MASTER IN COMPUTER SCIENCE

Application à la méthode B d'éléments de logique temporelle ATL

Safin, Antoine

Award date:
2006

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique.
Année académique 2005-2006

Application à la méthode B d'éléments de logique temporelle ATL

Antoine Safin

Mémoire présenté en vue de l'obtention du grade de licencié en informatique

Remerciements

Avant de commencer, il convient de remercier les personnes sans lesquelles ce travail n'aurait pu voir le jour.

Tout d'abord, je tiens à remercier M. Schobbens qui, dans son rôle de promoteur, a toujours pris le temps de me recevoir et de me guider dans l'accomplissement de ce travail, et également les personnes de son équipe, Emmanuel Dieul et Hubert Toussaint, qui prirent de leurs soirées, toujours avec bienveillance, pour me conseiller.

Plus généralement, je souhaite remercier toutes les personnes, professeurs, assistants, membres du personnel que j'ai cotoyés toutes ces soirées, et qui firent de ces années, dont le présent travail est l'achèvement, un succès total.

Ensuite, je souhaite remercier Charles pour sa disponibilité et sa gentillesse de chaque instant.

Enfin, un grand merci à Violaine pour son soutien indéfectible.

Table des matières

Introduction

Au cours de la dernière décennie, de nombreux développements ont été effectués dans le domaine des méthodes formelles de développement logiciel, qui offrent des perspectives intéressantes pour des systèmes critiques.

Parmi ces méthodes, nous retrouvons 2 méthodes qui constituent le contexte de travail de ce mémoire.

La méthode B, élaborée par J-R. Abrial, est en quelque sorte le successeur de la méthode Z qui eut un certain succès dans les années 80. La méthode B permet la spécification formelle de systèmes complexes par raffinements successifs, le tout fondé sur la preuve des différents composants.

Il existe en outre de nombreux outils permettant de traiter une spécification B tant en termes de preuves automatiques que de vérification de modèles (model-checking), qui ont permis à la méthode B de faire ses preuves dans le monde industriel.

ATL, pour Atlternated-time Temporal Logic (Logique temporelle du temps alterné) est une logique temporelle, théorisée ces dernières années qui se situe dans la lignée de LTL(Linear Temporal Logic) et CTL (Computational Tree Logic).

Ces logiques temporelles tentent d'étudier le comportement, au cours du temps, de systèmes informatiques en vue de pouvoir en vérifier certaines propriétés telles que la sécurité ou l'absence de deadlocks, le tout se basant sur l'étude du comportement d'automates.

De nombreux développements ont été faits dans ce domaine ces dernières années, se focalisant principalement sur les problèmes d'explosion du nombre d'états des automates associés à des systèmes complexes.

Le point de vue adopté pour ce travail est d'essayer, par le développement d'une méthode que nous avons appelée ATL-B, d'illustrer la possibilité d'un enrichissement de la méthode B par les concepts temporels définis par ATL.

L'objet du mémoire n'étant pas de concevoir une méthode complètement neuve, nous nous contenterons donc de définir une syntaxe permettant d'exprimer les concepts ATL en B, et nous analyserons les choses suivantes :

- la manière de modéliser une spécification en termes d'automates ATL

en vue d'y appliquer les techniques de vérification de modèles.
- l'impact sur la spécification en termes de preuves de la méthode B.
- la manière d'intégrer ces 2 méthodes dans un processus général de développement par raffinement.

.

Après avoir défini plus précisément la méthode B et ATL(Chapitre 1), nous expliquerons notre démarche quant à l'intégration des différents concepts(Chapitre 2). Nous expliquerons ensuite les apports syntaxiques (Chapitre 3), définirons la sémantique de ATL-B (Chapitre 4) ainsi que le raffinement tel que nous le réenvisageons (Chapitre 5). Nous terminerons par un exemple qui permettra de mettre en pratique les éléments précédemment définis (Chapitre 6)

Chapitre 1

Contexte

Avant d'aborder le vif du sujet, il convient de définir le contexte du présent document, les éléments indispensables à la suite du travail. Après une introduction à la méhode B et sa variante EventB puis à ATL, nous recadrerons ces différents éléments dans une perspective plus globale de développement en termes d'agents collaborants.

1.1 La méthode B

La méthode B, théorisée par Jean-Raymond Abrial([Abr96]) constitue une méthode originale de développement de logiciel sûr.

Cette méthode a fait ses preuves dans le monde industriel, entre autre à la RATP dans le développement de logiciel embarqué (METEOR).

B, qui prend ses origines dans la notation Z [00bds], décrit un processus formel de raffinement permettant de :

- confronter le cahier de charge à un modèle abstrait cohérent
- par palier successif (de raffinements) arriver à un modèle programmable
- prouver la conformité d'une implantation (raffinement final) par rapport au modèle abstrait.

1.1.1 Principe de base

Considérant un système à modéliser, une machine B est un module, permettant de décrire ce système de manière formelle. La machine B est au coeur de la méthode. Une machine B est fondamentalement constituée des éléments suivants :

- des variables décrivant l'état de la machine. Les variables sont contraintes par un invariant décrivant les états admissibles de la machine.

- des opérations permettant de modifier l'état de la machine : ces opérations peuvent être régies par une précondition et doivent respecter l'invariant, c'est-à-dire laisser la machine dans un état respectant l'invariant. Une opération spéciale appelée initialisation, permet de spécifier l'état initial d'une machine. Certaines opérations seront déterministes, d'autres indéterministes.
- des paramètres permettant de réutiliser une machine dans des conditions différentes : ces paramètres doivent respecter des contraintes définies par la machine.

Une machine peut être paramétrée afin de pouvoir être utilisée dans différents contextes, ces paramètres devront être actualisés lors du processus de développement en phase d'implémentation.

TAB. 1.1 – Exemple de machine B - Ensemble majoré

```
%

MACHINE
  Ensemble_Majore
VARIABLES
  y
INVARIANT
  y ∈ F(NAT1)
INITIALIZATION
  y:= ∅
OPERATIONS
  enter(n) ≡
    PRE n ∈ NAT1 THEN y := y ∪ {n} END;

  m ← maximum ≡
    PRE y ≠ ∅ THEN m := max(y) END
END

%
```

L'exemple 1.1 tiré de [Abr96] donne un petit exemple de machine B. Nous voyons modélisée une machine représentant un sous-ensemble majoré de $\mathbb{N}$. Il est possible d'insérer un entier dans un ensemble de ce type ainsi que d'en obtenir l'élément maximum.

Pour qu'une machine soit considérée comme correcte, il convient de fournir une preuve de différentes choses :

- l'initialisation respecte l'invariant.
- l'exécution de chacune des opérations, sous réserve que la précondition soit respectée, doit respecter l'invariant.

Nous verrons plus loin en quoi consiste plus formellement ces preuves mais il est assez immédiat de voir que notre exemple 1.1 peut être considéré comme cohérent.

1.1.2 Sémantique

Dans [Abr96], la sémantique de B est définie de deux manières complémentaires : en termes de substitutions généralisées, qui donnent une sémantique dans le cadre de la logique du premier ordre et une sémantique de transformateur d'ensemble plus apte à étudier des concepts d'ordre supérieur.

Substitutions généralisées

La méthode B repose, quant à sa sémantique sur la notion de substitution généralisée.

Une substitution généralisée est avant tout une construction syntaxique qui définit le langage des opérations d'une machine B.

Une substitution généralisée est, comme son nom l'indique dérivée de la notion de substitution de la logique du premier ordre à laquelle elle ajoute par rapport à sa forme simple [x :=E] les substitutions suivantes :

- la précondition P|S : $[P|S]R \Leftrightarrow P \wedge [S]R$, pour P et R deux formules, S une substitution.
- la garde P $\Longrightarrow$ S : $[P \Longrightarrow S]R \Leftrightarrow P \Longrightarrow S[R]$, pour P et R deux formules, S une substitution.
- le choix S []Q : $[S[]Q] \Leftrightarrow [S]R \vee [Q]R$, pour P et Q deux substitutions.
- la composition multiple S ||Q : $[S \parallel S]R \Leftrightarrow [S]R \wedge [Q]R$, pour P et Q deux substitutions.

Ces substitutions ne sont pas les seules mais permettent de définir toutes les substitutions utilisables en B

Transformateurs d'ensemble (set transformer)

A partir de la notion de substitution, [Abr96] donne un modèle dit "set transformer", que nous traduirons par transformateur d'ensembles.

Une machine B est vue, dès lors comme la fonction

$$str(S) \equiv \lambda p.(p \in P(s)|\{x|x \in s \wedge [S](x \in P))$$

ou S est la substitution envisagée, et s le produit cartésien des domaines des différentes variables. str(S)(p) traduit donc le lien entre les états (p) d'après-exécution de la substitution et les états antérieurs à l'application de la substitution.

Cette sémantique permet, en outre, de mieux formaliser les terminaisons de boucles en recourant à la notion de point fixe.

1.1.3 Raffinement

Le raffinement est un processus central de la méthode B par lequel on donne une version plus "implémentable" d'une machine, tout en prouvant que la seconde respecte bien la spécification décrite par la première.

Un raffinement est une relation de type "est un" entre 2 machines qui fait qu'une machine raffinement peut remplacer dans toute spécification la machine abstraite qu'elle raffine.

Au cours du procesus de raffinement, on lèvera en général l'indéterminisme des opérations, car le raffinement final, appelé implémentation, aura pour but d'être transformé en code exécutable qui ne peut, par nature, être indéterministe.

L'exemple 1.2, toujours tiré de [Abr96] fournit un raffinement (spécifié par la clause REFINES) de la machine *Ensemble_Majore* de l'exemple 1.1.

On peut montrer que cette machine est cohérente au sens que nous en avons donné dans la partie 1.1.1.

Dans l'invariant, nous voyons apparaître un invariant de collage qui permet de lier les variables de la machine $Ensemble\_Majore\_raffine$ aux variables de la machine $Ensemble\_majore$.

Pour que la relation de raffinement soit correcte, il faut en outre montrer que :

- la précondition de chacune des opérations est, au plus, aussi forte que celle de la machine abstraite. En d'autres termes, pour l'opération maximum de notre exemple, cela revient à montrer que y $\neq\emptyset \implies$ z $\neq$0 sous hypothèse de l'invariant de la machine.
- la post-condition de l'opération raffinée est incluse dans la post-condition de la machine abstraite, c'est-à-dire que, pour toute propriété P t.q. [Ensemble_Majoré_raffiné.maximum]P , on a [Ensemble_Majoré.maximum]P, autrement dit il faut montrer (pour l'opération *maximum*) que $m = z \implies m = max(y)$ sous hypothèse de l'invariant. Pour l'opération *enter*, il faut donc montrer que $z = max(z, n) \implies y = y \cup \{n\}$,

TAB. 1.2 – Exemple de raffinement - Ensemble majoré

```
MACHINE
  Ensemble_Majoré_raffiné
REFINES
  Ensemble_Majoré
VARIABLES
  z
INVARIANT
  z ∈ NAT
  z := max( { y } ∪ { 0 } )
INITIALIZATION
  z:= 0
OPERATIONS
  enter(n) ≡
    PRE n ∈ NAT1 THEN z := max(z,n) END;

  m ← maximum ≡
    PRE z ≠ 0  THEN m := z END
END
```

toujours sous hypothèse de l'invariant.

Nous voyons directement, dans l'exemple, l'intérêt d'un tel processus. La machine abstraite donne une description très limpide de ce qu'est un ensemble majoré (même si son comportement est réduit au minimum) A l'inverse, la machine raffinée, qui ne ressemble plus à la spécification de base s'avérera plus efficace à l'implémentation.

Le raffinement nous procure donc un moyen de donner différentes lectures d'un système : une lecture de haut niveau (la machine abstraite), facile à confronter aux besoins de l'utilisateur final, et une version de bas niveau(le raffinement) qui s'avère plus adaptée à l'implémentation (en termes d'efficacité, de modularité ou tout autre critère). Les preuves fournies assurent que la deuxième se comporte exactement comme la première.

1.1.4 Construction de systèmes complexes

La méthode B fournit également des moyens afin de construire des systèmes assez conséquents de façon modulaire à l'aide de différentes constructions telles que INCLUDES, SEES, USES et IMPORTS définis de cette façon :

- INCLUDES : cette clause est l'élément principal d'une construction modulaire de spécification. La clause INCLUDES permet de combiner différentes spécifications dans une seule. Inclure une machine nécessite d'actualiser les paramètres de la machine incluse. Pour des raisons d'encapsulation, les variables de la machine incluse ne sont pas modifiables directement dans la machine incluante (uniquement via les opérations de la machine incluse.)
- SEES : cette clause permet de spécifier qu'une machine voit une autre, c'est-à-dire qu'elle peut lire les variables de cette dernière mais pas les modifier. Une clause SEES, au contraire de la clause INCLUDES n'instancie pas la machine vue, et ne fournit donc pas les paramètres effectifs.
- USES : qui permet de copier la spécification d'une machine dans une autre ; on accole donc, clause par clause, les clauses de la machine utilisée dans la machine utilisante.
- IMPORTS : la clause est identique à la clause INCLUDES à ceci près qu'elle ne s'applique qu'à des implémentations.

1.1.5 Calculabilité

Les différents outils B existants fournissent en général un outil d'aide à la preuve, capable de réaliser la plupart des preuves automatiquement, même si dans certains cas, l'intervention humaine est nécessaire.

D'autres outils permettent également d'effectuer de la vérification de modèles(model checking). Cette technique pourra servir là où les preuves ne sont pas automatisables ou afin de fournir un contre-exemple lorsqu'une spécification est fausse.

1.1.6 EventB

EventB est une extension de B (voir [AM98] et [Abr99]) qui permet de donner une vision dynamique des systèmes étudiés.

L'idée est de décrire le comportement non plus en termes d'opérations mais en termes d'événements (la clause OPERATIONS est remplacée par une clause EVENTS). Un événement peut être déclenché, c-à-d. exécuté, à condition que la garde de cet événement soit satisfaite.

Une différence majeure de la façon de voir les choses est qu'on ouvre la machine : dans la méthode B, on avait une vue monolithique de ce qui se

passe. Une machine exécute les opérations d'une ou plusieurs autres, mais en fin de compte, une machine contrôle le déroulement du système entier. En EventB, aucun module supérieur ne dirige la manoeuvre : les événements sont susceptibles d'être déclenchés à tout moment et le système doit être assez réactif pour rester dans un état cohérent.

On peut alors spécifier des invariants spécifiques (appelés dynamiques) qui permettent de donner des invariants quant au comportement de la machine dans le temps. La satisfaisabilité de ces propriétés temporelles du système se vérifie à l'aide d'invariants et de variants à fournir, assimilant ainsi le comportement du système à une substitution généralisée du type "boucle".

Au niveau du raffinement, les choses évoluent également. En effet, pour un événement abstrait, nous allons pouvoir introduire plusieurs événements concrets (en fait un événement concret du même nom ainsi que de nouveaux événements). Le comportement de la machine concrète identique par rapport à la machine abstraite est assuré de par le fait que la garde de l'événement abstrait doit être couverte par les gardes des événements concrets (c'est à dire que la disjonction des gardes des événements concrets doit impliquer la disjonction des gardes des événements abstraits) ; en d'autres termes, la machine raffinée doit s'exécuter au moins aussi souvent que la machine abstraite.

Obligation de preuves

Les preuves à donner pour un système EventB sont, en plus des obligations de preuves de B , les suivantes([Abr99]) :

- la preuve de la limitation du renforcement des gardes des événements : pour éviter qu'une machine concrète agisse moins souvent que la machine abstraite, il convient de prouver, pour chaque événement abstrait, que la garde de ce dernier est moins restrictive que la disjonction des gardes de l'événement concret et des nouveaux événements de la machine concrète. Ceci permet de rajouter des événements dans le raffinement.
- preuve de l'impossibilité, pour les nouveaux événements, de monopoliser l'activité : il faut s'assurer que les nouveaux événements ne peuvent pas empêcher la machine concrète de fonctionner comme la machine abstraite, en restant "coincés" dans les nouveaux événements de la machine concrète.

1.2 ATL

ATL (Alternating-Time Temporal Logic) est une logique développée à l'origine par Rajeev Alur, Thomas A. Henzinger et Orna Kupferman [AHK02] et qui découle des développements précédents en logique temporelle [GHR94] ainsi que de la théorie des jeux.

Une implémentation de ATL a été développée via le projet MOCHA [ocB].

1.2.1 LTL,CTL

LTL (Linear Temporal Logic) et CTL (Computational Tree Logic), sont deux logiques permettant de décrire le comportement au cours du temps d'un système.

Il faut voir le système comme un automate dont les états sont libellés à l'aide de certaines propositions. Cet automate, appelé structure de Kripke va être à la base des logiques ici définies.

Une notion sur laquelle se base ces logiques est celle de chemin(path). Un chemin (d'exécution) d'un système (décrit sous forme de structure de Kripke) est une suite d'états , pour laquelle il existe naturellement une transition entre des éléments successifs. Cela nous donne donc une exécution possible de l'automate.

LTL

LTL, sur base d'une structure de Kripke permet de formuler certaines propriétés du système.

Etant donné un ensemble de propositions atomiques (qui libellent les états de l'automate), une formule LTL [1] est une formule qui étudie un chemin, à partir d'un état donné, en combinant ces propositions à l'aide des opérateurs logiques de la logique des propositions ET, OU et NON, et qui accepte des opérateurs temporels qui sont les suivants :

- $\bigcirc$, opérateur Next : $\bigcirc$P est vrai si l'état suivant du chemin rend P vrai.
- $\Diamond$opérateur Eventually : $\Diamond$P est vrai si P deviendra vrai dans la suite
- $\Box$opérateur Always : $\Box$P est vrai si P est vrai dans tous les états suivant l'état actuel.
- $\mathcal{U}$opérateur binaire est vrai : P $\mathcal{U}$Q est vrai si $\Diamond$Q est vrai et que P est vrai pour tous les états entre l'état actuel et le premier état ou Q est vrai.

A l'aide de ces formules, on peut dès lors étudier les chemins d'exécution d'une structure de Kripke.

[1] Définition tirée de [Wik06] et [BBF+01]

CTL

CTL (Computational Tree Logique) ajoute, par rapport à LTL la notion de quantifieur de chemin (path quantifier), qui permet d'étudier l'arbre des exécutions possibles de l'automate considéré.

Une formule CTL est une formule qui étudie une structure de Kripke dans un état donné x en reprenant la syntaxe de LTL et en y ajoutant les quantificateurs de chemin suivants :

- A , opérateur All : A P est vrai si la formule LTL P est vraie dans tous les chemins possibles contenant x.
- E , opérateur Exists : E P est vrai s'il existe un chemin passant par x tel que la formule LTL P soit vraie.

Nous le voyons, CTL permet de décrire des propriétés temporelles d'un système de manière générale.

Dans la littérature([BBF+01]), on retrouve une distinction entre CTL et CTL^*. CTL^*, plus général que CTL, permet dans une formule de combiner plusieurs quantificateurs de chemin au sein de cette formule,contrairement à CTL, qui n'admet de quantificateur de chemin qu'en début de formule. Par abus de langage, nous parlerons de CTL en faisant référence à CTL^* car :

- CTL^* est plus général que CTL : en effet, on peut exprimer des choses du type E□A◇P (il existe une manière de par laquelle je pourrais toujours obtenir P dans le futur) impossible à définir en CTL.
- CTL^* n'est pas plus complexe dans sa calculabilité : les algorithmes de résolution ([BBF+01]) de formules CTL^* sont les mêmes que CTL et leur complexité n'augmente pas significativement.

1.2.2 Principe de base de ATL

ATL est un paradigme qui étudie un système en termes d'agents évoluants dans un système ouvert.

L'évolution du système se fait en termes de tours. A chaque tour, les agents vont évoluer. Ils peuvent évoluer de 2 manières :

- chacun à son tour : chaque agent agit sur le système l'un après l'autre.
- simultanément : les agents agissent sur le système tous en même temps.

En pratique, on utilise [HA98a] un compromis entre les 2. Tous les agents jouent en même temps mais on peut définir un ordonnancement. Ceci est effectué en indiquant que certains agents vont attendre d'autres agents avant de jouer.

Modélisation

Une fois une série d'agents, ou de "Reactive Modules" (MOCHA) décrits, on va vouloir étudier certaines propriétés du système ainsi obtenu. Les principales propriétés que l'on cherche à étudier sont, selon [AHK02], les suivantes :

- réceptivité du système : le système peut-il continuer à s'exécuter indéfiniment, c'est-à-dire à rester vivant ?
- réalisabilité d'un objectif : étant donné un objectif P (donné sous la forme d'une formule logique), est-il possible de rendre P vrai ?
- possibilité de supervision : est-il possible d'ajouter un agent pour superviser le système, c'est-à-dire assurer que ce dernier satisfasse certaines propriétés ?

Pour se faire, il est possible d'indiquer, et c'est ceci l'apport principal par rapport à CTL, que certains agents vont coopérer.

Syntaxe

Une propriété du système sera donnée en une formule CTL dans laquelle on remplace les quantificateurs de chemin par une expression de la forme $<< A >> P$, où A représente l'ensemble des agents coopérant en vue de l'objectif P, et où P est une formule de logique temporelle LTL.

Nous avons, entre autres, en considérant Σ l'ensemble des agents du système, les propriétés remarquables suivantes liant CTL et ATL :

$$AP \equiv << \emptyset >> P \text{ et } EP \equiv << \Sigma >> P$$

1.2.3 Sémantique

ATL fournit une sémantique basée sur une structure de jeu concurrent (CGS : Concurrent Game Structure) qui permet d'étudier l'exécution (computation) globale d'un système.

CGS

Un CGS est (tiré de [AHK02]) un modèle mathématique $< k, Q, \Pi, \pi, d, \delta >$ composé de :

- k le nombre d'agents(joueurs) du système. Chaque joueur est identifié par un numéro distinct entre 1 et k.
- Q l'ensemble des états du système.
- Π un ensemble de propositions, dites observables.
- π une fonction d'étiquetage(labeling function) qui, à chaque état q, définit l'ensemble $\pi(q)$ des propositions (incluses dans Π) vraies à l'état q.

- d, une fonction qui, pour chaque état q du système et chaque agent a, définit un nombre $d_a(q) \in \mathbb{N}$ de mouvement possible. Chaque opération possible à l'état q est donnée par un chiffre entr 1 et $d_a(q)$. On définit également, pour chaque état q, l'ensemble $D(q) = \{\{1, ..., d_1(q)\} \times ... \times \{1, ..., d_k(q)\}\}$ des mouvements possibles. Cette fonction définit donc l'ensemble des vecteurs de mouvement possibles à l'état q.
- δ une fonction qui à chaque état q et chaque vecteur $< j_1, ..., j_k >$, associe un état $\delta(q, j_1, ..., j_k) \in Q$ qui représente l'état d'arrivée du système après le mouvement. δ est la fonction de transition(transition function).

Le principal inconvénient de cette représentation est un phénomène d'explosion du nombre d'états. En effet, en général, chaque état va représenter une valuation du système, c'est-à-dire l'attribution d'une valeur à chaque variable du système, ce qui pour des domaines de variable importants génère un nombre gigantesque de'états.

Exécutions et stratégie

A partir de la notion de CGS, on définit une exécution (computation) d'un CGS $< k, Q, \Pi, \pi, d, \delta >$ comme étant une suite $q_1, ..., q_n$ d'états tels que, pour 2 états successifs q_i, q_{i+1}, il existe un vecteur de mouvement $< j_1, ..., j_k > \in D(q)$ tel que $\delta(q_i, j_1, ..., j_k) = q_{i+1}$. On étend cette notion à la notion de traces comme étant la suite $\pi(q_0)...\pi(q_n)$.

En vue de pouvoir évaluer une formule ATL, il convient également de définir la notion de stratégie.

Une stratégie, d'un agent A du CGS $< k, Q, \Pi, \pi, d, \delta >$ est une fonction f_a qui, à toute séquence finie λ d'états de Q, associe un nombre $f_a(\lambda) \leq d(q)$, pour q le dernier état de la séquence λ. En d'autres termes, cette fonction $f_(a)$ définit les actions qui seront prises par l'agent A lors de l'évolution du système.

La résultante (outcome) d'un ensemble $F_A = \{f_a : a \in A\}$ de stratégies, une pour chaque agent de A, est l'ensemble

$$out(q, F_A) = \left\{ \begin{array}{l} \lambda = q_0...q_n \in Q^* \\ \quad t.q.q_0 = q \\ \quad \wedge \forall i > 0 \; \exists < j_1, ..., j_k > \in D(q_i) \\ \qquad t.q.j_a = f_a(\lambda[0...i]) \\ \qquad \wedge \delta(q_i, j_1, ..., j_k) = q_{i+1} \end{array} \right\}$$

En d'autres termes, la résultante d'un ensemble de stratégies définit toutes les exécutions pouvant être produites par cet ensemble.

Evaluation d'une formule ATL

A partir des définitions précédentes, nous pouvons définir la sémantique d'une formule ATL de la manière suivante : étant donné un CGS $< k, Q, \Pi, \pi, d, \delta >$ et un ensemble d'agents A, la formule $<< A >> P$ est vraie pour un état $q \in Q$ s'il existe un ensemble F_A de stratégies , une pour chaque agent de A, tel que la formule soit vraie pour toute exécution résultante de F_A.

On aura donc que :

- $<< A >> \bigcirc p$ pour q donné si, pour toute exécution $\lambda \in out(q, F_A)$, p est vrai à l'état $\lambda[1]$
- $<< A >> \Diamond p$ pour q donné si, pour toute exécution $\lambda \in out(q, F_A)$, il existe un i ≤ 0 tel que p soit vrai à l'état $\lambda[i]$.
- $<< A >> r\mathcal{U}p$ pour q donné si, pour toute exécution $\lambda \in out(q, F_A)$, il existe un i$\leq$ 0 tel que p soit vrai à l'état $\lambda[i]$ et que, pout tout j $0 \leq j < i$, r soit vrai.

1.2.4 Raffinement

Tout comme pour B, on peut définir en ATL une relation de raffinement. Dans ([AHKV98]), on définit 2 types de relation de raffinement : par simulation et par inclusion de trace.

Un raffinement est, en ATL, défini en fonction d'agents et dans le cadre des stratégies que ces derniers peuvent adopter.

Raffinement par simulation

Dans le contexte de raffinement par simulation, un système S'$= < k, Q, \Pi', \pi', d', \delta' >$ raffine S $= < k, Q, \Pi, \pi, d, \delta >$, en considérant un sous-ensemble d'agent $A \subset Q$ si il A-simule S, c'est-à-dire qu'il existe une relation de A-Simulation $H \subset Q \times Q'$ telle que :

- $\pi(q) = \pi'(q)$
- pour toute stratégie de A dans S, il existe une stratégie de A dans S' tel que , pour toute stratégie de $\Sigma \backslash A$ dans S', il existe une sratégie de $\Sigma \backslash A$ dans S telle que la résultante des stratégies dans S soit en relation de A-simulation avec la résultante des choix dans S' (H(q,q')).

En d'autre termes, il faut que les stratégies de A dans S ne soient pas "contrables" dans S' par les antagonistes de A.

Raffinement par inclusion de trace

Le raffinement par inclusion de trace (trace containment) consiste à voir le raffinement de la manière suivante : un système S est un A-raffinement du système S' si l'ensemble des traces que A peut provoquer dans S, c'est-à-dire pour lesquelles il existe une stratégie des éléments de A, est contenue dans

l'ensemble des traces que A peut provoquer dans S'.

C'est ce raffinement qui est en général utilisé dans la littérature.

Raffinement par rôles

Dans [RS02], les auteurs proposent une variante de raffinement : le raffinement par rôles.

Basé sur la notion de raffinement de signature défini, étant donné 2 CGS $< \Sigma, Q, \Pi, \pi, d, \delta >$et $< \Sigma\prime, Q\prime, \Pi\prime, \pi\prime, d\prime, \delta\prime >$et 2 ensembles d'agents A et A', une paire de fonctions(f,s) telle que :

- f lie les agents de A aux agents de A' (et ceux de $\Sigma \backslash A$ à ceux de $\Sigma\prime \backslash A\prime$)
- g liant les variables des 2 systèmes (en respectant naturellement, au niveau de l'appartenance à un agent, la relation f).

A partir de cette notion un raffinement S' $= < \Sigma\prime, Q\prime, \Pi\prime, \pi\prime, d\prime, \delta\prime >$d'un système S $= < \Sigma, Q, \Pi, \pi, d, \delta >$, prolongeant la signature (f,g) de A vers A', est un triplet (H,C,D) ayant les propriétés suivantes :

- H est une relation qui lie les états de S et S', en considérant la valeur des variables (en respectant g).
- C est une fonction qui pour tous états Q, Q' respectant H, fait correspondre les choix des agents de A' à ceux de A.
- D est une fonction qui pour tous états Q, Q' respectant H, fait correspondre les choix des agents de $\Sigma \backslash A$ à ceux de $\Sigma\prime \backslash A\prime$

Afin de préserver le comportement dynamique du système, il faudra veiller à ce que , pour tout choix de A persistant dans S', les choix de $\Sigma\prime \backslash A\prime$ ayant une correspondance dans S mènent à un état satisfaisant H. En d'autres termes, si des choix correspondants sont faits dans S et S', ils doivent mener à des états satisfaisants H.

Cette notion de raffinement travaille donc à deux niveaux : on spécialise les éléments de A , et on généralise les éléments en dehors de A.

Cette relation de raffinement préserve, comme démontré dans [RS02], les stratégies des antagonistes (c'est-à-dire de $\Sigma \backslash A$). En effet, les antagonistes pouvant opérer plus de choix, ils gardent les possibilités qu'ils avaient dans S. Par contre, ce n'est pas le cas des éléments de A.

Par rapport aux raffinements par simulation et par inclusion de trace, une différence majeure existe dans le fait qu'un raffinement par rôles permet de changer les agents et les variables entre le système raffiné et le système abstrait, la correspondance étant assurée par la signature du raffinement.

1.2.5 Calculabilité

A partir de modèle décrit en ATL, dont une des implémentations est fournie par l'outil MOCHA [ocB] au travers de Reactive modules [HA98b], il est intéressant d'étudier le modèle ainsi obtenu.

La vérification d'un modèle ATL (model checking) consiste donc à vérifier qu'un modèle respecte une formule ATL donnée, c'est-à-dire qu'il rend cette dernière vraie.

Cette vérification est effectuée à l'aide d'une vérification symbolique ([BBF+01] et [AHK02]) se basant sur des Ordered Binary Decision Diagrams(OBDD) et est effectuable en un temps polynomial.

Malheureusement, les techniques de model-checking souffrent d'un problème majeur : l'explosion du nombre d'états du CGS. En effet, les différentes techniques vont construire un automate reprenant un état par valuation, ceci pouvant rapidement devenir, pour des systèmes un tant soit peu complexes, gigantesque.

Les 2 techniques les plus classiques pour contrer ce phénomène sont les suivantes :

- le model checking symbolique qui essaye de réduire le nombre d'états du modèle en regroupant les états selon des propriétés communes de ces derniers, plusieurs valuations pouvant alors être représentées par un seul état.
- l'abstraction (inverse du raffinement), qui permet de donner une vue plus abstraite du système (et donc réduite quant au nombre d'états). Le problème est que, en général, seules certaines propriétés de sûreté (safety cfr [BBF+01]) sont préservées du système abstrait au système concret.

1.3 Objectifs

Nous avons décrit, au cours des 2 précédents chapitres, 2 visions de la modélisation formelle de systèmes informatiques.

La première, la méthode B, donne une décomposition statique d'un système et permet, à l'aide de preuves de développer rigoureusement des systèmes. Elle est, en outre, utilisée dans l'industrie et a déjà largement fait ses preuves.

La seconde, ATL, donne une vision dynamique d'un système, et permet d'étudier un système en termes d'agents aux intérêts divergents. Cette technique étudie donc des systèmes ouverts.

Nous avons également mentionné EventB, qui tente de donner une notion dynamique à B, tout en restant profondément lié à la logique des substitutions qui est à la base de B.

L'idée de ce travail est donc d'étudier la piste d'un enrichissement de la méthode B d'éléments d'ATL, permettant une étude de la dynamique de systèmes B ouverts.

Nous envisagerons cette intégration d'un point de vue pratique, c'est-à-dire que nous n'essayerons pas de créer un nouveau paradigme mais de donner un système permettant d'utiliser les développements effectués dans les 2 domaines, tout en gardant une cohérence globale et une expressivité optimale.

Chapitre 2

Méthodologie

Dans cette partie, nous allons décrire la méthodologie appliquée en vue d'obtenir un processus de développement formel que nous nommerons dès à présent ATL-B.

L'idée est donc, comme nous l'avons souligné en conclusion du chapitre précédent, de fournir une extension de B, que nous appellerons dès à présent ATL-B, permettant de gérer les aspects temporels d'une spécification en utilisant les développements effectués dans le cadre de ATL.

Etant donné que le cadre de ce travail n'est pas de redéfinir complètement une nouvelle méthode, mais plutôt de voir comment pouvoir intégrer 2 outils de spécification, nous avons décidé d'aborder la chose en assumant l'ambivalence des spécifications proposées.

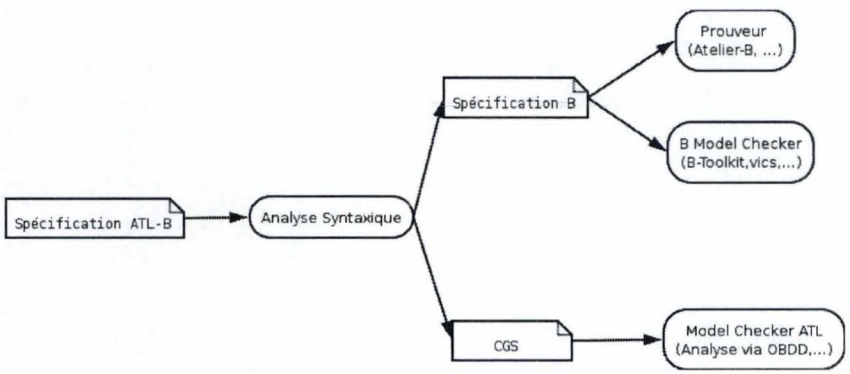

FIG. 2.1 – Processus de décomposition d'une spécification ATL-B

La figure 2.1 illustre notre processus de transformation, permettant, à partir d'une spécification ATL de générer 2 choses :

- une spécification B décrivant le comportement statique des différents agents du système. Cette spécification pourra directement être traitée par les outils existants pour vérifier une spécification B, tant via model-checking que via les prouveurs tels qu'Atelier-B.
- un CGS (ou plutôt un BCGS comme nous le verrons dans le chapitre 4) qui reprend une description du comportement temporel de notre système.

Il est à remarquer que , pour ce qui est du traitement de la spécification B obtenue à partir de ATL-B, il y aura un processus complet d'analyse syntaxique et sémantique qui suivra, contrairement à la partie ATL, qui sera traitée sous une forme retravaillée de CGS. Ceci se justifie pour différentes raisons :

- B est une méthode de développement basée sur une syntaxe clairement définie, au contraire d'ATL, qui, même si des implémentations telles que MOCHA existent, ne définit pas de syntaxe quant aux sytèmes spécifiés (même si une syntaxe existe bel et bien pour les formules ATL.)
- comme nous l'avons déjà indiqué, de nombreux outils existent pour traiter directement une spécification B. Ceci n'est pas le cas pour ATL, mais les techniques de model checking applicables à des automates tels que les CGS ont fait l'objet de beaucoup de publications scientifiques.

Afin d'illustrer le sujet, nous nous proposons de montrer, au travers d'un petit exemple, la méthodologie employée.

2.1 Un petit exemple ATL-B : L'ascenceur

Anticipant un peu sur la définition que nous donnerons plus loin du langage ATL-B, ce cas représente un cas assez classique d'un ascenceur. Ce système se compose des agents Lift(2.1), Utilisateur(2.2) qui intéragissent.

Même si la syntaxe n'est pas encore clairement définie, ceci nous donne une idée plus claire de la description que nous voudrions obtenir du système.

Nous avons, dans ce petit exemple, réduit le problème au minimum, afin de faire simple et étant donné qu'il s'agit juste ici d'illustrer le propos. Nous avons donc volontairement omis, par exemple, la spécification de l'interaction avec les boutons de l'ascenceur.

L'idée est donc de compléter B à l'aide des clauses supplémentaires : AGENT, SYSTEM, DYNAMIC CONSTRAINT,ACTION, etc...(nous serons complet à ce sujet dans la partie suivante.)

2.1.1 L'ascenceur

TAB. 2.1 – Agent ascenceur ATL-B

```
AGENT Lift
  VARIABLES
    posl,isopen
  INVARIANT
    posl ∈ 0..FloorMax
    isopen ∈ BOOL
  EXTERNAL_VARIABLES
    RequestFloor, RequestedFloor
  INITIALISATION
   posl :∈ 0...FloorMax || isopen :∈ BOOL
  OPERATIONS
    up ≡
     PRE  posl ≤  MaxFloor - 1  THEN
       posl:= posl + 1
  down ≡
     PRE posl ≥ 1 THEN
       posl := posl - 1
    open ≡
     PRE ¬ isopen
       open := ¬ TRUE ||  FloorsRequested = FloorsRequested - posl
    close ≡
     PRE isopen
      open := false
  ACTION
     posl ≤  MaxFloor - 1  ⟹  Up
     [] posl ≥ 1  ⟹  Down
     [] ¬ isopen  ⟹  open
     [] isopen  ⟹  close
     [] skip
END
```

L'agent ascenceur *Lift* (table 2.2) donne la spécification du composant ascenceur du système, cet agent contrôle la position de l'ascenceur(*posl*) ainsi que l'ouverture-fermeture des portes (*open*).

Il peut, au cours de l'exécution du système, ouvrir la porte (*open*), fermer la porte (*close*), monter (*up*) ou descendre (*down*).

L'opérateur :∈utilisé à l'initialisation veut dire qu'on assigne à la variable

située à gauche une valeur quelconque dans l'ensemble défini à droite. C'est une substitution indéterministe.

2.1.2 L'utilisateur

L'agent Utilisateur *User* (table 2.2)représente l'utilisateur de l'ascenceur, il modélise la personne désirant se rendre à un étage donné via l'ascenceur.

Il contrôle les éléments suivants :
- sa position en dehors ou à l'intérieur de l'ascenceur (*lu* qui est vrai si l'utilisateur est dans l'ascenceur)
- la demande faite à l'ascenceur pour désservir l'étage (si *RequestFloor* est vrai, l'utilisateur a demandé l'étage *RequestedFloor*).
- sa position dans les étages *posu*, qui ne change que lorsque l'utilisateur sort de l'ascenceur.

Les opérations que l'agent utilisateur peut effectuer sont les suivantes :
- la demande d'étage faite depuis l'intérieur (*pushin*) qui simule le fait, dans un système concret d'appuyer sur un bouton intérieur. L'utillisateur peut donc demander n'importe quel étage.
- la demande d'étage faite depuis l'extérieur (*pushout* qui simule le fait d'appuyer sur un bouton à l'extérieur de l'ascenceur, signifiant la volonté de l'utilisateur que l'ascenceur se rende à son étage.
- le fait d'entrer dans l'ascenceur (*enter*) pour peu que ce dernier soit ouvert au même étage que l'utilisateur, l'utilisateur étant à l'extérieur de l'ascenceur.
- le fait de sortir de l'ascenceur (*exit*) pour peu que ce dernier soit ouvert, et l'utilisateur à l'intérieur.

2.1.3 Le système

La spécification du système (table 2.3) rassemble les agents précédemment spécifiés et indique via la formule ATL $<< Lift >> \Box(RequestFloor \wedge i = RequestedFloor \implies \Diamond(posl = i \wedge open))$ que l'agent ascenceur doit conduire l'utilisateur où il a émis le souhait de se rendre.

2.2 Spécification B associée

Etant donné la richesse des éléments de B, nous voudrions quelque chose qui, par rapport à l'exemple précédent, se rapproche plus de B tant sémantiquement que syntaxiquement.

Dans les exemples 2.4, 2.5 et 2.6, nous obtenons une spécification B qui possède le même comportement que le système ATL-B. En fait, le seul

TAB. 2.2 – Agent utilisateur ATL-B

```
AGENT User
  EXTERNAL_VARIABLES
    posl,isopen
  EXTERNAL_CONSTRAINTS
    posl ∈ 0..FloorMax
    isopen ∈ BOOL
  VARIABLES
    lu, posu , RequestFloor, RequestedFloor
  INVARIANT
    posu ∈ 0..FloorMax
    lu ∈ BOOL
    RequestFloor ∈ BOOL
    RequestedFloor ∈  0...FloorMax
  INITIALISATION
    lu, posu , RequestFloor, RequestedFloor
        := FALSE,  choice(0...FloorMax) , FALSE, choice(0...FloorMax)
  OPERATIONS
    pushin ≡
      ANY i WHERE i ∈ 0...FloorMax
      RequestFloor, RequestedFloor = TRUE,i
    pushout ≡
      RequestFloor, RequestedFloor = TRUE, posu
    exit ≡
      PRE  lu  THEN
        lu , posu, RequestFloor := FALSE ,posl, FALSE
    enter ≡
      PRE ¬ lu  THEN
        lu, RequestFloor :=TRUE , FALSE
  ACTION
      lu  ⟹  pushin
      [] ¬ lu  ⟹  pushout
      [] isopen ∧ posl = posu ∧ lu  ⟹  exit
      [] isopen ∧ posl = posu ∧ ¬ lu  ⟹  enter
      [] skip
END
```

élément manquant est la clause DYNAMIC CONSTRAINTS qui spécifie le comportement dynamique en terme de stratégie de notre système mais qui

TAB. 2.3 – Système ATL-B

```
SYSTEM  system
AGENTS
    User,Lift
  DYNAMIC CONSTRAINTS
  <<   (>>Lift) □ ( RequestFloor ∧ i = RequestedFloor      ⟹
◇(posl=i∧ open))
```

sera étudié via les techniques d' ATL.

Les principales transformations opérées pour les agents sont les suivantes :

- la transformation de la clause ACTION en une opération *Machine_Action*.
- le remplacement des clauses EXTERNAL_VARIABLES et EXTERNAL_CONSTRAINTS par une clause SEES reprenant les agents du système afin qu'un agent puisse voir les variables des autres machines. Les machines de la clause SEES devant naturellement être compatibles avec la clause EXTERNAL_CONSTRAINTS de l'agent original.

Les principales transformations opérées pour le système sont les suivantes :

- l'inclusion, via la clause INCLUDE des agents du système.
- la suppression de la clause DYNAMIC CONSTRAINTS qui, n'ayant pas d'équivalent en B, n'est pas utilisée dans le modèle B.
- l'adjonction d'une opération, exécutant en parallèle les actions des différents agents (nous montrerons, dans la section 4.2.1 que cette opération est conforme malgré les restrictions de B.)

Nous voyons illustré, dans ces exemples, le fait qu'on peut obtenir assez immédiatement une spécification B reprenant le caractère statique de notre système ATL-B de départ et ainsi de prouver les propriétés de l'invariant de nos différentes machines.

Cette spécification peut être soumise à n'importe quel outil permettant de traiter une spécification B pour en prouver la correction.

TAB. 2.4 – Machine ascenceur B

```
MACHINE Lift
  SEES
     User
  VARIABLES
    posl,isopen
  INVARIANT
    posl ∈ 0..FloorMax
    isopen ∈ BOOL
  INITIALISATION
   posl :∈ 0...FloorMax || isopen :∈ BOOL
  OPERATIONS
    up ≡
      PRE  posl ≤  MaxFloor - 1  THEN
        posl:= posl + 1
  down ≡
      PRE posl ≥ 1 THEN
        posl := posl - 1
    open ≡
      PRE ¬ isopen
        isopen := ¬ TRUE ||  FloorsRequested = FloorsRequested - posl
    close ≡
      PRE open
        isopen := false
  ACTION
   Machine_action ≡
      SELECT posl ≤  MaxFloor - 1 THEN Up
      WHEN  posl ≥ 1 THEN Down
      WHEN ¬ isopen THEN open
      WHEN isopen THEN close
      WHEN TRUE THEN skip
END
```

2.3 CGS associé

De l'autre côté, nous allons fournir un CGS (cfr section 1.2.3) du système ATL-B de l'exemple.

Un CGS donné est donc un tuple $< k, Q, \Pi, \pi, d, \delta >$ tel que :

- $k = 2$, 1 pour l'ascenceur et 2 pour l'utilisateur
- Q qui représente l'ensemble des valuations possibles des variables du

TAB. 2.5 – Machine utilisateur B

```
MACHINE User
  SEES
    Lift
  VARIABLES
    lu, posu , RequestFloor, RequestedFloor
  INVARIANT
    posu ∈ 0..FloorMax
    lu ∈ BOOL
    RequestFloor ∈ BOOL
    RequestedFloor ∈  0...FloorMax
  INITIALISATION
     posu :∈ 0..FloorMax
    || lu :∈ BOOL
    || RequestFloor :∈ BOOL
    || RequestedFloor :∈  0...FloorMax
  OPERATIONS
    pushin ≡
      ANY i WHERE i ∈ 0...FloorMax
      RequestFloor, RequestedFloor = TRUE,i
    pushout ≡
      RequestFloor, RequestedFloor = TRUE, posu
    exit ≡
      PRE  lu  THEN
        lu , posu, RequestFloor := FALSE ,posl, FALSE
    enter ≡
      PRE ¬ lu  THEN
        lu, RequestFloor :=TRUE , FALSE

    Machine_action ≡
      SELECT lu THEN pushin
      WHEN ¬ lu THEN pushout
      WHEN isopen ∧ posl = posu ∧ lu THEN exit
      WHEN isopen ∧ posl = posu ∧ ¬ lu THEN enter
      WHEN TRUE THEN skip

END
```

système, c'est-à-dire des différents agents du système.

- Il reprend les propositions, pour chaque variable du système V_i les valeurs $v_{i,j}$ du domaine de V_i pour cette dernière sous forme de la

TAB. 2.6 – Machine Système ATL-B

```
MACHINE  system
  VARIABLES
  INVARIANT
  INCLUDE
    User,Lift
  OPERATIONS
    Machine_action ≡
      User.Machine_action || List.Machine_action
END
```

proposition $V_i = v_{i,j}$. On a donc une proposition possible pour chaque valeur de chaque variable.

- π qui étiquette chaque état à l'aide des valuations du système
- d qui, pour chaque état du système (et donc chaque valuation de ce dernier), donnera les actions possibles pour chaque agent, ces choix étant déclarés dans la clause ACTION et soumis à une garde (un choix n'étant possible que si la garde est respectée.)
- δ qui lie les valuations successives possibles de chaque état du système.

Comme indiqué dans la section 1.2.3, nous voyons que, pour ce simple exemple, nous obtenons un nombre déjà impressionnant d'états.

Il est maitenant possible de soumettre ce CGS aux méthodes de model checking applicables à un CGS afin de vérifier notre contrainte dynamique.

2.4 Justification

Dans les exemples précédents, nous avons donc vu que, à une spécification en langage ATL-B somme toute assez expressive en soi, nous pouvons faire correspondre un ensemble de machines B, ainsi qu'un automate de type CGS(section 1.2.3) utilisable afin d'étudier le système sous l'angle ATL.

En analysant l'exemple précédent, nous voyons que seules quelques clauses nouvelles sont nécessaires en vue d'obtenir un langage aussi expressif que B, nous permettant de faire des analyses sur un CGS.

Cette approche nous permet de tirer profit des développements théoriques et pratiques abondants dans la méthode B, et d'y greffer les notions de théorie des jeux propres à ATL.

2.5 Démarche générale

La démarche adoptée se fera donc en plusieurs temps.

Dans un premier temps, nous donnerons une syntaxe de ATL-B, partant comme de bien entendu de la syntaxe B, ainsi qu'un procédé de simplification de la syntaxe en vue d'obtenir un système B, équivalent du point de vue statique à notre système ATL-B.

De ce système ATL-B, nous verrons comment donner une sémantique. Pour cela, nous aborderons le problème en mettant en avant la dualité (à la fois ATL et B) d'un système ATL-B et il conviendra, au point de jonction, de montrer la cohérence du système considéré.

Enfin nous envisagerons la notion de raffinement, non abordée dans ce chapitre, dans le nouveau modèle. Là aussi, nous verrons comment combiner un raffinement B et un raffinement ATL de type "raffinement par rôle" vue dans la section 1.2.4.

Chapitre 3

Syntaxe ATL-B et transformation vers B

Dans cette partie, nous allons définir les éléments syntaxiques nécessaires à la mise en place d'une méthode ATL-B, permettant d'apporter à B la partie de modélisation dynamique d'un système.

Dans un premier temps, nous définirons les apports syntaxiques apportés à B en vue de pouvoir décrire des systèmes ATL-B.

Dans un second temps, nous décrirons complètement la transformation que nous avons abordée dans le chapitre précédent permettant d'obtenir, par transformation syntaxique une spécification B à partir d'un système ATL-B.

3.1 Syntaxe

Pour une définition complète de la syntaxe ATL-B, le lecteur se reportera à l'annexe 6.4 issue de la grammaire B tirée de [Cle02] et modifiée, reprenant une définition exhaustive de la syntaxe ATL-B. En plus de cette syntaxe nous admettrons la syntaxe B classique, qui servira principalement à la modularité en permettant de regrouper dans des machines certains comportements.

Nous allons aborder dans cette section les éléments principaux apportés à B.

3.1.1 AGENT

La clause AGENT est une nouvelle clause permettant de spécifier un agent de notre système ATL-B. D'un point de vue syntaxique, il est assimilable à une machine B, à ceci près qu'il doit contenir une clause ACTION.

Cette clause ACTION reprend une sélection d'opérations possibles, équivalentes, d'un point de vue B, à une substitution de sélection, indiquant les conditions

qui déterminent la possibilité d'exécuter ou non une opération.

Nous avons préféré introduire une syntaxe différente de la substitution de sélection B afin de bien mettre en évidence que cette clause a des implications quant à l'interprétation ATL des systèmes.

3.1.2 SYSTEM

Dans ATL-B, la différenciation syntaxique de B et de ATL-B se fait au sein d'une entité appelée système. Cette entité qui reprend la syntaxe d'une machine B hormis les points suivants :

- le système abstrait ne comprend pas de clause INVARIANT, c'est-à-dire qu'il ne contient pas d'invariant propre mais son invariant est constitué des invariants des machines incluses que nous dénommons agents. En ce qui concerne les systèmes raffinés, cette clause INVARIANT ne peut servir (nous expliquerons pourquoi dans le chapitre 4) qu'à l'invariant de collage.
- deux nouvelles clauses (DYNAMIC CONSTRAINT et DYNAMIC PROPERTIES) apparaissent, qui permettent de spécifier des invariants dynamiques de notre système.

Clauses

Les nouvelles clauses apportée à B sont les suivantes :

- DYNAMIC CONSTRAINTS : reprenant une formule ATL spécifiant le comportement attendu du système.
- DYNAMIC PROPERTIES : reprenant une formule ATL spécifiant les propriétés du système, c'est-à-dire les hypothèses faites sur un certains nombres d'agents.

La distinction entre propriétés dynamiques et contraintes dynamiques du système sera abordée au chapitre suivant.

Formules ATL

Une formule ATL (en fait ATL^*) est une formule B à laquelle on ajoute les opérateurs $\Box$, $\Diamond$, $\bigcirc$et $<< ... >>$.

Ces formules correspondent donc effectivement à la définition classique d'une formule ATL (comme définie dans [AHK02]), et même en fait ATL*.

3.2 Transformations syntaxiques vers B

Il nous faut maintenant donner une transformation syntaxique nous permettant d'obtenir une spécification B à partir d'une spécification ATL-B.

Par rapport à notre exemple de la section 2.1, nous avons apporté la modification suivante : nous avons préféré, à la clause INCLUDE de l'exemple,

une copie des différents éléments. Ceci correspond à la clause INCLUDE mais offre l'avantage de ne pas lier la machine SYSTEM aux différents agents. On peut donc étudier chaque agent ainsi que le système en soi, en tant que spécification B autonome.

La transformation en question reprend les différents éléments nouveaux apportés à la syntaxe B pour obtenir ATL-B. en nous référant à la syntaxe donnée en Annexe, cela concerne donc les clauses suivantes :

- Système
- Système_raffinement
- Agent_abstrait
- Agent_Raffinement
- Agent_Implantation
- Clause_external_constraints
- Clause_external_variables
- Clause_actions
- Clause_agents
- Clause_dynamic_constraints
- Clause_dynamic_properties

L'idée est donc de transformer un texte ATL-B en un texte B afin de pouvoir soumettre ce dernier aux outils permettant la validation de systèmes B.

3.2.1 Agents

Pour chaque agent, nous devons créer 2 machines : l'une, portant le nom de l'agent, reprend le comportement de l'agent et décrit les variables. L'autre simule l'environnement en regroupant les variables externes de l'agent ainsi que les contraintes associées à ces variables (qui forment l'invariant de cette seconde machine B). La machine "agent" voit alors la machine "environnement", ceci étant effectué via la clause SEES de B, qui permet à la machine "agent" d'accéder à l'environnement.

Cette approche permet de voir, du point de vue de B, chaque agent indépendamment (ou presque, vu qu'il intéragit tout de même avec le système via la seconde machine).

Etant donné un agent A contenant les variables externes (clause EXTERNAL_VARIABLES) $V_0, ...V_n$ et les contraintes externes (clause EXTERNAL_CONSTRAINTS) $C_0, ..., V_m$ la machine ainsi créée sera la suivante :

```
MACHINE A_EXTERN
  VARIABLES
    V0,...,Vn
  INVARIANT
```

$C_0 \wedge \ldots \wedge C_m$

END

3.2.2 Fonction de transformation

Etant donné $P(G_B)$ et $P(G_{ATLB})$ l'ensemble des protophrases obtenues respectivement à partir des grammaires de B (G_B [Cle02]) et de ATL-B (G_B voir annexe 6.4), nous allons donc définir la fonction $F_{Transform} : P(G_B) \rightarrow P(G_{ATLB})$ donnant la transformation nous permettant d'obtenir un composant B à partir d'un composant ATL-B.

Nous ne définissons ici que les transformations, pour tous les terminaux non répertoriés ci-dessous, nous aurons $F_{Transform}(x) = x$.

Pour une partie de texte non spécifiée ci-après, constituée d'une suite de Clause C_1,..., C_N, nous aurons la formule suivante : $F_{Transform}(C_1 ... C_N)) = F_{Transform}(C_1) ... F_{Transform}(C_N)$

Après application et afin de respecter la syntaxe, il conviendra de regrouper les clauses identiques d'une même machine sous la même clause.

Définitions préliminaires

Var(A) représente le contenu de la clause VARIABLES d'un agent A.

Inv(A) représente la clause INVARIANT d'un agent A.

Op(A) représente le contenu de la clause OPERATIONS d'un agent A.

Action(A) représente le contenu de la clause ACTION d'un agent A

Par la construction $\cup_{i=1}^{N} Var(A_i)$, nous entendons, dans le cadre d'un système S étudié, la liste des variables des agents du système.

Par la construction $\wedge_{i=1}^{N} Inv(A_i)$, nous entendons, dans le cadre d'un système S étudié, la conjonction des invariants agents du système.

Par la construction $\|_{i=1}^{N} Op_i$, nous entendons, dans le cadre d'un système S étudié, la composition des opérations Op_i .

Par la construction $[]_{i=1}^{N} P_i \implies Op_i$, nous entendons le contenu de la clause action composée des choix Op_i sous condition que la propriété P_i soit vraie.

Par la construction $SELECT_{i=1}^{N}\ P_i\ THEN\ Op_i$, nous entendons le contenu de la clause B de sélection

$$
\begin{array}{l}
SELECT\ P_1\ THEN\ Op_1 \\
\quad WHEN\ P_2\ THEN\ Op_2 \\
\quad \ldots \\
\quad WHEN\ P_N\ THENOp_N
\end{array}
$$

Système_abstrait

$$F_{Transform}\begin{pmatrix} "SYSTEM" En-tete \\ \quad Specification\_Systeme \\ "END" \end{pmatrix}$$

=

$$\begin{array}{l} "MACHINE" En-tete \\ \quad F_{Transform}(Specification\_Systeme) \\ "VARIABLES" \\ \quad \cup_{i=1}^{N} Var(A_i) \\ "INVARIANT" \\ \quad \wedge_{i=1}^{N} Inv(A_i) \\ "OPERATIONS" \\ \quad \wedge i = 1^{N} Op(A_i) \\ \quad "Machine\_action" \equiv \| \ i = 1^{N} Action(A_i) \\ "END" \end{array}$$

Système_raffinement

$$F_{Transform}\begin{pmatrix} "SYSTEM" En-tete \\ \quad Clause\_refines \\ \quad Specification\_Systeme \\ "END" \end{pmatrix}$$

=

$$\begin{array}{l} "MACHINE" En-tete \\ \quad Clause\_refines \\ \quad F_{Transform}(Specification\_Systeme) \\ "END" \end{array}$$

Agent_abstrait

$$F_{Transform}\begin{pmatrix} "AGENT" "Nom\_agent[Parametres]" \\ \quad Specification\_Agent \\ "END" \end{pmatrix}$$

=

$$\begin{array}{l} "MACHINE" Nom\_agent[Parametres] \\ \quad SEES Nom\_agent" \_EXTERN" \\ \quad F_{Transform}(Specification\_Agent) \\ "END" \end{array}$$

Agent_Raffinement

$$F_{Transform}\left(\begin{array}{l}"AGENT"Nom\_agent[Parametres]\\ \quad Clause\_refines\\ \quad Specification\_Agent\\ "END"\end{array}\right)$$

=

$$\begin{array}{l}"MACHINE"Nom\_agent[Parametres]\\ \quad Clause\_refines\\ \quad SEESNom\_agent"\_EXTERN"\\ \quad F_{Transform}(Specification\_Agent)\\ "END"\end{array}$$

Agent_Implantation

$$F_{Transform}\left(\begin{array}{l}"AGENT"Nom\_agent[Parametres]\\ \quad Clause\_refines\\ \quad Specification\_Agent\\ "END"\end{array}\right)$$

=

$$\begin{array}{l}"MACHINE"Nom\_agent[Parametres]\\ \quad Clause\_refines\\ \quad SEESNom\_agent"\_EXTERN"\\ \quad F_{Transform}(Specification\_Agent)\\ "END"\end{array}$$

Clause_external_constraints

$$F_{Transform}\,(Clause\_external\_constraints) = ""$$

Clause_external_variables

$$F_{Transform}\,(Clause\_external\_variables) = ""$$

Clause_actions

$$F_{Transform}\,("ACTIONS"[]_{i=1}^{N}P_i \implies Op_i)$$
$$= "OPERATIONS"Machine\_action" \equiv "SELECT_{i=1}^{N}P_i"THEN"Op_i$$

Une proposition P_i inexistante est en fait équivalente à TRUE et donnera donc lieu à une construction ... WHEN TRUE THEN ...

Clause_dynamic_constraints

$$F_{Transform}\,(Clause\_dynamic\_constratints) = ""$$

En effet, cette clause n'a pas d'équivalent en B et n'est pas transformée.

Clause_dynamic_properties

De la même façon, nous avons :

$$F_{Transform}\,(Clause\_dynamic\_properties) = ""$$

Chapitre 4

Sémantique d'un système ATL-B

Nous allons donner ici la sémantique de ATL-B telle que nous l'envisageons. Comme nous l'avons déjà souligné, nous opérerons, plutôt qu'en définissant une nouvelle sémantique, en mettant en évidence la dualité du système ATL-B. Ce dernier sera donc vu à la fois comme un système B et un système ATL, selon le point de vue adopté.

4.1 Sémantique statique

Etant donné que nous sommes partis de la modélisation B, nous avons adopté les règles de typage, de visibilité et de résolution de portée utilisées dans B [Cle02].

Puisque nous avons fait une correspondance avec B via la transformation syntaxique les règles sont aisées à transposer.

Les nouvelles règles de sémantique statique ne sont dès lors plus à définir que pour les clauses DYNAMIC CONSTRAINTS et DYNAMIC PROPERTIES qui sont les seuls nouveaux éléments concernés.

4.1.1 Typage

B définit la vérification de type comme une condition préalable à la preuve d'une spécification.

Les types sont, soit des types de base, soit des types construits par produit cartésien, des ensembles de partitions ou de records. Un type de base peut être défini comme type abstrait via la clause SETS d'une machine.

Chaque donnée utilisée dans un prédicat ou une substitution doit faire l'objet d'un typage. Ce typage est effectué via des prédicats ou substistutions spécifiques dites de typage. Ces prédicats doivent naturellement être présents dans la spécification B avant l'utilisation de la donnée.

Les variables sont typées via l'invariant, en général par un prédicat d'appartenance(ain). Les données locales à une opération le seront habituellement via la précondition ou l'utilisation de substitution ANY.

La vérification du type doit donc s'assurer qu'il n'y a pas d'incompatibilité dans la spécification, les règles étant données dans [Cle02].

Les variables utilisées dans les clauses DYNAMIC CONSTRAINTS et DYNAMIC PROPERTIES devront être typées de la même manière : soit via l'invariant pour les variables du système, soit via un prédicat de typage au sein même de la formule.

Les variables définies dans la clause EXTERNAL VARIABLES devront être typées via la clause EXTERNAL CONSTRAINTS. Cela donnera donc un typage classique B Pour la machine MACHINE_EXTERN obtenue après transformation.

4.1.2 Résolution de portée et Visibilité

Les notions de résolution de portée et de visibilité adoptées seront les mêmes que pour la méthode B. Le lecteur se reportera donc à [Cle02] pour une définition complète.

En ce qui concerne les nouvelles clauses DYNAMIC PROPERTIES et DYNAMIC CONSTRAINTS, les éléments visibles seront les variables du système, les paramètres des différentes machines ainsi que les constantes des différents agents du système.

4.2 Sémantique B

Etant donné la transformation définie de ATL-B vers B, nous pourrons donc envisager notre système ATL-B.

Les règles de preuves sont donc les mêmes que pour B et sont données dans [Abr96], nous en avons donné les éléments essentiels à la section 1.1.

Certains éléments doivent cependant, nous semble-t-il, être mis en évidence.

4.2.1 Composition multiple de substitution

Dans la transformation décrite dans le chapitre précédent, nous avons vu comment interpréter un système comme une machine B exécutant en parallèle (via l'opérateur ||de composition multiple) les actions des différents agents.

Or, une substitution simultanée ne peut se faire qu'entre des substitutions ne modifiant pas les mêmes variables.

Ceci ne pose en fait pas de problème car les agents ont leurs variables propres et ne peuvent pas modifier les variables des autres agents (même si

elles sont accessibles en lecture grace à la clause EXTERNAL_VARIABLES).

Une simple analyse B nous montrera en effet l'erreur étant donné l'emploi (après transformation en B) de la clause SEES qui ne permet l'usage des variables de la machine vues qu'en lecture seule.

Pour peu que les agents interagissant soient corrects, nous pouvons donc employer cette substitution.

4.2.2 Incidence sur l'invariant d'un système raffiné

Etant donné ce que nous venons d'expliquer, nous devons alors justifier la présence de la clause INVARIANT dans un système raffiné : elle sert à décrire l'invariant de collage entre le système abstrait et le système raffiné et le système abstrait, c'est-à-dire le lien qui permet de faire correspondre les états et contraintes dynamiques du système abstrait et du système raffiné.

4.3 BCGS

Dans ce chapitre, nous allons donner une définition d'un BCGS (B Concurrent Game Structure) définissant le cadre sémantique d'un système ATL-B. Cette définition repose sur l'extension d'un CGS donné par [AHK02].

Pour définir le CGS associé à une spécification ATL-B, nous nous sommes inspirés de [Mas01] qui a défini des correspondances avec les systèmes de transactions étiquetés utilisés dans le cadre de LTL.

4.3.1 Définitions préalables

Dans [Mas01], l'auteur définit, pour un ensemble X= $\{x_1,...,x_n\}$ de variables de domaine respectif $D_1,...,D_n$, l'ensemble suivant de propositions atomiques :

$$PA_i = x_i = v | v \in D_i$$

Cet ensemble reprend donc l'ensemble des propositions indiquant les valuations possibles de la variable x_i.

De là , on peut définir l'ensemble de propositions suivant :

$$\mathbb{D}_X = \wedge_{i=1}^{n} x_i = v | x_i = v \in PA_i$$

Cet ensemble reprend ce que nous appellerons les propositions désignant les valuations possibles des variables de X.

4.3.2 BCGS

Un BCGS est un t-uple $< S, Q, \Pi, \pi, d, \delta >$t.q :

- S soit l'ensemble des agents B du système conforme à la sémantique B classique, chaque agent comprenant en fait 2 machines B, comme nous l'avons vu au chapitre précédent.
- Q l'ensemble des états du système, ces états seront donnés par l'ensemble des valuations possibles du sytème, c'est-à-dire de toutes les valeurs possibles de chaque variable du système.
- Π l'ensemble des propositions observables, qui sera ici l'ensemble $\mathbb{D}_V$ défini précédemment ou V reprend l'ensemble des variables du système.
- π la fonction d'étiquetage (labeling function) qui, à chaque état q, définit l'ensemble $\pi(q)$ des propositions (incluses dans Π) vraies à l'état q.
- d, une fonction qui, pour chaque état q du système et chaque agent A, définit un nombre $d_A(q) \in \mathbb{N}$ de mouvements possibles. Ce nombre est donc, pour un agent A, le nombre de formules vraies à droite des $\Longrightarrow$ (appelées gardes) dans la clause ACTIONS (Une formule vraie étant assimilée à vraie)
- δ une fonction qui, à chaque état q et chaque vecteur $< j_1, ..., j_k >$, associe un état $\delta(q, j_1, ..., j_k) \in Q$ qui représente l'état d'arrivée du système après le mouvement. δ est la fonction de transition (transition function). Cette fonction est directement obtenue par application de la composition des substitutions des choix effectués par chacun des agents.

4.3.3 Problèmatique des opérations non-déterministes

Un problème reste à soulever si nous appliquons la modélisation sous forme de structures de jeux d'un système ATL-B : certaines substitutions B sont non-déterministe, comme par exemple l'opérateur $:\in$, les substitutions CHOICE ou ANY.

Dans notre CGS, si un des choix de la clause action est une substitution non-déterministes, cela reviendrait à avoir, pour ce choix, une fonction δ qui mènerait à 2 états distincts et ne serait dès lors plus une fonction.

Si nous considérons un système composé de l'agent du tableau 4.1 et que nous envisageons les transitions possibles, nous obtenons le schéma de la figure 4.1

Nous avons une transition qui mène vers 2 choix possibles (true ou false) représentant l'ensemble des possibilités dans le domaine défini par BOOL.

Nous voyons plusieurs manières de résoudre ce problème :

TAB. 4.1 – Agent avec assignation indéterministe

```
AGENT MyAgent
  VARIABLES
   v
  INVARIANT
   v ∈ BOOL
  INITIALISATION
   v :∈ BOOL
  OPERATION
   DoAction ≡ v :∈ BOOL
  ACTION
   DoAction
END
```

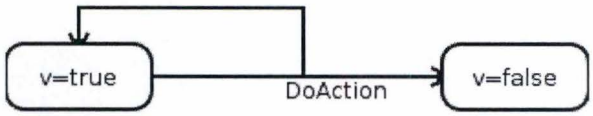

FIG. 4.1 – CGS pour un système indéterministe

Supprimer du langage ATL-B les substitutions indéterministes

En supprimant du langage ATL-B les substitutions indéterministes, on supprime de facto notre problème mais on ôte à la méthode B toute l'expressivité de certains systèmes abstraits. En effet, les substitutions non-déterministes permettent de reporter à l'implémentation certains choix qui n'apportent rien à la spécification abstraite d'un problème.

On doit donc changer la spécification de notre agent exemple par celui du tableau 4.2, qui nous donne les transitions possibles de la figure 4.2 .

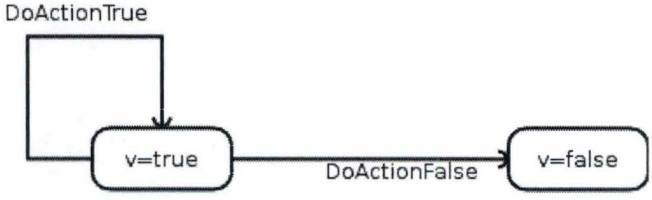

FIG. 4.2 – CGS sans indéterminisme

Nous avons donc modifié notre spécification en vue d'obtenir un sytème dont chaque action est déterministe.

TAB. 4.2 – Agent sans assignation indéterministe

```
AGENT MyAgent
  VARIABLES
   v
  INVARIANT
   v ∈ BOOL
  INITIALISATION
   v :∈ BOOL
  OPERATION
   DoActionTrue ≡ v := TRUE
   DoActionFalse ≡ v:= FALSE
  ACTION
   DoActionTrue
   [] DoActionFalse
END
```

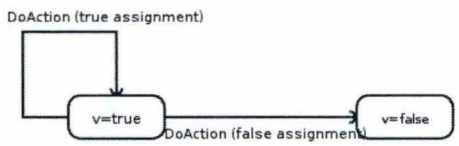

FIG. 4.3 – CGS avec choix supplémentaires

Augmenter le nombre de choix

Considérer chaque possibilité d'indétermination comme un choix supplémentaire pour l'agent. Dans ce cas là, un agent posséderait un choix supplémentaire, pour chaque valuation possible d'un même choix, qui ferait de δ de nouveau une fonction et rendrait à notre CGS sa validité.

La transition est alors illustrée dans la figure 4.3.

Il faut bien comprendre que cette façon de faire ne change rien à la spécification de départ, cela influe juste sur la manière de construire le CGS. On prend chaque valeur possible après opération et on en fait un nouveau choix pour l'agent, dans l'automate.

Ajouter un agent dual

Ajout d'un agent représentant l'indéterminisme. Chaque agent aurait un agent dual ; ces agent duaux, joueraient après les autres et, en fonction du

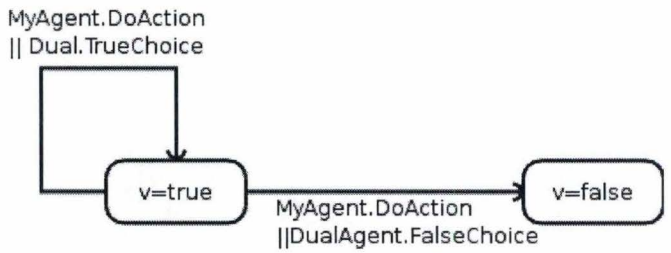

FIG. 4.4 – CGS avec agent dual

choix de l'agent d'origine, pourraient opérer un choix sur l'ensemble des valeurs possibles.

La nouvelle transition se retrouve dans la figure 4.4.

Dans ce cas-ci, comme pour le précédent, il ne s'agit pas de changer le texte de la spécification, mais la construction du CGS en y ajoutant les agents duaux.

Verdict

Les 2 dernières solutions, car permettant de ne pas amputer la méthode d'aspects importants, sont à ce stade-ci équivalentes. Nous verrons dans le chapitre suivant sur le raffinement qu'elles auront une incidence, et nous privilégierons la dernière solution.

4.4 Compatibilité entre agents

Afin de rendre notre système cohérent, quand on l'envisage dans son ensemble, nous devons aussi apporter une preuve de compatibilité entre les agents.

Chaque agent reprenant via les clause EXTERNAL VARIABLES et EXTERNAL CONSTRAINTS une vue propre du système, il faudra prouver que cette vue est cohérente par rapport au système effectif.

Nous devrons donc prouver, en considérant les agents $A_0, ..., A_n$, pour chaque agent A_k, que $\wedge_{i=1 i \neq k}^{N} INV(A_i) \implies EXTERNAL\_CONSTRAINTS(A_k)$.

4.5 Contraintes et propriétés dynamiques

Les contraintes et propriétés dynamiques, spécifiées respectivement dans les clauses DYNAMIC CONSTRAINTS et DYNAMIC PROPERTIES, permettent de spécifier ce que nous voulons du système au cours du temps.

Il s'agira donc, lors du model checking du système ATL-B, de montrer l'existence de stratégies (cfr. section 1.2.3) afin de vérifier la faisabilité de ces formules.

Nous verrons, dans le prochain chapitre, se rapportant au raffinement de système ATL-B, la justification principale de cette distinction.

Mais une raison d'être nous apparaît quand même importante à souligner ici : il s'agit de l'expressivité, qui doit être comprise de cette façon :

- une propriété dynamique du système doit être vue comme quelque chose qui nous est donné, éventuellement liée à des phénomènes physiques (par exemple, comme nous l'illustrons dans le chapitre 6, le fait qu'un canal de transmission laisse passer des paquets de temps en temps) ou autre sur une partie du système que nous ne contrôlons pas complètement. Dans le cas de propriétés dynamiques, les stratégies (bien que indispensables pour la vérification du système) ne seront pas intéressantes car elles ne reflètent pas une chose qui sera implémentée.
- une contrainte dynamique doit être vue comme une directive que le système devra implémenter : dans ce cas, la stratégie employée sera importante car elle sera implémentée.

Chapitre 5

Raffinement ATL-B

Le raffinement, processus clé de la méthode B par lequel on détaille un système en vue d'obtenir par étapes successives un élément programmable.

De par le nature ambivalente de ATL-B, nous discernerons deux types de raffinement :

- le raffinement d'agents plus proches d'un raffinement classique de B.
- le raffinement d'un système qui permet de raffiner un système dans sa globalité et qui se base sur le raffinement ATL.
- le raffinement des contraintes et/ou propriétés d'un système afin de définir l'impact sur les contraintes du système de certains choix quant à l'implémentation.

5.1 Raffinement d'agents

Le raffinement d'agents garde la sémantique classique de B telle que définie dans [Abr96] et mentionné dans la section 1.1.3.

Ce mécanisme permet, dans toute spécification, d'opérer le remplacement d'un agent par l'autre, c'est donc une relation de type "est un".

On envisagera donc pour le raffinement d'un agent la machine B sous-jacente pour organiser les preuves.

Pour rappel (cfr. section 1.1.3) , ceci devra être constitué des preuves suivantes :

- la précondition de chacune des opérations de l'agent raffiné est, au plus, aussi forte que celle de l'agent abstrait.
- la post-condition de l'opération raffinée est incluse dans la post-condition de la machine abstraite.

Pour ce raffinement, nous ne prendrons pas en considération l'opération issue de la clause ACTION des agents considérés, nous expliquerons pourquoi.

5.2 Raffinement des rôles d'un système

Le raffinement des rôles d'un système envisage le raffinement dans sa globalité, intégrant les différents rôles ([RS02]).

Pour traiter le raffinement, nous nous sommes basés principalement sur [AHKV98] qui définit les fondements de la relation de raffinement et [RS02] qui l'élargit en considérant une notion de rôle avec raffinement par raffinement d'agents et abstraction de l'environnement.

En partant des définitions tirées de [RS02], on va, dans un premier temps, définir un raffinement en termes de différents éléments :

- le raffinement de la signature du système
- l'extension au raffinement d'un système.
- le raffinement des contraintes et propriétés dynamiques

5.2.1 Raffinement de signature d'un CGS

Le raffinement de signature d'un système $< S, Q, \Pi, \pi, d, \delta >$vers un système $< S\prime, Q\prime, \Pi\prime, \pi\prime, d\prime, \delta\prime >$tel qu'étendu à partir de la définition dans [RS02] se compose des éléments suivants :

- une relation f $\in S \times S\prime$ qui définit la correspondance entre agents raffinés et agents abstraits.
- une relation g qui lie les variables du système abstrait et les variables du système concret tout en respectant f.

f sera donné, de façon classique en B par l'invariant de collage, tandis que g par l'élément ROLE OF (*Clause_agents_with_roles*) de la clause AGENT du système.

5.2.2 Extension du raffinement d'un système

De la même manière et toujours selon [RS02], on définit l'extension du raffinement d'un système par le triplet suivant :

- une relation H $\in$Q $\times$Q' qui fait correspondre les états de Σ aux états de Σ'
- une fonction C qui, pour tout choix des protagonistes dans Σ' fait correspondre un choix dans Σ (pour des états satisfaisant H)
- une fonction D qui, pour tout choix des antagonistes dans Σ fait correspondre un choix dans Σ' (pour des états satisfaisant H)

5.2.3 Types de raffinement

Nous avons défini précédemment les relations C et D qui donnent les correspondances entre antagonistes et protagonistes.

La clause *Clause_agents_with_roles* de la syntaxe permet de définir comment on considère les différents agents :

- la clause "EXTENDS ROLE OF" définit que l'agent est considéré comme un antagoniste. Ses choix seront donc préservés dans le système raffiné. Il sera donc répertorié dans la fonction D. Nous appellerons ce type de raffinement le raffinement par extension du rôle.
- la clause "IMPLEMENTS ROLE OF" définit que l'agent dans le système raffiné est considéré comme un protagoniste pour la définition qui précède. Il sera donc répertorié dans la fonction C. Nous appellerons ce type de raffinement le raffinement par implémentation.
- la clause "IS ROLE OF" définit que nous ne sommes dans aucun des 2 cas, en fait l'agent du système raffiné a le même comportement (en termes de choix que dans le système abstrait.)

Nous verrons comment utiliser ces différents types de raffinements par rôle.

Si nous envisageons les agents "duaux" évoqués à la section 4.3.3, nous le ferons toujours dans un raffinement par implémentation pour la raison que cela correspond au raffinement B classique, qui lève l'indéterminisme au fur et à mesure des raffinements successifs.

A remarquer que le raffinement d'événement envisagé dans Event-B (cfr section 1.1.6) est un raffinement par extension, car le raffinement de l'événement skip correspond à une extension des choix de l'agent.

5.3 Raffinement et contraintes dynamiques

En vertu des propriétés du raffinement par extension défini dans [RS02], le raffinement par extension aura la particularité de conserver les contraintes dynamiques faisant intervenir uniquement les éléments raffinés par extension d'un système abstrait dans le système abstrait.

Il ne sera pas donc nécessaire, si l'on montre la correction d'un raffinement par extension , de vérifier les contraintes dynamiques dans le système raffiné. Une application de la relation D donnera la correspondance des stratégies des différents agents.

Un système ne pourra être considéré comme implémenté que si tous les agents du système intervenant dans la clause des contraintes dynamiques sont des agents implémentés (c'est-à-dire qu'ils ne contiennent plus d'indéterminisme, et peuvent être codés).

C'est ici que nous voyons l'autre justification de la distinction entre propriétés dynamiques et contraintes dynamiques : dans le système final,

les propriétés dynamiques porteront sur les agents qui ne sont pas des implémentations, et font donc partie de la spécification de l'environnement de la partie implémentée.

5.4 Intégration du raffinement dans le processus de développement

Le raffinement est un moyen, de donner différentes lectures du système. Dans un système B classique, on définit le tout pour obtenir in fine une solution codable dans un langage classique.

Pour ATL-B, nous nous positionnons dans un système ou seuls quelques éléments (ceux intervenant dans les contraintes dynamiques) seront codables. Nous allons montrer ici comment nous envisageons de définir un développement ATL-B.

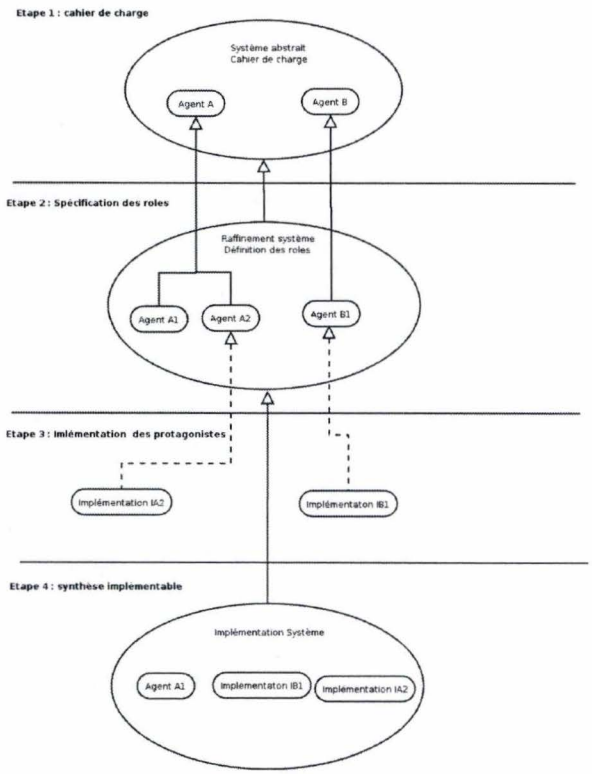

FIG. 5.1 – Processus de raffinement

Nous avons dégagé 4 étapes qui représentent, selon nous, les différentes étapes à appliquer dans le cadre ATL-B, et plus généralement la spécification formelle de systèmes ouverts. La figure 5.1 reprend les différentes étapes que

nous envisageons.

Ces étapes sont les suivantes :

- cahier de charge : cette étape correspond à la spécification du problème en interaction avec l'environnement. Il ne donnera en règle générale pas le détail du fonctionnement du système. Ce sera plus une spécification du type "use case" reprenant, de manière formelle, les exigences du système.
- spécification des rôles : cette étape consiste en le fait de décrire l'interaction interne du système que nous envisageons. Nous donnerons ici les propriétés dynamiques du système qui constitueront les hypothèses sur différents dispositifs implémentés à l'extérieur. On pourra définir plusieurs niveaux de raffinement en vue de dégager tous les agents de manière incrémentale.
- implémentation des agents : cette étape consiste en l'implémentation des agents que nous avons identifiés comme à implémenter. Cela se fait via un raffinement (ou une succession de raffinements) de type B classique.
- synthèse implémentable : les éléments implémentés dans la phase précédente sont réintégrés dans une implémentation de système. Il faudra donc s'assurer que les différents agents implémentent bien les stratégies qui auront été obtenues des étapes précédentes.

Chapitre 6

Une étude de cas : Spécification d'un réseau

Ce que nous voulons faire, dans cette partie, c'est éprouver ATL-B à l'aide d'un cas assez simple : un réseau.

Nous allons tenter de définir l'essence d'un réseau (sa spécification abstraite) pour en dégager certaines propriétés qui nous amènerons à raffiner ces derniers et à définir les agents interagissants.

L'exemple ci-dessous illustrera les 2 premières étapes de la section 5.4. La première spécification donnera le cahier de charge et les 2 raffinements détailleront les agents en jeu.

6.1 Spécification du problème abstrait

Le problème basique va être le suivant. Nous voulons un dispositif capable de transférer un message d'un point à un autre.

Nous allons, dans cette partie donner une vue de type "use-case" d'un réseau. Il s'agit d'une spécification qui ferait donc partie du cahier des charges en ce qui concerne le cycle de développement.

6.1.1 Description du problème

Pour modéliser le problème , nous allons procéder en décrivant la transmission d'un message d'un envoyeur vers le destinataire, comme représenté sur la figure 6.1. L'envoyeur prend l'initiative de choisir un message à envoyer, à charge du réseau de le transmettre.

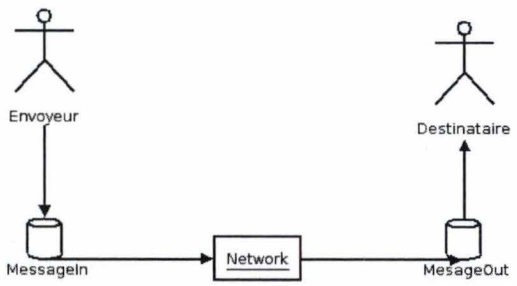

FIG. 6.1 – Réseau : spécification abstraite

Etant donné que seul l'envoyeur agit effectivement sur le système, le destinataire se contentant de lire le message et ne changeant donc en rien dans l'état proprement dit du système, nous ne modéliserons que l'envoyeur sous la forme de l'agent *User*.

Afin de simplifier le problème, nous ne considérerons ici que l'envoi d'un message.

6.1.2 Spécification ATL-B

Objets Communs

Le tableau 6.1 reprend, au sein d'une machine B classique, les objets communs aux différents agents

TAB. 6.1 – Machine reprenant les objets communs

```
MACHINE Common_Sets
  SETS
  MESSAGE
CONSTANTS
  EmptyMessage
PROPERTIES
EmptyMessage ∈ MESSAGE
card(MESSAGE) > 1
END
```

Il s'agit donc ici de donner l'ensemble des messages que l'envoyeur peut envoyer, repris sous la dénomination *MESSAGE*.

Cet ensemble contient un message spécial *EmptyMessage* qui servira à identifier l'absence de message dans la spécification.

Nous ajoutons la propriété *card(MESSAGE)* > *1* car, dans le cas où le seul élément de *MESSAGE* serait *MESSAGE*, la spécification n'aurait plus aucun sens, puisque l'utilisateur n'aurait aucune possibilité de choisir un message autre que *EmptyMessage*.

Agent réseau

Le tableau 6.2 reprend la spécification de l'agent réseau qui a pour charge de transférer (via l'opération *send*) le message de MessageIn vers MessageOut.

TAB. 6.2 – Réseau V0 - Agent réseau

```
AGENT NetworkAgent
  INCLUDE
    Common_Sets
  VARIABLES
    MessageOut
  EXTERNAL_VARIABLES
    MessageIn
  EXTERNAL_CONSTRAINTS
   MessageIn ∈ MESSAGE
  INVARIANT
    MessageOut ∈ MESSAGE
  INITIALIZATION
   MessageOut := EmptyMessage
  OPERATIONS
    Send ≡
     MessageOut := MessageIn
  ACTION
    skip
    [] Messagein ≠ EmptyMessage ⟹ Send
```

Il peut, à chaque tour, décider d'envoyer un message ou bien de ne rien faire (via le choix de substitution *skip*).

Agent utilisateur

Le tableau 6.3 reprend la spécification de l'utilisateur.Ce dernier peut, à chaque tour, soit envoyer un message *Emit*), ou ne rien faire (*skip*).

TAB. 6.3 – Réseau V0 - Utilisateur

```
AGENT User
  INCLUDE
    Common_Sets
  VARIABLES
    MessageIn
  INVARIANT
    MessageIn ∈ MESSAGE
INITILIZATION
    MessageIn := EmptyMessage
  OPERATION
    Emit ≡
       ANY m WHERE m ∈ MESSAGE
        ∧ MessageIn = EmptyMessage THEN
        MessageIn := m
  ACTION
    skip
    [] MessageIn = EmptyMessage ⟹ Send
```

Le système

Le tableau 6.4 reprend la spécification du système. Elle indique les différents agents qui interviennent dans l'interaction de l'envoi du message.

TAB. 6.4 – Réseau V0 - Système

```
SYSTEM NetworkSystem
  AGENTS
    NetworkAgent
    User
  DYNAMIC CONSTRAINTS
   << NetworkAgent >> ◇ MessageIn = MessageOut
```

La contrainte du système $<< NetworkAgent >> \Diamond MessageIn = MessageOut$ spécifie que nous souhaitons que le système puisse faire parvenir un

message de MessageIn vers MessageOut.

6.1.3 BCGS associé

Notre BCGS $< S, Q, \Pi, \pi, d, \delta >$, illustré par la figure 6.2 se compose donc des éléments suivants :

- S

$$S = \{User, NetworkAgent\}$$

- Q

$$Q = \{Transmitting_m, Transmitted_m | m \in message\}$$

L'état initial étant $Transmitted_{EmptyMessage}$

- Π

$$\Pi = \{MessageIn = m \wedge MessageOut = n | m, n \in MESSAGE\}$$

- π

$$\pi = \begin{array}{l} \{\bigcup_{m \in MESSAGE} \{(Transmitted_m, MessageIn = m \wedge MessageOut = m), \\ (Transmitting_m, MessageIn = m \wedge MessageOut = EmptyMessage)\}\} \end{array}$$

- D

$$d = \begin{array}{l} \{(Transmitted_{EmptyMessage}, User, 2), (Transmitted_{EmptyMessage}, NetworkAgent, 1)\} \\ \bigcup \{(Transmitting_m, USer, 1), (Transmitting_m, NetworkAgent, 2) \\ \quad | m \in MESSAGE \backslash \{EmptyMessage\}\} \end{array}$$

- δ qui définit les transitions, résultant du choix par les différents agents de leurs actions.

Nous voyons déjà que la taille du BCGS, pour un problème somme toute assez simple, peut se trouver énorme, pour peu que la taille de l'ensemble MESSAGE soit grand.

Si nous définissons l'ensemble MESSAGE comme étant l'ensemble {EmptyMessage,m}, nous obtenons la représentation graphique de la figure 6.2.

Considérer l'ensemble MESSAGE de cette façon ne pose en fait pas vraiment de problème, c'est-à-dire que cela n'enlève pas d'expressivité à notre spécification car :

- le traitement du message ne dépend pas de sa forme, de ses propriétés intrinsèques, si ce n'est pour EmptySet
- nous ne traitons qu'un message.

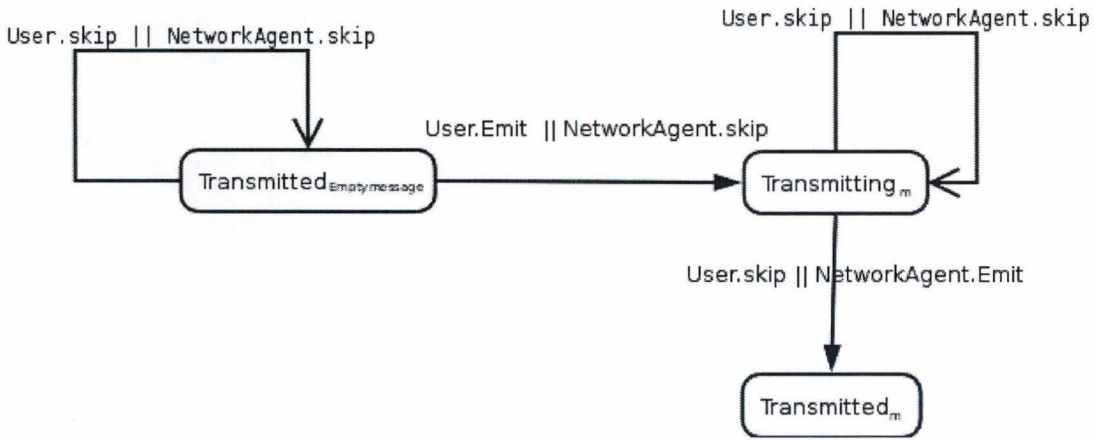

FIG. 6.2 – Représentation graphique du CGS du système Network

6.1.4 Preuves

Préservation de l'invariant

Il est assez trivial de démontrer que les opérations des différents agents préservent leurs invariants respectifs.

Notre système est donc cohérent quant à sa spécification statique(=B).

Contrainte dynamique

Pour pouvoir prouver la contrainte dynamique, nous devons trouver un ensemble de stratégies, pour le protagoniste qu'est l'agent *NetworkAgent* ayant comme résultante le fait que le message arrive au destinataire.

L'ensemble en question est immédiat : f telle que :

$$f(Transmitted_{EmptyMessage}...Transmiting_m)(NetworkAgent) = Send$$
$$f(Transmitted_{EmptyMessage}...Transmited_m)(NetworkAgent) = skip$$
$$f(Transmitted_{EmptyMessage}...Transmitted_{EmptyMessage})(NetworkAgent) = skip$$

avec m ∈MESSAGE \{EmptyMessage}

Ceci montre donc que notre système est cohérent quant à son comportement dynamique.

6.2 Premier raffinement : identification des agents

Dans cette deuxième version du système, nous allons tenter de décrire le fonctionnement interne de notre système. Le but ici est de mettre en

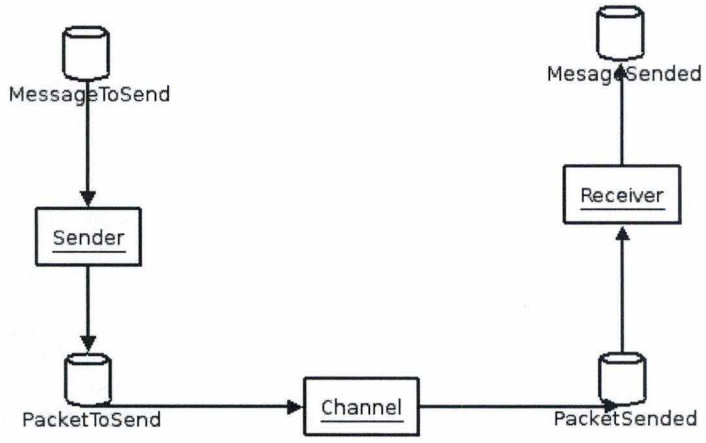

FIG. 6.3 – Réseau : Identification des agents

évidence les différents agents, que nous envisageons sous l'angle de leur rôle, intéragissant au sein du système.

6.2.1 Description du système

Nous voulons donc définir le fonctionnement d'un réseau dans sa plus simple expression. Comme nous le voyons sur la figure 6.3, un réseau se compose de 3 agents :

- l'émetteur qui envoie des données sur le canal en décomposant un message en paquets qu'il "dépose" dans *PacketsToTransmit*
- le récepteur qui attend des données et recompose le message d'origine
- le canal de communication qui permet la transmission des données de l'émetteur vers le récepteur via les ensembles *PacketsToTransmit* et *PacketsTransmitted*.

Nous allons maintenant mettre en évidence le découpage du message. Dans un réseau classique, le message est découpé en paquets avant d'être envoyé sur le canal de transmission. C'est ce qu'illustre la figure 6.4

On peut modéliser un système tel que décrit dans le schéma 6.3, on peut le faire à l'aide de la description ATL-B suivante.

6.2.2 Spécification ATL-B

Objets Communs

Le tableau 6.5 reprend les éléments communs aux différents agents.

On retrouve, en plus des définitions des objets communs du système abstrait les éléments suivants :

- PACKET représentant l'ensemble des paquets

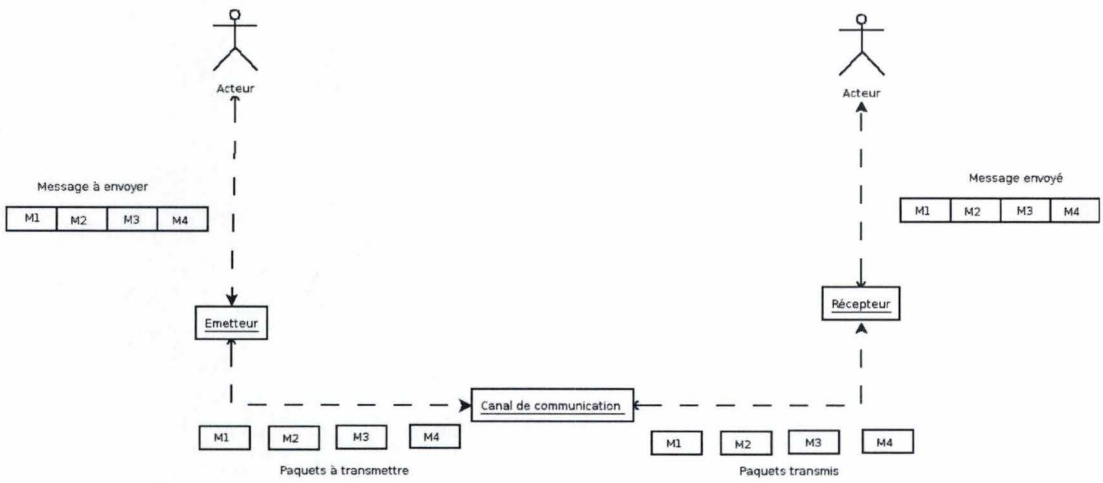

FIG. 6.4 – Flux d'un réseau

TAB. 6.5 – Machine reprenant les objets communs

```
MACHINE Common_Elements
  SETS
  MESSAGE, PACKET
CONSTANTS
 DecompositionFunction
 EmptyMessage
PROPERTIES
EmptyMessage ∈ MESSAGE
card(MESSAGE) > 1
DecompositionFunction ∈ MESSAGE >+> PACKETS
END
```

- la fonction DecompositionFunction qui détermine le découpage d'un message en ses différents paquets.

Agent émetteur

Le tableau 6.6 reprend la spécification de l'agent émetteur qui est l'agent en charge de décomposer le message entrant et de le communiquer au canal de transmission via l'ensemble de paquets *PacketstoTransmit* représentant, en quelque sorte, le buffer d'entrée du canal de transmission.

L'agent émetteur peut, soit transmettre un paquet au canal de transmission(*sendMessage*), soit ne rien faire (*skip*).

TAB. 6.6 – Réseau V1 - Emetteur

```
AGENT Sender (DecompositionFunction)
  INCLUDE
    Common_Elements
  VARIABLES
    PacketsToTransmit
  EXTERNAL_VARIABLES
    MessageToSend
  EXTERNAL_CONSTRAINTS
    MessageToSend ∈ MESSAGE
  INVARIANT
   PacketsToTransmit ∈ P(PACKET)
  INITIALIZATION
    PacketsToTransmit = ∅
  OPERATIONS
    sendPacket ≡
    ANY p WHERE p ∈ DecompositionFunction(messageToSend)
          ∧ p ∉ PacketsToTransmit
    THEN
      PacketsToTransmit := PacketsToTransmit ∪ {p}
  ACTION
  skip
  [] ∃ p . (p ∈ DecompositionFunction(messageToSend)
          ∧ p ∉ PacketsToTransmit)
          ⟹ sendMessage
```

Agent récepteur

Le tableau 6.7 reprend la spécification de l'agent récepteur qui a pour fonction de recomposer les messages transmis via le canal de transmission.

L'expression " DecompositionFunction[m]" correspond à l'image de m par DecompositionFunction, en d'autres termes, cela représente l'ensemble des paquets composants m.

L'agent récepteur peut, soit recomposer le message transmis (*Receive-Message*), soit ne rien faire(*skip*).

Agent Canal de communication

Le tableau 6.8 reprend la spécification de l'agent canal de communication dont le rôle est de transmettre les paquets de l'émetteur vers le récepteur,

TAB. 6.7 – Réseau V1 - Récepteur

```
AGENT Receiver
  INCLUDE
    Common_Elements
  VARIABLES
    MessageReceived
  EXTERNAL_VARIABLES
    PacketsTransmitted
  EXTERNAL_CONSTRAINTS
    PacketsTransmitted ∈ ℙ(PACKET)
  INVARIANT
    MessageReceived ∈ MESSAGE
  INITIALIZATION
    MessageReceived :=  EmptyMessage
  OPERATIONS
    ReceiveMessage ≡
      ANY m WHERE DecompositionFunction[m]  ⊆ PacketsTransmitted
        ∧ MessageReceived = EmptySet
      THEN MessageReceived := m
  ACTION
    skip
    [] ∃ m . (DecompositionFunction[m] ⊆ PacketsTransmitted)
                ∧ MessageReceived = EmptySet
              ⟹ ReceiveMessage
```

c'est-à-dire entre les ensembles *PacketsToTransmit* et *PacketsTransmitted.*

Ce dernier peut transférer un paquet (*ForwardPacket*) ou ne rien faire (*skip*).

Agent utilisateur

Le tableau 6.9 reprend la spécification de l'utilisateur.Ce dernier est le même que pour le système, au renommage près de *MessageIn* par *MessageToSend.*

Le Système

Le tableau 6.10 reprend la spécification du système raffiné. Il se compose des différents agents que nous venons de décrire et spécifie en outre quels

TAB. 6.8 – Réseau V1 - Agent Canal de communication

```
AGENT Channel
  INCLUDE
    Common_Sets
  VARIABLES
    PacketsTransmitted
  EXTERNAL_VARIABLES
   PacketsToTransmit
  EXTERNAL_CONSTRAINTS
   PacketsToTransmit ∈ ℙ(PACKET)
  INVARIANT
    PacketsTransmitted ∈ ℙ(PACKET)

  INITIALIZATION
    PacketsTransmitted = ∅
  OPERATIONS
    ForwardPacket ≡
      ANY p WHERE p in PacketsToTransmit ∧ p ∉ PacketsTransmitted
      THEN PacketsTransmitted = PacketsTransmitted ∪ p
  ACTION
    ∃ p.(p in PacketsToTransmit ∧ p ∉ PacketsTransmitted)
        ⟹ ForwardPacket
    [] skip
```

agents raffinent quels rôles.

La contrainte dynamique est une traduction de la contrainte du système abstrait en remplaçant *MessageIn* par *MessageToSend* et *MessageOut* par *MessageReceived.*

On remarquera aussi la présence de l'invariant de collage qui lie les variables du système raffiné et du système abstrait.

6.2.3 BCGS associé

Notre BCGS $< S, Q, \Pi, \pi, d, \delta >$se compose donc des éléments suivants :
– S

TAB. 6.9 – Réseau V1 - Utilisateur

```
AGENT User
  INCLUDE
    Common_Sets
  VARIABLES
    MessageToSend
  INVARIANT
    MessageToSend ∈ MESSAGE
INITILIZATION
    MessageToSend := EmptyMessage
  OPERATION
    Emit ≡
       ANY m WHERE m ∈ MESSAGE
        ∧ MessageToSend = EmptyMessage THEN
        MessageToSend := m
  ACTION
    skip
    [] MessageToSend = EmptyMessage ⟹ Send
```

$$S = \{User, Sender, Channel, Receiver\}$$

- Q

$$Q = \left\{ \begin{array}{c} Transmitting1_{m,s}, Transmitted1_m \\ |m \in MESSAGE \wedge s \in \mathbb{P}(DecompositionFunction[m]) \end{array} \right\}$$

L'état initial étant $Transmitted1_{EmptyMessage}$

- Π

$$\Pi = \begin{array}{l} \{MessageIn = m \wedge MessageOut = n | m, n \in MESSAGE\} \\ \cup \{packetsToTransmit = p, PacketsTransmitted = q | p, q \in \mathbb{P}PAcket \end{array}$$

- π

$$\pi = \begin{array}{ll} \{\bigcup_{m \in MESSAGE} \{(Transmitted1_m, & MessageIn = m \wedge MessageOut = m), \\ (Transmitting1_{m,p,q}, & MessageIn = m \wedge MessageOut = EmptyMessage \\ & \wedge PacketsTotransmit = p \wedge PacketsTransmitted = q) \end{array})\}\}$$

TAB. 6.10 – Réseau V1 - Système

```
SYSTEM NetworkSystemV1
  REFINES
    NetworkSystem
  INCLUDE
    Common_Elements
  EXTERNAL_VARIABLES
     messageToSend
  EXTERNAL_CONSTRAINTS
    MessageToSend ∈ MESSAGE
  INVARIANT
    MessageIn = MessageToSend
    MessageOut = MessageReceived
  AGENTS
    (Sender, Receiver, Channel) ROLE OF NetworkAgent
    Userv1 ROLE OF User

  DYNAMIC_CONSTRAINTS
   << Sender, Channel, Receiver >> ◇ MessageToSend = MessageReceived
```

- D qui indique les différents choix possibles pour les agents, c'est-à-dire le nombre de gardes de la clause action qui sont vraies à chaque état.
- δ qui reprend les transitions, résultant du choix par les différents agents d'une action dont la garde est vraie.

Nous remarquons dès à présent que ce BCGS, même en limitant le nombre de paquets des messages, commence à devenir difficilement représentable graphiquement.

La figure 6.5 en donne néammoins une représentation dans le cas assez simple ou MESSAGE = {EmptyMessage,m} et DecompositionFunction[m] = {p}. Toutefois, dans cette représentation, à des fins de lisisbilité, nous avons omis de spécifier les boucles correspondant au choix *skip* de chacun des agents, fait simultanément.

6.2.4 Analyse du raffinement

Le raffinement, qui est un raffinement par rôles spécifié par le terme ROLE OF de la spécification, est défini en reprenant les éléments évoqués dans le chapitre 5, de la manière suivante :

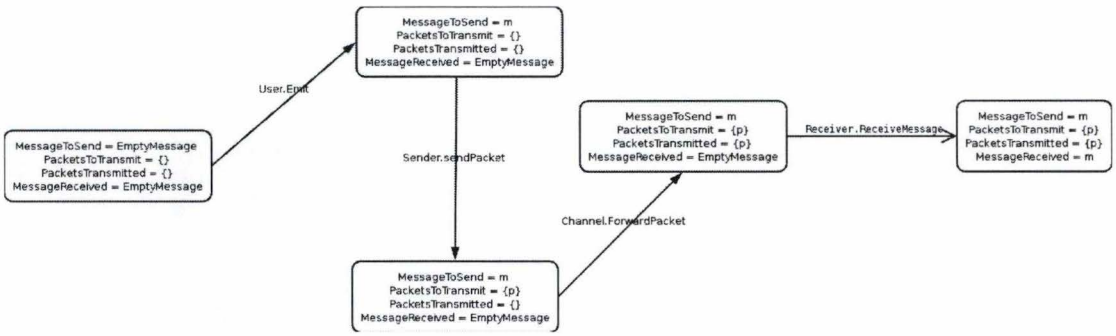

FIG. 6.5 – Représentation graphique du CGS du réseau raffiné

- la relation
  H =

$$\{(Transmitted_m, Transmitted1_m) | m \in MESSAGE\} \cup (\{Transmitting_m\} \times \{Transmitting1_{m,p,q} | p, q \in \mathbb{P}(DecompositionFunction[m]) | m \in MESSAGE\})\}$$

  qui lie donc bien les états du système abstraits à ceux du système concret.
- la relation D t.q

$$\begin{aligned} D = & \{((Transmitting_m, Transmitting1_{m,p,q}), (Sender, SendPacket), (NetworkAgent, skip))\} \\ & \cup \{((Transmitting_m, Transmitting1_{m,p,q}), (Sender, Skip), (NetworkAgent, skip))\} \\ & \cup \{((Transmitted_m, Transmitted1_m), (Sender, Skip), (NetworkAgent, skip))\} \\ & \cup \{((Transmitting_m, Transmitting1_{m,p,q}), (Receiver, ReceiveMessage), (NetworkAgent, Send))\} \\ & \cup \{((Transmitting_m, Transmitting1_{m,p,q}), (Receiver, Skip), (NetworkAgent, skip))\} \\ & \cup \{((Transmitted_m, Transmitted_m), (Receiver, Skip), (NetworkAgent, skip))\} \end{aligned}$$

- la relation C est inexistante car il n'y a pas de protagonistes, dans le sens défini à la section 5.2.3

La figure 6.6 reprend la correspondance des états entre les 2 systèmes.

6.2.5 Vérification

Etant donné que le raffinement que nous venons de définir est un raffinement par extension (cfr. section 5.2.3), la contrainte dynamique est toujours valable et l'ensemble de stratégies gagnantes est préservé. Après réécriture

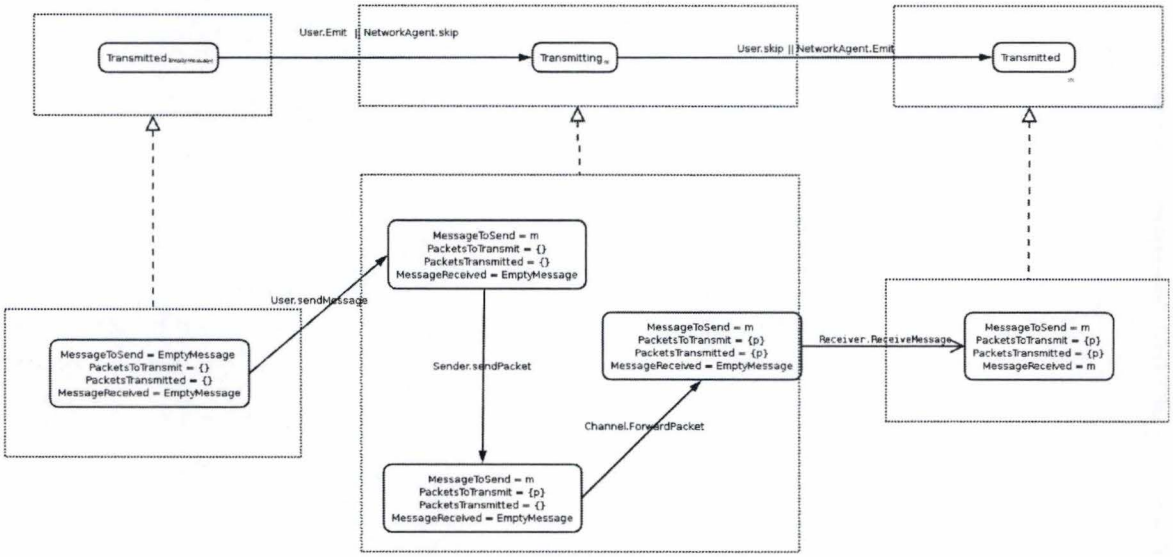

FIG. 6.6 – Correspondance des CGS entre le raffinement et le système abstrait

des stratégies du système abstrait, on obtient la fonction f suivante pour les différents agents :

- pour l'émetteur :

$$f(Transmitted1_{EmptyMessage}...Transmiting1_{m,p,q} \\ avec\ p \neq DecompositionFunction[m])(Sender) = SendPacket$$
$$f(Transmitted1_{EmptyMessage}...Transmiting1_{m,p,q} \\ avec\ p \subset DecompositionFunction[m])(Sender) = SendPacket$$
$$f(Transmitted1_{EmptyMessage}...Transmited1_m)(Sender) = skip$$
$$f(Transmitted1_{EmptyMessage}...Transmitted1_{EmptyMessage})(NetworkAgent) = skip$$

- pour le Canal de transmission :

$$f(Transmitted1_{EmptyMessage}...Transmiting1_{m,p,q} \\ avec\ p \subset q)(Channel) = ForwardPacket$$
$$f(Transmitted1_{EmptyMessage}...Transmiting1_{m,p,q} \\ avec\ p = q)(Channel) = SendPacket$$
$$f(Transmitted1_{EmptyMessage}...Transmited1_m)(Channel) = skip$$
$$f(Transmitted1_{EmptyMessage}...Transmitted1_{EmptyMessage})(Channel) = skip$$

- pour le récepteur :

$$f(Transmitted1_{EmptyMessage}...Transmiting1_{m,p,q} \\ avec\ p \subset q)(Receiver) = skip$$
$$f(Transmitted1_{EmptyMessage}...Transmiting1_{m,p,q} \\ avec\ p = q)(Receiver) = ReceiveMessage$$
$$f(Transmitted1_{EmptyMessage}...Transmited1_{m})(Receiver) = skip$$
$$f(Transmitted1_{EmptyMessage}...Transmitted1_{EmptyMessage})(Receiver) = skip$$

Il est à noter que la contrainte dynamique du système n'est pas à vérifier, étant donné que nous avons caractérisé le raffinement par une relation correcte de raffinement par extension de rôle. Néammoins, seule une analyse de l'automate peut nous permettre d'obtenir les stratégies gagnantes. Heureusement, comme nous connaissons la stratégie gagnante du système abstrait, nous n'avons plus qu'à analyser la partie qui diffère, c'est-à-dire les transitions entre les états du système concret reliés au même état du système abstrait par H.

6.3 Deuxième raffinement : contraintes du canal

Nous avons, au cours des étapes précédentes, défini le rôle des différents agents : dans un premier temps, du point de vue interaction avec le monde extérieur (dans ce cas l'utilisateur), puis dans sa dynamique interne.

L'étape suivante constitue en la prise en compte de certaines contraintes physiques.

6.3.1 Description du problème

Dans ce cas-ci, nous allons considérer que notre but est de construire un réseau, d'un point de vue logiciel, en se basant sur des composants externes. Dans ce cas-ci, le canal de transmission est un élément développé à l'extérieur que nous pouvons utiliser mais dont nous n'avons pas le maîtrise.

Comme nous le savons, ce genre de dispositifs possède, en général, un certain nombre de problèmes connus.

- Ll perte de données qui consiste en le fait, pour le canal de transmission de ne jamais transférer certains paquets (cela est possible dans le système décrit ci-avant)
- la duplication de paquets consistant à transmettre plusieurs fois un même paquet. Ceci n'est pas possible dans le système décrit.
- le réordonnancement par le canal de certains paquets. Dans notre spécification abstraite, cela ne pose pas de problèmes mais, dans une implémentation, cela pourrait en poser un.

Néammoins, nous savons, de par la construction de tels dispositifs, qu'ils laissent passer régulièrement certains paquets et ne peuvent donc bloquer indéfiniment.

Nous allons modéliser dans cette partie la perte de données.

6.3.2 Spécification ATL-B

Par rapport à la spécification du système abstrait, la distinction majeure se fait de par le fait que les ensembles textitPacketsToTransmit et textit-PacketsTransmitted, sont devenus des séquences, construction B modélisant une suite indexable d'éléments. Ceci permet de distinguer un paquet éventuellement réémis si le canal le perd.

Emetteur

Le tableau 6.11 reprend le code du nouvel émetteur.

Par rapport à sa version précédente, nous voyons qu'il peut réémettre (via *reSendPacket*) un paquet déjà émis, seule façon qu'il aura d'assurer l'envoi du message vu que le canal peut perdre des paquets.

Récepteur

Le tableau 6.12 reprend la nouvelle spécification du récepteur. Cette dernière ne change en fait pas si ce n'est quant au type de PacketsTransmit-tedv2.

Canal de transmission

Le tableau 6.13 reprend la spécification de l'agent Canal de transmission de notre système raffiné.

Dans ce système , nous avons modélisé la perte d'un paquet par le nouveau choix *LoosePacket* de l'agent en question.

Par rapport à la version précédente, nous voyons que le canal traite les messages, retenant quels messages sont traités (via *PacketsProcessed*). Chaque message est traité, soit en l'envoyant dans *packetsTransmittedv2*, soit en ne faisant rien, ce qui correspond à la perte du paquet.

Système

Le tableau 6.14 reprend la spécification de notre système.

Nous remarquons cette fois-ci l'apparition d'une propriété dynamique qui indique que l'agent *Channel* ne pourra pas faire en sorte que tout message non-traité soit systématiquement perdu.

TAB. 6.11 – Réseau V2 - Emetteur

```
AGENT SenderV2
  INCLUDE
    Common_Sets
  VARIABLES
    PacketsToTransmitv2
  EXTERNAL_VARIABLES
    MessageToSendv2
  EXTERNAL_CONSTRAINTS
    MessageToSendv2 ∈ MESSAGE
  INVARIANT
   PacketsToTransmitv2 ∈ seq(PACKET)
  INITIALIZATION
    PacketsToTransmitv2 = []
  OPERATIONS
    sendPacket ≡
    ANY p WHERE p ∈ DecompositionFunction(MessageToSendv2)
          ∧ p ∉ ran(PacketsToTransmitv2)
    THEN
     PacketsToTransmitv2 := PacketsToTransmitv2 ← m
  reSendPacket ≡
    ANY p WHERE p ∈ DecompositionFunction(MessageToSendv2)
          ∧ p ∈ ran(PacketsToTransmitv2)
    THEN
      PacketsToTransmitv2 := PacketsToTransmitv2 ← m

  ACTION
  skip
  [] ∃ p. (p ∈ DecompositionFunction(MessageToSendv2)
                    ∧ p ∉ ran(PacketsToTransmitv2)
              ⟹ sendMessage
  [] ∃ p. (p ∈ DecompositionFunction(MessageToSendv2)
                    ∧ p ∈ ran(PacketsToTransmitv2))
           ⟹ nextMessage
```

Ceci spécifie donc que cette propriété est donnée comme acquise et ne devra pas être implémentée dans l'implémentation finale du système.

TAB. 6.12 – Réseau V2 - Récepteur

```
AGENT ReceiverV2
  INCLUDE
    Common_Sets
  VARIABLES
    MessageReceivedv2
  EXTERNAL_VARIABLES
    PacketsTransmittedv2
  EXTERNAL_CONSTRAINTS
    PacketsTransmittedv2 ∈ seq1(PACKET)
  INVARIANT
    MessageReceivedv2 ∈ MESSAGE
  INITIALIZATION
    MessageReceivedv2 := EmptyMessage
  OPERATIONS
    ReceiveMessage ≡
      ANY m WHERE DecompositionFunction[m] ⊆ ran(PacketsTransmittedv2)
                  ∧ MessageReceived = EmptySet
       THEN MessageReceivedv2
                  := m
  ACTION
    skip
    [] ∃ m. (DecompositionFunction[m] ⊆ ran(PacketsTransmittedv2))
              ⟹ ReceiveMessage [] skip
```

6.3.3 BCGS

La figure 6.7 représente une partie du BCGS dont nous ne détaillerons pas les différents éléments , la définition étant similaire à ce que nous avons fait précédemment.

Nous voyons directement dans cet exemple, que même le cas simple que nous considérons devient impossible à représenter graphiquement et contient un nombre imposant d'états.

Dans ce cas-ci, il subsiste même un problème insurmontable à la représentation en mémoire, cet automate contient un nombre infini d'états, dû au fait qu'une séquence de paquets n'a, à priori, pas de limites.

Il faudra donc limiter, si nous voulons opérer des calculs sur ce modèle, soit limiter la taille de *PacketsToTransmitv2*, soit le représenter sous une forme symbolique représentable par un nombre fini d'états.

TAB. 6.13 – Réseau V2 - Agent Canal de communication

```
AGENT ChannelV2
  INCLUDE
    Common_Sets
  VARIABLES
    PacketsTransmittedv2
    Losed
  EXTERNAL_VARIABLES
   PacketsToTransmitv2
  EXTERNAL_CONSTRAINTS
   PacketsToTransmitv2 ∈ seq(PACKET)
  INVARIANT
    PacketsTransmittedv2 ∈ seq(PACKET)
    PacketsProcessed ∈ ℙ(ℕ)


  INITIALIZATION
    PacketsTransmittedv2 = []
  OPERATIONS
    ForwardPacket ≡
      ANY i WHERE i ∉ PacketsProcessed
      THEN PacketsTransmittedv2, PacketsProcessed
        := PacketsTransmittedv2 ← {PacketsToTransmitv2(i)}
           , PacketsProcessed  ∪ {i}

    LoosePacket ≡
      ANY i WHERE i ∉ PacketsProcessed
      THEN  PacketsProcessed
        :=  PacketsProcessed  ∪ {i}

  ACTION
    skip
    [] ∃ i ∉ PacketsProcessed
              ⟹ ForwardPacket
    []  ∃ i ∉ PacketsProcessed
              ⟹ LoosePacket
```

6.3.4 Analyse du raffinement

Les raffinements envisagés sont de nouveau des raffinements par extension. La correspondance peut s'obtenir de la même façon que pour l'étape

TAB. 6.14 – Réseau V2 - Système

```
SYSTEM NetworkSystemV2
  REFINES
    NetworkSystemV1
  INCLUDE
    Common_Sets
  EXTERNAL_VARIABLES
     messageIn
  EXTERNAL_CONSTRAINTS
    Messagein ∈ seq(MESSAGE)
  INVARIANT
    MessageToSendv2 = MessageToSend
    MessageReceivedv2 =MessageReceived
    ran(PacketsTransmittedv2) = PacketsTransmitted
    ran(PacketsToTransmitv2) = PacketsToTransmitv2
  AGENTS
    SenderV2 EXTENDS ROLE OF Sender
    ReceiverV2 EXTENDS ROLE OF Receiver
    ChannelV2 EXTENDS ROLE OF  Channel
    UserV2 IS ROLE OF Userv1

  DYNAMIC_PROPERTIES
    ¬ << Channel >> □ (∃ i .(i∉ PacketsProcessed
                        ∧ i < size(PacketsToTransmitv2))
                        ∧  ◯ PacketsToTransmitv2(i) ∉ ran(PacketsTransmittedv2))
  DYNAMIC_CONSTRAINTS
   << Sender, Receiver >> ◇ MessageToSendv2 = MessageReceivedv2
```

précédente.

6.4 Conclusion de l'exemple

Nous espérons avoir pu montrer via ce petit exemple, qu'il est possible de dégager une cohérence de la méthode ATL-B précédemment définie. L'exemple montre ainsi comment modéliser l'interaction d'agents et la façon dont on peut les décrire par raffinements successifs.

Si nous avions voulu être complet pour cet exemple, il nous manque 2 étapes dans le processus de développement.

– l'implémentation des agents : qui devra fournir une description non-

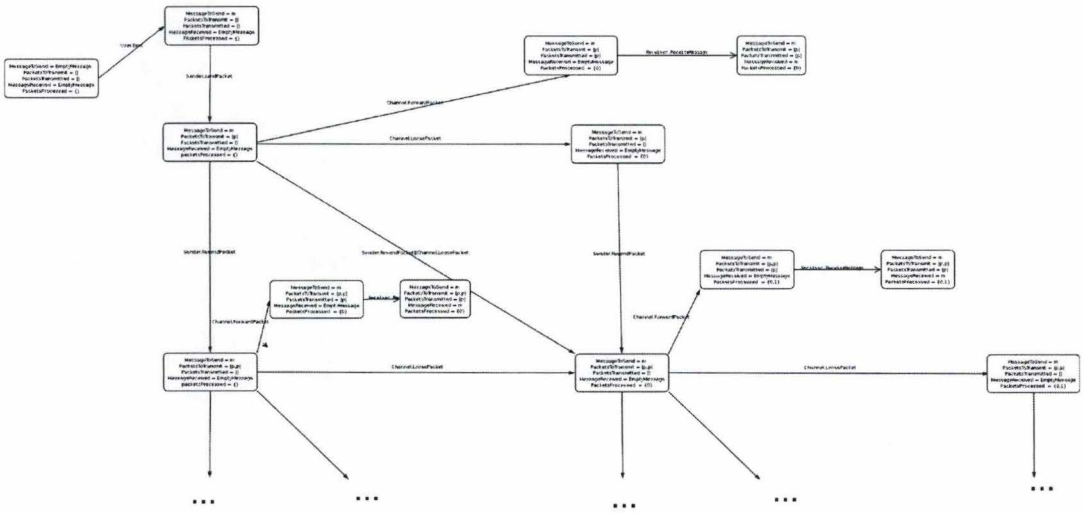

FIG. 6.7 – CGS du second raffinement

déterministe des agents émetteur et récepteur, c'est-à-dire qu'on décrira complètement la manière de faire communiquer ces agents.

- l'implémentation du système qui devra regrouper les émetteurs et récepteurs implémentés et redéfinir leur interaction. Il faudra vérifier, dans cette phase, le modèle obtenu pour s'assurer qu'il respecte le comportement du système original.

Conclusion

Nous avons, dans ce travail, tenté de montrer comment il était possible, tout en se plaçant dans le cadre d'une méthode basée sur la preuve de systèmes monolithiques, d'intégrer à cette dernière la dynamique issue de la théorie des jeux et de la logique temporelle, afin de fournir une description cohérente d'un système informatique.

Nous avons également mis en évidence la possibilité de procéder par niveaux d'abstractions, permettant d'assurer un processus de développement maîtrisé, dans un cadre sémantique homogène.

Il apparaît, au vu de ce que nous venons d'exposer, que les aspects statiques de B et dynamiques de ATL apportent une vision complémentaire, plutôt qu'antagoniste, d'un modèle de la réalité.

Nous ne voulons néammoins pas minimiser le travail à accomplir avant d'arriver à intégrer ces notions en un tout rigoureusement défini et exploitable. Nous l'avons vu, les structures étudiées ne sont pas, à l'heure actuelle, efficaces en termes de calculabilité et, une formalisation exhaustive fera d'autant plus apparaître les difficultés à unifier les concepts de ces deux paradigmes.

Bibliographie

[00bds] Z bibliography. URL : `http://www.comlab.ox.ac.uk/archive/z/bib.html`, 1990 onwards.

[Abr96] J.-R. Abrial. The B-book : assigning programs to meanings. Cambridge University Press, New York, NY, USA, 1996.

[Abr99] J.-R. Abrial. Event Driven System Construction. Clearsy, April 99.

[AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. J. ACM, 49(5) :672–713, 2002.

[AHKV98] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In Proc. 9th Conferance on Concurrency Theory, Lecture Notes in Computer Science, Nice, September 1998. Springer-Verlag.

[AM98] Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in B. In B'98 : The 2nd International B Conference, pages 83–128, 98.

[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. Systems and Software Verification. Model-Checking Techniques and Tools. Springer, 2001.

[Cle02] Clearsu. Manuel de référence du langage B. Clearsy, 1.8.5 edition, January 02.

[GHR94] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. Temporal logic (vol. 1) : mathematical foundations and computational aspects. Oxford University Press, Inc., New York, NY, USA, 1994.

[HA98a] Thomas A. Henzinger and R. Alur. Reactive modules. Technical Report UCB/ERL M98/41, EECS Department, University of California, Berkeley, 1998.

[HA98b] Thomas A. Henzinger and R. Alur. Reactive modules. Technical Report UCB/ERL M98/41, EECS Department, University of California, Berkeley, 1998.

[Mas01] P.-A. Masson. Vérification par model-checking modulaire de propriétés dynamiques PLTL exprimées dans le cadre de spécifications B événementielles. Thèse de Doctorat, 21 Décembre 2001. Rapporteurs : S. Bensalem (Verimag), D. Bert (LSR-IMAG), P. Schnoebelen (LSV - ENS Cachan). Examinateurs : F. Bellegarde, J. Julliand, H. Mountassir (Besançon). Directeur : J. Julliand.

[ocB] University of california Berkeley. Mocha website. http ://embedded.eecs.berkeley.edu/research/mocha/. consulté le 16/02/2005.

[RS02] Mark Ryan and Pierre-Yves Schobbens. Agents and roles : Refinement in alternating-time temporal logic. In ATAL '01 : Revised Papers from the 8th International Workshop on Intelligent Agents VIII, pages 100–114, London, UK, 2002. Springer-Verlag.

[Wik06] Wikipedia. Definition of ltl. `http://en.wikipedia.org/wiki/Linear_temporal_logic`, February 2006. consulté le 18/02/2006.

Annexe

Syntaxe ATL-B

Cette grammaire constitue une modification de la grammaire disponible dans [Cle02]

Axiome

```
Composant ::=
  Système
  |Système_raffinement
  | Agent_abstrait
  | Agent_Raffinement
  | Agent_Implantation
```

Clauses

```
Agent_abstrait ::=
  "AGENT" En-tête
    Clause_agent_abstrait*
    Clause_actions
  "END"

Clause_agent_abstrait ::=
    Clause_constraints
  | Clause_external_constraints
  | Clause_sees
  | Clause_includes
  | Clause_promotes
  | Clause_extends
  | Clause_uses
  | Clause_sets
  | Clause_concrete_constants
  | Clause_abstract_constants
```

```
  | Clause_properties
  | Clause_concrete_variables
  | Clause_abstract_variables
  | Clause_external_variables
  | Clause_invariant
  | Clause_assertions
  | Clause_initialisation
  | Clause_operations

En-tête ::=
  Ident [ "(" Ident+"," ")" ]

Agent_Raffinement ::=
  "AGENT" En-tête
  Clause_refines
  Clause_agent_raffinement*
  Clause_actions
  "END"

Clause_agent_raffinement ::=
  Clause_sees
  | Clause_includes
  | Clause_promotes
  | Clause_extends
  | Clause_sets
  | Clause_concrete_constants
  | Clause_abstract_constants
  | Clause_properties
  | Clause_concrete_variables
  | Clause_abstract_variables
  | Clause_invariant
  | Clause_assertions
  | Clause_initialisation
  | Clause_operations

Agent_Implantation ::=
  "AGENT" En-tête
  Clause_refines
  Clause_agent_implantation*
  Clause_actions
  "END"

Clause_implantation ::=
  Clause_sees
```

```
  | Clause_imports
  | Clause_promotes
  | Clause_extends_BO
  | Clause_sets
  | Clause_concrete_constants
  | Clause_properties
  | Clause_values
  | Clause_concrete_variables
  | Clause_invariant
  | Clause_assertions
  | Clause_initialisation_BO
  | Clause_operations_BO
  | Clause_operations_locales

Système_abstrait ::=
  "SYSTEM" En-tête
    Clause_système_abstrait*
  "END"

Clause_système_abstrait ::=
  | Clause_agents
  | Clause_sets
  | Clause_concrete_constants
  | Clause_abstract_constants
  | Clause_properties
  | Clause_external_variables
  | Clause_external_constraints
  | Clause_dynamic_constraints
  | Clause_dynamic_properties

Système_raffinement ::=
  "SYSTEM" En-tête
   Clause_refines
    Clause_système_raffinement*
  "END"

Clause_système_raffinement ::=
  | Clause_agents_with_roles
  | Clause_sets
  | Clause_concrete_constants
  | Clause_abstract_constants
  | Clause_invariant
  | Clause_properties
  | Clause_external_variables
```

```
  | Clause_external_constraints
  | Clause_dynamic_constraints
  | Clause_dynamic_properties


Clause_constraints ::=
  "CONSTRAINTS" Prédicat

Clause_refines ::=
  "REFINES" Ident

Clause_IMPORTS ::=
  "IMPORTS" ( Ident_ren [ "(" Instanciation_BO+"," ")" ] )+","

Instanciation_BO ::=
    Terme
  | Ensemble_entier_BO
  | "BOOL"
  | Intervalle_BO

Clause_sees ::=
  "SEES" Ident_ren+","

Clause_includes ::=
  "INCLUDES" ( Ident_ren [ "(" Instanciation+"," ")" ] )+","

Instanciation ::=
  Terme
  | Ensemble_entier
  | "BOOL"
  | Intervalle

Clause_promotes ::=
  "PROMOTES" Ident_ren+","

Clause_EXTENDS ::=
  "EXTENDS" ( Ident_ren [ "(" Instanciation+"," ")" ] )+","

Clause_EXTENDS_BO ::=
  "EXTENDS" ( Ident_ren [ "(" Instanciation_BO+"," ")" ] )+","

Clause_uses ::=
  "USES" Ident_ren+","
```

```
Clause_sets ::=
  "SETS" Ensemble+";"

Ensemble ::=
  Ident
  | Ident "=" "" Ident+"," ""

Clause_concrete_constants ::=
  "CONCRETE_CONSTANTS" Ident+","
  | "CONSTANTS" Ident+","

Clause_abstract_constants ::=
  "ABSTRACT_CONSTANTS" Ident+","

Clause_properties ::=
  "PROPERTIES" Prédicat

Clause_values ::=
  "VALUES" Valuation+";"

Valuation ::=
  Ident "=" Terme
  | Ident "=" Expr_tableau
  | Ident "=" Intervalle_BO

Clause_concrete_variables ::=
  "CONCRETE_VARIABLES" Ident_ren+","

Clause_abstract_variables ::=
  "ABSTRACT_VARIABLES" Ident_ren+","
  | "VARIABLES" Ident_ren+","

Clause_invariant ::=
  "INVARIANT" Prédicat

Clause_assertions ::=
  "ASSERTIONS" Prédicat+";"

Clause_initialisation ::=
  "INITIALISATION" Substitution

Clause_initialisation_BO ::=
  "INITIALISATION" Instruction
```

```
Clause_operations ::=
  "OPERATIONS" Opération+";"

Opération ::=
  Entête_opération "=" Substitution_corps_opération

Entête_opération ::=
  [ Ident+"," "leftarrow"] Ident_ren [ "(" Ident+"," ")" ]

Clause_operations_BO ::=
  "OPERATIONS" Opération_BO+";"
  Entête_opération "=" Instruction_corps_opération

Clause_operations_locales ::=
  "LOCAL_OPERATIONS" Opération+";"

Clause_external_variables ::=
  "EXTERNAL_VARIABLES" Ident_ren+","

Clause_external_constraints ::=
  "EXTERNAL_CONSTRAINTS" Prédicat

Clause_actions ::=
"ACTIONS" Action+"□"

Action ::=
[Prédicat " ⟹ "] Substitution_corps_opération

Clause_agents ::=
  Ident+","

Clause_agents_with_roles ::=
  ((Ident|("("Ident")")+"," "IS ROLE OF" Ident)+","
  ((Ident|("("Ident")")+"," "EXTENDS ROLE OF" Ident)+","
  ((Ident|("("Ident")")+"," "IMPLEMENTS ROLE OF" Ident)+","

Clause_dynamic_constraints ::=
  "DYNAMIC CONSTRAINTS" Prédicat_ATL

Clause_dynamic_properties
"DYNAMIC PROPERTIES" Prédicat_ATL
```

Termes et regroupement d'expressions

```
Terme ::= Terme_simple
  | Expression_arithmétique
  | Terme_record
  | Terme_record ( "'" Ident )+

Terme_simple ::=
  Ident_ren
  | Entier_lit
  | Booléen_lit
  | "bool" "(" Condition ")"
  | Ident_ren ( "'" Ident )+

Entier_lit ::= Entier_littéral
  | "MAXINT"
  | "MININT"

Booléen_lit ::= "FALSE"
  | "TRUE"

Expression_arithmétique ::=
  Entier_lit
  | Ident_ren
  | Ident_ren "(" Terme+"," ")"
  | Ident_ren ( "'" Ident )+
  | Expression_arithmétique "+" Expression_arithmétique
  | Expression_arithmétique "-" Expression_arithmétique
  | "-" Expression_arithmétique
  | Expression_arithmétique "x" Expression_arithmétique
  | Expression_arithmétique "/" Expression_arithmétique
  | Expression_arithmétique "mod" Expression_arithmétique
  | Expression_arithmétique Expression_arithmétique
  | "succ" "(" Expression_arithmétique ")"
  | "pred" "(" Expression_arithmétique ")"
  | "(" Expression_arithmétique ")"

Terme_record ::=
  "rec" "(" ( [ Ident ":" ] ( Terme | Expr_tableau ) )+"," ")"

Expr_tableau ::=
  Ident
  | "" ( Terme_simple+"↦" "↦" Terme )+"," ""
```

```
  | Ensemble_simple+"×" "×" "" Terme ""

Intervalle_B0 ::=
  Expression_arithmétique ".." Expression_arithmétique
  | Ensemble_entier_B0

Ensemble_entier_B0 ::=
  "NAT"
  | "NAT1"
  | "INT"
```

Conditions

```
Condition ::=
  Terme_simple "=" Terme_simple
  | Terme_simple "." Terme_simple
  | Terme_simple "<" Terme_simple
  | Terme_simple ">" Terme_simple
  | Terme_simple "=" Terme_simple
  | Terme_simple "=" Terme_simple
  | Condition "." Condition
  | Condition "." Condition
  | "¬" "(" Condition ")"
  | "(" Condition ")"
```

Instructions

```
Instruction ::=
  Instruction_bloc
  | Instruction_variable_locale
  | Substitution_identité
  | Instruction_devient_égal
  | Instruction_appel_opération
  | Instruction_conditionnelle
  | Instruction_cas
  | Instruction_assertion
  | Instruction_séquence
  | Substitution_tant_que

Instruction_corps_opération ::=
  Instruction_bloc
  | Instruction_variable_locale
  | Substitution_identité
```

```
  | Instruction_devient_égal
  | Instruction_appel_opération
  | Instruction_conditionnelle
  | Instruction_cas
  | Instruction_assertion
  | Substitution_tant_que

Instruction_bloc ::=
  "BEGIN" Instruction "END"

Instruction_variable_locale ::=
  "VAR" Ident+"," "IN" Instruction "END"

Instruction_devient_égal ::=
  Ident_ren [ "(" Terme+"," ")" ] ":=" Terme
  | Ident_ren ":=" Expr_tableau
  | Ident_ren ("'" Ident )+ ":=" Terme

Instruction_appel_opération ::=
  [ Ident_ren+"," "←" ] Ident_ren [ "(" (Terme | Chaîne_lit)+"," ")" ]

Instruction_séquence ::=
  Instruction ";" Instruction

Instruction_conditionnelle ::=
  "IF" Condition "THEN" Instruction
  ( "ELSIF" Condition "THEN" Instruction )*
  [ "ELSE" Instruction ]
  "END"

Instruction_cas ::=
  "CASE" Terme_simple "OF"
  "EITHER" Terme_simple+"," "THEN" Instruction
  ( "OR" Terme_simple+"," "THEN" Instruction )*
  [ "ELSE" Instruction ]
  "END"
  "END"

Substitution_tant_que ::=
  "WHILE" Condition "DO" Instruction
  "INVARIANT" Prédicat
  "VARIANT" Expression
  "END"
```

```
Instruction_assertion ::=
  "ASSERT" Condition "THEN" Instruction "END"
```

Prédicats

```
Prédicat ::=
  Prédicat_parenthésé
  | Prédicat_conjonction
  | Prédicat_négation
  | Prédicat_disjonction
  | Prédicat_implication
  | Prédicat_équivalence
  | Prédicat_universel
  | Prédicat_existentiel
  | Prédicat_égalité
  | Prédicat_inégalité
  | Prédicat_appartenance
  | Prédicat_non_appartenance
  | Prédicat_inclusion
  | Prédicat_inclusion_stricte
  | Prédicat_non_inclusion
  | Prédicat_non_inclusion_stricte
  | Prédicat_inférieur_ou_égal
  | Prédicat_strictement_inférieur
  | Prédicat_supérieur_ou_égal
  | Prédicat_strictement_supérieur

Prédicat_parenthésé ::= "(" Prédicat ")"

Prédicat_conjonction ::= Prédicat "∧" Prédicat

Prédicat_négation ::= "¬" "(" Prédicat ")"

Prédicat_disjonction ::= Prédicat "∨" Prédicat

Prédicat_implication ::= Prédicat " ⟹ " Prédicat

Prédicat_équivalence ::= Prédicat " ⟺ " Prédicat

Prédicat_universel ::= "∀" Liste_ident "." "(" Prédicat "." Prédicat ")"

Prédicat_existentiel ::= "∃" Liste_ident "." "(" Prédicat ")"
```

```
Prédicat_égalité ::= Expression "=" Expression

Prédicat_inégalité ::= Expression "≠" Expression

Prédicat_appartenance ::= Expression "∈" Expression

Prédicat_non_appartenance ::= Expression "∉" Expression

Prédicat_inclusion ::= Expression "⊆" Expression

Prédicat_inclusion_stricte ::= Expression "⊂" Expression

Prédicat_non_inclusion ::= Expression "⊈" Expression

Prédicat_non_inclusion_stricte ::= Expression "⊄" Expression

Prédicat_inférieur_ou_égal ::= Expression "≤" Expression

Prédicat_strictement_inférieur ::= Expression "<" Expression

Prédicat_supérieur_ou_égal ::= Expression "≥" Expression

Prédicat_strictement_supérieur ::= Expression ">" Expression
```

Prédicats ATL

```
Prédicat_ATL ::=
  PrédicatATL_parenthésé
  | PrédicatATL_conjonction
  | PrédicatATL_négation
  | PrédicatATL_disjonction
  | PrédicatATL_implication
  | PrédicatATL_équivalence
  | PrédicatATL_next
  | PrédicatATL_eventually
  | PrédicatATL_always
  | PrédicatATL_Until
  | Prédicat_universel
  | Prédicat_existentiel
  | Prédicat_égalité
  | Prédicat_inégalité
  | Prédicat_appartenance
  | Prédicat_non_appartenance
```

```
  | Prédicat_inclusion
  | Prédicat_inclusion_stricte
  | Prédicat_non_inclusion
  | Prédicat_non_inclusion_stricte
  | Prédicat_inférieur_ou_égal
  | Prédicat_strictement_inférieur
  | Prédicat_supérieur_ou_égal
  | Prédicat_strictement_supérieur

PrédicatATL_parenthésé ::= "(" Prédicat_ATL ")"

PrédicatATL_conjonction ::= Prédicat_ATL "∧" Prédicat_ATL

PrédicatATL_négation ::= "¬" "(" Prédicat_ATL ")"

PrédicatATL_disjonction ::= Prédicat_ATL "∨" Prédicat_ATL

PrédicatATL_implication ::= Prédicat_ATL " ⟹ " Prédicat_ATL

PrédicatATL_équivalence ::= Prédicat_ATL " ⟺ " Prédicat_ATL

PrédicatATL_next ::= coopération_agent "◯" Prédicat_ATL

PrédicatATL_eventually ::= coopération_agent "◇" Prédicat_ATL

PrédicatATL_always ::= coopération_agent "□" Prédicat_ATL

PrédicatATL_Until ::= coopération_agent  Prédicat_ATL "𝒰" Prédicat_ATL

coopération_agent ::= "<<" Ident*"," ">>"
```

Expressions

```
Expression ::=
  Expression_primaire
  | Expression_booléenne
  | Expression_arithmétique
  | Expression_de_couples
  | Expression_d_ensembles
  | Construction_d_ensembles
  | Expression_de_records
  | Expression_de_relations
```

```
  | Expression_de_fonctions
  | Construction_de_fonctions
  | Expression_de_suites
  | Construction_de_suites
  | Expression_d_arbres

Expression_primaire ::=
  Donnée
  | Expr_parenthésée
  | Chaîne_lit

Expression_booléenne ::=
  Booléen_lit
  | Conversion_bool

Expression_arithmétique ::=
  Entier_lit
  | Addition
  | Différence
  | Moins_unaire
  | Produit
  | Division
  | Modulo
  | Puissance
  | Successeur
  | Prédécesseur
  | Maximum
  | Minimum
  | Cardinal
  | Somme_généralisée
  | Produit_généralisé

Expression_de_couples ::=
  Couple

Expression_d_ensembles ::=
  Ensemble_vide
  | Ensemble_entier
  | Ensemble_booléen
  | Ensemble_chaînes

Construction_d_ensembles ::=
  | Produit
  | Ens_compréhension
```

```
  | Sous_ensembles
  | Sous_ensembles_finis
  | Ens_extension
  | Intervalle
  | Différence
  | Union
  | Intersection
  | Union_généralisée
  | Intersection_généralisée
  | Union_quantifiée
  | Intersection_quantifée

Expression_de_records ::=
  Ensemble_records
  | Record_en_extension
  | Champ_de_record

Expression_de_relations ::=
  Ensemble_relations
  | Identité
  | Inverse
  | Première_projection
  | Deuxième_projection
  | Composition
  | Produit_directe
  | Produit_parallèle
  | Itération
  | Fermeture_réflexive
  | Fermeture
  | Domaine
  | Codomaine
  | Image
  | Restriction_domaine
  | Soustraction_domaine
  | Restriction_codomaine
  | Soustraction_codomaine
  | Surcharge

Expression_de_fonctions ::=
  Fonction_partielle
  | Fonction_totale
  | Injection_partielle
  | Injection_totale
  | Surjection_partielle
```

```
  | Surjection_totale
  | Bijection_partielle
  | Bijection_totale

Construction_de_fonctions ::=
  Lambda_expression
  | Évaluation_fonction
  | Transformée_fonction
  | Transformée_relation

Expression_de_suites ::=
  Suites
  | Suites_non_vide
  | Suites_injectives
  | Suites_inj_non_vide
  | Permutations
  | Suite_vide
  | Suite_extension

Construction_de_suites ::=
  | Taille_suite
  | Premier_élément_suite
  | Dernier_élément_suite
  | Tête_suite
  | Queue_suite
  | Inverse_suite
  | Concaténation
  | Insertion_tête
  | Insertion_queue
  | Restriction_tête
  | Restriction_queue
  | Concat_généralisée

Expression_d_arbres ::=
  Arbres
  | Arbres_binaires
  | Construction_arbre
  | Racine_arbre
  | Fils_arbre
  | Aplatissement_préfixé
  | Aplatissement_postfixé
  | Taille_arbre
  | Symétrie_arbre
  | Rang_noeud
```

```
  | Père_noeud
  | Fils_noeud
  | Sous_arbre_noeud
  | Arité_noeud

Donnée ::= Ident_ren
  | Ident_ren"$0"
  Expr_parenthésée ::= "(" Expression ")"
  Chaîne_lit ::= Chaîne_de_caractères
  Booléen_lit ::= "FALSE"
  | "TRUE"

Conversion_bool ::= "bool" "(" Prédicat ")"
  Entier_lit ::= Entier_littéral
  | "MAXINT"
  | "MININT"

Addition ::= Expression "+" Expression

Différence ::= Expression "-" Expression

Moins_unaire ::= "-" Expression

Produit ::= Expression "×" Expression

Division ::= Expression "/" Expression

Modulo ::= Expression "mod" Expression

Puissance ::= Expression^Expression

Successeur ::= "succ" ["(" Expression ")"]

Prédécesseur ::= pred ["(" Expression ")"]

Maximum ::= "max" "(" Expression ")"

Minimum ::= "min" "(" Expression ")"

Cardinal ::= "card" "(" Expression ")"

Somme_généralisée ::= "Σ" Liste_ident "." "(" Prédicat "|" Expression ")"

Produit_généralisé ::= "Π" Liste_ident "." "(" Prédicat "|" Expression ")"
```

```
Couple ::= Expression "↦" Expression
  | Expression "," Expression

Ensemble_vide ::= "∅"

Ensemble_entier ::= "ℤ"
  | "ℕ"
  | "ℕ1"
  | "NAT"
  | "NAT1"
  | "INT"

Ensemble_booléen ::= "BOOL"

Ensemble_chaînes ::= "STRING"

Ens_compréhension ::= "" Ident+"," "|" Prédicat ""

Sous_ensembles ::= "ℙ" "(" Expression ")"
  | "ℙ1" "(" Expression ")"

Sous_ensembles_finis ::= "F" "(" Expression ")"
  | "F1" "(" Expression ")"

Ens_extension ::= "" Expression+"," ""

Intervalle ::= Expression ".." Expression

Union ::= Expression "∪" Expression

Intersection ::= Expression "∩" Expression

Union_généralisée ::= "union" "(" Expression ")"

Intersection_généralisée ::= "inter" "(" Expression ")"

Union_quantifiée ::= "⋃" Liste_ident "." "(" Prédicat "|" Expression ")"

Intersection_quantifée ::= "⋂" Liste_ident "." "(" Prédicat "|" Expression ")"

Ensemble_records ::= "struct" "(" ( Ident ":" Expression )+"," ")"

Record_en_extension ::= "rec" "(" ( [ Ident ":" ] Expression )+"," ")"
```

Champ_de_record ::= Expression "'" Ident

Ensemble_relations ::= Expression "↔" Expression

Identité ::= "id" "(" Expression ")"

Inverse ::= Expression $^{"-1"}$

Première_projection ::= "prj_1" "("Expression "," Expression ")"

Deuxième_projection ::= "prj_2" "("Expression "," Expression ")"

Composition ::= Expression ";" Expression

Produit_direct ::= Expression "⊗"Expression

Produit_parallèle ::= Expression "||" Expression

Itération ::= Expression$^{\text{Expression}}$

Fermeture_réflexive ::= Expression$^{"*"}$

Fermeture ::= Expression$^{"+"}$

Domaine ::= "dom" "(" Expression ")"

Codomaine ::= "ran" "(" Expression ")"

Image ::= Expression "[" Expression "]"

Restriction_domaine ::= Expression "◁" Expression

Soustractions_domaine ::= Expression "⩤" Expression

Restriction_codomaine ::= Expression "▷" Expression

Soustraction_codomaine ::= Expression "⩥" Expression

Surcharge ::= Expression "$^+$" Expression

Fonction_partielle ::= Expression "+ >" Expression

Fonction_totale ::= Expression "− >" Expression

Injection_partielle ::= Expression "$> + >$" Expression

Injection_totale ::= Expression "$> - >$" Expression

Surjection_partielle ::= Expression "$+ >>$" Expression

Surjection_totale ::= Expression "$- >>$" Expression

Bijection_partielle ::= Expression "$>> + >>$" Expression

Bijection_totale ::= Expression "$>> - >>$" Expression

Lambda_expression ::= "λ" Liste_ident "." "(" Prédicat "|" Expression ")"

Évaluation_fonction ::= Expression "(" Expression ")"

Transformée_fonction ::= "fnc" "(" Expression ")"

Transformée_relation ::= "rel" "(" Expression ")"

Suites ::= "seq" "(" Expression ")"

Suites_non_vide ::= "seq_1" "(" Expression ")"

Suites_injectives ::= "iseq" "(" Expression ")"

Suites_inj_non_vide ::= "iseq_1" "(" Expression ")"

Permutations ::= "perm" "(" Expression ")"

Suite_vide ::= "[]"

Suite_extension ::= "[" Expression^{+}"," "]"

Taille_suite ::= "size" "(" Expression ")"

Premier_élément_suite ::= "first" "(" Expression ")"

Dernier_élément_suite ::= "last" "(" Expression ")"

Tête_suite ::= "front" "(" Expression ")"

Queue_suite ::= "tail" "(" Expression ")"

```
Inverse_suite ::= "rev" "(" Expression ")"

Concaténation ::= Expression "^" Expression

Insertion_tête ::= Expression "->" Expression

Insertion_queue ::= Expression "←" Expression

Restriction_tête ::= Expression "↑" Expression

Restriction_queue ::= Expression "↓" Expression

Concat_généralisée ::= "conc" "(" Expression ")"

Arbres ::= "tree" "(" Expression ")"

Arbres_binaires ::= "btree" "(" Expression ")"

Construction_arbre ::= "const" "(" Expression "," Expression ")"

Racine_arbre ::= "top" "(" Expression ")"

Fils_arbre ::= "sons" "(" Expression ")"

Aplatissement_préfixé ::= "prefix" "(" Expression ")"

Aplatissement_postfixé ::= "postfix" "(" Expression ")"

Taille_arbre ::= "sizet" "(" Expression ")"

Symétrie_arbre ::= "mirror" "(" Expression ")"

Rang_noeud ::= "rank" "(" Expression "," Expression ")"

Père_noeud ::= "father" "(" Expression "," Expression ")"

Fils_noeud ::= "son" "(" Expression "," Expression "," Expression ")"

Sous_arbre_noeud ::= "subtree" (" Expression "," Expression ")"

Arité_noeud ::= "arity" "(" Expression "," Expression ")"

Arbre_binaire_en_extension ::= "bin" "(" Expression [ "," Expression "," Expression
```

```
Sous_arbre_gauche ::= "left" "(" Expression ")"

Sous_arbre_droit ::= "right" "(" Expression ")"

Aplatissement_infixé ::= "infix" "(" Expression ")"
```

Substitutions

```
Substitution ::=
  Substitution_bloc
  | Substitution_identité
  | Substitution_devient_égal
  | Substitution_précondition
  | Substitution_assertion
  | Substitution_choix_borné
  | Substitution_conditionnelle
  | Substitution_sélection
  | Substitution_cas
  | Substitution_choix_non_borné
  | Substitution_définition_locale
  | Substitution_devient_elt_de
  | Substitution_devient_tel_que
  | Substitution_variable_locale
  | Substitution_séquence
  | Substitution_appel_opération
  | Substitution_simultanée
  | Substitution_tant_que

Substitution_corps_opération ::=
  Substitution_bloc
  | Substitution_identité
  | Substitution_devient_égal
  | Substitution_précondition
  | Substitution_assertion
  | Substitution_choix_borné
  | Substitution_conditionnelle
  | Substitution_sélection
  | Substitution_cas
  | Substitution_any
  | Substitution_let
  | Substitution_devient_elt_de
```

```
  | Substitution_devient_tel_que
  | Substitution_variable_locale
  | Substitution_appel_opération

Substitution_bloc ::=
  "BEGIN" Substitution "END"
  Substitution_identité ::=
  "skip"

Substitution_devient_égal ::=
  Ident_ren+"," ":=" Expression+","
  | Ident_ren "(" Expression +"," ")" ":=" Expression
  | Ident_ren ("'" Ident )+ ":=" Expression

Substitution_précondition ::=
  "PRE" Prédicat "THEN" Substitution "END"

Substitution_assertion ::=
  "ASSERT" Prédicat "THEN" Substitution "END"

Substitution_choix_borné ::=
  "CHOICE" Substitution ( "OR" Substitution )* "END"

Substitution_conditionnelle ::=
  "IF" Prédicat "THEN" Substitution
  ( "ELSIF" Prédicat "THEN" Substitution )*
  [ "ELSE" Substitution ]
  "END"

Substitution_sélection ::=
  "SELECT" Prédicat "THEN" Substitution
  ( "WHEN" Prédicat "THEN" Substitution )*
  [ "ELSE" Substitution ]
  "END"

Substitution_cas ::=
  "CASE" Expression "OF"
  "EITHER" Terme_simple+"," "THEN" Substitution
  ( "OR" Terme_simple+"," "THEN" Substitution )*
  [ "ELSE" Substitution ]
  "END"
  "END"

Substitution_choix_non_borné ::=
```

```
  "ANY" Ident+"," "WHERE" Prédicat "THEN" Substitution "END"

Substitution_définition_locale ::=
  "LET" Ident+"," "BE"
  ( Ident "=" Expression )+"."
  "IN" Substitution "END"

Substitution_devient_elt_de ::=
  Ident_ren+"," ":∈" Expression

Substitution_devient_tel_que ::=
  Ident_ren+"," ":" "(" Prédicat ")"

Substitution_variable_locale ::=
  "VAR" Ident+"," "IN" Substitution "END"

Substitution_séquence ::=
  Substitution ";" Substitution

Substitution_appel_opération ::=
  [ Ident_ren+"," "←" ] Ident_ren [ "(" Expression+"," ")" ]

Substitution_simultanée ::=
  Substitution "||" Substitution

Substitution_corps_opération ::=
  Substitution_bloc
  | Substitution_identité
  | Substitution_devient_égal
  | Substitution_précondition
  | Substitution_assertion
  | Substitution_choix_borné
  | Substitution_conditionnelle
  | Substitution_sélection
  | Substitution_cas
  | Substitution_any
  | Substitution_let
  | Substitution_devient_elt_de
  | Substitution_devient_tel_que
  | Substitution_variable_locale
  | Substitution_appel_opération

Instruction_corps_opération ::=
  Instruction_bloc
```

```
  | Instruction_variable_locale
  | Substitution_identité
  | Instruction_devient_égal
  | Instruction_appel_opération
  | Instruction_conditionnelle
  | Instruction_cas
  | Instruction_assertion
  | Substitution_tant_que
```

Règles de syntaxe utiles

```
Liste_ident ::= Ident
  | "(" Ident+"," ")"

Ident_ren ::= Ident+"."

Typage_donnée_abstraite ::=
  Ident+"," "∈" Expression+"×"
  | Ident "⊆" Expression
  | Ident "⊂" Expression
  | Ident+"," "=" Expression+","

Typage_cte_concrète ::=
  Ident+"," "∈" Typage_appartenance_donnée_concrète+"×"
  | Ident "=" Typage_égalité_cte_concrète
  | Ident "⊆" Ensemble_simple
  | Ident "⊂" Ensemble_simple

Typage_appartenance_donnée_concrète ::=
  Ensemble_simple
  | Ensemble_simple+"×" "-->" Ensemble_simple
  | Ensemble_simple+"×" ">-->" Ensemble_simple
  | Ensemble_simple+"×" "-->>" Ensemble_simple
  | Ensemble_simple+"×" ">>-->>" Ensemble_simple
  | "" Terme_simple+"," ""
  | "struct" "(" (Ident ":" Typage_appartenance_donnée_concrète)+"," ")"

Typage_égalité_cte_concrète ::=
  | Terme
  | Expr_tableau
  | Intervalle
```

```
  | Ensemble_entier_B0
  | "rec" "(" ( [ Ident ":" ] Terme )+"," ")"

Ensemble_simple ::=
  Ensemble_entier_B0
  | "BOOL"
  | Intervalle_B0
  | Ident

Ensemble_entier_B0 ::=
  "NAT"
  | "NAT1"
  | "INT"

Expr_tableau ::=
  Ident
  | "" ( Terme_simple+"↦" "↦" Terme ) +"," ""
  | Ensemble_simple +"×" "×" "" Terme ""

Intervalle_B0 ::=
  Expression_arithmétique ".." Expression_arithmétique
  | Ensemble_entier_B0

Typage_var_concrète ::=
  Ident+"," "." Typage_appartenance_donnée_concrète+"×"
  | Ident "=" Terme

Typage_appartenance_donnée_concrète ::=
  Ensemble_simple
  | Ensemble_simple+"×" "- >" Ensemble_simple
  | Ensemble_simple+"×" ">- >" Ensemble_simple
  | Ensemble_simple+"×" "- >>" Ensemble_simple
  | Ensemble_simple+"×" ">>- >>" Ensemble_simple
  | "" Terme_simple+"," ""
  | "struct" "(" (Ident ":" Typage_appartenance_donnée_concrète)+"," ")"

Typage_param_entrée ::=
  Ident+"," "∈" Typage_appartenance_param_entrée+"×"
  | Ident "=" Terme

Typage_appartenance_param_entrée ::=
  Ensemble_simple
  | Ensemble_simple +"×" "- >" Ensemble_simple
  | Ensemble_simple +"×" ">- >" Ensemble_simple
```

```
  | Ensemble_simple +"×" "- >>" Ensemble_simple
  | Ensemble_simple+"×" ">> - >>" Ensemble_simple
  | "" Terme_simple+"," ""
  | "struct" "(" (Ident ":" Typage_appartenance_donnée_concrète)+"," ")"
  | "STRING"

Typage_param_mch ::=
  | Ident+"," "∈" Typage_appartient_param_mch+"×"
  | Ident+"," "=" Terme+","

Typage_appartient_param_mch ::=
  Ensemble_entier
  | "BOOL"
  | Intervalle_BO
  | Ident

Ensemble_entier ::=
  "ℤ"
  | "ℕ"
  | "ℕ1"
  | "NAT"
  | "NAT1"
  | "INT"
```

Grammaire des types B

```
Type ::= Type_de_base
  | "ℙ" "(" Type ")"
  | Type "×" Type
  | "struct" "(" (Ident ":" Type)+"," ")"
  | "(" Type ")"

Type_de_base ::=
  "ℤ"
  | "BOOL"
  | "STRING"
  | Ident
```