Conception et réalisation d'un vérificateur de modèles AltaRica

Aymeric Vincent

N° d'ordre : 2761

THÈSE

présentée à

L'UNIVERSITÉ BORDEAUX 1

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

par Aymeric VINCENT

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : Informatique

Conception et réalisation d'un vérificateur de modèles AltaRica

Soutenue le : 5 décembre 2003

Après avis de :

MM.	Antoine RAUZY, Chargé de Recherches CNRS	**Rapporteurs**
	Philippe SCHNOEBELEN, Directeur de Recherches CNRS	

Devant la commission d'examen formée de :

MM.	André ARNOLD, Professeur	**Directeur de thèse**
	Robert CORI, Professeur	**Président**
	Alain GRIFFAULT, Maître de Conférences	**Rapporteur**
	Andreas PODELSKI, Professeur	**Examinateur**
	Antoine RAUZY, Chargé de Recherches CNRS	**Examinateur**
	Philippe SCHNOEBELEN, Directeur de Recherches CNRS	**Examinateur**

- 2003 -

Remerciements

Merci à Antoine Rauzy et Philippe Schnoebelen d'avoir accepté d'être rapporteurs de cette thèse, à Andreas Podelski d'être venu d'Allemagne pour participer à ce jury. Je remercie plus particulièrement Robert Cori pour avoir accepté d'en être le président et surtout pour son attention toujours bienveillante.

Merci à André Arnold pour m'avoir soutenu dans ce défi qu'était la conception et la réalisation d'un outil et surtout pour sa très grande disponibilité et toutes ses longues explications sur le μ-calcul, les jeux et les automates. Je voudrais remercier chaleureusement Alain Griffault pour nos grandes discussions sur AltaRica, ses excellents conseils dans bien des domaines, et aussi sa grande gentillesse qui a rendu la tâche moins pénible dans les moments difficiles.

Merci à l'équipe MVTsi pour son accueil amical, en particulier à Grégoire Sutre et Frédéric Herbreteau pour leurs conseils avisés et les discussions à bâtons rompus sur les systèmes d'exploitation.

Merci aux membres du LaBRI, en particulier Pierre-André Wacrenier, Philippe Narbel, Robert Strandh et Pierre Castéran, pour ces échanges d'idées qui justifient à mes yeux l'existence d'un laboratoire.

Merci à Nicolas Ollinger pour toutes les discussions éclairées sur le monde en général, et celui de l'informatique en particulier.

Merci aux banlieusards parisiens avec qui j'ai vécu une année de DEA très sympathique et pleine de poésie : Alexandre, Nicolas, Patricia, et Marie.

Merci à Afif, Pascal, Pascal, Pierre, Pierre, Sylvie, Yon, Yvan, pour l'enthousiasme qui a rendu ces trois années agréables, toujours prêts à organiser un groupe de travail à l'improviste autour de sujets aussi variés qu'éloquents, mais que la morale m'empêche de citer dans ce document.

Merci à Claire Pagetti, ma "*sœur AltaRica*", pour tous nos moments de complicité qui m'ont procuré autant de bonheur que de motivation pour ce travail.

Merci aux amis qui ont accepté de supporter ma complète asocialité pendant les derniers mois de cette thèse : Alexis, Romaric, Isabelle, Jean-Philippe, Béatrice, Nicolas, et Caroline qui sont venus me soutenir dans l'effort ultime. Merci !

Enfin merci à ma famille, pour tous les chemins qu'on a pris ensemble et qui m'ont fait passer par cette thèse.

Table des matières

Chapitre 1

Introduction

Le but de cette thèse a été de réaliser un vérificateur de modèles AltaRica.

Le formalisme AltaRica qui a été créé conjointement au LaBRI avec des industriels en l'an 2000 propose des mécanismes évolués de description de comportements par contraintes et de synchronisation qui permettent de modéliser aisément des systèmes complexes de manière hiérarchique.

Grâce à la satisfaction des industriels impliqués dans le projet et aux concepts robustes sur lesquels le formalisme est assis, le projet AltaRica a continué de prendre de l'ampleur. Bien que la vérification de modèles soit utilisée depuis plusieurs années avec succès dans le monde de la conception de circuits électroniques, ce n'est que très récemment qu'une demande réelle émanant des utilisateurs d'AltaRica a vu le jour pour disposer d'un vérificateur de modèles AltaRica.

L'équipe MVTsi du LaBRI qui a développé pendant de nombreuses années le vérificateur de modèles Mec a accumulé un grand savoir-faire dans ce domaine et il était donc naturel qu'elle propose la réalisation d'un successeur de Mec capable de vérifier non plus des modèles donnés directement sous forme de systèmes de transitions mais plutôt en langage AltaRica.

La philosophie qui a présidé à la conception de Mec V est la même que celle qui avait fait le succès de Mec : proposer à l'utilisateur un outil capable de faire tout ce que la technologie actuelle permet de manière efficace.

Mec V utilise dans son moteur de calcul des BDDs (Diagrammes de Décisions Binaires). Cette structure de données permet de représenter de manière concise des fonctions booléennes, et donc des relations sur des domaines finis. Les algorithmes efficaces sur les BDDs incluent les opérations booléennes classiques d'union, d'intersection, et de négation, mais aussi de projection et de cylindrification. La logique du premier ordre est donc naturellement implémentable grâce aux BDDs.

La logique que nous avons définie et implémentée dans Mec V permet d'écrire des systèmes d'équations avec points fixes de relations, où les termes utilisés dans les équations sont des termes du premier ordre. Le même choix avait été fait par Antoine Rauzy au LaBRI dans l'outil Toupie. Cette logique coïncide exactement avec le μ-calcul de Park et permet d'exprimer toutes les propriétés exprimables dans les logiques usuelles en vérification, comme CTL, LTL, ou CTL*, ainsi que de définir des relations entre plusieurs modèles, comme par exemple la relation de bisimulation.

Durant ces années de thèse, nous avons assisté à une forte demande de certains utilisateurs d'AltaRica de spécialiser le langage AltaRica afin de proposer un outillage plus complet sur des modèles restreints. Cette demande a été prise en compte par la définition du langage AltaRica

DataFlow, qui oriente les flux (entrée/sortie) alors que ceux-ci étaient jusque-là uniquement des variables partagées sans restriction d'utilisation.

L'orientation des flux nécessitait une modification de la syntaxe du langage, qui a été répercutée et généralisée dans le langage AltaRica sous la forme d'attributs que l'on peut associer aux variables ou aux événements. Cet épisode résume bien la volonté du projet AltaRica : proposer un langage ouvert et versatile, duquel on puisse dériver des sous-langages spécifiques à un domaine ou à un métier. Les attributs, ainsi que d'autres modifications apportées au langage, sont détaillés dans le chapitre 4. Le lecteur familier du langage AltaRica trouvera une liste de ces évolutions section 4.6 page 38.

Mec V a été réalisé entièrement en langage C, et le code source de la version actuelle fait approximativement 10 000 lignes. Une attention très particulière a été portée sur sa conception, et l'architecture de l'outil le rend très facilement maintenable et extensible.

La réalisation de cet outil a permis d'atteindre le but qui avait été fixé au début de la thèse : vérifier des modèles AltaRica de manière efficace. Mais Mec V peut aussi servir de base à la synthèse de contrôleurs comme nous le montrons dans le chapitre 6. Certains à-côté non négligeables nous ont fait voir que la conception et la réalisation d'un outil sont des entreprises très enrichissantes : nous avons eu des conversations passionnantes sur la conception de Mec V, et son architecture a été utilisée pour expérimenter la conception par foncteurs en Objective Caml. De plus, nous proposons actuellement des travaux dirigés de "technologies de la vérification" dans lesquels nous faisons étendre le code source de Mec V pour expérimenter de manière très pratique les algorithmes de base de la vérification.

Une extension temporisée du formalisme et du langage AltaRica est actuellement étudiée par Claire Pagetti [41, 42], qui a réalisé une implémentation basée sur le code de Mec V.

A côté de Mec V, nous avons regardé le problème de la synthèse de contrôleurs sur lequel beaucoup de recherches théoriques sont en cours. Nous présentons succinctement une méthode originale et très générique dans le chapitre 6, par laquelle il est possible de spécifier les objectifs de contrôle en μ-calcul modal, ainsi que les propriétés de contrôlabilité de l'environnement que doivent satisfaire le contrôleur. L'essentiel de la méthode repose sur la division d'une spécification par un processus qui permet de réduire le problème de la synthèse de contrôleurs à un problème de satisfiabilité d'une formule du μ-calcul. Ce dernier problème se résume à trouver une stratégie gagnante dans un jeu de parité, et une telle stratégie est calculable par Mec V.

Dans le chapitre 2, nous posons le problème de la vérification de modèles, et présentons de manière simple les concepts que l'on rencontre dans ce domaine.

Dans le chapitre 3, nous présentons la logique de spécification que nous avons implémenté dans Mec V ainsi que les BDDs, qui permettent de l'implémenter naturellement.

Dans le chapitre 4, nous décrivons le formalisme AltaRica ainsi que les extensions qui y ont été apportées pendant cette thèse.

Dans le chapitre 5, nous passons en revue différents aspects de l'implémentation de Mec V après avoir présenté rapidement son architecture.

Dans le chapitre 6, nous exposons le problème de la synthèse de contrôleurs et la méthode que nous proposons pour le résoudre.

Enfin, dans le chapitre 7, nous montrons des exemples d'utilisation de Mec V.

On trouvera en annexe la norme du langage AltaRica, un manuel de l'utilisateur succinct pour Mec V, et les grammaires du langage AltaRica et du langage de spécification de Mec V.

Chapitre 2

La vérification de modèles

Nous expliquons dans ce chapitre la problématique de la vérification. De très nombreux articles et ouvrages existent dans ce domaine, qui décrivent de manière formelle tous les concepts qui sont évoqués ici. Nous invitons le lecteur à se référer par exemple à [49] ou [16] pour des présentations détaillées. Nous avons essayé ici de familiariser le lecteur non pas avec l'histoire du domaine pour laquelle il existe de nombreux états de l'art, mais avec les *concepts* qui nous semblent importants.

Le problème de la vérification consiste à se demander, étant donné un modèle d'un système, si ce modèle *satisfait* certaines propriétés. Nous allons présenter simplement dans ce chapitre l'étape de modélisation qui aboutit au modèle d'un système réel, puis le formalisme classique des systèmes de transitions qui permet de représenter un modèle, puis nous montrerons les concepts qui sont spécifiques aux logiques employées en vérification pour spécifier des propriétés, et enfin nous citerons quelques outils de vérification afin de donner une idée de la position de Mec V parmi les nombreux vérificateurs de modèles existants.

2.1 La modélisation

Modéliser un fragment de réalité consiste à représenter formellement les aspects "intéressants" de ce fragment. C'est une tâche assez difficile dès que le processus est "complexe", mais c'est aussi une tâche très courante dans la vie de tous les jours.

2.1.1 Nécessité d'abstraire

La difficulté de la modélisation consiste souvent à trouver le bon niveau de détail, en fonction de l'information que l'on souhaite extraire du modèle (que ce soit pour la communiquer, ou pour vérifier qu'elle est cohérente). Le fait de rendre plus concis un modèle en élaguant des informations s'appelle l'*abstraction*.

Un exemple de la vie courante

Prenons l'exemple d'une personne qui souhaite donner à quelqu'un l'information suffisante qu'il faut pour aller d'un lieu A à un lieu B.

Il est possible pour cela par exemple de fournir une maquette d'une région contenant A et B. Le degré de détail de la maquette est variable, mais elle contiendra sans doute beaucoup d'informations et de points de repère, comme les routes, le type de culture des champs, les rivières.

C'est un modèle de la réalité qui peut être très fidèle et qui doit permettre de trouver un chemin entre A et B.

Il est aussi possible (et plus simple à l'heure actuelle) de profiter du travail de modélisation fait par les cartographes, et de fournir une carte contenant les positions de A et B. Cette carte aura sans doute perdu les panneaux de signalisation, la hauteur des fossés, par exemple. Cependant, les montées et les descentes auront été abstraites par des courbes de niveaux, la distinction entre les routes et les rivières se fera par la couleur des traits les représentant.

Notez que dans ces deux cas, la personne qui utilise le modèle a tout loisir de choisir la route qu'elle emprunte. Ce modèle n'impose aucun mouvement, en ce sens on peut dire qu'il est complètement *non-déterministe*, et même "statique".

A l'inverse, on peut imaginer de fournir une vidéo filmée lors d'un trajet de A à B. Cette vidéo fera sans doute moins d'abstractions que la carte, mais le chemin à emprunter sera beaucoup plus *déterminé*.

Enfin, il est possible de fournir une suite de routes et de changements de directions à suivre : un itinéraire. Il s'agit d'une représentation très abstraite de la réalité, ne contenant que les informations *utiles au but fixé*, et pré-déterminée afin de simplifier le travail d'exploitation du modèle.

Cet exemple montre juste le concept essentiel d'abstraction qui est le moyen principal de modélisation. Cependant, il n'est pas un exemple typique des modèles que nous rencontrons en vérification car ces descriptions sont statiques ou proposent un enchaînement linéaire et simple, alors que l'intérêt principal de la vérification réside dans la capacité à étudier les *comportements* complexes des modèles.

2.1.2 Transitions et états

Du fait qu'elle cherche à représenter les comportements, la modélisation mathématique de processus doit faire apparaître le concept de changement, qu'il soit un écoulement de temps, la survenue d'événements, ou la réalisation par le processus d'actions.

Cette notion de changement est capturée par une *transition*. Si l'on décide de choisir les transitions comme concept de base, on est amené à introduire la notion duale d'*état*.

En effet, considérons une porte sur laquelle deux actions sont possibles : `ouvrir` et `fermer`. On peut donc modéliser une personne utilisant la porte comme effectuant une suite d'actions finie parmi les suivantes :

`ouvrir fermer ouvrir fermer ouvrir fermer ouvrir`..., ou
`fermer ouvrir fermer ouvrir fermer ouvrir fermer`...

Cependant, si l'on désire modéliser le fait que la personne peut aussi passer cette porte (par l'action `passer`), les suites d'actions possibles deviennent :

`ouvrir` (`passer passer`...)? `fermer ouvrir` (`passer passer`...)? ..., ou
`fermer ouvrir` (`passer passer`...)? `fermer ouvrir` ...

Cette façon de représenter les suites d'actions possibles est capturée formellement par les *expressions régulières*, qui permettent de représenter des comportements linéaires comme ceux-ci de manière finie, en utilisant uniquement l'union $(a + b)$ et l'itération finie $(a*)$. Par exemple, l'expression régulière correspondant aux deux séquences ci-dessus pourrait être :
(`ouvrir passer`$*$ `fermer`)$*$ + (`fermer ouvrir passer`$*$)$*$

L'action `passer` est possible dans ces suites d'actions seulement directement après une action `ouvrir` ou une autre action `passer`. Cette description des endroits où il est possible de *passer*

la porte n'est pas simple. De plus, si l'on revient à la porte réelle, il est clair intuitivement que ce qui gouverne la possibilité de passer est l'*état* dans lequel se trouve la porte : est-elle fermée ou ouverte ?

On peut donc introduire dans notre modèle l'état de la porte, qui est soit *fermée* soit *ouverte*.

Et voici une nouvelle description du système, qui entre-mêle les états et les transitions :

fermée `ouvrir` *ouverte* (`passer` *ouverte* ...)? `fermer` ..., ou
ouverte (`passer` *ouverte* ...)? `fermer` *fermée* `ouvrir` ...

Ce que l'on va appeler *système de transitions* (section 2.2.1 page 7) est une représentation formelle de ces comportements, représentée par un graphe dont les sommets sont les états et dont les arcs sont les transitions. La figure 2.1 représente le système de transitions correspondant à la modélisation d'une porte et d'une personne pouvant l'ouvrir, la fermer ou passer la porte.

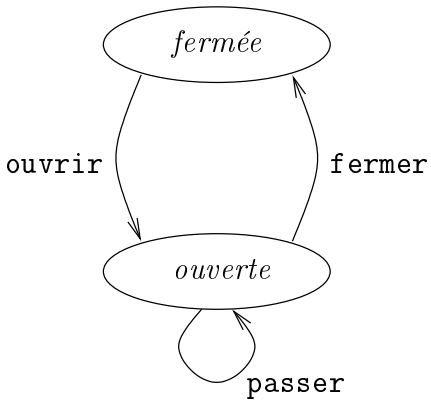


FIG. 2.1 – Système de transitions d'une porte et d'un passant

La représentation des comportements d'un système par un graphe est très pratique et classique, comme le montre la figure 2.2 page 6 qui donne une représentation graphique informelle du protocole TCP (*Transmission Control Protocol*) et qui est directement extraite du standard [47].

Un système de transitions contient en fait plus d'informations que les séquences d'états ou d'événements qui constituent ses *exécutions possibles*. Il est possible et parfois souhaitable d'utiliser la structure du graphe sous-jacent. Les deux systèmes de transitions de la figure 2.3 peuvent engendrer tous les deux les séquences de deux événements `payer café` et `payer thé`, mais dans le premier cas le choix de la boisson peut s'effectuer après le paiement alors que dans le second cas, le choix de l'une des deux branches `payer` impose *a priori* le choix de la boisson.

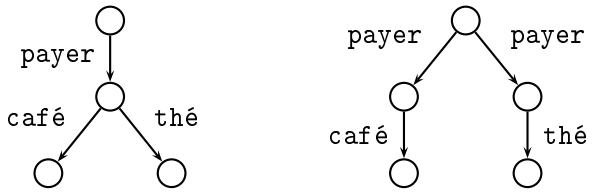


FIG. 2.3 – Comportements différents pour des mêmes séquences d'événements

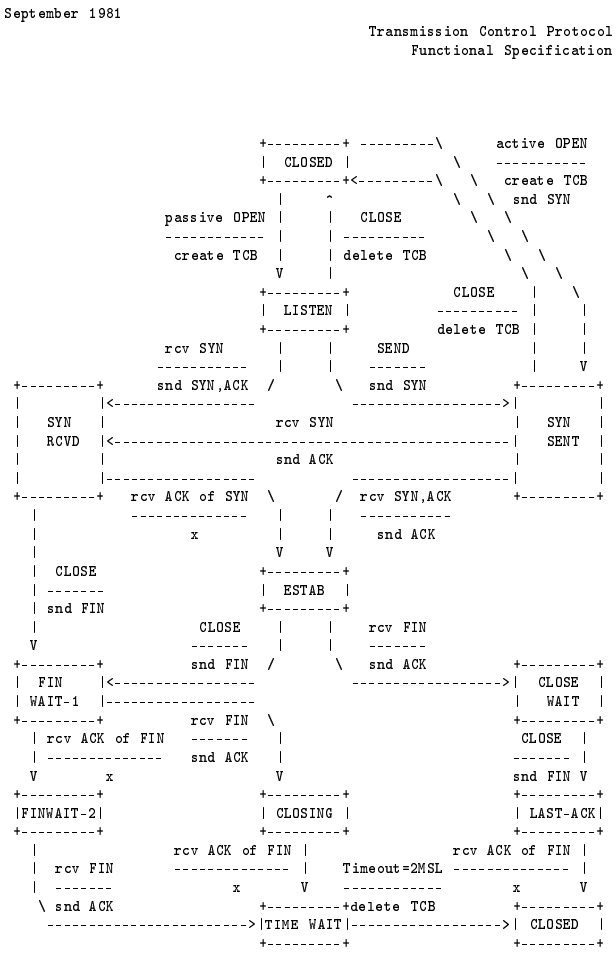


FIG. 2.2 – Comportement du protocole TCP

La notion de distingabilité des sommets du graphe (des états du système de transitions) permet de définir une notion de *bisimulation*[39]. Deux systèmes sont dits bisimilaires si à chaque état de l'un correspond un état de l'autre dans lequel les mêmes actions sont possibles et mènent à des états eux-mêmes bisimilaires. Par exemple, les deux systèmes de transitions de la figure 2.4 ne sont pas bisimilaires entre eux, mais chacun d'eux est bisimilaire à son pendant de la figure 2.3.

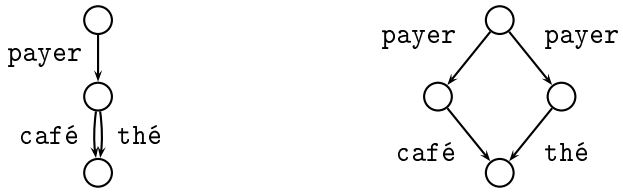


FIG. 2.4 – Deuxième exemple de non-bisimilarité

2.2 Des modèles formels

Nous présentons dans cette section les systèmes de transitions et quelques-unes de leurs variantes, qui capturent les notions que nous avons exhibées dans la section précédente.

2.2.1 Systèmes de transitions

On peut définir un système de transitions naturellement comme un ensemble d'états et de transitions, dont on précise quelles transitions peuvent avoir lieu dans quel état, et dans quel état elles mènent. Ceci s'écrit formellement comme ceci :

Définition 2.1 (système de transitions, 1ère) *Un système de transitions est un quadruplet* $\langle Q, T, \alpha : T \to Q, \beta : T \to Q \rangle$ *où* Q *et* T *sont des ensembles respectivement d'états et de transitions, et* α *et* β *sont deux applications précisant l'état source (resp. cible) d'une transition.*

Cette définition donne un rôle très dual aux états et aux transitions, et cette façon de voir un système de transitions est utilisée dans les précédentes versions de Mec [5] ou dans l'ouvrage [4].

On adjoint habituellement à ces systèmes de transitions des applications d'*étiquetage* des états et des transitions qui permettent de les regrouper par classes. Il est ainsi possible d'étiqueter tous les états d'un système qui satisfont une certaine propriété.

Comme une transition a au plus un état source et un état cible, la connaissance d'une transition identifie immédiatement l'endroit où elle se situe dans le graphe des comportements et rend donc en retour la spécification d'une transition difficile. Il est possible d'étiqueter les transitions qui se ressemblent, et c'est conceptuellement ce qui se passe lorsqu'on introduit la notion d'*événement*.

L'événement correspond à une *action* de la réalité, et le fait d'étiqueter une transition avec un événement nous permettra de pointer facilement les transitions qui réalisent cette action. La notion d'événement est tellement centrale dans la conception des modèles que la définition 2.1 de système de transitions que nous avons donnée est souvent étendue pour faire apparaître explicitement cette notion. On peut ajouter une fonction qui a une transition associe l'événement qui lui correspond.

Cependant, une fois que les événements ont été introduits, il devient possible d'*anonymiser* les transitions, que l'ont peut désormais caractériser chacune par un triplet – état de départ, événement, état d'arrivée. L'ensemble des transitions devient alors une relation ternaire sur les états, les événements et les états.

Cette façon d'envisager un système de transitions est exactement aussi expressive que la précédente et la traduction de l'un des deux formalismes à l'autre est immédiate. Ils correspondent

cependant à des schémas de pensée différents et se traduisent par des formalismes différents : voici la seconde définition de système de transitions.

Définition 2.2 (système de transitions, 2ème) *Un système de transitions est un triplet* $\langle Q, E, \Delta \subseteq Q \times E \times Q \rangle$ *où* Q *est un ensemble d'états,* E *est un ensemble d'événements, et* Δ *est la* relation de transition *du système de transitions.*

> Lorsque le système est *déterministe*, c'est-à-dire lorsque pour tout couple (q, e) il existe un unique état q' tel que $(q, e, q') \in \Delta$, on peut remplacer la relation de transition Δ par une *fonction* de transition δ qui à un tel (q, e) associe l'unique q' tel que $(q, e, q') \in \Delta$. Les processus que nous manipulerons dans le chapitre sur la synthèse de contrôleur sont de tels systèmes de transitions déterministes (définition 6.1 page 66).

Afin de réduire au minimum les concepts (qui se traduiront par des types) de base utilisés dans notre vérificateur de modèles, et puisque celui-ci sait manipuler naturellement des relations, c'est la deuxième définition qui représente le mieux la façon dont un modèle est perçu dans Mec V.

Le fait de ne plus avoir d'objet "transition" fait qu'il n'est plus possible d'étiqueter une transition par une fonction comme cela se faisait simplement dans le cas de la définition 2.1. Pour isoler des ensembles de transitions, il devient donc nécessaire d'adopter une approche "constructive", où l'on considère explicitement des sous-ensembles de triplets qui sont contenus dans la relation de transition.

> Cela se traduit de manière très nette dans les algorithmes de vérification (voir [38, 10]), car il est possible dans le premier cas de positionner un bit sur les transitions appartenant à un sous-ensemble pour les distinguer alors qu'il est nécessaire dans le deuxième cas de stocker séparément de la relation de transition ce sous-ensemble.

Les systèmes de transitions sont suffisamment simples et génériques pour être compris de tous et répondre à tous nos besoins d'expressivité. En revanche, il n'est pas facile de modéliser un système directement par un système de transitions puisqu'il faut énumérer explicitement les états dans lesquels le système peut se trouver. Or les systèmes qui nous intéressent comportent souvent au moins plusieurs milliers d'états, et d'autres formalismes sont plus adaptés et permettent de représenter de manière plus succincte les comportements d'un système. Lorsqu'on modélise des systèmes à espace d'états infini, il devient *nécessaire* d'employer un autre formalisme de modélisation.

Une manière très employée consiste à remplacer l'ensemble des états qui est trop abstrait par un ensemble de variables, dont le modèle va décrire les évolutions de leurs valuations.

Les systèmes de transitions sont un moyen très pratique pour donner la *sémantique* d'un modèle, et la plupart des formalismes sont définis par la donnée d'une fonction qui traduit un modèle de ce formalisme dans un système de transitions.

2.2.2 Systèmes à contraintes

L'idée qui sous-tend les systèmes à contraintes est de remplacer les états par les *valuations* de variables. On ne dira plus par exemple "l'état q_0", mais "l'état où *hauteur* vaut 50".

Il est très classique de modéliser le monde par des quantités numériques comme cela se fait en physique, et ces modèles par contraintes sont donc assez naturels à utiliser. On peut lire [24] pour une présentation très théorique et générale des automates à contraintes. Nous présentons les idées de ce formalisme qui ont été retenues dans AltaRica à la section 4.1, page 21.

On peut rapidement dire ici que le concept de transition est généralisé en *macro-transition*, qui décrit les états dans lesquels un événement donné peut se produire et la valeur que prendront les variables après la survenue de la macro-transition. Les ensembles d'états sont décrits par des contraintes sur les variables.

2.2.3 Systèmes de transitions étendus

Il existe une deuxième façon très proche de rendre plus agréable la modélisation de systèmes, et qui consiste à *étendre* un système de transitions avec des variables. La notion d'état abstrait reste, et les états du système de transitions représentant la sémantique d'un tel système sont des couples contenant l'état abstrait du système de transitions étendu *et* une valuation de ses variables.

Ce découplage des états et des valuations permet de distinguer dans le modèle l'aspect "*contrôle*" des aspects numériques. Ces systèmes de transitions étendus sont très naturels et pratiques lorsqu'il s'agit d'étendre les systèmes de transitions de manière minimale, seulement avec des variables ayant une *évolution propre*, et plus particulièrement continue comme des horloges.

Les systèmes de transitions étendus de cette manière par des horloges ont donné lieu au formalisme bien connu des automates temporisés [2].

2.2.4 Systèmes hiérarchiques

La *réutilisabilité* et la *modularité* des modèles sont deux concepts importants pour la modélisation de gros systèmes. Afin de proposer des moyens de lier les différents (sous-)modèles entre eux, la notion de *synchronisation* a été introduite. Il est ainsi possible de contraindre un événement d'un sous-modèle à avoir nécessairement lieu en même temps qu'un autre événement d'un autre sous-modèle. De la même manière, beaucoup de formalismes proposent de contraindre les valeurs des variables de sous-modèles différents.

Grâce à ces primitives de synchronisation, il est possible de modéliser des systèmes en plusieurs fois, en réutilisant des modèles plus petits. La plupart des formalismes utilisés par des outils proposent ces concepts, on peut citer par exemple [29, 14, 30]. Les principes de hiérarchie tels qu'ils sont traités dans le formalisme AltaRica sont présentés en détail à partir de la page 24.

2.3 Expression de propriétés

L'exploitation d'un modèle peut se faire de plusieurs manières, par exemple en simulant son exécution, en extrayant des arbres de défaillances, ou bien — et c'est ce qui nous intéresse ici — en exprimant des propriétés dont on souhaite savoir si le modèle les vérifie ou non.

Nous présentons ici la logique modale telle qu'elle a été introduite en philosophie, puis nous présentons la sémantique proposée par Saul Kripke [34], avant de revenir à la vérification de systèmes et aux logiques usuelles de la vérification.

2.3.1 Logiques modales

Comme les mathématiciens, les philosophes utilisent la logique pour formaliser leurs raisonnements et se prémunir des malentendus. La logique propositionnelle classique n'est pas assez expressive pour rendre compte de toute la complexité du monde. Par exemple, la proposition

"il pleut" n'a aucune valeur de vérité absolue, puisqu'en fonction du moment ou de l'endroit où cette proposition est énoncée ou considérée, elle s'avèrera fausse ou vraie.

De la même manière que l'on trouve dans le langage des modalités (*pouvoir*, *devoir*, ...), il est possible d'introduire dans la logique propositionnelle des modalités. Originellement, les deux modalités duales qui constituent la logique modale sont la *possibilité* et la *nécessité*. Il devient possible grâce à ces modalités d'énoncer des vérités, en adjoignant à une proposition une modalité qui en décrit l'extension. Ainsi par exemple, la proposition modale "il est possible qu'il pleuve" est toujours vraie, et la proposition "il est nécessaire qu'il pleuve" est toujours fausse.

Beaucoup d'autres modalités ont été introduites et utilisées par les philosophes pour constituer des logiques adaptées à leur discours. On peut citer les logiques déontiques basées sur les modalités duales d'obligation et d'interdiction, les logiques temporelles basées sur les modalités "il sera toujours vrai que" et "il sera un jour vrai que". Des logiques de la connaissance ("Untel croit que", "Untel sait que", ...) sont utilisées pour modéliser des bases de données [12] afin de rendre explicite le degré de véracité ou de confiance des réponses aux requêtes.

Indépendamment de la sémantique des modalités, qui sont introduites pour exprimer des propriétés sur le monde réel, les systèmes d'axiomes qui régissent ces différentes logiques partagent les mêmes structures. Ces systèmes d'axiomes sont répertoriés en fonction des logiques modales qu'ils permettent de représenter correctement.

Par exemple, pour la logique modale de la nécessité, l'axiome "nécessairement p implique p" est souhaitable, alors que dans le cas de la logique déontique par exemple, le monde réel ne peut pas être modélisé si on ajoute l'axiome "obligatoirement p implique p" puisqu'il est possible d'enfreindre une obligation. Le lecteur intéressé par ces différents systèmes d'axiomes et les noms qui leur sont donnés pourra consulter [26].

Les structures de Kripke

Le problème posé par ces logiques est qu'il était difficile étant donnée une formule comportant des modalités, d'en définir la sémantique. L'idée originale que l'on trouve dans [34] est de considérer un ensemble de "mondes possibles". La véracité des propositions seront alors liées à un "monde" : une même proposition peut être vraie dans un monde et fausse dans un autre. Par exemple, "il pleut" peut être vraie à un endroit et un instant donnés et fausse à un autre.

Une structure de Kripke est la donnée d'un ensemble de "mondes possibles" et d'une relation binaire les liant entre eux. Chacun des mondes définit quelles propriétés élémentaires sont vraies dans ce monde.

Ces structures ont permis de définir proprement la sémantique des logiques modales, et ont permis de manière très efficace de classer les différents systèmes d'axiomes en fonction des propriétés de la relation binaire qui lie les mondes : il existe un lien très fort entre les axiomes retenus dans la logique modale et les propriétés de la relation binaire entre mondes.

Bien que les motivations et le cheminement de pensée aient été très différents, on ne peut ici que constater le lien extrêmement étroit qui existe conceptuellement entre une structure de Kripke et un système de transitions. Ce lien se retrouve dans l'utilisation en vérification de "logiques modales".

Les logiques modales sur les systèmes de transitions

Il s'agit pour nous de renverser l'objet d'étude, puisque nous cherchons à vérifier des propriétés *d'un système de transitions* et non pas utiliser la structure des axiomes de la logique

pour prouver des propriétés logiques. C'est donc le système de transitions qui va imposer les modalités que l'on se donne.

Les deux modalités classiques [33] sont "il existe un successeur immédiat tel que" et "pour tous les successeurs immédiats". Ces deux modalités sont elles aussi duales. Il est possible de définir la sémantique de ces modalités à l'aide de formules du premier ordre lorsque l'on se donne la relation de transition du système de transitions. La logique utilisée dans Mec V ne fait pas apparaître de modalités puisqu'elle permet d'exprimer directement des propriétés du premier ordre.

Extensions par points fixes

Etant donnée une formule de la logique propositionnelle classique étendue par des modalités, il n'est possible de spécifier dans cette logique que des propriétés qui ont une profondeur d'exploration du graphe bornée par la taille de la formule : on pourra par exemple exprimer "dans trois jours il pleuvra", mais on ne pourra pas exprimer une propriété comme "un jour (non précisé) il pleuvra". Pour s'autoriser ce genre de spécification, on ajoute dans la logique des opérateurs de points fixes, qui permettent de calculer des ensembles d'états par induction, et donc de spécifier des propriétés beaucoup plus génériques. Ces logiques étendues par points fixes ont fait l'objet de très nombreuses études, et on peut trouver dans [7] un recueil des résultats et idées importants du domaine. Mec V utilise l'une des variantes les plus expressives de ces logiques (cf. chapitre 3).

2.3.2 Logiques temporelles

Bien que l'on puisse classer les logiques temporelles dans les logiques modales d'un point de vue technique, la distinction est toujours faite en informatique entre ces deux sortes de logiques. En effet, comme nous l'avons vu, les logiques modales se résument en fait à étendre une logique donnée avec deux modalités permettant de parcourir le système de transitions. Les logiques temporelles sont elles plus ciblées et offrent directement dans leurs opérateurs la possibilité de parler d'écoulement du temps[44, 15].

Les logiques modales s'intéressent à la structure du système de transitions, alors que les logiques temporelles abstraient la structure du système pour proposer des constructions logiques plus intuitives sur par exemple une exécution donnée du système.

Les opérateurs que l'on trouve en logique temporelle sont par exemple "au prochain coup", "un jour", "à partir de maintenant", "ceci sera vrai jusqu'à ce que". Le vocabulaire de temps employé fait qu'il n'est pas souhaitable de définir la sémantique de formules d'une logique temporelle sur un système de transitions. La sémantique est dans ce cas définie sur une *exécution* du système, puisque l'enchaînement des actions du système induit naturellement une notion de temps.

Comme nous l'avons vu en introduction, certains systèmes de transitions non-bisimilaires ont les mêmes séquences d'exécution. Afin de distinguer de tels systèmes en conservant l'intuition des logiques temporelles, il existe des logiques temporelles *arborescentes* qui considèrent les *arbres* d'exécution de systèmes et non plus seulement les *séquences* d'exécution.

Ces logiques très employées en vérification peuvent être traduites en μ-calcul modal [20] et donc à plus forte raison dans la logique de Mec V.

2.3.3 Quelques outils de vérification

L'objectif de cette thèse était d'écrire un vérificateur de modèles, c'est-à-dire un logiciel capable, étant donné un modèle et une propriété, de répondre si le modèle satisfait ou non la propriété.

De nombreux outils existent, et diffèrent en plusieurs points, notamment sur le langage d'entrée utilisé pour décrire les modèles et les propriétés, et aussi sur les structures de données et algorithmes employés pour effectuer les calculs.

On peut citer par exemple l'outil qui a précédé celui qui fait l'objet de cette thèse : MEC [5], qui charge des modèles spécifiés directement par des systèmes de transitions étiquetés et permet d'utiliser comme langage de spécification le μ-calcul sans alternance de points fixes, logique qui englobe CTL.

L'outil SMV [38] se distingue par le fait qu'il a été le premier outil de vérification à utiliser des structures de données "symboliques" [10] pour implémenter ses algorithmes de calcul : les états qui composent le système de transitions sous-jacent au modèle ne sont pas stockés explicitement en mémoire de manière à représenter le graphe correspondant, mais c'est la relation de transition de ce graphe qui est stockée de manière efficace. Mec V est très proche de SMV en ce qui concerne l'utilisation de ces structures de données, puisque comme SMV, Mec V utilise des BDDs (section 3.5, page 16). SMV propose un langage d'entrée pour les modèles qui lui est spécifique, et qui permet de représenter des processus ; les propriétés à vérifier sont spécifiées directement dans la logique temporelle CTL.

La tendance actuelle dans le milieu académique est plutôt de proposer des ateliers de vérification, ou des vérificateurs de modèles "ouverts", c'est-à-dire dont le but est de permettre une intégration facile de formalismes d'entrée divers et/ou d'expérimenter ou d'utiliser des algorithmes de vérification différents.

On peut citer parmi les pionniers de cette approche le Concurrency Workbench [17], qui accepte principalement en entrée des modèles écrits dans une algèbre de processus (CCS) et des propriétés écrites en μ-calcul modal.

Plus récemment, une réécriture complète de SMV a vu le jour : NuSMV [13, 14] qui offre plusieurs algorithmes de vérification différents pour des logiques différentes, notamment CTL et LTL. Le format des modèles d'entrée qui est proposé est celui de SMV.

Mec V se distingue donc de ces outils par la logique de spécification très expressive qu'il propose (chapitre 3) : la logique du premier ordre étendue par points fixes sur des *relations*. Les modèles d'entrée qu'il accepte sont des modèles AltaRica (chapitre 4). Nous avons aussi architecturé notre vérificateur de modèles de telle manière qu'il soit possible d'utiliser d'autres formalismes d'entrée pour les modèles, mais ce n'est qu'une conséquence de son architecture élégante (chapitre 5).

Chapitre 3

La logique de spécification de Mec V

Nous présentons dans ce chapitre la logique utilisée pour spécifier des propriétés dans Mec V. Il s'agit du μ-calcul proposé par Park [43] en 1976, dans lequel on peut définir des *relations* d'arité quelconque par points fixes. Les termes utilisés dans la définition des relations sont les termes du premier ordre.

Le choix de cette logique est justifié par deux raisons : une première d'ordre technique est que cette logique est en adéquation avec ce que les BDDs (section 3.5) permettent de calculer (opérations booléennes de base, projection, cylindrification). La deuxième raison est que toutes les logiques modales usuelles (CTL*, et donc LTL et CTL) sont moins expressives que le μ-calcul modal [20], et donc à plus forte raison sont moins expressives que le μ-calcul de Park. L'extension ECTL* de CTL* est équivalente à un fragment gardé du *mu*-calcul modal [22].

L'optique choisie de proposer un langage qui permette d'exploiter la puissance de diagrammes de décisions et qui soit en même temps dérivé d'une logique très étudiée auparavant (le μ-calcul), avait déjà été explorée par Antoine Rauzy et Marc-Michel Corsini au travers de l'outil Toupie [19, 18] ainsi que dans l'article célèbre [10].

Le μ-calcul fait l'objet de beaucoup de recherche, mais il est reconnu que certaines formules du μ-calcul sont difficilement compréhensibles. Mec V se veut un outil pratique et efficace d'expérimentation dans ce domaine, qui devrait permettre à un utilisateur de se familiariser avec cette logique.

D'autre part, l'outil de vérification développé et utilisé historiquement à Bordeaux jusque-là était Mec [5] dont les spécifications s'écrivent dans une logique très proche du μ-calcul sans alternance. Bien que Mec V soit un moteur de (μ-)calcul générique, il est avant tout un *model-checker*, et nous voulions proposer un langage de spécification dont les concepts de points fixes assurent la continuité avec l'ancien model-checker Mec.

Nous présentons dans ce chapitre la syntaxe abstraite et la sémantique du langage de spécification de propriétés de Mec V, puis nous introduisons la structure de données qui permet de l'implémenter : les BDDs. Ceci permet de faire le lien entre l'aspect théorique du langage et son implémentation.

3.1 Syntaxe abstraite

Soit $\mathcal{X}$ un ensemble dénombrable de variables, et $\mathcal{R}$ un ensemble dénombrable de symboles de relations. Pour une relation R, on note $ar(R)$ l'*arité* de R, qui est un entier naturel.

3.1.1 Expressions booléennes

Nous construisons l'ensemble des expressions booléennes $\mathcal{EB}$ comme suit :
- $\bot \in \mathcal{EB}$, $\top \in \mathcal{EB}$
- si $x \in \mathcal{X}$, alors $x \in \mathcal{EB}$
- si $x \in \mathcal{X}$, alors $\neg x \in \mathcal{EB}$
- si $\varphi \in \mathcal{EB}$ et $\psi \in \mathcal{EB}$, alors $\varphi \vee \psi \in \mathcal{EB}$
- si $\varphi \in \mathcal{EB}$ et $\psi \in \mathcal{EB}$, alors $\varphi \wedge \psi \in \mathcal{EB}$
- si $x \in \mathcal{X}$ et $\varphi \in \mathcal{EB}$, alors $\exists x.\varphi \in \mathcal{EB}$
- si $x \in \mathcal{X}$ et $\varphi \in \mathcal{EB}$, alors $\forall x.\varphi \in \mathcal{EB}$
- si $R \in \mathcal{R}$ et $e_1, e_2, \ldots, e_{ar(R)} \in \mathcal{EB}$, alors $R(e_1, e_2, \ldots, e_{ar(R)}) \in \mathcal{EB}$

> Comme on le voit ici, les relations prennent comme paramètres des expressions booléennes, ce qui nous contraint automatiquement au cadre de relations sur des booléens. Cette restriction évite d'avoir à typer les variables et les relations, ce qui alourdirait considérablement l'exposé. La section 3.3 page 15 montre comment cette restriction est levée dans Mec V.

3.1.2 Systèmes d'équations

Un système d'équations est donné comme dans [37] et [7] inductivement par une séquence d'équations.

Ceci diffère de la syntaxe concrète de Mec V qui permet de définir les systèmes d'équations par un ensemble non ordonné d'équations qui se voient chacune attribuer un *indice de parité*, par analogie avec les jeux de parité. Cependant, la connaissance de ces indices permet d'ordonner immédiatement les équations et donne aussi le type de point fixe (plus petit ou plus grand) pour chaque équation ; cela justifie de se ramener au cas de séquences d'équations de points fixes.

L'ensemble $\mathcal{E}$ des équations est défini comme suit :
- si $R \in \mathcal{R}$, $x_1, x_2, \ldots, x_{ar(R)} \in \mathcal{X}$ et $\varphi \in \mathcal{EB}$ alors $R(x_1, x_2, \ldots, x_{ar(R)}) \stackrel{\mu}{=} \varphi \in \mathcal{E}$
- si $R \in \mathcal{R}$, $x_1, x_2, \ldots, x_{ar(R)} \in \mathcal{X}$ et $\varphi \in \mathcal{EB}$ alors $R(x_1, x_2, \ldots, x_{ar(R)}) \stackrel{\nu}{=} \varphi \in \mathcal{E}$

Un système d'équations est une séquence d'équations.

3.2 Sémantique

La sémantique d'une spécification va être donnée en deux temps, comme la syntaxe abstraite. La sémantique d'une expression booléenne est un booléen (**0** ou **1**).

Un *environnement* ρ est la réunion de deux fonctions :
- $\rho_{\mathcal{X}}$ de $\mathcal{X}$ dans $\mathbb{B}$ qui à une variable associe sa valeur dans cet environnement
- $\rho_{\mathcal{R}}$ de $\mathcal{R}$ dans l'ensemble des fonctions booléennes. Pour tout $R \in \mathcal{R}$, si elle existe $\rho_{\mathcal{R}}(R)$ est une fonction booléenne d'arité $ar(R)$ qui représente la fonction caractéristique de la relation R

Pour un environnement ρ, on définit les deux *substitutions* :
- $\rho[b/x]$ qui est l'environnement ρ' égal à ρ sauf en x où $\rho'(x) = b$
- $\rho[f/R]$ qui est l'environnement ρ' égal à ρ sauf en R où $\rho'(R) = f$

La sémantique dénotationnelle d'un objet o, dans un environnement ρ, sera notée $[\![o]\!]_\rho$.

3.2.1 Expressions booléennes

- $[\![\bot]\!]_\rho = \mathbf{0}$, $[\![\top]\!]_\rho = \mathbf{1}$
- pour tout $x \in \mathcal{X}$, $[\![x]\!]_\rho = \rho(x)$
- pour tout $x \in \mathcal{X}$, $[\![\neg x]\!]_\rho = \mathbf{1} - [\![x]\!]_\rho$
- pour tous $\varphi \in \mathcal{EB}$ et $\psi \in \mathcal{EB}$, $[\![\varphi \vee \psi]\!]_\rho = [\![\varphi]\!]_\rho$ **ou** $[\![\psi]\!]_\rho$
- pour tous $\varphi \in \mathcal{EB}$ et $\psi \in \mathcal{EB}$, $[\![\varphi \wedge \psi]\!]_\rho = [\![\varphi]\!]_\rho$ **et** $[\![\psi]\!]_\rho$
- pour tous $x \in \mathcal{X}$ et $\varphi \in \mathcal{EB}$, $[\![\exists x.\varphi]\!]_\rho = [\![\varphi]\!]_{\rho[\mathbf{0}/x]}$ **ou** $[\![\varphi]\!]_{\rho[\mathbf{1}/x]}$
- pour tous $x \in \mathcal{X}$ et $\varphi \in \mathcal{EB}$, $[\![\forall x.\varphi]\!]_\rho = [\![\varphi]\!]_{\rho[\mathbf{0}/x]}$ **et** $[\![\varphi]\!]_{\rho[\mathbf{1}/x]}$
- pour tous $R \in \mathcal{R}$ et $e_1, e_2, \ldots, e_{ar(R)} \in \mathcal{EB}$,
$$[\![R(e_1, e_2, \ldots, e_{ar(R)})]\!]_\rho = [\![R]\!]_\rho([\![e_1]\!]_\rho, [\![e_2]\!]_\rho, \ldots, [\![e_{ar(R)}]\!]_\rho)$$

3.2.2 Systèmes d'équations

A un environnement ρ et une équation E de la forme $R(x_1, x_2, \ldots, x_n) \stackrel{\theta}{=} \varphi$ (avec $\theta \in \{\mu, \nu\}$), on associe une fonctionnelle $\mathcal{F}_{E,\rho}$ sur les fonctions booléennes d'arité n, définie comme suit :

$$\mathcal{F}_{E,\rho}(f)(a_1, \ldots, a_n) = [\![\varphi]\!]_{\rho[f/R, a_1/x_1, \ldots, a_n/x_n]}$$

La fonctionnelle $\mathcal{F}_{E,\rho}$ est monotone.

> La monotonicité de cette fonctionnelle est garantie par l'absence de négation d'une formule autre qu'une variable dans la syntaxe des expressions booléennes. Si la négation était introduite, il deviendrait nécessaire de vérifier dans l'équation qu'un symbole relationnel n'apparaît pas sous un nombre impair de négations.

Par le théorème de Knaster-Tarski, dont on peut trouver la preuve dans [7], la fonctionnelle $\mathcal{F}_{E,\rho}$ admet un plus petit et un plus grand points fixes. Nous les notons respectivement $[\![R(x_1, x_2, \ldots, x_n) \stackrel{\mu}{=} \varphi]\!]_\rho$ et $[\![R(x_1, x_2, \ldots, x_n) \stackrel{\nu}{=} \varphi]\!]_\rho$.

La sémantique dénotationnelle d'un système d'équations est un environnement, dans lequel les symboles de relations auront pour valeur la solution du système d'équations.

Nous nous inspirons ici de la notation de [37] pour un système d'équations : on note ϵ la séquence vide, et on utilise une variable $\mathbf{S}$ pour représenter un système d'équations quelconque.

- $[\![\epsilon]\!]_\rho = \rho$
- $[\![R(x_1, x_2, \ldots, x_n) \stackrel{\mu}{=} \varphi\ \mathbf{S}]\!]_\rho = [\![\mathbf{S}]\!]_{\rho[[\![R(x_1, x_2, \ldots, x_n) \stackrel{\mu}{=} \varphi]\!]_\rho / R]}$
- $[\![R(x_1, x_2, \ldots, x_n) \stackrel{\nu}{=} \varphi\ \mathbf{S}]\!]_\rho = [\![\mathbf{S}]\!]_{\rho[[\![R(x_1, x_2, \ldots, x_n) \stackrel{\nu}{=} \varphi]\!]_\rho / R]}$

3.3 Extension aux domaines finis

La logique a été définie formellement et sa sémantique a été donnée, mais en ne faisant apparaître que le fragment booléen. Ceci nous a permi de simplifier nettement l'exposé de cette logique, mais ne nuit en rien à sa généralité.

En effet, il est possible à partir de ce noyau de base d'exprimer des propriétés sur les domaines finis en étendant la syntaxe des expressions et en typant correctement les variables et les relations.

La sémantique sera donnée de manière similaire ; cette extension peut être vue comme du μ-calcul vectoriel, dont on sait (voir par exemple [7]) qu'il a les mêmes propriétés que le μ-calcul (non-vectoriel) dont il est issu.

Par exemple, donnons-nous l'équation

$$R(x) \stackrel{\mu}{=} (x = 2) \vee R(x - 1)$$

dans laquelle x est de type entier sur le domaine $[0, 3]$, et où la syntaxe est étendue pour pouvoir comparer des entiers $(x = 2)$ ou calculer la différence entre deux entiers $(x - 1)$.

Nous pouvons traduire cette équation immédiatement en codant la variable x en binaire en complément à deux. La traduction est donnée ci-dessous. On peut constater immédiatement que le premier paramètre (représentant le bit de poids fort) n'est en fait pas utile (il est toujours contraint à être faux), mais si on a à l'esprit un algorithme de traduction, celui-ci doit prendre en compte le cas où $x - 1$ devient négatif, même si ici $x - 1$ est tout de suite réutilisé dans une relation qui lui interdit de devenir négatif.

$$R(x_1, x_2, x_3) \stackrel{\mu}{=} (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee R(\bot, x_2 \wedge x_3, x_2 \wedge \neg x_3)$$

La façon dont les ensembles de variables sur les domaines finis sont codés dans Mec V est décrite dans la section 5.3, page 47.

De la même manière, cette logique est étendue avec les types "configuration d'un nœud AltaRica" et "vecteur d'événements" d'un nœud AltaRica, et la relation de transition et l'ensemble d'états initiaux d'un nœud AltaRica sont mis à la disposition de l'utilisateur du langage de spécification par l'intermédiaire de relations pré-définies, ce qui permet de vérifier des modèles AltaRica. L'implémentation de cet aspect est décrite dans la sous-section 5.5.7 page 55.

3.4 Expressivité par rapport aux logiques usuelles

L'expressivité de la logique de Mec V est très grande lorsqu'on la compare aux logiques disponibles habituellement dans les outils de model-checking (CTL, LTL, μ-calcul sans alternance).

Mads Dam démontre de manière constructive dans l'article [20] que la logique CTL* (qui contient LTL et CTL) peut être traduite en μ-calcul modal. La construction proposée par Mads Dam est coûteuse (doublement exponentielle en temps par rapport à la taille de la formule CTL* de départ).

La sémantique de CTL, en revanche, est souvent donnée en termes de points fixes (par exemple dans [49]) ; la traduction de formules CTL en μ-calcul est immédiate.

Enfin, les modalités du μ-calcul modal se codent aisément dans la logique de Mec V grâce aux quantificateurs du premier ordre si on se donne une relation de transition puisqu'il suffit alors d'écrire la définition des modalités directement.

3.5 Diagrammes de décisions binaires (BDDs)

Les diagrammes de décision binaires (*BDD, Binary Decision Diagrams*) sont une structure de données qui permet de représenter des relations entre variables booléennes. Ils ont été introduits par Randal E. Bryant dans [9] et font désormais partie des enseignements de base sur la vérification.

Nous allons présenter les BDDs de deux manières complémentaires : d'abord comme des programmes à choix uniques, puis comme des automates reconnaissant un ensemble de mots binaires. Enfin, nous faisons converger ces deux façons de voir en introduisant les notions d'*ordre sur les variables* et de *réduction*.

3.5.1 Programmes à choix uniques

Un BDD représente une cascade de choix permettant de déterminer si des variables booléennes satisfont une certaine relation ou non.

Par exemple, voici un petit programme qui vérifie, étant donnés trois booléens x, y et z, si $x \Rightarrow y$.

implique(x, y, z) $\equiv$
    si x **alors**
        si y **alors**
            retourner *vrai*
        sinon
            retourner *faux*
        finsi
    sinon retourner *vrai*
    finsi

Ce programme a la particularité suivante : lors d'une exécution, il ne testera au plus qu'une seule fois chaque variable.

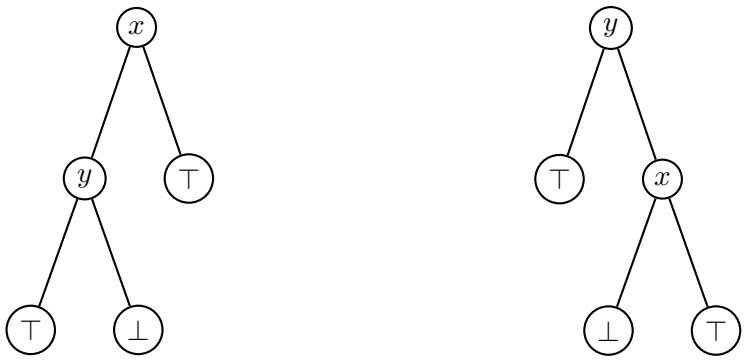
On remarque qu'il est aussi possible de l'écrire de manière différente comme suit, en inversant l'ordre dans lequel on teste les variables :

implique(x, y, z) $\equiv$
    si y **alors**
        retourner *vrai*
    sinon
        si x **alors**
            retourner *faux*
        sinon
            retourner *vrai*
        finsi
    finsi

Comme ces programmes ne contiennent (à dessein) que des tests, nous pouvons introduire une notation particulière pour les écrire : nous noterons le test **si** x **alors** P **sinon** Q par $(x?P : Q)$. Les actions **retourner** *vrai* et **retourner** *faux* sont écrites respectivement $\top$ et $\bot$.

Les deux programmes peuvent donc s'écrire respectivement :

$(x?\ (y?\ \top : \bot) : \top)$ et $(y?\ \top : (x?\ \bot : \top))$ ou plus graphiquement par les deux arbres de décisions suivants (l'arc de gauche est l'arc **alors**, et l'arc de droite est l'arc **sinon**) :



3.5.2 Automates sur des mots binaires

Pour représenter la relation qui unit x, y et z, on peut aussi définir un automate sur les mots binaires représentant les valuations des trois variables x, y, et z. Il faut pour cela décider d'un ordre grâce auquel chaque lettre du mot binaire sera associée à la variable correspondante.

Pour l'ordre $x \rightarrow y \rightarrow z$, les valuations qui satisfont $x \Rightarrow y$ sont les mots binaires 000, 010, 110, 001, 011, 111, qui sont reconnus par l'automate suivant :

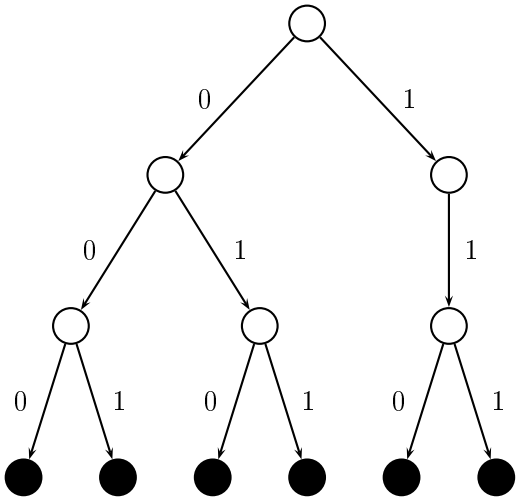


FIG. 3.1 – Automate déterministe reconnaissant les valuations de $x \Rightarrow y$

Tout automate acyclique déterministe peut être minimisé en temps linéaire, et voici l'automate que l'on obtient sur notre exemple.

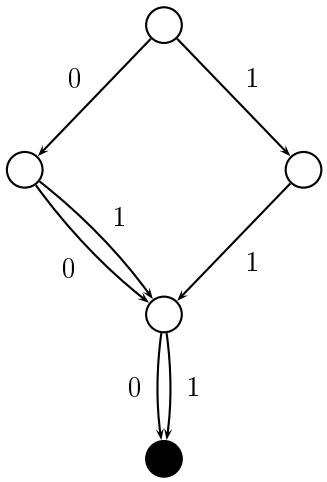


FIG. 3.2 – Automate déterministe minimal correspondant

3.5.3 Diagrammes de décisions binaires

Après avoir présenté deux façons intuitives de représenter des relations binaires, nous pouvons introduire la notion de diagramme de décisions binaires *ordonnés* et *réduits*.

Ces deux qualificatifs assurent un codage compact mais surtout *canonique* d'une relation binaire et offrent grâce à cela des algorithmes efficaces de calcul des opérations d'union, d'intersection, de négation, de projection, de cylindrification, et de substitution.

Les ROBDDs (Reduced Ordered BDDs) ont été introduits par Randal E. Bryant dans [9], et sont désormais une structure de données courante en vérification. On pourra consulter par exemple [40] pour un panorama détaillé sur ce sujet. Nous appellerons par abus de langage les ROBDDs des BDDs dans le reste de ce document.

Plutôt que de donner la description usuelle des BDDs qui se base sur la décomposition de Shannon d'une fonction booléenne, nous allons montrer les étapes qui permettent de transformer un automate déterministe acyclique représentant les valuations de la relation en un BDD.

Etant donnée notre relation $x \Rightarrow y$ définie sur les trois variables x, y, et z, il est possible de supprimer encore des sommets tout en conservant toute l'information en *étiquetant* les sommets par la variable dont on veut tester la valeur. Ceci permet d'omettre certains tests inutiles de variables intermédiaires dont le calcul ne dépend pas.

Enfin, pour avoir une structure de données homogène (tout sommet a deux successeurs), on utilise un deuxième état final symbolisant le rejet d'une valuation. Ces deux étapes sont représentées pour notre exemple sur la figure qui suit.

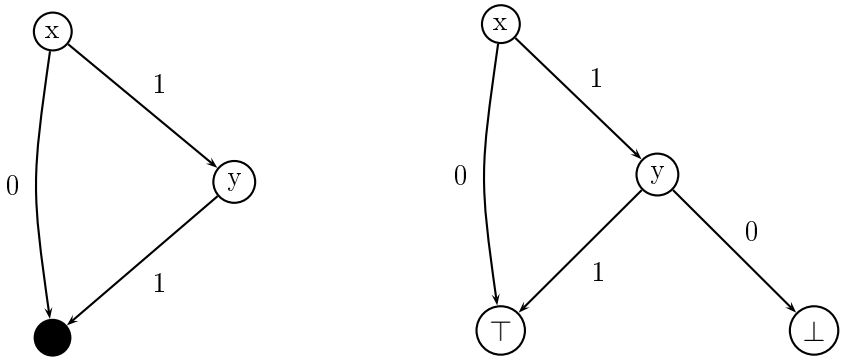


FIG. 3.3 – Les deux étapes de la transformation en BDD

Chapitre 4

Le formalisme AltaRica

Le formalisme AltaRica [6, 45] a été conçu au LaBRI pour permettre d'utiliser une même description d'un système avec plusieurs outils d'analyse. Il est par exemple possible à partir de la description AltaRica d'un modèle de le simuler, de générer des arbres de défaillances, ou de vérifier qu'il satisfait des propriétés logiques.

Plusieurs organismes ont participé à l'élaboration du formalisme et du langage associé, notamment le LaBRI et aussi des entreprises privées comme IXI (maintenant GFI Consulting) ou Dassault Systèmes. Cette collaboration a été fructueuse et nous a convaincus de l'importance des deux propriétés suivantes d'AltaRica :

- une sémantique clairement définie : on peut associer à toute description AltaRica le système de transitions qui lui correspond,
- un langage dont la syntaxe est proprement spécifiée.

Le but de ces deux points étant que les outils autour d'AltaRica, développés par les divers partenaires, puissent accepter les mêmes descriptions et que leurs résultats puissent être mis en corrélation.

De nombreuses études comme par exemple [27, 11, 28] ont été réalisées en utilisant le formalisme AltaRica, et ce formalisme est largement utilisé par nos partenaires industriels.

Nous allons dans ce chapitre présenter les concepts d'AltaRica, et nous utiliserons des exemples écrits dans le langage AltaRica. La syntaxe du langage est définie en annexe, page 99. La sémantique formelle du langage AltaRica est définie dans la thèse de Gérald Point [45].

Les évolutions apportées à AltaRica durant cette thèse sont répertoriées en fin de chapitre, page 38.

4.1 Décrire les comportements par contraintes

Nous donnons tout de suite un exemple simple qui permet de montrer comment on décrit un automate à contraintes en AltaRica.

```
node exemple
    state x : [ 0, 8 ];
    event e;

    trans
        x <= 2 |- e -> x := x + 1;
        x >= 3 |- e -> x := x + 2;
```

```
    init
        x := 0;
edon
```

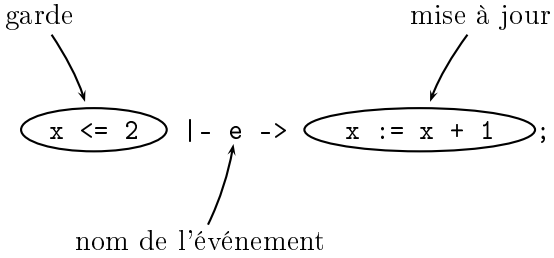
L'espace d'états est donné par les valuations possibles des variables d'état (ici, `x`, qui peut prendre 9 valeurs différentes). A l'exception de l'événement ε que nous décrirons plus loin, tous les événements doivent être nommés, et nous nous donnons donc ici un événement `e`.

Afin d'expliquer le comportement de modèles AltaRica, nous donnerons l'équivalent de ces modèles sous forme de systèmes de transitions.

4.1.1 Sémantique d'une macro-transition

Chaque *macro-transition* spécifiée dans la section `trans` d'un nœud AltaRica peut donner plusieurs transitions dans la sémantique du nœud.

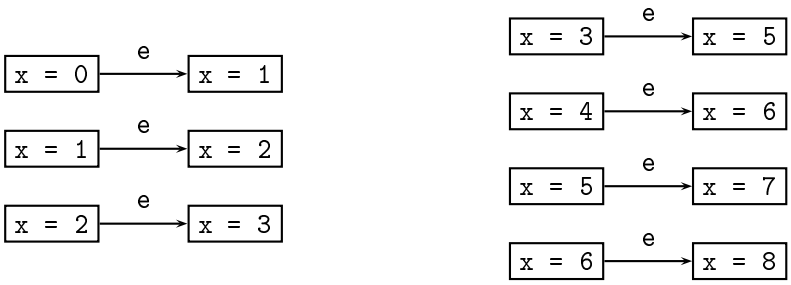
Voici comment s'effectue le passage de la macro-transition à sa sémantique sur l'exemple ci-dessus.



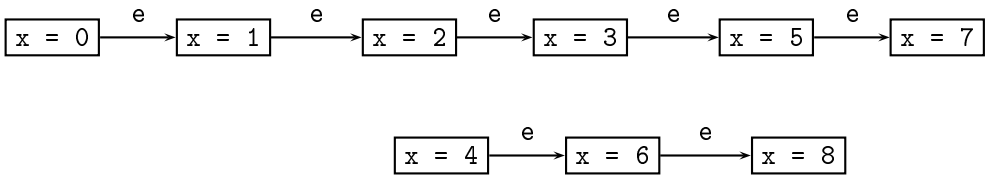
La garde spécifie l'ensemble des configurations qui peuvent potentiellement être source d'une transition engendrée par la macro-transition. Dans ce cas-ci, il s'agit des valuations pour lesquelles `x` vaut 0, 1, ou 2. Le nom de l'événement est celui qui sera utilisé pour chaque transition engendrée.

Pour chaque configuration de départ, les variables d'état sont affectées par la mise à jour, et cela donne l'ensemble des configurations qu'il est possible d'atteindre à partir de cette configuration.

Voici donc les transitions que l'on obtient pour la première et la deuxième macro-transitions :



Et nous obtenons graphiquement le système de transitions suivant après identification des configurations égales.



De plus, des ε-transitions n'apparaissant pas ici font partie de la sémantique du modèle, sous la forme de boucles sur chaque configuration.

Notez que depuis l'état initial `x = 0`, la composante où `x` vaut 4, 6, ou 8 n'est pas accessible.

4.1.2 Non-déterminisme

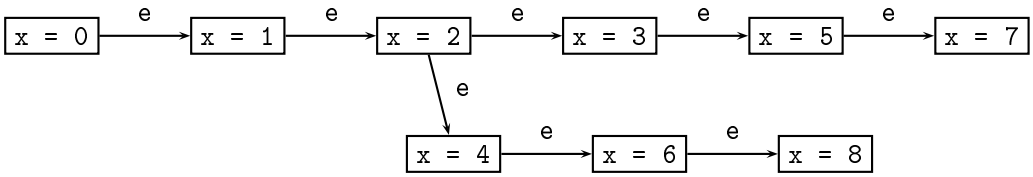
Si les gardes de deux macro-transitions ayant le même nom d'événement sont d'intersection non vide, plusieurs transitions différentes peuvent être tirées lorsque cet événement survient dans une configuration appartenant à l'intersection des deux gardes. Le système peut engendrer ou accepter les divers comportements indifféremment : c'est du *non-déterminisme*.

Le non-déterminisme est utile pour modéliser des comportements dont on ne sait pas *a priori* lequel aura lieu. En autorisant les deux comportements de manière non-déterministe, on est sûr que l'énumération de tous les comportements du système les contiendra tous.

Voici une légère modification de la garde de la deuxième macro-transition de l'exemple précédent.

```
node exemple
    state x : [ 0, 8 ];
    event e;

    trans
        x <= 2 |- e -> x := x + 1;
        x >= 2 |- e -> x := x + 2;

    init
        x := 0;
edon
```

On obtient alors le système de transitions suivant :



On voit ici que dans la configuration `x = 2`, le système peut passer indifféremment à la configuration `x = 3` ou à la configuration `x = 4` par un événement `e`.

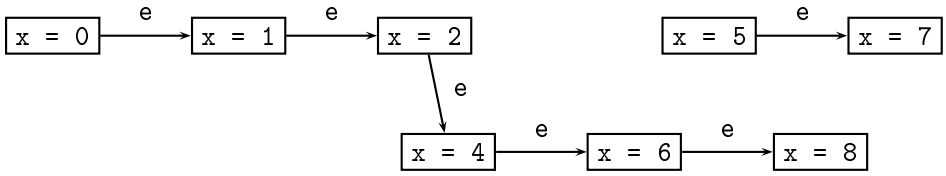
4.1.3 Assertion et contraintes de domaines

Il est possible de spécifier une contrainte appelée *assertion* qui *doit* être satisfaite par toute configuration du système.

Nous pouvons par exemple interdire la configuration dans laquelle `x` vaut 3. Il suffit pour cela d'ajouter à notre nœud `exemple` précédent le champ :

```
assert
    x != 3;
```

Ceci donnera le système de transitions suivant :



L'assertion contient aussi implicitement les contraintes de domaines des variables du nœud. Ainsi, dans les exemples précédents, nous n'avons pas eu besoin de préciser dans la garde de la deuxième macro-transition qu'elle n'était possible que si `x <= 6`. Du fait que les deux configurations `x = 7` et `x = 8` mèneraient à des valeurs de `x` hors de son domaine, ces transitions n'existent pas dans la sémantique du modèle. Le fait que la valeur des variables *après* tirage d'une transition puisse affecter la tirabilité de la transition est souvent appelé test de *post-condition*.

L'assertion dans sa totalité (et pas seulement les contraintes de domaines) sert à évaluer la post-condition.

4.2 Eléments introduits pour la hiérarchie

AltaRica permet de réutiliser des nœuds à l'intérieur d'autres nœuds. Nous allons ici présenter deux concepts qui, bien qu'ayant déjà un sens dans un nœud simple, prennent tout leur sens lorsqu'on souhaite modéliser un système qui contient des sous-nœuds.

> En accord avec [45], nous utiliserons parfois le terme *composant* pour désigner un nœud qui ne contient pas de sous-nœud.

4.2.1 Variables de flux/variables d'état

Il existe deux sortes de variables dans un nœud AltaRica : les variables d'état et les variables de flux.

Les variables d'état représentent l'état du système, elles peuvent être utilisées à tout endroit où une variable peut être utilisée : dans une garde, dans l'affectation qui suit une transition, ou dans un prédicat global appelé *assertion* qui contraint les valeurs que peuvent prendre les variables.

Les variables de flux sont comme les variables d'état, à ceci près qu'elles ne font pas partie de l'"état" du système : elles représentent des valeurs auxiliaires qui sont uniquement en relation avec l'état courant grâce à l'assertion. Les variables de flux ne peuvent pas être affectées explicitement lors du franchissement d'une transition.

Les variables de flux ont deux rôles possibles :

- elles permettent d'extraire de l'information du système, de plus haut niveau que simplement l'état courant du système, par exemple pour abstraire de l'information pertinente
- elles permettent aussi de contraindre l'état du système puisqu'elles apparaissent dans l'assertion, qui est un prédicat que toutes les configurations du système doivent satisfaire.

L'intérêt principal de ces variables de flux est de permettre la communication d'information entre nœuds. (cf. sous-section 4.3.2, page 29)

Il est temps d'introduire le vocabulaire adéquat : on appelle *état* d'un modèle AltaRica une valuation de ses variables d'état, et *configuration* une valuation de toutes ses variables, y compris les variables de flux.

4.2.2 Les ε-transitions

Comme nous l'avons laissé entendre un peu plus haut dans l'exemple, des transitions ε (c'est-à-dire sans nom) sont ajoutées à tous les états sous la forme de boucles.

On peut déjà noter que ces ε-transitions ne peuvent pas être spécifiées explicitement dans un modèle AltaRica. Elles font uniquement partie de la sémantique du modèle.

Leur rôle est d'assurer qu'un nœud ne "bloque" jamais : pour une configuration donnée, il est toujours possible de tirer une ε-transition sortante. Cette particularité est très utile lors de la composition hiérarchique de nœuds AltaRica, comme nous le verrons dans la section suivante.

Toutes les configurations correspondant au même état du système sont reliées par des ε-transitions. En particulier, à toute configuration est associée une boucle ε. Les variables de flux peuvent toujours changer de valeur lors du tirage d'une transition, et les ε-transitions ne font pas exception.

4.2.3 Un exemple

Afin d'éclaircir l'interaction entre les variables de flux et les ε-transitions, nous allons prendre l'exemple d'une salle qui peut contenir jusqu'à cinq personnes, et dont le battant de porte permet à une personne de se dissimuler (voir la figure 4.1).

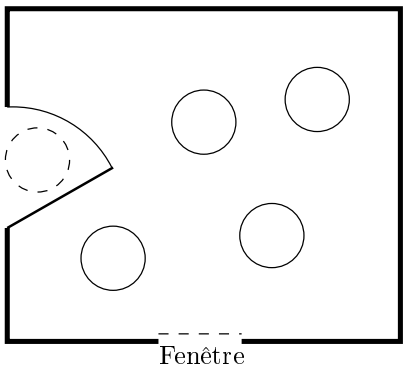


FIG. 4.1 – Salle avec vue partielle sur l'intérieur

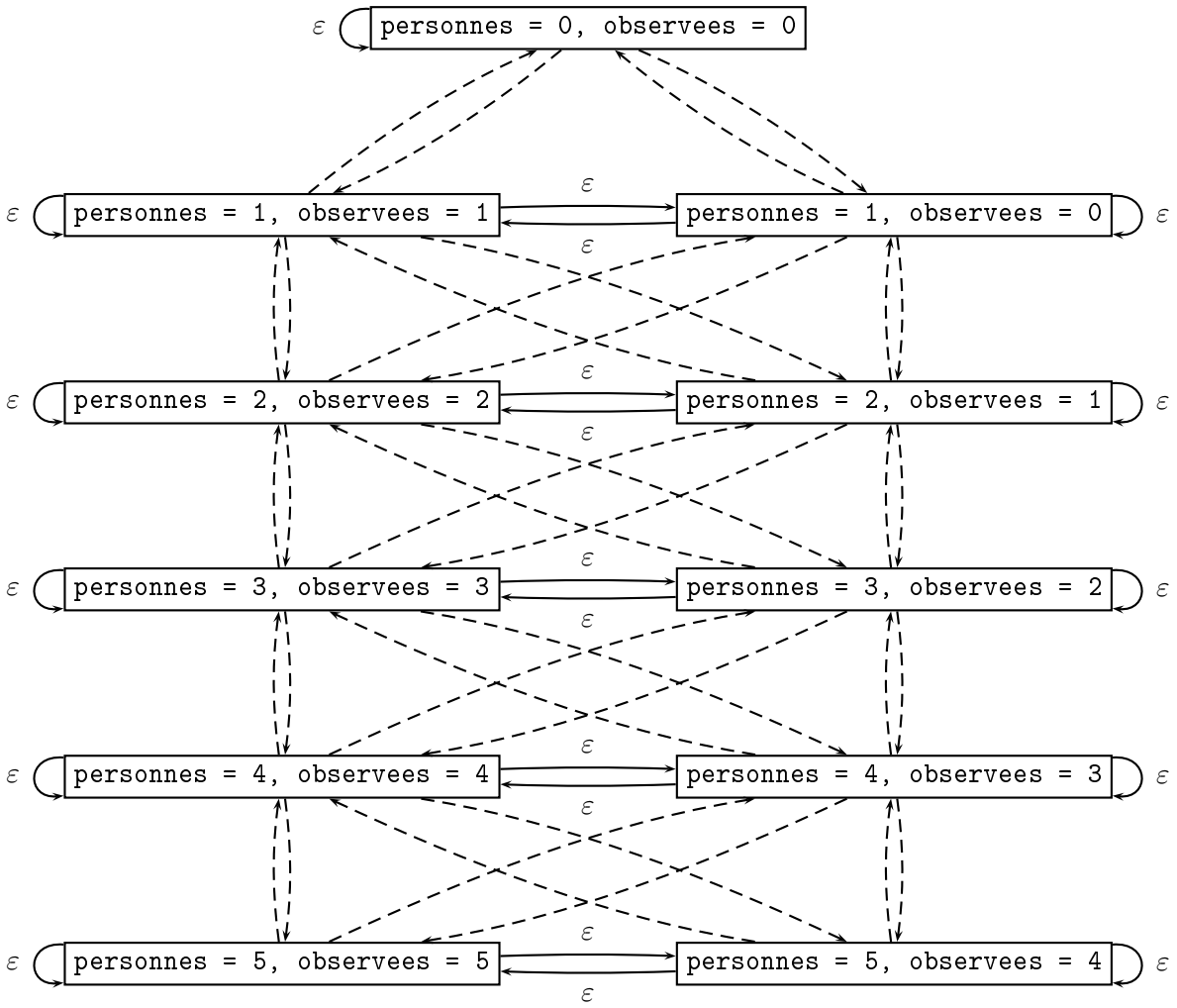
Une personne observant la salle depuis l'extérieur par la fenêtre ne peut jamais savoir s'il y a une personne derrière le battant de la porte. A tout instant, le nombre de personnes qu'elle observera dans la salle sera donc faux d'au plus une personne par défaut.

L'état de la salle est clairement constitué du nombre exact de personnes présentes et nous nous donnons une variable de flux pour représenter la quantité de personnes observables de l'extérieur de la salle. Ceci se traduit aisément en AltaRica, cf. figure 4.2.

```
node SalleAvecRecoin
    state personnes : [ 0, 5 ];
    flow observees : [ 0, 5 ];

    event entre, sort;

    trans
        personnes < 5 |- entre -> personnes := personnes + 1;
        personnes > 0 |- sort -> personnes := personnes - 1;

    assert
        observees <= personnes & observees >= (personnes - 1);
edon
```

FIG. 4.2 – Modèle AltaRica de la salle de la figure 4.1

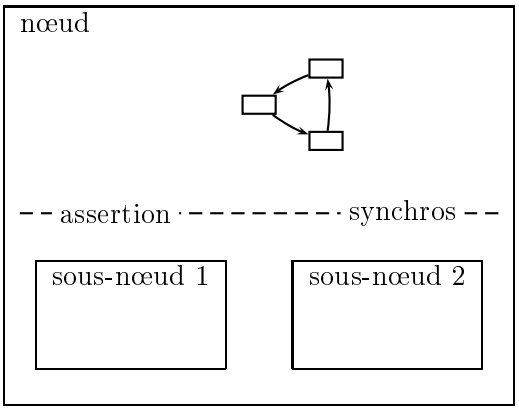
Nous représentons ci-dessous le graphe des configurations de ce nœud. Les événements `entre` et `sort` ne sont pas écrits pour que le graphe reste lisible mais leurs transitions sont représentées en lignes pointillées : les transitions descendantes correspondent à l'événement `entre`, les transitions montantes à l'événement `sort`. On peut observer que les ε-transitions conservent bien les états, mais laissent les flux libres à l'intérieur de leurs contraintes.



4.3 Composition hiérarchique

Le formalisme AltaRica permet d'utiliser un nœud AltaRica à l'intérieur d'un autre, ou de faire communiquer plusieurs nœuds entre eux. Dans ce dernier cas, il est toujours nécessaire qu'un nœud englobe les nœuds communiquants : ce nœud joue le rôle de contrôleur.

Un nœud se compose donc d'une partie locale et de ses sous-nœuds potentiels.



Les comportements de la partie locale du nœud et les sous-nœuds sont liés par l'assertion et les *synchronisations d'événements*, afin de décrire le comportement du nœud dans sa globalité.

Notez que les deux sous-nœuds n'ont pas connaissance de l'existence l'un de l'autre. La communication qui s'établit entre eux est *contrôlée* par leurs ancêtres communs.

4.3.1 Sémantique d'un nœud sans synchronisations

Nous allons d'abord décrire comment un nœud se comporte lorsqu'aucune interaction n'est imposée entre lui et ses sous-nœuds.

Chaque configuration du nœud est prise dans les valuations de la réunion des variables du nœud et de ses sous-nœuds. Cette réunion est disjointe et on utilise les noms des sous-nœuds comme préfixes pour nommer les variables afin de les distinguer. La variable `x` du sous-nœud `N` est nommée `N.x` lorsqu'on veut y faire référence.

Vecteurs d'événements

Une transition est désormais étiquetée par un *vecteur d'événements*. Chaque vecteur d'événements est constitué d'un des événements du nœud local et d'un événement par sous-nœud : un vecteur d'événements code donc précisément les événements de chacun des composants apparaissant dans le nœud, qui seront activés si ce vecteur l'est.

En l'absence d'autre contrainte, chacun des événements (local ou apparaissant dans un sous-nœud direct) constitue un vecteur d'événements où tous les autres éléments du vecteur sont l'événement ε du nœud correspodant. Dans ce contexte, deux événements nommés ne peuvent pas avoir lieu en même temps. On peut parler d'*asynchronisme fort*.

Exemple

Nous définissons dans la figure 4.3 un nœud `Simple` qui contient une variable booléenne qui passe de l'état faux à l'état vrai sur un événement `e`.

Le nœud `Simple` a donc deux événements : l'événement `e` qui change son état, et l'événement ε qui boucle sur chaque configuration (qui ne sont ici que des états).

Le nœud `PasDeSynchro` contient deux sous-nœuds `N1` et `N2` qui sont des nœuds `Simple`.

Comme il ne définit pas d'événement propre, son seul événement local est ε. Ses vecteurs d'événements sont donc <ε, `N1.`ε, `N2.`ε>, <ε, `N1.e, N2.`ε>, et <ε, `N1.`ε, `N2.e`>.

```
node Simple
    state v : bool;
    event e;
    trans
        ~v |- e -> v := true;
edon

node PasDeSynchro
    sub N1, N2 : Simple;
edon
```

FIG. 4.3 – Nœuds `Simple` et `PasDeSynchro`

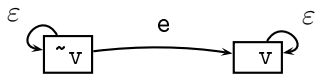


FIG. 4.4 – Sémantique du nœud `Simple`

4.3.2 Synchronisation par assertion

Il est possible dans l'assertion d'un nœud de contraindre à la fois les variables locales au nœud, et les variables de ses sous-nœuds. Grâce à cela, il est par exemple possible de partager des variables entre nœuds : il suffit de contraindre deux variables à être égales (figure 4.6). L'assertion ne peut qu'*interdire* certaines configurations, ainsi que par conséquent les transitions qui en partent ou qui y arrivent.

Toute autre contrainte est acceptée, et parmi les utilisations possibles, on peut par exemple contraindre un nœud prévu de manière générique à limiter ses variables dans un certain domaine, ce qui est illustré par la figure 4.7.

4.3.3 Synchronisation d'événements

L'autre mécanisme de communication disponible en AltaRica est la synchronisation d'événements. Elle permet de préciser les vecteurs d'événements que l'on souhaite garder dans le nœud. Ces vecteurs sont appelés *vecteurs de synchronisation*.

Chaque vecteur fait apparaître au plus un événement par sous-nœud ainsi qu'au plus un événement du nœud local, et est complété par des événements ε. Dès qu'un événement nommé d'un sous-nœud apparaît dans un vecteur de synchronisation, cet événement ne sera plus synchronisé avec l'événement ε du nœud local, comme c'est le cas en l'absence de synchronisation.

Voyons sur notre exemple simple ce qui se passe lorsqu'on synchronise les deux événements nommés des deux sous-nœuds (figure 4.8).

Comme on peut le voir en comparant les figures 4.5 (page 30) et 4.9, les vecteurs <ε, `N1.e`, `N2.`ε> et <ε, `N1.`ε, `N2.e`> n'apparaissent plus, puisque respectivement `N1.e` et `N2.e` sont explicitement cités dans le vecteur de synchronisation.

Il est d'ailleurs intéressant de constater sur cet exemple que l'on aurait obtenu exactement la même sémantique si au lieu de synchroniser les événements `N1.e` et `N2.e` on avait interdit

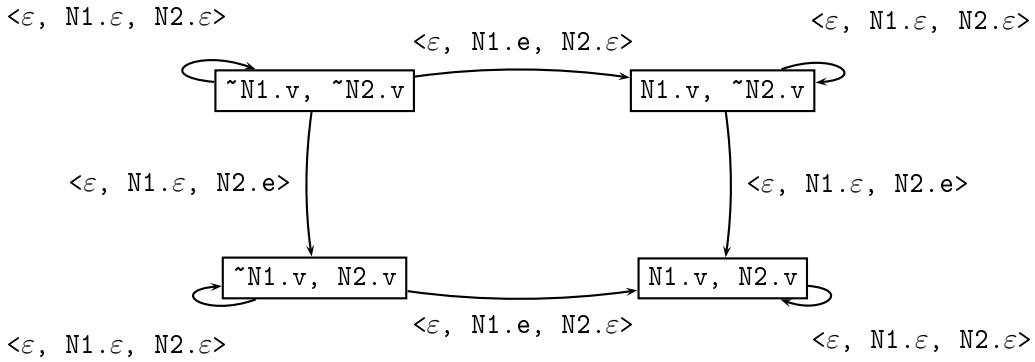


FIG. 4.5 – Sémantique du nœud `PasDeSynchro`

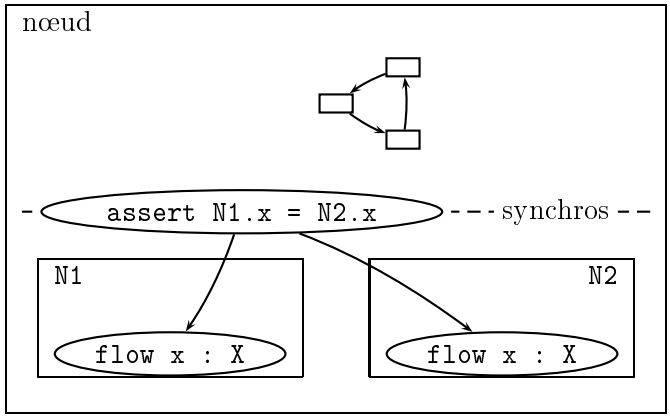


FIG. 4.6 – Partage de variables

les deux configurations où `N1.v` et `N2.v` sont différentes en imposant l'égalité dans l'assertion (`assert N1.v = N2.v`). Nous utilisons d'ailleurs cet exemple pour exposer la façon dont est stocké un nœud AltaRica dans Mec V, page 56.

Bien qu'anecdotique sur cet exemple, cette dualité entre contrainte sur les configurations et contrainte sur les vecteurs d'événements est une des caractéristiques importantes d'AltaRica. Elle est le témoin du fait que le langage AltaRica est un langage de haut niveau, qui permet de faire des choix de modélisation différents pour un "même" système.

Nous allons maintenant présenter deux outils pratiques pour modéliser des concepts de priorité : les priorités entre événements et les vecteurs de diffusion.

4.4 Priorités

Il est souvent intéressant de modéliser le fait qu'une action est "plus prioritaire" qu'une autre. Cet aspect apparaît à deux endroits en AltaRica : il est possible d'imposer des priorités entre événements locaux d'un même nœud, et indirectement de rendre plus prioritaires certains vecteurs d'événements par rapport à d'autres, par l'intermédiaire des *vecteurs de diffusion*.

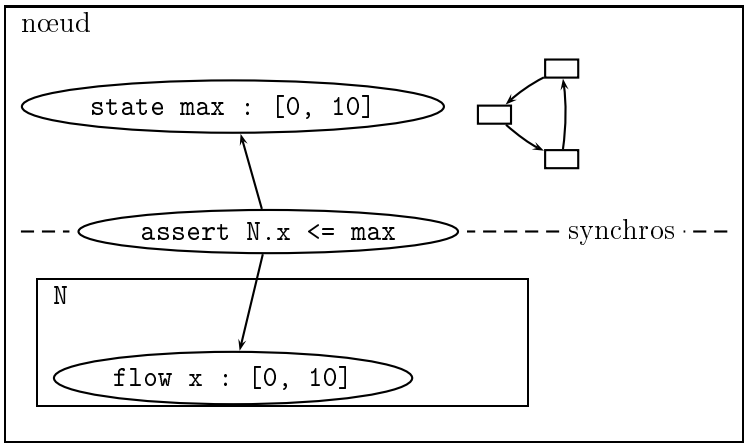


FIG. 4.7 – Contrainte sur une sous-variable

```
node SynchroSimple
    sub N1, N2 : Simple
    sync <N1.e, N2.e>
edon
```

FIG. 4.8 – Le nœud `SynchroSimple`

4.4.1 Priorités locales explicites

Il est possible en AltaRica de préciser un ordre partiel quelconque sur les événements nommés locaux d'un nœud, et ainsi de spécifier quels événements prennent le pas sur les autres.

Le fait qu'un événement `e1` soit prioritaire par rapport à un autre `e2` signifie que dans une configuration donnée, toute transition étiquetée par `e2` ne peut se produire que si aucune transition étiquetée par `e1` n'est tirable.

4.4.2 Vecteurs de diffusion

Les deux façons de lier les comportements de nœuds que nous avons vues précédemment (assertion et synchronisation) conservent toujours une symétrie dans les nœuds : aucune de ces deux méthodes ne "privilégie" l'un des nœuds. Si deux événements sont synchronisés et que l'un des deux n'est pas possible, aucun des deux ne se produira.

Il est souvent intéressant de modéliser des phénomènes où l'un des deux événements doit se produire *s'il peut* et n'empêche pas l'autre événement de se produire sinon.

Pour illustrer ce comportement, on peut prendre l'exemple de la condamnation centralisée des portes d'une voiture.

Nous modélisons simplement une portière par le fait qu'elle a un clapet qui peut être soit ouvert soit fermé.

```
node Portiere
    state clapetFerme : bool;
    event fermeClapet, ouvreClapet;
```

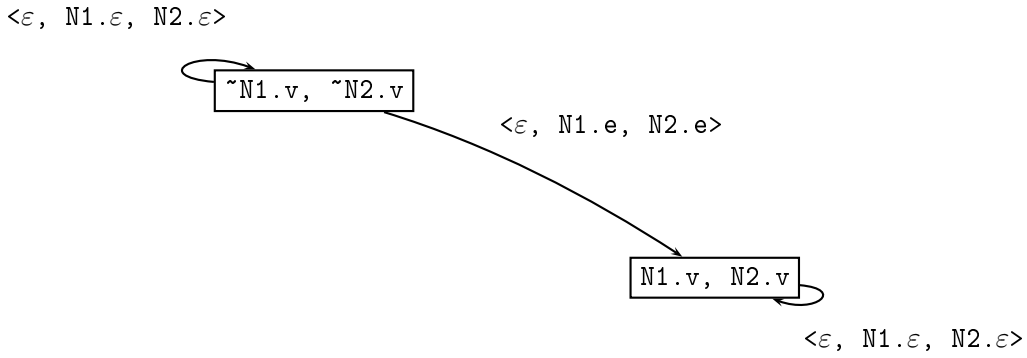


FIG. 4.9 – Sémantique du nœud `SynchroSimple`

```
    trans
        clapetFerme |- ouvreClapet -> clapetFerme := false;
        ~clapetFerme |- fermeClapet -> clapetFerme := true;
edon
```

Le comportement que l'on souhaite pour les quatre portières est le suivant :

(P1) Toute fermeture ou ouverture des clapets des portes avant (`PAvG` et `PAvD`) est répercutée sur les autres clapets des autres portes.

(P2) Les clapets des portes arrière (`PArG` et `PArD`) n'influencent aucun autre clapet lorsqu'ils sont actionnés.

Nous traduisons ces deux contraintes par des vecteurs de diffusion et de synchronisation.

La première contrainte se traduit par le fait que tous les événements de fermeture doivent être synchronisés. Cependant, du fait de la deuxième contrainte, il se peut que l'un des clapets des portes arrière soit déjà fermé et que la portière ne puisse donc pas répondre à la demande de fermeture (l'événement `fermeClapet` n'est pas tirable si le clapet est déjà fermé). La diffusion (représentée syntaxiquement par un point d'interrogation à la suite du nom d'événement) permet de répondre à ce problème : l'événement se produira lorsque le vecteur d'événement est tiré si et seulement si cet événement est tirable.

Le point clé est qu'un événement "diffusé" ne peut empêcher le vecteur de diffusion de se produire. Il correspond donc bien ici à ce que l'on veut : le fait que l'un des clapets avant se ferme correpond à une action physique de l'usager qui ne saurait être interdite par l'état d'un clapet arrière.

Il est nécessaire d'expliciter les contraintes de synchronisation de la propriété (P2) car, comme les événements `PArG.fermeClapet` et `PArD.fermeClapet` apparaissent dans les vecteurs de diffusion, ces événements sont considérés comme étant déjà pris en compte et les événements `<ε, PAvG.ε, PAvD.ε, PArG.fermeClapet, PArD.ε>` et `<ε, PAvG.ε, PAvD.ε, PArD.ε, PArD.fermeClapet>` ne sont pas ajoutés automatiquement à la sémantique du système (cf. page 28 et figure 4.5). On impose leur présence en les spécifiant manuellement. Le nœud AltaRica contenant toutes ces contraintes de synchronisations est représenté à la figure 4.10.

Le cas de l'ouverture des portes est exactement le même.

Un lecteur attentif remarquera que la voiture peut se retrouver dans des configurations "puits" pour les clapets avant : si les deux clapets avant sont décorrélés (le cas se présente puisque nous n'avons pas spécifié d'état initial), ils ne pourront plus jamais bouger.

```
node Voiture
  sub PAvG, PAvD, PArG, PArD : Portiere

  sync
    <PAvG.fermeClapet, PAvD.fermeClapet, PArG.fermeClapet?, PArD.fermeClapet?>;
    <PAvG.ouvreClapet, PAvD.ouvreClapet, PArG.ouvreClapet?, PArD.ouvreClapet?>;

    <PArG.fermeClapet>;
    <PArD.fermeClapet>;

    <PArG.ouvreClapet>;
    <PArD.ouvreClapet>;
edon
```

FIG. 4.10 – Le nœud `Voiture`

Sémantique de la diffusion

La sémantique de la diffusion peut être donnée en introduisant des priorités sur les vecteurs d'événements. Un vecteur de diffusion devient alors un ensemble de vecteurs d'événements qui correspondent à toutes les combinaisons possibles de présence ou non-présence des événements diffusés. S'il y a n événements diffusés dans un vecteur, il donnera lieu à 2^n vecteurs d'événements et les priorités de ces vecteurs sont données par le nombre d'événements diffusés présents dans le vecteur d'événements, le plus prioritaire étant celui qui a le moins d'occurrences d'événements ε.

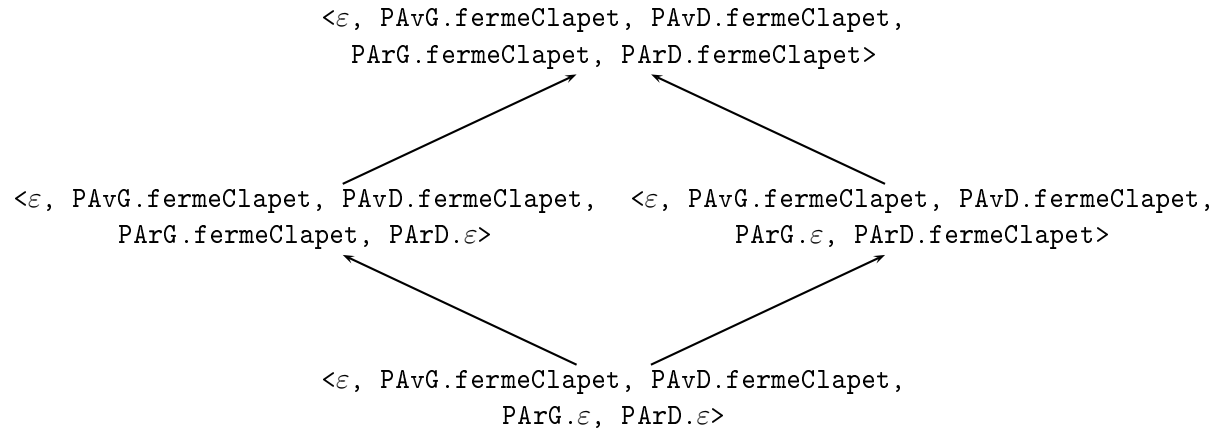
Pour le vecteur

<ε, PAvG.fermeClapet, PAvD.fermeClapet, PArG.fermeClapet?, PArD.fermeClapet?>

les quatre vecteurs d'événements seront donc :

- <ε, PAvG.fermeClapet, PAvD.fermeClapet, PArG.fermeClapet, PArD.fermeClapet>
- <ε, PAvG.fermeClapet, PAvD.fermeClapet, PArG.fermeClapet, PArD.ε>
- <ε, PAvG.fermeClapet, PAvD.fermeClapet, PArG.ε, PArD.fermeClapet>
- <ε, PAvG.fermeClapet, PAvD.fermeClapet, PArG.ε, PArD.ε>

ordonnés comme suit :



Le langage AltaRica offre aussi la possibilité d'imposer des contraintes numériques sur le nombre d'événements diffusés qui doivent apparaître dans les vecteurs d'événements déduits de ce vecteur de diffusion. Lorsque cette contrainte est une inégalité, on peut spécifier si l'on souhaite ordonner les vecteurs possibles dans l'ordre croissant ou décroissant du nombre d'événements apparaissant dans les vecteurs en utilisant des mots-clés `min` et `max`.

> Il est envisagé dans le projet AltaRica de considérer une nouvelle approche pour spécifier les vecteurs de diffusion, et cette idée est déjà utilisée dans le langage AltaRica DataFlow. Il s'agit de spécifier les événements qui peuvent ou doivent apparaître dans le "vecteur de synchronisation" par une formule logique plutôt qu'en imposant une contrainte numérique sur le nombre d'événements devant apparaître dans le vecteur ; ces deux façons de spécifier les vecteurs sont complémentaires.

4.5 Notion de visibilité

Jusqu'ici, nous avons montré plusieurs exemples d'assertions et de contraintes d'événements qui lient des nœuds différents entre eux. Ceci n'est possible qu'à la condidition que les éléments (variables ou événements) utilisés dans ces liaisons soient *visibles* depuis un autre nœud.

Ces propriétés de visibilité interviennent en amont de la sémantique : elles sont un aspect "langage" d'AltaRica. Les contraintes de visibilité interdisent l'utilisation de certaines variables dans certains nœuds, de manière purement "syntaxique".

Les trois propriétés de visibilité disponibles en AltaRica sont les suivantes : privé, parent, public.

Un élément privé n'est accessible que dans le nœud où il est déclaré, un élément ayant la visibilité "parent" sera en plus accessible dans tout nœud qui l'utilise comme sous-nœud. Enfin, un élément public est visible par tous les nœuds qui sont au-dessus de lui dans la hiérarchie AltaRica induite par la relation d'utilisation.

Il y a une analogie évidente avec les langages de programmation orientés objet puisqu'il s'agit de répondre au même problème d'autorisation d'accès. Le langage C++ propose les trois qualificateurs de visibilité privé, protégé, et public. Nous n'avons pas conservé le terme *protégé* car il correspond en C++ à une notion d'héritage alors qu'il correspond en AltaRica à une notion d'utilisation. Du fait qu'il n'existe pas de mécanisme de passage de paramètres en AltaRica, et donc à plus forte raison de mécanisme permettant de passer en paramètre un nœud AltaRica à un autre en dehors de la ré-utilisation statique d'un nœud, un élément "public" sera visible le

long du chemin qui remonte de parent en parent, et ne sera jamais visible à l'extérieur de ce chemin.

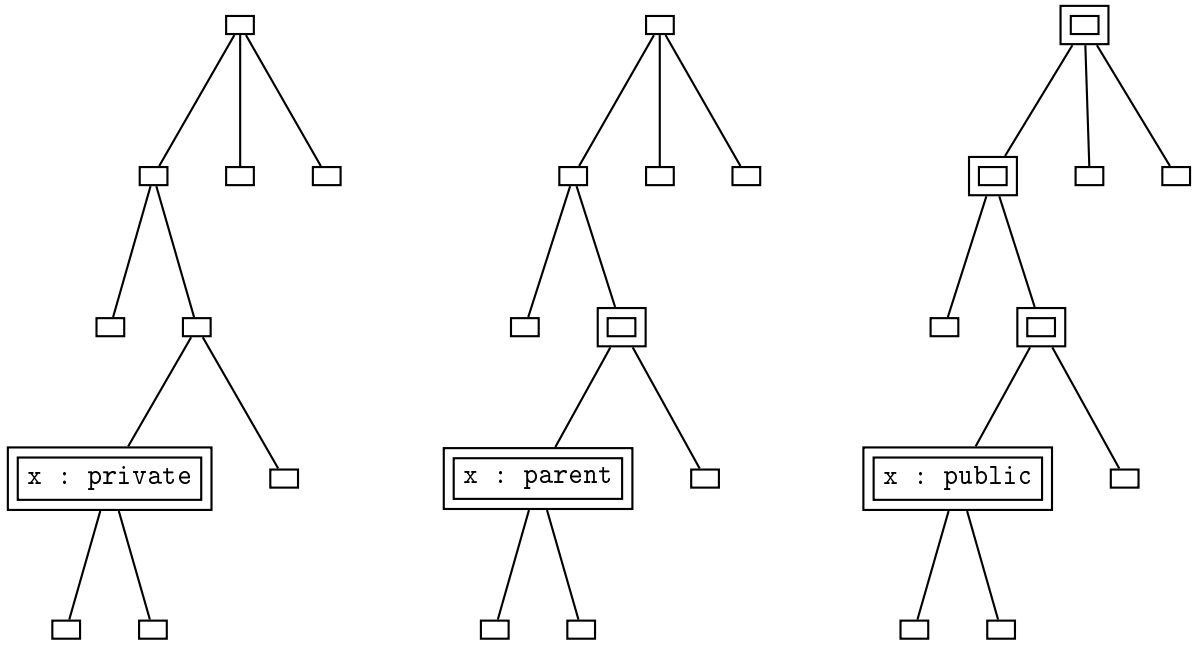


FIG. 4.11 – Les trois qualificatifs de visibilité

Ces règles sont résumées graphiquement dans la figure 4.11. Les nœuds AltaRica sont symbolisés par des rectangles, et la relation d'utilisation d'un nœud par les arêtes. Les nœuds dans lesquels l'élément `x` est visible sont encadrés par un double cadre. L'élement `x` peut être indifféremment une variable d'état ou de flux, ou un événement à l'exception des événements publics qui sont contraints par une règle supplémentaire.

4.5.1 Le cas particulier des événements publics

Les événements publics nécessitent de généraliser un peu la notion de synchronisation d'événements que nous avons présentée, et du point de vue de la visibilité ils sont restreints de la manière suivante : dès qu'un événement public est synchronisé dans un nœud, il cesse d'être public. Un schéma explicatif est donné par la figure 4.12.

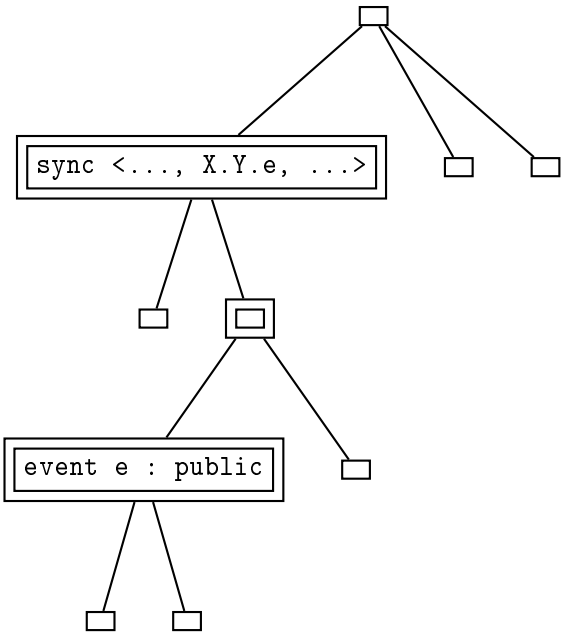


FIG. 4.12 – Propagation des événements publics

Dans l'hypothèse inverse où un événement public continuerait d'être visible après avoir été synchronisé, on aboutirait au paradoxe suivant.

```
node A
    event a : public;
edon

node B
    event b : public;
edon
```

```
node AB
    sub
        A : A;
        B : B;
    sync <A.a, B.b>;
edon
```

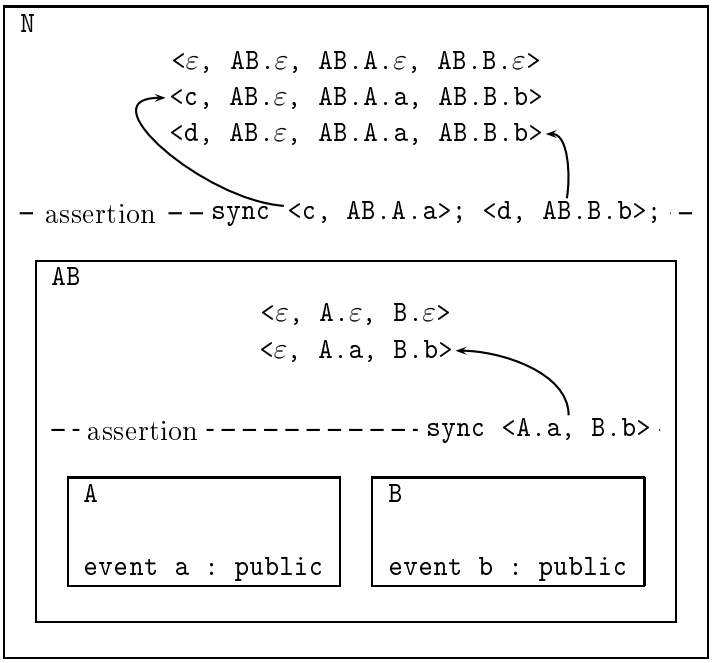
```
node N
    sub AB : AB;
    event c, d;
    sync
        <c, AB.A.a>;
        <d, AB.B.b>;
edon
```

Les synchronisations du nœud `N` imposent que si l'événement `AB.A.a` se produit, alors l'événement `c` se produit aussi. De même, si l'événement `AB.B.b` se produit, alors l'événement `d` se produit en même temps. Mais `a` et `b` étant déjà synchronisés dans le nœud `AB`, cela entraîne que `c` et `d` doivent se produire en même temps. Ceci est impossible car `c` et `d` sont deux événements locaux d'un même nœud AltaRica : on ne peut pas les synchroniser.

Il est toutefois possible de définir une sémantique cohérente pour des événements qui resteraient publics même après avoir été synchronisés une ou plusieurs fois.

La sémantique proposée serait que lorsqu'un événement public est référencé, ce soit en fait tous les vecteurs d'événements dans lesquels il apparaît qui le soient.

Cependant, la confusion qui découle de cette sémantique et que nous illustrons sur l'exemple donné juste avant nous a convaincus de ne pas adopter cette sémantique.



Comme on peut le voir sur cet exemple, à l'intérieur du nœud `N` l'utilisateur a spécifié qu'il synchronisait `c` avec `AB.A.a`. Comme `AB.A.a` n'apparaît nulle part ailleurs dans la description textuelle de ce nœud, il peut s'attendre à ce que toute occurrence de `AB.A.a` soit synchronisée avec `c`.

Mais l'autre vecteur de synchronisation entre `d` et `AB.B.b` cache en fait la synchronisation effectuée au niveau du nœud `AB` et fait que l'événement `AB.A.a` peut se produire sans être synchronisé avec `c`.

Nous pensons qu'il n'est pas souhaitable d'introduire une sémantique qui peut facilement aboutir à une mauvaise interprétation du modèle lorsqu'on en lit une description textuelle, et la règle la plus simple qui permet d'éviter cela consiste à interdire de synchroniser un événement public dans deux nœuds différents, ce qui revient à arrêter de rendre public un événement dès qu'il a été synchronisé.

4.5.2 Visibilités comme réécriture syntaxique

Le fait qu'une variable d'un sous-nœud `A` soit publique ou visible du nœud parent peut se voir comme une permission accordée d'accéder à cette variable dans le sous-nœud, mais peut aussi s'expliquer en imaginant que l'on remonte une image de cette variable dans le nœud parent. C'est cette vision que nous allons adopter ici car elle permet d'expliquer sans ambiguïté les règles de visibilité et de voir clairement l'impact qu'ont les événements publics sur la sémantique du nœud parent.

Objet dans le sous-nœud `A`	*Incarnation de l'objet dans le nœud parent*
`state s : bool : private;`	*Rien de visible*
`flow f : bool : private;`	
`event e : private;`	
`state s : bool : parent;`	`flow` s_A `: bool : private;` `assert` s_A `= A.s;`
`flow f : bool : parent;`	`flow` f_A `: bool : private;` `assert` f_A `= A.f;`
`event e : parent;`	`event` e_A `: private;`
`state s : bool : public;`	`flow` s_A `: bool : public;` `assert` s_A `= A.s;`
`flow f : bool : public;`	`flow` f_A `: bool : public;` `assert` f_A `= A.f;`

`event e : public;`	*si* `A.e` *n'est pas synchronisé dans ce nœud*
	`event` e_A `: public;`
	`sync <`e_A`, A.e>;`
	`trans true |-` e_A `->;`

`event e : public;`	*si* `A.e` *est synchronisé dans ce nœud*
	`event` e_A `: private;`
	`sync <`e_A`, A.e>;`
	`trans true |-` e_A `->;`

4.5.3 Visibilité par défaut et compatibilité ascendante

La possibilité de spécifier les qualificatifs de visibilité est une nouveauté ajoutée au langage AltaRica. Les visibilités par défaut correspondent à l'ancienne version du langage afin de préserver la compatibilité ascendante pour les modèles déjà écrits.

Dans la précédente version du langage, aucun élément n'était public, les variables d'état étaient privées, et les variables de flux et les événements étaient visibles des parents.

Le point de départ de cette extension a été le besoin exprimé par les industriels de rendre les pannes des feuilles visibles jusqu'à la racine de la hiérarchie. Comme les concepts privé/parent existaient déjà sans être nommés explicitement dans les modèles, c'est tout naturellement que ces trois qualificatifs ont été introduits et que leur sens a été défini.

Voici un tableau récapitulatif des règles de visibilité par défaut :

Type d'objet	*Visibilité par défaut*
Etat	privé
Flux	parent
Evénement	parent

4.6 Nouveautés

Le langage AltaRica a subi quelques modifications durant cette thèse, et nous les répertorions ici.

4.6.1 Clause `init`

Il est désormais possible de spécifier les configurations initiales d'un nœud directement dans une section `init` du nœud.

> Les versions précédentes du langage donnaient cette possibilité au travers d'une clause `extern initial_state`, qui était moins bien intégrée au langage.
> Cette modification a été adoptée directement du langage AltaRica DataFlow.

4.6.2 Attributs

Il est désormais possible d'associer à une variable ou à un événement une liste d'attributs, qui sont des identificateurs.

Certains identificateurs sont réservés : `private`, `parent`, et `public`, auxquels nous avons associé les notions de visibilité décrites en 4.5.

> Le langage AltaRica DataFlow propose les attributs `in` et `out` qui permettent de préciser la direction des flux. Nous avons généralisé ce concept par les attributs.
> Les visibilités ont été introduites pour permettre aux modèles AltaRica DataFlow d'être aussi décrits comme des modèles AltaRica. Jusque-là, les visibilités par défaut d'AltaRica DataFlow n'étaient plus compatibles avec celles d'AltaRica.

4.6.3 Syntaxe généralisée pour les priorités

La syntaxe des priorités de la version précédente du langage prêtait à confusion : l'écriture `e1 < e2 < e3` signifiait en fait `e1 < e2` et `e1 < e3`. Nous avons modifié la sémantique de cette écriture afin qu'elle signifie plus naturellement `e1 < e2` et `e2 < e3`.

Nous avons profité de cette modification de la sémantique pour généraliser l'écriture des priorités. On peut désormais faire apparaître un sous-ordre partout où pouvait apparaître un événement dans une priorité. On peut par exemple écrire : `e1 < { e2 > e3 } > e4` pour spécifier l'ordre `e1 < e2`, `e1 < e3`, `e3 < e2`, `e4 < e2`, `e4 < e3`.

Mec V peut sauvegarder l'ordre partiel des priorités au format dot ; pour l'exemple ci-dessus, on obtient l'ordre donné à la figure 4.13. Le sommet vide représente l'événement ε, qui est toujours incomparable aux autres événements.

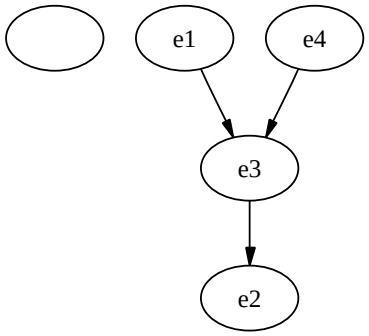


FIG. 4.13 – Ordre partiel généré par Mec V pour `e1 < { e2 > e3 } > e4`

Ce changement constitue une modification de la sémantique des priorités. Les anciens modèles AltaRica qui contiennent des comparaisons d'événements en cascade doivent être modifiés afin de retrouver leur ancienne sémantique.

4.6.4 Vecteurs de diffusion restreinte

Les vecteurs de diffusion cherchent à maximiser le nombre d'événements apparaissant des les vecteurs d'événements qui s'en déduisent (cf. "Sémantique de la diffusion", page 33). Il est parfois utile d'avoir le comportement inverse. Nous avons ajouté au langage AltaRica les mots-clés `min` et `max` qui, lorsqu'ils sont situés après un vecteur de diffusion, permettent de sélectionner l'un ou l'autre des deux comportements. En l'absence de ces mots-clés, c'est l'ancien comportement qui prévaut.

Chapitre 5

Implémentation de Mec V

Mec V manipule en interne plusieurs sortes d'objets. La plus centrale est la notion de relation, qui est encodée pour les calculs par des BDDs. Ces relations peuvent provenir principalement de deux sources : un nœud AltaRica qui définit sa relation de transition et son ensemble d'états initiaux, ou un système d'équations de points fixes.

Nous allons voir dans ce chapitre la façon dont nous avons implémenté le module de gestion de BDDs, puis les algorithmes utilisés pour construire et exploiter ces BDDs à partir des données fournies par l'utilisateur, sous forme de modèle AltaRica ou de système d'équations de points fixes.

> Nous avons écrit Mec V en langage C ANSI 89, à très peu d'exceptions près :
> – la fonction getopt() est utilisée (standard POSIX 1003.2)
> – la fonction sigsetjmp() est utilisée (standards POSIX 1003.1 ou ISO C99)
> – la fonction vsnprintf() est utilisée (BSD, ou standards ISO C99 ou UNIX98)
> Cependant, ces fonctions font partie du 'folklore UNIX', et sont disponibles sur énormément de systèmes dits compatibles POSIX.
> D'autres extensions ou bibliothèques sont utilisées lorsqu'elles sont disponibles, comme par exemple le type "long long" et la bibliothèque "readline".

5.1 Architecture générale

Mec V est constitué de modules qui ont beaucoup d'interactions entre eux. Cependant, le but de ce chapitre n'est pas d'être un manuel de maintenance pour Mec V, et nous laissons donc de côté beaucoup de détails importants mais de trop bas niveau dans cette description. La figure 5.1 représente quelques-unes des interactions qui peuvent exister entre différents modules, afin de donner une vue globale.

A l'exception des sommets AltaRica, Spec, et UI, chacun des sommets contient le nom d'un ou plusieurs modules. Les sommets AltaRica et Spec regroupent en fait conceptuellement l'analyse lexicale, l'analyse syntaxique, l'exploitation de l'arbre d'analyse syntaxique et l'écriture respectivement de fichiers AltaRica et de fichiers de spécification.

La fonctionnalité "UI" (User Interface, interface utilisateur) se décompose en plusieurs autres modules d'interface texte ou graphique.

Mec V contient aussi toute une série de petits modules utilitaires qui seront juste évoqués.

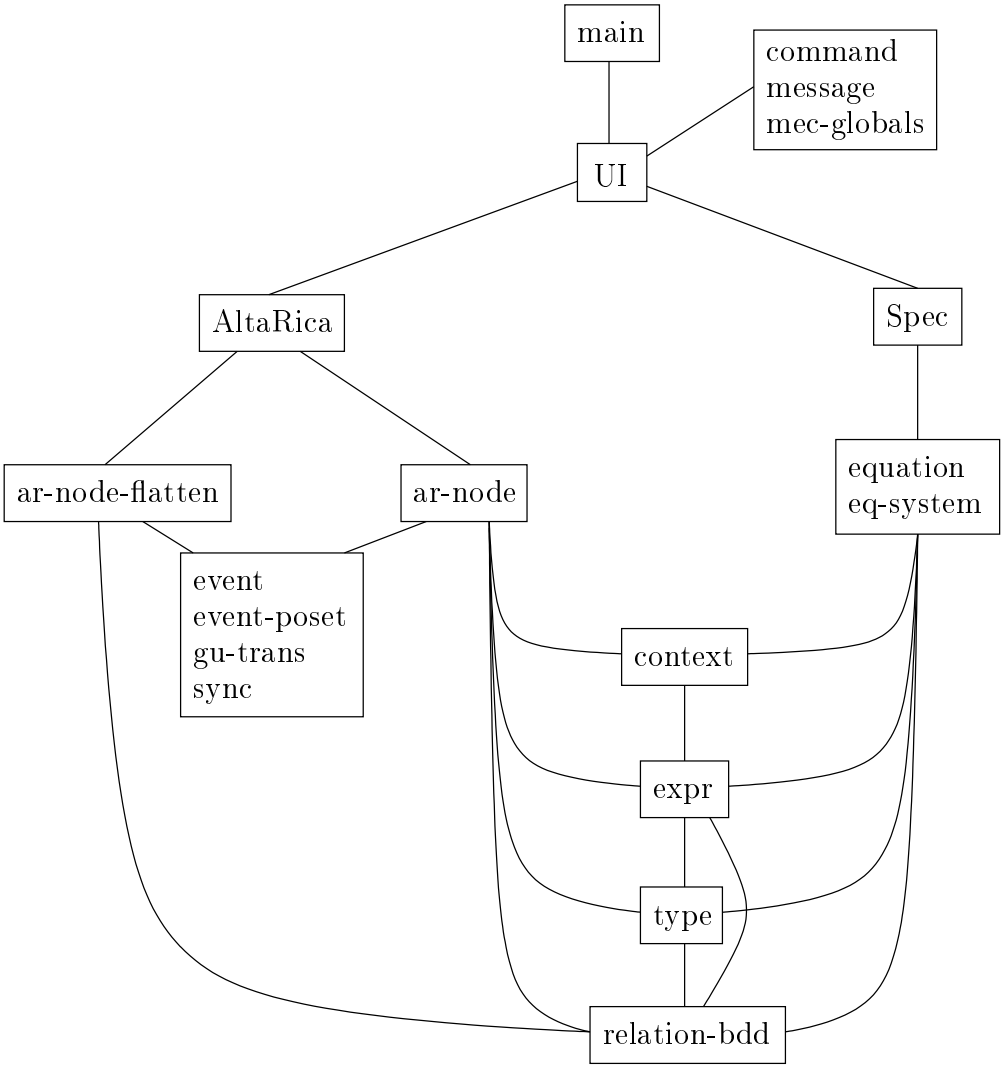


FIG. 5.1 – Architecture générale de Mec V

5.2 Le module BDD

Il existe de nombreux modules de gestion de BDDs mais nous avons choisi d'implémenter le nôtre pour plusieurs raisons. Il y a d'abord un intérêt pédagogique évident à cette démarche qui permet de comprendre où se situent les difficultés d'implémentation. Ensuite, bien que les BDDs aient apporté au monde de la vérification une grande amélioration dans la taille des systèmes traitables, il reste de nombreuses pistes à explorer, en particulier dans l'utilisation de structures hybrides mélant BDDs et polyèdres par exemple. L'expérimentation de ces techniques est bien plus aisée si on maîtrise déjà parfaitement la base sur laquelle elles sont construites.

Cependant, nous avons aussi clairement défini l'interface vers ce module afin qu'il soit aisé d'y connecter un autre module de gestion de BDDs si le besoin s'en faisait sentir.

Nous allons dans cette section focaliser sur quelques aspects concernant l'implémentation du module, et nous vous référons à la section 3.5 page 16 pour un exposé plus général concernant les diagrammes de décisions binaires.

5.2.1 La table d'unicité

Le module BDD se compose assez classiquement d'une table d'unicité, qui garantit la propriété essentielle d'avoir un représentant canonique pour chaque BDD. Ce service est rendu par une fonction dont le rôle est, étant donné un BDD, de retourner le BDD canonique correspondant. Il s'agit souvent de retourner un pointeur sur un BDD ayant les mêmes composantes (niveau, BDD, BDD), mais qui est déjà l'unique élément d'une table de hachage ayant ces composantes.

Il est ainsi garanti qu'à un BDD correspond un unique pointeur.

Cependant, cette fonction ne se contente pas d'une simple recherche dans une table de hachage : elle peut aussi retourner un BDD ayant la même sémantique mais codé de manière différente. C'est le cas avec les BDDs qui contiennent des arcs négatifs et qui nécessitent une contrainte de canonicité supplémentaire, comme cela est décrit un peu plus loin.

Cette table grossit de manière très importante lors des calculs, et dans beaucoup de situations les résultats intermédiaires ne sont plus utilisés par la suite. Nous avons donc implémenté, comme cela se fait classiquement, un ramasse-miettes. Tout BDD renvoyé par le module BDD doit être référencé par l'appelant afin d'assurer que ce BDD ne soit pas détruit. A tout moment, le ramasse-miettes peut se déclencher et il entraine dans ce cas la libération de tous les nœuds qui ne sont référencés ni à l'extérieur du module ni par le cache de termes. Il s'agit d'un ramasse-miettes de type "marque et efface", dont on peut trouver une description dans l'ouvrage [32].

5.2.2 Le cache de termes

Le deuxième élément essentiel du module est le cache de termes, qui évite d'effectuer plusieurs fois le même calcul. Il s'agit d'un tableau contenant des "termes", qui associent à un triplet (opérateur, BDD, BDD) le BDD canonique résultant de l'application de l'opérateur binaire aux deux BDDs. Le cache de termes est sans doute le facteur prédominant dans l'efficacité des calculs sur les BDDs.

Le cache de termes a une taille maximale fixée à la compilation et les termes sont ajoutés dans le cache sans se préoccuper de savoir si la place est déjà prise ou non par un autre terme. Il serait intéressant de déterminer si le surcoût engendré par une structure de données un peu plus élaborée mais permettant de choisir les termes à élaguer permet d'améliorer les performances de notre module dans la pratique.

5.2.3 Amélioration : les arcs négatifs

Le calcul de la négation d'un BDD est linéaire dans la taille du BDD. Il est possible de rendre ce temps de calcul constant en utilisant des "arcs négatifs".

L'utilisation des arcs négatifs était une amélioration naturelle à apporter à notre module de gestion de BDDs car le cache de termes étant prévu uniquement pour des opérateurs binaires par souci d'homogénéité, la négation était implémentée comme un *ou exclusif* avec le BDD *vrai*. L'utilisation des arcs négatifs permet d'éviter le calcul de ce *ou exclusif* et donc de ne pas polluer le cache de termes.

Le pointeur vers un BDD peut être étiqueté de manière à signifier qu'il faut interpréter ce BDD comme sa propre négation. Ce sont ces pointeurs étiquetés de cette manière qui sont appelés "arcs négatifs".

L'utilisation de tels pointeurs permet de se passer d'un des deux nœuds de base *vrai* ($\top$) ou *faux* ($\bot$). Nous avons choisi dans notre implémentation de garder $\bot$. Nous représentons un BDD pointé négativement en le faisant précéder d'une étoile (*) dans ce qui suit. Le BDD *vrai* sera donc représenté $*\bot$.

Prenons l'exemple d'une variable x et du BDD codant que cette variable est vraie : $(x?\ \top : \bot)$. Le même BDD avec arcs négatifs se code donc $(x?\ *\bot : \bot)$.

Le BDD représentant sa négation $(x?\ \bot : \top)$ se calcule de manière inductive sur le BDD en prenant la négation de tous les descendants. Dans le cas des arcs négatifs, on l'écrit tout naturellement $*(x?\ *\bot : \bot)$.

Canonicité des BDDs avec arcs négatifs

Si l'on n'y prend pas garde, l'introduction d'arcs négatifs fait perdre la propriété importante de canonicité des BDDs. En effet, dans l'exemple précédent, on aurait pu coder le fait que x est fausse avec le BDD suivant : $(x?\ \bot : *\bot)$.

En fait, en regardant de plus près, pour tout BDD $(x?\ b_1 : b_2)$, le BDD $*(x?\ *b_1 : *b_2)$ lui est équivalent. La règle que nous choisissons pour rendre un BDD avec arcs négatifs canonique est que sa branche "*then*" soit toujours pointée positivement.

Etant donnés une variable de niveau x et les deux BDDs b_1 et b_2, la figure 5.2 résume les différents cas possibles avec ces BDDs et donne l'équivalent canonique pour chacun d'eux.

BDD	BDD canonique équivalent
$*(x?\ *b_1 : *b_2)$	$(x?\ b_1 : b_2)$
$*(x?\ *b_1 : b_2)$	$(x?\ b_1 : *b_2)$
$(x?\ *b_1 : *b_2)$	$*(x?\ b_1 : b_2)$
$(x?\ *b_1 : b_2)$	$*(x?\ b_1 : *b_2)$

FIG. 5.2 – BDDs avec arcs négatifs et leur équivalent canonique

5.2.4 L'ordre sur les variables et l'égalité

Il est bien connu que l'ordre imposé sur les variables d'un BDD a un impact essentiel sur la taille des BDDs. Cela est particulièrement visible dans le cas de l'égalité.

Donnons-nous 6 variables booléennes x_1, x_2, x_3, y_1, y_2, et y_3 représentant respectivement deux variables $x_1x_2x_3$ et $y_1y_2y_3$ codées sur trois bits.

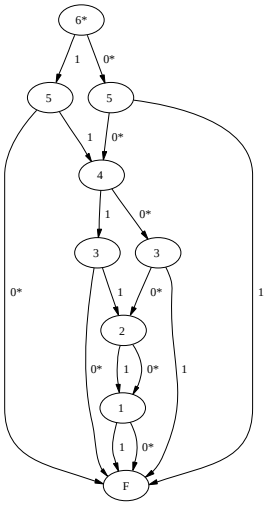


FIG. 5.3 – BDD représentant l'égalité entre $x_1x_2x_3$ et $y_1y_2y_3$, avec l'ordre $x_1 \leq y_1 \leq x_2 \leq y_2 \leq x_3 \leq y_3$

Le BDD qui code l'égalité entre $x_1x_2x_3$ et $y_1y_2y_3$ dépend énormément de l'ordre des variables. Si l'ordre est $x_1 \leq y_1 \leq x_2 \leq y_2 \leq x_3 \leq y_3$, on obtient le BDD de la figure 5.3 alors que si l'ordre est $x_1 \leq x_2 \leq x_3 \leq y_1 \leq y_2 \leq y_3$, on obtient le BDD de la figure 5.4.

De manière générale, coder l'égalité entre deux variables peut être soit linéaire soit exponentiel en le nombre de bits qui les composent. En effet, dans le premier cas, on établit l'égalité bit par bit, alors que dans le deuxième cas, le BDD contient tous les chemins correspondant aux valeurs possibles de la première variable, qui aboutissent aux chemins égaux de l'autre variable (qui eux bénéficient quand même du partage des suffixes communs). Ce phénomène se voit bien sur les figures 5.3 et 5.4.

De nombreux algorithmes d'ordonnancement statique de variables existent dans la littérature, mais aucun n'est encore implémenté dans Mec V. Ces algorithmes sont basés sur des heuristiques qui prennent en compte le type des objets codés par les BDDs, le cas le plus fréquemment étudié étant le cas des circuits booléens. On peut citer par exemple des articles historiques sur le sujet comme par exemple [25] et [31], ou un article plus récent comparant plusieurs heuristiques [1]. Une étude très détaillée des problèmes d'explosion combinatoire des BDDs se trouve dans [40].

5.2.5 Gestionnaires de BDDs

Les variables booléennes manipulées par un utilisateur sont souvent cloisonnées en plusieurs ensembles qui ne seront jamais mis en relation. Plutôt que d'assigner à l'intérieur des BDDs un niveau différent à toutes ces variables, on utilise un *gestionnaire de BDDs*, par l'intermédiaire duquel on peut coder des BDDs mettant en relation un sous-ensemble de toutes les variables booléennes étudiées.

Concrètement, dans Mec V, il faut imaginer qu'il existe un gestionnaire de BDDs par type de donnée : chaque nœud AltaRica fournit par exemple un gestionnaire de BDDs pour effectuer des calculs sur sa relation de transition.

Certaines fonctions sur les BDDs n'ont de sens qu'entre BDDs appartenant au même ges-

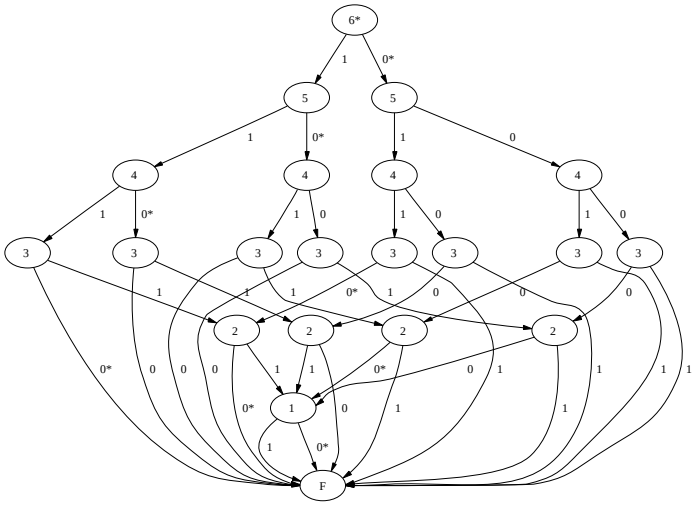


FIG. 5.4 – BDD représentant l'égalité entre $x_1x_2x_3$ et $y_1y_2y_3$, avec l'ordre $x_1 \leq x_2 \leq x_3 \leq y_1 \leq y_2 \leq y_3$

tionnaire (la réunion, l'intersection, la projection, ...). D'autres fonctions essentielles comme la substitution permettent de substituer des BDDs appartenant à un gestionnaire donné aux nœuds d'un BDD appartenant à un autre gestionnaire. C'est grâce à cette opération que peut s'effectuer l'application d'une relation, qui est définie dans son gestionnaire, à ses arguments, qui appartiennent tous à un même autre gestionnaire.

Le rôle essentiel d'un gestionnaire de BDDs est de coder l'ordre dans lequel les nœuds seront agencés dans les BDDs qu'il gère. En effet, les variables booléennes apparaissant dans les BDDs sont référencées par l'utilisateur avec des indices qui sont pratiques pour lui, mais leur codage interne ne respecte en fait pas cet ordre pour des raisons d'efficacité.

Dans [23], les auteurs montrent que le BDD codant la relation de transition d'un système de transitions obtenu par composition parallèle n'augmente que linéairement à condition d'entrelacer les variables représentant chacune des deux composantes de la relation.

En s'appuyant sur cet article, et compte tenu des observations faites sur l'ordre des variables à la sous-section précédente, Mec V entrelace les différents composantes d'un type lorsque celui-ci est composite.

Si par exemple on crée un gestionnaire de BDDs correspondant aux relations ternaires dont chaque paramètre est codé sur deux bits ($x_1x_2y_1y_2z_1z_2$), l'ordonnancement choisi par le gestionnaire de BDDs sera le suivant, résumé dans ce tableau :

Variable :	x_1	x_2	y_1	y_2	z_1	z_2
Indice externe :	0	1	2	3	4	5
Indice interne :	0	3	1	4	2	5
Ordre correspondant :	x_1	y_1	z_1	x_2	y_2	z_2

La gestion de l'ordre par le gestionnaire de BDDs permet de conserver la propriété que les indices (externes) vus par les utilisateurs du gestionnaire sont contigus pour une même variable : x_1x_2 a pour indices externes 0 et 1, etc. Ceci permet de décrire les indices qui codent une variable simplement par un intervalle.

5.3 Les vecteurs de BDDs

Un BDD code une application de $\mathbb{B}^m$ dans $\mathbb{B}$, où m est le nombre de variables (niveaux) du BDD. Pour représenter des calculs sur des domaines finis d'entiers, il est possible d'utiliser n BDDs, où n est le nombre de bits nécessaires pour représenter les entiers que ces calculs peuvent valoir.

Chacun de ces BDDs sera la contrainte qui décide quand le bit qui lui correspond dans la représentation binaire du calcul est positionné ou pas.

L'utilisation de vecteurs de BDDs pour coder des entiers a été par exemple utilisée dans le package BuDDy [36].

Pour une formule booléenne ϕ, nous allons noter $[\phi]$ l'entier qui vaut 0 si ϕ est fausse et 1 si ϕ est vraie. En général, ϕ dépend de variables, et donc $[\phi]$ aussi ; cependant, ces variables ne jouent aucun rôle explicite dans les calculs sur les vecteurs et nous omettons donc de les citer.

Un vecteur de BDDs représente donc un ensemble de valeurs possibles (qui dépendent des variables des formules qui le composent), que nous noterons

$$\sum_{i=0}^{n}[\phi_i]2^i$$

5.3.1 L'addition

$$\sum_{i=0}^{n}[\phi_i]2^i + \sum_{i=0}^{n}[\psi_i]2^i = \sum_{i=0}^{n}([\phi_i] + [\psi_i])2^i$$

Cette façon d'écrire la somme ne nous permet pas de déduire immédiatement un algorithme pour calculer le vecteur de BDDs résultat. En effet, la somme $[\phi_i] + [\psi_i]$ peut valoir 0, 1, ou 2 et ne peut donc pas prendre telle quelle une forme $[\rho_i]$.

De la même manière que lorsque l'on fait l'addition à la main, nous allons ici aussi introduire la notion de retenue. La seule différence est que cette retenue est symbolique : on ne doit jamais supposer à un point donné de l'algorithme que l'on connaît sa valeur.

Remarquons d'ores et déjà que s'il existe une valuation des ϕ_i, ψ_i telle qu'elles s'évaluent toutes à vrai, la somme des deux vecteurs vaudra au plus $2\sum_{i=0}^{n} 2^i$, i.e. $2(2^{n+1} - 1)$ qui est strictement inférieur à 2^{n+2}.

Le résultat que nous cherchons tient donc sur $n + 2$ bits, et nous choisissons de l'écrire $\sum_{i=0}^{n+1}[\rho_i]2^i$.

Rappelons que l'idée de la retenue est de propager une quantité qui ne tient pas dans la base fixée au coefficient du monôme de degré supérieur. La condition pour qu'il y ait ici une retenue est donc que $[\phi_i] + [\psi_i]$ soit strictement supérieur à 1. Appelons $[c_i]$ la retenue générée par le monôme de degré i. Cette retenue va elle aussi contribuer à augmenter (d'au plus 1) le coefficient du monôme de degré $i + 1$ et la condition devient donc $[\phi_i] + [\psi_i] + [c_{i-1}] \geq 2$, ce qui est le cas pour une valuation donnée si au moins deux parmi les trois formules ϕ_i, ψ_i, c_{i-1} s'évaluent à vrai.

Grâce à cette analyse, on peut donc définir la valeur qu'aura la retenue :

$$\begin{cases} c_{-1} = & \bot \\ c_{i+1} = & (\phi_i \wedge \psi_i) \vee (c_i \wedge \phi_i) \vee (c_i \wedge \psi_i),\ 0 \leq i \leq n \end{cases}$$

Et nous disposons donc de tout ce qu'il faut pour calculer le résultat :

$$\rho_i = \phi_i \oplus \psi_i \oplus c_i,\ 0 \leq i \leq n+1$$

Ceci en posant $\phi_{n+1} = \phi_n$ et $\psi_{n+1} = \psi_n$ (extension de signe si les vecteurs de BDDs sont codés en complément à deux).

Exemple 5.1 *Calculons par exemple le vecteur de BDDs associé à l'expression $x+1$, où x est une variable codée avec trois bits x_0, x_1, et x_2; x_0 étant le bit de plus faible poids.*

$$\boxed{x_2}\boxed{x_1}\boxed{x_0} + \boxed{\bot}\boxed{\bot}\boxed{\top}$$

Le résultat tiendra donc sur quatre bits et on procède en partant du bit de plus faible poids.

$\rho_0 =$	$x_0 \oplus \top \oplus \bot$	$=$	$\neg x_0$
$c_0 =$	$(x_0 \wedge \top) \vee (\bot \wedge x_0) \vee (\bot \wedge \top)$	$=$	x_0
$\rho_1 =$	$x_1 \oplus \bot \oplus x_0$	$=$	$x_1 \oplus x_0$
$c_1 =$	$(x_1 \wedge \bot) \vee (x_0 \wedge x_1) \vee (x_0 \wedge \bot)$	$=$	$x_0 \wedge x_1$
$\rho_2 =$	$x_2 \oplus \bot \oplus (x_0 \wedge x_1)$	$=$	$x_2 \oplus (x_0 \wedge x_1)$
$c_2 =$	$(x_2 \wedge \bot) \vee (x_0 \wedge x_1 \wedge x_2) \vee (x_0 \wedge x_1 \wedge \bot)$	$=$	$x_0 \wedge x_1 \wedge x_2$
$\rho_3 =$	$x_2 \oplus \bot \oplus (x_0 \wedge x_1 \wedge x_2)$	$=$	$x_2 \oplus (x_0 \wedge x_1 \wedge x_2)$

Voici donc le vecteur d'expressions booléennes représentant le calcul $x+1$:

$$\boxed{x_2 \oplus (x_0 \wedge x_1 \wedge x_2)}\boxed{x_2 \oplus (x_0 \wedge x_1)}\boxed{x_1 \oplus x_0}\boxed{\neg x_0}$$

Ajouter 1 à un nombre en écriture binaire consiste à modifier tous ses bits en partant du bit de poids faible et jusqu'à rencontrer un '1'. On retrouve bien l'expression de cette astuce connue dans le vecteur qui représente le calcul de $x+1$.

5.3.2 Gestion des nombres négatifs

Nous codons les nombres négatifs en complément à deux. Comme il est possible d'effectuer des opérations sur des vecteurs de BDDs qui représentent des calculs sur des domaines de tailles différentes, on ne doit pas fixer le nombre de bits utilisés pour représenter un calcul.

On fait donc en sorte à tout moment que la formule booléenne de plus fort poids puisse servir à déterminer le signe du vecteur ($\top$ signifiant que le nombre est négatif). Par exemple, la constante 1 ne se codera pas $\boxed{\top}$ mais $\boxed{\bot}\boxed{\top}$ puisqu'elle est positive.

De plus, le calcul de l'opposé d'un nombre contient une opération qui revient à ajouter un au vecteur. Cette addition risque de nécessiter un bit supplémentaire pour stocker le résultat, mais il n'est pas souhaitable d'augmenter de un la taille d'un vecteur à chaque fois qu'on en calcule l'opposé. En effet, si on faisait cela, le calcul de $--x (= x)$ donnerait un vecteur contenant deux BDDs inutiles. Nous avons donc choisi d'allouer suffisamment de BDDs dans chaque vecteur pour garantir qu'il y a suffisamment d'espace pour coder les valeurs de ce vecteur *et leurs opposés*.

Ainsi par exemple, la constante -1 ne se codera pas $\boxed{\top}$ mais $\boxed{\top}\boxed{\top}$ puisque sa négation est $\boxed{\bot}\boxed{\top}$.

5.4 Types et expressions

5.4.1 Les types élémentaires

On distingue dans Mec V comme en AltaRica les *types* des *domaines*. Deux variables ayant des types différents ne peuvent pas être comparées ensemble alors que deux variables appartenant à des domaines sur le même type peuvent l'être : un domaine est une restriction d'un type. Les liens entre types et domaines sont résumés plus loin sur la figure 5.5.

Les domaines de base disponibles dans Mec V sont les booléens, les intervalles finis d'entiers, les énumérations, les énumérations partagées et les entiers.

Les entiers ne sont utilisables que dans des expressions constantes, et ne peuvent donc pas être utilisés comme paramètres formels d'une relation. Cela s'explique naturellement par l'utilisation des BDDs comme structure de données, puisque ceux-ci ne peuvent manipuler que des variables ayant des domaines finis.

En AltaRica, seules les énumérations *partagées* sont visibles de l'utilisateur. Nous les appelons partagées car les ensembles de constantes qui les définissent sont inclus dans un ensemble commun de constantes. Ainsi, les deux énumérations partagées `{ a, b }` et `{ b, c }` ont en commun la constante `b`. En particulier, il est possible de comparer deux variables ayant pour domaine des énumérations partagées différentes : l'égalité sera vraie sur l'intersection des deux domaines.

Les énumérations (non-partagées) sont en revanche des *types* différents. Il n'est pas possible de comparer deux variables ayant pour type deux énumérations différentes, même si celles-ci contiennent les mêmes noms de constantes. Ces énumérations-là sont utilisées pour représenter les noms des événements.

5.4.2 Les types construits

Le type construit majeur est la *relation*, qui est construite à partir d'une séquence de types qui ne peut contenir de relation. Le type relation permet de représenter les éléments du produit cartésien des types qui le composent.

Les deux autres types construits sont les types des *configurations* et des *vecteurs d'événements* des nœuds AltaRica. Ceux-ci sont détaillés dans la sous-section 5.5.7.

5.4.3 Evaluation des expressions

L'évaluation des expressions est un problème important dans Mec V car elle intervient à de nombreux endroits dans les modèles AltaRica (gardes, affectations, assertions, conditions initiales), comme dans le langage de spécification (systèmes d'équations).

Nous avons respecté l'objectif de modularité que nous nous étions fixé pour Mec V, et avons donc fait le choix de n'avoir qu'un seul module d'évaluation d'expressions, suffisamment générique pour pouvoir être utilisé dans le cas (restreint) des expressions AltaRica et dans le cas plus général des expressions du premier ordre que l'on peut rencontrer dans une spécification.

Grâce aux primitives dont nous disposons sur les BDDs (union, intersection, projection, ...) et sur les vecteurs de BDDs (addition, négation, ...), l'évaluation d'une expression ne présente pas de difficulté majeure. La difficulté d'implémentation réside dans la liaison des symboles apparaissant dans une expression, et nous évoquons ce sujet dans la sous-section qui suit.

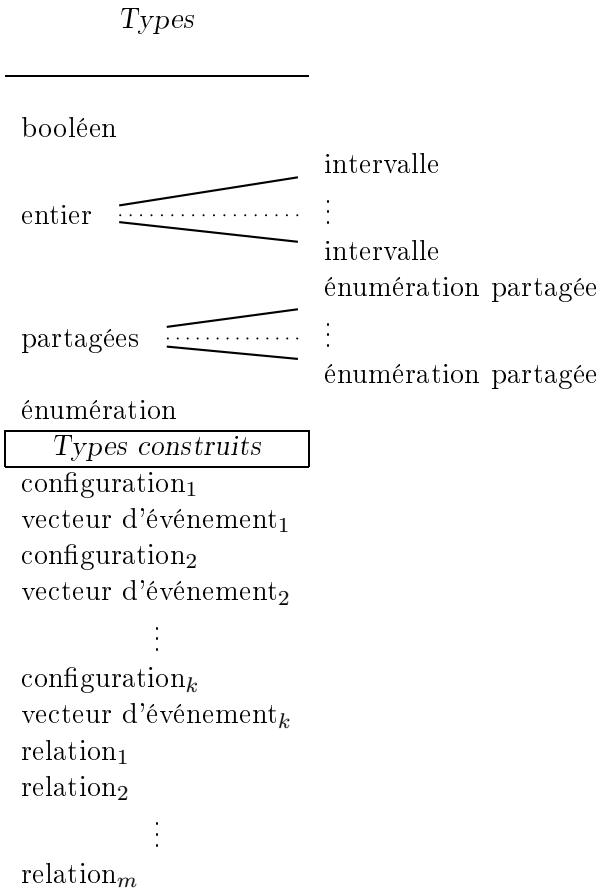


FIG. 5.5 – Les domaines dans Mec V

5.4.4 Liaison des symboles et inférence de types

Il est important dans Mec V de ne pas garder les symboles sous leur forme textuelle, mais de les *lier* à l'objet qu'ils représentent le plus tôt possible. En effet, du fait de l'existence d'un environnement global, deux occurences de la même expression évaluées à deux moments différents pourraient s'évaluer différemment.

> Ceci n'est pas souhaitable car ce cas se présente pour l'assertion d'un nœud, qui sont propagées dans les assertions de tout nœud utilisant ce nœud. Entre les définitions de ces deux nœuds, l'environnement global peut avoir été modifié et cela ne doit pas changer la sémantique des nœuds pour autant.

Un symbole `x` apparaissant dans une expression peut provenir des endroits suivants :
- une constante globale, `const x = 1;` dans un fichier AltaRica ou `x := 1;` dans une spécification
- une constante énumérée partagée apparaissant dans un domaine énuméré, `{ x, y, z }`
- une constante énumérée, `event x;` dans un nœud AltaRica
- une variable de nœud AltaRica, `flow x : bool`
- une variable introduite par un quantificateur du premier ordre, `<x>R(x)`
- un paramètre formel de relation, `R(x) := x = 2;`

Nous utilisons la notion de *contexte*, qui permet de lier un nom à l'objet qu'il représente. Un nœud AltaRica ou un système d'équations fournissent un contexte dans lequel les symboles d'une expression peuvent être liés.

L'existence de constantes énumérées (non partagées) rend la liaison des symboles délicate puisqu'il faut d'abord inférer l'énumération à laquelle appartient le symbole avant de pouvoir le lier.

Nous avons ajouté dans Mec V un mécanisme d'inférence de types qui résoud ce problème. Ce mécanisme est aussi très utile pour l'utilisateur puisqu'il le dispense de préciser la plupart des types des variables qu'il introduit comme paramètres de relation ou dans les quantificateurs du premier ordre. L'inférence de types consiste dans Mec V à parcourir une ou plusieurs expressions (dans le cas des systèmes d'équations) et à constituer lors de ces parcours des classes d'équivalence de types, en fonction des types des opérateurs et des relations utilisés. Une fois les classes d'équivalence constituées, il reste à vérifier qu'elles ne contiennent pas d'éléments incompatibles et à typer les éléments non encore typés avec le type de leur classe.

Ce mécanisme est itéré jusqu'à stabilisation car certains types nécessitent pour être connus que des symboles aient déjà été liés. Il est donc nécessaire d'alterner les phases de liage et de propagation de type.

Exemple 5.2 *Prenons l'exemple d'un nœud AltaRica* `A` *dont nous voulons sélectionner les vecteurs d'événements qui ont pour événement local l'événement* `ecris`*. Ceci s'écrit dans le langage de spécification de Mec :*

```
R(e : A!ev) := e. = ecris;
```

En paraphrasant, ceci définit une relation `R` *prenant un paramètre (*`e`*) de type* vecteur d'événements *du nœud* `A` *dont les éléments satisfont la contrainte que la composante locale du vecteur (*`e.`*) vaut* `ecris`*.*

Avant l'évaluation de l'expression `e. = ecris`*, l'inférence de types découvre que le type de* `e.` *doit être le même que celui de* `ecris`*. Ce n'est qu'une fois que* `e` *est lié que l'on en connaît le type (*`A!ev`*). La sous-expression* `e.` *a donc le type des événements locaux du nœud* `A` *et on peut alors de nouveau propager les types pour trouver que* `ecris` *est le nom d'un événement du nœud* `A`*. Ce qui permet de le lier à sa valeur.*

Pour une expression donnée, l'algorithme fonctionne comme suit :

1. Les équivalences entre les types apparaissant dans l'expression sont rassemblées dans une structure de données appelée "type_system"
2. Lors de ce passage, tous les types qui n'étaient pas connus donnent lieu à la création d'une petite structure de données "expr_fixup" qui contient suffisamment d'information pour pouvoir modifier le type de l'expression qu'il décrit lorsque son type aura été inféré
3. Le type_system est "résolu" : les équivalences accumulées sont utilisées pour remplir les expr_fixup avec les informations de types qui peuvent être inférées.
4. Chaque expr_fixup est "résolu" ; trois cas se présentent :
   (a) le type a été inféré et le symbole est lié à la variable qu'il représente
   (b) le type a été inféré, mais le symbole spécifiait une sous-variable (cas d'un type composite comme une configuration d'un nœud AltaRica), dans ce cas, le expr_fixup est modifié pour refléter le travail qui reste à faire

(c) le type n'a pas été inféré

5. Le processus est réitéré à partir de l'étape 3 tant que au moins un expr_fixup est tombé dans l'un des deux premiers cas énumérés ci-dessus.

A chaque boucle de cet algorithme, au moins un expr_fixup s'est trouvé dans le cas 4.(a) ou 4.(b). Si c'est le cas 4.(a), l'expression a été corrigée, et son expr_fixup est marqué comme tel et ne pourra plus jamais contribuer à modifier le système de type. Si c'est le cas 4.(b), le expr_fixup aura évolué pour descendre dans la hiérarchie représentée par le symbole, et comme cette hiérarchie est finie, ce expr_fixup ne pourra pas faire changer le système de types un nombre infini de fois.

Enfin, le nombre de expr_fixup est au plus égal au nombre de nœuds représentant des symboles dans une expression, et il est donc fini. Ceci finit donc la preuve de terminaison de cet algorithme.

5.5 Représentation des éléments d'un nœud AltaRica

Nous allons illustrer la façon dont les éléments qui composent un nœud AltaRica sont stockés en mémoire en nous basant sur le nœud `Simple` défini dans la figure 4.3, page 29, que nous reproduisons ici.

```
node Simple
    state v : bool;
    event e;
    trans
        ~v |- e -> v := true;
edon
```

5.5.1 Variables

Chaque variable AltaRica est représentée par un "context_slot" (un emplacement) qui décrit son nom, son type, un drapeau permettant de distinguer les variables de flux, et un indice. L'indice d'une variable est un entier qui identifie de manière unique cette variable à l'intérieur du nœud dans lequel elle est définie.

La connaissance du type de la variable permet à elle seule de calculer l'espace nécessaire au stockage des valeurs que peut prendre la variable.

Voici par exemple la description de la variable `v` du nœud `Simple` :

Variable `v`	
nom :	`"v"`
type :	`bool`
drapeaux :	*aucun*
indice :	0

L'introduction des indices permet d'accéder rapidement aux informations pertinentes d'une variable mais ce n'est pas là leur principal intérêt.

Les indices doivent être vus comme les instances des variables, et ils permettent de distinguer entre plusieurs variables apparaissant dans deux sous-nœuds ayant le même type. Ce dernier

aspect mérite qu'on s'y attarde et va être détaillé sur l'exemple 5.3 page 56, où l'on se donne un nœud `SynchroParAssert` qui synchronise deux nœuds `Simple` grâce à son assertion.

> Ces indices simplifient aussi l'écriture de petits algorithmes annexes. On pourrait imaginer un algorithme qui souhaite vérifier si toutes les variables d'un nœud donné sont utilisées à un moment ou à un autre. Il suffit pour cela d'allouer un tableau de bits et d'utiliser les indices des variables comme indices dans ce tableau de bit, chaque bit codant le fait qu'une variable a été utilisée.

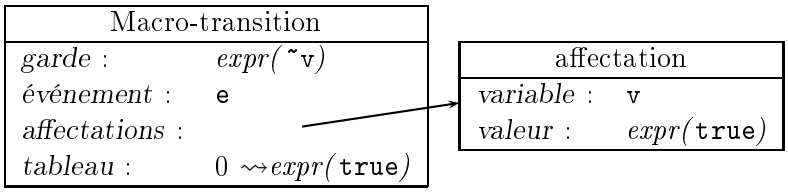
5.5.2 Macro-transitions

Les macro-transitions sont représentées par des structures qui servent principalement de stockage : dès que des calculs ont lieu qui permettent de calculer le BDD représentant cette macro-transition, ils sont répercutés et stockés dans l'événement correspondant à cette macro-transition.

Une macro-transition est constituée d'une garde (sous la forme d'une *expression* symbolique), d'une référence à l'événement lié à cette macro-transition, et de la liste des affectations qui doivent être effectuées lorsque la macro-transition est tirée.

Les affectations sont aussi stockées dans un tableau indexé par l'indice de chaque variable affectée dès que ces indices sont disponibles. Ceci permet de détecter les doublons ainsi que les variables qui ne sont pas affectées afin de leur ré-affecter leur valeur précédente s'il s'agit d'une variable d'état.

Dans tous ces cas, les valeurs à affecter aux variables sont stockées sous la forme d'expressions symboliques. Les expressions sont décrites dans la section 5.4 page 49.



5.5.3 Evénements

La structure qui décrit un événement dans Mec V contient une référence au nœud dans lequel l'événement a été défini, son nom, un indice qui est l'unique numéro attribué à l'événement dans ce nœud, des drapeaux de visibilité, et enfin la relation de transition associée à cet événement.

Evénement `e`	
nœud :	`Simple`
nom :	`"e"`
indice :	0
drapeaux :	*aucun*
transitions :	`(~v, e, v)`

La relation de transition est en fait un BDD, mais nous présentons dans la figure ci-dessus les triplets codés par le BDD pour des raisons de lisibilité.

5.5.4 Assertion et valeurs initiales

L'assertion d'un nœud AltaRica est stockée à la fois sous la forme d'une expression et sous la forme d'un BDD. Nous allons voir dans la section 5.5.8 que le fait de garder l'expression symbolique est nécessaire pour détecter les variables égales avant d'en calculer le BDD équivalent.

L'ensemble d'états initiaux est donné par une expression, qui est convertie en BDD dès que possible.

5.5.5 Sous-nœuds

Chaque sous-nœud est représenté par une petite structure contenant une référence au nœud, le nom de ce sous-nœud, un indice, le décalage qu'il faut ajouter aux indices des variables du sous-nœud pour les instancier dans ce nœud, et un décalage similaire pour les indices des variables booléennes à l'intérieur du gestionnaire de BDDs des événements (cf. 5.2.5). S'il y a n sous-nœuds, ils seront numérotés de 0 à $n-1$.

Le contenu des deux structures représentant chacune un sous-nœud de l'exemple de la figure 5.7 page 56 est donné ici :

Sous-nœud `N1`	
nom :	`"N1"`
nœud :	`Simple`
indice :	0
décalage variables :	0
décalage BDD événement :	2

Sous-nœud `N2`	
nom :	`"N2"`
nœud :	`Simple`
indice :	1
décalage variables :	1
décalage BDD événement :	5

Les décalages s'expliquent comme ceci :

décalage variables Le nœud parent `SynchroParAssert` n'a pas de variables qui lui sont propres, donc le décalage du premier nœud est 0. Celui du deuxième nœud est 1 puisque les variables du sous-nœud `N1` ont déjà été traitées et que le sous-nœud `N1`, qui est une instance du nœud `Simple`, contient une variable.

décalage BDD événement Le nœud parent n'a pas d'événement explicite et il n'a donc qu'un seul événement ε d'indice 0. Pour les raisons qui sont expliquées en 5.3.2 page 48, Mec V utilise deux bits pour coder cet événement. La sous-section 5.5.7 contient une explication sur le codage des vecteurs d'événements.

5.5.6 Vecteurs de diffusion

Chaque vecteur de diffusion est stocké dans une petite structure qui conserve la liste des événements appartenant à cette diffusion, et la contrainte de diffusion. Pour chaque événement appartenant à la liste, sont conservés l'indice du sous-nœud auquel il appartient et un drapeau permettant de repérer ceux qui sont *diffusés*, c'est-à-dire ceux qui sont suivis d'un point d'interrogations dans le fichier AltaRica.

Il est nécessaire de stocker l'indice du sous-nœud puisque ce sont des *instances* d'événements qui sont synchronisées.

Par exemple, le premier vecteur de diffusion du nœud `Voiture` de la figure 4.10 page 33 se code par les structures de la figure 5.6. Nous rappelons ce vecteur ici :

```
<PAvG.fermeClapet, PAvD.fermeClapet, PArG.fermeClapet?, PArD.fermeClapet?>
```

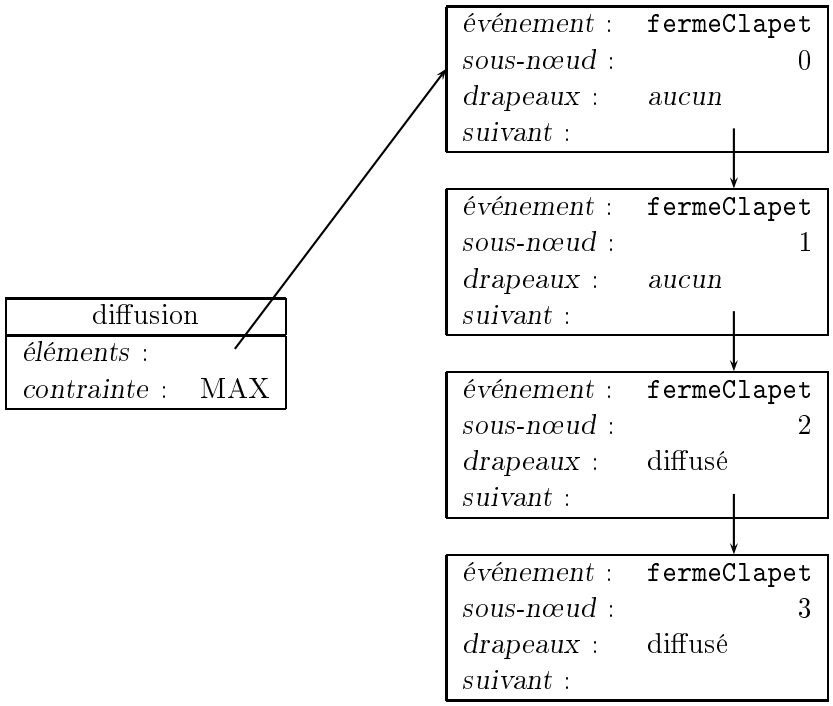


FIG. 5.6 – Le premier vecteur de diffusion du nœud `Voiture`

5.5.7 Les types de relations fournis par un nœud AltaRica

Un nœud AltaRica apparaît dans Mec V sous la forme de deux types et deux relations :
- le type de ses configurations (c)
- le type de ses vecteurs d'événements (ev)
- sa relation de transition ($\subseteq c \times ev \times c$)
- la relation de ses états initiaux ($\subseteq c$)

Nous verrons dans la section 5.6 qui suit comment les deux relations sont calculées. Nous allons ici décrire uniquement la façon dont les trois types c, ev et $c \times ev \times c$ sont constitués, et dont les relations ayant ces types seront stockées dans leurs gestionnaires de BDDs respectifs (cf. sous-section 5.2.5).

Etant donné un nœud AltaRica comportant n variables (y compris les variables apparaissant dans ses sous-nœuds) et m sous-nœuds, voici les différents codages.

Les configurations

Si on note les n variables $x_0, x_1, \ldots, x_{n-1}$ (d'indices respectifs $0, 1, \ldots, n-1$), la représentation binaire d'une configuration sera la valeur de chacune de ses variables dans cette configuration, mises bout à bout. Le bit à partir duquel apparaîtra la valeur de la variable x_i est noté dx_i. On a toujours $dx_0 = 0$, et $dx_{i+1} = dx_i + \textit{taille-en-bits}(x_i)$.

Variable :	x_0	x_1	...	x_{n-1}
Bit de position :	dx_0	dx_1	...	dx_{n-1}

Les vecteurs d'événements

Les vecteurs d'événements ont la même structure qu'une configuration : ils sont constitués d'une suite d'emplacements permettant de stocker l'indice d'un événement. Chaque emplacement (notés e_{loc}, e_{s_1}, ..., $e_{s_{n-1}}$) correspond soit au nœud local soit à l'un des sous-nœuds.

La relation de transition

La relation de transition est construite explicitement en tant que le produit cartésien $c \times ev \times c$. Le gestionnaire de BDDs va donc entrelacer les niveaux des variables, de manière transparente. Un élément de la relation de transition est donc vu comme un vecteur de bits de cette forme-là :

x_0	x_1	...	x_{n-1}	e_{loc}	e_{s_1}	...	$e_{s_{n-1}}$	x'_0	x'_1	...	x'_{n-1}

L'entrelacement effectué par le gestionnaire de BDDs est essentiel car dans la sémantique AltaRica, il y a une boucle ε sur chaque configuration, ce qui signifie que toutes les transitions où pour tout i $x_i = x'_i$ apparaissent nécessairement dans la relation de transition. Comme cela est expliqué dans la sous-section 5.2.4, l'absence d'entrelacement conduirait à des performances complètement inacceptables puisque tous les BDDs des relations de transition seraient de taille exponentielle.

5.5.8 Partage des variables égales

L'utilisation des assertions pour partager des variables est extrêmement fréquente dans les modèles AltaRica (cf. sous-section 4.3.2 sur la liaison par assertion).

Cela signifie qu'un grand nombre de variables apparaissant dans une configuration sont égales.

Plutôt que d'entrelacer les variables apparaissant dans une configuration, ce qui aboutirait plus ou moins à répartir aléatoirement les bits des variables à l'intérieur d'une configuration, Mec V détecte dans l'expression de l'assertion les variables qui seront toujours égales. Pour chacune de ces classes d'égalité, un seul jeu de variables booléennes permettant de représenter une de ces variables est allouée dans le BDD, et toutes les variables d'une même classe utilisent le même ensemble de variables booléennes.

Nous allons montrer sur un exemple simple ce qui se passe :

```
node Simple
    state v : bool : parent;
    event e;
    trans
        ~v |- e -> v := true;
edon
```

```
node SynchroParAssert
    sub N1, N2 : Simple;
    assert
        N1.v = N2.v;
edon
```

FIG. 5.7 – Les nœuds `Simple` et `SynchroParAssert`

Exemple 5.3 *La figure 5.7 présente le nœud* `SynchroParAssert`*, qui reprend l'exemple de la figure 4.3 de la page 29 au détail près que la variable* `v` *du nœud* `Simple` *est cette fois-ci exportée vers ses parents (par l'attribut* `parent`*) pour pouvoir être contrainte par l'assertion du nœud* `SynchroParAssert`*.*

Nous avons vu dans les sous-sections précédentes comment ces nœuds étaient représentés en mémoire.

Les deux variables `v` *des sous-nœuds sont contraintes par l'assertion à être égales. Elles ont chacune un indice différent puisqu'elles ne correspondent pas à la même instance de variable, mais elles sont toutes les deux représentées par la même variable booléenne dans les BDDs qui codent les configurations du nœud* `SynchroParAssert`*. Le gestionnaire de BDDs de ce nœud ne sera prévenu de l'existence que de l'une de ces deux instances.*

5.6 Calcul de la sémantique d'un modèle AltaRica

Le calcul de la sémantique d'un modèle AltaRica consiste dans Mec V à calculer la relation de transition et la condition initiale d'un nœud, qui contient potentiellement des sous-nœuds.

> Nous utiliserons ici le terme "symbolique" pour nous référer aux expressions lorsqu'elles sont codées sous la forme d'arbres, contenant des variables, et des opérateurs (union, intersection, ...). Bien que les BDDs soient souvent considérés à juste titre comme des structures "symboliques", nous précisons explicitement les cas où un objet est sous forme de BDD.

La *mise à plat* d'un nœud AltaRica consiste à transformer un nœud AltaRica en un composant (un autre nœud AltaRica sans sous-nœuds) de manière purement symbolique. Cette transformation est complexe du fait des post-conditions et des priorités ; elle est documentée dans la thèse de Gérald Point [45] et implémentée dans les outils associés [46].

Ici, du fait de l'utilisation des BDDs, on peut se permettre de calculer la sémantique d'un nœud directement à partir de la *sémantique* de ses sous-nœuds.

5.6.1 Préparation du nœud

La première étape du calcul de la sémantique consiste à calculer récursivement la sémantique des sous-nœuds apparaissant dans ce nœud. Ensuite, l'assertion et les conditions initiales de chacun des sous-nœuds sont incorporées de manière symbolique au nœud courant.

Une fois l'assertion symbolique constituée, les gestionnaires de BDDs de chacun des types offerts par le nœud (5.5.7) peuvent être créés. Ceci ne peut être fait avant puisque c'est l'assertion symbolique qui permet de détecter les variables qui vont être partagées (5.5.8).

La condition initiale est aussi gardée sous forme symbolique par homogénéité avec l'assertion (ces deux relations sont créées et utilisées de manière très semblable), mais rien n'empêcherait qu'elle soit seulement calculée sous forme de BDD.

L'assertion et la condition initiale sont ensuite évaluées sous forme de BDDs, dès que les gestionnaires de BDDs sont en place.

Les phases suivantes sont dédiées au calcul de la relation de transition.

5.6.2 Calcul des relations de transition des événements locaux

Chaque macro-transition définit une relation de transition, qui est calculée sous forme de BDD à partir de sa garde et de ses affectations.

Ensuite, la relation de transition associée à chaque événement local est calculée comme la réunion des relations de transition des macro-transitions étiquetées par cet événement.

Comme aucune macro-transition ne peut correspondre à l'événement ε, le BDD qui code la relation de transition de l'événement ε local est calculée et associée à cet événement.

5.6.3 Calcul de la relation de transition

Il reste à incorporer à ces relations de transition les comportements des sous-nœuds, en tenant compte des vecteurs de diffusion.

Les vecteurs de diffusion sont décomposés en vecteurs d'événements qui sont liés par un ordre partiel en fonction des événements diffusés.

Pour chaque vecteur d'événements (complété par des événements ε, cf. 4.3.3, page 29), sa relation de transition est calculée comme l'intersection des relations de transition des événements le composant.

Les contraintes de priorité données explicitement par l'utilisateur ou induites par les vecteurs de diffusion interdisent à certains événements de se produire lorsque d'autres peuvent l'être. La relation de transition de chaque vecteur d'événements est donc mise à jour pour tenir compte des restrictions nécessaires.

Enfin, la relation de transition du nœud est la réunion des relations de transition des vecteurs d'événements.

5.7 Gestion de la mémoire

Mec V utilise des "pools de mémoire" pour gérer les petites structures de données dont l'allocation et la libération sont fréquentes et conditionnent les performances de l'outil.

Dans le cas particulier du module de BDDs, un ramasse-miettes est utilisé, qui est décrit dans la section 5.2.1.

Un effort a été fait pour la gestion des identificateurs car celle-ci a un impact sur toutes les tables de hachage qui ont pour clé des identificateurs, et il y en a beaucoup dans Mec.

5.7.1 Pools de mémoire

Mec V propose deux modules de gestion de la mémoire. Le premier est un allocateur de mémoire générique qui délègue à la bibliothèque C standard le travail d'allocation. Le deuxième permet de créer des "pools" de mémoire : on précise à la création d'un pool la taille (qui doit être fixe) des éléments que l'on va stocker dans ce pool.

A partir de là, il est possible de manière très simple pour le programmeur et très efficace en termes de consommation mémoire et de consommation de temps de calcul d'allouer et de libérer des objets dans ce pool.

Il y a de multiples avantages à utiliser des pools de mémoire : nous avons déjà mentionné l'efficacité, mais cela permet aussi de collecter des statistiques sur l'utilisation qui est faite de la mémoire. Ainsi, chaque pool porte un nom et il est ainsi possible de savoir quel module dans Mec V consomme le plus de mémoire à un instant donné.

A l'heure actuelle, les pools sont utilisés pour les structures de données de petite taille et dont l'allocation et la libération sont des opérations sensibles en termes de performances : les nœuds des BDDs et les structures de données de chaque cellule dans les tables de hachage sont allouées de cette manière, ainsi que les éléments qui constituent les listes chaînées.

5.7.2 Stockage des identificateurs

Chaque identificateur apparaissant dans un modèle ou dans une spécification est stocké de manière unique dans une table de hachage. Ainsi, une fois que l'analyse lexicale d'un fichier est terminée, tout identificateur n'est stocké qu'en un seul exemplaire en mémoire et l'égalité entre

deux identificateurs peut être testée en temps constant, puisqu'il suffit de comparer les deux pointeurs.

5.8 Interface utilisateur

Nous avons choisi dans Mec V de proposer une utilisation interactive du logiciel, plutôt que de travailler uniquement sur des fichiers déjà saisis. Cela a impliqué un effort de développement important car une interface utilisateur n'est utilisable réellement que si elle propose certaines fonctionnalités auxquelles on s'est habitué au fil du temps, comme la possibilité d'éditer une ligne en cours de saisie, de revenir dans l'historique des commandes tapées, ou de pouvoir abréger les commandes dont le nom est trop long.

> L'inférence de types simple (cf. sous-section 5.4.4) a aussi été ajoutée dans cette optique-là : rendre la saisie de propriétés la plus naturelle et efficace possible.

5.8.1 Exécution de commandes

En plus de pouvoir saisir des propriétés, il est possible de demander à Mec d'effectuer des actions particulières, comme par exemple *charger un fichier AltaRica* (`:ar-load`), *afficher une variable* (`:display`), ...

Ces commandes sont recensées par un module `command` qui permet d'enrichir l'ensemble des commandes disponibles à la volée. Autrement dit, le module `command` offre une fonction permettant d'enregistrer à tout moment une nouvelle commande. Il est donc très facile d'ajouter à Mec des fonctionnalités comme par exemple :

- choisir quel sous-ensemble de commandes est disponible à l'utilisateur en fonction de paramètres connus seulement à l'exécution (par exemple l'existence d'une connexion graphique)
- charger dynamiquement des fonctionnalités (écriture de *plug-in*s)

Le module `command` gère aussi l'abréviation non-ambiguë de commande : il est possible de ne taper que les premières lettres d'une commande pour l'exécuter. En cas d'ambiguïté, le module `command` affichera les diverses possibilités.

5.8.2 Affichage de messages

Bien que Mec V ne propose à l'heure actuelle qu'une interface en mode texte, un module `message` a été écrit, qui permet deux choses :

- catégoriser les messages affichés (*avertissement*, *erreur*, *erreur du système*), afin de permettre des traitements différents (affichage dans une fenêtre séparée, changement de couleur, ...)
- enregistrer ou supprimer des *récepteurs de messages* (`listener`) qui feront le travail effectif d'affichage. Le module `message` se contente donc lorsqu'un message lui est fourni de le transmettre aux divers *récepteurs* qui se sont enregistrés.

Ce mécanisme d'affichage de message simplifiera grandement le développement de la partie graphique de Mec, ainsi que par exemple la possibilité de générer simplement des fichiers journaux de sessions.

5.8.3 Interaction effective avec l'utilisateur

Les deux modules `command` et `message` que nous venons de décrire sont d'assez haut niveau, et aucun des deux n'interagit réellement avec l'utilisateur. Cette partie concrète du travail d'affichage et de lecture des saisies de l'utilisateur est gérée par les modules "UI" (User Interface).

A l'heure actuelle, un module graphique très simple existe (pour "prouver le concept") mais n'est pas compilé dans Mec par défaut. Ses fonctionnalités sont beaucoup trop restreintes.

Le seul module compilé dans Mec utilise le terminal pour ses entrées et ses sorties. Il est assez évolué puisqu'il utilise la bibliothèque `readline` lorsqu'elle est disponible. La bibliothèque `readline` prend en charge l'édition des lignes et l'historique des commandes tapées.

5.9 Modules utilitaires

Mec V contient plusieurs modules "utilitaires", les plus visibles d'un point de vue algorithmique étant ceux qui implémentent des structures de données classiques (`array`, `bitvector`, `graph`, `hash`). Il y a aussi des modules de support afin de programmer en C comme on le ferait dans un langage de plus haut niveau (`exception`, `iterator`). Nous allons les passer en revue dans le reste de cette section.

5.9.1 Les constructions de haut niveau

Programmer un model-checker en langage C nécessite de se fixer un cadre de travail sérieux et efficace. Certaines constructions de langages plus évolués et certains concepts issus du génie logiciel sont extrêmement utiles pour éclaircir le code source d'un programme et faciliter sa maintenance. Deux des concepts que nous avons utilisés nécessitent une part de programmation, et font l'objet de deux modules ; il s'agit des exceptions et des itérateurs.

Nous avons aussi canalisé l'affichage des messages utiles au débuggage grâce à un module `debug` que nous décrivons.

Les exceptions

La gestion des erreurs est un problème récurrent en génie logiciel, et la solution la plus communément mise en œuvre est l'utilisation d'*exceptions*, qui permettent de remonter les erreurs directement au niveau d'une fonction sachant les traiter, sans forcer toutes les fonctions du programme à les propager.

Nous avons écrit un module `exception` qui permet d'instrumenter ce concept. Ce module présente les avantages suivants par rapport à d'autres implémentations disponibles sur l'Internet :

- il utilise la pile pour stocker les informations dont il a besoin, et il ne nécessite donc pas d'allocation de mémoire explicite
- il permet de coder une relation d'héritage simple entre exceptions
- il est utilisable librement

L'héritage entre exceptions est très utile pour catégoriser les exceptions. On distingue par exemple les erreurs de conception du programme qui aboutissent à un cas que le programme ne sait pas gérer (erreurs *logiques*), des erreurs qui viennent de l'environnement dans lequel le programme s'exécute (erreurs *à l'exécution*, *runtime errors*) comme par exemple un manque de mémoire ou une saisie erronnée de l'utilisateur. Par exemple, il est possible grâce à l'héritage de

décider de ne traiter que les erreurs *runtime* et de laisser les erreurs logiques générer des informations utiles au débuggage de l'application, et cela sans avoir à recenser toutes les différentes exceptions utilisées dans le logiciel.

Les itérateurs

Le module `iterator` permet de coder un objet *itérateur*, c'est-à-dire permettant d'itérer sur un ensemble d'éléments. Les fonctionnalités proposées par un itérateur sont les suivantes :

- tester s'il reste un élément
- retourner l'élément suivant
- retourner l'élément couple suivant
- se détruire

Les itérateurs sont utilisés à beaucoup d'endroits dans Mec V, et leur implémentation totalement générique les rend très utiles pour abstraire les algorithmes des structures de données sous-jacentes : une fois que l'on a obtenu un itérateur, on ne peut plus distinguer la structure de données sous-jacente dont il provient.

La fonctionnalité "retourner l'élément couple suivant" est utile pour itérer sur une structure comme une table de hachage, qui code en fait des couples d'éléments (clé, valeur) dans notre implémentation.

Messages de débuggage

La traque des erreurs de programmation fait partie intégrante du cycle de vie d'un logiciel et doit être prise en compte lors de sa conception. En effet, il est aisé d'afficher des messages qui permettent de tracer l'exécution du programme et qui donnent des informations utiles. Cependant, en fonction du sous-système que l'on est en train d'observer, on veut pouvoir activer ou désactiver les messages des autres sous-systèmes.

Nous avons donc implémenté un module `debug` qui propose une macro `DPRINTF` permettant de conditionner l'affichage d'un message en fonction d'une variable globale disant de quels sous-systèmes on souhaite voir les messages. Chaque appel à `DPRINTF` contient un message et un identifiant du sous-système qui génère ce message, qui permet à `DPRINTF` de choisir s'il faut afficher le message ou non.

Fonctions mathématiques simples

Le module `simple-math` propose une poignée de fonctions mathématiques simples qui reviennent fréquemment, comme le calcul du maximum de deux entiers, ou du logarithme en base 2 d'un entier, ...

5.9.2 Les structures de données

La plupart des algorithmes que nous avons implémentés nécessitent des structures de données classiques. Nous avons donc implémenté ces structures de manière générique afin d'éviter les duplications de code, et de favoriser l'homogénéité du code source.

array permet de représenter des tableaux de taille variable. Il est utilisé à de nombreaux endroits très différents comme par exemple pour stocker les chaînes de caractères saisies par l'utilisateur lorsque la bibliothèque `readline` n'est pas disponible, pour stocker des

tables inverses qui associent à un indice l'objet correspondant, pour implémenter le module `bitvector`, ...

bitvector permet de représenter des tableaux de bits de taille variable. Il est utilisé pour représenter les vecteurs de propriétés associés aux états et aux transitions dans un système de transitions représenté en extension, pour retourner les vecteurs de bits contenus dans un BDD, ... Le module propose aussi des fonctionnalités spécifiques, comme la possibilité d'énumérer tous les vecteurs de bits possibles d'une taille donnée, de trouver l'indice du premier bit positionné dans le vecteur, ...

graph permet de représenter un graphe orienté. Il est possible d'ajouter un sommet ou un arc, de tester l'existence d'un arc, d'étiqueter un sommet ou un arc avec des bits, ou d'obtenir un itérateur sur l'ensemble des sommets. Il n'est pour l'instant utilisé que pour représenter les ordres partiels sur les vecteurs d'événements, mais les modules `trans-sys`, `state`, et `transition` devraient être modifiés pour l'utiliser. Ils utilisent déjà une interface presque identique.

L'implémentation du module `graph` est décrite à la section 5.10.

hash permet de représenter une application d'un ensemble (de *clés*) dans un autre (de *valeurs*). Il est possible d'ajouter un couple (clé, valeur) dans la table, de demander la valeur associée à une clé donnée, ou d'obtenir un itérateur sur tous les éléments de la table. Lorsqu'on crée une table de hachage, il faut fournir une fonction de hachage qui à une clé associe un entier, et une fonction permettant de tester l'égalité entre deux clés. Deux couples classiques de telles fonctions sont fournis par le module, qui permettent de créer facilement des tables de hachage dont les clés sont des pointeurs ou des chaînes de caractères.

5.10 Compatibilité avec Mec 4

Deux commandes permettent dans Mec V de charger et afficher des systèmes de transitions au format Mec [5]. Nous indiquons ici brièvement les structures de données utilisées et les fonctionnalités offertes, mais l'apparition d'outils efficaces pour traiter des modèles AltaRica a rendu cette partie de Mec V obsolète au fur et à mesure de son développement.

5.10.1 Stockage des états et des transitions

Chaque état est représenté par une petite structure de données qui contient le nom de l'état, la liste des transitions ayant cet état pour source, et la liste des transitions ayant cet état pour cible. L'ensemble des états peut être chaîné dans une liste.

Chaque transition, qui est un triplet (état source, événement, état cible), est chaînée dans la liste des transitions sortantes de son état source et dans la liste des transitions arrivant dans son état cible.

Cette structure de données requiert un espace linéaire dans le nombre d'états et de transitions du système de transitions.

La figure 5.8 montre comment peut être codé le système de transitions donné à la figure 2.1 page 5.

Pour des raisons de clarté, nous présentons les deux chaînages des transitions sur deux schémas différents, mais la structure de données stockée en mémoire est la réunion des deux schémas, en identifiant les états et les transitions portant le même nom dans les deux schémas.

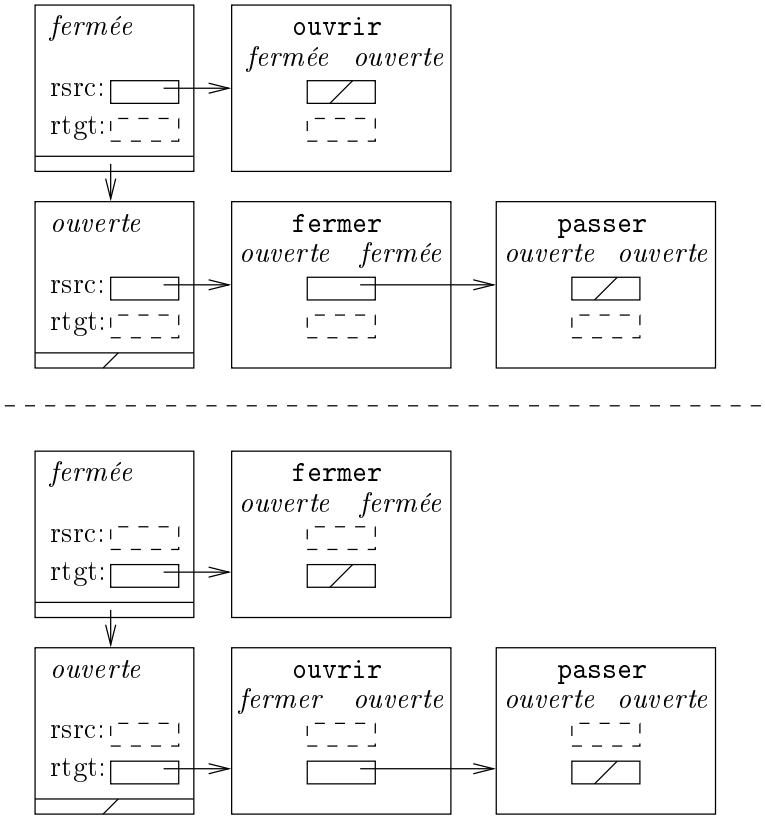


FIG. 5.8 – Exemple de codage explicite d'un système de transitions

De même, toujours dans la figure 5.8, les états source et cible d'une transition sont des pointeurs vers l'état correspondant, pas seulement du nom de l'état.

5.10.2 Calcul du produit synchronisé

Le produit synchronisé de deux systèmes de transitions est calculé par Mec V, et il est possible d'afficher le résultat de ce calcul. Cependant, aucun élément de la logique utilisée dans Mec 4 n'étant implémenté, ce calcul n'est pas réellement exploitable.

Chapitre 6

Synthèse de contrôleur

6.1 Introduction

Au-delà de la vérification de modèles, la théorie du contrôle propose de s'intéresser à un problème plus général : la synthèse d'un contrôleur.

Il s'agit, étant donné un processus souvent appelé $\mathcal{P}$ (pour *Plant*, usine en anglais), de synthétiser un *superviseur* $\mathcal{C}$ qui observe et intervient sur le processus supervisé afin d'assurer que l'*objectif de contrôle* soit satisfait.

Ramadge et Wonham [48] considèrent comme processus des systèmes de transitions déterministes et comme objectifs de contrôle des propriétés sur les exécutions (*linéaires*) du processus supervisé. Ils proposent alors une méthode qui permet de calculer un superviseur sous la forme d'une application qui en tenant compte de l'historique de l'exécution du processus autorise ou refuse certaines actions.

Après avoir présenté les concepts clé de la synthèse de contrôleurs et la méthode de Ramadge et Wonham, nous présentons une extension de leur théorie et une méthode faisant intervenir la théorie des jeux pour synthétiser un superviseur.

Notre méthode [50] permet de spécifier des objectifs de contrôle plus généraux, c'est-à-dire des systèmes d'équations de points fixes sur les comportements *arborescents* du processus. Le contrôleur fourni par la méthode est sous la forme d'un automate déterministe non bloquant.

Le problème de la synthèse de contrôleurs s'écrit donc

Etant donnés un processus $\mathcal{P}$ et une spécification φ,
construire s'il existe un contrôleur $\mathcal{C}$ tel que
$\mathcal{P}$ contrôlé par $\mathcal{C}$ *satisfasse* φ

6.2 Le cadre de Ramadge et Wonham

Nous décrivons ici les idées contenues dans le survey écrit par Ramadge et Wonham [48] en introduisant au fur et à mesure les notations que nous utiliserons.

6.2.1 Modélisation des systèmes à événements discrets

Ramadge et Wonham considèrent une catégorie de systèmes physiques appelés systèmes à événements discrets (DES, *Discrete Event Systems*). Ces systèmes tiennent leur nom du fait que l'on ne s'intéresse qu'aux événements qui décrivent leur comportement, sans se soucier de

leur état interne. Ils proposent ensuite un modèle "logique" de ces systèmes, qui est en fait un automate de mots déterministe dont l'alphabet est constitué des événements du DES. Ce modèle ne tient pas compte des dates auxquelles les événements surviennent : il s'agit d'automates de mots classiques non temporisés.

Avant de donner la définition que nous emploierons pour décrire ces systèmes, nous introduisons les deux notions fondamentales de la théorie du contrôle que sont la *contrôlabilité* et l'*observabilité*. Ces deux propriétés s'appliquent aux événements du système. Nous notons Σ l'ensemble de ces événements.

Les événements contrôlables : le système à contrôler et le contrôleur interagissent physiquement d'une certaine manière qui fait que certains événements peuvent être interdits par le contrôleur, alors que d'autres ne peuvent pas l'être.

Les événements contrôlables seront regroupés dans un sous-ensemble Σ_c de Σ, et les événements incontrôlables seront dans le complémentaire de Σ_c, que nous appelons Σ_{uc}.

Les événements observables : de la même manière, les contraintes physiques du processus et du contrôleur font que certains événements ne peuvent pas être observés par le contrôleur. Nous ne formalisons pas plus cette notion, qui peut aussi être décrite en partitionnant l'ensemble des événements, car nous ne nous intéresserons pas à cette propriété ici.

Nous choisissons naturellement de représenter le processus $\mathcal{P}$ à contrôler par un système de transitions déterministe.

Définition 6.1 (Processus) *Un processus est un triplet $\langle Q, \delta, q_0 \rangle$ avec Q un ensemble fini d'états, δ une fonction partielle de $Q \times \Sigma$ dans Q, et $q_0 \in Q$ l'état initial.*

Dans [8], nous nous intéressons à la notion d'observabilité, et la construction proposée repose sur un étiquetage des états par des propriétés. Cependant, comme nous avons pris le parti ici de ne pas tenir compte de l'observabilité, nous considérons des processus non étiquetés.

Dans [50], nous proposons aussi d'étiqueter les états par des propriétés afin de simplifier la modélisation des processus. Cependant, la remarque 6.2 permet de se ramener aisément à un processus sans étiquettes.

Remarque 6.2 *Etant donnés un ensemble de propriétés élémentaires Λ, et un processus étiqueté $\langle Q, \delta, q_0, \lambda \rangle$ avec une fonction partielle λ de Q dans Λ, il est possible de transformer ce processus étiqueté en un processus sans étiquettes sur lequel il est tout de même possible de vérifier les propriétés élémentaires grâce à des propriétés modales.*

Précisément, on étend l'alphabet Σ sur lequel est défini le processus étiqueté en un alphabet $\Sigma \cup \Sigma_\Lambda$ tel que Σ_Λ soit disjoint de Σ avec $\Sigma_\Lambda = \{p_l \,|\, l \in \Lambda\}$. Il reste alors à ajouter aux états étiquetés par l du processus initial une transition sortante ayant pour événement associé p_l.

Formellement, on définit le processus $\langle Q, \delta', q_0 \rangle$ en étendant la fonction de transition comme ceci : pour tout $q \in Q$,

$$\lambda(q) = l \Leftrightarrow \delta'(q, p_l) = q \text{ et } \forall a \in \Sigma, \delta'(q, a) = \delta(q, a)$$

La propriété "q est étiqueté par l" dans le processus étiqueté devient "il existe un successeur de q par l" dans le processus sans étiquettes d'états.

Afin de garantir que les propriétés faisant intervenir les étiquettes d'état aient la même sémantique que les propriétés modifiées comme ci-dessus, les nouveaux événements de Σ_Λ doivent être choisis incontrôlables.

6.2.2 Le contrôleur

Dans la théorie de Ramadge et Wonham, le superviseur est représenté par une fonction qui étant donné l'historique du processus supervisé lui fournit l'ensemble des actions contrôlables qui sont autorisées.

Comme cela est suggéré dans [48], il est possible de coder le contrôleur par un deuxième processus, synchronisé avec le processus supervisé. La synchronisation permet au contrôleur d'observer l'exécution du processus, et d'empêcher les actions *contrôlables* qui n'apparaissent pas dans son propre comportement. Un contrôleur est donc un processus qui peut toujours exécuter toute action incontrôlable.

Définition 6.3 (Contrôleur) *Un contrôleur est un processus* $\langle Q, \delta, q_0 \rangle$ *qui satisfait la propriété :*

$$\forall q \in Q, \forall u \in \Sigma_{uc},\ \delta(q,u) \text{ est défini}$$

6.2.3 Exemple

Voici un exemple de processus contrôlé extrait de [50]. Il s'agit d'assurer que le processus pourra toujours atteindre l'état 4, sachant que seul l'événement c est contrôlable.

Le processus et un contrôleur assurant cette propriété sont donnés à la figure 6.1.

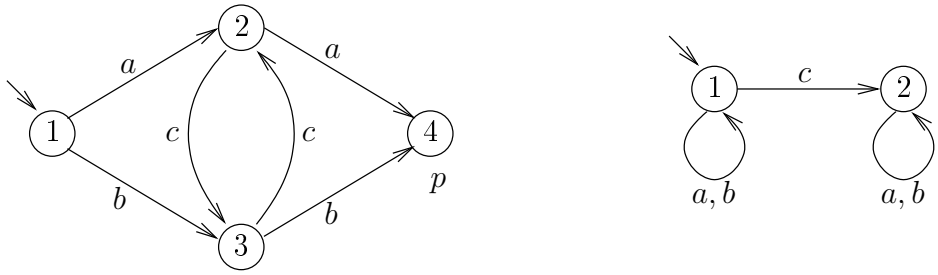


FIG. 6.1 – Un processus et un contrôleur

On voit sur le produit des deux, qui est le processus contrôlé, à la figure 6.2 que le contrôleur interdit tout événement c dès qu'un événement c s'est déjà produit. Un contrôleur moins permissif aurait pu interdire tous les événements c dès le début.

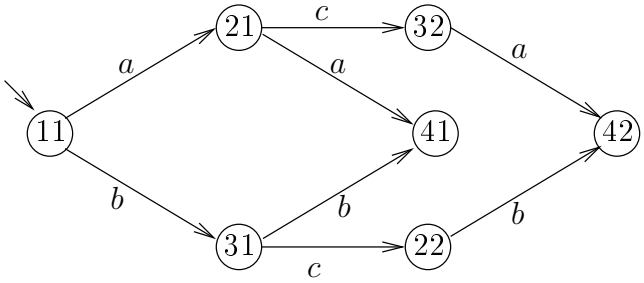


FIG. 6.2 – Le processus contrôlé

6.3 Jeux de parité

Un jeu de parité se joue à deux. L'arène de jeu est constituée d'un graphe orienté fini dont les sommets sont appelés *positions* et les arcs *déplacements*.

Définition 6.4 (jeu de parité) *Un jeu de parité est un tuple* $\langle V_0, V_1, E, rang\rangle$ *où* V_0 *et* V_1 *sont deux ensembles finis disjoints de* positions *des joueurs 0 et 1 respectivement,* $E \subseteq V_0 \times V_1 \cup V_1 \times V_0$ *décrit les* déplacements *possibles, et* $rang : V_0 \uplus V_1 \to \mathbb{N}$.

Une *partie* sur ce jeu se déroule comme suit : un jeton est posé sur une position, et chaque joueur, lorsque le jeton est sur une position lui appartenant, doit le pousser sur une position adjacente. Une partie est représentée par le mot de $(V_0 \uplus V_1)^\infty$ qui correspond aux positions visitées au cours de la partie. Une partie finie est perdue par le joueur qui ne peut plus jouer dans la dernière position de la partie. Pour une partie infinie π, la parité de $\max_i rang(\pi_i)$ détermine le joueur qui gagne. Si le max est pair c'est le joueur 0, sinon c'est le joueur 1.

> L'ensemble $\{rang(\pi_i) \mid i \in \mathbb{N}\}$ est toujours fini car le graphe est fini. C'est ce qui nous autorise à considérer le max de cet ensemble.

Une *mémoire* pour un jeu de parité est un ensemble H équipé d'une fonction de mise à jour *hist* allant de $H \times (V_0 \uplus V_1)$ dans H et d'un état de départ $h^0 \in H$.

Une *stratégie* avec mémoire pour le joueur 0 est une application de $V_0 \times H$ dans V_1, où H est une mémoire associée à la stratégie. La stratégie est dite à mémoire finie si l'ensemble H est fini. La stratégie est dite *positionnelle* si l'ensemble H est réduit à un singleton.

Etant donnée une stratégie σ pour le joueur 0, on peut définir les parties π *respectant* σ comme suit :

- On définit les états de la mémoire au cours de la partie par $h_0 = h^0$ et $\forall i \geq 1$, $h_i = hist(h_{i-1}, \pi_{i-1})$
- $\forall i$, $\pi_i \in V_0 \Rightarrow \pi_{i+1} = \sigma(\pi_i, h_i)$

Une stratégie est gagnante en une position (dite gagnante) pour le joueur 0 si quels que soient les déplacements effectués par le joueur 1, toute partie commençant dans cette position et respectant la stratégie est gagnée par le joueur 0.

Théorème 6.5 (Emerson, Jutla, 1991[21]) *Tout jeu de parité est* déterminé *: il existe une partition de l'ensemble* $V_0 \uplus V_1$ *en deux sous-ensembles* W_0 *et* W_1 *tels que les positions de* W_0 *sont gagnantes pour le joueur 0 et les positions de* W_1 *sont gagnantes pour le joueur 1.*

De plus, il existe une stratégie positionnelle pour chacun des joueurs qui est gagnante dans chacune des positions gagnantes pour ce joueur.

6.4 Objectifs de contrôle arborescents

Comme dans [50] et [8], nous étendons ici la méthode proposée par Ramadge et Wonham à des objectifs de contrôle arborescents, c'est-à-dire portant sur l'arbre des exécutions du processus supervisé et non pas seulement sur ses traces d'exécutions.

Nous allons exposer la méthode proposée dans [8] sans parler de l'observabilité. Ce choix permet de concentrer la présentation sur la notion de *division* d'un automate par un processus. Nous présentons rapidement les autres aspects de cet article dans la section 6.5.

La propriété particulière que doit satisfaire le contrôleur (définition 6.3) peut en fait s'écrire comme une conjonction de modalités en chaque état du système :

$$\bigwedge_{a \in \Sigma_{uc}} \langle a \rangle \top$$

Ce qui se traduit immédiatement dans l'équation de (plus grand) point fixe suivante, qui doit être satisfaite par les états du contrôleur :

$$x \stackrel{\nu}{=} \bigwedge_{a \in \Sigma_{uc}} \langle a \rangle x \wedge \bigwedge_{c \in \Sigma_c} [c]\, x$$

Comme les états inaccessibles du contrôleur ne joueront jamais de rôle dans la supervision, il est suffisant de vérifier que l'état initial du contrôleur satisfait cette formule.

Ceci permet d'exprimer des propriétés plus fines sur la contrôlabilité des événements, sans avoir à modifier le processus ou à changer l'ensemble des événements Σ. Par exemple, il devient possible d'exprimer qu'un événement n'est contrôlable que sous certaines conditions. ("La barrière ne peut être fermée que quand le garde-barrière n'est pas en train de manger")

Le problème de la synthèse de contrôleurs peut désormais s'écrire de manière plus générale :

Etant donnés un processus $\mathcal{P}$ et deux spécifications φ et ψ,
construire s'il existe un processus $\mathcal{C}$ tel que
$\mathcal{P} \times \mathcal{C}$ satisfasse φ et
$\mathcal{C}$ satisfasse ψ

6.4.1 Systèmes d'équations modales

Pour des raisons de commodité, nous présentons les spécifications sous la forme de systèmes d'équations de points fixes, avec les modalités existentielle et universelle habituelles. Le but est de rendre les constructions très lisibles, en s'autorisant la manipulation des modalités directement ; on peut lire un tel système immédiatement comme un automate non-déterministe d'arbres, et comme cela est expliqué dans [7], un tel système est équivalent à un terme du μ-calcul modal.

Ces systèmes d'équations sont fortement liés aux systèmes d'équations booléens présentés dans le chapitre 3, sauf qu'ici il ne s'agit pas de relations, et nous avons donc pu occulter les expressions booléennes qui sont codées implicitement par les alternances entre variables existentielles et universelles.

Définition 6.6 (système d'équations) *Un système de points fixes d'équations modales est un tuple*

$$\langle Var, Var^{\exists}, Var^{\forall}, v_0, \delta, rang \rangle$$

avec $Var = Var^{\exists} \uplus Var^{\forall}$, $v_0 \in Var^{\exists}$, $\delta : Var \to Terme(Var^{\exists}) \cup 2^{Var^{\forall}}$, *et* $rang : Var \to \mathbb{N}$.
Nous imposons de plus que $\delta(Var^{\exists}) \subseteq Var^{\forall}$ *et que* $\delta(Var^{\forall}) \subseteq Terme(Var^{\exists})$.

$Terme(Var^{\exists})$ *est l'ensemble des conjonctions de modalités, de la forme*

$$\bigwedge_{a \in \Sigma} (a) x_a$$

où x_a *désigne une variable de* $Var^{\exists}$ *et* (a) *est une modalité, soit existentielle (*$\langle a \rangle$*) soit universelle (*$[a]$*).*

Remarque 6.7 *Le système d'équations a quatre propriétés importantes :*
- *il est* gardé *: aucune variable n'apparaît dans un terme sans être conditionnée par une modalité*
- *chaque équation est sous forme normale disjonctive (on dirait pour un automate qu'il est* non-déterministe*) : les transitions partant d'une variable existentielle vont directement sur une variable universelle, sans modalité*
- *le système est* biparti *: l'ensemble des variables est séparé en deux sous-ensembles complémentaires et les transitions ne vont jamais d'un des deux sous-ensembles dans lui-même*
- *le système est* complet *: pour chaque lettre de l'alphabet Σ, tout terme contient une modalité la concernant.*

Nous donnons la sémantique d'un tel système $\mathcal{S} = \langle Var, Var^{\exists}, Var^{\forall}, \delta_{\mathcal{S}}, v_0, rang\rangle$ sur un processus $\mathcal{P} = \langle Q, \delta_{\mathcal{P}}, q_0\rangle$ sous la forme d'un jeu de parité $\mathcal{G}(\mathcal{S}, \mathcal{P}) = \langle V_0, V_1, E, rang\rangle$ défini comme suit :

$V_0 = Var^{\exists} \times Q$, $V_1 = Var^{\forall} \times Q$, et les déplacements possibles en (v, q) sont :
- si $v \in Var^{\exists}$, alors $((v, q), (\delta_{\mathcal{S}}(v), q)) \in E$
- si $v \in Var^{\forall}$, alors pour toute modalité apparaissant dans la conjonction $\delta_{\mathcal{S}}(v)$,
  - cas $[a]\, x_a$: si $q' = \delta_{\mathcal{P}}(q, a)$ est défini, $((v, q), (x_a, q')) \in E$.
  - cas $\langle a\rangle x_a$: si $q' = \delta_{\mathcal{P}}(q, a)$ est défini, alors $((v, q), (x_a, q')) \in E$.

Pour tout couple $(v, q) \in Var \times Q$, $rang((v, q)) = rang(v)$.

Le processus $\mathcal{P}$ satisfait le système $\mathcal{S}$ si la position (v_0, q_0) est gagnante pour le joueur 0.

6.4.2 Satisfiabilité et extraction d'un modèle

Etant donné un système d'équations $\mathcal{S} = \langle Var, Var^{\exists}, Var^{\forall}, \delta_{\mathcal{S}}, v_0, rang\rangle$, il est possible de déterminer tous les modèles finis le satisfaisant.

En effet, tout modèle du système correspond à une stratégie à mémoire finie dans le jeu $\mathcal{G}(\mathcal{S}) = \langle V_0, V_1, E, rang'\rangle$, que nous donnons ici. Inversement, chaque stratégie nous permet de construire un modèle satisfaisant le système $\mathcal{S}$.

$V_0 = Var^{\exists}$, $V_1 = Var^{\forall} \times 2^{\Sigma}$, et les déplacements sont donnés comme suit :
- pour $v \in V_0$, $(v, (v', A)) \in E$ ssi
  $v' \in \delta_{\mathcal{S}}(v)$ et pour tout $\langle a\rangle x_a$ apparaissant dans la conjonction $\delta_{\mathcal{S}}(v')$, on a $a \in A$.
- pour $v \in V_1$, $((v, A), v') \in E$ ssi $\langle a\rangle v'$ ou $[a]\, v'$ apparait dans $\delta_{\mathcal{S}}(v)$.

Pour tout $v \in V_0$, $rang'(v) = rang(v)$ et pour tout couple $(v, A) \in V_1$, $rang'((v, A)) = rang(v)$.

Nous savons que dans les jeux de parité, il existe une stratégie *positionnelle* (ou *sans mémoire*) gagnante dans toutes les positions gagnantes pour le joueur 0. Ces stratégies positionnelles donnent aussi un contrôleur correct, mais n'engendrent pas tous les contrôleurs.

6.4.3 Division d'un système par un processus

Etant donnés deux processus $\mathcal{P}$ et $\mathcal{Q}$ et un système $\mathcal{S}$, on cherche à construire un nouveau système $\mathcal{S}'$ tel que :

$$\mathcal{Q} \times \mathcal{P} \models \mathcal{S} \iff \mathcal{Q} \models \mathcal{S}'$$

Autrement dit, on souhaite transformer le système $\mathcal{S}$ afin qu'il prenne déjà en compte les comportements de $\mathcal{P}$.

Nous définissons l'opération de division de cette manière :

$$\mathcal{Q} \times \mathcal{P} \models \mathcal{S} \iff \mathcal{Q} \models \mathcal{S}/\mathcal{P}$$

Définition 6.8 *Etant donné un système d'équations* $\mathcal{S} = \langle \mathrm{Var}, \mathrm{Var}^{\exists}, \mathrm{Var}^{\forall}, \delta_{\mathcal{S}}, v_0, rang \rangle$ *et un processus* $\mathcal{P} = \langle Q, \delta_{\mathcal{P}}, q_0 \rangle$*, on note* $\mathcal{S}/\mathcal{P}$ *le système*

$$\langle \mathrm{Var}_{\mathcal{S}/\mathcal{P}}, (\mathrm{Var}^{\exists} \times Q) \cup \{q_\bot\}, (\mathrm{Var}^{\forall} \times Q) \cup \{q_\top\}, \delta_{\mathcal{S}/\mathcal{P}}, v^0_{\mathcal{S}/\mathcal{P}}, rang_{\mathcal{S}/\mathcal{P}} \rangle$$

avec
- *pour* (v, q) *tel que* $v \in \mathrm{Var}^{\exists}$, $\delta_{\mathcal{S}/\mathcal{P}}((v, q)) = \{(v', q) \mid v' \in \delta_{\mathcal{S}}(v)\}$
- *pour* (v, q) *tel que* $v \in \mathrm{Var}^{\forall}$*, chacune des modalités est transformée comme suit :*

$$[a]\,(x_a, q') \rightsquigarrow \begin{cases} [a]\,q_\top & \text{si } \delta_{\mathcal{P}}(q, a) \text{ n'existe pas} \\ [a]\,(x_a, q') & \text{sinon} \end{cases}$$

$$\langle a \rangle (x_a, q') \rightsquigarrow \begin{cases} \langle a \rangle q_\bot & \text{si } \delta_{\mathcal{P}}(q, a) \text{ n'existe pas} \\ \langle a \rangle (x_a, q') & \text{sinon} \end{cases}$$

Pour tout (v, q), $rang((v, q)) = rang(v)$.

6.4.4 Produit de deux systèmes

Le contrôleur doit satisfaire les objectifs de contrôle $\mathcal{S}$ que l'on sait désormais exprimer en fonction du processus $\mathcal{P}$ grâce à l'opération de division, mais aussi une propriété $\mathcal{S}'$ vérifiant qu'il s'agit bien d'un contrôleur.

Le contrôleur Q est donc un processus qui est un modèle du système :

$$\mathcal{S}/\mathcal{P} \times \mathcal{S}'$$

Ce produit est exactement l'équivalent de la conjonction entre les propriétés $\mathcal{S}$ et $\mathcal{S}'$, dont on sait qu'elle existe et est exprimable dans notre formalisme puisque celui-ci est équivalent au μ-calcul.

Il existe même une construction directe du système produit qui passe par des systèmes à multi-parités, qui se traduisent en systèmes à parité.

6.5 Généralisation

La méthode de synthèse de contrôleur présentée ici est plus générale encore, comme on peut le lire dans [8].

On peut citer comme extension l'introduction d'une modalité supplémentaire dans la logique de spécification qui conserve tout de même toutes les propriétés importantes du μ-calcul. Cette modalité permet de spécifier la notion d'(in-)observabilité.

La division d'une spécification par un processus est aussi étendue à la division d'une spécification par une autre spécification. Ceci est utilisé pour traiter des problèmes de contrôle décentralisé.

6.6 Calcul de stratégies gagnantes dans un jeu de parité

Il existe de nombreux algorithmes permettant de calculer des stratégies gagnantes dans un jeu de parité. La plupart d'entre eux, comme par exemple celui de Jens Vöge et Marcin Jurdziński [51], sont des algorithmes de graphes qui utilisent explicitement la structure sous-jacente du jeu.

Nous présentons ici une méthode en deux étapes, qui ramène le calcul des stratégies gagnantes *positionnelles* d'un jeu de parité à la résolution d'un système d'équations de points fixes booléen paramétré, ou autrement dit un système d'équations de points fixes booléen sur des relations. Un tel système peut être résolu avec Mec V.

Nous adoptons tout de suite une syntaxe abstraite qui plonge un jeu de parité (définition 6.4, page 68) $\mathcal{G} = \langle V_0, V_1, E, rang\rangle$ dans une vision "système d'équations booléen" en notant x_i les positions de V_0 et y_i les positions de V_1. Le lecteur intéressé par les liens entre le μ-calcul et les jeux de parité peut consulter par exemple [21], [52] ou [7].

Pour chacune des positions de V_0, on écrit l'équation $x_i \overset{rang(x_i)}{=} \bigvee_{j\in Y_i} y_j$ dans laquelle $\{y_j\}$ est l'ensemble des successeurs de x_i par E. On note de même $y_i \overset{rang(y_i)}{=} \bigwedge_{j\in X_i} x_j$ où $\{x_j\}$ est l'ensemble des successeurs de y_i par E. L'ensemble de ces équations forme un système booléen $\mathcal{S}$ tel qu'il existe une stratégie gagnante pour le joueur 0 dans la position x_i si et seulement si x_i est vraie dans la solution du système.

> Nous omettrons désormais de rappeler la présence du rang de la position x_i sur le signe d'égalité.

Le théorème bien connu 6.5 (cf. section 6.3 page 68) affirmant l'existence d'une stratégie gagnante positionnelle dans un jeu de parité pour chaque position gagnante se traduit sur le système d'équations $\mathcal{S}$ par le fait qu'il existe un système $\mathcal{S}'$ tel qu'en remplaçant chaque équation de la forme $x_i = \bigvee_{j\in Y_i} y_j$ par une équation où l'un seulement des y_j apparaît ($x_i = y_j$), les deux systèmes $\mathcal{S}$ et $\mathcal{S}'$ ont même solution.

Notre but est ici de modifier le système d'équations afin que la résolution du système modifié nous permette de savoir *quel* y_j a été choisi, ce qui nous donnera directement une stratégie positionnelle gagnante dans le jeu.

On définit donc le système $\mathcal{T}$ sur les relations booléennes en remplaçant les équations du système $\mathcal{S}$: $x_i = \bigvee_{j\in Y_i} y_j$ et $y_i = \bigwedge_{j\in X_i} x_j$, respectivement par les équations suivantes. Nous notons $\mathbf{a}$ la séquence des paramètres dont dépendent toutes les relations car il faut un paramètre booléen pour chaque élément de E ;
$\mathbf{a} = a_{11}, a_{12}, \ldots, a_{1|Y_1|}, \ldots, a_{i1}, a_{i2}, \ldots, a_{i|Y_i|}, \ldots, a_{n1}, a_{n2}, \ldots, a_{n|Y_n|}$

$$\mathcal{T} \begin{cases} R_i(\mathbf{a}) & = F_i(\mathbf{a}) \wedge \bigwedge\limits_{j\in Y_i} (a_{ij} \Rightarrow Z_j(\mathbf{a})) \\ Z_i(\mathbf{a}) & = \bigwedge\limits_{j\in X_i} R_j(\mathbf{a}) \end{cases}$$

Où $F_i(\mathbf{a})$ est une formule booléenne qui spécifie qu'il existe un unique j dans Y_i tel que a_{ij} soit vrai.

Théorème 6.9 *S'il existe une stratégie gagnante dans la position x_i du jeu $\mathcal{G}$, alors il existe une valuation $\mathbf{a}$ qui rend vraie R_i dans la solution du système $\mathcal{T}$.*

Preuve. La stratégie choisit pour chaque sommet x_i le successeur y_j qu'elle autorise. Ceci signifie qu'en remplaçant dans $\mathcal{S}$ l'équation de x_i par $x_i = y_j$, le système gardera le même ensemble de solutions. Ceci permet immédiatement de construire une valuation $\mathbf{a}$ telle que $R_i(\mathbf{a})$ soit vraie. On pose pour cela $a_{ij'} = \mathit{vrai}$ si et seulement si la stratégie autorise le passage de x_i à $y_{j'}$ dans $\mathcal{G}$, c'est-à-dire si $j' = j$. Le système $\mathcal{T}$ devient alors le système booléen

$$\begin{cases} R_i(\mathbf{a}) & = Z_j(\mathbf{a}) \\ Z_i(\mathbf{a}) & = \bigwedge_{j \in X_i} R_j(\mathbf{a}) \end{cases}$$

qui est en fait isomorphe au système $\mathcal{S}'$. □

Théorème 6.10 (réciproque) *S'il existe une valuation* $\mathbf{a}$ *qui rend vraie* R_i *dans la solution du système* $\mathcal{T}$*, alors il existe une stratégie gagnante dans la position* x_i *du jeu* $\mathcal{G}$*.*

Preuve. La preuve est symétrique à la précédente : une valuation $\mathbf{a}$ nous donne directement les y_j qui représentent une stratégie gagnante. □

Cependant, nous obtenons ainsi l'ensemble des stratégies gagnantes pour *une* position gagnante donnée. Mais on sait [7] qu'il existe une stratégie gagnante *uniforme*, c'est-à-dire gagnante *dans toutes les positions gagnantes.*

Les stratégies gagnantes uniformes sont codées par les valuations qui rendent vraies l'ensemble des R_i qui ne sont pas identiquement nulles.

On peut décrire cet ensemble de valuations par une relation W qui s'écrit comme ceci :

$$W(\mathbf{x}) = \bigwedge_i (\exists \mathbf{y} R_i(\mathbf{y}) \Rightarrow R_i(\mathbf{x}))$$

Chapitre 7

Exemples d'utilisation de Mec V

7.1 Mec V comme μ-calculette

Mec V peut être utilisé de manière très simple comme une "calculette" étendue par les opérateurs de points fixes du μ-calcul. Cette utilisation d'un vérificateur de modèles peut sembler étrange puisqu'elle ne fait pas intervenir de modèle, mais elle reflète justement la particularité de Mec V qui permet de calculer des relations indépendamment de tout modèle. Cette façon d'exploiter Mec V n'est pas du tout anecdotique et se révèle très pratique pour vérifier rapidement des conjectures ou se convaincre de la correction d'un résultat calculé à la main.

Nous allons dans cette section présenter quelques exemples simples tirés de [7], et montrer comment on peut tirer parti des calculs effectués par notre outil.

Bien qu'il ne s'agisse en aucun cas d'un système de calcul formel, les résultats purement calculatoires fournis par Mec V sont automatiquement généralisés à bien d'autres treillis puisque par exemple les μ-calculs sur les relations ou vectoriels se codent naturellement dans le μ-calcul booléen.

7.1.1 Points fixes de l'identité

[7, exemple 1.2.17 page 15]

Sur un ensemble ordonné E, les plus petit et plus grand points fixes de l'identité sont respectivement le plus petit et le plus grand élément de cet ensemble.

Nous allons calculer ces deux points fixes avec Mec V dans les cas où E est l'ensemble des relations sur un booléen et dans le cas où E est l'ensemble des relations sur le produit cartésien de deux ordres totaux.

```
[mec] R(x : bool) += R(x);
        R: (bool) -> bool
[mec] :rel-cardinal R
cardinal of R: 0
[mec] R(x : bool) -= R(x);
        R: (bool) -> bool
[mec] :rel-cardinal R
cardinal of R: 2
[mec] :display R
(true)
```

```
(false)
[mec]
```

On calcule ici d'abord la plus petite relation solution de l'application identité ; il s'agit de la relation vide. Le calcul du plus grand point fixe de cette même application donne bien la relation complète.

On peut aussi effectuer ce calcul avec les relations sur les couples d'entiers entre 0 et 5 par exemple :

```
[mec] R(x : [ 0, 5 ], y : [ 0, 5 ]) += R(x,y);
        R: ([ 0, 5 ], [ 0, 5 ]) -> bool
[mec] :rel-cardinal R
cardinal of R: 0
[mec] R(x : [ 0, 5 ], y : [ 0, 5 ]) -= R(x,y);
        R: ([ 0, 5 ], [ 0, 5 ]) -> bool
[mec] :rel-cardinal R
cardinal of R: 36
[mec] R(x, y) := ~R(x, y);
        R: ([ 0, 5 ], [ 0, 5 ]) -> bool
[mec] :rel-cardinal R
cardinal of R: 0
[mec]
```

De la même manière, on constate que le plus petit point fixe est la relation vide et que le plus grand est la relation complète.

7.1.2 Points fixes booléens

[7, exemple 1.2.9 page 11]

Soit $(E, \leq)$ un treillis complet distributif, a et b deux éléments de E, et soit $f : E \to E$ telle que $f(x) = a \vee (b \wedge x)$. L'application f est monotone, et ses plus petit et plus grand points fixes sont respectivement a et $a \vee b$.

Vérifions-le dans le cas booléen avec Mec V :

```
R(a : bool, b : bool) += a | (b & R(a, b));
        R: (bool, bool) -> bool
[mec] :display R
(true, true)
(true, false)
```

On constate bien ici que les paramètres a et b qui sont solutions de l'équation de plus petit point fixe sont les modèles de la formule "a", comme cela se voit sur la table de vérité donnée par Mec V.

```
[mec] R(a : bool, b : bool) -= a | (b & R(a, b));
        R: (bool, bool) -> bool
[mec] :display R
(true, true)
```

```
(false, true)
(true, false)
[mec]
```

Dans le cas du plus grand point fixe, on retrouve bien la table de vérité de la formule "$a \vee b$".

[7, exemple 1.4.8 page 31]

Dans cet exemple, le plus petit point fixe vectoriel $\mu\langle x, y, z\rangle.\langle x \vee z \vee a, x \wedge z, y \vee b\rangle$ est calculé, et vaut $\langle a \vee b, b, b\rangle$.

Vérifions cela encore une fois dans le cas où a et b sont booléens.

```
[mec] begin
.     X(a : bool, b : bool) += X(a, b) | Z(a, b) | a;
.     Y(a : bool, b : bool) += X(a, b) & Z(a, b);
.     Z(a : bool, b : bool) += Y(a, b) | b;
. end
        Z: (bool, bool) -> bool
        Y: (bool, bool) -> bool
        X: (bool, bool) -> bool
[mec] :display X
(true, true)
(false, true)
(true, false)
[mec] :display Y
(true, true)
(false, true)
[mec] :display Z
(true, true)
(false, true)
[mec]
```

Là encore, on reconnaît les tables de vérité de $a \vee b$ et de b (deux fois).

[7, exemple 1.4.15 page 37]

Terminons cette section par un exemple de calcul de points fixes imbriqués très simples. Le système d'équations $x \overset{\mu}{=} y, y \overset{\nu}{=} x$ a pour solution $\langle \top, \top\rangle$ et le système d'équations $x \overset{\mu}{=} x, y \overset{\nu}{=} x$ a pour solution $\langle \bot, \bot\rangle$.

Nous pouvons le vérifier sur le treillis des relations booléennes monadiques :

```
[mec] begin
.     Y(foo : bool) -1= X(foo);
.     X(foo : bool) +2= Y(foo);
. end
        Y: (bool) -> bool
        X: (bool) -> bool
[mec] :display X
```

```
(true)
(false)
[mec] :display Y
(true)
(false)
[mec]
```

Nous constatons bien ici que les solutions X et Y sont la relation qui contient tous les éléments.

Inversement, le calcul du second système d'équations donne bien les relations vides :

```
[mec] begin
.     Y(foo : bool) -1= X(foo);
.     X(foo : bool) +2= X(foo);
. end
        Y: (bool) -> bool
        X: (bool) -> bool
[mec] :display X
[mec] :display Y
[mec]
```

7.2 Jeux

Nous allons voir dans cette section deux exemples de jeux modélisés en AltaRica : l'un de nature un peu théorique qui est un jeu d'allumettes dans lequel il existe une stratégie gagnante si et seulement si le nombre d'allumettes au départ n'est pas un nombre de Fibonacci ; l'autre est un exemple de modèle AltaRica qui épuise la mémoire de Mec V sur des machines disposant de trois giga-octets de mémoire adressable.

7.2.1 Jeu de Fibonacci

Le jeu de Fibonacci se joue à deux joueurs et on dispose au départ d'un tas de n allumettes. Les règles du jeu sont les suivantes :

1. A tour de rôle, un joueur enlève au moins une allumette et au plus le double du nombre d'allumettes enlevées par l'autre joueur au tour précédent.
2. Au premier tour, le premier joueur peut enlever le nombre d'allumettes qu'il souhaite à condition d'en enlever au moins une et d'en laisser au moins une.

Le joueur qui ne peut plus jouer perd.

Modélisation des configurations du jeu

A chaque instant, l'état du jeu est constitué du nombre d'allumettes restantes et du nombre d'allumettes qui ont été retirées au coup précédent. La figure 7.1 donne le source AltaRica de cette modélisation.

Les variables d'état s et d représentent respectivement le nombre d'allumettes restant dans le tas et la quantité d'allumettes enlevées au tour précédent. On se donne un événement `c` qui correspond à un coup joué par un des deux joueurs.

```
node Fibonacci
    state s, d : [1, MAXI];
    flow f : [1, MAXI];

    event c;

    trans
        true |- c -> d := f, s := s - f;

    assert
        (s = MAXI) => (f < MAXI);
        (s < MAXI) => (f <= d+d);

    init
        d := MAXI,
        s := MAXI;
edon
```

FIG. 7.1 – Modélisation en AltaRica du jeu de Fibonacci

La seule macro-transition apparaissant dans le modèle décrit ce qui se passe lors d'un coup : on enlève un certain nombre d'allumettes (ici f) et on stocke ce nombre dans la variable d pour pouvoir l'utiliser au coup suivant.

Il est très intéressant de constater qu'ici, la variable de flux f est utilisée comme "générateur de non-déterminisme" : sa valeur n'est contrainte que par l'assertion et peut changer à chaque fois qu'une transition se produit.

L'assertion assure que les règles sont bien respectées en contraignant la valeur de f à être plus petite ou égale que le double de la dernière quantité d'allumettes enlevées (cas général), ou à être plus petite que le nombre d'allumettes dans le tas (au départ). On peut noter que du fait de l'existence des post-conditions, le coup `c` ne peut être effectué que s'il aboutit dans une configuration autorisée, et les domaines des variables et l'assertion garantissent qu'il s'agit d'une configuration autorisée par les règles du jeu.

Calcul des positions gagnantes du premier joueur

Pour étudier ce système avec Mec V, on commence par charger le modèle après avoir précisé le nombre d'allumettes de départ. Puis on se donne la relation de transitions, restreinte aux seules transitions étiquetées par `c`. Cela permet de supprimer les boucles ε et de raccourcir l'écriture.

```
[mec] MAXI := 15;
        MAXI: integer
[mec] :ar-load exemples/fibonacci.alt
[mec] T(u, v) := <e>((e. = c) & Fibonacci!t(u, e, v));
        Fibonacci!t: (Fibonacci!c, Fibonacci!ev, Fibonacci!c) -> bool
        Fibonacci!init: (Fibonacci!c) -> bool
        T: (Fibonacci!c, Fibonacci!c) -> bool
```

```
[mec]
```

Parmi toutes les configurations du jeu, nous sommes intéressés par celles qui sont gagnantes, c'est-à-dire par celles qui sont atteintes et pour lesquelles quoi que joue le premier joueur, le coup suivant permette de ramener le jeu dans une configuration gagnante. Ceci se définit tout naturellement par un plus petit point fixe, qui se lit "pour tout successeur, il existe un successeur de ce successeur qui est lui-même gagnant".

```
[mec] G(u) += [v](T(u, v) => <w>(T(v, w) & G(w)));
        G: (Fibonacci!c) -> bool
```

Si la configuration initiale est gagnante, cela signifie que le joueur qui commence à jouer en premier a une stratégie gagnante. La relation `W` ci-dessous nous donne donc l'ensemble de coups que le premier joueur a le droit de jouer lors du premier tour s'il souhaite pouvoir encore être sûr de gagner.

```
[mec] W(u) := G(u) & Fibonacci!init(u);
        W: (Fibonacci!c) -> bool
[mec] :rel-cardinal  W
cardinal of W: 1
[mec] :display W
({s = 15, d = 15, f = 2})
[mec]
```

Ici, il faut que le premier joueur commence par enlever deux allumettes (choix dicté par la valeur du flux f dans la position gagnante).

On peut montrer que les seules parties pour lesquelles le premier joueur n'a pas de stratégie gagnante sont celles où le nombre d'allumettes au départ est un nombre de Fibonacci, d'où le nom du jeu.

7.2.2 Jeu du solitaire

Le jeu du solitaire est bien connu : il s'agit de 32 pions répartis sur une grille en forme de croix, dans lequel il faut essayer de se ramener à une configuration où il n'y a qu'un seul pion.

On enlève un pion lorsqu'on "le prend" avec un autre pion, c'est-à-dire quand on déplace cet autre pion sur la case qui est symétrique par rapport au pion supprimé. Il n'est pas possible de prendre un pion en diagonale, et deux pions ne peuvent pas se chevaucher. Au départ la case du milieu est vide (figure 7.2).

L'ensemble des états accessibles de cet exemple ne peut pas être calculé avec Mec V actuellement ; nous le donnons parce que sa modélisation est intéressante, et pour donner un exemple des limites de notre outil.

On commence par modéliser une case du plateau par un nœud AltaRica qui a pour état un entier codant la présence ou l'absence de pion dans cette case, qui peut changer lors des événements `p` et `v`.

```
node C
    state x : [0, 1];
    event p, v;
```

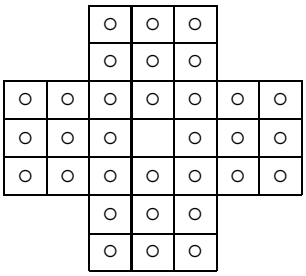
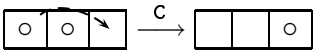


FIG. 7.2 – Configuration de départ du jeu du solitaire

```
    trans
        x = 1 |- v -> x:=0;
        x = 0 |- p -> x:=1;
    assert true;
edon
```

La modélisation en AltaRica de ce jeu est donnée dans la figure 7.3 page 95.

L'événement `c` représente ici aussi un coup joué. On le synchronise donc avec chacun des coups possibles dans le jeu du solitaire. Par exemple, pour la première ligne, la première synchronisation correspond à la transition suivante :



On aurait très bien pu coder l'état d'une case par un booléen, mais le fait de le coder par un entier va nous permettre d'utiliser une astuce de modélisation dans le nœud représentant le jeu du solitaire : l'assertion nous permet de relier le contenu des cases et le nombre de cases occupées par une simple addition. Ceci est utile pour spécifier la condition de gain ($n = 1$) et la configuration initiale ($n = 32$ et $c_{44}.x = 0$).

Malheureusement, le calcul d'atteignabilité échoue même sur une machine disposant d'un espace adressable de trois giga-octets. Voici néanmoins la spécification utilisée :

```
I(s : Solitaire!c) := (s.n = 32) & (s.c44.x = 0);
F(s : Solitaire!c) := (s.n = 1);
Reach(s) += I(s) | <s'>(Reach(s') & <e> Solitaire!t(s',e,s));
OK(s) := Reach(s) & F(s);
```

On devrait donc trouver dans la relation `OK` les configurations finales atteignables dans le jeu du solitaire.

7.3 Expression de propriétés usuelles avec Mec V

Nous allons dans cette section utiliser Mec V pour vérifier des propriétés "usuelles" d'atteignabilité et de vivacité sur un modèle classique d'ascenseur. Nous reprenons pour la majeure partie l'exemple proposé dans la thèse de Gérald Point [45].

7.3.1 Un modèle d'ascenseur

Le nombre d'étages est fixé à trois, plus le rez-de-chaussée :

```
const MaxFloor = 3;
```

Notre modèle va être constitué d'une cabine (`Cabin`), de quatre étages (`Floor`s), qui auront besoin de deux modèles de base : des portes (`Door`s) et des boutons lumineux (`Button`s).

Une porte est dans l'état fermée ou ouverte, et ses changements d'états sont liés aux événements `open` (ouvrir) et `close` (fermer). Dans son état initial, une porte est fermée.

```
node Door
    state
        closed : bool : public;
    event
        open, close;
    trans
        closed |- open -> closed := false;
        ~closed |- close -> closed := true;
    init
        closed := true;
edon
```

Un bouton lumineux possède comme la porte deux états (allumé ou éteint) et deux événements : `push` (appuyer) et `off` (éteindre) qui représentent respectivement l'appui sur le bouton et l'extinction de la lumière. L'appui peut avoir lieu même si le bouton est déjà allumé, puisqu'il s'agit normalement d'un événement extérieur, non contrôlé par le système.

```
node Button
    flow
        light : bool;
    state
        on : bool;
    event
        push : public;
        off;
    trans
        true |- push -> on := true;
        on |- off -> on := false;
    assert
        light = on;
    init
        on := false;
edon
```

La cabine d'ascenseur contient une porte, et se souvient de l'étage où elle se trouve. Son comportement modélise le fait que si elle peut monter (si elle n'est pas au dernier étage et que sa porte est fermée), elle le fait sur l'événement `up`. De même, l'événement `down` rend compte du fait qu'elle descend. Les événements `open` et `close` de la porte sont synchronisés tel quel avec

deux nouveaux événements de même nom afin de les rendre disponibles au nœud parent, et aussi d'éviter qu'ils ne se synchronisent automatiquement avec l'événement ε local.

```
node Cabin
    state
        floor : [0, MaxFloor] : parent;
    sub
        D : Door;
    event
        up, down, close, open;
    trans
        (floor < MaxFloor) & D.closed |- up -> floor := floor + 1;
        (floor > 0) & D.closed |- down -> floor := floor - 1;
        true |- open, close -> ;
    sync
        <close, D.close>;
        <open,  D.open>;
    init
        floor := 0;
edon
```

Chaque étage `Floor` est constitué d'une porte `D` et d'un bouton `BF`. Pour simplifier le modèle, le bouton correspondant à cet étage mais qui se trouve dans la cabine de l'ascenseur est aussi intégré (`BC`) dans ce nœud.

```
node Floor
    flow
        request : bool;
    sub
        BC, BF : Button;
        D : Door;
    event
        close, open;
    trans
        true |- close, open -> ;
    sync
        <close, D.close, BC.off?, BF.off?>;
        <open,  D.open,  BC.off?, BF.off?> >= 1;
    assert
        request = (BC.light | BF.light);
edon
```

La version actuelle de Mec V ne traite pas encore la sémantique particulière des événements publics. Or le nœud `Button` contient un événement public `push`. Cet exemple est donc l'occasion de montrer comment cette sémantique peut être simulée. Il faut introduire un événement local par événement public de sous-nœud, comme cela est indiqué dans le tableau page 38, puis synchroniser ces nouveaux événements avec les événements "publics" des sous-nœuds. Ceci garantit que ces derniers ne se synchronisent pas avec l'événement ε du nœud `Floor`.

```
    event
        push1, push2;
    trans
        true |- push1, push2 -> ;
    sync
        <push1, BC.push>; <push2, BF.push>;
```

Le système global contient les constituants décrits ci-dessus, et régit l'interaction entre les composants. Il utilise deux flux `mayGoUp` et `mayGoDown` qui servent uniquement de calcul intermédiaire pour savoir à tout moment s'il est possible que la cabine monte ou descende. On utilise ici des priorités entre les événements pour préciser qu'à un étage donné, l'ascenseur préfèrera ouvrir la porte plutôt que de continuer son chemin. Ceci garantit que les personnes n'attendront pas indéfiniment pendant que la cabine de l'ascenseur bouge pour rien. Une variable d'état `climb` garde la direction dans laquelle l'ascenseur évolue, afin d'éviter qu'il fasse des aller-retours entre certains étages en en oubliant d'autres.

```
node Lift
    flow
        mayGoUp, mayGoDown : bool : private;
    state
        climb : bool;
    sub
        F0, F1, F2, F3 : Floor;
        C : Cabin;
    event
        open0, open1, open2, open3;
        { up, down } < {open0, open1, open2, open3};
    trans
        (C.floor = 0) |- open0 -> ;
        (C.floor = 1) |- open1 -> ;
        (C.floor = 2) |- open2 -> ;
        (C.floor = 3) |- open3 -> ;
        climb & mayGoUp               |- up   -> ;
        ~climb & ~mayGoDown & mayGoUp |- up   -> climb := true;
        ~climb & mayGoDown            |- down -> ;
        climb & ~mayGoUp & mayGoDown  |- down -> climb := false;
    sync
        <up,    C.up>;
        <down,  C.down>;
        <open0, C.open, F0.open>;
        <open1, C.open, F1.open>;
        <open2, C.open, F2.open>;
        <open3, C.open, F3.open>;
        <C.close, F0.close?, F1.close?, F2.close?, F3.close?> = 1;
    assert
        mayGoUp   = ((C.floor=0 & (F3.request | F2.request | F1.request)) |
                 (C.floor=1 & (F3.request | F2.request)) |
                 (C.floor=2 & (F3.request)));
```

```
        mayGoDown = ((C.floor=3 & (F0.request | F1.request | F2.request)) |
                 (C.floor=2 & (F0.request | F1.request)) |
                 (C.floor=1 & (F0.request)));
    init
        climb := false;
edon
```

7.3.2 Les propriétés à vérifier

Voici la liste des propriétés à vérifier, que nous reprenons d'un exemple disponible sur le site WWW du projet AltaRica. Les propriétés traitées dans cet exemple sont tirées de la thèse de François Laroussinie [35].

1. (a) Quand un bouton est enfoncé, son voyant s'allume.

   (b) Quand la requête correspondante est satisfaite, le voyant s'éteint.

2. A chaque étage, la porte n'est jamais ouverte si la cabine n'est pas là.
3. Chaque requête doit être satisfaite "un jour".
4. L'ascenseur ne dessert que les étages pour lesquels il existe une requête.
5. Lorsqu'il n'y a pas de requête, la cabine reste à l'étage où elle se trouve.
6. Lorsque la cabine se déplace, elle doit s'arrêter aux étages par lesquels elle passe s'ils font l'objet d'une requête.
7. Lorsqu'il existe plusieurs requêtes, l'ascenseur doit traiter prioritairement celle(s) permettant de continuer dans la même direction que le dernier déplacement.

Quand un bouton est enfoncé, son voyant s'allume

Cette propriété est une propriété de sûreté [49], et il suffit donc de la vérifier sur le nœud `Button` directement plutôt que de la vérifier pour chacune des instances apparaissant dans le nœud `Lift`.

On vérifie ici qu'aucune configuration ne satisfait la condition inverse, qui serait qu'à la suite d'un événement `push`, la lampe soit éteinte.

> Notez qu'ici cette condition inverse est fausse sur toutes les configurations, on constate donc *a posteriori* qu'il n'était pas nécessaire de restreindre notre requête aux états accessibles du modèle.

```
[mec] notP1a(s' : Button!c) := <s><e>(e. = push & Button!t(s, e, s')) &
.    ~s'.light;
       Button!t: (Button!c, Button!ev, Button!c) -> bool
       Button!init: (Button!c) -> bool
       notP1a: (Button!c) -> bool
[mec] :rel-cardinal notP1a
cardinal of notP1a: 0
```

Quand la requête correspondante est satisfaite, le voyant s'éteint

De manière très semblable, cette propriété de sûreté se vérifie directement sur le nœud `Floor`. On prend comme événement caractéristique d'une requête satisfaite la fermeture de la porte de l'étage.

```
[mec] notP1b(s' : Floor!c) := <s><e>(e. = close & Floor!t(s, e, s')) &
.    s'.BC.light & s'.BF.light;
        Floor!t: (Floor!c, Floor!ev, Floor!c) -> bool
        Floor!init: (Floor!c) -> bool
        notP1b: (Floor!c) -> bool
[mec] :rel-cardinal notP1b
cardinal of notP1b: 0
```

Pour les propriétés suivantes, nous allons nous intéresser à tout le système, et nous commençons donc par calculer ses configurations accessibles ainsi que la relation de transition restreinte à ces configurations.

```
[mec] R(s) += Lift!init(s) | <e><t>(R(t) & Lift!t(t,e,s));
        Lift!t: (Lift!c, Lift!ev, Lift!c) -> bool
        Lift!init: (Lift!c) -> bool
        R: (Lift!c) -> bool
[mec] :rel-cardinal R
cardinal of R: 3072
[mec] T(s, e, t) := Lift!t(s,e,t) & R(s);
        T: (Lift!c, Lift!ev, Lift!c) -> bool
[mec] :rel-cardinal T
cardinal of T: 30714
```

Le système complet a donc 3072 configurations accessibles et 30714 transitions les relient.

A chaque étage, la porte n'est jamais ouverte si la cabine n'est pas là

On ne donne ici la formule que pour le rez-de-chaussée, mais elle est vérifiée par tous les autres étages. On pourrait tous les faire figurer dans la même conjonction.

```
[mec] notP2(s) := R(s) & ~s.F0.D.closed & s.C.floor != 0;
        notP2: (Lift!c) -> bool
[mec] :rel-cardinal notP2
cardinal of notP2: 0
```

Chaque requête doit être satisfaite "un jour"

Nous souhaitons ici vérifier que depuis n'importe quelle configuration du système où une requête a lieu, il finira toujours par atteindre une configuration où la requête a été satisfaite.

Nous allons donc calculer l'ensemble des états à partir desquels il existe un chemin qui ne contient que des requêtes actives. Si un tel chemin existe, il restera à vérifier qu'on ne peut jamais y aboutir en partant d'une configuration où une requête a lieu.

```
[mec] EGrequest(s) -= R(s) & s.F0.request & <e><s'>(T(s,e,s') &
    e. != "" & EGrequest(s'));
        EGrequest: (Lift!c) -> bool
[mec] :rel-cardinal EGrequest
cardinal of EGrequest: 2304
```

Il y a 2304 configurations à partir desquelles partent des chemins sur lesquels une requête ne peut être satisfaite. Ceci est en fait du aux événements `push` qui peuvent survenir à tout moment, et se répéter infiniment souvent sans que l'ascenseur ne bouge.

Si on interdit tous les événements `push` sur un chemin, on voit que le système est obligé d'aboutir dans une configuration où la requête est satisfaite :

```
[mec] EGrequest(s) -= R(s) & s.F0.request & <e><s'>(T(s,e,s') &
    e. != "" &
    e.F0.BC. != push & e.F0.BF. != push &
    e.F1.BC. != push & e.F1.BF. != push &
    e.F2.BC. != push & e.F2.BF. != push &
    e.F3.BC. != push & e.F3.BF. != push &
    EGrequest(s'));
        EGrequest: (Lift!c) -> bool
[mec] :rel-cardinal EGrequest
cardinal of EGrequest: 0
```

L'ascenseur ne dessert que les étages pour lesquels il existe une requête

Pour que cette condition ne soit pas vérifiée, il faudrait qu'une configuration, à un étage donné, ne soit pas une requête et qu'elle ait un successeur par l'événement `open`.

```
[mec] notP4(s) := R(s) & ~s.F0.request & <e><t>(e.F0. = open & T(s, e, t));
        notP4: (Lift!c) -> bool
[mec] :rel-cardinal notP4
cardinal of notP4: 0
```

Lorsqu'il n'y a pas de requête, la cabine reste à l'étage où elle se trouve

```
[mec] notP5(s) := R(s) &
.    ~s.F0.request & ~s.F1.request & ~s.F2.request & ~s.F3.request &
.    <e><t>((e.C. = up | e.C. = down) & T(s, e, t));
        notP5: (Lift!c) -> bool
[mec] :rel-cardinal notP5
cardinal of notP5: 0
```

Lorsque la cabine se déplace, elle doit s'arrêter aux étages par lesquels elle passe s'ils font l'objet d'une requête

Cette propriété peut être reformulée comme suit : si on quitte un étage, il n'a plus de requête qui le concerne.

```
[mec] notP6(s) := R(s) & <e><t>((e.C. = up | e.C. = down) & T(s, e, t)) &
.     s.C.floor = 0 & s.F0.request;
        notP6: (Lift!c) -> bool
[mec] :rel-cardinal notP6
cardinal of notP6: 0
```

Lorsqu'il existe plusieurs requêtes, l'ascenseur doit traiter prioritairement celle(s) permettant de continuer dans la même direction que le dernier déplacement

On se donne d'abord une relation précisant les configurations dans lesquelles l'ascenseur sait qu'il devra monter à un moment ou à un autre. Il s'agit des configurations dans lesquelles il existe une requête pour un étage qui se trouve au-dessus de la cabine.

```
[mec] eventuallyUp(s) := R(s) & (
.     (s.C.floor = 0 & (s.F1.request | s.F2.request | s.F3.request)) |
.     (s.C.floor = 1 & (s.F2.request | s.F3.request)) |
.     (s.C.floor = 2 & s.F3.request));
        eventuallyUp: (Lift!c) -> bool
```

Notre but est de trouver s'il existe des configurations qui suivent un événement `up` (c'est-à-dire des configurations dans lesquelles l'ascenseur est en train de monter), et qui mènent à un événement `down` alors qu'il reste encore des étages supérieurs à desservir.

On va donc se donner une relation `coreachDown` qui contiendra les configurations à partir desquelles il est possible d'accéder à un événement `down` en passant par des configurations dans lesquelles il y a toujours une requête pour un étage supérieur et sans passer par un événement `up`.

```
[mec] coreachDown(s) += eventuallyUp(s) & (
.     <e><s'>(e.C. = down & T(s, e, s')) |
.     <e><s'>(e.C. != up & T(s, e, s') & coreachDown(s')));
        coreachDown: (Lift!c) -> bool
```

Toute configuration qui est dans `coreachDown` et qui est accessible directement par un événement `up` est fautive : l'ascenseur devrait continuer à monter.

```
[mec] notP7(s) := coreachDown(s) & <s'><e>(e.C. = up & T(s', e, s));
        notP7: (Lift!c) -> bool
[mec] :rel-c notP7
cardinal of notP7: 0
```

La même propriété est vérifiée lorsqu'on considère l'autre sens de déplacement de l'ascenseur.

7.4 Bisimulation

Nous allons dans cette section tirer parti du fait que l'on peut exprimer dans Mec V des relations qui ne sont pas monadiques et qui font intervenir les quantificateurs du premier ordre : nous allons décrire deux implémentations différentes d'une file et montrer qu'elles sont bisimilaires.

Chacune de ces deux implémentations utilise des cases (*buffers*) qui peuvent soit être vides, soit contenir une valeur parmi deux possibles.

7.4.1 File avec un pointeur

La première implémentation que nous allons décrire est constituée d'une séquence de buffers pouvant décaler leur contenu (lors d'une opétation de défilage), d'un successeur à son prédecesseur. Le buffer (libre) dans lequel on mettra la valeur à enfiler sera pointé par son indice dans la séquence.

Voici le modèle AltaRica d'un tel buffer ; l'absence de contenu est notée par la valeur 0 :

```
node BufferTypeCompteur
  flow   entree : [0,2];
         depot : bool;
  state  valeur : [0,2] : parent;
  event  ecrire, decaler;
  trans  valeur = 0 & depot |- ecrire -> valeur := 1;
         valeur = 0 & depot |- ecrire -> valeur := 2;
         true |- decaler -> valeur := entree;
  init   valeur := 0;
edon
```

On voit que lorsque l'événement `ecrire` se déclenche, il peut de manière non-déterministe écrire un ou deux dans le buffer. En conséquence, le modèle de file aura tous les comportements possibles, de manière non-déterministe.

La file est représentée par un certain nombre de buffers `BufferTypeCompteur` montés en cascade, et par un pointeur (`tete`) qui retient où se trouve la fin de la file, car c'est à cet endroit qu'un élément doit être écrit lorsqu'il est enfilé.

Voici la description AltaRica d'une telle file à trois buffers (`M` = 3) :

```
node FIFOCompteurValeurs
  sub    B0, B1, B2 : BufferTypeCompteur;
  flow   vide, plein : bool;
         tete : [0, 2];
  state  nb : [0, M];
  event  put, get;
  trans  true |- put -> nb := nb + 1;
         true |- get -> nb := nb - 1;
  sync   <put, B0.ecrire>;
         <put, B1.ecrire>;
         <put, B2.ecrire>;
         <get, B0.decaler, B1.decaler, B2.decaler>;
  assert vide = (nb = 0);
         plein = (nb = M);
         tete = B2.valeur;
         B0.depot = ((M-nb) = 1);
         B1.depot = ((M-nb) = 2);
         B2.depot = ((M-nb) = 3);
         B0.entree = 0;
         B1.entree = B0.valeur;
```

```
        B2.entree = B1.valeur;
  init  nb := 0;
edon
```

L'utilisation des booléens `depot` permet de n'activer que le buffer situé en tête de file lors d'une écriture, comme le ferait un démultiplexeur dans un circuit logique.

7.4.2 File avec deux pointeurs

La deuxième implémentation est très classique puisqu'elle représente un tableau, dans lequel la tête (`retrait`) et la queue (`depot`) de la file se déplacent dans le tableau de manière à simuler un tableau circulaire.

Les buffers doivent dans ce cas être étendus par une opération de lecture puisque ce n'est pas toujours le même buffer qui contient la donnée en tête de file. Il faut donc généraliser le mécanisme de lecture de la même manière que l'était déjà celui d'écriture. En revanche, le mécanisme de décalage disparait. Voici la description AltaRica d'un tel buffer :

```
node BufferTypeCirculaire
  flow   depot, retrait : bool;
  state  valeur : [0,2] : parent;
  event  ecrire, lire;
  trans  valeur = 0 & depot |- ecrire -> valeur := 1;
         valeur = 0 & depot |- ecrire -> valeur := 2;
         valeur != 0 & retrait |- lire -> valeur := 0;
  init   valeur := 0;
edon
```

et la description d'une file implémentée par un tableau circulaire pour le cas où il y a trois buffers (`M` = 3) :

```
node FIFOCirculaireValeurs
  sub    B0, B1, B2 : BufferTypeCirculaire;
  flow   tete : [0,2];
  state  vide, plein : bool : parent;
         depot, retrait : [0,M-1];
  event  put, get;
  trans  ~plein |- put -> depot   := if (depot+1<M) then depot+1
                                                   else 0,
                         plein   := if (depot+1<M) then (depot+1)=retrait
                                                    else 0=retrait,
                         vide    := false;
         ~vide  |- get -> retrait := if (retrait+1<M) then retrait+1
                                                      else 0,
                         plein   := false,
                         vide    := if (retrait+1<M) then (retrait+1)=depot
                                                      else 0 = depot;
  sync   <put, B0.ecrire>;
```

```
         <put, B1.ecrire>;
         <put, B2.ecrire>;
         <get, B0.lire>;
         <get, B1.lire>;
         <get, B2.lire>;
  assert vide => (tete = 0);
         (retrait = 0 & ~vide) => (tete = B0.valeur);
         (retrait = 1 & ~vide) => (tete = B1.valeur);
         (retrait = 2 & ~vide) => (tete = B2.valeur);
         B0.depot = (depot=0);
         B0.retrait = (retrait=0);
         B1.depot = (depot=1);
         B1.retrait = (retrait=1);
         B2.depot = (depot=2);
         B2.retrait = (retrait=2);
  init   depot := 0,
         retrait := 0,
         vide := true,
         plein := false;
edon
```

7.4.3 Calcul de la relation de bisimulation

Nous allons effectuer avec Mec V le calcul de la relation de bisimulation entre les deux implémentations de files décrites ci-dessus. Du fait des choix de modélisation effectués, de nombreuses configurations ne sont pas accessibles depuis la configuration initiale. Nous commençons donc par calculer l'ensemble des états accessibles ainsi que la relation de transition de chacun des modèles, restreintes aux états accessibles.

```
[mec] M := 3;
        M: integer
[mec] :ar-load exemples/BufferTypeCirculaire.alt
[mec] :ar-load exemples/BufferTypeCompteur.alt
[mec] :ar-load exemples/FIFOCirculaireValeurs.alt
[mec] :ar-load exemples/FIFOCompteurValeurs.alt
[mec] ReachCo(t) +=
.   FIFOCompteurValeurs!init(t) |
.   <s><e>(FIFOCompteurValeurs!t(s,e,t) & ReachCo(s));
        FIFOCompteurValeurs!t: (FIFOCompteurValeurs!c, FIFOCompteurValeurs!ev, F
IFOCompteurValeurs!c) -> bool
        FIFOCompteurValeurs!init: (FIFOCompteurValeurs!c) -> bool
        ReachCo: (FIFOCompteurValeurs!c) -> bool
[mec] ReachCi(t) +=
.   FIFOCirculaireValeurs!init(t) |
.   <s><e>(FIFOCirculaireValeurs!t(s,e,t) & ReachCi(s));
        FIFOCirculaireValeurs!t: (FIFOCirculaireValeurs!c, FIFOCirculaireValeurs
!ev, FIFOCirculaireValeurs!c) -> bool
        FIFOCirculaireValeurs!init: (FIFOCirculaireValeurs!c) -> bool
```

```
        ReachCi: (FIFOCirculaireValeurs!c) -> bool
[mec] ReachTo(s,e,t) := FIFOCompteurValeurs!t(s,e,t) & ReachCo(s);
        ReachTo: (FIFOCompteurValeurs!c, FIFOCompteurValeurs!ev, FIFOCompteurVal
eurs!c) -> bool
[mec] ReachTi(s,e,t) := FIFOCirculaireValeurs!t(s,e,t) & ReachCi(s);
        ReachTi: (FIFOCirculaireValeurs!c, FIFOCirculaireValeurs!ev, FIFOCircula
ireValeurs!c) -> bool
```

La relation de bisimulation que nous allons calculer est la relation de bisimulation *interfacée* [6, 45], c'est-à-dire que nous allons aussi considérer dans cette relation la partie visible des configurations, en plus des événements comme c'est le cas pour la bisimulation classique. Nous commençons donc par définir deux relations sur les couples de configurations et d'événements.

```
[mec] stateEq(a, a') :=
.   ReachCo(a) & ReachCi(a') &
.   (a.vide = a'.vide) & (a.plein = a'.plein) & (a.tete = a'.tete);
        stateEq: (FIFOCompteurValeurs!c, FIFOCirculaireValeurs!c) -> bool
[mec] eventEq(e : FIFOCompteurValeurs!ev, e' : FIFOCirculaireValeurs!ev) :=
.   (e. = put & e'. = put) |
.   (e. = get & e'. = get) |
.   (e. = ""   & e'. = "") ;
        eventEq: (FIFOCompteurValeurs!ev, FIFOCirculaireValeurs!ev) -> bool
```

Il reste ensuite à écrire simplement la définition d'une relation de bisimulation. Deux configurations sont bisimilaires si elles sont indistingables par leur partie visible et que pour tout successeur d'une configuration, il existe dans l'autre système un successeur bisimilaire tel qu'on puisse arriver dans ces deux successeurs par des événements indistingables.

```
[mec] bisim(a, a') -=
.   stateEq(a,a') &
.   ([e][b] (ReachTo(a,e,b) =>
.     <e'><b'> (ReachTi(a',e',b') & eventEq(e,e') & bisim(b,b')))) &
.   ([e'][b'] (ReachTi(a',e',b') =>
.     <e><b> (ReachTo(a,e,b) & eventEq(e,e') & bisim(b,b')))) ;
        bisim: (FIFOCompteurValeurs!c, FIFOCirculaireValeurs!c) -> bool
```

Il reste à vérifier que les configurations initiales sont bisimilaires, nous définissons donc une relation pour coder le booléen qui répond à cette question, et on constate qu'effectivement les deux modèles sont bisimilaires.

```
[mec] isBisim(x) :=
.   x = (([a] (FIFOCompteurValeurs!init(a) =>
.         <a'> (FIFOCirculaireValeurs!init(a') & bisim(a,a')))) &
.        ([a'] (FIFOCirculaireValeurs!init(a') =>
.         <a> (FIFOCompteurValeurs!init(a) & bisim(a,a'))))));
        isBisim: (bool) -> bool
[mec] :display isBisim
(true)
```

On peut aussi afficher les tailles des différentes relations qui entrent en jeu dans cet exemple.

```
[mec] :rel-cardinal ReachCo
cardinal of ReachCo: 15
[mec] :rel-cardinal ReachTo
cardinal of ReachTo: 43
[mec] :rel-cardinal FIFOCompteurValeurs!t
cardinal of FIFOCompteurValeurs!t: 243
[mec] :rel-cardinal ReachCi
cardinal of ReachCi: 45
[mec] :rel-cardinal ReachTi
cardinal of ReachTi: 129
[mec] :rel-cardinal FIFOCirculaireValeurs!t
cardinal of FIFOCirculaireValeurs!t: 1620
[mec] :rel-cardinal bisim
cardinal of bisim: 45
[mec]
```

7.5 Performances

Nous donnons ici quelques temps de calculs observés avec Mec V. Les exemples que nous avons mis en place dans ce but devraient nous permettre de vérifier que les futures versions de Mec V ne feront pas perdre en termes de performances.

La machine utilisée pour ces tests est à base de Pentium IV à 2,8GHz avec un giga-octets de mémoire et fonctionne avec le système d'exploitation NetBSD. L'impact du système d'exploitation se situe uniquement dans l'allocation de la mémoire et est donc assez limité puisque Mec V utilise actuellement des blocs de mémoire pré-alloués et gère la mémoire de manière efficace.

7.5.1 Cas du jeu de Fibonacci

La constante `MAXI` (page 79) permet de générer des modèles de plus en plus gros à loisir. Nous avons utilisé cette possibilité pour montrer de manière flagrante l'impact que peut avoir le cache de termes (sous-section 5.2.2, page 43) dans les performances de Mec V. Cette expérimentation montre bien que lorsque le cache de termes est rempli trop fréquemment, les calculs deviennent extrêmement lents.

La figure 7.4 page 96 montre le temps pris pour calculer les positions gagnantes dans le jeu de Fibonacci en fonction du nombre d'allumettes de départ pour les deux tailles du cache qui sont cent mille et cinq millions de termes. Nous donnons quelques-unes des valeurs représentées sur le graphique dans le tableau 7.5 page 96 pour plus de précision.

7.5.2 Cas de la bisimulation

Nous avons écrit un programme simple qui génère chacune des deux implémentations de files pour un nombre quelconque de buffers (page 88).

Voici résumés dans un tableau les temps pris pour le calcul de la relation de bisimulation avec un cache de termes limité soit à 100 000 d'entrées, soit à 5 millions d'entrées.

M	#ReachCo	#ReachTo	#ReachCi	#ReachTi	#bisim	temps $100k$	temps $5M$
1	3	7	3	7	3	0,04s	0,08s
2	7	19	14	38	14	0,27s	0,28s
3	15	43	45	129	45	1,16s	1,09s
4	31	91	124	364	124	4,33s	3,52s
5	63	187	315	935	315	13,96s	9,74s
6	127	379	762	2274	762	39,28s	24,25s
7	255	763	1 785	5 341	1 785	180,42s	59,58s
8	511	1 531	4 088	12 248	4 088	1 559,34s	143,29s

On constate ici encore que la taille du cache de termes est déterminante et que lorsque celui-ci est trop souvent plein, les performances du module BDD deviennent inacceptables.

```
node Solitaire
    flow n : [0, 33];
    sub
        c73,c74,c75,
        c63,c64,c65,
c51,c52,c53,c54,c55,c56,c57,
c41,c42,c43,c44,c45,c46,c47,
c31,c32,c33,c34,c35,c36,c37,
        c23,c24,c25,
        c13,c14,c15:C;
    event c;
    trans
        true |- c ->;
    sync
        // ligne 1
        <c, c13.v, c14.v, c15.p>;
        <c, c13.p, c14.v, c15.v>;

        // ligne 2
        <c, c23.v, c24.v, c25.p>;
        <c, c23.p, c24.v, c25.v>;

        // ligne 3
        <c, c31.v, c32.v, c33.p>;
        <c, c32.v, c33.v, c34.p>;
        <c, c33.v, c34.v, c35.p>;
        <c, c34.v, c35.v, c36.p>;
        <c, c35.v, c36.v, c37.p>;

        <c, c31.p, c32.v, c33.v>;
        <c, c32.p, c33.v, c34.v>;
        <c, c33.p, c34.v, c35.v>;
        <c, c34.p, c35.v, c36.v>;
        <c, c35.p, c36.v, c37.v>;

        ...

    assert
        n =(c73.x + c74.x + c75.x + c63.x + c64.x + c65.x +
            c51.x + c52.x + c53.x + c54.x + c55.x + c56.x + c57.x +
            c41.x + c42.x + c43.x + c44.x + c45.x + c46.x + c47.x +
            c31.x + c32.x + c33.x + c34.x + c35.x + c36.x + c37.x +
            c23.x + c24.x + c25.x + c13.x + c14.x + c15.x);
edon
```

FIG. 7.3 – Modélisation en AltaRica du jeu du solitaire

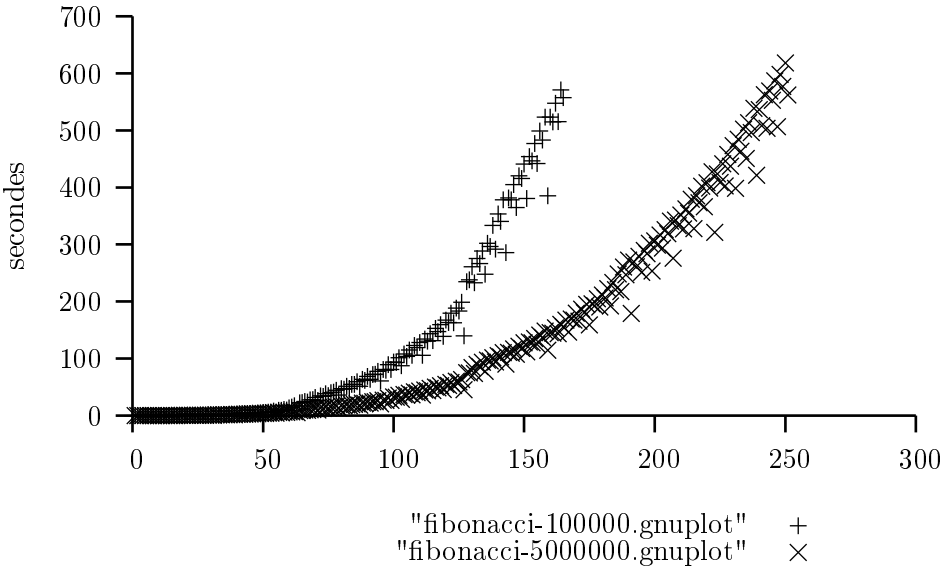


FIG. 7.4 – Impact du cache de termes sur le calcul de positions gagnantes

MAXI	#transitions	temps $100k$	temps $5M$
10	2 594	0,06s	0,11s
20	39 040	0,32s	0,35s
30	192 644	0,93s	0,87s
40	600 480	2,60s	2,18s
50	1 453 344	5,39s	3,82s
60	2 995 820	13,52s	6,55s
70	5 526 194	31,66s	12,22s
80	9 396 560	47,12s	17,24s
90	15 012 694	68,61s	23,50s
100	22 834 200	94,02s	32,90s
110	33 374 344	128,18s	43,17s
120	47 200 240	167,19s	54,52s
130	64 932 644	261,15s	84,38s
140	87 246 180	353,57s	105,30s

FIG. 7.5 – Temps de calcul des positions gagnantes du jeu de Fibonacci

Annexes

Annexe A

Le langage AltaRica

Nous présentons dans la suite de cette section la grammaire du langage AltaRica au format Backus-Naur. Il s'agit de la grammaire du langage AltaRica doté des extensions décrites dans la section 4.6 page 38.

Le but de cette grammaire est de documenter de manière précise, mais aussi pratique, la syntaxe du langage AltaRica. Elle doit pouvoir être utilisée comme référence par un utilisateur du langage, et doit être respectée par les outils souhaitant lire des fichiers au format AltaRica.

> Une version anglaise de ce texte apparaîtra prochainement, et c'est elle qui tiendra lieu de norme officielle pour le langage.

A.1 Conventions

Les règles de production de la grammaire sont données sous la forme suivante :

non-terminal :
    $règle_1$
    $règle_2$
    ...

et chaque règle peut contenir soit des non-terminaux, soit des `mots-clés`, soit des expressions régulières (utilisées seulement pour définir les identificateurs et les entiers).

De plus, afin de rendre la grammaire plus lisible, nous avons utilisé la notation mot_{opt} pour signifier qu'un mot est optionnel.

A.2 Eléments lexicaux de base

Un fichier AltaRica est constitué d'une succession d'éléments lexicaux, pouvant être séparés par des caractères d'espacement.

A.2.1 Commentaires

Il est possible d'ajouter un commentaire partout où un caractère d'espacement peut apparaître.

Les commentaires peuvent prendre deux formes :

- Si le couple de caractères `//` apparaît, alors ces deux caractères ainsi que toute la suite de la ligne dans laquelle ils apparaissent sont ignorés
- Si le couple de caractères `/*` apparaît, alors ces deux caractères ainsi que tous les caractères qui suivent jusqu'au prochain couple `*/` (compris) sont ignorés

> Ceci dévie de la norme AltaRica DataFlow, qui autorise explicitement les commentaires entre `/*` et `*/` à être imbriqués.

A.2.2 Mots-clés

Certaines suites de caractères sont réservées pour constituer les mots-clés du langage AltaRica. Ces mots-clés ne peuvent servir d'identificateur.

Chaque lettre d'un mot-clé du langage AltaRica peut être écrite indifféremment en majuscule ou en minuscule. Ainsi, les mots-clés `node`, `Node`, et `NoDe` sont équivalents.

Les mots-clés du langage sont les suivants :
`and`, `assert`, `bool`, `case`, `const`, `domain`, `else`, `event`, `extern`, `false`, `flow`, `if`, `imply`, `init`, `min`, `max`, `node`, `not`, `or`, `state`, `sub`, `sync`, `then`, `trans`, `true`

A.2.3 Constantes pré-définies

Une suite non vide de chiffres est interprétée comme un nombre entier (en base 10), sauf si elle apparaît dans un identificateur. Un tel élément lexical a donc le type *entier*.

Les deux mots-clés `true` et `false` sont respectivement interprétés comme des *booléens* ayant pour valeur vrai et faux respectivement.

A.2.4 Identificateurs

Un identificateur est une séquence non-vide de caractères alphanumériques ou souligné (`_`) qui ne commence pas par un chiffre et qui n'est pas un mot-clé.

Il est aussi possible d'utiliser des identificateurs plus compliqués en les entourant avec des apostrophes (`'`). De tels identificateurs ne doivent pas contenir d'apostrophe ni de point (`.`) et ne peuvent pas être le mot vide (`''`).

Les identificateurs peuvent représenter plusieurs objets différents, selon le contexte où ils apparaissent :

- noms de domaines
- noms de constantes
- noms de variables de nœuds
- noms d'événements
- noms de type de nœud AltaRica
- noms d'instance de nœud AltaRica
- attributs

A.3 Structure d'un fichier AltaRica

Un fichier AltaRica contient une suite de définitions d'objets. Chaque définition peut être suivie d'un point-virgule.

fichier-altarica :
définition ;$_{opt}$ *fichier-altarica*
ϵ

Il est possible de définir des domaines nommés, des constantes, et des nœuds AltaRica.

définition :
définition-de-domaine
définition-de-constante
définition-de-nœud

Une constante nommée se définit simplement comme suit :

définition-de-constante :
`const` *identificateur* `=` *expression*
La constante aura le type de l'expression.

A.4 Les types et les domaines

Les variables qui apparaissent dans une spécification AltaRica sont chacune associée à un domaine, qui décrit les valeurs qu'elle peut prendre. Chaque domaine est inclus dans un type. C'est le type qui détermine les expressions dans lesquelles une variable peut apparaître.

Il y a trois types distincts en AltaRica :
- les booléens
- les entiers
- les constantes énumérées

Les domaines utilisables dans un fichier AltaRica sont :
- les booléens ; les constantes booléennes sont `true` et `false`. Elles sont de type booléen
- les intervalles ; chaque expression délimitant l'intervalle doit s'évaluer en une constante entière, à l'endroit où l'expression apparaît dans le fichier. Les constantes qui forment un intervalle sont les entiers allant de la valeur de la plus petite des deux expressions à la valeur de la plus grande, incluses.
- les énumérations partagées ; la liste d'identificateurs donne exactement les éléments qui peuvent être affectés à une variable ayant ce domaine. Si un identificateur apparaît dans deux énumérations partagées différentes, il s'agit de la même constante. La réunion de toutes les énumérations partagées constitue le type des constantes énumérées

spécification-de-domaine :
`bool`
`[` *expression* `,` *expression* `]`
`{` *liste-d-identificateurs* `}`
identificateur

Il est possible de définir un nouveau nom de domaine, et de l'utiliser partout où il serait possible de faire figurer sa définition.

définition-de-domaine :
`domain` *identificateur* `=` *spécification-de-domaine*

A.5 Les expressions

Les expressions sont utilisées à de nombreux endroits dans un fichier AltaRica.

expr-atomique :
`(` *expression* `)`
chemin
entier
`true`
`false`

Une expression atomique peut être :
- un chemin, auquel cas elle s'évalue en la valeur de la variable dans le contexte où elle apparaît
- un entier, auquel cas elle s'évalue en la constante entière correspondante
- `true`, auquel cas elle s'évalue en la constant booléenne *vrai*
- `false`, auquel cas elle s'évalue en la constant booléenne *faux*

Les négations arithmétique et logique s'expriment de manière similaire, par une expression unaire commençant respectivement par `-` ou `~`. Chacun de ces opérateurs ne s'applique respectivement qu'à une expression entière, resp. booléenne. Il forme lui-même une expression du même type que son argument.

expr-unaire :
`-` *expr-unaire*
`~` *expr-unaire*
expr-atomique

Les opérateurs arithmétiques classiques sont disponibles et ne sont appliquables qu'aux expressions entières. Ils forment eux-mêmes des expressions entières.

expr-additive :
expr-additive `+` *expr-multiplicative*
expr-additive `-` *expr-multiplicative*
expr-multiplicative

expr-multiplicative :
expr-multiplicative `*` *expr-unaire*
expr-multiplicative `/` *expr-unaire*
expr-unaire

Les opérateurs de comparaison arithmétique s'appliquent à des expressions entières et forment des expressions booléennes.

expr-comparaison-arithmétique :
expr-comparaison-arithmétique `<` *expr-additive*
expr-comparaison-arithmétique `<=` *expr-additive*
expr-comparaison-arithmétique `>` *expr-additive*
expr-comparaison-arithmétique `>=` *expr-additive*

Les opérateurs logiques d'équivalence, de non-équivalence et d'implication s'appliquent à des expressions booléennes et forment des expressions booléennes.

Notez que les opérateurs d'équivalence et de non-équivalence sont surchargés : ils permettent aussi de comparer des expressions entières entre elles, ou des expressions (variable et/ou constante) énumérées entre elles.

expr-comparaison-logique :
expr-comparaison-logique `=` *expr-comparaison-arithmétique*
expr-comparaison-logique `!=` *expr-comparaison-arithmétique*
expr-comparaison-logique `=>` *expr-comparaison-arithmétique*
expr-comparaison-arithmétique

La disjonction et la conjonction sont également possibles entre expressions booléennes, et elles forment aussi une expression booléenne.

expr-disjonction :
expr-disjonction `|` *expr-conjonction*
expr-conjonction

expr-conjonction :
expr-conjonction `&` *expr-comparaison-logique*
expr-comparaison-logique

Une expression conditionnelle `if`/`then`/`else` prend la valeur d'une de deux expressions en fonction de la valeur d'une troisième expression booléenne. Chaque fois que l'expression booléenne est vraie, l'expression conditionnelle prend la valeur de l'expression qui suit le mot-clé `then`, sinon elle prend la valeur de l'expression qui suit le mot-clé `else`. Les deux expressions doivent être du même type.

expr-conditionnelle :
`if` *expression* `then` *expression* `else` *expression*
expr-case
expr-disjonction

Il est possible d'exprimer une cascade d'expressions `if`/`then`/`else` avec l'opérateur `case`. Il s'agit d'un raccourci syntaxique, et l'ordre dans l'énumération des différents cas est donc important pour les cas dont les expressions conditionnelles se chevauchent.

L'expression `case {` e_1 `:` v_1`,` e_2 `:` v_2`,` ... `else` v_n `}` est équivalente à l'expression suivante :

(if e_1 then v_1
else if e_2 then v_2
...
else v_n)

expr-case :
case { *expr-liste-de-cas* }

expr-liste-de-cas :
expr-un-cas , *expr-liste-de-cas*
else *expression*

expr-un-cas :
expression : *expression*

Enfin, toute expression peut être une expression conditionnelle.

expression :
expr-conditionnelle

A.6 Définition d'un nœud AltaRica

La définition d'un nœud AltaRica produit en fait deux objets portant le même nom : un type de nœud AltaRica, et une instance de nœud AltaRica. La description de leur contenu se fait par une succession de sections appelées champs du nœud.

Les champs d'un nœud peuvent être délimités par un point-virgule (;).

définition-de-nœud :
node *identificateur liste-de-champs-de-nœud* edon

liste-de-champs-de-nœud :
champ-de-nœud $;_{opt}$ *liste-de-champs-de-nœud*
directives-externes$_{opt}$

champ-de-nœud :
définition-de-variables
déclaration-d-événements
définition-de-transitions
assertions
définition-de-sous-nœuds
définition-de-vecteurs-de-diffusion
définition-d-état-initial

A.7 Les champs `state` et `flow`

Les deux champs `state` et `flow` permettent de définir les variables (respectivement d'état et de flux) appartenant à un nœud AltaRica. Ces variables sont nécessairement associées à un domaine, et peuvent avoir des attributs (cf. section A.15 page 109).

Le domaine et les attributs spécifiés s'appliquent à la liste d'identificateurs qui les précèdent.

définition-de-variables :
`state` *liste-de-variables-définies*
`flow` *liste-de-variables-définies*

liste-de-variables-définies :
variables-définies ; *liste-de-variables-définies*
variables-définies

variables-définies :
liste-d-identificateurs : *spécification-de-domaine* $attributs_{opt}$

A.8 Le champ `event`

Les champs `event` permettent de définir l'ordre partiel sur les événements du nœud, ainsi que de leur attacher des attributs.

Un événement peut apparaître plusieurs fois ; si plusieurs listes d'attributs lui sont associées, il aura pour attributs la réunion (ensembliste) de ces listes d'attributs. Certains attributs sont utilisés pour indiquer la visibilité d'un événement (section A.16 page 109).

Une comparaison d'événements est une séquence d'atomes de comparaisons, séparés par des opérateurs de comparaison. L'ordre induit par l'opérateur s'applique aux événements qui apparaissent dans les deux atomes qui lui sont adjacents.

déclaration-d-événements :
`event` *liste-d-événements-attr*

liste-d-événements-attr :
liste-de-comparaisons-d-événements $attributs_{opt}$; *liste-d-événements-attr*
liste-de-comparaisons-d-événements

liste-de-comparaisons-d-événements :
comparaison-d-événements , *liste-de-comparaisons-d-événements*
comparaison-d-événements

comparaison-d-événements :
comparaison-d-événements < *atome-de-comparaison-d-événements*
comparaison-d-événements > *atome-de-comparaison-d-événements*
atome-de-comparaison-d-événements

atome-de-comparaison-d-événements :
`{` *liste-de-comparaisons-d-événements* `}`
identificateur

A.9 Le champ `trans`

Les transitions d'un nœud sont specifiées dans les champs `trans`. Chaque transition est *gardée* par une expression booléenne qui précise les configurations dans lesquelles la transition peut être franchie.

Plusieurs transitions peuvent partager les mêmes gardes, auquel cas on peut omettre de répéter la garde. Plusieurs transitions peuvent partager les mêmes gardes et les mêmes affectations ; il suffit alors de donner les événements qui les différencient par une liste d'identificateurs d'événements.

Seules les variables d'état peuvent être affectées, et chacune doit l'être au plus une fois. Une variable qui n'est pas affectée conserve la valeur qu'elle avait dans l'état source de la transition.

définition-de-transitions :
`trans` *liste-de-transitions*

liste-de-transitions :
transition `;` *liste-de-transitions*
transition `;`$_{opt}$

transition :
expression liste-de-successeurs

liste-de-successeurs :
successeur liste-de-successeurs
successeur

successeur :
`|-` *liste-d-identificateurs* `->` *liste-d-affectations*$_{opt}$

liste-d-affectations :
affectation `,` *liste-d-affectations*
affectation

affectation :
identificateur `:=` *expression*

A.10 Le champ `assert`

Les variables sont contraintes par des assertions, qui sont des expressions booléennes. Toutes les configurations d'un nœud doivent satisfaire toutes les assertions.

assertions :
`assert` *liste-d-assertions*

liste-d-assertions :
expression ; *liste-d-assertions*
expression

A.11 Le champ `sub`

Le champ `sub` permet d'incorporer dans un nœud d'autres sous-nœuds. Chaque définition permet de définir plusieurs instances d'un même type de nœud.

Les variables et les événements des sous-nœuds sont accessibles par une notation pointée commençant par le nom de l'instance. Cet accès est sujet aux règles de visibilité décrites section A.16 page 109.

définition-de-sous-nœuds :
`sub` *liste-de-sous-nœuds*

liste-de-sous-nœuds :
sous-nœuds ; *liste-de-sous-nœuds*
sous-nœuds

sous-nœuds :
liste-d-identificateurs : *identificateur*

A.12 Le champ `sync`

La synchronisation d'événements est possible entre événements d'un nœud et de ses sous-nœuds. Un vecteur de diffusion contient exactement un événement par sous-nœud et un événement du nœud local. Si aucun événement n'apparaît pour un de ces nœuds, c'est l'événement implicite ε de ce nœud qui est synchronisé.

définition-de-vecteurs-de-diffusion :
`sync` *liste-de-vecteurs-de-diffusion*

liste-de-vecteurs-de-diffusion :
vecteur-de-diffusion ; *liste-de-vecteurs-de-diffusion*
vecteur-de-diffusion

vecteur-de-diffusion :
< *liste-de-diffusions* > $contrainte\text{-}de\text{-}diffusion_{opt}$ $politique\text{-}de\text{-}diffusion_{opt}$

liste-de-diffusions :
diffusion , *liste-de-diffusions*
diffusion

Chaque événement peut de plus être *diffusé*, en lui adjoignant un point d'interrogation. Si c'est le cas, deux vecteurs de synchronisation peuvent être générés, l'un avec l'événement, et l'autre avec l'événement ε du nœud auquel il appartient.

diffusion :
chemin `?`$_{opt}$

La contrainte de diffusion précise une contrainte sur le nombre d'événements diffusés apparaissant dans chaque vecteur de synchronisation dérivé du vecteur de diffusion.

contrainte-de-diffusion :
`<` *entier*
`<=` *entier*
`>` *entier*
`>=` *entier*

Les vecteurs de synchronisation générés à partir d'un vecteur de diffusion sont ensuite ordonnés en fonction du nombre d'événements diffusés apparaissant dedans. Le mot-clé `max` précise un ordre croissant (les vecteurs qui synchronisent plus d'événements diffusés sont plus prioritaires), alors que le mot-clé `min` demande un ordre décroissant. La politique par défaut en l'absence de mot-clé est l'ordre croissant (correspondant au mot-clé optionnel `max`).

politique-de-diffusion :
`max`
`min`

A.13 Le champ `init`

Il est possible de préciser l'état initial du système par une liste d'affectations sur les variables d'états, y compris celles de tous ses sous-nœuds. Cette information peut être ignorée par les outils pour lesquels elle n'a pas de sens.

Si la valeur initiale d'une variable d'une instance de sous-nœud est définie alors que la valeur initiale de cette variable était déjà définie dans le type de ce sous-nœud, c'est l'affectation du nœud parent qui prévaut : il est possible de redéfinir la valeur initiale d'une variable.

> Les affectations peuvent être indifféremment séparées par des points-virgules ou des virgules. Les points-virgules sont autorisés par analogie avec la forme de tous les autres champs de nœud, et les virgules sont autorisées par compatibilité ascendante ainsi que par analogie avec les listes d'affectations qui apparaissent dans les successeurs de transitions.

définition-d-état-initial :
`init` *liste-d-affectations-de-chemins*

liste-d-affectations-de-chemins :
affectation-de-chemin `;` *liste-d-affectations-de-chemins*
affectation-de-chemin `,` *liste-d-affectations-de-chemins*
affectation-de-chemin

affectation-de-chemin :
chemin `:=` *expression*

A.14 Le champ `extern`

Les outils peuvent choisir d'ajouter des informations qui leur sont propres dans le champ `extern`.

Les parties de ce champ qu'un outil ne sait pas décoder doivent être ignorées sans erreur. Cependant, un outil peut choisir d'afficher un message d'avertissement afin de prévenir l'utilisateur qu'une partie des informations contenues dans le fichier a été ignorée.

Le champ `extern` est le seul champ de nœud qui ne peut être suivi d'aucun autre champ. Il se finit en même temps que le nœud, sur la lecture du mot-clé `edon`.

directives-externes :
`extern` ...

A.15 Les attributs

Les attributs permettent d'associer une liste d'identificateurs à une variable ou à un événement. Leur utilisation est laissée à la discrétion des outils, à l'exception des attributs `private`, `parent`, et `public` qui codent la visibilité d'une variable ou d'un événement. De ce fait, les attributs de visibilité sont mutuellement exclusifs. Les notions de visibilité sont décrites dans la section A.16, page 109.

attributs :
`:` *liste-d-identificateurs*

liste-d-identificateurs :
identificateur `,` *liste-d-identificateurs*
identificateur

A.16 Règles de visibilité des identificateurs

A.16.1 Constantes et noms de domaines

Les constantes et noms de domaines sont utilisables à partir de l'endroit où ils sont déclarés dans un fichier.

Pour les identificateurs qui apparaissent dans un nœud, l'accès aux identificateurs des sous-nœuds se fait en utilisant une notation pointée qui contient une séquence de noms d'instances de nœuds, suivie de l'identificateur à accéder.

chemin :
identificateur `.` *chemin*
identificateur

A.16.2 Variables d'état et de flux

La visibilité d'une variable d'état ou de flux est conditionnée par un attribut de visibilité.

L'attribut `private` restreint la visibilité de la variable au nœud courant. L'attribut `parent` restreint la visibilité de la variable au nœud courant et à son parent lorsqu'il apparaît comme sous-nœud. Enfin, l'attribut `public` rend la variable visible dans tous les nœuds qui se trouvent au-dessus de lui dans la hiérarchie.

Par défaut, une variable d'état est `private`, et une variable de flux est `parent`.

A.16.3 Evénements

La visibilité d'un événement est aussi conditionnée par un attribut de visibilité de manière similaire aux variables, sauf pour l'attribut `public` qui diffère légèrement.

L'attribut `private` restreint la visibilité de l'événement au nœud courant. L'attribut `parent` restreint la visibilité de l'événement au nœud courant et à son parent lorsqu'il apparaît comme sous-nœud.

L'attribut `public` rend l'événement disponible dans tous les nœuds qui se trouvent au-dessus de lui dans la hiérarchie, *jusqu'à ce qu'il apparaisse dans un vecteur de synchronisation.* Une fois que l'événement a été synchronisé, aucun nœud parent ne peut plus le re-synchroniser.

Par défaut, un événement est `parent`.

Annexe B

Utilisation de Mec V

B.1 Premier contact

Au démarrage, Mec V affiche un petit message de bienvenue et attend les entrées de l'utilisateur.

```
Welcome to Mec 5.  Type :help to get a list of commands.

[mec]
```

Lorsque l'invite de Mec est affichée, l'utilisateur peut soit entrer une commande Mec V, soit taper une formule du langage de spécification.

Toute ligne tapée par l'utilisateur qui commence par le caractère deux-points (:) est une commande destinée à Mec.

Une formule du langage de spécification peut s'étendre sur plusieurs lignes. Mec V prévient dans ce cas-là qu'il attend la fin de la formule en affichant un point comme invite au lieu de l'invite habituelle. Toute définition d'un objet entraîne l'affichage de son type. Par exemple :

```
[mec] R(x) :=
.         x = true;
        R: (bool) -> bool
[mec] cst := false;
        cst: bool
[mec]
```

Si Mec V est compilé avec une bibliothèque respectant l'interface *readline*, il est possible d'éditer la ligne en cours et de remonter dans l'historique des lignes tapées précédemment. Dans ce cas, vous pouvez vous référer à la documentation de la bibliothèque readline pour plus d'informations.

B.2 Calcul de relations

Le principal but de Mec V est de calculer des relations sur des domaines finis. Nous avons essayé partout où cela était possible de nous inspirer du langage AltaRica pour ce qui concerne la syntaxe, afin de présenter un tout cohérent à l'utilisateur. Cependant plusieurs exceptions sont notables :

Les identificateurs

Afin de permettre l'utilisation de variables primées, le caractère apostrophe (') peut maintenant apparaître dans un identificateur. Si on souhaite faire référence à des identificateurs quelconques, il faut utiliser les guillemets ("). De plus, ces identificateurs ne peuvent contenir ni point (.) ni guillemet.

identificateur :
`[a-zA-Z_][0-9a-zA-Z'_]*`
`"[^".]*"`

Les opérateurs

Les opérateurs logiques ne sont plus disponibles sous leur forme de mot-clé en Anglais, cela afin de limiter la pollution de l'espace de nom des variables au maximum. Par exemple, la formule `x or y` est valide dans un modèle AltaRica mais doit s'écrire par exemple `x | y` dans une spécification Mec.

B.2.1 Les domaines de base

Les domaines de base disponibles comprennent les domaines de base AltaRica (p. 101).

spécification-de-domaine :
`bool`
`[` *expression* `,` *expression* `]`
`{` *liste-d-identificateurs* `}`
identificateur
identificateur-de-nœud

Les domaines suivants sont ajoutés :

identificateur-de-nœud :
identificateur `!` *identificateur*

Pour chaque nœud AltaRica `n` chargé, deux domaines `n!c` et `n!ev` sont définis, qui représentent respectivement le type *configuration du nœud* `n` et le type *vecteur d'événements du nœud* `n`.

B.2.2 Les expressions

La syntaxe des expressions AltaRica a été respectée au maximum, à l'exception des deux remarques en début de section. Elle a de plus été étendue afin de permettre l'utilisation de prédicats et d'opérateurs du premier ordre. Nous ne décrivons ici que les différences par rapport aux expressions AltaRica, et nous référons le lecteur à la page 102 où sont décrites les expressions AltaRica.

Il est désormais possible d'utiliser un prédicat, ou un quantificateur du premier ordre à la place d'une expression unaire.

expr-unaire :

- *expr-unaire*
~ *expr-unaire*
expr-prédicat

La syntaxe d'un quantificateur du premier ordre utilise les chevrons et les crochets pour symboliser respectivement l'existentiel et l'universel, comme il est d'usage dans les logiques modales.

Tout symbole introduit dans un quantificateur peut être typé explicitement. Par exemple `<x : D> P(x)` se lit $\exists x.(x \in \mathcal{D} \wedge P(x))$.

L'utilisation d'un prédicat passe par une notation parenthésée d'appel de fonction classique dans beaucoup de langages.

expr-prédicat :
`<` *identificateur* `>` *expr-unaire*
`<` *identificateur-typé* `>` *expr-unaire*
`[` *identificateur* `]` *expr-unaire*
`[` *identificateur-typé* `]` *expr-unaire*
identificateur `(` *liste-d-expressions* `)`
identificateur-de-nœud `(` *liste-d-expressions* `)`
expr-atomique

liste-d-expressions :
expression `,` *liste-d-expressions*
expression

identificateur-typé :
identificateur `:` *spécification-de-domaine*

B.2.3 Définition de nouveaux objets

Hormis le chargement d'un fichier qui peut induire la création de nouveaux objets (des nœuds AltaRica par exemple), une définition d'objet peut prendre l'une de ces formes :

définition :
`domain` *identificateur* `=` *spécification-de-domaine* ;
identificateur `:=` *expression* ;
équation ;
`begin` *système-d-équations* `end`

Définition de type

La définition d'un nouveau type se fait avec la même syntaxe qu'en AltaRica, à ceci près que le point-virgule final est obligatoire.

```
[mec] domain etats = { vide, plein };
```

Définition de constante

La définition d'une nouvelle constante s'écrit comme suit :
identificateur `:=` *expression* `;`

```
[mec] cst := 25;
        cst: integer
```

Définition d'une relation

Il est possible de définir une relation en donnant l'expression qui contraint ses paramètres.

```
[mec] R(x : [ 0, 10 ]) := x = 2 | x = 4;
        R: ([ 0, 10 ]) -> bool
[mec] :display R
(4)
(2)
```

Il est aussi possible de donner une définition par (plus petit ou plus grand) point fixe.

équation :
identificateur `(` *liste-de-paramètres* `)` *opérateur-de-point-fixe expression*

opérateur-de-point-fixe :
`+=`
`+`*entier*`=`
`-=`
`-`*entier*`=`
`:=`

liste-de-paramètres :
identificateur
identificateur-typé
identificateur `,` *liste-de-paramètres*
identificateur-typé `,` *liste-de-paramètres*

```
[mec] S(x : [ 0, 10 ]) += x = 4 | S(x + 1);
        S: ([ 0, 10 ]) -> bool
[mec] :display S
(4)
(3)
(2)
(1)
(0)
```

Il est aussi possible de définir un système d'équations, afin de pouvoir décrire des points fixes imbriqués.

On peut citer un exemple repris de l'outil Toupie :

```
[mec] domain vertex = [1, 9];
[mec] g(S : vertex, T : vertex) += (T = S + 1) |
.                 (T = S & (S = 2 | S = 4 | S = 6 | S = 8)) |
.                 (T = 1 & S = 3);
        g: ([ 1, 9 ], [ 1, 9 ]) -> bool
[mec] odd(S : vertex) += <n : [0, 4]>(S = n + n + 1);
        odd: ([ 1, 9 ]) -> bool
[mec] begin
.         tau(U : vertex) -1= aux(U);
.  local  aux(V : vertex) +2= <W : vertex>(g(V, W) & aux(W)) |
.                            <W : vertex>(g(V, W) & odd(W) & tau(W));
. end
        tau: ([ 1, 9 ]) -> bool
[mec] :display tau
(3)
(2)
(1)
[mec]
```

B.3 Les commandes

La liste des commandes disponibles peut être obtenue par la commande `:help`. Cette liste varie d'une version à l'autre de Mec et dépend aussi des options de compilation utilisées. Certaines commandes peuvent donc ne pas être disponibles dans votre version de Mec.

B.3.1 Les commandes générales

:help

permet d'afficher la liste des commandes disponibles.

On peut aussi lui passer en paramètre le nom d'une autre commande. Dans ce cas, un synopsis rudimentaire de la commande sera affiché.

```
[mec] :help display
display string
        displays the given object
```

:quit

quitte Mec.

:display

affiche une constante préalablement définie

```
[mec] cst := false;
        cst: bool
[mec] :display cst
false
```

:rel-cardinal

affiche le nombre d'éléments appartenant à la relation passée en paramètre.

```
[mec] R(x:[0,4], y:[0,4]) := x + y = 4;
        R: ([ 0, 4 ], [ 0, 4 ]) -> bool
[mec] :rel-cardinal R
cardinal of R: 5
[mec] :display R
(0, 4)
(4, 0)
(2, 2)
(1, 3)
(3, 1)
```

:spec-load

charge le fichier donné en paramètre en tant que fichier de spécification. Comme ce qui est saisi interactivement, ce fichier peut contenir des commandes Mec ou des formules de spécification.

```
[mec] :spec-load try.spec
```

B.3.2 Commandes BDD

:rel-write

sauve le BDD qui code la relation donnée en paramètre sous la forme d'un graphe au format GraphViz.

```
[mec] :rel-write R R.dot
```

:rel-enumerate

énumère les mots binaires contenus dans le BDD codant la relation passée en paramètre. Cette commande n'est utile qu'à des fins de débuggage.

```
[mec] R(x:[0,4], y:[0,4]) := x + y = 4;
        R: ([ 0, 4 ], [ 0, 4 ]) -> bool
[mec] :rel-enumerate R
exhaustive:
00000010
00100000
01000100
10001100
11001000
```

B.3.3 Commandes AltaRica

:ar-load

charge un fichier AltaRica

```
[mec] :ar-load examples/Peterson+N2.alt
```

:ar-poset-write

sauvegarde au format GraphViz le graphe représentant l'ordre partiel sur les événements du nœud AltaRica donné en paramètre.

```
[mec] :ar-poset-write jeu priorites.dot
```

:ar-display

affiche quelques informations sur le contenu d'un nœud AltaRica. La sortie n'est actuellement que très partielle et ne permet pas de sauvegarder un nœud AltaRica. Cette commande est utile à des fins de débuggage : certains BDDs calculés lors de la mise à plat du nœud sont sauvegardés au format GraphViz.

```
[mec] :ar-display jeu
node jeu
local variables
        state s : [ 1, 61 ];
        state d : [ 1, 61 ];
        flow f : [ 1, 61 ];
assertion:
rel size: 246
saved in file bdd00000.dot
relation for event  has size 861
relation for event c has size 3269
edon
```

Ici, par exemple, le BDD de l'assertion contient 246 nœuds et est sauvegardé dans le fichier bdd00000.dot

Les tailles des BDDs représentant les relations de transition des événements ε et c sont respectivement 861 nœuds et 3269 nœuds.

B.4 Compiler Mec V

La compilation de Mec V est normalement simplifiée par l'utilisation d'un script. Voici la marche à suivre pour compiler Mec V à partir de ses sources. Nous symbolisons l'invite du shell par le symbole `%`. Le site `http://altarica.labri.fr/` permet de signaler les bugs et de récupérer les nouvelles versions de Mec V.

La première chose à faire est de désarchiver les sources en utilisant une commande de cette forme :

```
% gzip -cd mec.tar.gz | tar xf -
```

Ceci crée un répertoire `mec`. Le script de compilation doit être lancé depuis le répertoire `mec/build` :

```
% cd mec/build
% ./do-build
```

Enfin, une fois le processus de compilation terminé, l'exécutable `mec` se trouve dans un sous-répertoire du répertoire `mec/build` dont le nom dépend de la machine sur laquelle vous avez compilé Mec V. Le programme peut normalement être copié où bon vous semble.

Voici un exemple de session de compilation :

```
% gzip -cd mec.tar.gz | tar xf -
% cd mec/build
% ./do-build
Checking where distribution is... found in ../..
Checking whether make is GNU make... no
Checking for X11... not used yet
Checking for flex... /usr/bin/flex
Checking for yacc... /usr/bin/yacc
Checking for cc... /usr/bin/cc
Checking for <readline/readline.h>... found.
Checking for readline with -ledit -ltermcap... found.
Generating Makefile rules... done
Configuration done.
yacc -d -p spec_yy -b spec-parser ../../io/spec/syntax.yacc
[...]
% ls
do-build     netbsd-i386
% cd netbsd-i386
% ./mec

Welcome to Mec 5.  Type :help to get a list of commands.

[mec]
```

Annexe C

Grammaire AltaRica

fichier-altarica :
définition ;$_{opt}$ *fichier-altarica*
ϵ

définition :
définition-de-domaine
définition-de-constante
définition-de-nœud

— *définition-de-domaine*

définition-de-domaine :
`domain` *identificateur* `=` *spécification-de-domaine*

spécification-de-domaine :
`bool`
`[` *expression* `,` *expression* `]`
`{` *liste-d-identificateurs* `}`
identificateur

— *définition-de-constante*

définition-de-constante :
`const` *identificateur* `=` *expression*

— *définition-de-nœud*

définition-de-nœud :
`node` *identificateur liste-de-champs-de-nœud* `edon`

liste-de-champs-de-nœud :
champ-de-nœud ;$_{opt}$ *liste-de-champs-de-nœud*
directives-externes$_{opt}$

champ-de-nœud :
définition-de-variables
déclaration-d-événements
définition-de-transitions
assertions
définition-de-sous-nœuds
définition-de-vecteurs-de-diffusion
définition-d-état-initial

— *définition-de-variables*

définition-de-variables :
`state` *liste-de-variables-définies*
`flow` *liste-de-variables-définies*

liste-de-variables-définies :
variables-définies ; *liste-de-variables-définies*
variables-définies

variables-définies :
liste-d-identificateurs : *spécification-de-domaine attributs*$_{opt}$

— *déclaration-d-événements*

déclaration-d-événements :
`event` *liste-d-événements-attr*

liste-d-événements-attr :
liste-de-comparaisons-d-événements attributs$_{opt}$; *liste-d-événements-attr*
liste-de-comparaisons-d-événements

liste-de-comparaisons-d-événements :
comparaison-d-événements , *liste-de-comparaisons-d-événements*
comparaison-d-événements

comparaison-d-événements :
comparaison-d-événements < *atome-de-comparaison-d-événements*
comparaison-d-événements > *atome-de-comparaison-d-événements*
atome-de-comparaison-d-événements

atome-de-comparaison-d-événements :
`{` *liste-de-comparaisons-d-événements* `}`
identificateur

— *définition-de-transitions*

définition-de-transitions :
`trans` *liste-de-transitions*

liste-de-transitions :
transition `;` *liste-de-transitions*
transition `;`$_{opt}$

transition :
expression liste-de-successeurs

liste-de-successeurs :
successeur liste-de-successeurs
successeur

successeur :
`|-` *liste-d-identificateurs* `->` *liste-d-affectations*$_{opt}$

liste-d-affectations :
affectation `,` *liste-d-affectations*
affectation

affectation :
identificateur `:=` *expression*

— *assertions*

assertions :
`assert` *liste-d-assertions*

liste-d-assertions :
expression `;` *liste-d-assertions*
expression

— *définition-de-sous-nœuds*

définition-de-sous-nœuds :
`sub` *liste-de-sous-nœuds*

liste-de-sous-nœuds :
sous-nœuds ; *liste-de-sous-nœuds*
sous-nœuds

sous-nœuds :
liste-d-identificateurs : *identificateur*

— *définition-de-vecteurs-de-diffusion*

définition-de-vecteurs-de-diffusion :
`sync` *liste-de-vecteurs-de-diffusion*

liste-de-vecteurs-de-diffusion :
vecteur-de-diffusion ; *liste-de-vecteurs-de-diffusion*
vecteur-de-diffusion

vecteur-de-diffusion :
`<` *liste-de-diffusions* `>` *contrainte-de-diffusion*$_{opt}$ *politique-de-diffusion*$_{opt}$

liste-de-diffusions :
diffusion , *liste-de-diffusions*
diffusion

diffusion :
chemin `?`$_{opt}$

contrainte-de-diffusion :
`<` *entier*
`<=` *entier*
`>` *entier*
`>=` *entier*

politique-de-diffusion :
`max`
`min`

— *définition-d-état-initial*

définition-d-état-initial :
`init` *liste-d-affectations-de-chemins*

liste-d-affectations-de-chemins :
affectation-de-chemin ; *liste-d-affectations-de-chemins*
affectation-de-chemin , *liste-d-affectations-de-chemins*

affectation-de-chemin

affectation-de-chemin :
chemin `:=` *expression*

— *directives-externes*

directives-externes :
`extern` ...

— *expression*

expression :
expr-conditionnelle

expr-conditionnelle :
`if` *expression* `then` *expression* `else` *expression*
expr-case
expr-disjonction

expr-case :
`case {` *expr-liste-de-cas* `}`

expr-liste-de-cas :
expr-un-cas `,` *expr-liste-de-cas*
`else` *expression*

expr-un-cas :
expression `:` *expression*

expr-disjonction :
expr-disjonction `|` *expr-conjonction*
expr-conjonction

expr-conjonction :
expr-conjonction `&` *expr-comparaison-logique*
expr-comparaison-logique

expr-comparaison-logique :
expr-comparaison-logique `=` *expr-comparaison-arithmétique*
expr-comparaison-logique `!=` *expr-comparaison-arithmétique*
expr-comparaison-logique `=>` *expr-comparaison-arithmétique*
expr-comparaison-arithmétique

expr-comparaison-arithmétique :
    expr-comparaison-arithmétique `<` *expr-additive*
    expr-comparaison-arithmétique `<=` *expr-additive*
    expr-comparaison-arithmétique `>` *expr-additive*
    expr-comparaison-arithmétique `>=` *expr-additive*

expr-additive :
    expr-additive `+` *expr-multiplicative*
    expr-additive `-` *expr-multiplicative*
    expr-multiplicative

expr-multiplicative :
    expr-multiplicative `*` *expr-unaire*
    expr-multiplicative `/` *expr-unaire*
    expr-unaire

expr-unaire :
    `-` *expr-unaire*
    `~` *expr-unaire*
    expr-atomique

expr-atomique :
    `(` *expression* `)`
    chemin
    entier
    `true`
    `false`

— *attributs*

attributs :
    `:` *liste-d-identificateurs*

— *entier*

entier :
    `[0-9][0-9]*`

— *identificateur*

identificateur :
    `[a-zA-Z_][0-9a-zA-Z_]*`
    `'[^'.][^'.]*'`

chemin :
    identificateur . *chemin*
    identificateur

liste-d-identificateurs :
    identificateur , *liste-d-identificateurs*
    identificateur

Annexe D

Grammaire de Mec V

Ce chapitre présente la grammaire du langage de spécification de Mec V, et nous avons partout où cela était possible utilisé les mêmes noms de non-terminaux que dans la grammaire du langage AltaRica afin d'insister sur le lien fort que nous avons souhaité entre ces deux langages.

Cependant, lorsque des règles spécifiques au langage de spécification sont ajoutées à des non-terminaux déjà présents dans la grammaire AltaRica, nous les encadrons afin d'aider le lecteur à situer les différences entre les deux grammaires.

— *définition*

définition :
`domain` *identificateur* `=` *spécification-de-domaine* `;`
identificateur `:=` *expression* `;`
équation `;`
`begin` *système-d-équations* `end`

spécification-de-domaine :
`bool`
`[` *expression* `,` *expression* `]`
`{` *liste-d-identificateurs* `}`
identificateur
identificateur-de-nœud

— *équation*

équation :
identificateur `(` *liste-de-paramètres* `)` *opérateur-de-point-fixe expression*

opérateur-de-point-fixe :
`+=`
`+`*entier*`=`
`-=`

-entier=
`:=`

liste-de-paramètres :
identificateur
identificateur-typé
identificateur , *liste-de-paramètres*
identificateur-typé , *liste-de-paramètres*

— *système-d-équations*

système-d-équations :
local équation ; *système-d-équations*
équation ; *système-d-équations*
ϵ

— *expression*

expression :
expr-conditionnelle

expr-conditionnelle :
`if` *expression* `then` *expression* `else` *expression*
expr-case
expr-disjonction

expr-case :
`case {` *expr-liste-de-cas* `}`

expr-liste-de-cas :
expr-un-cas , *expr-liste-de-cas*
`else` *expression*

expr-un-cas :
expression : *expression*

expr-disjonction :
expr-disjonction `|` *expr-conjonction*
expr-conjonction

expr-conjonction :
expr-conjonction `&` *expr-comparaison-logique*
expr-comparaison-logique

expr-comparaison-logique :
expr-comparaison-logique = *expr-comparaison-arithmétique*
expr-comparaison-logique != *expr-comparaison-arithmétique*
expr-comparaison-logique => *expr-comparaison-arithmétique*
expr-comparaison-arithmétique

expr-comparaison-arithmétique :
expr-comparaison-arithmétique < *expr-additive*
expr-comparaison-arithmétique <= *expr-additive*
expr-comparaison-arithmétique > *expr-additive*
expr-comparaison-arithmétique >= *expr-additive*

expr-additive :
expr-additive + *expr-multiplicative*
expr-additive - *expr-multiplicative*
expr-multiplicative

expr-multiplicative :
expr-multiplicative * *expr-unaire*
expr-multiplicative / *expr-unaire*
expr-unaire

expr-unaire :
- *expr-unaire*
~ *expr-unaire*
expr-prédicat

expr-prédicat :
< *identificateur* > *expr-unaire*
< *identificateur-typé* > *expr-unaire*
[*identificateur*] *expr-unaire*
[*identificateur-typé*] *expr-unaire*
identificateur (*liste-d-expressions*)
identificateur-de-nœud (*liste-d-expressions*)
expr-atomique

expr-atomique :
(*expression*)
chemin
entier
true
false

liste-d-expressions :
expression , *liste-d-expressions*
expression

— *entier*

entier :
`[0-9][0-9]*`

— *identificateur*

identificateur :
`[a-zA-Z_][0-9a-zA-Z'_]*`
`"[^".]*"`

identificateur-de-nœud :
identificateur `!` *identificateur*

identificateur-typé :
identificateur `:` *spécification-de-domaine*

chemin :
identificateur `.` *chemin*
identificateur `.`
identificateur

liste-d-identificateurs :
identificateur `,` *liste-d-identificateurs*
identificateur

Table des figures

Index

Bibliographie

[1] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Improving the efficiency of circuit-to-BDD conversion by gate and input ordering. In *CD: International Conference on Computer Design: VLSI in Computers and Processors*, pages 64–69. IEEE Computer Society, September 2002.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[3] Henrik Andersen and Henrik Hulgaard. Boolean expression diagrams. *Information and Computation*, 179(2):194–212, December 2002.

[4] André Arnold. *Finite transition systems.* Prentice-Hall, 1994.

[5] André Arnold, Didier Bégay, and Paul Crubillé. *Construction and analysis of transition systems with MEC.* World Scientific, 1994.

[6] André Arnold, Alain Griffault, Gérald Point, and Antoine Rauzy. The altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40(2–3):109–124, 1999.

[7] André Arnold and Damian Niwiński. *Rudiments of μ-calculus.* Studies in Logic and the Foundations of Mathematics. North-Holland, 2001.

[8] André Arnold, Aymeric Vincent, and Igor Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 303(1):7–34, June 2003.

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

[10] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS: IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[11] Charles Castel and Christel Seguin. Modèles formels pour l'évaluation de la sûreté de fonctionnement des architectures logicielles d'avionique modulaire intégrée. In *AFADL: Approches Formelles dans l'Assistance au Développement de Logiciels*, June 2001.

[12] Laurence Cholvy, Frédéric Cuppens, and Robert Demolombe. Logiques modales et bases de données. *Revue Technique et Science Informatiques*, 17(3), March 1998.

[13] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *CAV: International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, July 2002.

[14] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *CAV: International Conference on Computer Aided*

Verification, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer, 1999.

[15] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *LP: Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, May 1981.

[16] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, January 2000.

[17] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[18] Marc-Michel Corsini and Antoine Rauzy. First experiments with toupie. Research Report 581-93, LaBRI, 1993.

[19] Marc-Michel Corsini and Antoine Rauzy. Toupie user's manual. Research Report 586-93, LaBRI, 1993.

[20] Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, April 1994.

[21] E. Allen Emerson and Charanjit S. Jutla. Tree automata, μ-calculus and determinacy. In *FOCS: IEEE Symposium on Foundations of Computer Science*, pages 368–377. IEEE Computer Society Press, 1991.

[22] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model-checking for the μ-calculus and its fragments. *Theoretical Computer Science*, 258:491–522, 2001.

[23] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating BDDs for symbolic model checking in CCS. *Journal of Distributed Computing*, 6(3):155–164, 1993.

[24] Laurent Fribourg and Marcos Veloso Peixoto. Automates concurrents à contraintes. *Revue Technique et Science Informatiques*, 13(6):837–866, 1994.

[25] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *CAD: International Conference on Computer Aided Design*, pages 2–5. IEEE Computer Society Press, November 1988.

[26] James Garson. Modal logic. *The Stanford Encyclopedia of Philosophy*, 2003. Forthcoming URL: http://plato.stanford.edu/archives/win2003/entries/logic-modal/.

[27] Alain Griffault. Conception et validation d'un protocole avec le modèle AltaRica. In *AFADL: Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 293–307, January 2003.

[28] Alain Griffault and Aymeric Vincent. Vérification de modèles AltaRica, October 2003. MAJECSTIC: Manifestation des jeunes chercheurs STIC.

[29] Thomas A. Henzinger. Masaccio: A formal model for embedded components. In *IFIP: International Conference on Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 549–563. Springer-Verlag, August 2000.

[30] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV: International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer-Verlag, 1998.

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

[32] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. John Wiley & sons, 1996.

[33] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[34] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16, 1963.

[35] François Laroussinie. *Logique temporelle avec passé pour la spécification et la vérification des systèmes réactifs*. PhD thesis, I.N.P. de Grenoble, November 1994.

[36] Jørn Lind-Nielsen. BuDDy: Binary decision diagram package v2.2, November 2002.

[37] Angelika Mader. *Verification of modal properties using boolean equation systems*. PhD thesis, Fakultät Informatik, Technische Universität München, 1997.

[38] Kenneth L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, May 1992.

[39] Robin Milner. *A calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[40] Maria Nikolskaia. *Binary Decision Diagrams and applications to reliability analysis*. PhD thesis, LaBRI, Université Bordeaux 1, Janvier 2000.

[41] Claire Pagetti. Une extension temporisée d'AltaRica. In *MSR: Congrès Modélisation des Systèmes Réactifs*, pages 327–342. Hermès Science, September 2003.

[42] Claire Pagetti. *AltaRica temporisé*. PhD thesis, IRCCyN, Ecole Centrale de Nantes, 2004. Ongoing.

[43] David Park. Finiteness is μ-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.

[44] Amir Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[45] Gérald Point. *Altarica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. PhD thesis, LaBRI, Université Bordeaux 1, January 2000.

[46] Gérald Point. Altatools, May 2003. `http://altarica.labri.fr/Tools/Altatools/`.

[47] Jon Postel. Transmission Control Protocol. RFC 793, 1981.

[48] Peter Ramadge and Walter Murray Wonham. The control of discrete event systems. In *Proceedings of the IEEE*, volume 77, pages 79–98, January 1989.

[49] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels, Techniques et outils du model-checking*. Vuibert, April 1999.

[50] Aymeric Vincent. Synthèse de contrôleurs et stratégies gagnantes dans les jeux de parité. In *MSR: Congrès Modélisation des Systèmes Réactifs*, pages 87–98. Hermès Science, October 2001.

[51] Jens Vöge and Marcin Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV: International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2000.

[52] Igor Walukiewicz. Monadic second-order logic on tree-like structures. *Theoretical Computer Science*, 275:311–346, 2002.