UNIVERSITÉ LIBRE DE BRUXELLES

Faculté des Sciences

Département Informatique

Compilation et répartition de programme ELECTRE pour Lego Mindstorms

MEUTER Cédric

Mémoire réalisé sous la direction de Monsieur le Professeur Jean-Francois Raskin en vue de l'obtention du grade de maître en informatique

Année académique 2001-2002

Préface

Dans le cadre d'études en informatique à l'Université Libre de Bruxelles, nous avons été amenés à réaliser un stage de 300 heures, soit environ 4 mois, du 1^{er} octobre 2001 au 31 janvier 2002. C'est à l'Institut de Recherche en Communication et Cybernétique de Nantes (IRCCyN), en France, que nous avons réalisé ce stage sous la tutelle de M. Franck Cassez. Pendant ces quatre mois, nous avons été amenés à étudier la compilation et répartition de programme ELECTRE pour Lego Mindstorms. Ce document présente le travail réalisé durant ces quatre mois.

Table des matières

Introduction

Systèmes réactifs

Notre travail se base sur le langage ELECTRE. Ce langage permet de modéliser des *systèmes réactifs*, c'est-à-dire des systèmes qui évoluent en réagissant à des stimuli de leur environnement. Ce type de système est utilisé pour modéliser des dispositifs physiques qui répondent à des requêtes par des actions apropriées, afin de maintenir la stabilité du système. Par exemple, un système de contrôle de température réagira à un signal marquant la détection d'une température trop élevée par une diminution de l'intensité du chauffage.

En matière de programmation de tels systèmes réactifs, la gestion des stimuli (requêtes), modélisés par des *événements discrets*, se fait selon deux approches :

1. La première est la *programmation synchrone*. Dans cette approche, il est fait hypothèse que le temps nécessaire à la prise en compte d'une occurrence d'événement est négligeable. Cette prise en compte est donc considérée comme instantanée. Sous cette hypothèse, le système est toujours à même de prendre en compte une occurrence d'événement. Malheureusement, en pratique, cette hypothèse n'est que rarement vérifiée.
2. La *programmation asynchrone* y constitue une alternative. Au contraire de son homologue *synchrone*, aucune hypothèse n'est faite quant au temps de prise en compte d'une occurrence d'événement. Dès lors, il se peut qu'une telle occurrence survienne alors que le système n'est pas à même de le prendre en compte. Se fait sentir, alors, le besoin de mémoriser de telles occurrences. C'est dans cette catégorie que s'inscrit le langage ELECTRE qui nous intéresse ici.

Model checking

De nos jours, dans la conception de systèmes réactifs, la vérification prend une place de plus en plus importante. Le *model checking* offre une technique qui permet de vérifier

de tels systèmes de manière automatique. Ce processus de model checking se découpe en trois grandes étapes :

1. **Modélisation** La première étape consiste à réaliser un modèle du système suivant un certain formalisme. Dans notre cas, le formalisme utilisé est le langage ELECTRE. Si le système est trop complexe, une certaine forme d'abstraction devra être introduite au cours de cette étape.
2. **Spécification** Avant d'entamer la vérification proprement dite, il est nécessaire d'énumérer un certain nombre de propriétés, dites comportementales, que le système doit satisfaire afin d'être considéré comme correct. Il existe deux types de telles propriétés :
   (a) *propriétés de sûreté* (*safety properties*) qui permettent de s'assurer que le système n'aboutit pas dans un état non désiré.
   (b) *propriétés de vivacité* (*liveness properties*) qui permettent de s'assurer que le système est toujours en mesure d'évoluer, autrement dit que quelque chose finira toujours par avoir lieu.

   Ces propriétés sont généralement décrites sous la forme de formule de logique temporelle.
3. **Vérification** La dernière étape consiste à vérifier que le modèle satisfait les propriétés. Cette étape devrait idéalement être automatique, mais dans la pratique, elle nécessite une intervention humaine.

Motivation du travail

Une fois l'étape de vérification terminée, le modèle devra être implémenté. Si cette étape est réalisée "manuellement", elle peut être source d'erreurs. Afin d'en empêcher l'introduction, il apparaît donc nécessaire de systématiser ce passage du modèle à l'implémentation.

De plus, dans le cadre de système distribué, après la réalisation et vérification d'un modèle global, une phase de répartition vient s'intercaler avant l'implémentation. Cette étape peut, elle aussi, être source d'erreurs. A nouveau, une systématisation de cette étape en empêchera l'introduction.

Ceci nous conduit donc à l'élaboration d'outils de compilation et répartition, qui permettent, après avoir construit et validé un modèle en ELECTRE, d'en réaliser une implémentation distribuée en Lego Mindstorms fiable, en ce sens qu'elle maintient les propriétés de ce modèle.

Plan du document

Ce document se découpe en deux parties. Dans la première, nous installons les différentes notions nécessaires à la compilation et à la répartition de programme ELECTRE. Après avoir introduit brièvement notre plate-forme de travail, à savoir les Lego Mindstorms dans le chapitre 1, nous détaillons le langage ELECTRE dans le chapitre 2, de manière constructive au travers de multiples exemples. Nous poursuivons ensuite dans le chapitre 3 avec le modèle des FIFFO-AUTOMATES, utilisé comme structure intermédiaire, à la fois pour la compilation et pour la répartition. Cette première partie se termine, au chapitre 4, où nous introduisons la logique temporelle qui permet de spécifier des propriétés comportementales sur des FIFFO-AUTOMATES.

La deuxième partie, quant à elle, aborde les problèmes de compilation et de répartition, respectivement dans les chapitres 5 et 6. Pour conclure notre étude, nous présentons dans le chapitre 7 une étude de cas reprenant les méthodes abordées.

En annexe A le lecteur trouvera des informations supplémentaires sur la programmation des Lego Mindstorms. L'annexe B fournit pour terminer un complément sur le langage ELECTRE, sous la forme de la grammaire du langage et de sa sémantique opérationnelle.

Première partie

Notions abordées

Chapitre 1

Lego Mindstorms

Les Lego Mindstorms s'articulent autour d'une brique Lego particulière : la brique RCX . Pour le développement de cette brique, Lego s'est fortement inspiré d'un projet du *Massachusetts Institute of Technology* : la *MIT Programmable Brick*. Les figures 1.1(a) et 1.1(b) présentent respectivement la brique RCX et son précurseur, la *MIT Programmable Brick*. Dans la section 1.1, nous présentons cette brique RCX. Ensuite, dans la section 1.2, nous présentons différentes possibilités pour la programmer.

(a) La brique RCX

(b) MIT Programmable Brick

FIG. 1.1 – Briques programmables

1.1 La brique RCX

La brique RCX n'est autre qu'un microcontrôleur Hitachi 8300, avec trois entrées et trois sorties et un écran LCD. Les premières versions ne fonctionnaient que sur piles. Les versions plus récentes (à partir de 2.0), disposent d'un adaptateur secteur. Elle est vendue avec un ensemble de composants lui permettant d'interagir avec le monde extérieur. Ces différents composants sont présentés à la figrue 1.1(a). Parmi ces composants, nous distinguons principalement deux catégories :

1. les moteurs : ils sont au monbre de deux par boîte Lego Mindstorms. Grâce à la brique RCX, l'utilisateur peut les faire fonctionner dans un sens de rotation, ou dans l'autre, à vitesse souhaitée.
2. les capteurs : ils sont, quant à eux, au nombre de trois. Parmi ceux-ci, on compte deux capteurs de touché, qui permettent, par exemple, de détecter des obstacles ou de détecter la fermeture d'une barrière. Le capteur restant est un capteur de lumière, ou plus exactement, un capteur d'intensité lumineuse. Il permet, par exemple, à la brique RCX de faire la différence entre du blanc et du noir.

Cette brique RCX est programmée grâce à un ordinateur. Les programmes sont développés sur machine, pour ensuite être téléchargés sur la brique RCX, via son port infra-rouge. De plus, deux briques RCX peuvent communiquer entre elles via ce même port infra-rouge.

1.2 Développement Lego Mindstorms

Chaque boîte Lego Mindstorms est fournie avec un programme de développement qui permet, grâce à une interface graphique simple, de programmer la brique RCX. Cependant, pour les buts qui nous importent ici, ce programme se révèle insuffisant. Heureusement, un certain nombre d'alternatives ont été développées pour programmer la brique RCX. Avant d'aborder les problèmes qui nous incombent dans ce document, il était impératif de choisir une plate-forme adaptée. Nous avons exploré les différentes solutions brièvement présentées ci-après.

Not Quit C

Not Quit C est un langage de programmation relativement proche du C développé par Dave Baum [?]. Les programmes Not Quit C sont compilés en code binaire compatible avec le firmware de la brique RCX, comme illustré à la figure 1.2. Ceci permet d'utiliser

à la fois des programmes développés avec l'environnement graphique proposé par Lego, et des programmes Not Quit C. Cependant, en contrepartie, un programme Not Quit C doit respecter les différentes contraintes imposées par ce firmware :

- les variables doivent être de type entier (pas de support string ou point flottant),
- le nombre de variables est limité à 32 globales pour la version 1.0 du firmware, et à 32 globales et 16 locales pour la version 2.0,
- les vecteurs sont limités en taille et en dimension.

Ces différentes contraintes nous ont très vite amenés à éliminer Not Quit C comme choix de plate-forme d'implémentation, la principale limitation étant celle du nombre de variables.

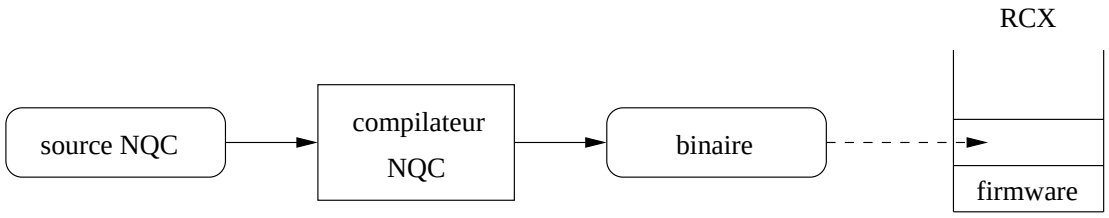


FIG. 1.2 – Processus de compilation avec Not Quit C

lejOS

lejOS [?] offre la première alternative au firmware de la brique RCX. Il propose une JVM[1] réduite remplaçant ce firmware et qui interprète du bytecode. Comme montré à la figure 1.3, ce bytecode est obtenu à partir de programme Java et compilé par lejOS. Le premier avantage évident de cette solution est que les différentes contraintes imposées précédemment ne sont plus de rigueur. Le nombre de variables n'est dès lors plus limité, un support string réduit est disponible. Tout ceci semble faire de lejOS le candidat idéal. Cependant, l'absence d'un *garbage collector*[2] se fait cruellement sentir dès lors que le programme devient relativement important. De plus, la JVM occupe une place importante sur la brique (9Kb), ce qui pour la brique RCX, n'est pas négligeable au vu de la quantité de mémoire dont elle dispose (32Kb).

[1] Java Virtual Machine

[2] tâche qui s'occupe de récupérer à l'exécution des espaces mémoires n'étant plus utilisés

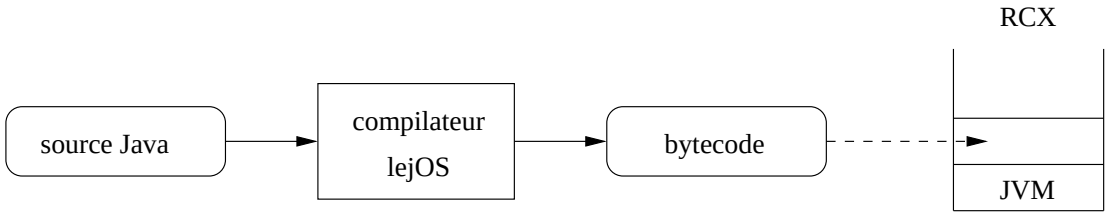


FIG. 1.3 – Processus de compilation avec lejOS

legOS

legOS [?] est, quant à, lui un système d'exploitation pour la brique RCX. Il remplace également le firmware de la brique et permet ainsi de lever les limitations de ce dernier. Un programme legOS est en fait rédigé en C/C++, pour ensuite être compilé par une *cross-compiler*[3] pour la brique RCX. Le code binaire ainsi obtenu est exécuté sur la brique. Le principal avantage de legOS par rapport à lejOS est la compacité du code ainsi produit. De plus, l'existence d'un protocole réseau (LNP[4]) permet de gérer des communications de manière plus fine. De plus amples détails sur la programmation avec legOS peuvent être trouvés en annexe A. Celle-ci présente les principales fonctions qui permettent de contrôler la brique RCX.

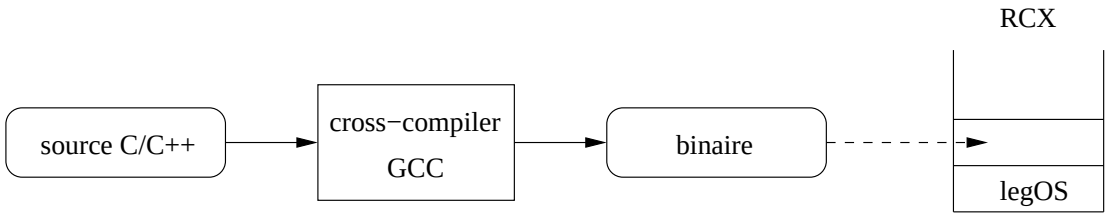


FIG. 1.4 – Processus de compilation avec legOS

[3]compilateur dont le binaire produit est destiné à une machine différente de celle sur laquelle s'exécute le compilateur

[4]*Lego Network Protocol*

Chapitre 2

Le langage ELECTRE

ELECTRE est un langage réactif asynchrone qui permet de décrire l'évolution de tâche, constituant une application, en fonction d'événements. Il permet d'exprimer le comportement de ces tâches, les unes par rapport aux autres (séquencement, parallélisme, répétition), ainsi que des tâches par rapport aux événements (préemption, activation). Ce langage s'articule autour de deux types d'entités : les *modules* et les *événements*, présentés respectivement aux sections 2.1 et 2.2. Tout programme du langage ELECTRE est construit sur base de ces entités au moyen d'*opérateurs*. Dans la section 2.3, nous présentons de manière constructive ces différents *opérateurs* au travers d'exemples, illustrés de chronogrammes. Nous terminons en donnant, à la section 2.4, des solutions au problème des lecteurs/écrivains.

2.1 Les modules

Dans un programme ELECTRE, les *modules* représentent les tâches composantes d'une application. Un module est un morceau de code séquentiel qui ne contient pas de point de synchronisation bloquant (où la tâche est en attente d'un événement). Ces modules peuvent être écrits dans des langages de programmation distincts, indifféremment du programme ELECTRE.

La durée d'exécution d'un module doit être finie et non nulle. La seule exception à cette règle est le module *idle*, dénoté par "1", qui représente l'attente active, dont la durée d'exécution est infinie. L'intérêt de ce module est que certaines applications nécessitent l'introduction d'une tâche pendant laquelle le processeur n'a aucun travail particulier à effectuer, si ce n'est d'attendre l'occurrence d'un événement.

Dans un programme ELECTRE, les identifiants de modules sont reconnaissables au fait qu'ils commencent par une majuscule. Nous noterons dans la suite par M, l'ensemble de tels identifiants d'un programme ELECTRE.

2.2 Les événements

Le rôle des événements, dans un programme ELECTRE, est de faire évoluer l'exécution des modules de ce programme. Lorsqu'une occurrence d'événement est prise en compte, cela peut avoir pour conséquence que certains modules soient préemptés et d'autres activés. Nous insistons sur le fait que c'est bien lorsque l'occurrence est prise en compte, et non lorsque l'occurrence survient que l'activation et la préemption se produisent. En effet, de par la nature asynchrone du langage, certains événements devront être mémorisés avant d'être pris en compte. Nous pouvons distinguer différents types d'événements, de par la politique de gestion de leurs occurrences :

1. Les événements *à mémorisation unique*, pour lesquels une occurrence, au plus, peut être mémorisée. Ces événements sont dits consommés lorsque les modules éventuellement activés lors de la prise en compte se terminent. Entre l'instant de prise en compte et l'instant de consommation, l'événement est dit *vivant*. Ces événements serviront, typiquement, à modéliser une information binaire, comme par exemple, une barrière s'est ouverte, ou s'est fermée.
2. Les événements *à mémorisation multiple*, pour lesquels chaque occurrence est mémorisée. L'instant de consommation est le même que pour les événements à mémorisation multiple. Ce type d'événements servira, par exemple, à modéliser l'arrivée d'un composant sur un tapis roulant, dont chaque occurrence doit être traitée.
3. Les événements *fugaces* dont aucune occurrence ne sera mémorisée. Lorsque l'occurrence d'un tel événement se produit, soit elle est prise en compte directement, soit elle est perdue. L'instant de consommation, pour ces événements, est confondu avec l'instant de prise en compte. Ces événements ne seront, par conséquent, jamais vivants. Ce type d'événements sert à modéliser des informations éphémères.
4. Les événements *à consommation immédiate*, qui du point de vue de la mémorisation se comportent comme les événenements à mémorisation unique. Ils se démarquent de ces derniers par le fait que l'instant de leur consommation est confondu avec l'instant de prise en compte. Ils ne seront donc jamais vivants.

La figure 2.1 récapitule les propriétés de mémorisation et de vivacité des différents types d'événements.

	mémorisation	**vivacité**
simple	au plus 1 fois	oui
mémorisation multiple	autant que nécessaire	oui
fugace	jamais	non
consommation immédiate	au plus une fois	non

FIG. 2.1 – Type d'événements

Les événements mémorisés seront traités dès que possible, la priorité étant donnée au plus anciens événements. Cette politique de gestion, appelée *first input first fireable output* (FIFFO), est expliquée plus en détail dans le chapitre suivant, à la section 3.2.2

Les identifiants d'événements, contrairement aux identifiants de modules, commencent par une minuscule. Dans la suite, nous noterons par E, l'ensemble de ces identifiants.

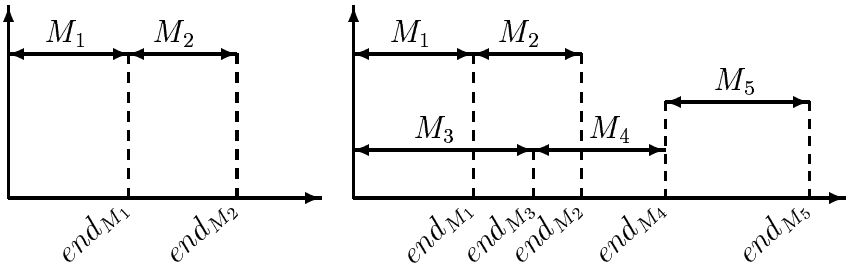
Remarquons que les modules sont capables d'émettre des événements. Un bon exemple est la terminaison d'un module. Un module M_1, pour marquer la fin de son exécution, produira une occurrence de l'événement end_{M_1} lorsqu'il se termine.

2.3 Les opérateurs

Dans cette section, nous allons, petit à petit, introduire les différents opérateurs qui permettent de construire un programme ELECTRE. Notre intention n'est pas ici de donner une définition formelle au langage mais bien de fournir une intuition quant au comportement de ces opérateurs. Le lecteur intéressé pourra touver en annexe B la grammaire du langage, ainsi que sa sémantique opérationnelle.

2.3.1 Séquencement et parallélisme

Dans un premier temps, les seules entités dont nous allons nous préoccuper, sont les modules. Nous introduisons ici deux opérateurs, qui permettent de construire des *structures de modules*. Ces opérateurs sont le $';'$ pour le séquencement et le $' \,||'$ pour le parallélisme. Ils s'appliquent tous deux, bien entendu, aux modules, mais aussi aux structures de modules. Le séquencement se traduit par les exécutions successives des parties gauche et droite de l'opérateur. Quant au parallélisme, il spécifie que les deux branches de l'opérateur s'exécutent simultanément et qu'il est nécessaire à ces deux branches de se terminer pour pouvoir continuer l'exécution du programme. Remarquons que le groupement de modules en structure est noté entre crochets.



(a) $M_1; M_2$ (b) $[\ [\ M_1; M_2\ ] \mid\mid [\ M_3; M_4\ ]\ ]; M_5$

FIG. 2.2 – Séquencement et parallélisme

Exemple 2.1

Nous considérons les programmes $M_1; M_2$ et $[\ [\ M_1; M_2\ ] \mid\mid [\ M_3; M_4\ ]\ ]; M_5$ dont les comportements respectifs sont illustrés par les chronogrammes des figures 2.2(a) et 2.2(b). Dans la figure 2.2(b), on remarque bien que l'activation du module M_5 s'effectue après avoir attendu les fins des modules M_2 et M_4, qui marquent respectivement les fins des programmes $M_1; M_2$ et $M_3; M_4$.

2.3.2 Répétition

Nous enrichissons ensuite, notre sous-langage par un opérateur de répétition, noté $'*'$. Cet opérateur postfixé permet de répéter indéfiniment l'exécution d'un module ou d'une structure de modules. Remarquons qu'une structure bouclée ne se termine jamais, sauf en cas de préemption (cf. section 2.3.3).

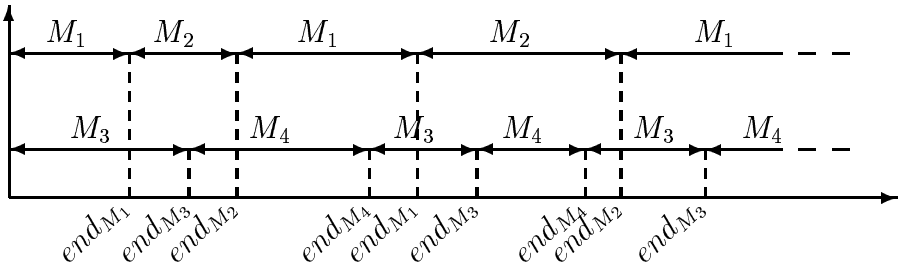


FIG. 2.3 – Répétition, $[\ [\ M_1; M_2\ ]* \mid\mid [\ M_3; M_4\ ]*\ ]$

Exemple 2.2

Nous considérons ici le programme $[\ [\ M_1; M_2\ ]* \mid\mid [\ M_3; M_4\ ]*\ ]$ illustré par le chronogramme de la figure 2.3

2.3.3 Préemption et activation

Jusqu'à présent, nous nous sommes limités aux seuls modules. Nous intégrons, ici, les événements dans le langage au travers des opérateurs de *préemption* et d'*activation*.

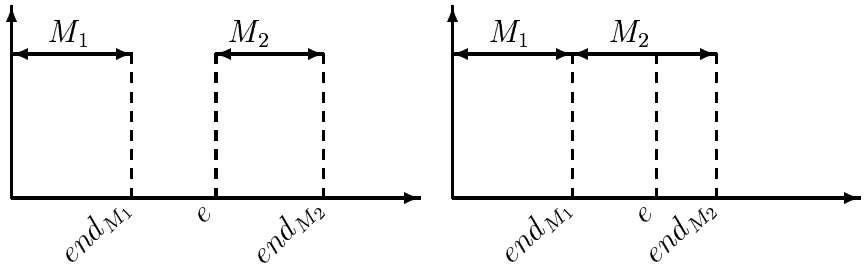
Préemption

Grâce aux opérateurs de préemption, il est maintenant possible d'exprimer dans un programme ELECTRE qu'une occurrence d'un événement peut être prise en compte. On peut, par exemple, exprimer que pendant l'exécution d'un module, si une occurrence d'un certain événement survient, l'exécution de ce module est interrompue. On dit alors que le module est préempté. Il existe deux formes de préemption :

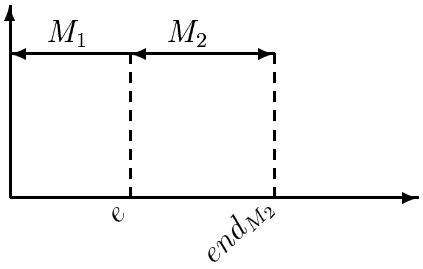
1. la *préemption nécessaire* (ou *préemption forte*), dénotée par l'opérateur $'/'$. Dans un programme ELECTRE, cet opérateur succède à une structure de modules et précède une structure événementielle[1] et permet d'exprimer deux éventualités :
   (a) pendant l'exécution de la structure de modules, si une occurrence de l'événement qui succède à l'opérateur se produit, la structure de modules en cours est interrompue.
   (b) par contre, si la structure de modules se termine sans qu'une occurrence de l'événement qui succède l'opérateur ne se produise, le programme est bloqué en attente d'une telle occurrence (d'où le terme de préemption nécessaire).
2. la *préemption non-nécessaire* (ou *préemption faible*), dénotée par l'opérateur $' \uparrow'$. Cet opérateur s'utilise de la même manière, entre une structure de modules et une structure événementielle. Il permet également d'exprimer deux éventualités :
   (a) pendant l'exécution de la structure de modules, si une occurrence de l'événement qui succède à l'opérateur se produit, le comportement est le même que dans le cas précédent, à savoir que la structure de modules est interrompue.
   (b) si la structure de modules se termine, le programme continue son exécution de manière séquentielle, et c'est là que se différencie la préemption non-nécessaire.

Exemple 2.3
Nous considérons ici les programmes $[\ M_1/e\ ]; M_2$ et $[\ M_1 \uparrow e\ ]; M_2$. Les figures 2.4(a) et 2.4(b) illustrent respectivement des exécutions du premier (préemption nécessaire) et du deuxième (préemption non-nécessaire) programme où aucune occurrence de e survient pendant l'exécution

[1] cette notion est détaillée dans la suite ; pour l'instant, elle n'englobe que les événements.



(a) [M_1/e]; M_2 (b) [$M_1 \uparrow e$]; M_2



(c) [M_1/e]; M_2 et [$M_1 \uparrow e$]; M_2

FIG. 2.4 – Préemption

de M_1. Il est important de remarquer à la figure 2.4(c) que tant qu'aucune occurrence de e ne se produit, le programme ne fait rien. La figure 2.4(c) illustre, quant à elle, une exécution commune aux deux programmes où une occurrence de e vient interrompre l'exécution de M_1.

Activation

Couplé à ces opérateurs de préemtpion, l'opérateur d'*activation* dénoté par $'$:$'$ permet à la suite d'une préemption (nécessaire ou non), d'activer une structure de modules. Cet opérateur vient se positionner après la structure événementielle, à la droite d'un opérateur de préemption. Il précède la structure de modules à activer. Dans un tel cas, lorsqu'une occurrence de l'événement attendu active une structure de modules, cet événement est vivant pendant que celle-ci s'exécute, et ce uniquemement pour les événements à mémorisation unique et multiple.

Exemple 2.4

Nous considérons ici le programme [$M1/e : M_2$]; M_3 illustré par le chronogramme de la figure 2.5.

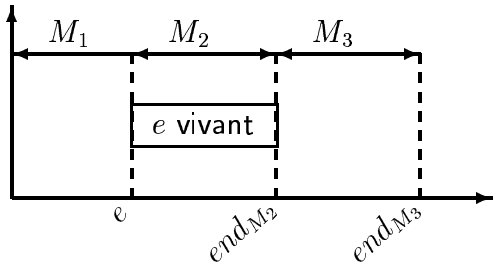


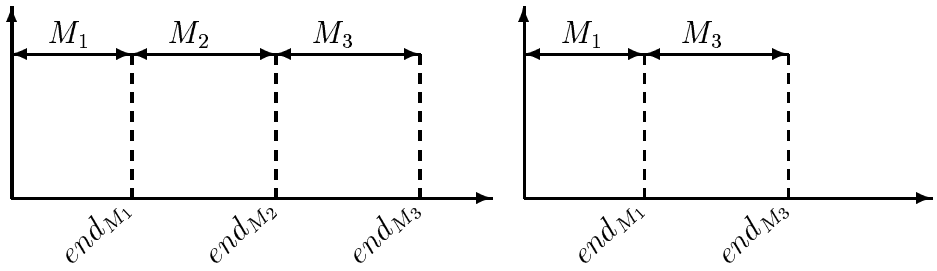
FIG. 2.5 – Activation, [$M_1/e : M_2$]; $M_3$3

2.3.4 Structures d'interruption

Précédemment, nous avions introduit la notion de structure événementielle. Elle se réduisait alors à un simple événement. Le langage ELECTRE permet bien plus. Il est, en effet, possible d'exprimer l'activation en disjonction exclusive ou en parallèle de structures de modules. Ces structures événementielles d'interruption se note entre accolades.

Structure d'interruption simple

Les *structures d'interruption simples* sont composées d'un événement, éventuellement activateur, c'est-à-dire un événement suivi de l'opérateur d'activation, et d'une structure de modules à activer.



(a) [$M_1 \uparrow e : M_2$]; M_3 (b) [$M_1 \uparrow \{ e : M_2 \}$]; M_3

FIG. 2.6 – Structure d'interruption simple

Exemple 2.5

Il existe une différence fondamentale entre l'utilisation de l'opérateur d'activation et l'utilisation de la structure d'interruption simple. Considérons les programmes suivants : [$M_1 \uparrow e : M_2$]; M_3 et [$M_1 \uparrow \{ e : M_2 \}$]; M_3. Dans le premier, comme illustré dans la figure 2.6(a), si aucune occurrence de e ne survient pas pendant l'exécution de M_1, lorsque M_1 se termine, le programme

continue son exécution en activant le module M_2. Tandis que dans le deuxième programme, comme illustré à la figure 2.6(b), la structure d'interruption simple $\{ e : M_2 \}$ garantit que M_2 ne s'exécute que lorsqu'une occurrence de e se produit. De plus, dans ce dernier cas, pendant l'exécution de M_2, l'événement e est vivant, alors que dans le premier cas, e n'est pas vivant.

Structure d'interruption disjonctive

Les *structures d'interruption disjonctives* sont, quant à elles, composées de plusieurs événements, éventuellement activateurs, séparés par l'opérateur $'|'$. Parmi les structures de modules activées, une seule peut s'exécuter à la fois.

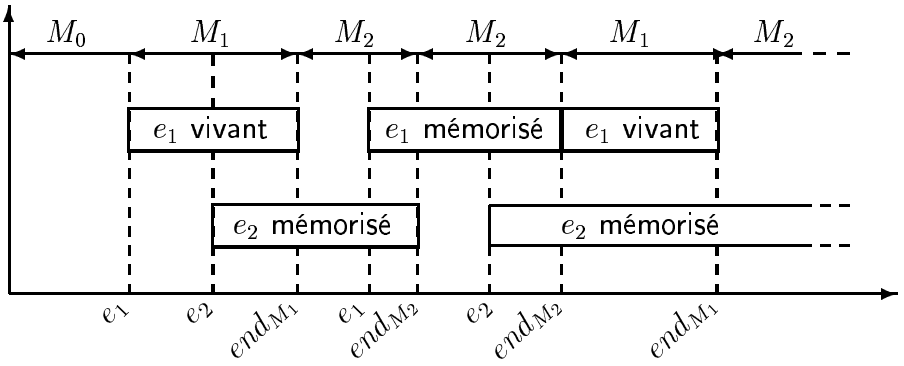


FIG. 2.7 – Structure disjonctive, $[\ M_0/\{\ e_1 : M_1 \mid e_2\ \} : M_2\ ]*$

Exemple 2.6
Nous considérons ici le programme $[\ M_0/\{\ e_1 : M_1 \mid e_2\ \} : M_2\ ]*$ illustré par le chronogramme de la figure 2.7.

Structure d'interruption parallèle

Les *structures d'interruption parallèles* sont également composées de plusieurs événements, éventuellement activateurs. Les structures de modules activées peuvent, à la différence des structures d'interruption disjonctive, s'exécuter en même temps. Il existe deux types de structures d'interruption parallèles :

1. les *structures d'interruption parallèles fortes* se terminent lorsque **toutes les branches** sont terminées et sont construites grâce à l'opérateur $'||'$.
2. les *structures d'interruption parallèles faibles* se terminent lorsque **toutes les branches démarées** sont terminées et sont construites grâce à l'opérateur $'|||'$.

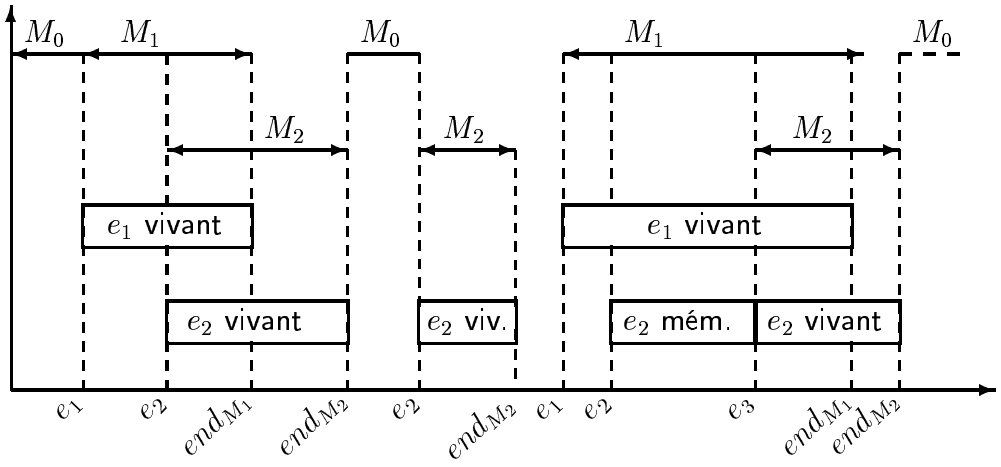


FIG. 2.8 – Structure parallèle forte, [[$M_0/\{\ e_1 : M_1 \ ||\ e_2 : M_2\ \}$] $\uparrow e_3$]$*$

Exemple 2.7

Nous considérons ici le programme [[$M_0/\{\ e_1 : M_1 \ ||\ e_2 : M_2\ \}$] $\uparrow e_3$]$*$ illustré par le chronogramme de la figure 2.8. Cet exemple nous permet de faire remarquer la fonction de la vivacité des événements. Dans l'exemple, lors de l'occurrence de e_3, on peut voir que M_1, activé précédemment par e_1, continue son exécution et que l'occurrence mémorisée de e_2 est alors prise en compte et active M_2. Ceci s'explique par le fait que lors de l'occurrence de e_3, la structure bouclée du programme en relance l'exécution au début. A ce point, l'événement e_1 est toujours vivant et donc l'exécution de M_1 doit être continuée. De plus, puisque la précédente exécution de M_2 est complètement terminée, l'occurrence mémorisée de e_2 peut être prise en compte et déclenche donc l'activation de M_2.

Exemple 2.8

Nous considérons ici le programme [$M_0/\{\ e_1 : M_1 \ |||\ e_2 : M_2\ \}$]$*$ illustré par le chronogramme de la figure 2.9.

Remarquons que, bien entendu, ces différentes structures d'interruption peuvent s'imbriquer. Il est donc possible de construire des structures événementielles utilisant à la fois du parallélisme et de la disjonction.

Exemple 2.9

Nous considérons ici un exemple de telle structure imbriquée dans le programme suivant : [$M_0/\{\ e_1 : M_1 \mid \{\ e_2 : M_2 \ ||\ e_3 : M_3\ \}\ \} : M_4$]$*$.

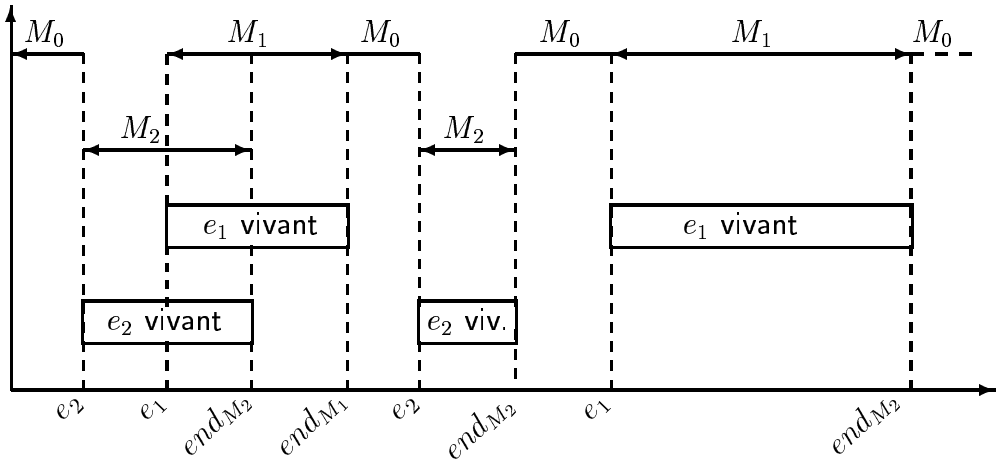


FIG. 2.9 – Structure parallèle faible, [M_0/{ $e_1 : M_1 \;|||\; e_2 : M_2$ }]*

2.3.5 Propriété des modules et des événements

Nous avons jusqu'ici décrit les opérateurs permettant de construire un programme ELECTRE autour des deux types d'entité principale, à savoir les modules et les événements. Dans cette section, nous introduisons des opérateurs unaires préfixés qui permettent de préciser certaines propriétés sur ces entités.

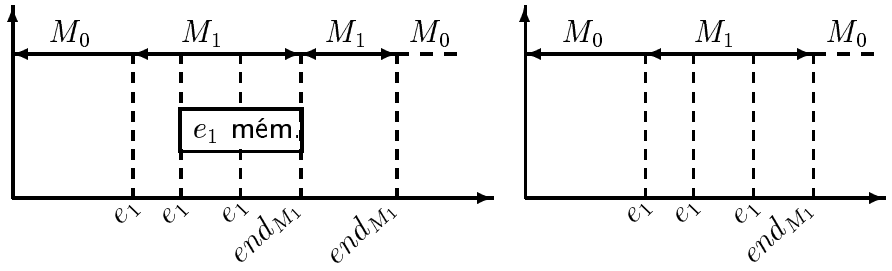
Type d'événements

Comme nous l'avons expliqué dans la section 2.2, il existe, en ELECTRE, différents types d'événements. Par défaut un événement est à mémorisation unique. Pour les autres types d'événements, l'utilisation des opérateurs suivants est nécessaire.

1. '$' pour marquer un événement à consommation immédiate.
2. '#' pour marquer un événement à mémorisation multiple .
3. '@' pour marquer un événement fugace.

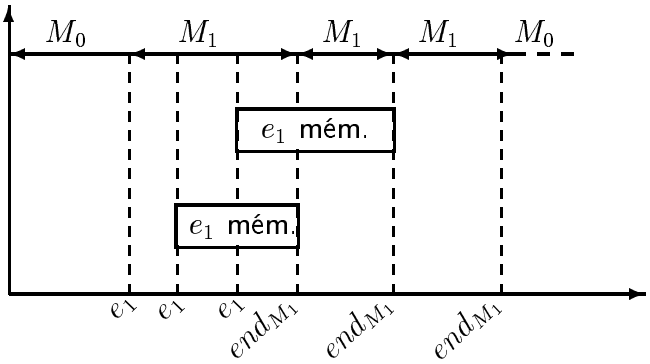
Exemple 2.10

Nous considérons des programmes ELECTRE reprenant les différents types d'événements dont la mémorisation est illustréé par les chronogrammes de la figure 2.10. Nous avons délibérément omis la vivacité des événements pour plus de clarté. A la figure 2.10(a), l'événement e_1 est soit à mémorisation multiple, soit à consommation immédiate. Ses occurrences ne seront donc mémorisées au plus qu'une fois. Ensuite, à la figure 2.10(b), l'événement e_1 est de type fugace, ce qui interdit la mémorisation de ses occurrences. Dans ce cas toute occurrence inattendue



(a) [M_0/{ $e_1 : M_1$ }]∗ et [M_0/{ $\$e_1 : M_1$ }]∗

(b) [M_0/{ $@e_1 : M_1$ }]∗



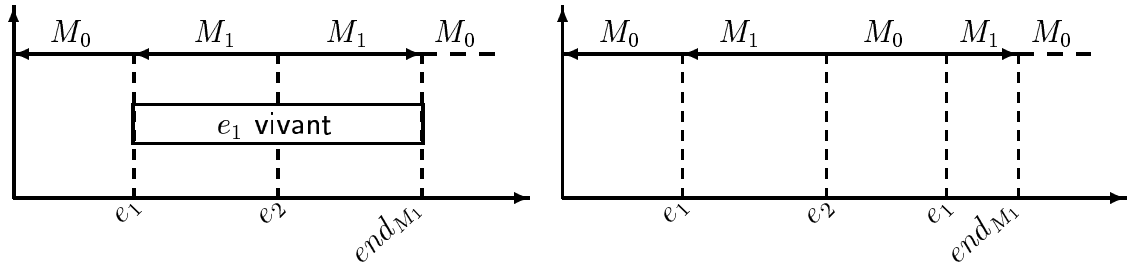
(c) [M_0/{ $\#e_1 : M_1$ }]∗

FIG. 2.10 – Types d'événements - mémorisation

sera perdue. Pour finir, à la figure 2.10(c), l'événement e_1 est à mémorisation multiple, et donc chaque occurrence inattendue sera mémorisée pour être traitée ultérieurement.

Exemple 2.11

Nous considérons des programmes ELECTRE reprenant les différents types d'événements dont la vivacité est illustrée par les chronogrammes de la figure 2.11. A la figure 2.11(a), l'événement e_1 est soit à mémorisation unique, soit à mémorisation multiple. Dès lors, comme e_1 est activateur du module M_1, tant que l'exécution de ce dernier n'est pas terminée, e_1 est vivant. Lorsque une occurrence de e_2 vient interrompre la structure de modules, la structure bouclée en reprend l'exécution, et comme e_1 est vivant, réactive le module M_1. Par contre, comme illustré à la figure 2.11(b), si e_1 est soit fugace, soit à consommation immédiate, il ne sera jamais vivant, et lorsque la structure bouclée reprend l'exécution de la structure de modules, c'est M_0 qui est activé.



(a) $[\,[\,M_0/\{\,e_1 : M_1\,\}\,]\uparrow e_2\,]*$ et $[\,[\,M_0/\{\,\#e_1 : M_1\,\}\,]\uparrow e_2\,]*$

(b) $[\,[\,M_0/\{\,@e_1 : M_1\,\}\,]\uparrow e_2\,]*$ et $[\,[\,M_0/\{\,\$e_1 : M_1\,\}\,]\uparrow e_2\,]*$

FIG. 2.11 – Types d'événements - vivacité

Modules non-préemptibles

Nous avons vu précédemment que les modules pouvaient voir leur exécution interrompue, en cas de préemption. Il est toutefois possible d'empêcher qu'un module soit préempté, en utilisant l'opérateur $'!'$. La préemption est alors reportée après la fin de l'exécution de ce module.

Modules à reprise au début

Par défaut, lorsqu'il est fait référence à un module dans un programme ELECTRE, cela dénote l'activation de ce module. Si auparavant, ce même module avait déjà commencé son exécution, pour ensuite être préempté, ce module reprendra son exécution là où elle avait été interrompue. Malgré tout, il est possible de forcer le module à recommencer son exécution depuis le début, grâce à l'opérateur $'>'$.

2.4 Un exemple : les lecteurs/écrivains

Maintenant que nous avons passé en revue les différentes constructions du langage ELECTRE, nous allons essayer de mettre ces notions en pratique au travers d'un exemple concret : le problème des lecteurs/écrivains. Dans ce problème, une ressource unique est accessible en lecture comme en écriture par un ensemble de processus, ici, deux processus lecteurs, représentés par les modules R_1, R_2, et deux processus écrivains, représentés par les modules W_1 et W_2. Tous les lecteurs ont la possibilité d'accéder en même temps à la ressource, alors qu'un seul écrivain ne peut y accéder à un moment donné, ceci parce qu'il peut modifier cette ressource. Chaque lecteur (écrivain) doit effectuer une requête avant de pouvoir accéder en lecture (en écriture) à la ressource. Ces requêtes sont modélisées

par les événements r_1, r_2 pour les lecteurs et les événements w_1, w_2 pour les écrivains. Ces événements seront à mémorisation multiple afin que chaque requête soit traitée en son temps. Une première solution au problème est donnée par le programme suivant :

$$[\ 1/\{\ \{\ \#r_1 : R_1 \ |||\ \#r_2 : R_2\ \} \mid \#w_1 : W_1 \mid \#w_2 : W_2\ \}\ ]*$$

L'utilisation de l'opérateur $'\ |'$ assure l'exclusion mutuelle d'un accès en écriture par rapport à l'autre et par rapport aux accès en lecture. Ces derniers peuvent s'effectuer en même temps grâce à l'utilisation de l'opérateur $'\ |||'$. Le module 1 est utilisé pour modéliser la tâche de fond, pendant laquelle le programme attend qu'une requête soit formulée. L'opérateur $'*'$ exprime le comportement répétitif des accès à la ressource.

Dans cette solution, les lecteurs et écrivains sont sur le même pied d'égalité. Une autre solution consiste à donner la priorité aux écrivains de manière à ce que toute demande d'écriture soit traitée dès que la ressource devient disponible. Cette solution est exprimée en ELECTRE par le programme suivant :

$$[\ [\ 1/\{\ \#r_1 : !R_1 \ |||\ \#r_2 : !R_2\ \}\ ] \uparrow \{\ \#w_1 : W_1 \mid \#w_2 : W_2\ \}\ ]*$$

Les modules représentant les lecteurs (R_1, R_2) sont ici qualifiés de manière à être non-préemptibles. Ceci est nécessaire pour marquer le fait qu'avant qu'un écrivain n'accède à la ressource, toutes les lectures en cours doivent être terminées. En effet, l'utilisation de l'opérateur $'!'$ reporte la préemption à la fin des modules lecteurs qui sont actifs au moment d'une requête d'écriture (occurrence de w_1 ou w_2).

Chapitre 3

Les Fiffo-Automates

Dans le chapitre 2, nous avons décrit les différentes constructions du langage Electre. Dans le présent chapitre, nous allons introduire un modèle d'automate à file qui sert à modéliser les comportements de tels programmes. Nous commençons, à la section 3.1, en décrivant la méthode permettant de compiler un programme Electre en automate à file, ce qui en définit la sémantique. Cette étape de compilation est par ailleurs détaillée dans [?]. Nous poursuivons, à la section 3.2, en donnant une définition formelle aux automates à file utilisés, appellés *automates à file réactifs*. Remarquons que ces automates à file réactifs sont étudié dans [?]. Cependant, ce modèle d'automate n'est pas suffisamment complet en vue de la compilation (et de la répartition) pour les Lego Mindstorms. En effet, ils ne reprennent pas les actions sur les modules (i.e. activation, préemption, reprise au début). Pour cette raison, nous présentons, dans la section 3.3, une extension de ces automates à file réactifs : les Fiffo-Automates que nous allons utiliser comme structure intermédiaire.

3.1 Sémantique et compilation de programme Electre

La sémantique opérationnelle du langage Electre, présenté en annexe B, consiste en un ensemble de règles de réécriture. Ces règles formalisent les notions décrites dans le chapitre 2. Elles permettent de décrire comment doit se poursuivre l'exécution d'un programme vis-à-vis d'une occurrence d'événements, et ce en déterminant le nouveau programme restant à exécuter. Grâce à ces règles de réécriture, nous allons montrer comment compiler un programme Electre en automate à file. Afin d'illustrer nos propos, reprenons le programme des lecteurs/écrivains, sans priorité, de la section 2.4 :

$$[\ 1/\{\ \{\ \#r_1 : R_1 \ |||\ \#r_2 : R_2\ \} \mid \#w_1 : W_1 \mid \#w_2 : W_2\ \}\ ]*$$

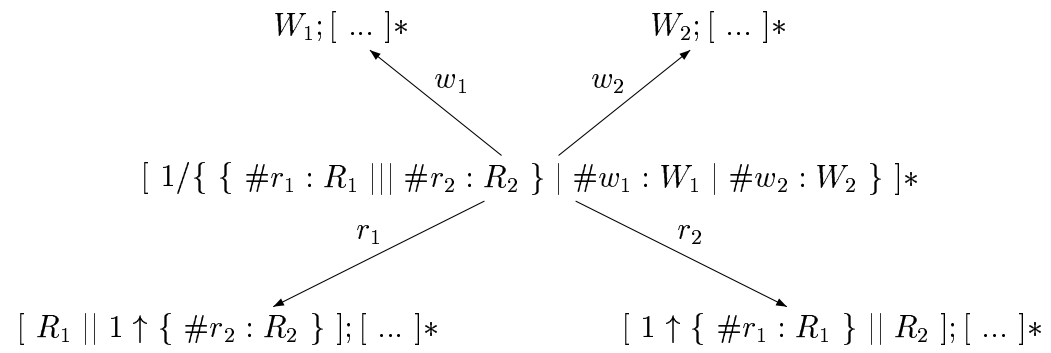


FIG. 3.1 – Lecteurs/Ecrivains - Première réécriture

Ce programme contient 4 événements : r_1, r_2, w_1 et w_2, tous à mémorisation multiple. L'exécution du programme est dirigée par les occurrences de ces événements. Ce sont les règles de réécriture qui décrivent comment le programme réagit face à ces occurrences, en fournissant de nouveaux programmes restant à exécuter. Chaque programme ELECTRE ainsi obtenu sera modélisé par un état. La réécriture d'un programme en un autre est, quant à elle, modélisée par une transition étiquetée de l'événement qui la déclenche reliant ces deux programmes. Dans notre exemple, le programme des lecteurs/écrivains peut réagir à ces quatre événements, ce qui conduit, par des réécritures, à quatre programmes ELECTRE, présentés à la figure 3.1.

Rappelons que lorsqu'un module termine son exécution, il émet une occurrence d'un événement marquant sa fin. Ainsi, par exemple, dans le programme $W_1;[\ ...\ ]*$, le module W_1 étant actif, émettra, une fois terminé, une occurrence de l'événement end_{W_1}. Il est important de prendre ces événements en compte dans les réécritures successives de programme. Remarquons, au passage, que ces événements de fin de module sont de type fugace, c'est-à-dire qu'ils ne peuvent être mémorisés.

En tenant compte de tous les événements, par réécritures successives, nous construisons un graphe, appelé *graphe de contrôle*, qui reprend l'évolution du programme initial. La figure 3.2 présente le graphe de contrôle du programme des lecteurs/ecrivains.

Théorème 3.1

Le graphe de contrôle obtenu par compilation d'un programme ELECTRE est fini (Cassez & Roux [?]). ■

Ce graphe de contrôle reste néanmoins incomplet. En effet, il ne spécifie pas le comportement à suivre lorsqu'une occurrence ne peut être prise en compte directement. Plus

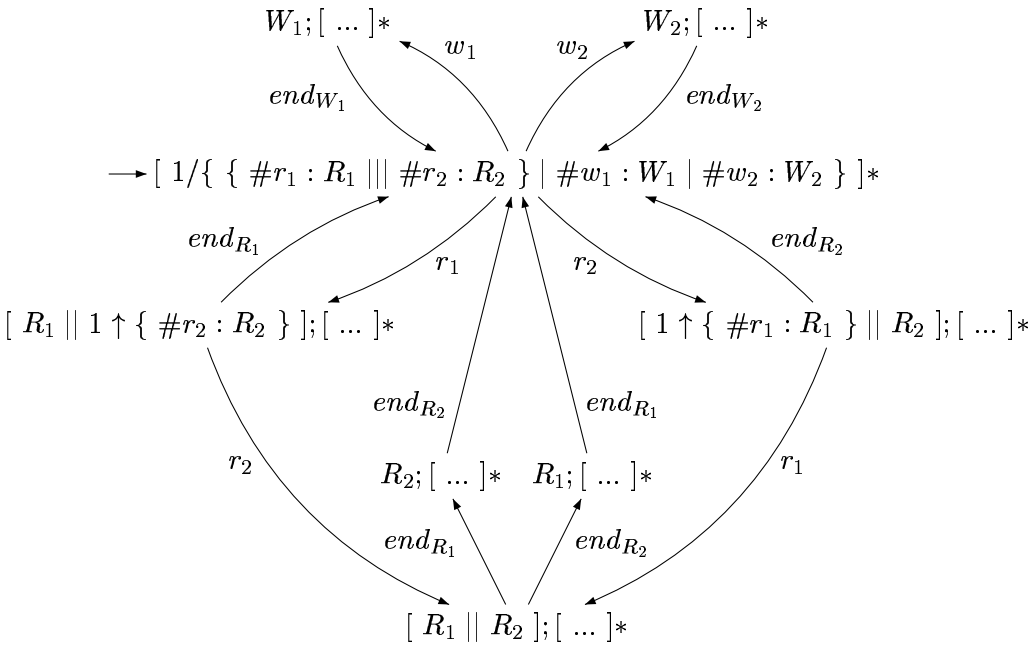


FIG. 3.2 – Lecteurs/Ecrivains - Graphe de contrôle

particulièrement, il ne tient pas compte de la mémorisation des événements. Afin d'obtenir une modélisation correcte du programme ELECTRE de départ, il est donc impératif de compléter ce graphe de contrôle de deux manières, comme illustré sur un morceau du graphe, à la figure 3.3.

1. **Mémorisation d'occurrences** Lorsque, dans le graphe de contrôle, rien n'est spécifié quant à la prise en compte d'occurrences d'un événement mémorisable e, dans un état, nous rajoutons une transition de mémorisation étiquetée $!e$. Cette transition a pour origine et pour extrémité l'état en question.
2. **Prise en compte différée** Lorsque le système prend en compte une occurrence d'événement, aucune différence ne doit être faite sur la provenance de cette occurrence. Dès lors, toute occurrence d'événement mémorisable pouvant être prise en compte directement, peut également l'être de manière différée (lorsque l'occurrence provient de la file d'attente), ce qui est modélisé, sur le graphe de contrôle en doublant chaque transition de prise en compte directe, étiquetée e, d'une transition de prise en compte différée, étiquetée $?e$.

Après enrichissement du graphe de contrôle, nous obtenons un automate appellé *automate à file réactifs.*

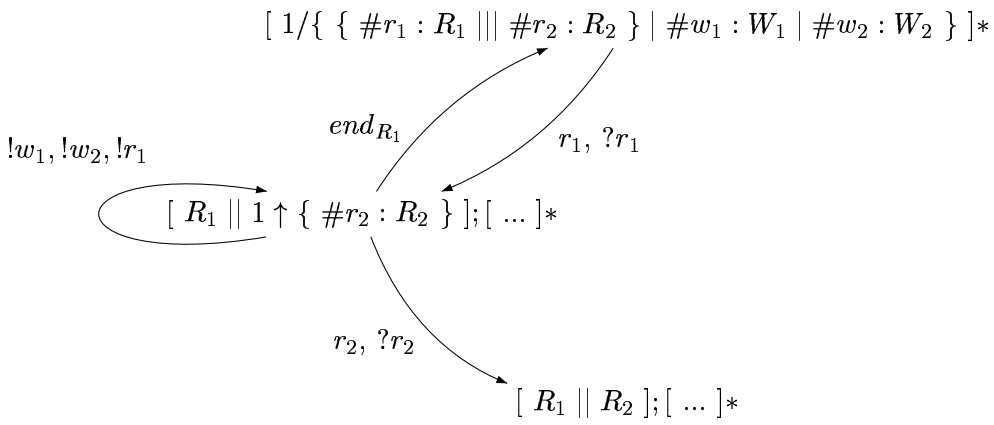


FIG. 3.3 – Lecteurs/Ecrivains - Enrichissement du graphe de contrôle

3.2 Les automates à file réactifs (AFR)

Les *automates à file réactifs*, notés AFR dans la suite, ont donc été introduits pour modéliser des comportements de programme ELECTRE. Dans la section suivante, nous donnons une définition formelle au AFR. Par la suite, dans la section 3.2, nous définissons la sémantique de ces AFR.

3.2.1 Définition

Avant de donner une définition aux AFR, rappelons que E dénote l'ensemble des identifiants d'événements. De par les différentes politiques de mémorisation, E peut être partitionné en deux :

1. E_F : l'ensemble des identifiants d'événements fugaces[1].
2. E_M : l'ensemble des identifiants d'événements mémorisables. Cet ensemble est lui même scindé en deux parties : E_{M_1} et E_{M_*} qui dénotent, respectivement, les ensembles d'identifiants à mémorisation unique et à mémorisation multiple.

De par leur définition, $(E_{M_1} \cap E_{M_*}) = \phi$, $(E_{M_1} \cup E_{M_*}) = E_M$, $(E_M \cap E_F) = \phi$ et $(E_M \cup E_F) = E$.

[1] événements qui ne peuvent être mémorisés, cf. section 2.2

Définition 3.1 - Automate à file réactif

Un *automate à file réactif* [?] est un quadruplet $\mathcal{A} = (Q_\mathcal{A}, \Sigma_\mathcal{A}, \Delta_\mathcal{A}, q^0_\mathcal{A})$ où :

1. $Q_\mathcal{A}$ est l'*ensemble d'états* (ensemble fini)
2. $\Sigma_\mathcal{A} = E \cup (\{!, ?\} \times E_M)$ est l'*alphabet*, c'est-à-dire l'ensemble des *actions* de $\mathcal{A}$: e marque la prise en compte de l'événement e de l'environnement, $!e$ la mémorisation de l'événement e dans la file et $?e$, la prise en compte de l'événement e de cette même file d'attente
3. $\Delta_\mathcal{A} \subseteq Q_\mathcal{A} \times \Sigma_\mathcal{A} \times Q_\mathcal{A}$ est la *relation de transition*
4. $q^0_\mathcal{A} \in Q_\mathcal{A}$ est l'*état initial* de $\mathcal{A}$

Théorème 3.2

Un AFR résultant de la compilation d'un programme ELECTRE est déterministe.

Preuve

Les règles de réécriture sur lesquelles se base la construction du graphe de contrôle en grantissent le déterminisme. En effet, pour chaque règle, un et un seul programme (et par conséquent, un et un seul état) est obtenu après réécriture vis-à-vis d'une occurrence d'événement. Ensuite lors de l'enrichissement de ce graphe, les transitions de mémorisation et de prise en comtpe différée ajoutée n'introduisent pas de non-déterminisme. ■

Remarquons que, par construction, pour un AFR provenant de la compilation d'un programme ELECTRE, dans un état donné, une occurrence d'événement est soit prise en compte[2], soit mémorisée. Nous formalisons cela ci-après.

Propriété 3.1

Soit un AFR $\mathcal{A} = (Q_\mathcal{A}, \Sigma_\mathcal{A}, \Delta_\mathcal{A}, q^0_\mathcal{A})$ résultant de la compilation d'un programme ELECTRE, nous avons $\forall q \in Q_\mathcal{A}, e \in E_M$,

1. $(q, !e, q') \in \Delta_\mathcal{A} \rightarrow (q, e, q') \notin \Delta_\mathcal{A}$
2. $(q, ?e, q') \in \Delta_\mathcal{A} \rightarrow (q, e, q') \in \Delta_\mathcal{A}$

[2]la prise en compte directe et différée pour les événements de E_M, et elle est uniquement directe pour les événements de E_F

Pour un AFR $\mathcal{A}$, nous posons $in_{\mathcal{A}}(q) = \{x \in \Sigma_{\mathcal{A}} | \exists\ q' \in Q_{\mathcal{A}}, (q', x, q) \in \Delta_{\mathcal{A}}\}$ l'*ensemble des étiquettes entrantes* en q. De même, nous posons $out_{\mathcal{A}}(q) = \{x \in \Sigma_{\mathcal{A}} | \exists\ q' \in Q_{\mathcal{A}}, (q, x, q') \in \Delta_{\mathcal{A}}\}$ l'*ensemble des étiquettes sortantes* de q. Nous pouvons, très facilement, étendre ces définitions aux ensembles d'états : $in_{\mathcal{A}}(Q) = \cup_{q \in Q} in_{\mathcal{A}}(q)$, $out_{\mathcal{A}}(Q) = \cup_{q \in Q} out_{\mathcal{A}}(q)$. Nous posons également $Succ_{\mathcal{A}}(q) = \{q' \in Q_{\mathcal{A}} | \exists\ x \in \Sigma_{\mathcal{A}}, (q, x, q') \in \Delta_{\mathcal{A}}\}$ l'*ensemble des états successeurs de* q. Et de même, $Pred_{\mathcal{A}}(q) = \{q' \in Q_{\mathcal{A}} | \exists\, x \in \Sigma_{\mathcal{A}}, (q', x, q) \in \Delta_{\mathcal{A}}\}$, l'*ensemble des états prédécesseurs de* q. Pour terminer, nous posons $E_M^q = \{e \in E_M \mid (q, !e, q) \in \Delta_{\mathcal{A}}\}$ l'*ensemble des événements mémorisés en l'état* q.

Un *chemin* sur $\mathcal{A}$ est une séquence finie ou infinie de transition (q_0, x_0, q_1), (q_1, x_1, q_2), (q_2, x_2, q_3),... tel que $q_0 = q_{\mathcal{A}}^0$. Un chemin est dit *maximal* s'il est infini ou s'il se termine en q_n tel que $Succ(q_n) = \phi$. Nous notons $[\![\mathcal{A}]\!]$ l'ensemble des chemins sur $\mathcal{A}$.
Une *trace* de $\mathcal{A}$ est un mot fini ou infini m sur $\Sigma_{\mathcal{A}}$ tel qu'il existe un chemin (q_0, x_0, q_1), (q_1, x_1, q_2), (q_2, x_2, q_3),... sur $\mathcal{A}$ avec $m = x_0 x_1 x_2 ...$ L'ensemble des traces de $\mathcal{A}$ définit un langage noté $\mathcal{L}(\mathcal{A})$.

Exemple 3.1

Nous reprenons ici le programme ELECTRE des lecteurs/écrivains sans priorité : $[\ 1/\{\ \{\ \#r_1 : R_1\ |||\ \#r_2 : R_2\ \}\ |\ \#w_1 : W_1\ |\ \#w_2 : W_2\ \}\ ]*$. L'AFR résultant de la compilation de ce programme est illustré à la figure 3.4.

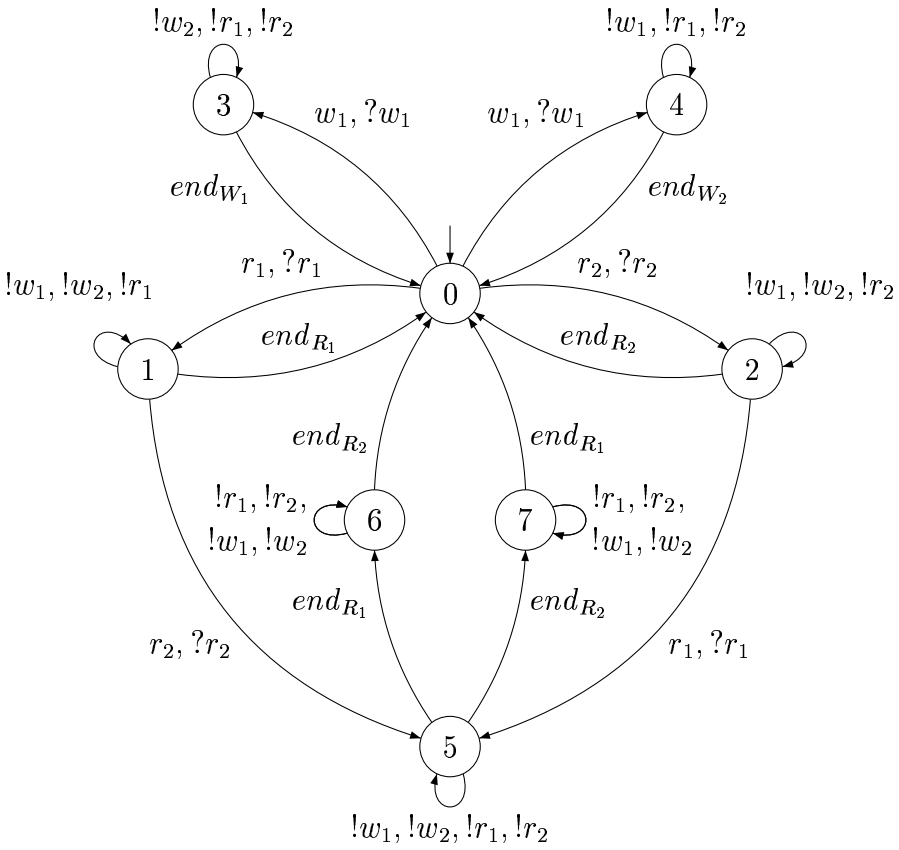


FIG. 3.4 – lecteurs/écrivains sans priorité : AFR

3.2.2 Sémantique

La sémantique d'un AFR $\mathcal{A}$ est donnée par un système de transition $\mathcal{S}$ (possiblement infini) appelé *système à file réactif* (noté SFR dans la suite), obtenu à partir de $\mathcal{A}$ en considérant la file de mémorisation et son évolution par rapport aux occurrences d'événements. En effet, tout comme pour ELECTRE, les événements qui ne peuvent être pris en compte directement sont mémorisés dans une file d'attente pour être traités dès que cela devient possible. Une configuration de $\mathcal{S}$ est donc formée d'un couple (q, w) où q est l'état de $\mathcal{A}$ de la configuration et w un mot de E_M représentant l'état de la file d'attente dans cette configuration. Notons que w est toujours fini.

Définition 3.2 - Système à file réactif (SFR)
Un *système à file réactif* [?] est un quadruplet $\mathcal{S} = (Q_\mathcal{S}, \Sigma_\mathcal{S}, \Delta_\mathcal{S}, q^0_\mathcal{S})$ défini à partir d'un AFR $\mathcal{A} = (Q_\mathcal{A}, \Sigma_\mathcal{A}, \Delta_\mathcal{A}, q^0_\mathcal{A})$ par :

1. $Q_\mathcal{S} = Q_\mathcal{A} \times (E_M)^*$ est l'*ensemble des configurations* de $\mathcal{S}$.
2. $\Sigma_\mathcal{S} = \Sigma_\mathcal{A}$ est l'*alphabet*, c'est-à-dire l'*ensemble des actions* de $\mathcal{S}$.
3. $\Delta_\mathcal{S} \subseteq Q_\mathcal{S} \times \Sigma_\mathcal{S} \times Q_\mathcal{S}$ est la plus petite relation de transition définie par :
   (a) $((q, w), !e, (q, w')) \in \Delta_\mathcal{S}$ si $(q, !e, q) \in \Delta_\mathcal{A}$ et $w \in (E^q_M)^*$, avec $w' = w$ si $(|w|_e \geq 1) \wedge (e \in E_{M^1})$, et $w' = w.e$ sinon ([3])
   (b) $((q, w), e, (q', w)) \in \Delta_\mathcal{S}$ si $(q, e, q') \in \Delta_\mathcal{A}$ et $w \in (E^q_M)^*$
   (c) $((q, w), ?e, (q', w')) \in \Delta_\mathcal{S}$ si $(q, ?e, q') \in \Delta_\mathcal{A}$ et $\exists\ w_1 \in (E^q_M)^*$ et $w_2 \in (E_M)^*$ tels que $w = w_1 e w_2$ et $w' = w_1 w_2$
   (d) $((q, w), e, (q, w)) \in \Delta_\mathcal{S}$ si $e \notin (out(q) \cup E_M)$ et $w \in (E^q_M)^*$ ([4])
4. $q^0_\mathcal{S} = (q^0_\mathcal{A}, \varepsilon) \in Q_\mathcal{S}$ est la *configuration initiale* de $\mathcal{S}$.

Les caractéristiques de la politique de gestion FIFFO (***F**irst **I**nput **F**irst **F**ireable **O**utput*) sont exprimées au travers des points (4.a) à (4.d) de la définition 3.2. Ces caractéristiques sont les suivantes :

1. **priorité au traitement d'occurrences mémorisées** Dans un état, si une occurrence de la file d'attente peut être traitée, doit l'être. Ceci est vérifié par la condition $w \in (E^q_M)^*$ des points (4.a), (4.b) et 4(d) qui ne permet ni de mémoriser, ni de traiter

[3] $|w|_e$ dénote le nombre d'occurrences de e dans le mot w
[4] exprime le fait que si un événement e se produit alors qu'il n'est pas à prendre en compte, $\mathcal{S}$ reste dans la même configuration

de nouvelles occurrences d'événements. En effet, si cette condition n'est pas vérifiée ($w \notin (E_M^q)^*$) cela signifie que la file d'attente w contient une occurrence d'événement qui peut être prise en compte en q. Des transitions du type (4.a), (4.b) ou (4.d) ne sont pas permises. Dès lors, seules des transitions de prise en compte diférée du type (4.c) sont permises.

2. **traitement des occurrences "dès que possible"** Lors d'une prise en compte différée, ce n'est pas forcément l'occurrence en tête de la file d'attente qui doit être prise en compte, mais bien la première occurrence traitable. La condition $w_1 \in (E_M)^*$ du point (4.c) qui autorise à prendre en compte en q une autre occurrence d'événement que celle présente en tête de la file w.
3. **priorité à l'occurrence mémorisée la plus ancienne** Lors d'une prise en compte différée, parmi toutes les occurrences traitables de la file d'attente w, c'est la première qui doit être traitée. Ceci est vérifié par la condition $w_1 \in (E_M^q)^*$ du point (4.c). Elle impose que chaque occurrence précédant e dans w ait été memorisés en q. Dès lors, e est bien le premier événement de la file w que l'on puisse traiter en q.

Un *chemin* sur $\mathcal{S}$ est une séquence finie ou infinie de transitions $((q_0, w_0), x_0, (q_1, w_1))$, $((q_1, w_1), x_1, (q_2, w_2))$, $((q_2, w_2), x_2, (q_3, w_3))$,... tel que $((q_i, w_i), x_i, (q_{i+1}, w_{i+1})) \in \Delta_{\mathcal{S}}$ et $(q_0, w_0) = q_{\mathcal{S}}^0$. Un chemin est dit *maximal* s'il est infini ou s'il se termine en (q_n, w_n) tel que $Succ(q_n) = \phi$. Nous notons $[\![\mathcal{S}]\!]$ l'ensemble des chemins sur $\mathcal{S}$.
Une *trace* de $\mathcal{S}$ est un mot fini ou infini m sur $\Sigma_{\mathcal{S}}$ tel qu'il existe un chemin $((q_0, w_0), x_0, (q_1, w_1))$, $((q_1, w_1), x_1, (q_2, w_2))$, $((q_2, w_2), x_2, (q_3, w_3))$,... sur $\mathcal{S}$ avec $m = x_0 x_1 x_2 ...$ L'ensemble des traces de $\mathcal{S}$ définit un langage noté $\mathcal{L}(\mathcal{S})$.

3.3 Les FIFFO-AUTOMATES (FA)

Les AFR se concentrent sur la modélisation du comportement d'un système vis-à-vis des occurrences d'événements. Ces automates ne contiennent aucune information quant aux actions à effectuer sur les modules (i.e. activation, préemption, reprise au début). C'est une des raisons pour lesquelles nous allons nous intéresser à une extension de ce modèle appelé les FIFFO-AUTOMATES (noté FA dans la suite). Ces derniers se différencient des AFR par l'ajout de trois fonctions d'étiquetage : ACT, PRE et REP qui vont préciser pour chaque transition de l'automate les modules à activer (ACT), préempter (PRE) et reprendre au début (REP) lorsque cette transition est tirée. Une autre information importante est l'ensemble des modules actifs à l'état initial.

Les AFR ont été introduits pour modéliser des comportements de systèmes réactifs spécifiés en ELECTRE. Ces systèmes sont sensés ne jamais se terminer, alors que certains programmes ELECTRE le peuvent. Un exemple de ce genre de comportement est donné par le programme [$M_1; M_2$], dont l'AFR est présenté à la figure 3.5. C'est une raison supplémentaire qui nous a poussé à ajouter, dans le modèle des FA, un ensemble d'états finals. En effet, nous nous intéressons ici aux programmes ELECTRE dans leur plus grande généralité, et non aux programmes ELECTRE modélisant des systèmes réactifs.

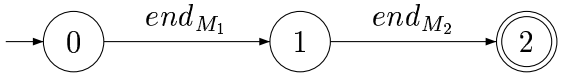


FIG. 3.5 – AFR du programme $M_1; M_2 \xrightarrow{end_{M_1}} M_2 \xrightarrow{end_{M_2}} nil$

3.3.1 Définition

La définition des FIFFO-AUTOMATES est basée sur la définition des AFR, en y ajoutant un ensemble de modules actifs à l'état initial, un ensemble d'états finals, et des fonctions d'activation, de préemption et de reprise au début. Pour rappel, M dénote l'ensemble des identifiants de modules.

Définition 3.3 - Fiffo-Automate (FA)
Un FIFFO-AUTOMATE est un tuple $\mathcal{F} = (Q_\mathcal{F}, \Sigma_\mathcal{F}, \Delta_\mathcal{F}, q^0_\mathcal{F}, A^0_\mathcal{F}, F_\mathcal{F}, \text{ACT}_\mathcal{F}, \text{PRE}_\mathcal{F}, \text{REP}_\mathcal{F})$ où $Q_\mathcal{F}, \Sigma_\mathcal{F}, \Delta_\mathcal{F}$, $q^0_\mathcal{F}$ sont définis de la même manière que pour un AFR (définition 3.1), et où :

1. $A^0_\mathcal{F} \subseteq M$ est l'*ensemble des modules actifs initialement*.
2. $F_\mathcal{F} \subseteq Q_\mathcal{F}$ est l'*ensemble des états finals*.
3. $\text{ACT}_\mathcal{F} : \Delta_\mathcal{F} \to 2^M$ qui indiquent pour chaque transition *les modules à activer* lorsque cette transition est tirée.
4. $\text{PRE}_\mathcal{F} : \Delta_\mathcal{F} \to 2^M$ qui indiquent pour chaque transition *les modules à préempter* lorsque cette transition est tirée.
5. $\text{REP}_\mathcal{F} : \Delta_\mathcal{F} \to 2^M$ qui indiquent pour chaque transition *les modules à réprendre au début* lorsque cette transition est tirée.

Dans un FA $\mathcal{F}$, il est possible de déterminer l'ensemble des modules actifs dans un état. Ceci est possible grâce à $A^0_\mathcal{F}$, qui donne l'ensemble des modules actifs à l'état initial

$q^0_{\mathcal{F}}$, ainsi qu'aux fonctions ACT, PRE et REP. L'unicité de l'ensemble de modules actifs dans un état $q \in Q_{\mathcal{F}}$ est garantie par le fait que les FIFFO-AUTOMATES proviennent de la compilation de programme ELECTRE. En effet, comme nous le verrons à la section 3.1, à chaque état $q \in Q_{\mathcal{F}}$ correspond un programme ELECTRE. Pour chaque programme on peut calculer de manière univoque un ensemble de modules actifs correspondants. Cet ensemble de modules est calculé par un attribut synthétisé sur la grammaire d'ELECTRE. Le calcul de cet attribut est détaillé dans [**?**].

Définition 3.4 - Modules actifs dans un état ($A_{\mathcal{F}}$)
Soit un FA $\mathcal{F}$. L'ensemble des modules actifs dans un état est donné par la fonction $A_{\mathcal{F}}$: $Q_{\mathcal{F}} \rightarrow 2^M$ définie par :

1. $A_{\mathcal{F}}(q^0_{\mathcal{F}}) = A^0_{\mathcal{F}}$
2. $A_{\mathcal{F}}(q') = (A_{\mathcal{F}}(q) \setminus \text{PRE}(q, x, q')) \cup \text{ACT}(q, x, q') \cup \text{REP}(q, x, q')$ pour toute transition $(q, x, q') \in \Delta_{\mathcal{F}}$ et pour tout $q' \neq q^0_{\mathcal{F}}$

Pour conclure, remarquons qu'une transition de mémorisation du type $(q, !e, q)$, avec $e \in E_M$, n'active, ne préempte ni ne reprend au début aucun module. Ceci paraît naturel car l'événement n'est pas alors traité, mais bien mémorisé pour être traité ultérieurement. Ceci impose $\text{ACT}_{\mathcal{F}}(q, !e, q) = \phi$, $\text{PRE}_{\mathcal{F}}(q, !e, q) = \phi$ et $\text{REP}_{\mathcal{F}}(q, !e, q) = \phi$ pour toute transition $(q, !e, q) \in \Delta_{\mathcal{F}}$. Nous avons bien $A_{\mathcal{F}}(q) = (A_{\mathcal{F}}(q)/\text{PRE}_{\mathcal{F}}(q, !e, q)) \cup \text{ACT}(q, x, q') \cup \text{REP}(q, x, q')$.

Un *chemin* sur $\mathcal{F}$ est défini de la même manière que pour un AFR. L'ensemble de chemins sur $\mathcal{F}$ est noté $[\![\mathcal{F}]\!]$. Une *trace* de $\mathcal{F}$ est un mot fini ou infini m sur $(\Sigma_{\mathcal{F}} \times 2^M \times 2^M \times 2^M)$ tel qu'il existe un chemin (q_0, x_0, q_1), (q_1, x_1, q_2), (q_2, x_2, q_3),... sur $\mathcal{F}$ avec $m = (x_0, A_0, P_0, R_0)\ (x_1, A_1, P_1, R_1)\ (x_2, A_2, P_2, R_2)$... où $A_i = \text{ACT}_{\mathcal{F}}(q_i, x_i, q_{i+1})$, $P_i = \text{PRE}_{\mathcal{F}}(q_i, x_i, q_{i+1})$, $R_i = \text{REP}_{\mathcal{F}}(q_i, x_i, q_{i+1})$. L'ensemble des traces de $\mathcal{F}$ définit un langage noté $\mathcal{L}(\mathcal{F})$.

3.3.2 Sémantique

De la même manière que pour un AFR, la sémantique d'un FA $\mathcal{F}$ est donnée par un système de transition $\mathcal{M}$ (possiblement infini) appelée FIFFO-MACHINE, obtenue à partir de $\mathcal{F}$, en considérant toujours la file de mémorisation des événements et son évolution, et en étendant les fonctions ACT, PRE et REP aux configurations.

Définition 3.5 - Fiffo-Machine (FM)

Une FIFFO-MACHINE *(FM)* est un tuple $\mathcal{M} = (Q_\mathcal{M}, \Sigma_\mathcal{M}, \Delta_\mathcal{M}, q^0_\mathcal{M}, F_\mathcal{M}, \text{ACT}_\mathcal{M}, \text{PRE}_\mathcal{M}, \text{REP}_\mathcal{M})$ définie à partir d'un FA $\mathcal{F} = (Q_\mathcal{F}, \Sigma_\mathcal{F}, \Delta_\mathcal{F}, q^0_\mathcal{F}, A^0_\mathcal{F}, F_\mathcal{F}, \text{ACT}_\mathcal{F}, \text{PRE}_\mathcal{F}, \text{REP}_\mathcal{F})$,où $Q_\mathcal{M}$, $\Sigma_\mathcal{M}$, $\Delta_\mathcal{M}$ et $q^0_\mathcal{M}$ sont définis de la même manière que pour un SFR (définition 3.2),et où :

1. $F_\mathcal{M} = \{(q, w) \mid q \in Q_\mathcal{F}\} \subseteq Q_\mathcal{M}$ est l'*ensemble des configurations finales* de $\mathcal{M}$.
2. $\text{ACT}_\mathcal{M} : \Delta_\mathcal{M} \to 2^M$ est définie par $\text{ACT}_\mathcal{M}((q, w), x, (q', w')) = \text{ACT}_\mathcal{F}(q, x, q')$, pour toute transition $((q, w), x, (q', w')) \in \Delta_\mathcal{M}$.
3. $\text{PRE}_\mathcal{M} : \Delta_\mathcal{M} \to 2^M$ est définie par $\text{PRE}_\mathcal{M}((q, w), x, (q', w')) = \text{PRE}_\mathcal{F}(q, x, q')$, pour toute transition $((q, w), x, (q', w')) \in \Delta_\mathcal{M}$.
4. $\text{REP}_\mathcal{M} : \Delta_\mathcal{M} \to 2^M$ est définie par $\text{REP}_\mathcal{M}((q, w), x, (q', w')) = \text{REP}_\mathcal{F}(q, x, q')$, pour toute transition $((q, w), x, (q', w')) \in \Delta_\mathcal{M}$.

Un *chemin*

sur $\mathcal{M}$ est défini de la même manière que pour un SFR. L'ensemble de chemins sur $\mathcal{M}$ est noté $[\![\mathcal{M}]\!]$. Une *trace* de $\mathcal{M}$ est un mot fini ou infini m sur $(\Sigma_\mathcal{F} \times 2^M \times 2^M \times 2^M)$ tel qu'il existe un chemin $((q_0, w_0), x_0, (q_1, w_1))$, $((q_1, w_1), x_1, (q_2, w_2))$, $((q_2, w_2), x_2, (q_3, w_3))$,... sur $\mathcal{M}$ avec $m = (x_0, A_0, P_0, R_0)\ (x_1, A_1, P_1, R_1)\ (x_2, A_2, P_2, R_2)$... où $A_i = \text{ACT}_\mathcal{M}((q_i, w_i),\ x_i,\ (q_{i+1}, w_{i+1}))$, $P_i = \text{PRE}_\mathcal{M}((q_i, w_i),\ x_i,\ (q_{i+1}, w_{i+1}))$, $R_i = \text{REP}_\mathcal{M}((q_i, w_i),\ x_i,\ (q_{i+1}, w_{i+1}))$. L'ensemble des traces de $\mathcal{M}$ définit un langage noté $\mathcal{L}(\mathcal{M})$. De plus, nous pouvons bien entendu étendre la fonction $A_\mathcal{F}$ aux FM très simplement.

Définition 3.6 - Modules actifs dans une configuration ($A_\mathcal{M}$)

Soit une FM $\mathcal{M}$ définie à partir d'un FA $\mathcal{F}$, $A_\mathcal{F}$ donnant l'ensemble des modules actifs dans une configuration de $\mathcal{M}$ est définie par $A_\mathcal{M}(q, w) = A_\mathcal{F}(q)$, pour toute configuration $(q, w) \in Q_\mathcal{M}$.

3.3.3 Produit synchronisé

Dans la suite, plus particulièrement lorsque nous aborderons la répartition au chapitre 6, nous serons amenés à considérer des produits synchronisés de FIFFO-AUTOMATES. Dès lors, nous formalisons cette notion ci-après.

Définition 3.7 - Produit synchronisé de Fiffo-Automates
Le produit synchronisé de deux FA $\mathcal{F}_1 = (Q_{\mathcal{F}_1}, \Sigma_{\mathcal{F}_1}, \Delta_{\mathcal{F}_1}, q^0_{\mathcal{F}_1}, A^0_{\mathcal{F}_1}, F_{\mathcal{F}_1}, \text{ACT}_{\mathcal{F}_1}, \text{PRE}_{\mathcal{F}_1}, \text{REP}_{\mathcal{F}_1})$ et $\mathcal{F}_2 = (Q_{\mathcal{F}_2}, \Sigma_{\mathcal{F}_2}, \Delta_{\mathcal{F}_2}, q^0_{\mathcal{F}_2}, A^0_{\mathcal{F}_2}, F_{\mathcal{F}_2}, \text{ACT}_{\mathcal{F}_2}, \text{PRE}_{\mathcal{F}_2}, \text{REP}_{\mathcal{F}_2})$ (noté $\mathcal{F}_1 \times \mathcal{F}2$ dans la suite) est défini par :

1. $Q_{\mathcal{F}_1 \times \mathcal{F}_2} = Q_{\mathcal{F}_1} \times Q_{\mathcal{F}_2}$.
2. $\Sigma_{\mathcal{F}_1 \times \mathcal{F}_2} = \Sigma_{\mathcal{F}_1} \cup \Sigma_{\mathcal{F}_2}$.
3. $\Delta_{\mathcal{F}_1 \times \mathcal{F}_2}$ est défini par :
   (a) $((q_1, q_2), x, (q'_1, q'_2)) \in \Delta_{\mathcal{F}_1 \times \mathcal{F}_2}$ ssi $x \in \Sigma_{\mathcal{F}_1} \cap \Sigma_{\mathcal{F}_2}$, $(q_1, x, q'_1) \in \Delta_{\mathcal{F}_1}$ et $(q_2, x, q'_2) \in \Delta_{\mathcal{F}_2}$ avec $x \in \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$.
   (b) $((q_1, q_2), x, (q'_1, q_2)) \in \Delta_{\mathcal{F}}$ ssi $x \notin \Sigma_{\mathcal{F}_1} \cap \Sigma_{\mathcal{F}_2}$, $x \in \Sigma_{\mathcal{F}_1}$ et $(q_1, x, q'_1) \in \Delta_{\mathcal{F}_1}$.
   (c) $((q_1, q_2), x, (q_1, q'_2)) \in \Delta_{\mathcal{F}}$ ssi $x \notin \Sigma_{\mathcal{F}_1} \cap \Sigma_{\mathcal{F}_2}$, $x \in \Sigma_{\mathcal{F}_2}$ et $(q_2, x, q'_2) \in \Delta_{\mathcal{F}_2}$.
4. $q^0_{\mathcal{F}_1 \times \mathcal{F}_2} = (q^0_{\mathcal{F}_1}, q^0_{\mathcal{F}_2})$.
5. $A^0_{\mathcal{F}_1 \times \mathcal{F}_2} = A^0_{\mathcal{F}_1} \cup A^0_{\mathcal{F}_2}$.
6. $F_{\mathcal{F}_1 \times \mathcal{F}_2} = F_{\mathcal{F}_1} \times F_{\mathcal{F}_2}$.
7. $\text{ACT}_{\mathcal{F}_1 \times \mathcal{F}_2}$ est défini par :
   (a) $\text{ACT}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{ACT}_{\mathcal{F}_1}(q_1, x, q'_1) \cup \text{ACT}_{\mathcal{F}_2}(q_2, x, q'_2)$ si $x \in \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$.
   (b) $\text{ACT}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{ACT}_{\mathcal{F}_1}(q_1, x, q'_1)$ si $x \notin \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$ et $x \in \Sigma_{\mathcal{F}_1}$.
   (c) $\text{ACT}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{ACT}_{\mathcal{F}_2}(q_2, x, q'_2)$ si $x \notin \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$ et $x \in \Sigma_{\mathcal{F}_2}$.
8. $\text{PRE}_{\mathcal{F}_1 \times \mathcal{F}_2}$ est défini par :
   (a) $\text{PRE}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{PRE}_{\mathcal{F}_1}(q_1, x, q'_1) \cup \text{PRE}_{\mathcal{F}_2}(q_2, x, q'_2)$ si $x \in \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$.
   (b) $\text{PRE}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{PRE}_{\mathcal{F}_1}(q_1, x, q'_1)$ si $x \notin \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$ et $x \in \Sigma_{\mathcal{F}_1}$.
   (c) $\text{PRE}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{PRE}_{\mathcal{F}_2}(q_2, x, q'_2)$ si $x \notin \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$ et $x \in \Sigma_{\mathcal{F}_2}$.
9. $\text{REP}_{\mathcal{F}_1 \times \mathcal{F}_2}$ est défini par :
   (a) $\text{REP}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{REP}_{\mathcal{F}_1}(q_1, x, q'_1) \cup \text{REP}_{\mathcal{F}_2}(q_2, x, q'_2)$ si $x \in \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$.
   (b) $\text{REP}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{REP}_{\mathcal{F}_1}(q_1, x, q'_1)$ si $x \notin \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$ et $x \in \Sigma_{\mathcal{F}_1}$.
   (c) $\text{REP}_{\mathcal{F}_1 \times \mathcal{F}_2}((q_1, q_2), x, (q'_1, q'_2)) = \text{REP}_{\mathcal{F}_2}(q_2, x, q'_2)$ si $x \notin \Sigma_{\mathcal{F}_1 \times \mathcal{F}_2}$ et $x \in \Sigma_{\mathcal{F}_2}$.

3.3.4 Equivalences

Nous définissons ci-après, deux formes d'équivalence entre FA : l'*équivalence de trace* et la *bissimilarité*. Nous donnons ensuite une propriété reliant les deux formes d'équivalence. Pour terminer nous montrons que le produit synchronisé est congruent par rapport à la bissimilarité.

Afin de définir l'équivalence de trace, nous avons besoin de poser quelques notations. Soit un FA $\mathcal{F}$ et un état $q \in Q_{\mathcal{F}}$, nous noterons, $\mathcal{L}_{\mathcal{F}}(q)$ le langage défini par l'ensemble de tous les mots $m = (x_0, A_0, P_0, R_0)\ (x_1, A_1, P_1, R_1)\ (x_2, A_2, P_2, R_2)...$ tel qu'il existe une séquence de transitions $((q_0, w_0), x_0, (q_1, w_1))$, $((q_1, w_1), x_1, (q_2, w_2))$, $((q_2, w_2), x_2, (q_3, w_3))$,... avec $((q_i, w_i), x_i, (q_{i+1}, w_{i+1})) \in \Delta_{\mathcal{F}}$ et $q = q_0$.

Définition 3.8 - Equivalence de trace
Soit deux FA $\mathcal{F}_1$ et $\mathcal{F}_2$. Deux états $q_1 \in Q_{\mathcal{F}1}$ et $q_2 \in Q_{\mathcal{F}2}$ sont dits *équivalents* (au sens de l'*équivalence de trace*) si et seulement si $\mathcal{L}_{\mathcal{F}_1}(q_1) = \mathcal{L}_{\mathcal{F}_2}(q_2)$, où

Deux FA $\mathcal{F}_1$ et $\mathcal{F}_2$ sont dits *équivalents*, au sens de l'équivalence de trace, si $\mathcal{L}_{\mathcal{F}_1}(q^0_{\mathcal{F}_1}) = \mathcal{L}_{\mathcal{F}_2}(q^0_{\mathcal{F}_2})$ et $A^0_{\mathcal{F}_1} = A^0_{\mathcal{F}_2}$.

Définition 3.9 - Bissimulation
Soit deux FA $\mathcal{F}_1$ et $\mathcal{F}_2$. Une relation $\mathcal{B} \subseteq Q_{\mathcal{F}_1} \times Q_{\mathcal{F}_2}$ est une relation de *bissimulation* entre $\mathcal{F}_1$ et $\mathcal{F}_2$ si et seulement si pour tout $q_1 \in Q_{\mathcal{F}_1}$ et $q_2 \in Q_{\mathcal{F}_2}$, si $\mathcal{B}(q_1, q_2)$ alors :

1. Pour chaque état $q'_1 \in Q_{\mathcal{F}_1}$ tel que $(q_1, x, q'_1) \in \Delta_{\mathcal{F}_1}$, il existe un état $q'_2 \in Q_{\mathcal{F}_2}$ tel que $(q_2, x, q'_2) \in \Delta_{\mathcal{F}_2}$, que $\mathcal{B}(q'_1, q'_2)$, et que $\textsc{Act}_{\mathcal{F}_1}(q_1, x, q'_1) = \textsc{Act}_{\mathcal{F}_2}(q_2, x, q'_2)$, $\textsc{Pre}_{\mathcal{F}_1}(q_1, x, q'_1) = \textsc{Pre}_{\mathcal{F}_2}(q_2, x, q'_2)$ et $\textsc{Rep}_{\mathcal{F}_1}(q_1, x, q'_1) = \textsc{Rep}_{\mathcal{F}_2}(q_2, x, q'_2)$.
2. Pour chaque état $q'_2 \in Q_{\mathcal{F}_2}$ tel que $(q_2, x, q'_2) \in \Delta_{\mathcal{F}_2}$, il existe un état $q'_1 \in Q_{\mathcal{F}_1}$ tel que $(q_1, x, q'_1) \in \Delta_{\mathcal{F}_1}$,que $\mathcal{B}(q'_1, q'_2)$, et que $\textsc{Act}_{\mathcal{F}_1}(q_1, x, q'_1) = \textsc{Act}_{\mathcal{F}_2}(q_2, x, q'_2)$, $\textsc{Pre}_{\mathcal{F}_1}(q_1, x, q'_1) = \textsc{Pre}_{\mathcal{F}_2}(q_2, x, q'_2)$ et $\textsc{Rep}_{\mathcal{F}_1}(q_1, x, q'_1) = \textsc{Rep}_{\mathcal{F}_2}(q_2, x, q'_2)$.

Deux FA $\mathcal{F}_1$ et $\mathcal{F}_2$ sont dits *bissimilaires* (noté $\mathcal{F}_1 \equiv \mathcal{F}_2$) s'il existe une relation $\mathcal{B}$ de bissimulation entre $\mathcal{F}_1$ et $\mathcal{F}_2$ tel que $\mathcal{B}(q^0_{\mathcal{F}_1}, q^0_{\mathcal{F}_2})$ et si $A^0_{\mathcal{F}_1} = A^0_{\mathcal{F}_2}$. Remarquons que si $\mathcal{F}_1 \equiv \mathcal{F}_2$, alors $\Sigma_{\mathcal{F}_1} = \Sigma_{\mathcal{F}_2}$. En effet, s'il existe $x \in \Sigma_{\mathcal{F}_1}$ $(\Sigma_{\mathcal{F}_2})$ tel que $x \notin \Sigma_{\mathcal{F}_2}$ $(\Sigma_{\mathcal{F}_1})$, tout état $q_1 \in Q_{\mathcal{F}_1}$ $(q_2 \in Q_{\mathcal{F}_2})$ tel que $x \in out_{\mathcal{F}_1}(q_1)$ $(out_{\mathcal{F}_2}(q_2))$, il n'existe aucun état q_2 (q_1) ayant une transition sortante étiquetée x, puisque $x \notin \Sigma_{\mathcal{F}_1}$ $(\Sigma_{\mathcal{F}_2})$.

Propriété 3.2

Soit deux FA $\mathcal{F}_1$ et $\mathcal{F}_2$. Si $\mathcal{F}_1$ et $\mathcal{F}_2$ sont bissimilaires, alors $\mathcal{F}_1$ et $\mathcal{F}_2$ sont équivalents au sens de l'équivalence de trace : $(\mathcal{F}_1 \equiv \mathcal{F}_2) \rightarrow (\mathcal{L}(\mathcal{F}_1) = \mathcal{L}(\mathcal{F}_2))$

Théorème 3.3

Le produit synchronisé est congruent par rapport à la bissimulation. Soit $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$ et $\mathcal{F}_4$ des FA :

$$\left(\begin{array}{c} \mathcal{F}_1 \equiv \mathcal{F}_2 \\ \wedge \\ \mathcal{F}_3 \equiv \mathcal{F}_4 \end{array} \right) \rightarrow (\mathcal{F}_1 \times \mathcal{F}_3 \equiv \mathcal{F}_2 \times \mathcal{F}_4)$$

Démonstration

Soit $\mathcal{B}_{(1,2)}$ la relation de bissimulation entre $\mathcal{F}_1$ et $\mathcal{F}_2$, et $\mathcal{B}_{(3,4)}$ la relation de bissimulation entre $\mathcal{F}_3$ et $\mathcal{F}_4$. Nous contruisons $\mathcal{B} \subseteq (Q_{\mathcal{F}_1 \times \mathcal{F}_3} \times Q_{\mathcal{F}_2 \times \mathcal{F}_4})$ tel que $\mathcal{B}((q_1, q_3), (q_2, q_4))$ si et seulement si $(\mathcal{B}_{(1,2)}(q_1, q_2) \wedge \mathcal{B}_{(3,4)}(q_3, q_4))$. Nous montrons ensuite que $\mathcal{B}$ est une relation de bissimulation entre $\mathcal{F}_1 \times \mathcal{F}_3$ et $\mathcal{F}_2 \times \mathcal{F}_4$. Soit $(q_1, q_3) \in Q_{\mathcal{F}_1 \times \mathcal{F}_3}$ et $(q_2, q_4) \in Q_{\mathcal{F}_2 \times \mathcal{F}_4}$ tels que $\mathcal{B}((q_1, q_3), (q_2, q_4))$:

1. pour tout $(q'_1, q'_3) \in Q_{\mathcal{F}_1 \times \mathcal{F}_3}$ tel que $((q_1, q_3), x, (q'_1, q'_3)) \in \Delta_{\mathcal{F}_1 \times \mathcal{F}_3}$, nous devons montrer qu'il existe (q'_2, q'_4) tel que $((q_2, q_4), x, (q'_2, q'_4)) \in \Delta_{\mathcal{F}_2 \times \mathcal{F}_4}$. Comme $((q_1, q_3), x, (q'_1, q'_3)) \in \Delta_{\mathcal{F}_1 \times \mathcal{F}_3}$:
   (a) soit $x \in \Sigma_{\mathcal{F}_1} \cap \Sigma_{\mathcal{F}_3}$ avec $(q_1, x, q'_1) \in \Delta_{\mathcal{F}_1}$ et $(q_3, x, q'_3) \in \Delta_{\mathcal{F}_3}$. Dans ce cas, par définition de $\mathcal{B}$, nous savons que $\mathcal{B}_{(1,2)}(q_1, q_2)$ et $\mathcal{B}_{(3,4)}(q_3, q_4)$. Par conséquent, il existe $q'_2 \in Q_{\mathcal{F}_2}$ et $q'_4 \in Q_{\mathcal{F}_4}$ tels que $(q_2, x, q'_2) \in \Delta_{\mathcal{F}_2}$ et $(q_4, x, q'_4) \in \Delta_{\mathcal{F}_4}$ avec $\mathcal{B}_{(1,2)}(q'_1, q'_2)$ et $\mathcal{B}_{(3,4)}(q'_3, q'_4)$. Par la définition du produit synchronisé nous avons bien $((q_2, q_4), x, (q'_2, q'_4)) \in \Delta_{\mathcal{F}_2 \times \mathcal{F}_4}$. De plus, par définition de $\mathcal{B}$, nous avons bien $\mathcal{B}((q'_1, q'_3), (q'_2, q'_4))$.
   (b) soit $x \notin \Sigma_{\mathcal{F}_1} \cap \Sigma_{\mathcal{F}_3}$ et $x \in \Sigma_{\mathcal{F}_1}$ avec $(q_1, x, q'_1) \in \Delta_{\mathcal{F}_1}$. Dans ce cas, nous savons que $x \in \Sigma_{\mathcal{F}_1}$, et donc $x \in \Sigma_{\mathcal{F}_2}$, puisque $\mathcal{F}_1 \equiv \mathcal{F}_2$. De même, comme $x \notin \Sigma_{\mathcal{F}_3}$, nous savons que $x \notin \Sigma_{\mathcal{F}_4}$, puisque $\mathcal{F}_3 \equiv \mathcal{F}_4$. De plus, dans la transition $((q_1, q_3), x, (q'_1, q'_3))$, $q_3 = q'_3$ puisque $x \notin \Sigma_{\mathcal{F}_3}$. Nous choisissons donc $q'_4 = q_4$. Nous savons également, par définition de $\mathcal{B}$, que $\mathcal{B}_{(1,2)}(q_1, q_2)$. Par conséquent, il existe $q'_2 \in Q_{\mathcal{F}_2}$ tel que $(q_2, x, q'_2) \in \Delta_{\mathcal{F}_2}$ avec $\mathcal{B}_{(1,2)}(q'_1, q'_2)$. Par définition du produit synchronisé, nous avons bien $((q_2, q_4), x, (q'_2, q_4)) \in \Delta_{\mathcal{F}_2 \times \mathcal{F}_4}$. De plus, par définition de $\mathcal{B}$, nous avons bien $\mathcal{B}((q'_1, q_3), (q'_2, q_4))$, puisque $\mathcal{B}_{(1,2)}(q'_1, q'_2)$ et $\mathcal{B}_{(3,4)}(q_3, q_4)$.

(c) soit $x \notin \Sigma_{\mathcal{F}_1} \cap \Sigma_{\mathcal{F}_3}$ et $x \in \Sigma_{\mathcal{F}_3}$ avec $(q_3, x, q_3') \in \Delta_{\mathcal{F}_3}$. La démonstration est similaire au cas (b).

De plus, dans les trois cas précédents :

(a) $\text{ACT}_{\mathcal{F}_1 \times \mathcal{F}_3}((q_1, q_3), x, (q_1', q_3'))$

$$\begin{aligned} &= \text{ACT}_{\mathcal{F}_1}(q_1, x, q_1') \cup \text{ACT}_{\mathcal{F}_3}(q_3, x, q_3') && \text{(par définition de } \times) \\ &= \text{ACT}_{\mathcal{F}_2}(q_2, x, q_2') \cup \text{ACT}_{\mathcal{F}_4}(q_4, x, q_4') && (\mathcal{F}_1 \equiv \mathcal{F}_2 \text{ et } \mathcal{F}_3 \equiv \mathcal{F}_4) \\ &= \text{ACT}_{\mathcal{F}_1 \times \mathcal{F}_3}((q_2, q_4), x, (q_2', q_4')) && \text{(par définition de } \times) \end{aligned}$$

(b) $\text{PRE}_{\mathcal{F}_1 \times \mathcal{F}_3}((q_1, q_3), x, (q_1', q_3'))$

$$\begin{aligned} &= \text{PRE}_{\mathcal{F}_1}(q_1, x, q_1') \cup \text{PRE}_{\mathcal{F}_3}(q_3, x, q_3') && \text{(par définition de } \times) \\ &= \text{PRE}_{\mathcal{F}_2}(q_2, x, q_2') \cup \text{PRE}_{\mathcal{F}_4}(q_4, x, q_4') && (\mathcal{F}_1 \equiv \mathcal{F}_2 \text{ et } \mathcal{F}_3 \equiv \mathcal{F}_4) \\ &= \text{PRE}_{\mathcal{F}_1 \times \mathcal{F}_3}((q_2, q_4), x, (q_2', q_4')) && \text{(par définition de } \times) \end{aligned}$$

(c) $\text{REP}_{\mathcal{F}_1 \times \mathcal{F}_3}((q_1, q_3), x, (q_1', q_3'))$

$$\begin{aligned} &= \text{REP}_{\mathcal{F}_1}(q_1, x, q_1') \cup \text{REP}_{\mathcal{F}_3}(q_3, x, q_3') && \text{(par définition de } \times) \\ &= \text{REP}_{\mathcal{F}_2}(q_2, x, q_2') \cup \text{REP}_{\mathcal{F}_4}(q_4, x, q_4') && (\mathcal{F}_1 \equiv \mathcal{F}_2 \text{ et } \mathcal{F}_3 \equiv \mathcal{F}_4) \\ &= \text{REP}_{\mathcal{F}_1 \times \mathcal{F}_3}((q_2, q_4), x, (q_2', q_4')) && \text{(par définition de } \times) \end{aligned}$$

2. pour tout $(q_2', q_4') \in Q_{\mathcal{F}_2 \times \mathcal{F}_4}$ tel que $((q_2, q_4), x, (q_2', q_4')) \in \Delta_{\mathcal{F}_2 \times \mathcal{F}_4}$, nous devons montrer qu'il existe (q_1', q_3') tel que $((q_1, q_3), x, (q_1', q_3')) \in \Delta_{\mathcal{F}_1 \times \mathcal{F}_3}$. La démonstration dans ce cas est similaire au cas 1.

Pour terminer, par définition du produit synchronisé, nous savons que :

1. $q^0_{\mathcal{F}_1 \times \mathcal{F}_3} = (q^0_{\mathcal{F}_1}, q^0_{\mathcal{F}_3})$
2. $q^0_{\mathcal{F}_2 \times \mathcal{F}_4} = (q^0_{\mathcal{F}_2}, q^0_{\mathcal{F}_4})$
3. $A^0_{\mathcal{F}_1 \times \mathcal{F}_3} = A^0_{\mathcal{F}_1} \cup A^0_{\mathcal{F}_3}$
4. $A^0_{\mathcal{F}_2 \times \mathcal{F}_4} = A^0_{\mathcal{F}_2} \cup A^0_{\mathcal{F}_4}$

et, par hypothèse, que

1. $\mathcal{B}_{(1,2)}(q^0_{\mathcal{F}_1}, q^0_{\mathcal{F}_2})$
2. $\mathcal{B}_{(3,4)}(q^0_{\mathcal{F}_3}, q^0_{\mathcal{F}_4})$
3. $A^0_{\mathcal{F}_1} = A^0_{\mathcal{F}_2}$
4. $A^0_{\mathcal{F}_3} = A^0_{\mathcal{F}_4}$

Par conséquent, nous avons bien $\mathcal{B}((q^0_{\mathcal{F}_1}, q^0_{\mathcal{F}_3}), (q^0_{\mathcal{F}_2}, q^0_{\mathcal{F}_4}))$ (par définition de $\mathcal{B}$) et $A^0_{\mathcal{F}_1 \times \mathcal{F}_3} = A^0_{\mathcal{F}_2 \times \mathcal{F}_4}$ (par hypothèse). De là, découle $\mathcal{F}_1 \times \mathcal{F}_3 \equiv \mathcal{F}_2 \times \mathcal{F}_4$. ■

Chapitre 4

Logique temporelle

Dans les chapitres 2 et 3, nous avons présenté le langage ELECTRE et le modèle des FIFFO-AUTOMATES qui permettent de spécifier des systèmes réactifs. Afin de vérifier ces systèmes, il est nécessaire de spécifier un certain nombre de propriétés comportementales que le système doit vérifier afin d'être considéré comme correct. Le formalisme utilisé pour spécifier ces propriétés est la *logique temporelle*.

La logique temporelle fut d'abord introduite par les philosophes pour étudier la façon dont le temps était utilisé dans des arguments du langage naturel. Elle fut ensuite introduite dans le model checking afin d'exprimer des propriétés décrivant l'ordre selon lequel des actions se produisent dans des systèmes informatiques. Nous présentons, dans ce chapitre, deux formes de logique temporelle : linéaire (LTL), et branchante (CTL), respectivement aux sections 4.1 et 4.2.

4.1 Linear Temporal Logic (LTL)

Dans la logique temporelle linéaire, le temps est vu de manière linéaire : une succession d'étapes. En chaque étape, il existe un ensemble de propriétés atomiques qui sont vérifiées. Dans notre cas, ces propositions atomiques modélisent soit l'occurrence, la mémorisation ou le traitement différé[1] d'événements, soit l'activation, la préemption ou la reprise au début de modules. Nous noterons dans la suite $P = (\{?, !\} \times E_M) \cup E \cup (\{\text{ACT}, \text{PRE}, \text{REP}\} \times M)$ l'ensemble de telles propositions. A partir de cet ensemble de propositions, nous pouvons formaliser la notion de temps sous la forme d'une *structure temporelle linéaire* définie ci-après.

[1] de la file d'attente

Définition 4.1 - Structure temporelle linéaire
Une *structure temporelle linéaire* σ est une séquence finie ou infinie d'ensemble de propositions telle que $\forall i, 0 < i < |\sigma|, \sigma_i \in 2^P$, où :

- $|\sigma| = n + 1$ si $\sigma = \sigma_1, \sigma_2, ..., \sigma_n$ est finie.
- $|\sigma| = +\infty$ si $\sigma = \sigma_1, \sigma_2, ...$ est infinie.

4.1.1 Syntaxe

Une formule de la logique LTL est construite sur l'ensemble des propositions grâce aux opérateurs de la logique propositionnelle $(\wedge, \neg)$ et aux opérateurs temporels : *next* $(\bigcirc)$ et *until* $(\mathcal{U})$. La syntaxe de la logique LTL est présentée ci-après en BNF, où Φ dénote une formule LTL et p une proposition.

$$\Phi \quad ::= \quad true \mid false \mid p \mid \neg\Phi \mid \Phi \vee \Phi \mid \bigcirc\Phi \mid \Phi \,\mathcal{U}\, \Phi$$

De plus, nous utiliserons dans la suite les notations suivantes :

$$\begin{aligned}
\Phi_1 \wedge \Phi_2 &= \neg(\neg\Phi_1 \vee \neg\Phi_2) \\
\Phi_1 \rightarrow \Phi_2 &= \neg\Phi_1 \vee \Phi_2 \\
\Phi_1 \leftrightarrow \Phi_2 &= (\Phi_1 \rightarrow \Phi_2) \wedge (\Phi_2 \rightarrow \Phi_1) \\
\Diamond\Phi &= true \,\mathcal{U}\, \Phi \\
\Box\Phi &= \neg\Diamond\neg\Phi
\end{aligned}$$

4.1.2 Sémantique

La sémantique d'une formule LTL est définie sur une structure temporelle linéaire σ et un entier i :

$$\begin{aligned}
(\sigma, i) &\models true \\
(\sigma, i) &\not\models false \\
(\sigma, i) &\models p && \text{si et seulement si } 0 < i < |\sigma| \text{ et } p \in \sigma_i \\
(\sigma, i) &\models \neg\Phi && \text{si et seulement si } (\sigma, i) \not\models \Phi \\
(\sigma, i) &\models \Phi_1 \vee \Phi_2 && \text{si et seulement si } (\sigma, i) \models \Phi_1 \text{ ou } (\sigma, i) \models \Phi_2 \\
(\sigma, i) &\models \bigcirc\Phi && \text{si et seulement si } (\sigma, i+1) \models \Phi \\
(\sigma, i) &\models \Phi_1 \,\mathcal{U}\, \Phi_2 && \text{si et seulement si } \exists j \geq i, (\sigma, j) \models \Phi_2 \text{ et } \forall k,\ i \leq \\
& && k < j, (\sigma, k) \models \Phi_1
\end{aligned}$$

où $(\sigma, i) \models \Phi$ signifie que dans une structure σ, à l'indice i, la formule Φ est satisfaite. Une structure temporelle linéaire σ satisfait une formule LTL Φ (noté $\sigma \models \Phi$) si et seulement si $(\sigma, 1) \models \Phi$.

4.1.3 LTL et les FIFFO-AUTOMATE

Afin de s'assurer qu'une FA $\mathcal{F}$ vérifie une propriété modélisée par une formule LTL Φ, nous allons expliquer comment à partir d'un chemin c sur la FM $\mathcal{M}$ définissant la sémantique de $\mathcal{F}$, il est possible de construire une structure temporelle linéaire σ_c correspondante.

Définition 4.2 - Structure temporelle linéaire associée à un chemin
A partir d'une FM $\mathcal{M} = (Q_\mathcal{M}, \Sigma_\mathcal{M}, \Delta_\mathcal{M}, q^0_\mathcal{M}, A^0_\mathcal{M}, F_\mathcal{M}, \text{ACT}_\mathcal{M}, \text{PRE}_\mathcal{M}, \text{REP}_\mathcal{M})$ et d'un chemin fini ou infini c sur $\mathcal{M}$, nous construisons une structure temporelle linéaire σ^c de la manière suivante :

- si $c = ((q_0, w_0), x_0, (q_1, w_1)), ((q_1, w_1), x_1, (q_2, w_2)),..., ((q_{n-1}, w_{n-1}), x_{n-1}, (q_n, w_n))$ est fini, alors

  $$\sigma^c = \sigma^c_1, \sigma^c_2, ..., \sigma^c_n$$

  où $\forall i, 0 < i \leq n$, $\sigma^c_i = \{x_{i-1}\} \cup (\{\text{ACT}\} \times \text{ACT}_\mathcal{M}((q_{i-1}, w_{i-1}), x_{i-1}, (q_i, w_i))) \cup (\{\text{PRE}\} \times \text{PRE}_\mathcal{M}((q_{i-1}, w_{i-1}), x_{i-1}, (q_i, w_i))) \cup (\{\text{REP}\} \times \text{REP}_\mathcal{M}((q_{i-1}, w_{i-1}), x_{i-1}, (q_i, w_i)))$.
- si $c = ((q_0, w_0), x_0, (q_1, w_1)), ((q_1, w_1), x_1, (q_2, w_2)),...$ est infini, alors

  $$\sigma^c = \sigma^c_1, \sigma^c_2, ...$$

  où $\forall i > 0$, $\sigma^c_i = \{x_{i-1}\} \cup (\{\text{ACT}\} \times \text{ACT}_\mathcal{M}((q_{i-1}, w_{i-1}), x_{i-1}, (q_i, w_i))) \cup (\{\text{PRE}\} \times \text{PRE}_\mathcal{M}((q_{i-1}, w_{i-1}), x_{i-1}, (q_i, w_i))) \cup (\{\text{REP}\} \times \text{REP}_\mathcal{M}((q_{i-1}, w_{i-1}), x_{i-1}, (q_i, w_i)))$.

Une FM $\mathcal{M}$ satisfait une formule LTL Φ (noté $\mathcal{M} \models \Phi$) si et seulement si pour tout chemin $c \in [\![\mathcal{M}]\!]$, $\sigma^c \models \Phi$. Un FA $\mathcal{F}$ satisfait une formule LTL Φ (noté $\mathcal{F} \models \Phi$) si et seulement si la FM $\mathcal{M}$ définissant la sémantique de $\mathcal{F}$, satisfait Φ.

Théorème 4.1
Soit $\mathcal{F}_1$, $\mathcal{F}_2$ deux FA, et $\mathcal{M}_1$, $\mathcal{M}_2$ les FM en définissant la sémantique. Si $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$, alors pour toutes formules LTL Φ, $\mathcal{M}_1 \models \Phi$ si et seulement si $\mathcal{M}_2 \models \Phi$, et donc $\mathcal{F}_1 \models \Phi$ si et seulement si $\mathcal{F}_2 \models \Phi$ ■

Exemple 4.1

Nous présentons ici quelques exemples de formules LTL :

- "Est-ce que le module M sera finalement activé ?" : $\Diamond \text{ACT}[M_2]$
- "Est-ce que le module M se termine tout le temps ?" : $\Box(\text{ACT}[M_1] \rightarrow \Diamond end_{M_1})$

4.2 Computational Tree Logic (CTL)

Dans la logique CTL, le temps n'est plus vu, comme dans la logique LTL, de manière linéaire, mais de manière branchante. En effet, le temps est vu sous la forme d'une structure reprenant les différentes exécutions possibles. A nouveau, à chaque étape, il existe un ensemble de propriétés atomiques qui sont vérifiées. Ces propriétés sont modélisées par l'ensemble de propositions P défini précédemment. A partir de cet ensemble de propositions, nous pouvons formaliser cette structure sous la forme d'une *structure de Kripke* définie ci-après.

Définition 4.3 - Structure de Kripke

Une *structure de Kripke* M est définie par un tuple (S, S^0, R, μ) où

1. S est un ensemble d'états.
2. $S^0 \subseteq S$ est l'ensemble des états initiaux.
3. $R \subseteq S \times S$ est une relation de transition.
4. $\mu : S \rightarrow 2^P$ est une fonction qui étiquette les états de l'ensemble des propositions atomiques vraies en cet état.

Dans la suite, nous posons $Succ_M(s) = \{s' \in S | R(s, s')\}$ l'ensemble des successeurs de s dans une structure de Kripke.

4.2.1 Syntaxe

Une formule de la logique CTL est construite sur l'ensemble des propositions grâce aux opérateurs de la logique propositionnelle $(\vee, \neg)$ et aux opérateurs temporels définis précédemment $(\bigcirc, \mathcal{U})$ couplés à des quantificateurs sur les chemins : *existentiel* $(\exists)$ et *universel* $(\forall)$. La syntaxe de la logique CTL est présentée ci-après en BNF, où Φ dénote une formule CTL et p une proposition.

$$\Phi \quad ::= \quad true \mid false \mid p \mid \neg\Phi \mid \Phi \vee \Phi \mid \exists\bigcirc\Phi \mid \forall\bigcirc\Phi \mid \exists\Phi \,\mathcal{U}\, \Phi \mid \forall\Phi \,\mathcal{U}\, \Phi \mid$$

De plus, nous utiliserons dans la suite les notations suivantes :

$$
\begin{array}{lcl}
\Phi_1 \wedge \Phi_2 & = & \neg(\neg\Phi_1 \vee \neg\Phi_2) \\
\Phi_1 \rightarrow \Phi_2 & = & \neg\Phi_1 \vee \Phi_2 \\
\Phi_1 \leftrightarrow \Phi_2 & = & (\Phi_1 \rightarrow \Phi_2) \wedge (\Phi_2 \rightarrow \Phi_1) \\
\exists\Diamond\Phi & = & \exists\ true\ \mathcal{U}\ \Phi \\
\forall\Diamond\Phi & = & \forall\ true\ \mathcal{U}\ \Phi \\
\exists\Box\Phi & = & \neg\forall\Diamond\neg\Phi \\
\forall\Box\Phi & = & \neg\exists\Diamond\neg\Phi
\end{array}
$$

4.2.2 Sémantique

La sémantique d'une formule CTL est définie sur une structure de Kripke $M = (S, S^0, R, \mu)$ par :

$$
\begin{array}{lcll}
(M,s) & \models & true & \\
(M,s) & \not\models & false & \\
(M,s) & \models & p & \text{si et seulement si } p \in \mu(s_0) \\
(M,s) & \models & \neg\Phi & \text{si et seulement si } (M,s) \not\models \Phi \\
(M,s) & \models & \Phi_1 \vee \Phi_2 & \text{si et seulement si } (M,s) \models \Phi_1 \text{ ou } (M,s) \models \Phi_2 \\
(M,s) & \models & \exists\bigcirc\Phi & \text{si et seulement s'il existe } s' \in Succ_M(s) \text{ tel que} \\
 & & & (M,s') \models \Phi \\
(M,s) & \models & \forall\bigcirc\Phi & \text{si et seulement si pour tout } s' \in Succ_M(s) \text{ tel} \\
 & & & \text{que} \\
 & & & (M,s') \models \Phi \\
(M,s) & \models & \exists\Phi_1\ \mathcal{U}\ \Phi_2 & \text{si et seulement s'il existe une séquence d'états} \\
 & & & s_0, s_1, s_2, \ldots \text{ avec } R(s_0,s_1), R(s_1,s_2), \ldots \text{ et } s_0 = s, \\
 & & & \text{telle que } \exists\ j \geq 0, (M,s_j) \models \Phi_2 \wedge \forall k,\ 0 \leq k < j, \\
 & & & (M,s_k) \models \Phi_1 \\
(M,s) & \models & \forall\Phi_1\ \mathcal{U}\ \Phi_2 & \text{si et seulement si pour toutes séquences d'états} \\
 & & & s_0, s_1, s_2, \ldots \text{ avec } R(s_0,s_1), R(s_1,s_2), \ldots \text{ et } s_0 = s, \\
 & & & \text{telle que } \exists\ j \geq 0, (M,s_j) \models \Phi_2 \wedge \forall k,\ 0 \leq k < j, \\
 & & & (M,s_k) \models \Phi_1
\end{array}
$$

où $(M,s) \models \Phi$ signifie que dans une structure M, l'état s satisfait la formule Φ. Une structure de Kripke M satisfait une formule CTL Φ (noté $M \models \Phi$), si et seulement si

$(M, s) \models \Phi$ pour tout $s \in S^0$.

4.2.3 CTL et les FIFFO-AUTOMATES

De la même manière que nous l'avons fait à la section 4.1.3, afin de s'assurer qu'un FA $\mathcal{F}$ vérifie une propriété modélisée par une formule CTL Φ, nous allons expliquer comment construire une structure de Kripke associée à la FM $\mathcal{M}$ définissant la sémantique de $\mathcal{F}$.

Définition 4.4 - Structure de Kripke associée à une FM
A partir d'une FM $\mathcal{M} = (Q_\mathcal{M}, \Sigma_\mathcal{M}, \Delta_\mathcal{M}, q^0_\mathcal{M}, A^0_\mathcal{M}, F_\mathcal{M}, \text{ACT}_\mathcal{M}, \text{PRE}_\mathcal{M}, \text{REP}_\mathcal{M})$, nous construisons une structure de Kripke $M_\mathcal{M} = (S_\mathcal{M}, S^0_\mathcal{M}, R_\mathcal{M}, \mu_\mathcal{M})$ définie par :

1. $S_\mathcal{M} = \Delta_\mathcal{M}$.
2. $S^0_\mathcal{M} = \{((q, w), x, (q', w')) \in \Delta_\mathcal{M} | (q, w) = q^0_\mathcal{M}\}$.
3. $R_\mathcal{M} = \{(\ ((q_1, w_1), x_1, (q'_1, w'_1)), ((q_2, w_2), x_2, (q'_2, w_2))\ ) \mid (q'_1, w'_1) = (q_2, w_2)\}$
4. $\mu_\mathcal{M}$ est définie par $\mu_\mathcal{M}((q, w), x, (q', w')) = \{x\} \cup (\{\text{ACT}\} \times \text{ACT}_\mathcal{M}((q, w), x, (q', w'))) \cup (\{\text{PRE}\} \times \text{PRE}_\mathcal{M}((q, w), x, (q', w'))) \cup (\{\text{REP}\} \times \text{REP}_\mathcal{M}((q, w), x, (q', w')))$ pour toute transition $((q, w), x, (q', w')) \in \Delta_\mathcal{M}$.

Une FM $\mathcal{M}$ satisfait une formule CTL Φ (noté $\mathcal{M} \models \Phi$) si et seulement si $M_\mathcal{M} \models \Phi$. Un FA $\mathcal{F}$ satisfait une formule CTL Φ (noté $\mathcal{F} \models \Phi$) si et seulement si la FM $\mathcal{M}$ définissant la sémantique de $\mathcal{F}$, satisfait Φ.

Théorème 4.2
Soit deux FA $\mathcal{F}_1$ et $\mathcal{F}_2$. Si $\mathcal{F}_1 \equiv \mathcal{F}_2$ (bissimilaires), alors pour toutes formules CTL Φ, $\mathcal{F}_1 \models \Phi$ si et seulement si $\mathcal{F}_2 \models \Phi$. ■

Exemple 4.2
Nous présenons ici quelques exemples de formules CTL :

- "Pour tout chemin, est-ce que M_1 ou M_2 est finalement activé ?" : $\forall\Diamond(\text{ACT}[M_1] \vee \text{ACT}[M_2])$
- "Existe-t-il un chemin pour lequel une occurence de e est mémorisée ?" : $\exists\Diamond!e$

Deuxième partie

Compilation et répartition

Chapitre 5

Compilation

Dans le chapitre 2, nous avons présenté le langage ELECTRE. Le premier objectif du présent travail consiste à compiler ces programmes ELECTRE pour les rendre exécutables en Lego Mindstorms. Nous avons déjà, dans le chapitre 3, présenté la première étape de cette compilation, à savoir le passage d'un programme ELECTRE à un FIFFO-AUTOMATE. Dans le présent chapitre, nous allons expliquer comment, partant d'un tel FIFFO-AUTOMATE, il est possible de générer un programme legOS qui en reflète le comportement. Nous commençons, à la section 5.1 en décrivant comment sont codés les FIFFO-AUTOMATE. Nous poursuivons, à la section 5.2, en présentant comment les modules et les événements sont modélisés en terme de programme legOS, et à la section 5.3 en expliquant comment est traduit le FIFFO-AUTOMATE. Nous terminons, à la section 5.4, en donnant des éléments de justification, quant à la correction de la méthode.

5.1 Fichier source

Un FIFFO-AUTOMATE se présente sous la forme d'un fichier ".tef". A titre d'exemple, un morceau du fichier ".tef" correspondant au problème des lecteurs/écrivains sans priorité du chapitre 2 est présenté à la figure 5.1. Les détails de la syntaxe sont ici peu importants. Par contre, il est intéressant de remarquer que chaque état de l'automate est divisé en deux parties. En effet, en examinant les transitions partant, par exemple, de l'état 0 (lignes 27 à 35), nous pouvons remarquer que l'état est noté de deux manières différentes : St&0 et St0. Ces deux notations représentent deux parties de l'état : une partie dite *instable* (St&0) et une partie dite *stable* (St0). Le rôle de la partie instable est de prendre en compte les occurrences éventuelles de la file d'attente. S'il n'existe aucune occurrence de ce type, une "pseudo-transition" dite *de stabilisation*, étiquetée "&" (ligne 27 pour l'état 0), est

déclenchée. Celle-ci mène à la partie stable de l'état, dans laquelle sont prises en compte les occurrences de l'environnement. Remarquons que lorsque l'automate change d'état, il aboutit dans la partie instable de cet état. Cette méthode garantit que la priorité soit donnée aux occurrences mémorisées, en accord avec la politique de gestion FIFFO.

```
 1 **              ELECTRE compiler V6 (OCaml) - Feb 2000  **
 2 **                I R C Cy N     (U.M.R. 6597)          **
 3 **              BP 92101 -- 1, rue de la noe            **
 4 **              44321 NANTES CEDEX 03 FRANCE            **
 5 **                electre@irccyn .ec-nantes.fr          **
 6
 7 Source ELECTRE program:
 8 [1/{{#r1:R1|||#r2:R2}|#w1:W1|#w2:W2}]*.
 9
10 List of modules:
11 W2 W1 R2 R1.
12
13 List of events:
14 w2 w1 r2 r1 endW2 endW1 endR2 endR1.
15
16 List of fleeting events:
17 .
18
19 List of single storage events:
20 .
21
22 List of multiple storage events:
23 w2 w1 r2 r1.
24
25 Automaton:
26
27 St0-&->St&0:A=[] D=[]
28 St&0-r1->St1:P=[]
29 St0-&r1->St1:"-r1"
```

FIG. 5.1 – Lecteurs/Ecrivains - fichier ".tef"

```
30
31 St&0-r2->St2:P=[]
32 St0-&r2->St2:"-r2"
33 St&0-w1->St3:P=[]
34 St0-&w1->St3:"-w1"
35 St&0-w2->St4:P=[]
36 St0-&w2->St4:"-w2"
37
38 St1-&->St&1:A=[R1] D=[]
39 St&1-r1->St&1:"*r1"
40 St&1-w1->St&1:"*w1"
41 St&1-w2->St&1:"*w2"
42 St&1-endR1->St9:
43 St&1-r2->St10:P=[]
44 St1-&r2->St10:"-r2"
45
46 St2-&->St&2:A=[R2] D=[]
47 St&2-r2->St&2:"*r2"
48 St&2-w1->St&2:"*w1"
49 St&2-w2->St&2:"*w2"
50 St&2-endR2->St5:
51 St&2-r1->St6:P=[]
52 St2-&r1->St6:"-r1"
53
54 St3-&->St&3:A=[W1] D=[]
55 St&3-r1->St&3:"*r1"
56 St&3-r2->St&3:"*r2"
57 St&3-w1->St&3:"*w1"
58 St&3-w2->St&3:"*w2"
59 St&3-endW1->St3:
   ...
```

FIG. 5.1 – Lecteurs/Ecrivains - fichier ".tef" (suite)

5.2 Traduction des modules et des événements

Afin de traduire FIFFO-AUTOMATE, contenu dans un fichier ".tef" en un programme legOS, il est nécessaire de modéliser les événements et les modules de façon à être accessibles à ce programme.

Chaque événement sera modélisé, en legOS, par une fonction event_e_happens() qui renvoie la valeur 1 lorsque l'événement en question se produit et la valeur 0, sinon. Chaque module est, quant à lui, représenté par quatre fonctions :

1. module_M_activated() à exécuter lorsque le module est activé,
2. module_M_preempted() à exécuter lorsque le module est préempté,
3. module_M_restarted() à exécuter lorsque le module est repris au début,
4. module_M_stopped() à exécuter lorsque le module est arrêté

qui devront être complétées par l'utilisateur pour modéliser le comportement souhaité. Dans ce sens, nous divergeons de la sémantique naturelle du langage ELECTRE où les modules représentent des morceaux de code non bloquants qui s'exécutent en un temps fini. Ce choix est dû à une limitation de legOS. En effet, nous aurions pu modéliser un module par une fonction non bloquante. La reprise au début du module aurait été modélisée par le lancement d'une tâche (thread) legOS exécutant la fonction, la préemption aurait été modélisée par l'arrêt de cette tâche. Cependant, pour modéliser l'activation d'un module, il aurait fallu pouvoir reprendre l'exécution de la tâche à l'endroit où celle-ci aurait été arrêtée, ce qui, malheureusement, n'est pas possible en legOS. La fonction modélisant l'événement de terminaison de ce module devra renvoyer la valeur 1 lorsque le module a terminé son exécution. Remarquons que la modélisation que nous avons adoptée n'est pas trop restrictive dans le cadre d'application Lego Mindstorms. En effet, l'activation, préemption ou reprise au début des modules se traduisent souvent par la mise sous ou hors tension d'un moteur, ce qui est réalisable en appelant des fonctions de l'API legOS.

5.3 Traduction de l'automate

Pour simuler le comportement de cet automate, une variable permettra en permanence de connaître l'état dans lequel l'automate se trouve, et si ce dernier est dans la partie stable ou instable de cet état. Suivant l'information contenue dans cette variable, le programme devra s'attendre à prendre en compte des occurrences de différents événements. Ces occurrences proviendront de la file d'attente dans la partie instable de l'état, et de l'environnement dans sa partie stable.

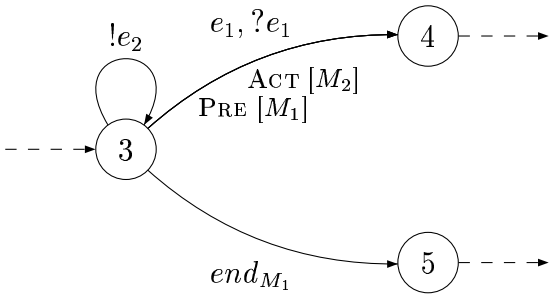


FIG. 5.2 – Exemple pour illustrer la compilation

Afin d'illustrer comment générer un programme C à partir d'un FA, considérons le morceau de FA de la figure 5.2. La figure 5.3 présente le morceau de la fonction principale du programme correspondant. Cette fonction principale consiste en une boucle, qui continue tant que l'état courant (currentState, ligne 1) n'est pas un état final. Dans le corps de la boucle, en fonction de l'état courant (switch(...) ligne 2), il faut déterminer l'ensemble des occurrences auxquelles l'automate est à même de répondre. Dans la partie instable de l'état (lignes 4 à 18), il faut examiner la file d'attente pour déterminer si une occurrence de cette file peut être traitée. Dans l'état 3 de l'exemple, seul l'événement e_1 est à considérer (ligne 5 à 6). Si une telle occurrence se présente, il faut la traiter, en activant, préemptant et/ou reprenant au début un ou plusieurs modules (lignes 10, 11), ainsi qu'en changeant d'état (ligne 12). Si, par contre, aucune occurrence de la file d'attente ne peut être traitée, (la fonction firstFireableEvent() renvoie alors NONE, lignes 7 et 14), il faut passer à la partie stable de l'état (ligne 15). Dans cette partie stable (lignes 19 à 38), il faut à nouveau vérifier si une occurrence se produit, mais cette fois de l'environnement, et en fonction de cette occurrence, effectuer les actions adéquates. Remarquons que lorsqu'il s'agit de mémoriser une occurrence d'événement, si cet événement est à mémorisation unique, il faut tester si la file d'attente ne contient pas déjà une occurrence avant de rajouter la nouvelle occurrence (ceci est géré par la fonction memorize(), ligne 32).

```
1  while(! final ( currentState )) {
2      switch(currentState ) {
3          (...)
4          case UNSTABLE_ST3:
5              cleanBitset (eventsToWaitFromQueue);
6              addToBitset(eventsToWaitFromQueue, EVENT_e1);
7              anEvent =  firstFireableEvent (theQueue, eventsToWaitFromQueue);
8              switch(anEvent) {
9                  case EVENT_e1:
10                     module_M1_prempted();
11                     module_M2_activated();
12                     currentState = UNSTABLE_ST4;
13                     break;
14                 case NONE:
15                     currentState = STABLE_ST3;
16                     break;
17             }
18             break;
19         case STABLE_ST3:
20             cleanBitSet (eventsToWaitFromEnv);
21             addToBitSet(eventsToWaitFromEnv, EVENT_e1);
22             addToBitSet(eventsToWaitFromEnv, EVENT_e2);
23             addToBitSet(eventsToWaitFromEnv, EVENT_endM1);
24             anEvent = wait(eventsToWaitFromEnv);
25             switch(anEvent) {
26                 case EVENT_e1:
27                     module_M1_prempted();
28                     module_M2_activated();
29                     currentState = UNSTABLE_ST4;
30                     break;
31                 case EVENT_e2:
32                     memorize(queue, EVENT_e2);
33                     break;
34                 case EVENT_endM1:
35                     module_M1_stopped();
36                     currentState = UNSTABLE_ST5;
37                     break;
38             }
39             break;
40         (...)
41     }
42 }
```

FIG. 5.3 – Fonction principale du programme

5.4 Correction

Nous avons, dans les sections précédentes, expliqué comment compiler un programme ELECTRE en programme C, qui peut ensuite être implanté sur la brique RCX. Dans l'introduction, nous avons motivé le besoin de systématiser cette étape de compilation pour éviter l'introduction d'erreurs par rapport au comportement initial du programme ELECTRE. Cependant, pour cela, il est impératif que la compilation soit correcte, en ce sens que le programme C produit soit cohérent vis-à-vis du programme ELECTRE de départ. C'est ce que nous allons essayer de justifier dans la suite de cette section.

La première phase de la compilation consiste à traduire le programme ELECTRE de départ en FIFFO-AUTOMATE. La cohérence est assurée par l'utilisation des règles de réécriture, constituant la sémantique opérationnelle du langage. En effet, le graphe de contrôle s'obtient explicitement en utilisant ces règles. Lors de la deuxième phase de traduction en programme C, la cohérence est garantie car le programme ainsi obtenu traduit fidèlement le comportement du FIFFO-AUTOMATE obtenu lors de la première phase, en ce sens que ce programme C reflète la FIFFO-MACHINE correspondante. Plus particulièrement la politique gestion FIFFO est respectée :

1. **priorité au traitement d'occurrences mémorisées** Dans la boucle du programme principal, chaque changement d'état aboutit dans la partie instable de cet état (ligne 12,29 et 36), où les occurrences d'événements de la file d'attente sont traitées. De plus, au départ, la variable d'état courant dénote la partie instable de l'état initial de l'automate. Les occurrences mémorisées sont donc bien traitées en priorité.
2. **traitement des occurrences "dès que possible"** Dans un état, ou plus exactement dans la partie instable de cet état, si une occurrence peut être traitée, elle sera détectée par la fonction firstFireableEvent(), et sera traitée juste après.
3. **priorité à l'occurrence mémorisée la plus ancienne** La fonction firstFireableEvent() passe en revue les éléments de la file d'attente en commençant par le début jusqu'à trouver une occurrence traitable dans l'état courant.

Chapitre 6

Répartition

Le deuxième point qui nous importe dans le présent travail est la répartition de programmes ELECTRE. Pour ce faire, nous allons utiliser la structure intermédiaire des FIFFO-AUTOMATES du chapitre 3. Notre but est de décomposer le FA résultant de la compilation du programme ELECTRE en plusieurs FA, chacun destiné à un site d'exécution. De cette manière, chaque FA pourra être compilé. La phase d'implémentation se déroulera en trois étapes de la manière suivante :

1. **Compilation en FA** La première étape consistera à compiler le programme ELECTRE en un FA en utilisant la méthode expliquée à la section 3.1, page 24.
2. **Répartition** Ensuite, ce FIFFO-AUTOMATE sera décomposé en différents FA par les méthodes abordées dans ce chapitre.
3. **Compilation pour legOS** Les différents FIFFO-AUTOMATES seront compilés en programmes C pour legOS par la méthode expliquée au chapitre 5.

La méthode de répartition se décompose en deux parties, présentées aux sections 6.1 et 6.2. La première consiste à construire, pour chaque site d'exécution, un FA destiné à y être implémenté. La deuxième, quant à elle, consiste à essayer de réduire les automates produits lors la première étape.

6.1 Projection

Dans un programme ELECTRE, et a fortiori dans un FIFFO-AUTOMATE, ce qu'il faut répartir sur différents sites, c'est l'exécution des modules. Avant de présenter la méthode proprement dite, nous allons faire l'hypothèse que pour chaque module, nous connaissons

le site sur lequel il sera exécuté. Si nous notons SITE, l'ensemble des sites d'exécution, nous pouvons formaliser cette connaissance par la *fonction de localisation de module* définie ci-après. Rappelons que M dénote l'ensemble des modules.

Définition 6.1 - Fonction de localisation de modules (λ_M)
La *fonction de localisation de modules* $\lambda_M : M \rightarrow \text{SITE}$ où $\lambda_M(M_i) = s_i$ signifie que l'exécution du module M_i se fera sur le site s_i.

Dans la suite, nous noterons $M_{s_i} = \lambda_M^{-1}(s_i)$ l'ensemble des modules localisés en s_i.

Nous avons expliqué précédemment que les FIFFO-AUTOMATE agissaient sur les modules lorsqu'une transition était tirée. Pour rappel, ces actions sont l'activation, la préemption et la reprise au début. Cependant, un FA "global" agit sur tous les modules d'un système, alors qu'un FA "réparti", destiné à être implémenté sur un site s_i bien particulier, ne doit agir que sur un sous-ensemble de ces modules : les modules gérés sur ce site d'exécution (M_{s_i}). Dès lors, afin de répartir un FA $\mathcal{F}$ sur plusieurs sites d'exécution, nous allons placer sur chaque site s_i, un FA $\mathcal{F}_i$ pour lequel les actions sont restreintes aux modules de M_{s_i}. Nous appelons cette étape la *projection*, formalisée ci-après.

Définition 6.2 - Projection (π_N)
La projection d'un FA $\mathcal{F} = (Q_\mathcal{F}, \Sigma_\mathcal{F}, \Delta_\mathcal{F}, q^0_\mathcal{F}, A^0_\mathcal{F}, F_\mathcal{F}, \text{ACT}_\mathcal{F}, \text{PRE}_\mathcal{F}, \text{REP}_\mathcal{F})$ sur un ensemble de modules $N \subseteq M$ (noté $\pi_N(\mathcal{F})$) est un FA défini par :

1. $Q_{\pi_N(\mathcal{F})} = Q_\mathcal{F}$.
2. $\Sigma_{\pi_N(\mathcal{F})} = \Sigma_\mathcal{F}$.
3. $\Delta_{\pi_N(\mathcal{F})} = \Delta_\mathcal{F}$.
4. $q^0_{\pi_N(\mathcal{F})} = q^0_\mathcal{F}$.
5. $A^0_{\pi_N(\mathcal{F})} = A^0_\mathcal{F} \cap N$.
6. $F_{\pi_N(\mathcal{F})} = F_\mathcal{F}$.
7. $\text{ACT}_{\pi_N(\mathcal{F})}(q, x, q') = \text{ACT}_\mathcal{F}(q, x, q') \cap N$, $\forall (q, x, q') \in \Delta_{\pi_N(\mathcal{F})}$.
8. $\text{PRE}_{\pi_N(\mathcal{F})}(q, x, q') = \text{PRE}_\mathcal{F}(q, x, q') \cap N$, $\forall (q, x, q') \in \Delta_{\pi_N(\mathcal{F})}$.
9. $\text{REP}_{\pi_N(\mathcal{F})}(q, x, q') = \text{REP}_\mathcal{F}(q, x, q') \cap N$, $\forall (q, x, q') \in \Delta_{\pi_N(\mathcal{F})}$.

Théorème 6.1

Soit $\mathcal{F}$ un FA et $N \subseteq M$ un sous-ensemble de modules. $\mathcal{F}$ et le produit synchronisé des projections respectives de $\mathcal{F}$ sur N et son complément, sont bissimilaires :

$$\mathcal{F} \equiv \pi_N(\mathcal{F}) \times \pi_{M/N}(\mathcal{F})$$

Démonstration

Nous commençons par construire $\pi_N(\mathcal{F}) \times \pi_{M/N}(\mathcal{F})$:

1. $Q_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})} = Q_{\pi_N(\mathcal{F})} \times Q_{\pi_{M/N}(\mathcal{F})}$ (par définition de $\times$)
   $= Q_{\mathcal{F}} \times Q_{\mathcal{F}}$ (par définition de π)
2. $\Sigma_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})} = \Sigma_{\pi_N(\mathcal{F})} \cup \Sigma_{\pi_{M/N}(\mathcal{F})}$ (par définition de $\times$)
   $= \Sigma_{\mathcal{F}} \cup \Sigma_{\mathcal{F}}$ (par définition de π)
   $= \Sigma_{\mathcal{F}}$ (par la théorie des ensembles)
3. $((q_1, q_2), x, (q_1', q_2')) \in \Delta_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$ ssi $(q_1, x, q_1') \in \Delta_{\pi_N(\mathcal{F})}$ et $(q_2, x, q_2') \in \Delta_{\pi_{M/N}(\mathcal{F})}$ avec $x \in \Sigma_{\mathcal{F}_1\times\mathcal{F}_2}$. (par définition de $\times$)
   $((q_1, q_2), x, (q_1', q_2')) \in \Delta_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$ ssi $(q_1, x, q_1') \in \Delta_{\mathcal{F}}$ et $(q_2, x, q_2') \in \Delta_{\mathcal{F}}$ avec $x \in \Sigma_{\mathcal{F}}$. (par définition de π)
   Remarquons que comme x est toujours dans l'intersection des alphabets, seules des transitions du type (a) de la définition de $\times$ (transitions avec synchronisation) sont permises.
4. $q^0_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})} = (q^0_{\pi_N(\mathcal{F})}, q^0_{\pi_{M/N}(\mathcal{F})})$ (par définition de $\times$)
   $= (q^0_{\mathcal{F}}, q^0_{\mathcal{F}})$ (par définition de π)
5. $A^0_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})} = A^0_{\pi_N(\mathcal{F})} \cup A^0_{\pi_{M/N}(\mathcal{F})}$ (par définition de $\times$)
   $= (A^0_{\mathcal{F}} \cap N) \cup (A^0_{\mathcal{F}} \cap (M/N))$ (par définition de π)
   $= A^0_{\mathcal{F}}$ (par la théorie des ensembles)
6. $F_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})} = F_{\pi_N(\mathcal{F})} \times F_{\pi_{M/N}(\mathcal{F})}$ (par définition de $\times$)
   $= F_{\mathcal{F}} \times F_{\mathcal{F}}$ (par définition de π)
7. $\mathrm{ACT}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1, q_2), x, (q_1', q_2'))$
   $= \mathrm{ACT}_{\pi_N(\mathcal{F})}(q_1, x, q_1') \cup \mathrm{ACT}_{\pi_{M/N}(\mathcal{F})}(q_2, x, q_2')$ (par définition de $\times$)
   $= (\mathrm{ACT}_{\mathcal{F}}(q_1, x, q_1') \cap N) \cup (\mathrm{ACT}_{\mathcal{F}}(q_2, x, q_2') \cap (M/N))$ (par définition de π)

8. $\text{PRE}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1,q_2),x,(q_1',q_2'))$
$= \text{PRE}_{\pi_N(\mathcal{F})}(q_1,x,q_1') \cup \text{PRE}_{\pi_{M/N}(\mathcal{F})}(q_2,x,q_2')$
(par définition de $\times$)
$= (\text{PRE}_{\mathcal{F}}(q_1,x,q_1') \cap N) \cup (\text{PRE}_{\mathcal{F}}(q_2,x,q_2') \cap (M/N))$
(par définition de π)

9. $\text{REP}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1,q_2),x,(q_1',q_2'))$
$= \text{REP}_{\pi_N(\mathcal{F})}(q_1,x,q_1') \cup \text{REP}_{\pi_{M/N}(\mathcal{F})}(q_2,x,q_2')$
(par définition de $\times$)
$= (\text{REP}_{\mathcal{F}}(q_1,x,q_1') \cap N) \cup (\text{REP}_{\mathcal{F}}(q_2,x,q_2') \cap (M/N))$
(par définition de π)

Nous montrons ensuite par induction que dans $\pi_N(\mathcal{F}) \times \pi_{M/N}(\mathcal{F})$, seuls les états (q_1,q_2) tels que $q_1 = q_2$ sont atteignables :

Etape initiale Par construction, l'état initial $q^0_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})} = (q^0_{\mathcal{F}}, q^0_{\mathcal{F}})$. Nous avons bien $q^0_{\mathcal{F}} = q^0_{\mathcal{F}}$.

Etape d'induction Soit un état (q_1,q_2) atteignable tel que $q_1 = q_2$. Par construction, $\forall x \in \Sigma_{\mathcal{F}}$, $((q_1,q_2),x,(q_1',q_2')) \in \Delta_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$ ssi $(q_1,x,q_1') \in \Delta_{\mathcal{F}}$ et $(q_2,x,q_2') \in \Delta_{\mathcal{F}}$. Or, comme $q_1 = q_2$ et que $\mathcal{F}$ est déterministe (théorème 5.2), il faut $q_1' = q_2'$.

Nous poursuivons, en posant $\mathcal{B} \subseteq Q_{\mathcal{F}} \times Q_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$ telle que $\mathcal{B}(q,(q_1,q_2))$ si et seulement si $q_1 = q_2 = q$ et en montrant que $\mathcal{B}$ est une relation de bissimulation entre $\mathcal{F}$ et $\pi_N(\mathcal{F}) \times \pi_{M/N}(\mathcal{F})$. En effet, pour tout $q \in Q_{\mathcal{F}}$ et $(q_1,q_2) \in Q_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$, si $\mathcal{B}(q,(q_1,q_2))$ alors :

1. Pour chaque état $q' \in Q_{\mathcal{F}}$ tel que $(q,x,q') \in \Delta_{\mathcal{F}}$, il faut qu'il existe un état $(q_1',q_2') \in Q_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$ tel que $((q_1,q_2),x,(q_1',q_2')) \in \Delta_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$ et que $\mathcal{B}(q',(q_1',q_2'))$. Soit cet état (q_1',q_2')=(q',q'). Comme $\mathcal{B}(q,(q_1,q_2))$, $q = q_1 = q_2$ et nous avons bien $((q,q),x,(q',q')) \in \Delta_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$. En effet, par construction ceci est vrai si et seulement si $(q,x,q') \in \Delta_{\mathcal{F}}$, ce qui est le cas. De plus
   (a) $\text{ACT}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1,q_2),\ x,\ (q_1',q_2'))$
   $= \text{ACT}_{\pi_N(\mathcal{F})}(q_1,x,q_1') \cup \text{ACT}_{\pi_{M/N}(\mathcal{F})}(q_2,x,q_2')$
   (par définition de $\times$)
   $= \text{ACT}_{\pi_N(\mathcal{F})}(q,x,q') \cup \text{ACT}_{\pi_{M/N}(\mathcal{F})}(q,x,q')$
   ($q = q_1 = q_2$ et $q' = q_1' = q_2'$)

$$= (\text{ACT}_{\mathcal{F}}(q,x,q') \cap N) \cup (\text{ACT}_{\mathcal{F}}(q,x,q') \cap (M/N))$$
(par définition de π)
$$= \text{ACT}_{\mathcal{F}}(q,x,q')$$ (par la théorie des ensembles)

(b) $\text{PRE}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1,q_2),\, x,\, (q_1',q_2'))$
$$= \text{PRE}_{\pi_N(\mathcal{F})}(q_1,x,q_1') \cup \text{PRE}_{\pi_{M/N}(\mathcal{F})}(q_2,x,q_2')$$
(par définition de $\times$)
$$= \text{PRE}_{\pi_N(\mathcal{F})}(q,x,q') \cup \text{PRE}_{\pi_{M/N}(\mathcal{F})}(q,x,q')$$
($q = q_1 = q_2$ et $q' = q_1' = q_2'$)
$$= (\text{PRE}_{\mathcal{F}}(q,x,q') \cap N) \cup (\text{PRE}_{\mathcal{F}}(q,x,q') \cap (M/N))$$
(par définition de π)
$$= \text{PRE}_{\mathcal{F}}(q,x,q')$$ (par la théorie des ensembles)

(c) $\text{REP}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1,q_2),\, x,\, (q_1',q_2'))$
$$= \text{REP}_{\pi_N(\mathcal{F})}(q_1,x,q_1') \cup \text{REP}_{\pi_{M/N}(\mathcal{F})}(q_2,x,q_2')$$
(par définition de $\times$)
$$= \text{REP}_{\pi_N(\mathcal{F})}(q,x,q') \cup \text{REP}_{\pi_{M/N}(\mathcal{F})}(q,x,q')$$
($q = q_1 = q_2$ et $q' = q_1' = q_2'$)
$$= (\text{REP}_{\mathcal{F}}(q,x,q') \cap N) \cup (\text{REP}_{\mathcal{F}}(q,x,q') \cap (M/N))$$
(par définition de π)
$$= \text{REP}_{\mathcal{F}}(q,x,q')$$ (par la théorie des ensembles)

2. Pour chaque état $(q_1', q_2') \in Q_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$ tel que $((q_1,q_2),x,(q_1',q_2')) \in \Delta_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}$, il faut qu'il existe un état $q' \in Q_{\mathcal{F}}$ tel que $(q,x,q') \in \Delta_{\mathcal{F}}$ et que $\mathcal{B}(q',(q_1',q_2'))$. Soit cet état $q' = q_1' = q_2'$ (nous savons que (q_1',q_2') atteignable et donc $q_1' = q_2'$). Comme $\mathcal{B}(q,(q_1,q_2))$, $q = q_1 = q_2$, et nous avons $((q_1,q_2),x,(q_1',q_2')) = ((q,q),x,(q',q'))$. Par conséquent, $(q,x,q') \in \Delta_{\mathcal{F}}$.
   Comme précédemment,
   (a) $\text{ACT}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1,q_2),\, x,\, (q_1',q_2')) = \text{ACT}_{\mathcal{F}}(q,x,q')$
   (b) $\text{PRE}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1,q_2),\, x,\, (q_1',q_2')) = \text{PRE}_{\mathcal{F}}(q,x,q')$
   (c) $\text{REP}_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})}((q_1,q_2),\, x,\, (q_1',q_2')) = \text{REP}_{\mathcal{F}}(q,x,q')$

Par définition de $\mathcal{B}$, nous avons bien $\mathcal{B}(q_{\mathcal{F}}^0,(q_{\mathcal{F}}^0,q_{\mathcal{F}}^0))$ et par construction, nous avons bien $A^0_{\pi_N(\mathcal{F})\times\pi_{M/N}(\mathcal{F})} = A^0_{\mathcal{F}}$. Dès lors, $\mathcal{F}$ et $\pi_N(\mathcal{F}) \times \pi_{M/N}(\mathcal{F})$ sont bien bissimilaires. ■

Ce théorème est fondamental pour la répartition. Il garantit en effet que si l'on projette un FA $\mathcal{F}$ sur un sous-ensemble de l'ensemble des modules, le produit synchronisé de cette projection avec la projection de $\mathcal{F}$ sur le reste des modules est bissimilaire avec l'automate de départ $\mathcal{F}$. Ceci nous permet donc par projections successives de répartir un FA $\mathcal{F}$ sur plusieurs sites, comme illustré à la figure 6.1, tout en conservant la bissimilarité avec à $\mathcal{F}$. En effet, par le théorème 3.3, nous savons que le produit synchronisé est congruent par rapport à la bissimulation. Dès lors, par le théorème 4.2, toute formule CTL satisfaite par $\mathcal{F}$, l'est aussi par le système réparti, résultant des produits synchronisés des automates projetés.

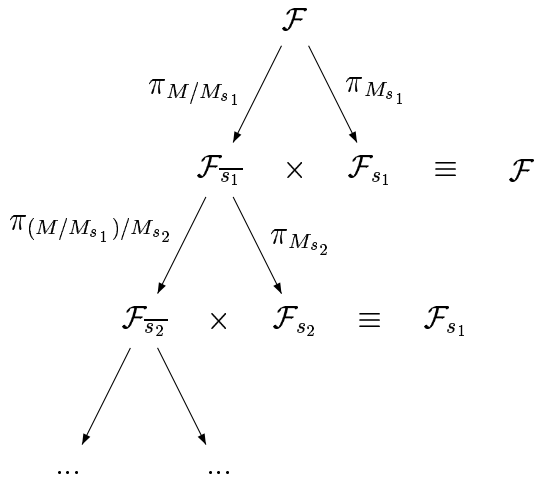
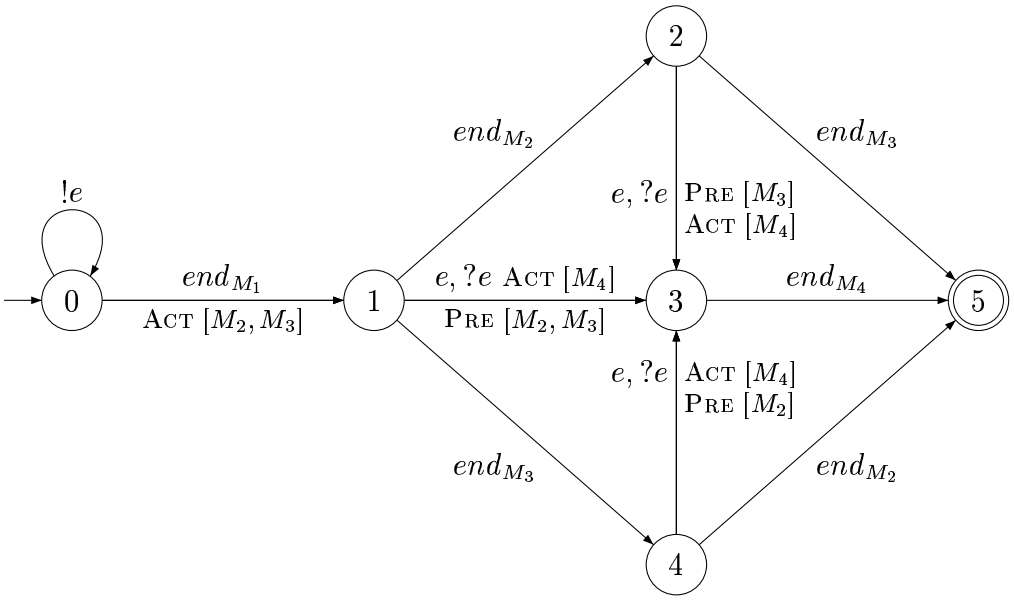


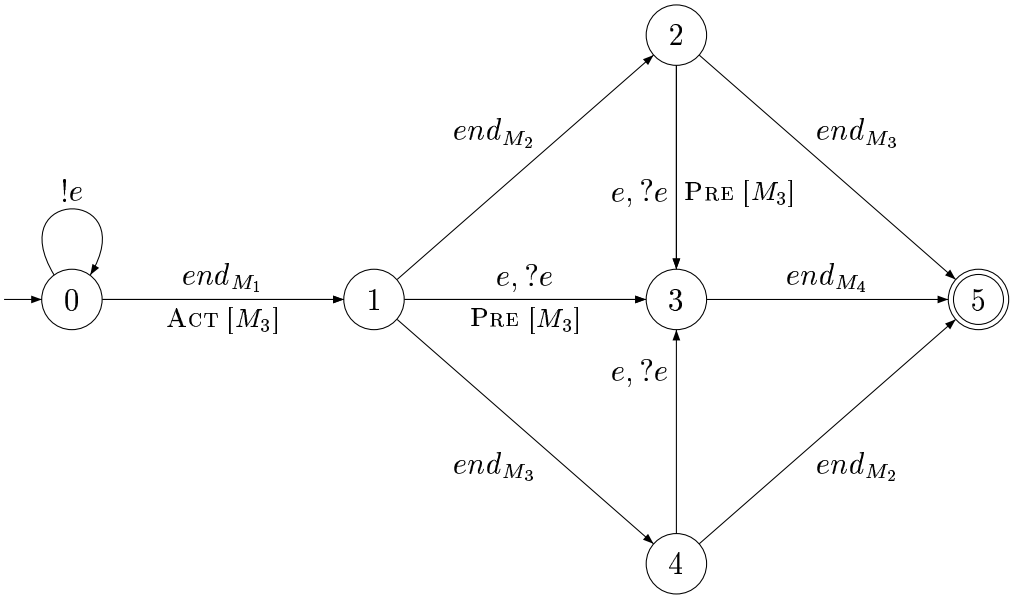
FIG. 6.1 – Méthode de répartition par projection

Exemple 6.1

Nous considérons ici le FIFFO-AUTOMATE $\mathcal{F}$ résultant de la compilation du programme ELECTRE [M_1;[M_2 || M_3] ↑ { e : M_4 }] illustré à la figure 6.2(a). Nous présentons ensuite, respectivement aux figures 6.2(b) et 6.2(c), les projections de ce FA sur $\{M_1, M_3\}$ ($\pi_{\{M_1,M_3\}}(\mathcal{F})$) et sur $\{M_2, M_4\}$ ($\pi_{\{M_2,M_4\}}(\mathcal{F})$). Remarquons que l'ensemble des modules actifs de $\pi_{\{M_1,M_3\}}(\mathcal{F})$ est le même que pour $\mathcal{F}$ ($A^0_{\pi_{\{M_1,M_3\}}(\mathcal{F})} = A^0_{\mathcal{F}}$), et que l'ensemble des modules actifs de $\pi_{\{M_2,M_4\}}(\mathcal{F})$ est vide ($A^0_{\pi_{\{M_1,M_3\}}(\mathcal{F})} = \phi$). Le théorème 6.1 nous assure que le produit synchronisé de $\pi_{\{M_1,M_3\}}(\mathcal{F})$ et $\pi_{\{M_2,M_4\}}(\mathcal{F})$ est bissimilaire à $\mathcal{F}$.

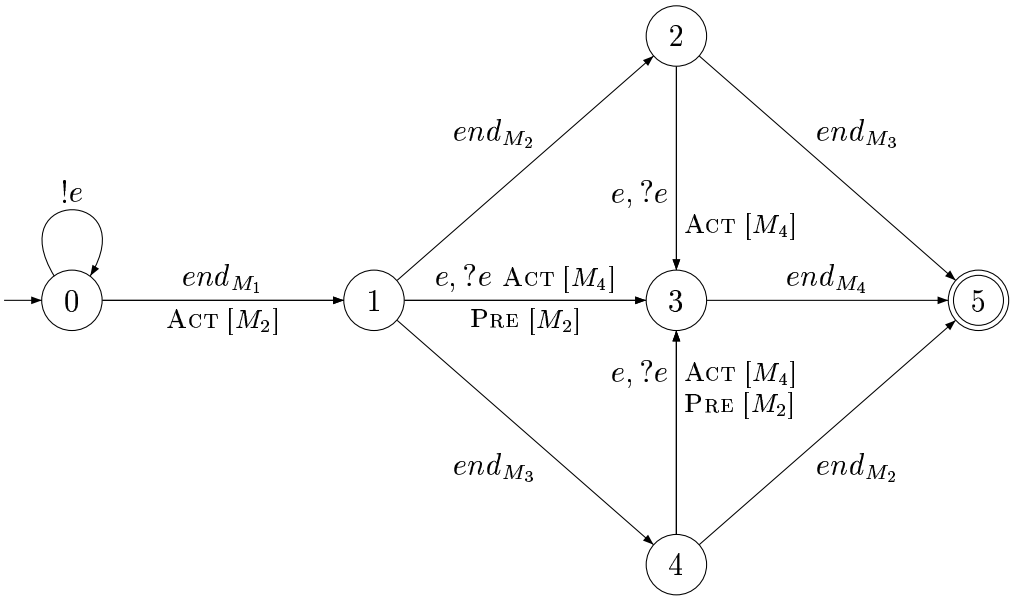


(a) FIFFO-AUTOMATE



(b) FIFFO-AUTOMATE projeté sur $\{M_1, M_3\}$

FIG. 6.2 – Exemple de projection - $[\ M_1; [\ M_2 \mid\mid M_3\ ] \uparrow \{\ e : M_4\ \}\ ]$



(c) FIFFO-AUTOMATE projeté sur $\{M_2, M_4\}$

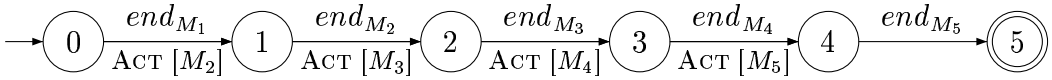
FIG. 6.2 – Exemple de projection (suite) - $[\ M_1; [\ M_2 \parallel M_3\ ] \uparrow \{\ e : M_4\ \}\ ]$

6.2 Réduction

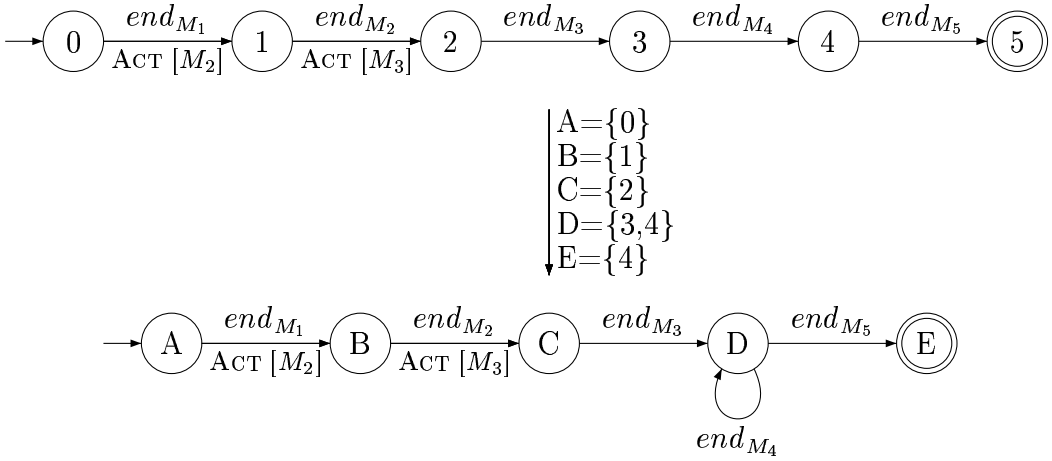
Bien que, par le théorème 6.1, la projection nous apporte une solution au problème de répartition, nous pouvons remarquer qu'en terme de nombre d'états et de transitions, les automates répartis sont identiques. Il est légitime de se demander s'il ne serait pas possible de réduire les automates projetés en regroupant des états qui, par la projection, sont devenus semblables du point de vue de leur comportement.

Exemple 6.2

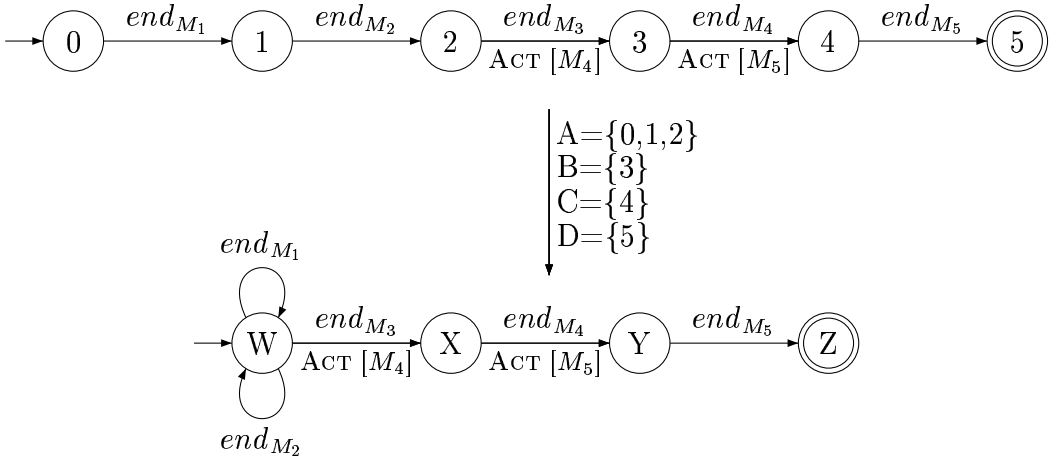
Afin d'illustrer nos propos, nous considérons ici l'exemple suivant : un FA $\mathcal{F}$ résultant de la compilation du programme ELECTRE $M_1; M_2; M_3; M_4; M_5$, présenté à la figure 6.3(a). Nous présentons ensuite les projections respectives de $\mathcal{F}$ sur $\{M_1, M_2, M_3\}$ et sur $\{M_4, M_5\}$, aux figures 6.3(b) et 6.3(c), ainsi que des regroupements d'états avec les automates ainsi produits. Nous terminons, à la figure 6.3(d) en présentant le produit synchronisé de ces FA. Nous pouvons observer que ce FA est bissimilaire à $\mathcal{F}$. En effectuant ce regroupement d'états, nous avons donc "gagné" deux états dans chaque FA réparti, tout en conservant la bissimulation.



(a) Fiffo-Automate de départ

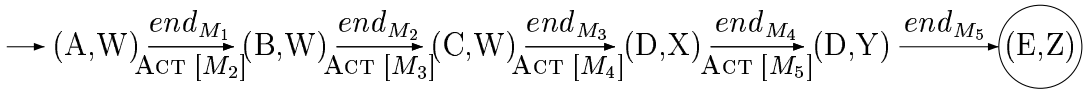


(b) Regroupement dans le Fiffo-Automate projeté sur $\{M_1, M_2, M_3\}$



(c) Regroupement dans le Fiffo-Automate projeté sur $\{M_4, M_5\}$

Fig. 6.3 – Regroupement d'états - $M_1; M_2; M_3; M_4; M_5$



(d) Produit synchronisé de (b) et (c)

FIG. 6.3 – Regroupement d'états - $M_1; M_2; M_3; M_4; M_5$

Définition 6.3 - Regroupement d'états (ρ)
Un *regroupement d'états* dans un FA $\mathcal{F} = (Q_\mathcal{F}, \Sigma_\mathcal{F}, \Delta_\mathcal{F}, q^0_\mathcal{F}, A^0_\mathcal{F}, F_\mathcal{F}, \text{ACT}_\mathcal{F}, \text{PRE}_\mathcal{F}, \text{REP}_\mathcal{F})$ (noté $\rho(\mathcal{F})$) est un FA défini par :

1. $Q_{\rho(\mathcal{F})} \subseteq 2^{Q_\mathcal{F}}$ est un partitionnement de l'ensemble d'états de départ $Q_\mathcal{F}$:
   (a) $\cup_{Q \in Q_{\rho(\mathcal{F})}} Q = Q_\mathcal{F}$
   (b) $\forall Q, Q' \in Q_{\rho(\mathcal{F})},\ Q \cap Q' = \phi$
2. $\Sigma_{\rho(\mathcal{F})} = \Sigma_\mathcal{F}$
3. $\Delta_{\rho(\mathcal{F})}$ est telle que $(Q, x, Q') \in \Delta_{\rho(\mathcal{F})}$ si et seulement si $\exists q \in Q$ et $q' \in Q'$ tels que $(q, x, q') \in \Delta_\mathcal{F}$.
4. $Q^0_{\rho(\mathcal{F})} = [q^0_\mathcal{F}]$ où $[q]$ désigne la partition $Q \in Q_{\rho(\mathcal{F})}$ contenant l'état q.
5. $A^0_{\rho(\mathcal{F})} = A^0_\mathcal{F}$
6. $F_{\rho(\mathcal{F})} = \{Q \in Q_{\rho(\mathcal{F})} | \forall q \in Q,\ q \in F_\mathcal{F}\}$
7. $\text{ACT}_{\rho(\mathcal{F})}(Q, x, Q') = \cup_{\{(q,x,q') \in \Delta_\mathcal{F} | q \in Q, q' \in Q'\}} \text{ACT}_\mathcal{F}(q, x, q'),\ \forall (Q, x, Q') \in \Delta_{\rho(\mathcal{F})}$.
8. $\text{PRE}_{\rho(\mathcal{F})}(Q, x, Q') = \cup_{\{(q,x,q') \in \Delta_\mathcal{F} | q \in Q, q' \in Q'\}} \text{PRE}_\mathcal{F}(q, x, q'),\ \forall (Q, x, Q') \in \Delta_{\rho(\mathcal{F})}$.
9. $\text{REP}_{\rho(\mathcal{F})}(Q, x, Q') = \cup_{\{(q,x,q') \in \Delta_\mathcal{F} | q \in Q, q' \in Q'\}} \text{REP}_\mathcal{F}(q, x, q'),\ \forall (Q, x, Q') \in \Delta_{\rho(\mathcal{F})}$.

Cette définition reste néanmois très générale. Ce qui nous intéresse, c'est de regrouper des états qui, par projection, sont devenus semblables du point de vue de leur comportement. Sur les figures 6.3(b) et 6.3(c), nous pouvons, par exemple, observer que

1. les modules actifs des états regroupés dans les automates projetés sont identiques :
   - dans l'automate $\pi_{\{M_1,M_2,M_3\}}(\mathcal{F})$ (figure 6.3(b)) $A(0) = \{M_1\}$, $A(1) = \{M_2\}$, $A(2) = \{M_3\}$ et $A(3) = A(4) = A(5) = \phi$.
   - dans l'automate $\pi_{\{M_4,M_5\}}(\mathcal{F})$ (figure 6.3(c)) $A(0) = A(1) = A(2) = \phi$, $A(3) = \{M_4\}$, $A(4) = \{M_5\}$ et $A(5) = \phi$.

2. les états finals et non-finals ne sont pas regroupés :
   - les états 3 et 4, dans l'automate $\pi_{\{M_1,M_2,M_3\}}(\mathcal{F})$ (figure 6.3(b)) ne sont pas regroupés avec l'état 5, bien qu'ils aient les mêmes modules actifs.
   - les états 0,1 et 2 dans l'automate $\pi_{\{M_4,M_5\}}(\mathcal{F})$ (figure 6.3(c)) ne sont pas regroupés avec l'état 5, bien qu'ils aient les mêmes modules actifs.

De plus, il est impératif que l'automate produit par regroupement soit déterministe. En effet, si ce n'était pas le cas, le produit synchronisé des automates répartis pourrait être non-déterministe. Ce produit synchronisé ne serait, dès lors, plus équivalent à l'automate de départ. Nous formalisons donc, ci-après, ces quelques propriétés par la notion de *réduction*.

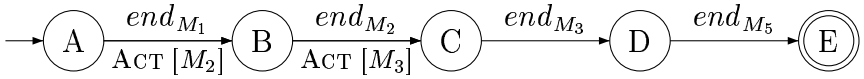
Définition 6.4 - Réduction

Une *réduction* est un regroupement d'états ρ satisfaisant :

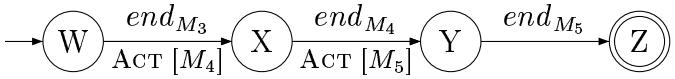
1. $\forall q \in Q^0_{\rho(\mathcal{F})}, A_{\mathcal{F}}(q) = A^0_{\mathcal{F}}$
2. $\forall (Q, x, Q') \in \Delta_{\rho(\mathcal{F})}, \forall q \in Q$, si $(q, x, q') \in \Delta_{\mathcal{F}}$, alors
   (a) $\textsc{Act}_{\rho(\mathcal{F})}(Q, x, Q') = \textsc{Act}_{\mathcal{F}}(q, x, q')$
   (b) $\textsc{Pre}_{\rho(\mathcal{F})}(Q, x, Q') = \textsc{Pre}_{\mathcal{F}}(q, x, q')$
   (c) $\textsc{Rep}_{\rho(\mathcal{F})}(Q, x, Q') = \textsc{Rep}_{\mathcal{F}}(q, x, q')$
3. $\forall (Q, x, Q') \in \Delta_{\rho(\mathcal{F})}, \forall q \in Q$, si $(q, x, q') \in \Delta_{\mathcal{F}}$, alors $q' \in Q'$
4. $\forall Q \in Q_{\rho(\mathcal{F})}, \forall q, q' \in Q$, $q \in F_{\mathcal{F}}$ si et seulement si $q' \in F_{\mathcal{F}}$

Les conditions (1) et (2) garantissent que deux états regroupés ont les mêmes modules actifs. La condition (3) assure le déterminisme de l'automate réduit et la condition (4), quant à elle, garantit qu'un état final ne soit regroupé avec un état non-final.

Remarquons qu'une fois cette réduction effectuée, certaines transitions n'ont plus lieu d'être. En effet, si partout dans l'automate, les transitions de prise en compte (directe ou différée) d'occurrence d'un événement e deviennent dégénérées, elles peuvent être supprimée, ainsi que les transitions de mémorisation de ce même événement e. Nous entendons, par transitions dégénérées, des transitions pour lesquelles l'état de départ est regroupé avec l'état d'arrivée, qui n'activent, ne préemptent, ni ne reprennent au début aucun module. En effet, par la sémentique des FIFFO-AUTOMATE, lors d'une occurrence d'événement, si aucune transition de mémorisation, ni de prise en compte n'existe pour cet événement, le comportement par défaut est de rester dans le même état, de ne pas activer, préempter ou reprendre au début des modules, et de ne pas changer le contenu de la file d'attente.



(a) FIFFO-AUTOMATE réduit de la figure 6.3(b)



(b) FIFFO-AUTOMATE réduit de la figure 6.3(c)

FIG. 6.4 – Suppression des transitions dégénérées

Exemple 6.3

Nous reprenons ici les automates de l'exemple 6.2. Les figures 6.4(a) et 6.4(b) présentent les automates réduits des figures 6.3(b) et 6.3(c) après suppression des transitions dégénérées.

L'intérêt de ces réductions est de diminuer le nombre d'états et de transitions des automates projetés de manière cohérente. Cependant, il faut garder à l'esprit le but qui nous anime ici : la répartition de FA. Autrement dit, il faut que les réductions des automates projetés fournissent une répartition correcte. Autrement dit il doit exister une certaine forme d'équivalence entre l'automate de départ et le produit synchronisé des automates obtenus après réduction des automates projetés. Dès lors il convient de rechercher un couple de réduction ρ_1, ρ_2 tel que $(\rho_1\pi_N(\mathcal{F}) \times \rho_2\pi_{M/N}(\mathcal{F})) \approx \mathcal{F}$, où $\approx$ est une forme d'équivalence (bissimilarité, équivalence de trace), minimisant un critère particulier, comme par exemple :

- minimiser la somme des nombres d'états des automates répartis.
- minimiser l'alphabet de synchronisation, c'est-à-dire, l'intersection des alphabets des automates répartis.
- ...

Nous présentons ci-après un théorème qui fournit un critère suffisant pour qu'un couple de réduction forme une répartition correcte au sens de l'équivalence de trace.

Théorème 6.2

Soit $\mathcal{F}$ un FA, $N \subseteq M$ un sous ensemble de l'ensemble d'états, et ρ_1, ρ_2 des réductions respectives de $\pi_N(\mathcal{F})$ et $\pi_{M/N}(\mathcal{F})$. Nous notons $\mathcal{F}_1 = \rho_1(\pi_N(\mathcal{F}))$ et $\mathcal{F}_2 = \rho_2(\pi_{M/N}(\mathcal{F}))$. Si les deux conditions suivantes sont vérifiées :

(C1) $\forall((Q_1, Q_2), x, (Q_1', Q_2')) \in \Delta_{\mathcal{F}_1 \times \mathcal{F}_2}$, $\forall q \in Q_1 \cap Q_2$,
$$(q, x, q') \in \Delta_{\mathcal{F}} \rightarrow q' \in Q_1' \cap Q_2'$$

(C2) $\forall Q_1 \in Q_{\mathcal{F}_1}, Q_2 \in Q_{\mathcal{F}_2}$, $\forall q \in Q_1 \cap Q_2$,
$$out_{\mathcal{F}_1}(Q_1) \cap out_{\mathcal{F}_2}(Q_2) = out_{\mathcal{F}}(q)$$

alors, si $\mathcal{L}_{\mathcal{F}}(q)$ dénote le langage accepté par $\mathcal{F}$ à partir de l'état q,

$$\forall Q_1 \in Q_{\mathcal{F}_1}, Q_2 \in Q_{\mathcal{F}_2}, q \in Q_1 \cap Q_2,\ \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2) = \mathcal{L}_{\mathcal{F}}(q)$$

Démonstration

Nous montrons, par induction sur la longueur du mot w, que

$$w \in \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2) \text{ si et seulement si } w \in \mathcal{L}_{\mathcal{F}}(q)$$

Etape initiale $|w| = 0$:

Soit $q \in Q_1 \cap Q_2$ (q existe car $Q_1 \cap Q_2 \neq \phi$)

($\Leftarrow$) Deux cas se présentent :

- soit $\mathcal{L}_{\mathcal{F}}(q) = \varepsilon$ si $q \in F_{\mathcal{F}}$: dans ce cas, $\forall q' \in Q_1, q'' \in Q_2$, $q' \in F_{\mathcal{F}}$ et $q'' \in Q_{\mathcal{F}}$, par la condition (4) de la définition 6.4 (ρ_1 et ρ_2 sont des réductions). Par conséquent, $(Q_1, Q_2) \in F_{\mathcal{F}_1 \times \mathcal{F}_2}$ et $\mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2) = \varepsilon$.
- soit $\mathcal{L}_{\mathcal{F}}(q) = \phi$ si $q \notin F_{\mathcal{F}}$: dans ce cas, $\forall q' \in Q_1, q'' \in Q_2$, $q' \notin F_{\mathcal{F}}$ et $q'' \notin Q_{\mathcal{F}}$, également par la condition (4) de la définition 6.4. Par conséquent, $(Q_1, Q_2) \notin F_{\mathcal{F}_1 \times \mathcal{F}_2}$ et $\mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2) = \phi$.

Dans les deux cas, si $w \in \mathcal{L}_{\mathcal{F}}(q))$, alors $w \in \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2)$.

($\Rightarrow$) Deux cas se présentent également :

- soit $\mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2) = \varepsilon$: dans ce cas, $\forall q' \in Q_1, q'' \in Q_2$, $q' \in F_{\mathcal{F}}$ et $q'' \in Q_{\mathcal{F}}$ car $(Q_1, Q_2) \in F_{\mathcal{F}_1 \times \mathcal{F}_2}$. Dès lors si $q \in Q_1 \cap Q_2$ alors $q \in F_{\mathcal{F}}$ et $\mathcal{L}_{\mathcal{F}}(q) = \varepsilon$.
- soit $\mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2) = \phi$: dans ce cas, $\forall q' \in Q_1, q'' \in Q_2$, $q' \notin F_{\mathcal{F}}$ et $q'' \notin Q_{\mathcal{F}}$ car $(Q_1, Q_2) \notin F_{\mathcal{F}_1 \times \mathcal{F}_2}$. Dès lors si $q \in Q_1 \cap Q_2$ alors $q \notin F_{\mathcal{F}}$ et $\mathcal{L}_{\mathcal{F}}(q) = \phi$.

Dans les deux cas, si $w \in \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2)$, alors $w \in \mathcal{L}_{\mathcal{F}}(q))$.

Etape inductive $|w| > 0$:

Nous faisons l'hypothèse que si $|w| = n$, $w \in \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2)$ si et seulement si $w \in \mathcal{L}_{\mathcal{F}}(q)$ et nous montrons que cela reste vrai pour $|w| = n+1$. Soit $q \in Q_1 \cap Q_2$ (q existe car $Q_1 \cap Q_2 \neq \phi$), et $w = xw'$.

($\Leftarrow$)
$(q, x, q') \in \Delta_{\mathcal{F}}$ (par hypothèse, $w \in \mathcal{L}_{\mathcal{F}}(q)$)
$\rightarrow x \in out_{\mathcal{F}}(q)$ (par définition de out)
$\rightarrow x \in out_{\mathcal{F}_1}(Q_1) \cap out_{\mathcal{F}_2}(Q_2)$ (par C2)
$\rightarrow (Q_1, x, Q'_1) \in \Delta_{\mathcal{F}_1}$ et $(Q_2, x, Q'_2) \in \Delta_{\mathcal{F}_2}$ (par définition de out)
$\rightarrow ((Q_1, Q_2), x, (Q'_1, Q'_2)) \in \Delta_{\mathcal{F}_1 \times \mathcal{F}_2}$ (par définition de $\times$)
$\rightarrow q' \in Q'_1 \cap Q'_2$ (par C1)

Dès lors, comme $w' \in \mathcal{L}_{\mathcal{F}}(q')$, par hypothèse d'induction, $w' \in \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_1}(Q'_1, Q'_2)$ car $|w'| = n$. Pour terminer, comme $((Q_1, Q_2), x, (Q'_1, Q'_2)) \in \Delta_{\mathcal{F}_1 \times \mathcal{F}_2}$, nous avons bien $w = xw' \in \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2)$.

($\Rightarrow$)
$((Q_1, Q_2), x, (Q'_1, Q'_2)) \in \Delta_{\mathcal{F}_1 \times \mathcal{F}_2}$ (par hypothèse, $w \in \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2)$)
$\rightarrow x \in out_{\mathcal{F}1 \times \mathcal{F}_2}(Q_1, Q_2)$ (par définition de out)
$\rightarrow x \in out_{\mathcal{F}_1}(Q_1) \cap out_{\mathcal{F}_2}(Q_2)$ (par définition de $\times$)
$\rightarrow x \in out_{\mathcal{F}}(q)$ (par C2)
$\rightarrow (q, x, q') \in \Delta_{\mathcal{F}}$ (par définition de out)
$\rightarrow q' \in Q'_1 \cap Q'_2$ (par C1)

Dès lors, comme $w' \in \mathcal{L}_{\mathcal{F}_1 \times \mathcal{F}_2}(Q_1, Q_2)$ et , par hypothèse d'induction, $w' \in \mathcal{L}_{\mathcal{F}}(q')$ car $|w'| = n$. Pour terminer, comme $(q, x, q') \in \Delta_{\mathcal{F}}$, nous avons bien $w = xw' \in \mathcal{L}_{\mathcal{F}}(q)$.

■

6.3 Cas particulier

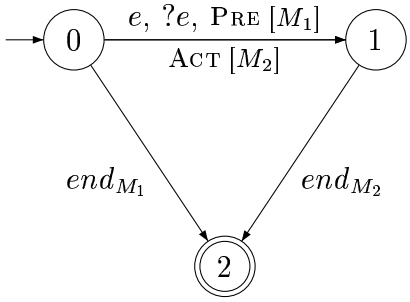
Dans les méthodes que nous avons abordées jusqu'à présent, nous nous sommes intéressés au FA provenant de la compilation de programmes ELECTRE en général. Il existe néanmoins des cas particuliers pour lesquel nous tendons à penser que la répartition puisse se faire au niveau sur le programme ELECTRE lui-même. Ce type de cas particulier se présente lors de l'utilisation de l'opérateur de parallèlisme (' ||'). En effet, nous pensons que lorsque les modules utilisés dans les deux branches de l'opérateur sont disjoints, si les deux branches sont compilées séparément en FA, le produit synchronisé de ces deux FA est bissimilaire au FA obtenu par compilation du programme tout entier.

Exemple 6.4

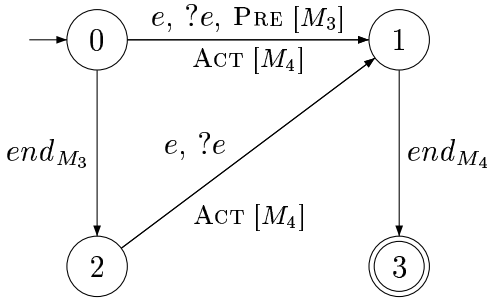
Afin d'illustrer notre intuition, nous considérons le programme ELECTRE suivant :

$$[\ M_1 \uparrow \{\ e : M_2\ \} \mid\mid M_3/\{\ e : M_4\ \}\ ]$$

Soit $\mathcal{F}$, le FA obtenu par compilation de ce programme, présenté à la figure 6.5(c), $\mathcal{F}_1$, le FA obtenu par compilation du sous-programme $[\ M_1 \uparrow \{\ e : M_2\ \}\ ]$, présenté à la figure 6.5(a) et $\mathcal{F}_2$ le FA obtenu par compilation du sous-programme $[\ M_3/\{\ e : M_4\ \}\ ]$, présenté à la figure 6.5(b). Nous pouvons remarquer que $\mathcal{F} \equiv \mathcal{F}_1 \times \mathcal{F}_2$. Pour s'en convaincre, à coté de chaque état de $\mathcal{F}$, nous avons précisé le couple d'états de $\mathcal{F}_1 \times \mathcal{F}_2$ bissimilaire.

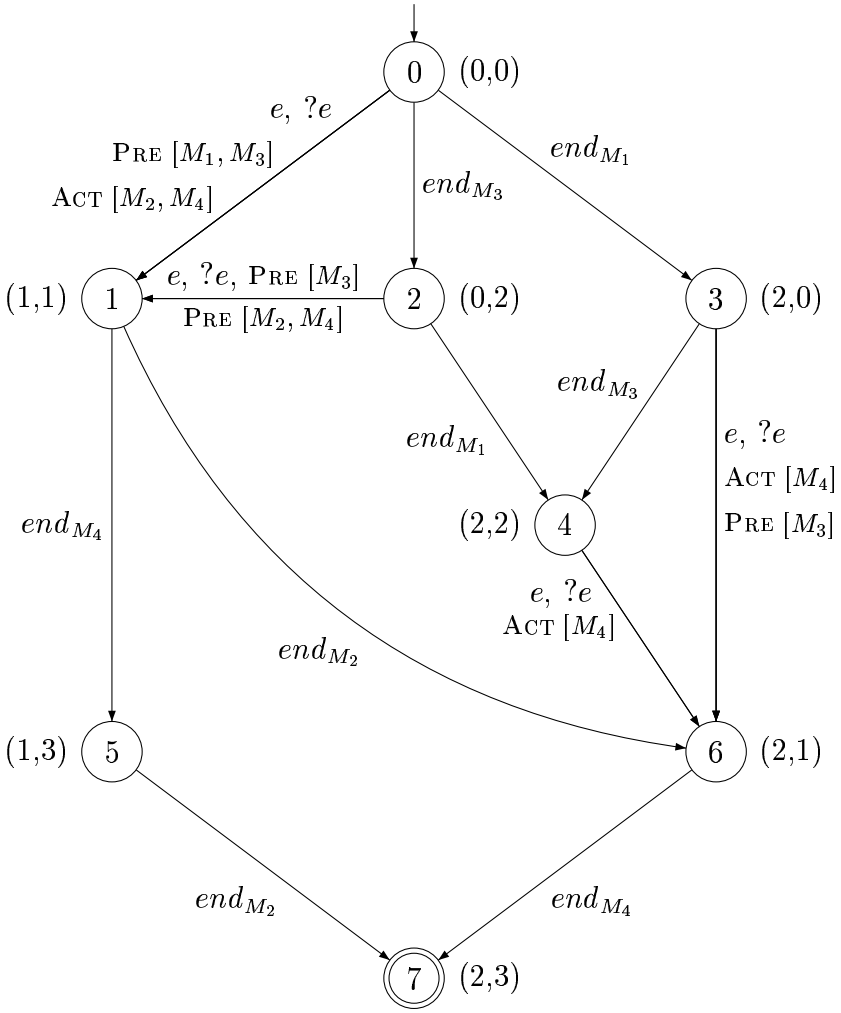


(a) $[\ M_1 \uparrow \{\ e : M_2\ \}\ ]$



(b) $[\ M_1 \uparrow \{\ e : M_2\ \}\ ]$

FIG. 6.5 – Répartition et parallélisme (suite)



(c) [$M_1 \uparrow \{\ e : M_2\ \} \parallel M_3/\{\ e : M_4\ \}$]

FIG. 6.5 – Répartition et parallélisme

Chapitre 7

Etude de cas

Aux chapitres 5 et 6, nous avons présenté des méthodes pour compiler et répartir un programme ELECTRE, en utilisant comme structure intermédiaire les FIFFO-AUTOMATES. Dans le présent chapitre, nous allons mettre en pratique tout ceci. Nous commençons, à la section 7.1, en présentant le problème que nous allons aborder. Ensuite, aux sections 7.2 et 7.3, nous présentons la modélisation du problème en programme ELECTRE. Nous terminons ensuite, à la section 7.4 en présentant une répartition.

7.1 Enoncé du problème

Le problème consiste à modéliser un contrôleur pour la gestion de deux transporteurs dans une usine. Ces transporteurs suivent un parcours cyclique reliant une aire de chargement à une aire de déchargement, comme illustré à la figure 7.1. Les transporteurs se déplacent dans le sens anti-horlogique sur leur circuit. Initialement les deux transporteurs se trouvent dans leur zone de déchargement. Les deux parcours se croisent en deux points où le contrôleur devra éviter les collisions. Pour ce faire, des zones de collision (A et B), autour de ces points de croisement, sont accessibles à un seul robot simultanément.

7.2 Modules et événements

La première étape dans cette étude de cas est la modélisation du problème en programme ELECTRE. Pour ce faire nous introduisons ci-après les modules et les événements que nous allons utiliser.

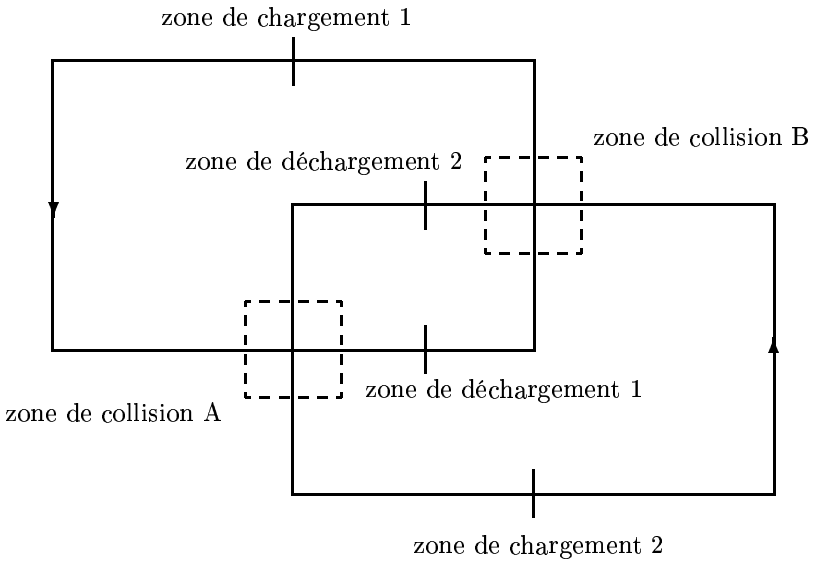


FIG. 7.1 – Schéma de parcours des deux transporteurs

Modules

1. $Load_1$, $Load_2$: modules qui modélisent le chargement des transporteurs.
2. $Unload_1$, $Unload_2$: modules qui modélisent le déchargement des transporteurs.
3. $Move_1$, $Move_2$: modules qui modélisent le déplacement des transporteurs sur leur circuit respectif.

Evénements

1. $loadRequest_1$, $loadRequest_2$: événements qui modélisent les demandes de chargement à l'intention des transporteurs.
2. $atLoadingBay_1$, $atLoadingBay_2$: événements qui modélisent l'arrivée des transporteurs à leur zone de chargement respective.
3. $atUnloadingBay_1$, $atUnloadingBay_2$: événements qui modélisent l'arrivée des transporteurs à leur zone de déchargement respective.
4. $atEntranceA_1$, $atEntranceA_2$, $atEntranceB_1$, $atEntranceB_2$: événements qui modélisent l'arrivée des transporteurs aux entrées des zones de collision, respectivement A et B.
5. $atExitA_1$, $atExitA_2$, $atExitB_1$, $atExitB_2$: événements qui modélisent l'arrivée des transporteurs aux sorties des zones de collision, respectivement A et B.

Remarquons que chaque demande de chargement doit être prise en compte. Par conséquent, les événements qui modélisent ces demandes seront à mémorisation multiple. Cependant, les événements restants représentent des informations éphémères (arrivée à un point particulier d'un circuit). Ils seront donc, quant à eux, de type fugace.

7.3 Programme ELECTRE

Le programme ELECTRE présenté à la figure 7.2 consiste en la mise en parallèle de deux structures de modules. Chacune de ces structures de modules s'occupe de la gestion d'un des deux transporteurs. Si nous examinons la première de ces deux structures (lignes 2 à 18), responsable du premier transporteur, nous pouvons remarquer qu'elle se décompose en trois parties, mises en séquence et reprises dans une structure bouclée. Ces trois parties sont les suivantes :

1. la première partie a pour but d'amener le premier transporteur à sa zone de chargement. Durant le trajet que ce transporteur doit emprunter pour y arriver, il sera amené à traverser la zone de collision B. Afin d'éviter une collision, le principe utilisé est le suivant : le premier des deux transporteurs qui arrive à l'entrée de la zone, aura la priorité. Dès lors si le premier transporteur arrive à l'entrée de la zone B ($@atEntranceB1$, ligne 5), il continue directement jusqu'à la zone de chargement (ligne 8). Si par contre, c'est le deuxième transporteur qui y rentre en premier ($@atEntranceB2$, ligne 7), le premier transporteur devra attendre que l'autre en sorte avant d'y rentrer, et continuer vers sa zone de chargement.
2. la deuxième partie, quant à elle, modélise l'attente d'une demande de chargement ($\#loadRequest1$, ligne 10), ainsi que le chargement qui s'ensuivra.
3. la troisième partie, pour terminer, a pour but de ramener le premier transporteur à sa zone de déchargement afin de se vider de son contenu. C'est cette fois la zone de collision A qui se trouve sur le trajet du transporteur. Le principe employé est similaire à celui utilisé dans la première partie. Si le premier transporteur arrive en premier ($@atEntranceA1$, ligne 13), il continue directement jusqu'à destination et se décharge (ligne 16). Par contre, si c'est le deuxième transporteur qui arrive à l'entrée de la zone A, le premier transporteur doit attendre qu'il en sorte avant d'y rentrer (ligne 15), et continuer vers sa zone de déchargement.

La deuxième moitié du programme (lignes 20 à 36) est responsable de la gestion du deuxième transporteur. Les principes, pour cette deuxième moitié de programme, sont identiques à ceux utilisés précédemment. Seules changent les zones de collision traversées

```
1  [
2      [
3          [
4              Move_1/{
5                  @atEntranceB_1
6              |
7                  { @atEntranceB_2 : [ [ Move_1/@atEntranceB_1 ]/@atExitB_2 ] }
8              } : Move_1/@atLoadingBay_1
9          ];
10         [ 1/{ #loadRequest_1 : Load_1 } ];
11         [
12             Move_1/{
13                 @atEntranceA_1
14             |
15                 { @atEntranceA_2 : [ [ Move_1/@atEntranceA_1 ]/@atExitA_2 ] }
16             } : Move_1/{ @atUnloadingBay_1 : Unload_1 }
17         ]
18     ]*
19 ||
20     [
21         [
22             Move_2/{
23                 { @atEntranceA_1 : [ [ Move_2/@atEntranceA_2 ]/@atExitA_1 ] }
24             |
25                 @atEntranceA_2
26             } : Move_2/@atLoadingBay_2
27         ];
28         [ 1/{ #loadRequest_2 : Load_2 } ];
29         [
30             Move_2/{
31                 { @atEntranceB_1 : [ [ Move_2/@atEntranceB_2 ]/@atExitB_1 ] }
32             |
33                 @atEntranceB_2
34             } : Move_2/{ @atUnloadingBay_2 : Unload_2 }
35         ]
36     ]*
37 ]
```

FIG. 7.2 – Etude de cas - programme ELECTRE

durant les déplacements du deuxième transporteur. En effet, c'est la zone A que le deuxième transporteur traverse pour se rendre à sa zone de chargement, ainsi que la zone B, pour se rendre à sa zone de chargement.

7.4 Répartition

Le programme présenté à la section précédente modélise un contrôleur. Ce contrôleur s'occupe de gérer le comportement des deux transporteurs. Afin de répartir ce programme, il est nécessaire de spécifier les différents sites d'exécution, ainsi que la répartition des modules à adopter. En ce qui concerne les sites d'exécution, il paraît naturel de modéliser chaque tansporteur par un site d'exécution SITE $= \{Tr_1, Tr_2\}$. La fonction de localisation de modules λ_M sera alors définie de la manière suivante :

$$\lambda_M(Load_1) = \lambda_M(Unload_1) = \lambda_M(Move_1) = Tr_1$$
$$\lambda_M(Load_2) = \lambda_M(Unload_2) = \lambda_M(Move_2) = Tr_2$$

Nous notons ensuite mF le FIFFO-AUTOMATE résultant de la compilation du programme ELECTRE de la figure 7.2. Remarquons que le FIFFO-AUTOMATE résultant de la compilation de ce programme est composé de 517 états et de 3395 transitions. La méthode de répartition par projection nous permet de calculer deux automates projétes $\mathcal{F}_1$ et $\mathcal{F}_2$. Ces deux automates sont les projections respectives de $\mathcal{F}$ sur M_{Tr_1} et M_{Tr_2} ([1]) :

$$\mathcal{F}_1 = \pi_{M_{Tr_1}}(\mathcal{F}) \text{ et } \mathcal{F}_2 = \pi_{M_{Tr_2}}(\mathcal{F})$$

Pour terminer, notons que $M_{Tr_1} \cup M_{Tr_2} = M$. Par conséquent, le théorème 6.1 nous assure que $\mathcal{F} \equiv \mathcal{F}_1 \times \mathcal{F}_2$.

[1] pour rappel M_{s_i} dénote l'ensemble des module localisé sur le site s_i

Conclusion

Le premier problème que nous avons été amenés à étudier était la compilation de programmes ELECTRE. Pour ce faire, nous avons étudié, au chapitre 3, les FIFFO-AUTOMATES qui permettent de modéliser des comportements de programmes ELECTRE. Ensuite, au chapitre 5, nous avons expliqué comment à partir de FIFFO-AUTOMATES, il était possible de construire un programme legOS qui en reflétait le comportement de manière fiable. Cette méthode de compilation a été implantée dans l'outil teflego. Cet outil fut testé avec succès sur plusieurs exemples, notamment, la réalisation d'un robot suiveur de ligne.

Le deuxième problème qui nous incombait était la répartition de ces mêmes programmes ELECTRE. Dès lors, nous avons développé et prouvé une méthode de répartition par projection qui offre une solution à ce problème. Cette méthode fut implantée dans l'outil distritef. A ce sujet, il serait intéressant d'étudier les problèmes de synchronisation entre les programmes provenant de la compilation des automates répartis. Ensuite, nous avons vu que les automates projetés sont loin d'être minimaux. Nous avons dès lors, introduit la notion de réduction afin d'essayer de remédier à ce problème. Cependant, afin de pouvoir implémenter de manière efficace cette réduction, il serait nécessaire de développer un algorithme pour la recherche d'un couple de réduction garantissant une répartition correcte, et ce suivant différents critères.
Toujours concernant la répartition, nous avions fait hypothèse de connaître la localisation des modules sur les différents sites d'exécution. Cette hypothèse, n'est dans le cadre des Lego Mindstorms, pas trop restrictive. Cependant, dans un cadre plus général, il serait bon d'examnier le problème de répartition sans cette hypothèse.

Pour terminer, nous tenons à signaler que ce stage de recherche fut, pour nous, une expérience très enrichissante, tant sur le plan personnel que sur le plan scientifique. C'est pour cette raison que nous tenons à remercier toutes les personnes qui ont rendu cela possible.

Annexe A

Programmation avec legOS

Nous présentons ici les principales fonctions qui permettent de contrôler la brique RCX en legOS. Celles-ci sont écrites en C. Remarquons qu'il est également possible de programmer la brique en C++. De plus amples informations sur la programmation en legOS peuvent être trouvées dans [?].

A.1 Entrées

Dans cette section, nous abordons les fonctions qui permettent, dans un programme legOS, de percevoir l'environnement de la brique RCX.

Capteur de lumière (dsensor.h)

1. LIGHT_X : variable contenant la valeur lue par le capteur de lumière connecté au port X = 1, 2 ou 3.
2. ds_passive(enum sensor) : fonction qui permet de mettre le capteur voulu en mode passif. Le paramètre sensor est de type énuméré qui peut prendre trois valeurs : SENSOR_1, SENSOR_2 ou SENSOR_3 suivant le port auquel est connecté le capteur.
3. ds_active(enum sensor) : fonction qui permet de mettre le capteur voulu en mode actif. Le paramètre sensor est un paramètre de type énuméré qui peut prendre trois valeurs : SENSOR_1, SENSOR_2 ou SENSOR_3 suivant le port auquel est connecté le capteur.

Les fonctions (2) et (3) permettent de contrôler le mode de fonctionnement du capteur. En mode actif, une petite lumière présente sur le capteur est allumée et permet d'éclairer

la zone dont il faut détecter l'intensité lumineuse. En mode passif, cette petite lumière est éteinte.

Capteur de touché (dsensor.h)

TOUCH_X : variable qui permet de détecter si le capteur connecté au port X (= 1, 2 ou 3) est enfoncé (TOUCH_X à la valeur 1) ou non (TOUCH_X à la valeur 0).

Bouton (dbutton.h et dkey.h)

1. PRESSED(button_state(), NAME) : macro qui permet de tester si le bouton NAME (= BUTTON_RUN ou BUTTON_VIEWED) est enfoncé.
2. RELEASED(button_state(), NAME) : macro qui permet de tester si le bouton NAME (= BUTTON_RUN ou BUTTON_VIEWED) est relaché.

Remarquons que seuls les boutons RUN et VIEW sont utilisables. En effet, les boutons ON/OFF et PROGRAM remplissent toujours le même rôle, à savoir allumer/éteindre la brique (ON/OFF) et selectionner le programme à exécuter ou de terminer le programme en cours (PROGRAM).

A.2 Sorties

Dans cette section, nous abordons les fonctions qui permettent, dans un programme legOS, d'agir sur l'environnement de la brique RCX.

Moteur (dmotor.h)

1. motor_X_dir(enum motor_dir) : fonction qui contrôle le sens de rotation du moteur connecté au port X (= a, b ou c). Le paramètre motor_dir est de type énuméré et peut prendre quatre valeurs possibles : fwd (en avant), rev (en arrière), off (à l'arrêt) et brake (freiner = les roues se bloquent).
2. motor_X_speed(int motor_speed) : fonction qui contrôle la vitesse de rotation du moteur X (= a, b ou c). Le paramètre motor_dir est de type entier et doit être compris entre MIN_SPEED (0) et MAX_SPEED (255).

Ecran LCD (dlcd.h)

1. lcd_int(int i) : fonction qui permet d'afficher un entier i sur l'écran LCD.
2. lcd_clear() : fonction qui permet d'effacer l'écran LCD.
3. cputs(char* s) : fonction qui permet d'afficher un string s de 5 caractères sur l'écran[1]
4. lcd_show(enum picto) et lcd_hide(enum picto) : fonctions qui permettent d'afficher et d'effacer les différents pictogrammes présents sur l'écran LCD (flèche, batterie,...).

A.3 Programmation distribuée

legOS offre quelques fonctionnalités pour la programmation distribuée : une gestion réduite du multi-tâche et de la programmation temps réel. De plus legOS, offre sous la forme de LNP, un protocole réseau qui permet de faire communiquer des briques RCX ou des terminaux ensemble, via leur port infra-rouge.

Multi-tâche (unistd.h)

1. execi(&function_name, int argc, char **argv, int prior, DEFAULT_STACK_SIZE) : fonction qui permet de lancer une tâche exécutant la fonction function_name). Les paramètres argc et argv contiennent le nombre et les paramètres de la fonction, prior la priorité. Cette fonction renvoie un entier qui représente le PID[2] et identifie la tâche univoquement.
2. kill(int pid) : fonction qui permet de tuer la tâche dont le PID est pid (renvoyé précédemment par un appel à execi()).

Programmation temps réel (unistd.h)

1. sleep(int s) et msleep(int ms) : fonctions qui permettent d'attendre respectivement un certain nombre de secondes s et de millisecondes ms.
2. wait_event(function_name, wakeup_t data) : fonction qui permet de suspendre l'exécution de la tâche qui y fait appel en attendant que function_name() renvoie une valeur non-nulle. Le paramètre data permet de passer des données à cette fonction.

[1] tout ce qui se trouvait avant l'écriture sur l'écran n'est pas effacé !
[2] process id

Lego Network Protocol (lnp.h et liblnp.h)

LNP est un protocole réseau qui permet à plusieurs briques RCX et terminaux de communiquer entre eux grâce à leur port infra-rouge. Le protocole est implémenté dans legOS et pourra être accédé au travers des fonctions de lnp.h sur la brique RCX. Cependant, un programme qui s'exécutera sur ce terminal pourra y accéder au travers des fonctions de liblnp.h. De plus, pour ce faire, il est nécessaire d'installer et de lancer *LNP Daemon* (lnpd) sur ce terminal. Ce démon s'occupe de la gestion du protocole. Ci-après, sont présentées les fonctions principales qui permettent de communiquer.

1. lnp_init(int tcp_host, int tcp_port, int lnp_address, int lnp_mask, int flags) : fonction qui devra être appelée pour initialiser le protocole. Pour chaque paramètre, la valeur 0 désigne la valeur par défaut. Les paramètres tcp_host et tcp_port sont à utiliser dans un programme qui s'exécutera sur un terminal pour se connecter au lnpd, via TCP. Les paramètres lnp_address et lnp_mask permettent de préciser l'adresse LNP du programme. Le paramètre flags permet de modifier le comportement de lnpd.
2. lnp_addressing_write(unsigned char *data, unsigned char length, int dest_addr, int port) : fonction qui permet d'envoyer des données sur le réseau. Les paramètres data et length permettent respectivement de spécifier les données à envoyer et leur longueur. Les paramètres dest_addr et port permettent respectivement de spécifier l'adresse de destination du paquet et le port sur lequel envoyer les données.
3. lnp_addressing_set_handler(int port, handler_function_name) : fonction qui permet d'enregistrer un gestionnaire de paquet. Lorsqu'un paquet sera reçu sur le port spécifié par le paramètre port, la fonction handler_function_name sera appelée pour le traiter. Cette fonction de gestion devra être déclarée par "void handler_function_name(unsigned char *data, unsigned char length, unsigned char src))".

Annexe B

Complément sur le langage ELECTRE

B.1 Grammaire du langage ELECTRE

Nous présentons ici la grammaire du langage ELECTRE. Les non-terminaux sont notés par les lettres majuscules P, C, R, G, I, K, J, E et M. Les terminaux sont représentés, pour les modules et les événements par *id* (= suite de caractère alphanumérique, commençant par une lettre) et par les opérateurs. Rappelons que les identifiants de modules commencent par une majuscule, alors que les identifiants d'événements commencent, eux, par une minuscule. La règle principale de la grammaire est celle définissant le non-terminal P.

$$
\begin{array}{lcl}
P & ::= & C \\
 & | & C \,||\, P \\
C & ::= & R \\
 & | & R; C \\
 & | & R/I \\
 & | & R \uparrow I \\
R & ::= & G \\
 & | & G* \\
G & ::= & M \\
 & | & [\ P\ ] \\
I & ::= & K \\
 & | & K : C \\
K & ::= & E \\
 & | & \{\ J\ \}
\end{array}
$$

$$
\begin{array}{lcl}
J & ::= & I \\
 & | & I \mid J \\
 & | & I \parallel J \\
 & | & I \mid\mid\mid J \\
E & ::= & id \\
 & | & @id \\
 & | & \#id \\
 & | & \$id \\
M & ::= & id \\
 & | & > id \\
 & | & !id \\
 & | & 1
\end{array}
$$

B.2 Sémantique opérationnelle d'ELECTRE

Les règles de réécriture du langage ELECTRE sont notées suivant le formalisme SOS introduit dans [**?**] qui s'interprète de la façon suivante :

$$\frac{\{e\} \vdash A_1 \xrightarrow{*} a_1 \rhd A'_1 \xrightarrow{*} a'_1 \wedge \cdots \wedge A_n \xrightarrow{*} a_n \rhd A'_n \xrightarrow{*} a'_n}{\{e\} \vdash B_1 \xrightarrow{*} b_1 \rhd B'_1 \xrightarrow{*} b'_1 \wedge \cdots \wedge B_k \xrightarrow{*} b_k \rhd B'_k \xrightarrow{*} b'_k}$$

"Dans le contexte de l'occurrence d'un événement e, et sachant que la dérivation $A_1 \xrightarrow{*} a_1$ se réécrit en $A'_1 \xrightarrow{*} a'_1$,..., et la dérivation $A_n \xrightarrow{*} a_n$ se réécrit en $A'_n \xrightarrow{*} a'_n$, nous déduisons que la dérivation $B_1 \xrightarrow{*} b_1$ se réécrit en $B'_1 \xrightarrow{*} b'_1$,..., et la dérivation $B_k \xrightarrow{*} b_k$ se réécrit en $B'_k \xrightarrow{*} b'_k$."

Une dérivation $A_1 \xrightarrow{*} a_1$ constitue simplement un chemin d'un non terminal A_1 de la grammaire considérée à un terminal a_1. Une dérivation est directe, et notée $A_1 \rightarrow a_1$ lorsqu'elle n'implique qu'une seule des règles de la grammaire. Dans notre cas, nous considérons les règles de la grammaire d'ELECTRE décrite dans la section B.1.

Règle (i) $P ::= C$

(i.1) $$\frac{\{e\} \vdash C \xrightarrow{*} m \rhd C' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash P \xrightarrow{*} m \rhd P' \xrightarrow{*} \texttt{nil}}$$

(i.2) $$\frac{\{e\} \vdash C \xrightarrow{*} m \rhd C' \xrightarrow{*} m'}{\{e\} \vdash P \xrightarrow{*} m \rhd P' \xrightarrow{*} m'}$$

Règle (ii) $P_1 ::= C \mid\mid P_2$

(ii.1) $$\frac{\{e\} \vdash C \xrightarrow{*} m \rhd C' \xrightarrow{*} m' \wedge P_2 \xrightarrow{*} p \rhd P_2' \xrightarrow{*} p'}{\{e\} \vdash P_1 \xrightarrow{*} m \mid\mid p \rhd P_1' \xrightarrow{*} m' \mid\mid p'}$$

(ii.2) $$\frac{\{e\} \vdash C \xrightarrow{*} m \rhd C' \xrightarrow{*} \texttt{nil} \wedge P_2 \xrightarrow{*} p \rhd P_2' \xrightarrow{*} p'}{\{e\} \vdash P_1 \xrightarrow{*} m \mid\mid p \rhd P_1' \xrightarrow{*} p'}$$

(ii.3) $$\frac{\{e\} \vdash C \xrightarrow{*} m \rhd C' \xrightarrow{*} m' \wedge P_2 \xrightarrow{*} p \rhd P_2' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash P_1 \xrightarrow{*} m \mid\mid p \rhd P_1' \xrightarrow{*} m'}$$

(ii.4) $$\frac{\{e\} \vdash C \xrightarrow{*} m \rhd C' \xrightarrow{*} \texttt{nil} \wedge P_2 \xrightarrow{*} p \rhd P_2' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash P_1 \xrightarrow{*} m \mid\mid p \rhd P_1' \xrightarrow{*} \texttt{nil}}$$

Règle (iii) $C ::= R$

(iii.1) $$\frac{\{e\} \vdash R \xrightarrow{*} r \rhd R' \xrightarrow{*} r'}{\{e\} \vdash C \xrightarrow{*} r \rhd C' \xrightarrow{*} r'}$$

(iii.2) $$\frac{\{e\} \vdash R \xrightarrow{*} r \rhd R' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash C \xrightarrow{*} r \rhd C' \xrightarrow{*} \texttt{nil}}$$

Règle (iv) $C_1 ::= R \ ; \ C_2$

Notons que dans ce cas, et conformément à la notion de séquentialité exprimée par l'opérateur $';'$, seule la transformation de R est possible.

(iv.1) $$\frac{\{e\} \vdash R \xrightarrow{*} r \rhd R' \xrightarrow{*} r' \wedge C_2 \xrightarrow{*} c \rhd C_2' \rightarrow c'}{\{e\} \vdash C_1 \xrightarrow{*} r \ ; \ c \rhd C_1' \xrightarrow{*} r' \ ; \ c}$$

(iv.2) $$\frac{\{e\} \vdash R \xrightarrow{*} r \rhd R' \xrightarrow{*} \texttt{nil} \wedge C_2 \xrightarrow{*} c \rhd C_2' \rightarrow c'}{\{e\} \vdash C_1 \xrightarrow{*} r \ ; \ c \rhd C_1' \xrightarrow{*} c'}$$

(iv.3) $$\frac{\{e\} \vdash R \xrightarrow{*} r \rhd R' \xrightarrow{*} r' \wedge C_2 \xrightarrow{*} c \rhd C_2' \rightarrow \texttt{nil}}{\{e\} \vdash C_1 \xrightarrow{*} r \ ; \ c \rhd C_1' \xrightarrow{*} r' \ ; \ c}$$

Règle (v) $C ::= R/I$

La transformation du non-terminal I en C dans la règle (v.4) s'explique par la règle (xiii.3)

(v.1) $$\frac{\{e\} \vdash R \xrightarrow{*} m \rhd R' \xrightarrow{*} m' \wedge I \xrightarrow{*} i \rhd I' \xrightarrow{*} i'}{\{e\} \vdash C \xrightarrow{*} m/i \rhd C' \xrightarrow{*} m'/i'}$$

(v.2) $$\frac{\{e\} \vdash R \xrightarrow{*} m \rhd R' \xrightarrow{*} \texttt{nil} \wedge I \xrightarrow{*} i \rhd I' \xrightarrow{*} i'}{\{e\} \vdash C \xrightarrow{*} m/i \rhd C' \xrightarrow{*} 1/i'}$$

(v.3) $$\frac{\{e\} \vdash R \xrightarrow{*} m \rhd R' \xrightarrow{*} m' \wedge I \xrightarrow{*} i \rhd I' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash C \xrightarrow{*} m/i \rhd C' \xrightarrow{*} \texttt{nil}}$$

(v.4) $$\frac{\{e\} \vdash R \xrightarrow{*} m \rhd R' \xrightarrow{*} m' \wedge I \xrightarrow{*} i \rhd C'' \xrightarrow{*} p'}{\{e\} \vdash C \xrightarrow{*} m/i \rhd C' \xrightarrow{*} p'}$$

Règle (vi) $C ::= R \uparrow I$

(vi.1) $$\frac{\{e\} \vdash R \xrightarrow{*} m \rhd R' \xrightarrow{*} m' \wedge I \xrightarrow{*} i \rhd I' \xrightarrow{*} i'}{\{e\} \vdash C \xrightarrow{*} m \uparrow i \rhd C' \xrightarrow{*} m' \uparrow i'}$$

(vi.2) $$\frac{\{e\} \vdash R \xrightarrow{*} m \rhd R' \xrightarrow{*} \texttt{nil} \wedge I \xrightarrow{*} i \rhd I' \xrightarrow{*} i'}{\{e\} \vdash C \xrightarrow{*} m \uparrow i \rhd C' \xrightarrow{*} \texttt{nil}}$$

La comparaison des règles (v.2) et (vi.2) met en valeur la différence entre les opérateurs / et $\uparrow$.

(vi.3) $$\frac{\{e\} \vdash R \xrightarrow{*} m \rhd R' \xrightarrow{*} m' \wedge I \xrightarrow{*} i \rhd I' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash C \xrightarrow{*} m \uparrow i \rhd C' \xrightarrow{*} \texttt{nil}}$$

(vi.4) $$\frac{\{e\} \vdash R \xrightarrow{*} m \rhd R' \xrightarrow{*} m' \wedge I \xrightarrow{*} i \rhd C'' \xrightarrow{*} p'}{\{e\} \vdash C \xrightarrow{*} m \uparrow i \rhd C'' \xrightarrow{*} p'}$$

Règle (vii) $R ::= G$

(vii.1) $$\frac{\{e\} \vdash G \xrightarrow{*} p \rhd G' \xrightarrow{*} p'}{\{e\} \vdash R \xrightarrow{*} p \rhd R' \xrightarrow{*} p'}$$

(vii.2) $$\frac{\{e\} \vdash G \xrightarrow{*} p \rhd G' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash R \xrightarrow{*} p \rhd R' \xrightarrow{*} \texttt{nil}}$$

Règle (viii) $R ::= G_1* \ ; \ G_2$

Lors de la compilation une structure répétitive de la forme $G*$ est réécrite sous la forme $G_1* \ ; \ G_2$ où G_1 est la réécriture de G obtenue lors de cette étape de compilation, et G_2 est égal à G. G_1 représente donc l'exécution courante de G et G_2, ses exécutions futures. Bien entendu, G_2 n'a pas la possibilité d'évoluer lors de ces réécritures.

$$\text{(viii.1)}\ \frac{\{e\} \vdash G_1 \xrightarrow{*} p_1 \rhd G_1' \xrightarrow{*} p_1' \ \wedge \ G_2 \xrightarrow{*} p_2 \rhd G_2' \xrightarrow{*} p_2}{\{e\} \vdash R \xrightarrow{*} p_1* \ ; \ p_2 \rhd R' \xrightarrow{*} p_1'* \ ; \ p_2}$$

$$\text{(viii.2)}\ \frac{\{e\} \vdash G_1 \xrightarrow{*} p_1 \rhd G_1' \xrightarrow{*} \texttt{nil} \ \wedge \ G_2 \xrightarrow{*} p_2 \rhd G_2' \xrightarrow{*} p_2}{\{e\} \vdash R \xrightarrow{*} p_1* \ ; \ p_2 \rhd R' \xrightarrow{*} p_2* \ ; \ p_2}$$

Règle (ix) $G ::= M$

$$\text{(ix.1)}\ \frac{\{e\} \vdash M \xrightarrow{*} m \rhd M' \xrightarrow{*} m'}{\{e\} \vdash G \xrightarrow{*} m \rhd G' \xrightarrow{*} m'}$$

$$\text{(ix.2)}\ \frac{\{e\} \vdash M \xrightarrow{*} m \rhd M' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash G \xrightarrow{*} m \rhd G' \xrightarrow{*} \texttt{nil}}$$

Règle (x) $G ::= [\ P\ ]$

$$\text{(x.1)}\ \frac{\{e\} \vdash P \xrightarrow{*} p \rhd P' \xrightarrow{*} p'}{\{e\} \vdash G \xrightarrow{*} [\ p\ ] \rhd G' \xrightarrow{*} [\ p'\ ]}$$

$$\text{(x.2)}\ \frac{\{e\} \vdash P \xrightarrow{*} p \rhd P' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash G \xrightarrow{*} [\ p\ ] \rhd G' \xrightarrow{*} \texttt{nil}}$$

Règle (xi) $I ::= K$

$$\text{(xi.1)}\ \frac{\{e\} \vdash K \xrightarrow{*} e \rhd K' \xrightarrow{*} e'}{\{e\} \vdash I \xrightarrow{*} e \rhd I' \xrightarrow{*} e'}$$

$$\text{(xi.2)}\ \frac{\{e\} \vdash K \xrightarrow{*} e \rhd K' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash I \xrightarrow{*} e \rhd I' \xrightarrow{*} \texttt{nil}}$$

Règle (xii) $I ::= K \ : \ C$

De même que pour les règles (iv.1) à (iv.3) concernant l'opérateur séquentiel ';', toute transformation du non terminal C ne peut se produire que lorsque le non terminal K a été réduit par la dérivation $K \xrightarrow{*} \texttt{nil}$.

$$\text{(xii.1)} \ \frac{\{e\} \vdash K \xrightarrow{*} e \rhd K' \xrightarrow{*} e' \ \wedge \ C \xrightarrow{*} c \rhd C' \xrightarrow{*} c'}{\{e\} \vdash I \xrightarrow{*} e \ : \ c \rhd I' \xrightarrow{*} e' \ : \ c'}$$

$$\text{(xii.2)} \ \frac{\{e\} \vdash K \xrightarrow{*} e \rhd K' \xrightarrow{*} e' \ \wedge \ C \xrightarrow{*} c \rhd C' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash I \xrightarrow{*} e \ : \ c \rhd I' \xrightarrow{*} e' \ : \ c'}$$

$$\text{(xii.3)} \ \frac{\{e\} \vdash K \xrightarrow{*} e \rhd K' \xrightarrow{*} \texttt{nil} \ \wedge \ C \xrightarrow{*} c \rhd C' \xrightarrow{*} c'}{\{e\} \vdash I \xrightarrow{*} e \ : \ c \rhd C'' \xrightarrow{*} c'}$$

Règle (xiii) $K ::= E$

$$\text{(xiii.1)} \ \frac{\{e\} \vdash E \xrightarrow{*} i \rhd E' \xrightarrow{*} i'}{\{e\} \vdash K \xrightarrow{*} i \rhd K' \xrightarrow{*} i'}$$

$$\text{(xiii.2)} \ \frac{\{e\} \vdash E \xrightarrow{*} i \rhd E' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash K \xrightarrow{*} i \rhd K' \xrightarrow{*} \texttt{nil}}$$

Règle (xiv) $K ::= \{ \ J \ \}$

$$\text{(xiv.1)} \ \frac{\{e\} \vdash J \xrightarrow{*} i \rhd J' \xrightarrow{*} i'}{\{e\} \vdash K \xrightarrow{*} \{ \ i \ \} \rhd K' \xrightarrow{*} \{ \ i' \ \}}$$

$$\text{(xiv.2)} \ \frac{\{e\} \vdash J \xrightarrow{*} i \rhd J' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash K \xrightarrow{*} \{ \ i \ \} \rhd K' \xrightarrow{*} \texttt{nil}}$$

Règle (xv) $J ::= I$

$$\text{(xv.1)} \ \frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i'}{\{e\} \vdash J \xrightarrow{*} i \rhd J' \xrightarrow{*} i'}$$

$$\text{(xv.2)} \ \frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} \texttt{nil}}{\{e\} \vdash J \xrightarrow{*} i \rhd J' \xrightarrow{*} \texttt{nil}}$$

$$\text{(xv.3)} \ \frac{\{e\} \vdash I \xrightarrow{*} i \rhd C \xrightarrow{*} c}{\{e\} \vdash J \xrightarrow{*} i \rhd C' \xrightarrow{*} c}$$

La réécriture de $J \xrightarrow{*} i$ en $C' \xrightarrow{*} c$ se produit lorsque I se tranforme en une structure de module sous l'effet de l'opérateur $'$:$'$ conformément à la régle (xiii.3).

Règle (xvi) $J_1 ::= I \mid J_2$

(xvi.1) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \mid j \rhd J'_1 \xrightarrow{*} i' \mid j'}$$

(xvi.2) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd C \xrightarrow{*} c \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \mid j \rhd C' \xrightarrow{*} c}$$

(xvi.3) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} \texttt{nil} \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \mid j \rhd J'_1 \xrightarrow{*} \texttt{nil}}$$

(xvi.4) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd C \xrightarrow{*} c}{\{e\} \vdash J_1 \xrightarrow{*} i \mid j \rhd C' \xrightarrow{*} c}$$

(xvi.5) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} \texttt{nil}}{\{e\} \vdash J_1 \xrightarrow{*} i \mid j \rhd J'_1 \xrightarrow{*} \texttt{nil}}$$

La comparaison de ces cinq règles avec les dix suivantes (correspondant à la composition parallèle forte pour les cinq premières et à la composition parallèle faible pour les suivantes) met en valeur la différence entre les compositions exclusive (ci-dessus) et parallèles (ci-dessous).

Règle (xvii) $J_1 ::= I \parallel J_2$

Malgré la surcharge de l'opérateur $' \parallel'$, il est aisé de distinguer les cas où il correspond à une composition d'événements de ceux où il sert à la composition de structures de programmes comme dans les règles (xvii.2) et (xvii.4).

(xvii.1) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \parallel j \rhd J'_1 \xrightarrow{*} i' \parallel j'}$$

(xvii.2) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd C \xrightarrow{*} c \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \parallel j \rhd C' \xrightarrow{*} c \parallel 1 \,/\, j'}$$

(xvii.3) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} \texttt{nil} \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \parallel j \rhd J'_1 \xrightarrow{*} j'}$$

(xvii.4) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd C \xrightarrow{*} c}{\{e\} \vdash J_1 \xrightarrow{*} i \;||\; j \rhd C' \xrightarrow{*} 1 \,/\, i' \;||\; c}$$

(xvii.5) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} \texttt{nil}}{\{e\} \vdash J_1 \xrightarrow{*} i \;||\; j \rhd J'_1 \xrightarrow{*} i'}$$

Règle (xviii) $J_1 ::= I \;|||\; J_2$

Dans les règles (xviii.3) et (xviii.5), il suffit que l'un des non terminaux I ou J (respectivement) s'annihile pour que la structure compositionnelle s'annihile elle aussi, ce qui constitue la différence avec la composition forte, où l'annihilation du second non terminal est requise pour l'annihilation de toute la structure.

(xviii.1) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \;|||\; j \rhd J'_1 \xrightarrow{*} i' \;|||\; j'}$$

(xviii.2) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd C \xrightarrow{*} c \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \;|||\; j \rhd C' \xrightarrow{*} c \;|||\; 1 \uparrow j'}$$

(xviii.3) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} \texttt{nil} \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} j'}{\{e\} \vdash J_1 \xrightarrow{*} i \;|||\; j \rhd J'_1 \xrightarrow{*} \texttt{nil}}$$

(xviii.4) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd C \xrightarrow{*} c}{\{e\} \vdash J_1 \xrightarrow{*} i \;|||\; j \rhd C' \xrightarrow{*} 1 \uparrow i' \;|||\; c}$$

(xviii.5) $$\frac{\{e\} \vdash I \xrightarrow{*} i \rhd I' \xrightarrow{*} i' \wedge J_2 \xrightarrow{*} j \rhd J'_2 \xrightarrow{*} \texttt{nil}}{\{e\} \vdash J_1 \xrightarrow{*} i \;|||\; j \rhd J'_1 \xrightarrow{*} \texttt{nil}}$$

Règle (xix) $E ::= id$

(xix.1) $\{e\} \vdash E \rightarrow e' \rhd E' \rightarrow \texttt{nil}$ si $e = e'$

(xix.2) $\{e\} \vdash E \rightarrow e' \rhd E' \rightarrow e'$ si $e \neq e'$

Règle (xx) $M ::= id$

(xx.1) $\{e\} \vdash M \rightarrow A \rhd M \rightarrow \texttt{nil}$ si $e = end_A$

(xx.2) $\{e\} \vdash M \rightarrow A \rhd M \rightarrow A$ si $e \neq end_A$

Les deux dernières règles déclenchent l'effet des occurrences d'événements sur les programmes ELECTRE en annihilant les événements qui surviennent ou les modules dont l'exécution s'achève, ce qui se répend par la suite dans tout le programme par l'intermédiaire des autres règles.

Bibliographie

[1] Pablo Argón. *Etude sur l'application de méthodes formelles à la compilation et à la validation de programme* ELECTRE. PhD thesis, Université de Nantes, Ecole Centrale de Nantes, 1998.

[2] Benoit Caillaud, Paul Caspi, Alain Girault, and Claude Jard. Distributing automata for asynchronous networks of processors. Technical Report RR-2341.

[3] Franck Cassez. *Compilation et vérification de programme* ELECTRE. PhD thesis, Université de Nantes, Ecole Centrale de Nantes, 1993.

[4] Franck Cassez and Olivier Roux. Compilation of the ELECTRE reactive language into finite transition systems. In *Theoretical Computer Science,* 146, 1995.

[5] Conrado Daws and Sergio Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95),* Pisa, Italy, pages 66–75, 1995.

[6] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics*, pages 995–1072. MIT Press, 1990.

[7] Frédéric Herbeteau. *Automates à file réactifs embarqués, application à la vérification de systèmes temps-réels.* PhD thesis, Université de Nantes, Ecole Centrale de Nantes, 2001.

[8] Dave Baum's NQC hompage. World Wide Web, `http://www.enteract.com/~dbaum/nqc/`.

[9] Edmund C. Clarcke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[10] legOS @ Sourceforge. World Wide Web, `http://legos.sourceforge.net/`.

[11] Markus L. Noga's legOS hompage. World Wide Web, `http://www.noga.de/legOS/`.

[12] lejOS @ Sourceforge. World Wide Web, `http://lejos.sourceforge.net/`.

[13] Lego Mindstorms official page. World Wide Web, `http://mindstorms.lego.com/`.

[14] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[15] Olivier Roux, Denis Creusot, Franck Cassez, and Jean-Pierre Elloy. Le langage réactif asynchrone ELECTRE. *Technique et Science Informatique (TDI)*, 11 :35–66, 1992.

[16] Luis Villa. *legOS HOWTO*, octobre 2000.

Index

P

R

S

T