \in S|\ s \in \xi^n} A(s).Cont \qquad (5.20)$$

$$Ai_{ce} = \bigcup_{s \in S|\ \uparrow X_s \in \mathcal{E}_{cp}} A(s).Impuls \qquad (5.21)$$

**Remarques :** Il n'est pas nécessaire d'évaluer ces ensembles à chaque micro-pas car ils ne participent pas aux évolutions. Le démarrage ou le maintien des actions continues ne dépendent que de l'état final atteint. L'émission des actions impulsionnelles est une conséquence directe de la présence des fronts d'activation d'étapes $\uparrow X$.

### 5.4.2 Algorithme du Point Fixe Itéré (PFI)

Nous appelons ainsi notre algorithme car il fonctionne de manière itérative, avec à chaque itération un calcul de point fixe pour $T^k$ noté $T'$ dans cet algorithme. Il assure la détection des situations anormales : la présence d'un cycle instantané cause l'avortement du calcul ; l'absence de cycle conduit à une situation stable. Celle-ci est à son tour rejetée en cas d'indéterminisme ou de cycles de causalité.

**Algorithme PFI:**

début
[1] lire $< e_{\mathcal{I}}\,,\ v_{\mathcal{I}} >$
[2] $\xi \leftarrow X^{\theta}$
[3] $T_{cnf} \leftarrow f_1{}'()$ (Eq. 5.12)
[4] $T_{cf} \leftarrow \emptyset$
[5] $\mathcal{E}_{cp} \leftarrow e_{\mathcal{I}}$
[6] $\mathcal{E}_{ca} \leftarrow \mathcal{E}_{\mathcal{I}} \setminus e_{\mathcal{I}}$
[7] $V \leftarrow \left( \bigwedge_{(y \in \mathcal{I}) \wedge (v(y)=1)} y \right) \wedge \left( \bigwedge_{(y \in \mathcal{I}) \wedge (v(y)=0)} \neg y \right) \cdots$
$\cdots \wedge \left( \bigwedge_{s \in S} X_s \right) \wedge \left( \bigwedge_{s \notin S} \neg X_s \right)$

[8] itérer /* micro-pas */
[9] Calculer $T_{pf}$ par Eq. 5.1
[10] Calculer $T'$ par Eq. 5.5 et Eq. 5.6
[11] sortir si $T' = \emptyset$
[12] si $T' \cap T_{cf} \neq \emptyset$ alors
[13] exit (erreur instabilité)
[14] sinon
[15] $\xi \leftarrow (\xi \setminus {}^{\circ}T') \cup T'^{\circ}$
[16] $T_{cf} \leftarrow T_{cf} \cup T'$
[17] Calculer $\mathcal{E}_{cp}$ par Eq. 5.10
[18] Calculer $T_{cnf}$ par Eq. 5.17
[19] Calculer $\mathcal{E}_{ca}$ par Eq. 5.18
[20] Calculer $[\![V]\!]$ par Eq. 5.19
[21] finsi
[22] fin itérer
[23] si $T_{cf} \cup T_{cnf} = T$ alors
[24] $X^{\theta+1} \leftarrow \xi$ /* LA situation suivante */
[25] calculer $Ai_{ce}$ par Eq. 5.21
[26] calculer $Ac_{ce}$ par Eq. 5.20
[27] sinon
[28] exit (erreur causalité ou non-déterminisme)
[29] fin si
fin

**Commentaires sur l'algorithme PFI:**

Les initialisations $[1-7]$ figent une image de l'environnement [1] et calculent les parties statiquement connues des ensembles $[3-7]$. Le corps de l'itération $[8-22]$ effectue un micro-pas. La détection d'un cycle instantané $[12-13]$ provoque une sortie de l'algorithme avec signalisation de l'erreur. Ce cas est celui où une transition a été franchie lors d'un micro-pas précédent et elle redevient franchissable.

La sortie normale de la boucle [11] se fait lorsque $T'$ est vide, c'est à dire qu'il n'y a plus de transition franchissable. La stabilité est alors atteinte ou l'algorithme ne peut plus affiner la connaissance globale de la réaction. Le test [23] vérifie si les ensembles obtenus $T_{cf}$ et $T_{cnf}$ constituent une partition de $T$. Si tel est le cas, on peut affecter *la* situation suivante [24] et déterminer l'ensemble des actions à émettre dans l'instant $[25-26]$, sinon l'algorithme n'a pas pu déterminer pour certaines transitions, si elles étaient ou non franchissables. Ceci correspond aux situations d'erreur de causalité ou d'indéterminisme [29].

Notons que la boucle est exécutée au maximum cardinal($T$) fois.

### 5.4.3 Comparaison des modes d'évolution du Grafcet et du S-grafcet

Avec la définition complète du S-GRAFCET, nous sommes maintenant en mesure de comparer les modes d'évolution des deux modèles. Pour le GRAFCET, nous considérons que les évolutions sont accomplies avec recherche de stabilité (ARS), suivant l'algorithme MPI (3.3.2.2) et vues en temps nul par l'environnement.

#### 5.4.3.1 Synthèse des différences sémantiques

Au niveau du S-GRAFCET, les principales différences d'interprétation des évolutions portent sur les points suivants :

1. Dans une évolution S-GRAFCET (*réaction*), l'ordre d'apparition des fronts $\uparrow X_s$, $\downarrow X_s$ (activation et désactivation d'une étape $s$) n'intervient pas dans le choix de la situation suivante. Seule la présence et l'absence de ces événements sont significatives (conformément à la sémantique des langages synchrones). En GRAFCET, cet ordre est important. Il est susceptible de modifier plusieurs fois l'état des réceptivités et l'ensemble des transitions validées au cours d'une même évolution. Il définit également l'ordre d'émission des actions impulsionnelles dont les effets sont directement disponibles au niveau des réceptivités.

2. Dans une *réaction*, toute variable voit sa valeur figée. Ceci induit deux conséquences importantes. D'une part, les variables $X_s$ ont une sémantique différente en S-GRAFCET : elles représentent l'état de l'étape $s$ *avant* la réaction. D'autre part, les actions n'interviennent pas dans la réaction : en particulier, aucun résultat de calcul[7] n'est pris en compte pendant l'évolution.

3. En S-GRAFCET, le résultat d'un calcul externe peut être pris en compte dès l'instant logique suivant. Il peut même *définir* l'instant suivant si la terminaison du calcul est associée à un événement déclencheur élémentaire.

4. En GRAFCET, lorsqu'une étape $s$ reste active après franchissement *d'une* transition, il n'est pas précisé s'il faut émettre ou non les fronts $\uparrow X_s$ et $\downarrow X_s$. En S-GRAFCET, l'émission de ces deux événements est obligatoire. Ils peuvent d'ailleurs intervenir instantanément sur l'évolution.

5. Le modèle S-GRAFCET permet d'utiliser les événements élémentaires internes dans les réceptivités sans restriction ou risque d'ambiguïté. Il est

7. Comme les actions impulsionnelles de comptage.

ainsi possible d'utiliser l'absence d'événement (externe ou interne) au niveau des réceptivités. De plus tous ces événements élémentaires peuvent être considérés comme *gardes* de la transition ou intervenir via leur prédicat de présence.

6. Le S-GRAFCET ne retient pas l'hypothèse de "non-simultanéité des événements non-corrélés".

#### 5.4.3.2 Conditions suffisantes d'équivalence de comportement

Dans un grand nombre de cas, les modèles GRAFCET et S-GRAFCET induisent les mêmes évolutions. Pour cela les grafcets doivent respecter deux conditions suffisantes :

1. Toutes les divergences en OU doivent être disjonctives. Cette "disjonction" ne doit pas être basée sur l'hypothèse de "non-simultanéité des événements non-corrélés".
2. Les réceptivités ne dépendent que de l'environnement (événements d'entrées, variables booléennes d'entrées). En GRAFCET, il existe deux sous-cas où cette condition est rendue fausse.
   (a) les conséquences des actions impulsionnelles sont prises en compte dans les réceptivités.
   (b) Les variables d'état sont utilisées dans les réceptivités par l'intermédiaire de leur valeur ou de leurs fronts.

#### 5.4.3.3 Traduction d'un grafcet en s-grafcet

Les grafcets qui ne vérifient pas l'une des conditions précédentes ne peuvent pas être transcrits directement en S-GRAFCET. Nous pouvons cependant apporter quelques précisions pour faciliter cette traduction :

- **cas 1 :** Les grafcets qui possèdent des divergences en OU non-disjonctives, cachent souvent des erreurs de modélisation. Pour les autres, les réceptivités incriminées peuvent toujours être rendues disjonctives (quitte à ajouter une transition supplémentaire dans la divergence).

  Les disjonctions basées sur les événements, peuvent être réécrites en S-GRAFCET en donnant une priorité à l'un des événements (fig.5.21). Il faut cependant vérifier que cette priorité ne modifie pas le comportement du grafcet initial.

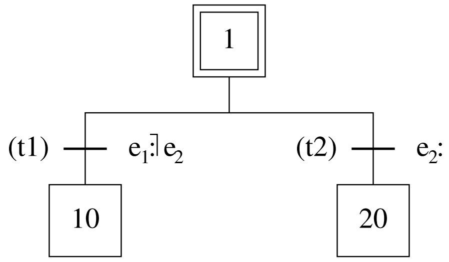



FIG. 5.21 - *Priorité de $e_2$ sur $e_1$ pour assurer la disjonction*

- **cas 2.a :** Ce problème peut être contourné en “gardant” la transition qui utilise le résultat du calcul dans sa réceptivité. Par exemple l'événement déclencheur pourrait s'appeler “donnée disponible”.

- **cas 2.b :** Les grafcets qui ne respectent pas cette hypothèse sont les plus difficiles à traduire car les variables $X_s$ n'ont pas la même signification dans les deux modèles. Nous pouvons déjà tenter d'éliminer ces variables en ajoutant des convergences en ET pour mieux spécifier les synchronisations. Ensuite nous pouvons remplacer les réceptivités de type “$X_s$” par “: $X_s \vee \uparrow X_s$”. Mais cette réécriture ne permet pas toujours de conserver le comportement du grafcet initial.

### 5.4.4 Principe de la compilation d'un s-grafcet

Dans ce chapitre, nous ne décrivons bien-sûr pas tous les algorithmes mis en œuvre dans le compilateur car la plupart concernent des changements de représentation interne de données (décodage, conversion de listes en tableaux, suppression des actions redondantes ...). Par contre les trois algorithmes qui sont maintenant présentés caractérisent le compilateur.

#### 5.4.4.1 Recherche d'accessibilité

Cette recherche consiste à trouver pour chaque étape l'ensemble des étapes accessibles par transition instantanée. Pour chaque couple (étape initiale, étape finale) trouvé, l'algorithme détermine de plus la fonction booléenne de transition (sous forme de BDD). Cette fonction est très importante par la suite pour déterminer le véritable ensemble des entrées booléennes significatives, dès que l'on connaît partiellement la valeur de certaines entrées. Cet ensemble est déjà utilisé au niveau des valuations dans l'algorithme PFI. Il sera utilisé dans une prochaine version du compilateur pour réduire le nombre d'appels inutiles de l'algorithme PFI.

```
  /* premiere etape : initialisation de la matrice accessibilite */
  /* conditions de transition instantanee de l'etape i vers l'etape j */
  pour toute etape i de S (S ensemble des etapes)
    pour toute etape j de S
      accessibilite[i,j] ← FAUX
    fin pour
  fin pour

  /* seconde etape : gestion des transitions directes */
  /* les gardes n'entrent pas en compte */
  /* pour toutes les etapes de depart */
  pour toute etape e de S
    pour toute transition t de post(e) (divergence en OU)
      pour toute etape esuiv de post(t) (divergence en ET)
        accessibilite[e,esuiv] ← R(t).cond
      fin pour
    fin pour
  fin pour

  /* troisieme etape: etude des transitions accessibles par transitivite */
  /* les gardes doivent etre prises en compte */
  /* pour toutes les etapes de depart */
  faire
  termine ← VRAI; /* a priori */
  pour toute etape i de S
    pour toute etape j de S
      si accessibilite[i,j] n'est pas tautologiquement fausse alors
        pour toute transition t de post(j) (divergence en OU)
          si la transition t n'est pas gardee alors
            chemin ← accessibilite[i,j] ∧ R(t).cond
            pour toute etape jsuiv de post(t)
              nouv_chemin ← chemin ∨ accessibilite[i,jsuiv]
              /* la transition de i vers jsuiv */
              /* peut deja etre affectee par un autre chemin */
              si nouv_chemin est different de accessibilite[i,jsuiv] alors
                accessibilite[i,jsuiv] ← nouv_chemin
                termine ← FAUX
              fin si
            fin pour
          fin si
        fin pour
      fin si
    fin pour
  fin pour
jusqu'a termine
```

Il faut remarquer que cet algorithme est statique et ne dépend que de la topologie du réseau. Il fournit en réalité un sur-ensemble des étapes effectivement atteintes. En effet il n'a qu'une *vision locale* des évolutions instantanées (non prise en compte des convergences en ET et des évolutions des autres étapes). Seul l'algorithme PFI possède cette *vision globale* et détermine l'ensemble réel des étapes atteintes.

Pour montrer l'évolution de la matrice d'accessibilité, considérons de nouveau la figure 5.17. Sur cet exemple, l'algorithme converge en 4 itérations :

première et seconde étape : franchissement simple

étapes accessibles

| | | | | |
|---|---|---|---|---|
| étapes | 0 | $\neg b$ | $a \wedge b$ | 0 |
| | 0 | 0 | $a$ | 0 |
| | 0 | 0 | 0 | 1 |
| | $\neg a$ | 0 | 0 | 0 |

troisième étape, première itération :

étapes accessibles

| | | | | |
|---|---|---|---|---|
| étapes | 0 | $\neg b$ | $a$ | $a$ |
| | 0 | 0 | $a$ | $a$ |
| | $\neg a$ | 0 | 0 | 1 |
| | $\neg a$ | $\neg a \wedge \neg b$ | 0 | 0 |

troisième étape, seconde itération :

étapes accessibles

| | | | | |
|---|---|---|---|---|
| étapes | 0 | $\neg b$ | $a$ | $a$ |
| | 0 | 0 | $a$ | $a$ |
| | $\neg a$ | $\neg a \wedge \neg b$ | 0 | 1 |
| | $\neg a$ | $\neg a \wedge \neg b$ | 0 | 0 |

troisième étape, troisième itération :

étapes accessibles

| | | | | |
|---|---|---|---|---|
| étapes | 0 | $\neg b$ | $a$ | $a$ |
| | 0 | 0 | $a$ | $a$ |
| | $\neg a$ | $\neg a \wedge \neg b$ | 0 | 1 |
| | $\neg a$ | $\neg a \wedge \neg b$ | 0 | 0 |

**Remarque :** Conformément au résultat du premier algorithme d'accessibilité, l'étape 3 est accessible à partir de 1 *seulement si* $a$ est vraie.

#### 5.4.4.2 Implantation de l'algorithme PFI

Dans un grafcet, l'ensemble des étapes actives à un instant donné définit *l'état du système*. À partir de cet état, le compilateur détermine l'ensemble des *états suivants* en faisant varier les entrées du système. Pour cela le compilateur utilise l'algorithme PFI qui fournit *l'état suivant* connaissant *l'état actuel* et la valeur des entrées. Remarquons que la connaissance des entrées peut être partielle, pourvu qu'elle inclut l'ensemble des entrées devenues réellement significatives (voir algorithme précédent).

Cette capacité est encore mal prise en compte par le compilateur qui utilise l'algorithme PFI sur des cas inutiles. Des aménagements ont été apportés au niveau des valuations des réceptivités par l'utilisation intensive et la réduction au fur et à mesure de BDD [Bry86]. Dès que les BDDS associés à une transition validée sont réduits à la constante $True$ ou $False$, l'algorithme PFI en tient compte dans le micro-instant courant pour construire itérativement les ensembles $T^k_{pf}$ (Eq. 5.1) et $T^k_{cnf}$ (Eq. 5.17).

#### 5.4.4.3 Optimisation et génération de l'automate équivalent

La recherche de tous les états suivants, aboutit à la constitution d'un automate fini déterministe. Cet automate est ensuite décrit en OC pour être installé par exemple sur une machine cible. Cette description influence énormément la qualité du code généré. Prenons par exemple l'automate de la figure 5.22.

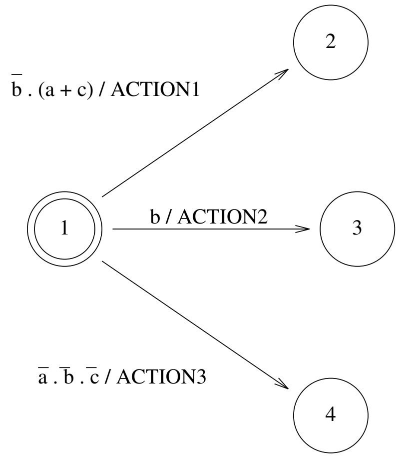



FIG. 5.22 - *Exemple d'automate*

Par la structure "IF THEN ELSE" des BDDS et l'ordre (a,b,c) des variables, cet automate peut facilement s'implanter sous la forme :

```
  state 1:
  if a then
    if ! b then
      emit ACTION1 goto state 2
    endif
  else
    if ! b then
      if c then
emit ACTION1 goto state 2
      endif
    endif
  endif
  if b then
    emit ACTION2 goto state 3
  endif
  if ! a then
    if ! b then
      if ! c then
emit ACTION3 goto 4
      endif
    endif
  endif
  end state
  ...
```

Cette description est “naïve” pour au moins deux raisons :

- L'ordre des variables n'est pas adapté à l'évaluation de la première transition. Il aurait fallu prendre comme ordre d'évaluation (b,a,c) ou (b,c,a). Remarquons d'ailleurs qu'un ordre correct pour une transition n'est pas forcément correct pour une autre.
- l'évaluation de la troisième transition se fait en 6 tests au lieu des 3 attendus. Sur un système réel, le franchissement de cette transition prendrait déjà le double de temps (alors que l'exemple reste très simple). Il faudrait en fait factoriser globalement les tests au niveau de cet état.

À partir de ces deux constatations, est née l'heuristique d'optimisation employée par le compilateur : Connaissant l'ensemble des transitions quittant l'état considéré, il compte pour chaque variable le nombre de fois ou cette dernière est significative. Il trie ensuite les variables suivant ce critère. Il évalue ensuite chaque variable dans ce nouvel ordre et simplifie les BDD transitions au fur et à mesure. Ainsi, il construit pour chaque état, un arbre d'évaluation dont chaque feuille est l'état suivant et la liste des actions à émettre. Sur l'exemple précédent, la compilation aboutit à :

```
  state 1 :
  if b then
    emit ACTION 2 goto state 3
  else
    if a then
      emit ACTION 1 goto state 2
    else
      if c then
emit ACTION 1 goto state 2
      else
emit ACTION 3 goto state 4
      endif
    endif
  endif
  end state
  ...
```

**Remarque :**

Les actions communes à plusieurs transitions pourraient être remontées des feuilles vers les nœuds communs. Cette optimisation mémoire n'est pas faite par le compilateur car elle n'accélère pas la vitesse du code.

#### 5.4.4.4 Génération d'un automate booléen équivalent

Les résultats fournis par l'algorithme PFI, peuvent aussi être intégrés dans un automate booléen constitué d'un ensemble d'équations booléennes. celui-ci peut se définir par :

$$f_{init}(v_1, \ldots, v_n) = 1,$$

$$V' = \begin{pmatrix} \vdots \\ v'_i \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ f_i(v_1, \ldots v_n, e_1, \ldots e_m) \\ \vdots \end{pmatrix}$$

avec : $n \in \mathbb{N}^*$ : le nombre de variables booléennes d'états
$m \in \mathbb{N}^*$ : le nombre d'entrées
$V \in \mathbb{B}^n$ : le vecteur d'état booléen
$e_k$ : la $k^{ième}$ entrée / $\forall k \in 1 \ldots m, e_k \in \mathbb{B}$
$f_{init}$ : la fonction de caractérisation de l'état initial
$f_i$ : la $i^{ième}$ fonction booléenne de transfert /
$\forall i \in 1 \ldots n, f_i : \mathbb{B}^{n+m} \rightarrow \mathbb{B}$

Cette représentation des machines d'état est très compacte. Pour $n$ variables, elle permet de représenter jusqu'à $2^n$ états. On remarque de plus la très forte similitude avec le GRAFCET. Nous avons en effet vu que la *situation* d'un grafcet est définie par l'ensemble des étapes actives. Nous pouvons donc associer à chaque étape une et une seule variable d'état et utiliser l'ensemble de toutes les réactions autorisées du GRAFCET (noté $\mathcal{R}$) pour constituer le corps des fonctions $f_i$.

Chaque réaction $r$ est un triplet $< X^\theta, X^{\theta+1}, V_i >$. Pour chaque variable d'état $X_i$, nous pouvons lui associer le cube $F_i^r$ qui vaut 0 ou 1 suivant que la variable d'état est maintenue ou forcée à 0 ou à 1. Pour chaque $X_i$, discriminons $\mathcal{R}$ en deux ensembles $\mathcal{R}_{i0}$ (resp. $\mathcal{R}_{i1}$) qui force la variable à 0 (resp. à 1).

Nous pouvons maintenant construire les deux fonctions globales de forçage $F_{i0}$ et $F_{i1}$ tel que:

$$F_{i0} = \bigvee_{r \in \mathcal{R}_{i0}} F_{i\,0}^{r} \tag{5.22}$$

$$F_{i1} = \bigvee_{r \in \mathcal{R}_{i1}} F_{i\,1}^{r} \tag{5.23}$$

**Remarques:**

- $F_{i0} \wedge F_{i1} \equiv 0$ par construction,
- Si $\mathcal{R}$ couvre tout l'ensemble des réactions possibles du GRAFCET, alors $F_{i0} \vee F_{i1} \equiv 1$ et la fonction vectorielle de transfert $(f_i)$ est entièrement spécifiée: $(f_i) = (F_{i1})$ ou $(f_i) = (\neg F_{i0})$.
- Dans le cas général $(f_i)$ est incomplètement spécifiée. Toute fonction impliquant $(F_{i1})$ et $(\neg F_{i0})$ sera solution.

Nous profitons du cas général pour optimiser les fonctions $f_i$. L'heuristique choisie s'appuie sur la représentation interne sous forme de BDD. Elle consiste à supprimer les variables une par une de chaque $f_i$ en respectant les deux implications précédentes. Sur un diagramme de Karnaugh, cela reviendrait à former des îlots de 0 et de 1 disjonctifs de plus en plus grand.

La génération d'automate booléen de comportement équivalent sur l'ensemble des états accessibles offre de nombreux avantages. Il permet l'obtention d'un code beaucoup plus compact en taille qu'un automate fini équivalent (par contre le temps de transition est plus long). Il conserve de plus le parallélisme du GRAFCET et facilite sa répartition. Enfin la bijection complète entre *variables d'états* et *étape* permet de manipuler ces étapes grâce à de puissants outils de preuves sur les automates booléens comme BAC [Hal94]. Au chapitre 6.3, nous décrirons d'ailleurs la représentation adoptée pour un GRAFCET compilé.

### 5.4.5 Le compilateur G2OC

#### 5.4.5.1 Présentation

Le compilateur de grafcet G2OC écrit en C++, met en œuvre tous les principes et les algorithmes de compilation présentés dans le chapitre 5.4. Il est actuellement opérationnel et permet de générer soit un automate d'état au format OC, soit un automate booléen pour l'outil de preuve BAC. Il utilise bien sûr l'interprétation S-GRAFCET en fonctionnement normal. Il peut cependant utiliser l'interprétation GRAFCET classique définie dans la dernière norme [AFN93] et basée sur la double échelle de temps. De ce fait, le couple (G2OC,BAC) permet d'établir des preuves formelles de comportement sur tout *grafcet normalisé*.

#### 5.4.5.2 Chaîne de compilation

Ce compilateur de GRAFCET s'inscrit dans la chaîne de compilation générale d'ESTEREL et de LUSTRE comme le montre la figure 5.23. En particulier le code généré (OC5) est compatible avec les traducteurs OCDEBUG, OCC, OCC++. Par contre ce code n'est pas compatible avec xsimul (simulateur ESTEREL), mais possède son propre simulateur: EG7 [Pas94].

Le fichier d'entrée (format GT) peut être généré à partir de l'éditeur EG7 ou de l'éditeur EDGE [Lep94] développé au LIMI (Université de Brest).

Au niveau preuves, G2OC détecte tout grafcet *instable*. Comme nous l'avons vu au § 5.4.4.4, il permet aussi de générer l'automate sous forme d'équations booléennes de changement d'état (automate booléen), tout en conservant la notion d'*étape* GRAFCET. Ce code est compréhensible par l'analyseur BAC. Nous verrons au chapitre 6 que celui-ci peut vérifier des propriétés logiques de comportements sur le grafcet initial de cette manière.

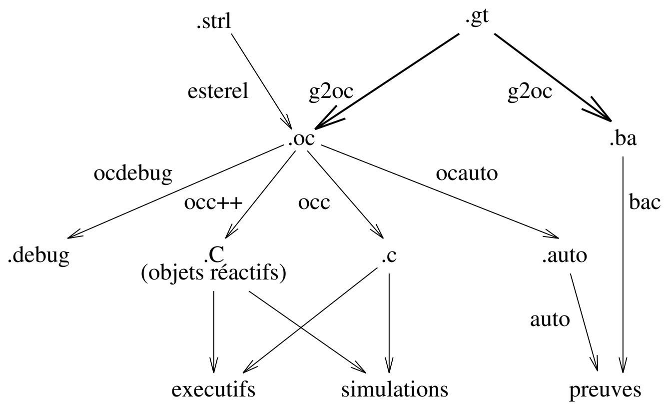



FIG. 5.23 - *Chaîne de compilation de Grafcet*

## 5.5 Conclusion

Les deux sémantiques comportementales que nous venons de présenter, peuvent être qualifiées de "effectives" car elles permettent la compilation du modèle S-GRAFCET. Elles sont en fait complémentaires.

La modélisation du comportement par ESTEREL permet de lier les deux formalismes au sein d'une même application, en conservant les avantages syntaxiques de chacun et les avantages sémantiques d'ESTEREL. Elle assure ensuite la détection de tout grafcet instable. Le code généré par le compilateur G2E peut directement intégrer la chaîne de compilation d'ESTEREL et bénéficier d'outils de preuves et de simulation. La traduction permet enfin une génération de code exécutable efficace sous forme d'automate d'états qui sera ensuite porté sur une machine cible.

L'approche par point fixe itéré offre d'autres avantages. Elle spécifie complètement la notion de stabilité, lève les ambiguïtés sémantiques propres au modèle GRAFCET. Elle donne également des conditions suffisantes pour assurer l'équivalence de comportement entre GRAFCET classique et S-GRAFCET. Le compilateur associé (G2OC) permet de générer un automate d'états optimisé au format OC. Cet automate peut donc intégrer la plate-forme synchrone commune ESTEREL-LUSTRE et bénéficier de ces outils. Le compilateur peut également générer un automate booléen tout en conservant la notion d'*étape* GRAFCET. Nous verrons d'ailleurs au chapitre 6 les avantages que cela apporte au niveau preuves de comportement. Notons enfin que G2OC est aussi capable de compiler un grafcet avec l'interprétation classique définie dans la norme.

# Chapitre 6

# Preuves formelles de comportement en Grafcet

## 6.1 Introduction

Dans la conception d'un système réactif de commande, la phase de *validation* est très importante. Elle permet de vérifier le comportement attendu d'un système soumis à des contraintes normales ou exceptionnelles. Elle est de plus chargée de mesurer la *robustesse* d'un système face à des événements imprévus ou à des pannes. Cette dernière qualité est souvent essentielle pour la sécurité des usagers.

Deux grandes méthodes de validation existent : la vérification formelle et la simulation. La seconde méthode est actuellement la plus employée (en particulier en Grafcet). Elle consiste à *observer* le système lorsque celui-ci est soumis à différentes séquences d'entrée. Dans cet esprit, nous avons réalisé dans notre équipe, l'éditeur de grafcet EG7 [Pas94] qui appelle le compilateur G2OC pour simuler un grafcet. Mais, la simulation atteint rapidement ses limites dès que le système a un nombre important d'états et d'entrées. Pour apporter la preuve du bon fonctionnement, il faut a priori essayer toutes les combinaisons possibles des entrées sur tous les états accessibles. Lorsque la robustesse est une notion importante, il faut aussi intégrer les pannes et leurs conséquences ! Enfin les systèmes temps-réels ont des contraintes temporelles spécifiques qui sont difficiles à représenter.

C'est pourquoi nous nous intéresserons exclusivement aux *méthodes de preuves formelles de comportement* applicables au GRAFCET. Nous nous restreindrons aux preuves *logiques* de comportement. En effet la vérification effective des contraintes temporelles réelles, est un problème d'implémentation fondamentalement lié à la définition de *machine d'exécution* présentée au chapitre 2.3. Leur prise en compte sort du cadre de cette étude.

## 6.2 Avancées dans le domaine

### 6.2.1 Propriétés recherchées en Grafcet

Bon nombre de propriétés de comportement peuvent être vérifiées en Grafcet. Elles s'intègrent aux propriétés spécifiques des machines d'états. Celles-ci se regroupent en trois familles essentielles, notées *déterminisme* (*Predictability*), *sûreté* (*Safety*) et *vivacité* (*Liveness*) dans la littérature.

- Le qualificatif "déterministe" est réservé aux systèmes qui bannissent le hasard de leur évolution.
- Sous l'appellation de "safety" (sûreté), les concepteurs rassemblent toutes les propriétés qui expriment l'immunité aux mauvais fonctionnements. Parmi elles, nous pouvons citer :
  - Le grafcet ne possède pas d'instabilité.
  - Il n'existe pas de situation de blocage total ou partiel.
  - Les sorties incompatibles ne sont jamais émises en même temps.
  - Il n'existe pas de séquence d'actions susceptible de conduire à un accident corporel ou à une dégradation de matériel.
  - Le système assure la sécurité des personnes et son intégrité, même en cas de dysfonctionnement ou de situation externe imprévue.
- Le vocable "liveness" est réservé aux bonnes propriétés qui doivent se révéler une ou plusieurs fois dans l'avenir. Parmi celles-ci, nous pouvons citer en GRAFCET :
  - Aucune étape n'est totalement inactive (toutes les actions sont potentiellement émissibles).
  - Une situation donnée est toujours accessible (par exemple l'état initial).
  - Telle action sera émise "un jour", quelle que soit l'histoire d'entrée.
  - Telle séquence d'entrée conduit inévitablement à cette situation.

**Remarques :**

En GRAFCET normalisé, les évolutions se déterminent en *temps interne* et peuvent faire apparaître des invariants. Cependant ces propriétés sont difficilement transposables en temps *externe* car les deux notions de temps sont indépendantes. Le S-GRAFCET ne connait pas ce problème car il ne possède pas cette double échelle de temps.

L'approche synchrone (GRAFCET et langages) assure la prévisibilité d'une application. En particulier, ESTEREL sait générer un automate déterministe de comportement équivalent. En GRAFCET, le déterminisme est imposé dans la norme. Elle a conduit à la définition des algorithmes de *recherche de l'état stable suivant*. Ce problème a donc été résolu *de fait* dans le chapitre 5.

Que ce soit en GRAFCET normalisé, ou en S-GRAFCET, le compilateur G2OC détecte tout grafcet instable et indique l'ensemble des transitions qui participent à cette instabilité. Si le grafcet est préalablement exprimé en ESTEREL, la stabilité est étudiée par le compilateur ESTEREL lui-même.

### 6.2.2 Exemples de propriétés vérifiables par la recherche des états accessibles

Ces preuves nécessitent d'abord l'établissement d'un automate fini ou d'un système de transitions qui se réfère à l'ensemble des états accessibles. Les travaux existants [Lep94, Rou94] utilisent l'outil de preuves formelles MEC [ABC94] pour vérifier des propriétés de "safety" et de "liveness".

Ainsi Leparc attache à chaque état une liste de paramètres (en particulier l'état $e_n$ actif ou inactif de chaque étape $E_n$). Il peut montrer, par exemple, que $n$ étapes ne peuvent jamais être activées simultanément. Pour cela, il utilise les opérateurs ensemblistes tels que : $\bigcup$, $\bigcap$ et de relations état-transition $src()$, $tgt()$, $rsrc()$, $rtgt()$ de MEC [ABC94, Lep94].

```
etats_inactives := * - (e1 \/ ... \/ en);
etats_actives := ( e1 /\ ... \/ en);
trans_origine := rsrc(etats_inactives);
trans_but := rtgt(etats_actives);
trans_resultat := trans_origine /\ trans_but;
```

Il détermine d'abord l'ensemble des états où toutes les étapes GRAFCET sont inactives et l'ensemble des états où toutes les étapes sont actives. Ces deux ensembles permettent d'isoler les transitions successeur de l'un et prédécesseur de l'autre. Si l'ensemble de transitions résultat est vide la propriété est vérifiée.

De même, Roussel utilise MEC sur les automates issus de la recherche des états accessibles. Comme le système de transition généré peut être très grand, l'auteur préconise de réduire la taille de l'automate dès sa production en ne prenant en compte que les critères d'observation voulus par l'utilisateur.

Au niveau des preuves GRAFCET, il note par exemple que des séquences d'actions peuvent devenir dangereuses pour le système. La figure 6.1 est extraite de sa thèse [Rou94]. Elle montre un exemple de deux vérins bistables agissant dans un même espace. Ici, la non-simultanéité des actions n'est pas une condition suffisante de sûreté. En effet les deux vérins restent en position sortie après disparition de l'ordre SORTIE_VERIN (position stable). Donc toute séquence de la forme :

```
sortir_verin1, ( non rentrer_verin1)*, sortir_verin2, ...
```

ou

```
sortir_verin2, ( non rentrer_verin2)*, sortir_verin1, ...
```

est destructive car elle engendre une collision des deux vérins.

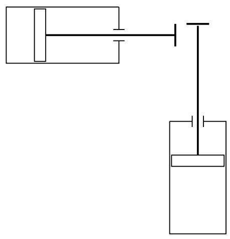


FIG. 6.1 - *Interaction entre actionneurs*

Pour vérifier l'absence de telles séquences, l'auteur étudie les différentes configurations des vérins et manipule grâce aux opérateurs de MEC les ensembles associés.

De même, l'intégration du GRAFCET dans la plate-forme synchrone, permet l'utilisation de l'outil de preuves AUTO [Ver87, Roy90]. Les automates OC issus de la compilation directe du GRAFCET, doivent être préalablement exprimés dans le format d'entrée de AUTO (SA) avant d'être traités. Cette fonctionnalité est fournie par l'environnement ESTEREL.

L'idée principale est de restreindre l'automate (abstraction) suivant un critère d'observation séquentielle. En AUTO, ce critère prend la forme d'une expression régulière.

- Pour vérifier une propriété de potentialité de franchissement, il suffit de réduire l'automate avec une séquence typique des entrées et des sorties attendues, puis de vérifier que l'automate final n'est pas vide.

- Pour vérifier une propriété de sûreté, il suffit de réduire l'automate avec la séquence contre-exemple, puis de vérifier que l'automate final est vide.

Sur l'exemple précédent, la collision éventuelle des vérins s'exprimera par le critère suivant :

```
parse-criterion COLLISION=
A = ((not /sortir_verin1!) // (not /sortir_verin2!))*:
    /sortir_verin1!:(not /rentrer_verin1!)*: /sortir_verin2!:true*
    +
    ((not /sortir_verin1!) // (not /sortir_verin2!))*:
    /sortir_verin2!: (not /rentrer_verin2)*: /sortir_verin1!:true*;
    +
    (not (/sortir_verin1! // /sortir_verin2!))*:
    (/sortir_verin1! // /sortir_verin2!): true*
```

Cette définition suit la syntaxe :

| | |
|---|---|
| "x!" | : émission du signal "x", |
| "/x!" | : ensemble des processus qui émettent "x", |
| "*" | : répétition du processus un nombre arbitraire de fois (0 compris), |
| ":" | : séquencement de processus, |
| "//" | : processus parallèles, |
| "true" | : n'importe quel processus, |
| "not" | : complément sur les ensembles, |
| "+" | : union de séquences. |

L'abstraction de l'automate `verins` suivant le critère `COLLISION` aboutit à l'automate `AUTOM_COLLISION` dont les états sont accessibles par le chemin `A`.

```
AUTOM_COLLISION = abstract(VERINS,COLLISION);
```

Si l'automate est vide, la collision des vérins n'est jamais réalisée.

### 6.2.3 Conclusion sur cette approche

L'étude de l'automate (image de toutes les situations accessibles) est réalisable par des outils formels tels que AUTO ou MEC. En particulier, MEC possède une grande puissance d'expression car il permet de raisonner aussi bien sur les transitions que sur les états. De plus il autorise les équations de points fixes sur des ensembles. Par contre la puissance d'expression engendre une perte de lisibilité. Ce même phénomène peut être reproché à Auto dont la syntaxe n'est pas très conviviale. En fait, ces outils généraux de preuves sur les systèmes d'états n'ont pas été conçus pour répondre aux problèmes spécifiques du GRAFCET, ce qui limite leur efficacité.

Plus gênant, ces outils imposent la mise à plat de tous les comportements possibles. Ceci conduit souvent à une explosion combinatoire qui limite rapidement la taille des applications réellement vérifiables.

### 6.2.4 Autre méthode de preuves

Les travaux de Leparc sur la traduction de GRAFCET vers SIGNAL, permettent d'utiliser l'outil de preuve dédié SIGALI. Les équations obtenues se présentent sous la forme de polynômes sur le corps $\mathbb{Z}/3\mathbb{Z}$. Cette forme évite l'explosion des états de l'approche précédente.

Remarquons par contre, que ces équations expriment la dynamique interne du grafcet. Elles ne permettent pas de connaître directement l'état stable suivant. Elles spécifient en fait tous les états internes dont une partie seulement apparaîtra en temps externe. En particulier l'existence de ces équations ne signifient pas que le grafcet d'origine soit *stable* dans toutes les situations. En ce sens cette méthode de preuves est moins intéressante que la précédente qui travaille directement sur le résultat issu de la compilation du grafcet.

Avec cette approche, toute preuve commence par une projection des polynômes sur les états stables. Il existe pour cela une variable d'état particulière qui passe à 1 lorsque le système ne peut plus évoluer. La propriété à prouver est ensuite exprimée sous forme d'équations et le système complet est résolu par SIGALI.

Celui-ci manipule formellement les expressions en utilisant des théorèmes mathématiques sur les polynômes. Il permet la prise en compte de nouvelles fonctions dont certaines peuvent même être définies par point fixe.

## 6.3 L'outil de preuves BAC

### 6.3.1 Présentation

BAC (Boolean Automaton Checker) [Hal94] est un outil de vérification symbolique de comportements sur les automates booléens développé par N. Halbwachs à l'IMAG. Il répond aux besoins de son concepteur qui voulait prouver certaines propriétés logiques sur les automates issus de LUSTRE [HCRP91]. Il a développé la chaîne de compilation décrite figure 6.2. On remarquera en particulier le traducteur "ec2ba" qui permet de transcrire un fichier LUSTRE compilé en fichier compréhensible par BAC.

Un automate booléen est une machine d'états finis évoluant sur occurrence d'événements. Il se caractérise par sa représentation condensée des états sous

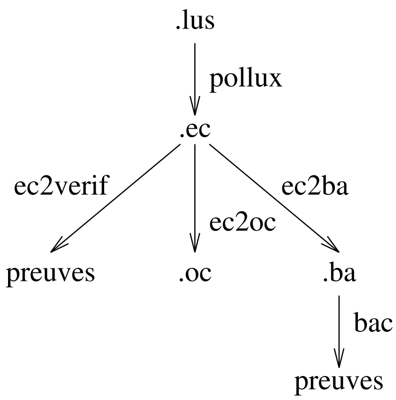



FIG. 6.2 - *Chaîne de compilation de Lustre*

forme de vecteurs de variables booléennes et des transitions sous forme d'une matrice booléenne de transfert. Nous avons vu au chapitre 5.4.4.4 que cette représentation s'apparente à celle du GRAFCET où l'état du système est défini par l'ensemble des étapes actives.

Outre la description de l'automate booléen, BAC peut prendre en compte des propriétés (appelées "assertions") qui doivent rester vraies quel que soit l'état atteint et qui permettent de réduire l'ensemble des comportements possibles de l'automate. Enfin BAC peut vérifier si l'automate muni de ces "assertions" garantit certaines propriétés logiques (appelées "invariants"). Si cette vérification échoue, il indique alors quelle (plus courte) histoire des entrées conduit à la violation des propriétés.

### 6.3.2 Modélisation du comportement d'un Grafcet par un Automate Booléen

Pour bénéficier des avantages de BAC, nous avons développé un générateur d'automate booléen qui est maintenant intégré dans G2OC. Cet automate a un comportement équivalent au grafcet d'origine. A priori BAC est plus limité dans ses ambitions que MEC ou AUTO puisqu'il s'intéresse uniquement à la vérification des propriétés de sûreté. Nous montrerons donc dans le prochain paragraphe, comment BAC peut exprimer ces preuves de comportement.

Cette représentation offre des avantages majeurs :

- Un automate booléen est beaucoup plus compact qu'un automate fini de comportement équivalent. Il évite d'énumérer les états. BAC est donc susceptible de prendre en compte des systèmes beaucoup plus conséquents. L'idée de générer un système d'équations a déjà été exploitée avec Signal et Sigali (§ 6.2.4). Mais la ressemblance avec nos travaux s'arrête ici, car nos équations caractérisent *uniquement* les états stables du grafcet résultat de la recherche par point fixe.

- De son côté, BAC n'explore pas tous les cas possibles pour vérifier les propriétés. Il manipule *formellement* les expressions de changement d'états.

- L'automate booléen est directement généré à partir de la recherche de *l'état suivant* (PFI ou MPI) présenté au paragraphe 5.4.2. Il n'est donc pas nécessaire de construire l'automate fini correspondant.

  Ce dernier point doit être nuancé : générer d'un automate booléen ne permet pas d'éviter au niveau de G2OC, l'explosion du nombre d'états accessibles. En effet un état nouvellement trouvé, peut à son tour conduire à la découverte d'autres états via l'algorithme PFI. ou MPI. Ces états devront à leur tour être étudiés *séparément.*

- Dans le code généré, nous pouvons conserver la *notion d'étape Grafcet* et toutes les propriétés logiques qui s'y rapportent. En fait chaque variable d'état correspond à *une et une seule* étape GRAFCET. En conséquence la notion *d'invariant* de BAC convient très bien pour exprimer, puis vérifier toute propriété relative aux étapes. Comme BAC n'impose aucun nom de variables d'états, nous les avons nommées $X_k$ pour faciliter l'interprétation des résultats.

Pour illustrer notre propos, considérons le grafcet Fig 6.3 suivant :

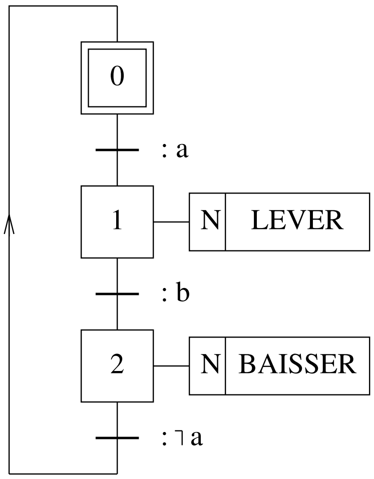



FIG. 6.3 - *Grafcet simple*

Ce grafcet est compilé par G2OC qui génère l'automate booléen qui suit :

```
state X0, X1, X2;
inputs b, a;
locals X0next, X1next, X2next, LEVER, BAISSER;

initial X0 and (not X1 and (not X2));

transition
X0'= X0next;
X1'= X1next;
X2'= X2next;

definitions
X0next= X1 and (b and (not a)) or not X1 and (not a);
X1next= X1 and (not X2 and (not b)) or not X1 and
        (not X2 and (not b and a));
X2next= X2 and a or not X2 and (b and a);

LEVER= X0 and (not X1 and (not X2 and (not b and a))) or
       not X0 and (X1 and (not X2 and (not b)));
BAISSER= X0 and (not X1 and (not X2 and (b and a))) or
       not X0 and (X1 and (not X2 and (b and a)) or
       not X1 and (X2 and a));
```

Dans le fichier généré, nous pouvons reconnaître:

- les variables d'états : $X0$, $X1$, $X2$,
- les entrées : a, b,
- les variables locales (nouvel état et actions) : $X0next$, $X1next$, $X2next$, LEVER, BAISSER. Les deux derniers signaux sont définies comme "locals" au lieu de "outputs" car les concepteurs de BAC nous ont pour l'instant déconseillé d'utiliser les sorties,
- l'état initial du système : dans notre cas, seule l'étape 0 est active,
- les équations de transitions : sous la forme : $Xk^+ = Xknext$
- les définitions des états suivants : sous la forme : $Xknext = f_k(X0, X1, X2, a, b)$ (machine de Mealy classique),
- les équations d'émission d'actions : sous la forme: $An = f_n(X0, X1, X2, a, b)$. Rappelons que les actions continues Grafcet sont en réalité définies à partir de la *nouvelle* situation stable soit: $An = f_n(X0next, X1next, X2next, a, b)$. Le compilateur G2OC se charge de réécrire ces équations sous forme de machine de Mealy.

Le fichier généré peut être ensuite complété par des propriétés logiques à vérifier, avant d'être traité par BAC.

### 6.3.3 Preuves de comportement directement réalisables

#### 6.3.3.1 Accessibilité et fonctionnement dégradé

La recherche d'accessibilité consiste à trouver l'ensemble des états possibles du grafcet en parcourant a priori toutes les "histoires" d'entrées. En théorie, un grafcet composé de $n$ étapes aura au plus $2^n$ états possibles. En pratique le nombre d'états est beaucoup plus faible à cause des synchronisations et des corrélations d'entrées. De plus l'environnement induit un certain nombre de contraintes physiques sur les entrées qui réduit encore la liberté d'évolution du grafcet. Ces contraintes peuvent être facilement intégrées dans BAC grâce à la notion *d'assertion.*

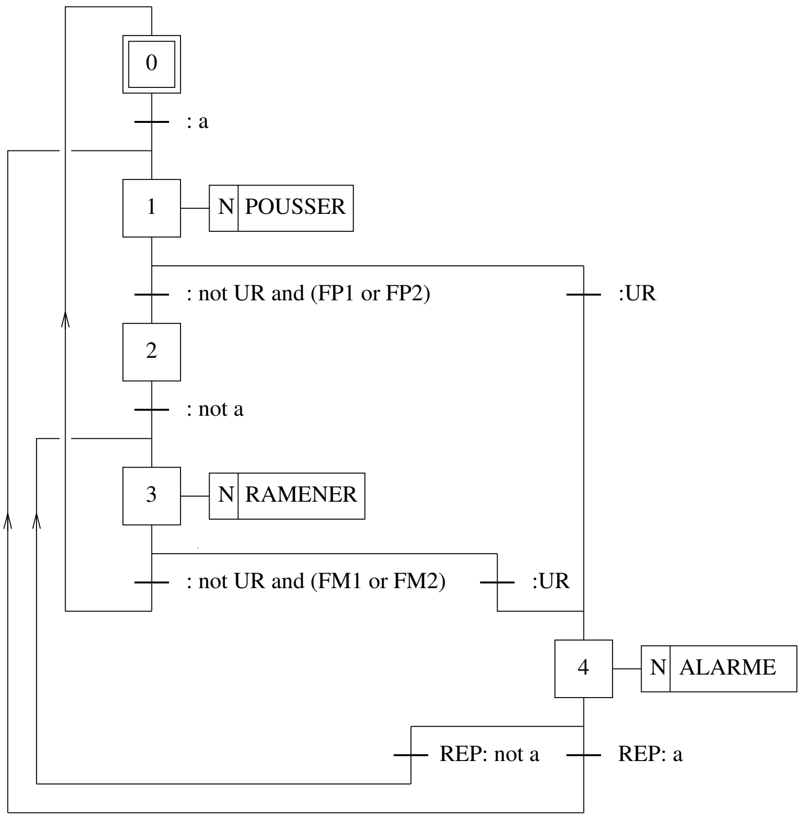



FIG. 6.4 - *Gestion d'un vérin*

Considérons par exemple le grafcet Fig.6.4 qui assure le contrôle d'un vérin. L'arrêt du vérin est commandé par deux capteurs de fin de course plus (FP1,FP2) et deux capteurs de fin de course moins (FM1,FM2). Ces capteurs sont dupliqués pour augmenter la sûreté de fonctionnement. Le système est également pourvu d'un arrêt d'urgence qui le rend totalement inactif jusqu'a l'occurrence d'un signal de reprise (REP).

Nous pouvons nous intéresser à l'ensemble des états accessibles lorsqu'une panne de capteur dégrade le fonctionnement du système :

### Rupture du capteur FP1 :

Dans ce cas FP1 prend toujours l'état logique 0. Ceci peut être facilement pris en compte par BAC grâce à la notion d'assertion :

```
assertion FP1 eq 0;
```

ou tout simplement :

```
assertion not FP1;
```

Les *assertions* sont des propriétés qui restent vraies sur tous les états construits à partir de l'état initial. Avec l'assertion précédente, BAC donne comme résultat :

```
NORMALIZATION
  No combinatorial loop
REACHABLE STATES
  Reachable states computed after 3 steps
   not X0 and  not X1 and  not X2 and  not X3 and X4 and  not FP1 or
   not X0 and  not X1 and  not X2 and X3 and  not X4 and  not FP1 or
   not X0 and  not X1 and X2 and  not X3 and  not X4 and  not FP1 or
   not X0 and X1 and  not X2 and  not X3 and  not X4 and  not FP1 or
  X0 and  not X1 and  not X2 and  not X3 and  not X4 and  not FP1
```

Chaque ligne de l'équation indique une configuration possible des étapes. Par exemple la première ligne montre que seule l'étape 4 est active. L'ensemble des états accessibles est donc : $\{\{X0\}, \{X1\}, \{X2\}, \{X3\}, \{X4\}\}$.

### Court-circuit du capteur FP1 :

Cette panne peut s'écrire sous la forme :

```
  assertion FP1 eq 1;
```

Après compilation par G2OC et ajout de cette spécification, le fichier est transmis à BAC qui donne comme résultat :

```
NORMALIZATION
  No combinatorial loop
REACHABLE STATES
  Reachable states computed after 3 steps
   not X0 and  not X1 and  not X2 and  not X3 and X4 and FP1 or
   not X0 and  not X1 and  not X2 and X3 and  not X4 and FP1 or
   not X0 and  not X1 and X2 and  not X3 and  not X4 and FP1 or
  X0 and  not X1 and  not X2 and  not X3 and  not X4 and FP1
```

soit l'ensemble des états: $\{\{X0\}, \{X2\}, \{X3\}, \{X4\}\}$

Sur cet exemple, les deux recherches d'accessibilité montrent que le système réagit encore correctement lorsque FP1 est en rupture (toutes les étapes peuvent être actives). Par contre FP1 en court-circuit, dégrade le fonctionnement de l'automate booléen puisque l'état $X1$ n'est plus accessible. En particulier l'action "POUSSER" n'est plus émise.

#### 6.3.3.2 Quelques propriétés de Sûreté

La sûreté recouvre un ensemble de propriétés qui ne doivent *jamais* arriver.

**Un état indésirable n'est jamais atteint :**

Dans beaucoup de systèmes de commande, il existe des états de fonctionnement indésirables (états de blocage, comportements non prévus et destructifs ...). Pour s'en protéger les concepteurs ont souvent recours à une *surspécification* de leurs systèmes. Cette habitude est néfaste pour la lisibilité, la capacité d'évoluer et souvent ne résout pas le problème car un cas particulier n'a pas été prévu et conduit évidemment à l'état indésirable! Sur l'exemple de la figure 6.4, on peut par exemple démontrer que l'arrêt d'urgence est exclusif avec l'activation du vérin. Nous pouvons utiliser pour cela la notion d'invariant de BAC sous la forme :

```
invariant not (X4 and (X1 or X2));
```

BAC sait vérifier une propriété logique invariante pour tous les états accessibles, sans avoir besoin d'énumérer ces derniers. Sa notion "d'invariant" a été définie pour assurer la vérification de propriétés de sûreté. Ceci est très intéressant vu que la syntaxe utilisée par BAC est très simple vis à vis des autres outils de preuves. Ainsi la propriété précédente peut aisément se lire "X4 n'est jamais actif en même temps que X1 ou X2".

Bien sûr, la lecture du grafcet montre tout de suite que la propriété est vraie dans notre exemple simple, mais en pratique cela est rarement le cas. Ici la réponse de BAC est sans ambiguïté :

```
NORMALIZATION
  No combinatorial loop
FORWARD INVARIANT CHECKING
  Property satisfied after 3 steps
```

Avec une propriété fausse, cet outil aurait donné les plus courtes histoires d'entrée qui mènent au contre-exemple.

**Des actions conflictuelles ne sont jamais émises en même temps :**

Cette propriété ressemble beaucoup à la précédente, de part le lien étape-action qui existe en GRAFCET. Elle est tout de même plus générale car une action peut être émise par *plusieurs* étapes en même temps. Elle est de plus très importante pour les systèmes automatisés car certains actionneurs ne "supportent pas" de recevoir des ordres contradictoires. De même les exemples d'automatismes ne manquent pas où une activation simultanée de plusieurs actionneurs conduit à une destruction mécanique pure et simple du système. Enfin elle constitue souvent un critère de sécurité pour les usagers.

Sur notre exemple, nous pouvons évidemment vérifier que POUSSER et RAMENER ne sont jamais émis en même temps. La propriété s'exprimera sous la même forme que précédemment:

```
invariant not (POUSSER and RAMENER);
```

#### 6.3.3.3 Equivalence faible (observationnelle)

Deux machines d'états finis sont dites "faiblement équivalentes" si pour une même histoire quelconque des entrées, elles fournissent la même histoire de sorties. En grafcet, cette propriété est donc intéressante pour simplifier un système de commande. Notons qu'il existe de nombreux de critères d'équivalence de systèmes d'états. Une synthèse a été présentée dans [Arn92].

Considérons par exemple les deux grafcets de la figure 6.5. A priori différents, ils sont en fait parfaitement équivalents. Pour le montrer, il suffit de compiler ces deux grafcets ensemble en ayant préalablement renommé les étapes et les actions du second grafcet [1]. La question se pose facilement par un invariant :

```
invariant (MONTER eq MONTER2) and
          (DESCENDRE eq DESCENDRE2) and
          (PIVOTER eq 0) and (PERMUTER eq 0) and
          (AVANCER eq 0) and (RECULER eq 0);
```

1. Pour ce cas, MONTER a été renommé en MONTER2, DESCENDRE en DESCENDRE2 ...

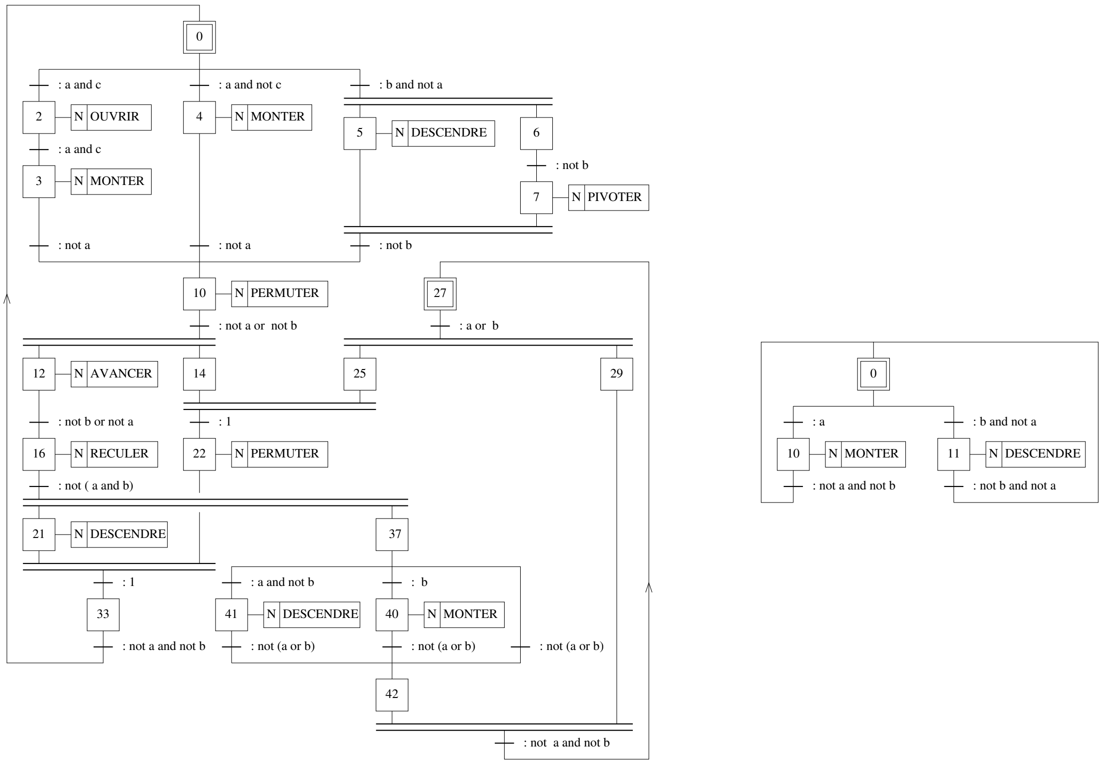



FIG. 6.5 - *Exemple de deux grafcets de commande équivalents*

Les actions "PIVOTER", "PERMUTER" et "AVANCER" n'ont aucun équivalent dans le second grafcet. Les grafcets sont donc équivalents ssi ces actions ne sont jamais émises; ce qui explique l'écriture de l'invariant précédent.

Pour ce cas, BAC donne comme réponse:

```
NORMALIZATION
  No combinatorial loop
FORWARD INVARIANT CHECKING
  Property satisfied after 3 steps
```

### 6.3.4 Conclusion

Vis à vis des autres outils de preuves, l'utilisation de BAC apporte des avantages mais aussi des inconvénients. Du côté des avantages, l'expression des propriétés par BAC est beaucoup plus explicite. En effet notre compilation en automate booléen conserve la notion d'étape ce qui n'est pas forcément le cas avec les outils classiques manipulant des automates finis. Ensuite ces outils (excepté AUTO), sont souvent basés sur une logique temporelle particulière et leur syntaxe peut surprendre un concepteur de grafcet dont la sensibilité provient plutôt de l'Automatique. De son côté, BAC exprime les propriétés "simplement" et permet facilement de contraindre le système.

Nous avons déjà exposé dans le paragraphe 6.2.3 le problème de l'explosion du nombre d'états lorsque les systèmes de preuves étudient toutes les situations accessibles. BAC est beaucoup moins sensible à ce phénomène car il manipule formellement des expressions et des vecteurs d'état. Ceci lui permet d'ailleurs de montrer rapidement l'équivalence observationnelle de systèmes. La seule contrainte d'implémentation est donc située sur la taille des BDDs manipulés.

Mais de nombreuses propriétés de sûreté font intervenir un caractère séquentiel qui est difficile à transcrire en BAC. Ce dernier n'offre pas de possibilités directes pour exprimer les propriétés temporelles. Le seul opérateur de base mis à notre disposition est "pre". Il permet de définir les équations de changement d'états. Pour traduire des propriétés plus intéressantes, nous allons maintenant introduire la notion "d'observateur".

## 6.4 Notion d'observateur

### 6.4.1 Présentation

Les problèmes de sûreté ne sont pas spécifiques à notre approche synchrone des systèmes temps-réel. Plusieurs formalismes ont servi de base à l'expression de ces propriétés comme les *réseaux de Petri* [AD93], processus communicants ou moniteur de Hoare [Hoa85], les algèbres de processus, les logiques temporelles [Arn92]. Tous visent à exprimer de manière précise et sans ambiguïté la propriété visée. Il est de plus souhaitable d'exprimer cette propriété dans le même formalisme que le programme.

À partir de ces deux idées, les concepteurs de LUSTRE ont proposé de vérifier leurs programmes par des propriétés elles aussi exprimées en LUSTRE [HCRP90]. Comme ce langage peut être considéré comme une logique temporelle particulière [PH88], de nombreuses propriétés de sûreté $P$ peuvent être exprimées par une expression booléenne $B$ qui doit rester vraie sur toutes exécutions possibles du programme. Si $B$ est bien une tautologie, la propriété $P$ sera vérifiée. De cette manière, un ensemble de propriétés génériques exprimées en LUSTRE a été proposé ("`onceBfromAtoC`","`since`","`never`") dans [Hal93]. La propriété est ensuite intégrée au programme LUSTRE initial pour former un nouveau programme. L'ensemble est compilé, puis vérifié par un outil de preuves tel que LESAR [Ber92].

La figure 6.6 extraite de [Hal93] résume cette approche. Le programme initial `P` est encapsulé dans un programme `P'` qui ne possède qu'une seule sortie résultat de la propriété `B` à vérifier. Le programme `P'` est appelé *observateur* de `P`. Cette solution a le grand mérite de réduire la taille de l'automate en prenant

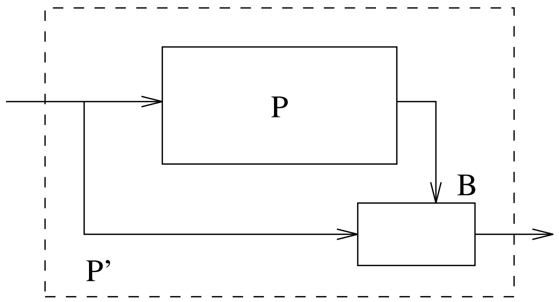



FIG. 6.6 - *Programme de vérification*

en compte la propriété dès la génération du graphe. En GRAFCET, cette idée est aussi soutenue dans [Rou94].

Indépendamment des travaux de N. Halbwachs et de son équipe, nous avons cherché un moyen de réaliser des preuves de sûreté sur le GRAFCET en partant des mêmes hypothèses de départ et de nos résultats acquis par ailleurs:

- Une propriété de sûreté peut être exprimée par une fonction booléenne qui doit rester vraie.
- BAC sait justement vérifier formellement ce type de propriété grâce à sa notion d'invariant.
- Tout grafcet peut être compilé en automate booléen équivalent compréhensible par BAC.

Nous pouvons assimiler les propriétés de sûreté à un système réactif synchrone autonome. Ce module est plus conséquent qu'une simple équation booléenne, il possède un état propre qui évolue avec *l'observation* des entrées et des réactions du système à prouver. Ces entités sont identiques aux "observateurs" de N. Halbwachs. Nous avons donc conservé cette appellation.

Un observateur simple considère la propriété potentiellement vraie jusqu'à l'instant où il trouve un contre-exemple. À partir de ce moment, il se référera au passé et indiquera la propriété comme fausse. Des observateurs plus complexes peuvent aussi être construits pour évaluer des propriétés composées par exemple de plusieurs occurrences d'une même implication. Comme ces observateurs sont des machines de Mealy classiques manipulant des variables booléennes, ils peuvent s'exprimer sous forme d'automates booléens. Tester une propriété de sûreté avec BAC se résume donc à ajouter au code final, toutes les équations spécifiques à l'observateur puis de vérifier que la sortie `propriété` reste vraie quelle que soit l'évolution du système d'origine.

En pratique, les observateurs peuvent prendre des formes différentes. Sur les systèmes LUSTRE, il est intégré au système et permet de filtrer les sorties et d'optimiser le code généré. Le notre est autonome et n'intervient pas sur le système initial. Par ce côté, il est plus proche de la notion "d'observateur" connue en Physique. Il peut s'exprimer dans le même formalisme que l'application, mais ceci n'est pas obligatoire. Sa principale qualité est de ne pas nécessiter la recompilation du système pour chaque preuve. Bien sûr, nous perdons l'optimisation rendue possible par la compilation conjointe, mais ceci n'est pas gênant dans notre cas car le grafcet est déjà compilé de manière compacte.

D'un point de vue utilisation, un compilateur LUSTRE vers BAC est disponible ce qui nous permet d'exprimer au départ nos propriétés en LUSTRE et de rester compréhensible par les utilisateurs de GRAFCET. Mais notre définition d'*observateur* permet aussi d'exprimer les propriétés directement en S-GRAFCET, puis de les compiler séparément par G2OC.

### 6.4.2 Expression des observateurs

Nous allons maintenant présenter plusieurs observateurs de base pour réaliser des preuves de comportement. Ces opérateurs pourraient s'écrire en LUSTRE. Pour faciliter leur compréhension intuitive, nous avons préféré les exprimer en S-GRAFCET) car ce modèle est parfaitement adapté. Bien sûr la liste présentée n'est pas exhaustive, mais elle permet d'imaginer ou de construire d'autres observateurs plus complexes.

Pour être cohérent avec nos définitions, nous avons implanté nos observateurs en BAC. Pour certains, nous avons directement écrit les équations de changement d'état, pour d'autres nous avons conservé le code issu de la compilation grafcet.

#### 6.4.2.1 neverA

L'observateur `neverA(A)` a été identifié dans [Hal93, HPOG88]. Il rend `vrai` tant que `A` est absent. Il mémorise l'occurrence de `A` et répond `faux` à partir de cet instant. Cet opérateur s'exprime en LUSTRE par l'équation :

```
node neverA(A:bool) returns(propriete: bool);
let
  propriete = (not A) -> (not A and pre(neverA))
tel
```

Cet observateur s'exprime aussi très simplement en S-GRAFCET :

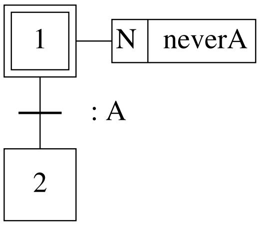



FIG. 6.7 - *Définition grafcet de l'observateur "neverA(A)"*

Il sera finalement instancié en BAC pour être ajouté à l'application. Son code pourra prendre la forme:

```
state neverApre;
inputs A;
locals propriete;

initial neverApre;

transition
neverApre'= propriete;

definitions
propriete= neverApre and (not A);
```

Avec cet opérateur, nous remarquons déjà la simplicité d'expression et la compréhension intuitive que donne le S-GRAFCET. Nous allons donc utiliser ce modèle pour nos définitions à venir.

#### 6.4.2.2 neverA_afterOratB

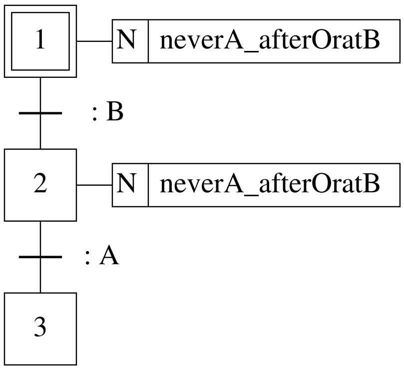



FIG. 6.8 - *Observateur "neverA_afterOratB(A,B)"*

Tant que `B` est absent, `neverA_afterOratB(A,B)` est insensible à `A` et rend `vrai`. Son comportement change avec l'occurrence de `B` : l'opérateur rend alors `faux` dès l'occurrence de `A`. En particulier la valeur fausse est rendue si `A` et `B` sont présents dans le même instant.

Sur ce principe, nous pouvons définir plusieurs observateurs très voisins :

| | |
|---|---|
| `alwaysA_afterOratB` | `A` doit rester vraie dès l'apparition d'un facteur `B` |
| `A_afterOratB` | tant que `B` est absent, la propriété est maintenue à `vrai`. L'occurrence de `B` force la propriété à `faux` jusqu'à l'occurrence de `A` |
| `onceA_afterOratB` | diffère du précédent observateur par l'unicité de l'occurrence de `A` |

Ces différents observateurs sont décrits en annexe.

#### 6.4.2.3 B_afterOratA_beforeC

L'opérateur `B_afterOratA_beforeC(A,B,C)` rend `vrai` s'il existe au moins une occurrence de `B` dans l'intervalle temporel `[A,C[`. Si `A` et `C` sont présents en même temps, la valeur `faux` est rendue.

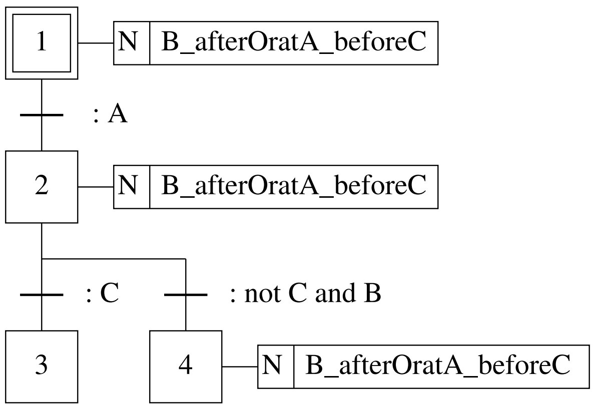



FIG. 6.9 - *Observateur "B_afterOratA_beforeC(A,B,C)"*

Il peut également se présenter sous diverses variantes :

| | |
|---|---|
| `onceB_afterOratA_beforeC` | une et une seule occurrence de `B` |
| `neverB_afterOratA_beforeC` | aucune occurrence de `B` |
| `alwaysB_afterOratA_beforeC` | `B` toujours présent |

#### 6.4.2.4 Variantes vis à vis des bornes d'étude

Les opérateurs de type `B_afterOratA_beforeC` travaillent sur l'intervalle temporel `[A,C[`. Rien n'empêche d'imaginer des observateurs travaillant sur les intervalles `[A,C]`, `]A,C[` ou `]A,C]`. Ce dernier intervalle donne en particulier naissance à la classe d'opérateurs B_afterA_beforeOratC :

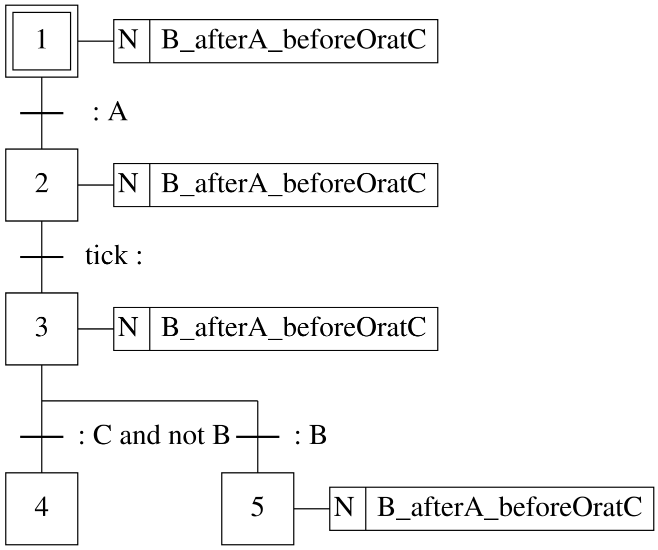



FIG. 6.10 - *Observateur "B_afterA_beforeOratC(A,B,C)"*

Toute occurrence de `B` qui arrive avant ou en même temps que `A` n'est pas prise en compte. On remarquera sur la figure 6.10 la réceptivité "`tick:`" spécifique au S-GRAFCET, qui permet de tester `B` dès l'instant qui suit l'occurrence de `A`.

#### 6.4.2.5 Classe des observateurs "anytime"

Tous les observateurs que nous venons de présenter, vérifient la propriété une seule fois durant la vie du système. Ceci n'est généralement pas suffisant pour couvrir toutes les propriétés de sûreté. En effet, certaines caractéristiques (la prise en compte d'arrêt d'urgence par exemple) doivent se vérifier un nombre de fois *quelconque*. Tous les observateurs précédents, doivent donc être redéfinis en conséquence. Nous supposerons ici qu'ils sont non-réentrants : si plusieurs occurrences de `A` précèdent `B` et `C`, une seule sera prise en compte.

Ainsi l'opérateur `cyclic_B_afterOratA_beforeC(A,B,C)` est issu de `B_afterOratA_beforeC(A,B,C)`. Il rend `vrai` si à chaque nouvelle occurrence de `C` :

- soit `A` n'est pas apparu entre-temps,
- soit `B` est apparu après (ou en même temps) que `A`.

Si `A`, `B`, `C` apparaissent en même temps la propriété n'est pas vérifiée. Cet observateur est très intéressant pour montrer qu'un système peut toujours réagir dans un temps imposé. Il a déjà été référencé dans [Hal93] sous l'appellation "`onceBfromAtoC`".

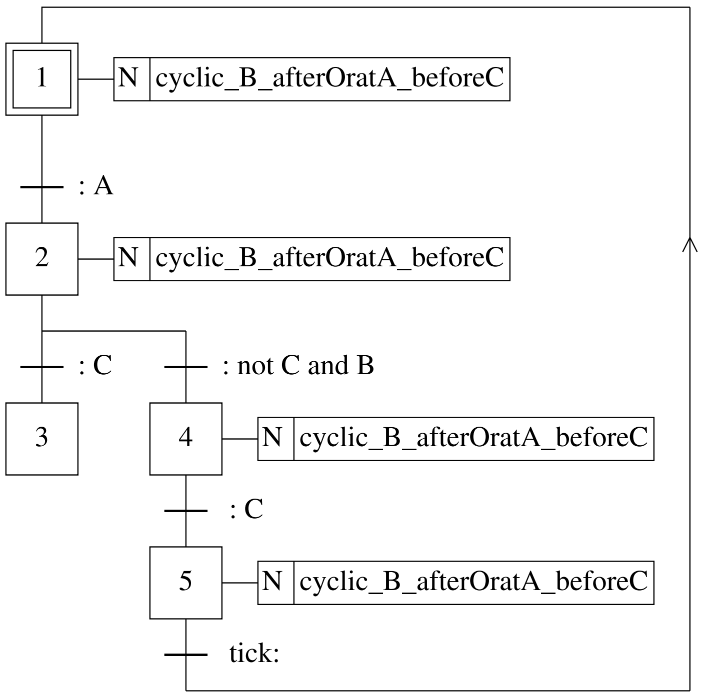



FIG. 6.11 - *Observateur "cyclic_B_afterOratA_beforeC(A,B,C)"*

### 6.4.3 Application

Pour montrer l'intérêt de la notion d'observateur, reprenons l'exemple de gestion de vérins (fig.6.4). Durant le processus, la sortie complète du vérin peut être nécessaire pour placer correctement une pièce à usiner. Pour cela, le grafcet doit assurer la fonctionnalité suivante:

"Le vérin ne peut être ramené en position de départ, que si une des fins de course plus a été activée."

Or l'activation de ces capteurs fait suite à l'émission de l'ordre `POUSSER`. Pour prouver cette propriété, nous pouvons définir l'opérateur `cyclic_B_afterA_beforeOratC(A,B,C)`, tel que `A` soit instancié par `POUSSER`, `B` soit instancié par `FP1 or FP2` et `C` par `RAMENER`.

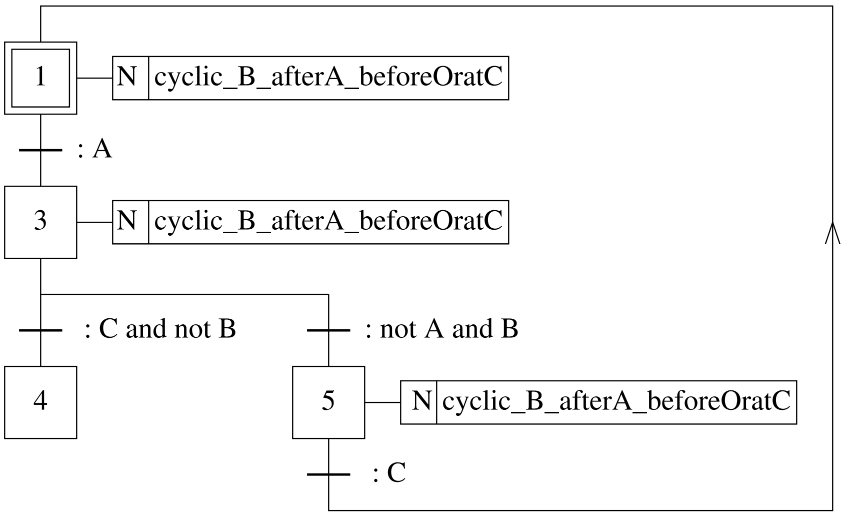



FIG. 6.12 - *Observateur "cyclic_B_afterA_beforeOratC(A,B,C)"*

Les équations de l'observateur et du grafcet compilé sont ensuite regroupées en un fichier BAC unique. La sortie de l'observateur peut alors faire l'objet d'une vérification d'invariant.

Pour ce cas BAC répond :

```
NORMALIZATION
  No combinatorial loop
FORWARD INVARIANT CHECKING
  Property violated at step 3
  Here are all the shortest bad scenarios:
    step 1 **    not FP2 and  not FP1 and  not UR and a
    step 2 **    not FP2 and  not FP1 and UR
    step 3 **    not FM2 and  not FM1 and  not FP2 and
                 not FP1 and  not UR and  not a and REP
```

Ceci signifie qu'il existe des cas où le vérin ne va pas jusqu'en bout de course. BAC indique `a` vrai (action `POUSSER` émise), puis arrêt d'urgence. Ces deux phases ont lieu avant que les fins de course `FP1` et `FP2` soient atteintes. Ensuite le système redémarre avec `REP` mais l'opérateur a demandé un changement de sens du vérin $\neg a$. Au vu de ces renseignements, nous nous apercevons sur le grafcet, que la reprise de cycle qui suit un arrêt d'urgence, ne se fait pas obligatoirement au même endroit. Pour cela, il suffit d'inverser le sens du vérin pendant la phase d'arrêt d'urgence. À la reprise, le vérin change effectivement de sens sans avoir préalablement activé les capteurs de fins de course. Pour pallier ce phénomène, la phase d'arrêt d'urgence doit mémoriser le sens de marche du vérin, (fig. 6.13).

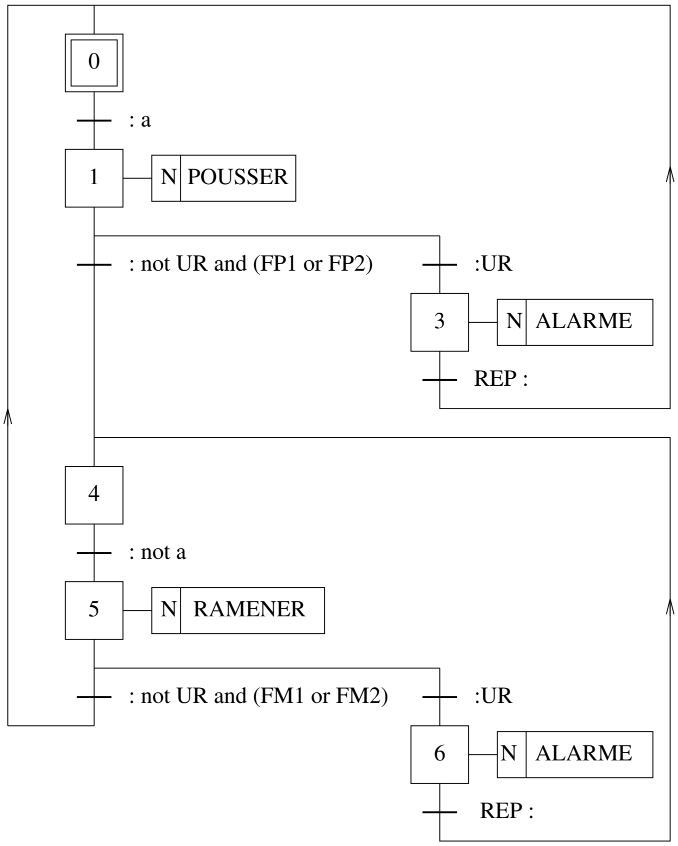



FIG. 6.13 - *Nouvelle gestion du vérin*

Avec ce nouveau grafcet, la propriété est cette fois satisfaite :

```
NORMALIZATION
  No combinatorial loop
FORWARD INVARIANT CHECKING
  Property satisfied after 6 steps
```

## 6.5 Conclusion

La représentation du grafcet sous forme d'automate booléen, permet de manipuler des exemples conséquents et de vérifier des preuves de comportement de manière formelle grâce à l'outil de preuves BAC. Celui-ci évite l'étude énumérative de tous les états accessibles. Par rapport aux autres outils, les propriétés à prouver s'expriment très simplement. Cette facilité provient essentiellement de la bijection (étape grafcet - variable d'état automate) qui a pu être conservée par notre génération et des notions d'invariant et d'assertion existant dans l'outil de preuves.

La limitation de BAC concernant l'expression des propriétés complexes de sûreté a pu être levée grâce à notre définition des observateurs. Ceux-ci s'expriment d'ailleurs simplement en S-GRAFCET, ce qui renforce encore leur intérêt et facilite leur création.

Les propriétés de vivacité restent difficiles à exprimer. Sur les systèmes Temps-réel, nous pouvons nous ramener à des propriétés de sûreté, grâce à l'existence de contraintes temporelles connues (ou imposées). Nous retombons alors sur des opérateurs du type `cyclic_B_afterOratA_beforeC` où `C` est la limite temporelle ("deadline") à ne pas dépasser.

Au niveau application, nos observateurs sont actuellement instanciés à la main pour chaque preuve à établir et pour chaque fichier BAC issu de la compilation LUSTRE ou GRAFCET. Des travaux sont en cours, pour automatiser cette phase et fournir à l'utilisateur tout un environnement de spécification de propriétés.

# Chapitre 7

# Conclusion et perspectives

Pour exposer les nombreux avantages de l'approche synchrone dans la conception des applications temps réel, nous avons été amené à présenter des langages synchrones textuels et graphiques. Nous nous sommes principalement basé sur ESTEREL, qui par sa simplicité et sa rigueur a fait progresser la compréhension des modèles synchrones. Coté graphique, nous nous sommes intéressé principalement au GRAFCET. Sa syntaxe est très conviviale. Elle permet d'exprimer simplement de nombreux comportements.

Ces langages sont *complémentaires*. Ils possèdent leurs qualités syntaxiques et sémantiques propres. En fait, chacun exprime la sensibilité de son domaine d'origine. La coopération ESTEREL-GRAFCET est par exemple intéressante pour gérer les modes de marche et d'arrêt des SAP. Vue l'importance du GRAFCET dans le milieu industriel et les atouts sémantiques des langages synchrones, nous avons dès lors, cherché à intégrer le GRAFCET dans une plate-forme synchrone commune. Plus généralement, nous avons voulu dans ce mémoire, resserrer les liens entre automaticiens et informaticiens sur les problèmes de fond concernant les *modèles*, la *formalisation des évolutions synchrones* et la *validation des systèmes*.

## Modèle

L'intégration du GRAFCET impose que toutes les ambiguïtés sémantiques du modèle soient préalablement levées. La sémantique d'évolution du GRAFCET doit en plus être compatible avec "l'esprit" synchrone et respecter l'hypothèse d'atomicité des réactions. C'est pourquoi, nous avons consacré le chapitre 3 à ces problèmes.

Nous sommes revenu sur la notion de réceptivité, en insistant sur la nécessaire distinction syntaxique et sémantique entre *événements* et *conditions*. Pour formaliser les évolutions *avec recherche de stabilité*, nous n'avons pas retenu l'algorithme classique de détermination de la situation suivante. Celui-ci s'appuie sur la *double échelle de temps* du modèle et la notion de temps interne est incompatible avec l'hypothèse de synchronicité forte d'ESTEREL. Nous avons aussi été amené à redéfinir la notion de variable d'état pour qu'elle s'apparente beaucoup plus à celle rencontrée dans les langages synchrones.

## Formalisation des évolutions synchrones et compilation

Les remarques précédentes nous ont conduit à définir un nouveau modèle et sa sémantique : le S-GRAFCET. La dynamique du modèle est exprimée de manière formelle grâce à des ensembles d'événements et chaque situation-solution se caractérise par un *point fixe*. Nous avons volontairement cherché à modifier *le moins possible* le modèle GRAFCET standard. Afin de ne pas déconcerter l'utilisateur, il faut qu'un très grand nombre de grafcets aient les mêmes comportements suivant les deux modèles. Pour cela, des conditions suffisantes sont connues :

- Les réceptivités des divergences en OU sont disjonctives, et
- Les réceptivités ne font intervenir que les entrées du système.

La première condition est généralement satisfaite par les utilisateurs eux-mêmes qui veulent limiter tout risque de parallélisme interprété. La seconde condition concerne principalement l'utilisation des variables d'état dans les réceptivités. En S-GRAFCET, ces variables ont un sens différent.

La caractérisation par point fixe n'est tout de même pas suffisante pour définir complètement le modèle. Elle ne permet pas d'identifier les grafcets instables et dans le cas général la solution trouvée peut laisser indéterminé le franchissement de certaines transitions. Enfin, établir l'existence d'une solution unique, ne donne pas forcément une méthode pour l'obtenir.

Nous avons ainsi été amené à définir deux sémantiques d'exécution qui résolvent ces problèmes. La première est basée sur la modélisation des comportements par ESTEREL. Le S-GRAFCET devient alors un sous-ensemble d'ESTEREL. Ceci permet d'intégrer la chaîne de compilation de ce langage et de bénéficier de ses outils de simulation et de preuves. Pour faciliter cette intégration, nous avons développé le compilateur Grafcet-Esterel G2E. Nous nous sommes imposé la génération de code ESTEREL pur, afin de préserver les possibilités de preuves et d'assurer une implémentation plus efficace.

Le modèle retenu permet surtout de reporter le problème des cycles instantanés S-GRAFCET en problèmes de causalité ou de cycle instantané ESTEREL. Dès lors, un compilateur qui respecte entièrement la sémantique comportementale d'ESTEREL est capable de *détecter formellement* tout grafcet instable et de compiler tout grafcet stable. Actuellement seul le compilateur V3 accepte de résoudre les équations de signaux ESTEREL générés par notre compilateur G2E : la version V4 est beaucoup moins performante pour cette classe de programme. De plus, nous n'avons pas l'*absolue certitude* que le compilateur V3 ne rejette pas à tort des programmes corrects malgré l'aide fournie par G2E pour limiter les fausses causalités.

Pour pallier ce phénomène, nous avons défini une seconde sémantique d'exécution directement issue du modèle de point fixe. Elle est basée sur la notion de *micro-pas* et s'appuie sur un algorithme itératif de recherche de l'état suivant (PFI). Cet algorithme accepte tout grafcet stable et rejette les autres. Son implantation a donné naissance au compilateur G2OC qui génère directement un automate au format OC à partir du grafcet initial. Ce compilateur peut également générer un automate booléen au format BAC. Par souci de portabilité, G2OC est également capable de compiler un grafcet en respectant le modèle GRAFCET normalisé au lieu du modèle S-GRAFCET. Pour toutes les raisons déjà évoquées, nous déconseillons cette dernière utilisation.

La mise en place des deux compilateurs a nécessité un lourd investissement en temps. Ces outils ne forment pourtant pas l'objectif ultime de la thèse. Ils facilitent seulement la conception et plus particulièrement la validation des systèmes.

## Validation des systèmes

L'intégration du S-GRAFCET au sein d'une plate-forme synchrone commune basée sur ESTEREL ou directement sur OC, nous a permis d'utiliser les outils de simulation et de preuves existants pour valider les grafcets. Nous avons ainsi montré comment des outils généraux comme AUTO pouvaient réaliser des preuves de comportement.

Toutefois, si ces outils semblent bien adaptés aux préoccupations des concepteurs d'automatismes, des progrès restent à accomplir dans le sens d'une meilleure convivialité dans l'interface d'accès aux outils de preuves formelles. Il faut en particulier exprimer les propriétés à vérifier dans un langage proche du domaine d'application. C'est pourquoi nous nous sommes tourné vers l'outil BAC qui supporte les notions "d'invariance" et d'assertion". De plus, sa syntaxe est très intuitive pour un automaticien. Avec une génération adaptée, nous pouvons conserver la notion d'étape GRAFCET, ce qui renforce encore la pertinence des preuves réalisées. Dans le même esprit, nous proposons d'exprimer les propriétés de sûreté directement en GRAFCET et nous avons donné quelques exemples d'opérateurs. L'idée est de compiler séparément le grafcet de contrôle et le grafcet de propriété, pour les présenter ensemble à BAC.

## Perspectives

Nous envisageons de nouveaux développements dans les différents domaines abordés.

Au niveau réalisation, les compilateurs G2E et G2OC sont déjà opérationnels et disponibles sur le réseaux. Les deux futures versions intégreront les actions conditionnées en plus des actions continues et impulsionnelles déjà acceptées. Au niveau génération, la prochaine version du compilateur G2OC pourra de plus produire un automate booléen dans un fichier OC monoboucle ou un système d'équations sous forme de nœud LUSTRE. A moyen terme, nous pensons produire également des architectures matérielles.

Des travaux pour améliorer l'éditeur-simulateur EDGE sont prévus. Le plus intéressant concerne son interfaçage avec le simulateur ESTEREL XES. Au niveau des outils de preuves, nous développons actuellement une interface graphique pour faciliter l'élaboration des propriétés à prouver et automatiser les différentes phases.

Au niveau théorique, nous souhaitons reformaliser les évolutions GRAFCET pour éviter l'étude de toutes les situations accessibles lors de la compilation. Une idée serait de simplifier algébriquement les équations d'évolution. Pour cela, nous aurons besoin de l'élément neutre comportemental. Nous envisageons aussi de continuer nos travaux sur les preuves formelles de comportement : les propriétés de vivacité restent, en effet, encore difficilement vérifiables en GRAFCET. En sûreté, nous souhaitons trouver des théorèmes de répartition, qui permettent de conserver globalement les propriétés établies sur chaque sous-système. Ce problème déborde du cadre “GRAFCET” et concerne tous les formalismes synchrones.

# Annexe A

# Grammaire du format commun d'entrée GT

La grammaire complète de ce format est donné dans [Lep94]. Elle a l'avantage de préciser les notations, même si elle ne fait pas encore l'objet d'une normalisation. Les compilateurs G2E et G2OC utilisent un sous-ensemble de cette grammaire pour être compatible avec l'éditeur EDGE et EG7.

Le format GT se décompose en 5 parties:

- Le nom du fichier
- description des étapes
- description des actions
- description des transitions
- description des réceptivités

## Les étapes

Chaque étape est codée sur une ligne avec le format suivant:

```
<type_etape> <nom_etape> <liste_actions_associees> 'ES'
```

Le champ "type_etape" peut prendre les valeurs IS ou NS suivant que l'étape est initiale ou non.

Le champ <nom_etape> définit l'identificateur (nom ou numéro) de l'étape.

La liste <liste_actions_associees> indique l'ensemble des actions associées à l'étape. Chaque action est définie par son identificateur et séparée de la précédente par ":". La liste complète est délimitée par "ES".

Exemple:

```
IS 1: a1 : a2 ES
```

## Les actions

Chaque action est codée sur une ligne avec le format suivant:

```
'AC' <nom_action> '[' <qualificatif_action> ':' <nom_action_grafcet> ':='
     <option_action> ']' 'EA'
```

Le champ <nom_action> définit l'identificateur de l'action. Pour ce champ, les éditeurs EDGE et EG7 ont pris comme convention de mettre la lettre "a" suivi d'un numéro. Ceci n'est pas une obligatoire et G2OC accepte n'importe quel identificateur formé de lettres et de chiffres.

Le champ <qualificatif_action> est représenté par une lettre seule: "N" pour Normale ou "P" pour impulsionnelle.

Le champ <nom_action_grafcet> indique le véritable nom de l'action dans le grafcet.

Les champs <option_action> et ":=" sont optionnels : ils donnent des explications sur l'action à effectuer.

Exemple :

```
AC a1 [N: LEVER ]
```

## Les transitions

Chaque transition est codée sur une ligne suivant le format:

```
'TR' <num_transition> 'FR' <liste_etapes_amonts> 'TO'
     <liste_etapes_avales> ':=' <nom_receptivite> 'ET'
```

Le champ <num_transition> définit l'identificateur (nom ou numéro) de la transition. Les listes <liste_etapes_amonts> et <liste_etapes_avales> sont définies par un couple de parenthèses et contiennent des identificateurs d'étapes séparés par des virgules. Le champ <nom_receptivite> définit l'identificateur de la réceptivité.

Exemple :

```
TR 1 FR 2 TO (6,3) := r2 ET
```

## Les réceptivités

Chaque réceptivité est codée suivant le format:

```
'RE' <nom_receptivite> '[' <condition_recep_ev> ':'
     <condition_recep_bool> ']' 'ER'
```

Le champ <nom_receptivité> définit l'identificateur (nom) de la réceptivité. Pour ce champ, les éditeurs EDGE et EG7 ont pris comme convention de mettre la lettre “r” suivi d'un numéro. Ceci n'est pas une obligatoire et G2OC accepte n'importe quel identificateur formé de lettres et de chiffres.

Les champs <condition_recep_ev> et <condition_recep_bool> sont des expressions classiques. La seule différence de syntaxe touche les éléments neutres “1” est représenté par “tick” et “0” est représenté par “not tick” dans l'espace des événements.

Exemple :

```
RE r4 [tick : a or not b ] ER
```

## Description de la grammaire utilisé par G2OC et G2E

Pour représenter les identificateurs, nous utiliserons les lexèmes CHAINE et NOMBRE. Le premier est constitué d'une lettre suivie de lettre(s) ou de chiffre(s). Le second est constitué de chiffre(s).

```
<grafcet> ::= <nom_module> <liste_etapes> <liste_actions>
      <liste_transitions> <liste_receptivites>

<nom_module> ::= CHAINE

<liste_etapes> ::= VIDE |
   <liste_etapes> <etape>

<etape> ::= <type_etape> <nom_etape> <liste_actions_associees> 'ES'
```

```
<type_etape> ::= 'IS' | 'NS'

<nom_etape> ::= CHAINE | NOMBRE

<liste_actions_associees> ::= VIDE |
      <liste_actions_associees> <action_associees>

<action_associee> ::= ':' <nom_action>

<nom_action> ::= CHAINE

<liste_actions> ::= VIDE |
    <liste_actions> <action>

<action> ::= 'AC' <nom_action> '[' ':' ']' 'EA' |
     'AC' <nom_action> '[' <qualificatif_action> ':' <nom_action_grafcet> ']' 'EA' |
     'AC' <nom_action> '[' <qualificatif_action> ':' <nom_action_grafcet> ':='
  <option_action> ']' 'EA'

<qualificatif_action> ::= 'N' | 'P'

<nom_action_grafcet> ::= CHAINE

<option_action> ::= CHAINE

<liste_transitions> ::= VIDE |
<liste_transitions> transition

<transition> ::= 'TR' <num_transition> 'FR' <liste_etapes_amonts> 'TO'
 <liste_etapes_avales> ':=' <nom_receptivite> 'ET' |
 'TR' <num_transition> 'FR' <liste_etapes_amonts> 'TO'
 <liste_etapes_avales> 'ET'

<num_transition> ::= NOMBRE

<liste_etapes_amonts> ::= VIDE |
  <nom_etape> |
  '(' <sous_liste_etapes> ')'

<liste_etapes_avales> ::= VIDE |
  <nom_etape> |
  '(' <sous_liste_etapes> ')'

<sous_liste_etapes> ::= <nom_etape> ',' <nom_etape> |
```

```
<sous_liste_etapes> ',' <nom_etape>

<nom_receptivite> ::= CHAINE

<liste_receptivites> ::= VIDE |
 <liste_receptivites> <receptivite>

<receptivite> ::= 'RE' <nom_receptivite> '[' <condition_recep_ev> ':'
   <condition_recep_bool> ']' 'ER'

<condition_recep_ev> ::= <condition_generale>

<condition_recep_bool> ::= <condition_generale>

<condition_generale> ::= VIDE | <condition_recep>

<condition_recep> ::= <condition_recep> 'OR' <terme_recep> |
      <terme_recep>

<terme_recep> ::= <terme_recep> 'AND' <facteur_recep> |
  <facteur_recep>

<facteur_recep> ::= '(' <condition_recep> ')' |
    'NOT' <facteur_recep> |
    '!' <facteur_recep> |
    CHAINE |
    '0' | '1'
```

## Remarques

– Conformément à la syntaxe concrète définie au § 4, les réceptivités sont décomposées en deux champs : “événement : condition” dont l'un d'entre eux peut être vide. Dans ce dernier cas, la distinction entre réceptivité événementielle et réceptivité conditionnelle se fait par la place des “:”.

– Au niveau des réceptivités, les noms “*tick*” (n'importe quel événement déclencheur d'évolution), “X<numero>” (prédicat associé à l'activité d'une 'étape), “RE<numero>” (front montant de l'étape), “FE<numero>” (front descendant de l'étape) sont des mots réservés.

# Annexe B

# Compléments d'information sur les compilateurs de Grafcet

## B.1 Le compilateur Grafcet-Esterel : G2E

### B.1.1 Options de compilation de G2E

La syntaxe générale de G2E est:

```
g2e <nom_fichier> [ -noaction] [ -norename]
  [ -selfcont] [ -simul] [ -single]
```

**Options :**

**-h :** donne la liste des options.

**-noaction :** considère les actions continues GRAFCET comme des *outputs* ESTEREL à émettre à chaque *instant*. Normalement ces actions continues sont gérées par deux signaux de commande :SET_<action> et RESET_<action> qui permettent via des bascules SR de sortie de les implémenter de manière continue.

**-norename :** ne génère pas l'extension '_A' sur le nom des actions continues SET_<action> et RESET_<action>. Cette option permet aussi de ne pas renommer le module <module> en GRAFCET_<module>. Elle est obligatoire avec MDLC.

**-selfcont3 ou -selfcont5 :** ajoute au code généré, les bibliothèques ic3 ou ic5 de comportement nécessaires au compilateur ESTEREL.

**-simul :** génère les signaux de sortie ZX<etape>_EXPORTED qualifiant l'état des étapes GRAFCET. Cette option est obligatoire avec EG7.

### B.1.2 bibliothèque Esterel complémentaire

Cette bibliothèque contient les comportements des entités STEP (§ 5.3.2.1) et ACTION_N_MANAGER. Elle est nécessaire pour compiler le code ESTEREL issu de G2E. Elle peut être générée de manière automatique en IC grâce aux options "-selfcont3" et "-selfcont5".

```
module STEP:
input Xk_INIT, Xk_SET, Xk_RESET;
output Xk, ZXk, RTLk, REk, FEk;
signal SKIP_INACTIVE, Xk_stable in

present Xk_INIT then
  emit RTLk;
  present Xk_RESET then
    emit FEk;
    present Xk_SET then
      emit Xk_stable;
      emit SKIP_INACTIVE
    end present
  else
    emit Xk_stable;
    emit SKIP_INACTIVE
  end present
else
  present Xk_SET then
    emit REk;
    emit RTLk;
    present Xk_RESET then
      emit FEk
    else
      emit Xk_stable;
      emit SKIP_INACTIVE
    end present
  end present
end present;
loop
```

```
  present SKIP_INACTIVE else
    trap INACTIVE in
      every tick do
        present Xk_SET then
          emit REk;
          emit RTLk;
          present Xk_RESET then
            emit FEk
          else
            emit Xk_stable;
            exit INACTIVE
          end present
        end present
      end every
    end trap
  end present;

  trap ACTIVE in
    every tick do
      emit RTLk;
      present Xk_RESET then
        emit FEk;
        present Xk_SET then
          emit REk;
          emit Xk_stable
        else
          exit ACTIVE
        end present
      else
        emit Xk_stable
      end present
    end every
  end trap
end loop
||
[
  present Xk_INIT then
    emit Xk
  end present;
  loop
    present Xk_stable then
      emit ZXk;
      await tick;
      emit Xk
    else
```

```
      await tick
    end present
  end loop
]
end signal
end module


module action_N_manager:
input ACTk;
output ACTk_set, ACTk_reset;
signal ACTk_pre in
  loop
    % regeneration du ACTk_pre a partir du ACTk
    present ACTk then
      await tick;
      emit ACTk_pre
    else
      await tick
    end present
  end loop
  ||
  loop
    present ACTk then
      present ACTk_pre else
        emit ACTk_set
      end present
    else
      present ACTk_pre then
        emit ACTk_reset
      end
    end
  each tick
end signal
end module
```

## B.2 Le compilateur Grafcet-OC : G2OC

### B.2.1 Options de compilation de G2OC

La syntaxe générale de G2OC est:

```
g2oc <nom_fichier> [-noaction] [-simul] [-pure] [-MPI]
   [-explain] [-bac] [-oc4]
```

```
OU g2oc -h
```

**options:**

**-h :** donne la liste des options.

**-noaction :** considère les actions continues GRAFCET comme des *outputs* ESTEREL à émettre à chaque *instant*. Cette option est identique à celle du compilateur G2E. En particulier, le comportement du grafcet de la figure B.1 pourra être modélisé suivant les deux automates de la figure B.2.

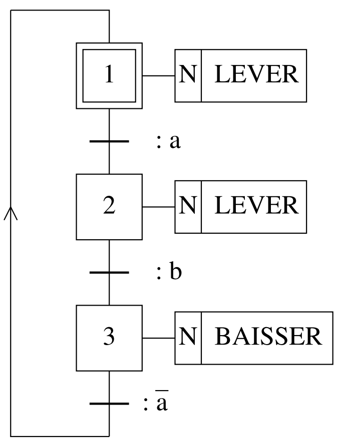



FIG. B.1 - *Exemple simple*

**Remarque :** La génération des actions impulsionnelles n'est pas affectée par cette option. Ces dernières sont en effet sémantiquement équivalentes à des *outputs* ESTEREL.

**-simul :** génère les signaux de sortie ZX<etape>_EXPORTED qualifiant l'état des étapes GRAFCET. Cette option est obligatoire avec EG7 pour permettre la simulation du grafcet.

**-pure :** génère tous signaux d'entrée comme des *pure inputs* ESTEREL. Normalement seuls les événements sont générés de cette manière. Cette option est obligatoire avec EG7.

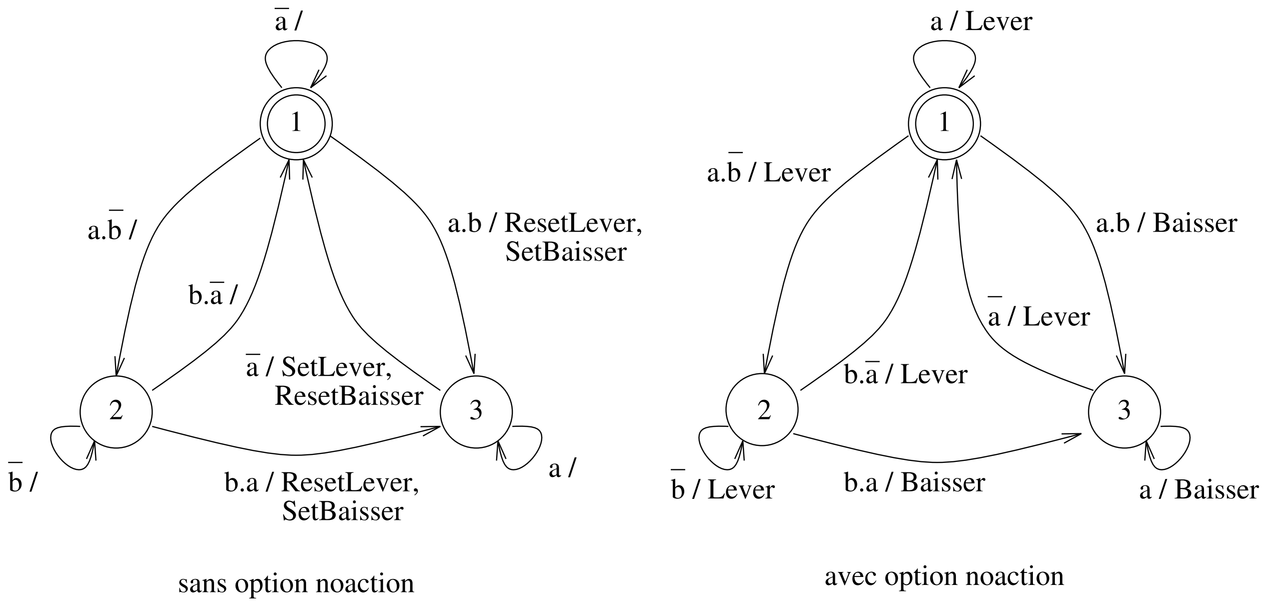



FIG. B.2 - *Automates de comportement équivalent*

**-MPI :** utilise l'algorithme MPI classique (§ 3.3.2.2) pour déterminer les états "suivants". Le compilateur utilise en standard l'algorithme PFI présenté au chapitre 5.4.2 basé sur la sémantique S-GRAFCET. L'activation de cette option change également la sémantique des variables $Xk$ pour permettre l'intégration de la double échelle de temps de la norme GRAFCET.

**-explain :** explicite dans le fichier "LOG", l'analyse du fichier "GT" et le déroulement de l'algorithme de recherche des "états suivants".

**-bac :** génère un fichier "BA" d'équations booléennes pour l'analyseur BAC. Cette option est incompatible avec -oc4.

**-oc4 :** génère l'automate au format OC4 (par défaut : OC5).

# Annexe C

# Expression des observateurs

Nous décrivons ici l'ensemble des observateurs qui n'ont pas été présentés dans le chapitre 6.

## alwaysA_afterOratB

Par cet observateur nous pouvons exprimer qu'une propriété `A` doit rester vraie dès l'apparition d'un facteur `B`. Cet opérateur diffère de "neverA_afterOratB(A,B)" par sa sensibilité à $\neg$ `A` au lieu de `A`.

## A_afterOratB

Tant que `B` est absent, `A_afterOratB(A,B)` rend `vrai`. L'occurrence de B force la sortie à `faux` jusqu'à l'occurrence de A. Cet opérateur n'est pas l'opérateur `AsinceB(A,B)` de [Hal93], qui est réinitialisé à chaque nouvelle occurrence de `B`.

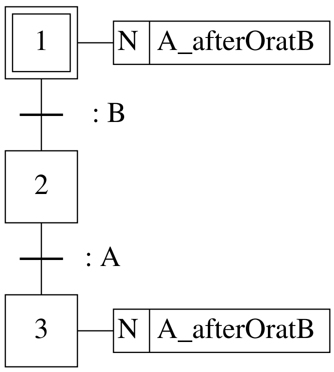



FIG. C.1 - *Observateur "A_afterOratB(A,B)"*

## onceA_afterOratB

Cet observateur diffère du précédent par l'unicité de l'occurrence de `A`. Nous remarquons l'utilisation du garde événement "tick" spécifique du S-GRAFCET. La transition ainsi gardée devient sensible à `A` uniquement lors de la prochaine évolution du système. En GRAFCET standard, cette propriété serait difficile à exprimer.

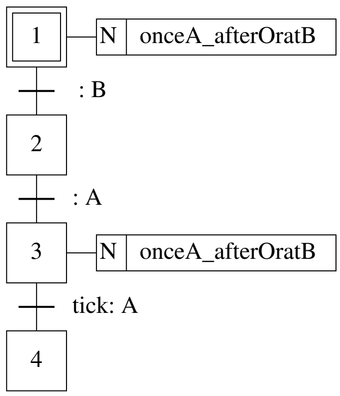



FIG. C.2 - *Observateur "onceA_afterOratB(A,B)"*

## onceB_afterOratA_beforeC

L'opérateur `onceB_afterOratA_beforeC(A,B,C)` rend `vrai` s'il existe une et une seule occurrence de `B` dans l'intervalle temporel `[A,C[`. Si `A` et `C` sont présents en même temps, la valeur `faux` est rendue.

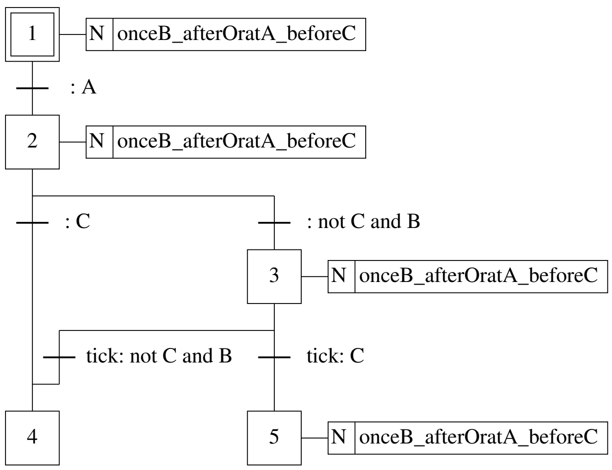



FIG. C.3 - *Observateur "onceB_afterOratA_beforeC(A,B,C)"*

## neverB_afterOratA_beforeC

L'opérateur `neverB_afterOratA_beforeC(A,B,C)` rend `vrai` s'il n'existe pas d'occurrence de `B` dans l'intervalle temporel `[A,C[`. Si `A` et `C` sont présents en même temps, la valeur `vrai` est rendue.

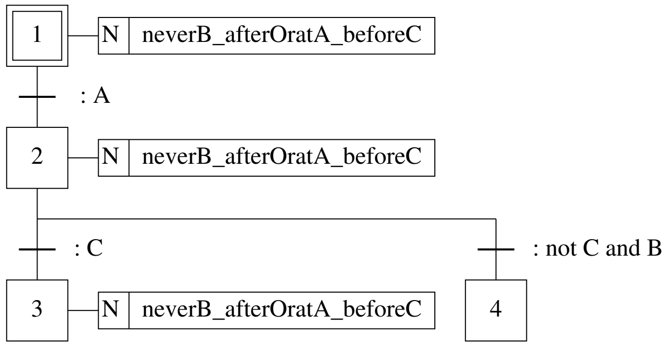



FIG. C.4 - *Observateur "neverB_afterOrAtA_beforeC(A,B,C)"*

## alwaysB_afterOratA_beforeC

L'opérateur `alwaysB_afterOratA_beforeC(A,B,C)` rend `vrai` si `B` est toujours vrai dans l'intervalle temporel `[A,C[`. Si `A` et `C` sont vrais en même temps, la valeur de `B` à cet instant est rendue.

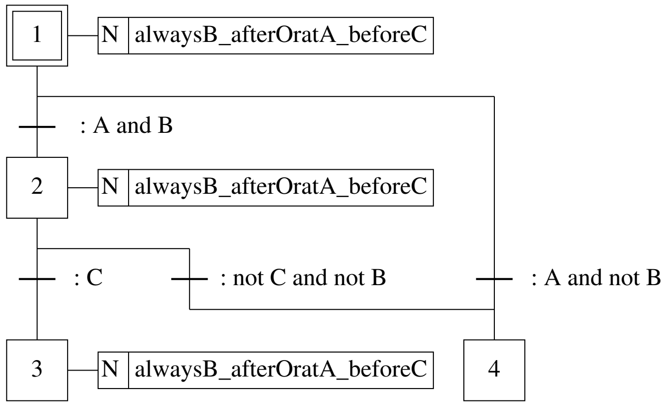



FIG. C.5 - *Observateur "alwaysB_afterOratA_beforeC(A,B,C)"*

# Bibliographie


[ABC94] A. Arnold, D. Begay, and P. Crubillé. *Construction and Analysis of Transition Systems with* MEC, volume 3 of *AMAST Series in Computing*. World Scientific, Singapore, 1994.

[AD93] P. Aygalinc and J.P. Denat. Validation of functional GRAFCET models and performance evaluation of the associated system using petri nets. *APII*, 27(1):81–93, january 1993.

[AFN82] AFNOR, Paris (France). *Diagramme fonctionnel* GRAFCET *pour la description des systèmes logiques de commande*, juin 1982. Norme Française, NF C 03 190.

[AFN93] AFNOR, Paris (France). *Diagramme fonctionnel* GRAFCET. *Extension des concepts de base*, juin 1993. Norme Française, NF C 03 191.

[AG94] C. André and D. Gaffé. Coopération GRAFCET/ESTÉREL. In *Colloque AGI'94*, pages 221–224, Poitiers, Juin 1994. Association AGI.

[AMP91] C. André, J-P. Marmorat, and J-P. Paris. An execution machine for ESTEREL. In *ECC'91*, volume 2, pages 1672–1677, Grenoble, France, July 1991. Hermès.

[AP92a] C. André and M-A. Péraldi. GRAFCET et langages synchrones. In *Conférence Grafcet'92*, pages 91–100, Paris, Mars 1992. Afcet.

[AP92b] C. André and M-A. Péraldi. Hard real-time system implementation on a microcontroller. In *IBRA/BIRA International Workshop on real-time programming*, pages 185–189, Bruges (Belgium), June 1992. IFAC.

[AP93] C. André and M-A. Péraldi. GRAFCET and synchronous languages. *A.P.I.I.*, 27(1):95–105, 1993.

[Arn89] A. Arnold. MEC: a system for constructing and analysing transition systems. volume 407. LNCS, Springer-Verlag, 1989.

[Arn92] A. Arnold. *Systèmes de transition finis et sémantique des processus communicants*. Etudes et recherches en informatique. Masson, Paris, 1992.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceeding of the IEEE*, 79(9):1270–1282, September 1991.

[BBC+92] N. Bouteille, P. Brard, G. Colombari, N. Cotaina, and D. Richet. *Le* GRAFCET. Cépaduès Editions, France, 1992.

[BC84] G. Berry and L. Cosserat. The synchronous programming languages Esterel and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer Verlag Lecture Notes in Computer Science 197, 1984.

[BdS91] F. Boussinot and R. de Simone. The Esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1293–1304, 1991.

[Ben89] A. Benveniste. Les langages synchrones: des logiciels pour la spécification et la conception des systèmes temps-réel de traitement de l'information. Technical report, INRIA/IRISA, Mai 1989. C2A.

[Ber89] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.

[Ber92] B. Berkane. *Vérification des systèmes matériels numériques séquentiels synchrones: Application du langage* LUSTRE *et de l'outil de vérification* LESAR. PhD thesis, Institut Polytechnique de Grenoble, Octobre 1992.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[BGP93] J-L. Bergerand, P. Ghaleb, and D. Pilaud. Utilisation d'un langage synchrone à flôts de données pour la réalisation de logiciels temps-réel embarqués. Conférence Real-time systems, Paris (France), Janvier 1993.

[Bou93] F. Boulanger. *Intégration de Modules Synchrones dans la Programmation par Objets*. PhD thesis, Supélec / Université de Paris-sud, Centre d'Orsay, Décembre 1993.

[BRR87] W. Brauer, W. Reisig, and G. Rozenberg. *Petri Nets: central models and their properties*, volume 254–255 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transaction on Computers*, C-35(8):677–691, 1986.

[Bur84] Bureau d'orientation de la normalisation en Informatique. SCEPTRE: proposition de noyau normalisé pour les exécutifs temps-réel. *Techniques et Sciences Informatique*, 3:45–62, 1984.

[Cal90] J-P. Calvez. *Spécification et Conception des Systèmes: une méthodologie.* Masson, 1990.

[Cas87] E. Castelain. *Modélisation et simulation interactive de cellules de production flexibles dans l'industrie manufacturière.* PhD thesis, Université de Lille, 1987.

[CHPR91] P. Caspi, N. Halbwachs, D. Pilaud, and P. Raymond. The synchronous data flow programming language LUSTRE. *Proceeding of the IEEE*, 79(9):1305–1320, September 1991.

[CIS88] CISI Ingénierie, Valbonne (FRANCE). ESTEREL C *Simulation Manual*, 1988.

[CIS91] CISI, Valbonne (France). ESTEREL V3*: User's manuals*, 1991.

[DA89] R. David and H. Alla. *Du* GRAFCET *aux réseaux de Petri.* Automatique. Hermès, Paris, 1989.

[Dav95] R. David. Grafcet: A powerful tool for specification of logic controllers. *IEEE Transactions on Control Systems Technology*, 3(3):253–268, September 1995.

[Elk93] S. Elkhattabi. *Intégration de la surveillance de bas niveau dans la conception des systèmes à événements discrets: Application aux systèmes de production flexibles.* PhD thesis, Université des Sciences et Technologies de Lille, Septembre 1993.

[Ell88] J-P. Elloy. Groupe de réflexion temps-réel du CNRS: Le temps réel. *TSI*, 7:493–500, 1988.

[Ell92] Sentowitch Ellen. Sis: a system for sequential circuit synthesis. Technical report, University of California, Berkeley, may 1992. Memorandum UCB/ERL M92/41.

[FC92] J.P. Frachet and G. Colombari. Elements for a semantics of the time in GRAFCET and dynamic systems using non-standard analysis. *APII*, 27(1):107–125, january 1992.

[Fer88] J-C. Fernandez. *Aldébaran : un système de vérification par réduction de processus communicants.* PhD thesis, Université de Grenoble, 1988.

[For95] F-X. Fornari. *Optimisation du contrôle et implantation en circuits de programmes Esterel.* PhD thesis, Université de Nice-Sophia Antipolis, Mars 1995.

[Gaf91] D. Gaffé. Conception d'un exécutif temps-réel pour esterel. Technical report, Ecole Doctorale SPI – DEA Informatique, Signaux et Systèmes, Septembre 1991. Rapport de stage de DEA.

[Ghe92] G. Gherardi. SAHARA*: un environnement de mise au point graphique pour les programmes* ESTEREL. PhD thesis, Université de Nice-Sophia Antipolis, Decembre 1992.

[Gon88] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones. Application à* ESTEREL. PhD thesis, Université de Paris-sud, Centre d'Orsay, Mars 1988.

[GRE85] GREPA. *Le* GRAFCET*: de nouveaux concepts.* Cépaduès, France, 1985.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, Amsterdam, 1993.

[Hal94] N. Halbwachs. BAC: A boolean automaton checker. Technical report, VERIMAG, Montbonnot (France), February 1994.

[Har87] D. Harel. STATECHARTS: A visual formalism for complex systems. *Science of computer programming*, 8:231–274, 1987.

[HCRP90] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactif à l'aide du langage flot de données synchrone LUSTRE. Technical Report L9, IMAG, Janvier 1990.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs: Le langage LUSTRE. *Technique et Science Informatique*, 10(2):139–157, 1991.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the develpment of complex reactive systems. *IEEE Transaction on Software Engineering*, 16:477–498, 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems in logic and models of concurrent systems. *NATO ASI Series, K.R Apt Ed., Springer-Verlag*, 13:477–498, 1985.

[HPOG88] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.C. Glory. Specifying, programming and verifying real-time systems using a synchronous declarative language. Technical Report L6, IMAG, July 1988.

[HS91] W.A. Halang and A.D. Stoyenko. *Constructing Predictable Real-Time Systems.* Kluwer Academic Publishers, Amsterdam, 1991.

[HS92] W.A. Halang and K.M. Sacha. *Real-Time Systems: Implementation of Industrial Computerised Process Automation.* World Scientific, Singapore, 1992.

[IEC84] IEC, Genève (CH). *Binary Logical operators*, january 1984. International standard IEC 617-12.

[IEC88] IEC, Genève (CH). *Preparation of function charts for control systems*, december 1988. International standard IEC 848.

[IEC93] IEC, Genève (CH). *standard for programmable controllers part 3: Programming languages*, 1993. International standard IEC 1131.

[LA90] S.T. Levi and A.K. Agrawala. *Real-Time System Design.* Mc Graw-Hill, New York, 1990.

[Lap93] P.A. Laplante. *Real-Time Systems: Design and Analysis, an engineer's Handbook.* IEEE Press, 1993.

[Lep94] P. Leparc. *Apports de la méthodologie synchrone pour la définition et l'utilisation du langage Grafcet.* PhD thesis, Université de Rennes 1, Janvier 1994.

[LGLL91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceeding of the IEEE*, 79(9):1321–1336, September 1991.

[LPR92] P. Lhoste, H. Panetto, and M. Roesch. GRAFCET: from syntax to semantics. *APII*, 27(1):127–141, january 1992.

[LR92] J-J. Lesage and J-M. Roussel. Preuve de la cohérence d'une hiérarchie entre grafcets partiels. pages 125–134, Paris, Mars 1992. Conférence Grafcet'92, Afcet.

[Mar90] F. Maraninchi. ARGOS*: un langage graphique pour la conception, la description et la validation des systèmes réactifs*. PhD thesis, Université Joseph Fourier, Grenoble I, Janvier 1990.

[M.B79] M.Blanchard. *Comprendre maîtriser et appliquer le GRAFCET*. Cépaduès édition, France, 1979.

[Mig94] F. Mignard. *Compilation du langage Esterel en Systèmes d'équations booléennes*. PhD thesis, Université de Nice", octobre 1994.

[ML92] L. Marcé and P. Le Parc. Modélisation de la sémantique du GRAFCET à l'aide de processus synchrones. In *Conférence Grafcet'92*, pages 101–110, Paris, Mars 1992. Afcet.

[Moa85] M. Moalla. Réseaux de petri interprétés et grafcet. *Rairo Automatique*, 4(1):17–30, 1985.

[Pan91] H. Panetto. *Une contribution au génie automatique: Le prototypage des machines et systèmes automatisés de production*. PhD thesis, Université de Nancy I, Janvier 1991.

[Par92] J-P. Paris. *Exécution de tâches asynchrones depuis* ESTEREL. PhD thesis, Université de Nice-Sophia Antipolis, Juillet 1992.

[Pas94] L. Pastorelli. Machine d'exécution pour le formalisme synchrone GRAFCET. Technical report, Ecole Doctorale SPI – DEA Informatique, Juin 1994. Rapport de stage de DEA.

[Per93] M-A. Peraldi. *Conception et Réalisation de Systèmes Temps-Réel par une Approche Synchrone*. PhD thesis, Université de Nice-Sophia Antipolis, Juillet 1993.

[Pet80] C.A. Petri. Introduction to general net theory. In W. Brauer, editor, *Net theory and Applications*, volume 84 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 1980.

[PH88] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. Number 331 in LNCS. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer-Verlag, September 1988.

[PS90] J.P. Paris and J.B. Saint. The LUSTRE - ESTEREL portable format (version oc3). Technical report, Ecole Nationale Supérieure des Mines de Paris, 1990.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proc. Theoretical Aspects of Computer Sciences*. TACS-91, Springer-Verlag, LNCS 526, Sept. 1991.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederic Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, 1991.

[Rei85] W. Reisig. *Petri nets: an introduction*. Monograph on Theoretical Computer Science. Springer-Verlag, Berlin, 1985.

[RL93] J.M. Roussel and J.J. Lesage. Une algèbre de boole pour l'approche événementielle des systèmes logiques. *APII*, 27(5):541–560, 1993.

[Rou94] J. M. Roussel. *Analyse de Grafcets par génération logique de l'automate équivalent*. PhD thesis, École Normale Supérieure de Cachan, décembre 1994.

[Roy90] V. Roy. Autograph: Un outil de visualisation pour les calculs de processus. Thèse d'informatique, Université de Nice, 1990.

[RR94] O. Roux and V. Rusu. Du GRAFCET au langage réactif ELECTRE. *APII*, 28(2):131–157, 1994.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. J. Wiley Publ., 1994.

[Sif82] J. Sifakis. Property preserving homomorphisms and a notion of simulation for transition systems. Technical Report RR 332, IMAG, November 1982.

[Ver87] D. Vergamini. Vérification de réseaux d'automates finis par équivalence observationnelle: le système Auto. Thèse d'informatique, Université de Nice, 1987.

[von94] M. von der Beeck. A comparison of statecharts variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. FTRTFT'94, Springer-Verlag, 1994.

[Zah80] J. Zahnd. *Machines Séquentielles*, volume XI of *Traité d'Electricité*. Editions Georgi, Suisse, 1980.