N° d'ordre :...../Faculté des sciences/UMBB 2012

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE.
MINISTERE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE.
UNIVERSITÉ M'HAMED BOUGARA - BOUMERDES
FACULTÉ DES SCIENCES

Mémoire de Magister

(École doctorale)

Présenté par :
Lila BELKACEMI.

Filière : Systèmes Informatiques et Génie des Logiciels
Spécialité : Spécification Logiciels et Traitement de l'Information

---

Formalisation de la logique temporelle dans Coq

Devant le jury :

M. Mohammed AHMED NACER	Professeur	USTHB	Président
M. Mohammed MEZGHICHE	Professeur	UMBB	Rapporteur
Mme. Thouraya TEBIBEL	MCA	ESI	Examinateur
M. Ahmed AIT BOUZIAD	MCB	UMBB	Examinateur

Année universitaire : 2012/2013

Remerciements

Arrivée au terme de mon travail de magister, je souhaite adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce travail.

J'adresse, donc, mes profonds remerciements à M. Mohammed MEZGHICHE, professeur à l'UMBB, qui m'a orienté, encouragé et soutenu depuis le début de ma formation en post graduation.

Ensuite je tiens à remercier tout particulièrement M. Mohammed CHAABANI pour son aide précieuse, ses encouragements et ses conseils prodigues. Ce travail n'aurait jamais abouti sans son précieux soutien et sa disponibilité.

Je présente mes respects et mes remerciements aux membres du jury qui m'ont fait l'honneur d'avoir accepté d'évaluer ce travail.

Je souhaite ensuite remercier ma famille et mes proches pour la patience dont ils ont fait preuve tout au long de ces années, ainsi que pour le soutien sans faille qu'ils m'ont accordé. Mes pensées vont plus particulièrement à mes parents pour le soutien qu'ils m'ont apporté dans les moments les plus difficiles.

Enfin, merci à tous ceux qui, de près ou de loin, ont contribué à la réalisation de ce travail.

Résumé

L'objectif de ce travail est de fournir une formalisation de la syntaxe ainsi que la sémantique de la logique temporelle linéaire propositionnelle($\mathcal{LTLP}$) en utilisant le langage de spécification GALLINA de l'assistant de preuves Coq. Par la suite, nous donnerons une vérification formelle de la terminaison de l'algorithme du tableau sémantique pour cette logique dans l'assistant de preuve Coq

Mots clés : Logique temporelle, formalisation, assistant de preuves Coq, méthode des tableaux sémantiques, terminaison.

Abstract

The objective of this work is to provide a formalisation of the syntax and the semantic of the propositional linear temporal logic ($\mathcal{PLTL}$) while using the language of specification GALLINA of the Coq proof assistant. Thereafter, we will give a formal verification of the termination of the algorithm of the semantic tableau for this logic in the Coq proof assistant.

Keywords : Temporal logic, formalisation, the Coq proof assistant, semantic tableau method, termination.

ملخص

يهدف هذا العمل إلى اقتراح صيغة شكلية للمنطق الزمني الخطي والافتراضي في مساعد الحجج "كوك". و في هذا الإطار نستعمل لغة التخصيص "قلينا" لمساعد الحجج "كوك". و من ثم سوف نقوم بتقديم مراجعة شكلية لمآل لوغاريتم الجدول المفاهيمي لهذا المنطق في إطار مساعد الحجج "كوك".

الكلمات الأساسية : المنطق الزمني, الصيغة الشكلية, مساعد الحجج "كوك", طريقة الجداول المفاهيمية , مآل.

Table des matières

Liste des Figures

Liste des Tableaux

Introduction générale

Aujourd'hui, l'informatique est appliquée à la plupart des activités humaines, y compris les plus critiques. Elle est utilisée dans, par exemple, l'automatisation du transport ferroviaire, les centrales nucléaires, la gestion de paiements électroniques. Dans ce genre d'application les enjeux sont énormes, une erreur dans le logiciel peut être très coûteuse, comme en témoigne la destruction de la fusée *Ariane 5* lors de son premier vol en 1996 [01] ou peut coûter des vies humaines comme le prouvent la défaillance d'un antimissile *Patriot* en 1991 [02] et l'overdose de radiation à l'Instituto Oncologico Nacional à Panamá en 2000 [03].

Pour parer à de tels scénarios catastrophes, il est plus que nécessaire de mettre en oeuvre des méthodes destinées à la spécification et la vérification de logiciel. Les méthodes traditionnelles utilisées pour détecter les erreurs contenue dans un programme ne sont pas suffisantes pour démontrer que ce dernier est correct, c'est-à-dire qu'il ne contient pas d'erreurs. Il faut donc faire appel à des approches plus rigoureuses qui garantissent de façon absolue l'absence de toute défaillance dans certains logiciels. C'est ce que proposent les *méthodes formelles*. Ces méthodes, souvent basées sur la logique mathématique, permettent de formaliser la sémantique des systèmes et offrent un cadre pour les vérifier formellement.

Il existe essentiellement deux approches dans le domaine de la vérification formelle : celle basée sur les modèles (*model-checking*) [04] pour des systèmes à ensemble d'états fini, et celle basée sur la preuve de théorèmes (*theorem-proving*) qui permettent, à la différence du model checking, de traiter des systèmes ayant un nombre infini d'états. Cette approche consiste à spécifier et à démontrer les propriétés du système à l'aide d'un assistant de preuves tels que PVS [05], HOL [06] et Coq [07] que nous avons utilisé dans notre travail.

Les logiques temporelles sont des extensions du calcul propositionnel par adjonction d'opérateurs temporels qui représentent en fait des paquets de quantificateurs sur les états et les séquences d'exécution d'un système. Il a été démontré que les logiques temporelles sont de bon formalismes pour la spécification et la vérification des programmes parallèles et des systèmes rèactifs. De plus, elles sont très bien adaptée pour exprimer des propriétés sur ces systèmes tels que la terminaison, l'équité, l'absence de blocage et bien d'autres.

Depuis quelques années un certain nombre de logiques temporelles ont été proposé. Parmi ces logiques, on distingue [04] les logiques temporelles linéaires, comme LTL (Linear Temporal Logic), et les logiques temporelles arborescentes, comme CTL(Computational Tree Logic). Chacune a un pouvoir d'expression différent. Les premières spécifient des propriétés sur l'ensemble des exécutions d'un système considérées individuellement, c'est à dire qu'elles ne considèrent pas la

façon dont les exécutions sont organisées en arbre. Les secondes permettent d'exprimer des propriétés prenant en compte l'aspect arborescent des exécutions.

L'assistant de preuve Coq a été choisi comme méta-langage pour formaliser différentes théories logiques et ce grâce à sa richesse de types (types inductifs, types dépendants). A titre d'exemple, citons la formalisation du raisonnement géométrique par Julien NARBOUX [08] et de la logique de description ALC par M.CHAABANI, M.MEZGHICHE et M.STRECHER [09]. Notre travail vient pour prolonger ces travaux en offrant une formalisation de la logique temporelle linéaire propositionnelle ($\mathcal{LTLP}$) dans Coq.

Parmi plusieurs méthodes utilisées comme des procédures de décision pour les logiques temporelles, on cite la méthode des *tableaux sémantiques* [10]. Cette méthode consiste en la recherche systématique d'un modèle d'une formule donnée. Le fait d'imposer une valeur à une formule peut déterminer univoquement la valeur des composants de la formule, ou au contraire laisser plusieurs choix possibles. La recherche systématique d'un modèle conduit à la construction progressive d'une structure arborescente particulière, appelée *tableau sémantique*. Cette méthode vient faire face au problème de l'explosion combinatoire lié à l'utilisation des tables de vérités pour vérifier la satisfiabilité ou la validité d'une formule.

Ce document est organisé comme suit : le chapitre 1 est consacré à la présentation de l'assistant de preuve Coq. La logique temporelle linéaire propositionnelle est détaillée dans le Chapitre 2. Dans le chapitre 3, nous présentons le tableau sémantique pour la logique temporelle linéaire. Le problème de terminaison des systèmes de réécriture est étudié au quatrième chapitre. Dans le dernier chapitre, nous décrivons la formalisation de la $\mathcal{LTLP}$ et la preuve de terminaison de l'algorithme du tableau sémantique pour cette logique dans Coq.

Chapitre 1

L'assistant de preuve Coq

Dans ce chapitre nous présentons Coq, l'assistant de preuves que nous avons utilisé. Une première partie est consacrée à la présentation de techniques de vérification formelles : model-cheking et theorem proving. Nous introduisons également quelques notions importantes comme le λ-calcul, l'isomorphisme de Curry-Howard, la sémantique de Heyting, nous présentons ensuite le système de preuves.

1.1 Approches de vérification formelle

La vérification formelle consiste à établir les propriétés souhaitables d'un système informatique et à vérifier leur satisfaction [11]. Actuellement, la vérification s'effectue par deux méthodes différentes [11] [12] [13] : la vérification de modèle (model checking) et la preuve de théorèmes (theorem proving).

1.1.1 Model-checking

Le *model-checking* est une technique de vérification de logiciel formelle qui consiste à énumérer toutes les situations possibles auxquelles peut mener le programme et à montrer la validité de propriétés dans chacune de ces situtations [12], ainsi on s'assure qu'aucune de ces situations n'est en contradiction avec les comportements que l'on désirait.

L'avantage majeur de cette approche est sa grande automatisation. Le processus de vérification de modèle permet de décider de la validité des propriétés à vérifier sans avoir besoin d'aucune intervention d'utilisateur. Le rôle de l'utilisateur se réduit à modéliser le système, spécifier les propriétés à vérifier et finalement lancer le calcul. Cette approche a aussi l'avantage de pouvoir exhiber un modèle de comportement invalidant la propriété à vérifier(un contre-exemple) quand celle-ci est insatisfaite.

En pratique, cette méthode est souvent utilisée dans la vérification des systèmes possédant un nombre fini d'états. Quand on considère les systèmes avec un nombre infini d'états, ce qui est souvent le cas pour les logiciels, le model-cheking ne permet pas d'énumérer toutes les situations possibles car cela conduit rapidement à un dépassement des capacités de l'ordinateur de stockage en mémoire. Cela constitue le premier inconvénient de la méthode. Le deuxième point faible

de cette méthode est lié à la fiabilité. En effet, nous devons faire confiance au processus de la vérification de modèle car aucun témoin de la correction des propriétés n'est retourné par ce processus. En travaillant avec des systèmes qui sont de plus en plus complexes et dont la correction est cruciale, une telle vérification n'est pas tout à fait satisfaisante.

1.1.2 Preuve de théorèmes

Introduite par Hoare [13]. Cette méthode consiste à spécifier et à démontrer les propriétés du système à l'aide d'outils logiciels appelés *assistants de preuves* tels que Coq, HOL, Isabelle, LEGO, NuPRL. Toute preuve est effectuée pas à pas par l'utilisateur mais sa validité est garantie par ce système. Cette approche a l'avantage d'être indépendante de la taille de l'espace d'états du système, et peut donc s'appliquer sur des modèles avec un très grand nombre d'états, potentiellement indéterminé ou infini. Néanmoins, son inconvénient est qu'elle nécessite l'intervention et la créativité de l'utilisateur pour compléter la preuve, ce qui est parfois un processus assez laborieux. Un autre inconvénient est son incapacité à fournir un contre-exemple lorsque la propriété à vérifier est prouvée insatisfaite.

1.2 Les assistants de preuves

Au cours de ces dernières années, beaucoup de travaux ont porté sur la conception et la mise en oeuvre d'assistants de preuves. Ces efforts ont été concrétisés par la réalisation de systèmes tels que HOL, PVS, Coq.

En informatique, un assistant de preuve est un programme qui permet d'écrire des théories mathématiques composées de définitions, d'axiomes et de théorèmes et de prouver ces derniers formellement [05]. La majorité des assistants sont écrits en langage de programmation fonctionnelle comme *Lisp* et *ML*. Pour réaliser une preuve, l'utilisateur doit formaliser le problème en utilisant le langage de l'assistant de preuves. Différents choix sont disponibles pour le langage : principalement la logique du premier ordre (classique ou intuitionniste), ou la logique d'ordre supérieur. La preuve peut être décrite de différentes manières : en donnant des étapes d'inférences, par l'application de tactiques ou stratégies, en démontrant des sous-preuves ou lemmes, etc. Le système vérifie que chaque étape donnée par l'utilisateur dans la preuve est correcte. La partie du système chargée de garantir qu'une preuve est correcte s'appelle *le vérificateur de preuves*.

Les assistants de preuves sont utilisés dans plusieurs travaux de certification. Par exemple, le système *Isabelle/HOL* est utilisé dans le projet *Verisoft* [14] qui ambitionne la vérification complète des systèmes informatiques incluant matériels, systèmes d'exploitations et applications utilisateurs. Le système PVS a été utilisé pour prouver la correction d'un algorithme pour la détection et la résolution des conflits dans l'espace aérien[15]. L'assistant de preuve Coq est utilisé pour extraire des programmes Ocaml certifiés[16] et prouver des propriétés de programmes numériques partant du code en C[17].

Une objection à l'usage d'assistants de preuve est que, outre l'entraînement demandé pour les utiliser, les interfaces d'aide à la preuve ne sont pas conviviales

pour tout le travail qui ne se fait pas automatiquement. Pour remédier à ce problème, les partisans des méthodes formelles travaillent à minimiser l'interaction avec les assistants de preuves jusqu'à ce que, idéalement, aucune interaction ne soit requise. Une autre objection est que la sécurité des preuves obtenues repose sur le bon fonctionnement de l'assistant. En effet, les assistants de preuves sont de gros logiciels complexes, dont on peut soupçonner qu'ils soient eux-même bogués. Un assistant de preuve bogué peut permettre de démontrer l'absurde.

1.3 Le lambda calcul

Les langages fonctionnels tels qu'OCaml, Haskell ou Lisp sont [18] des langages dérivés du lambda-calcul. Le lambda-calcul (ou λ-calcul) [19][20] est un langage de programmation introduit par *Alonzo Church* dans les années 30. Nous rappelons dans ce qui suit la syntaxe du λ-calcul :

Soit $V = \{x,y, ...\}$ un ensemble infini dénombrable de variables de programme, l'ensemble des λ-termes est constitué de la façon suivante :

- Si x est une variable, alors c'est également un λ-terme ;
- Si x est une variable et u un λ-terme déjà construit, alors $\lambda x.u$ est un λ-terme, nous appellons un tel λ-terme une *abstraction*. Intuitivement, $\lambda\ x.u$ est la fonction qui envoie x vers u ;
- Si u,v sont des λ termes déja construits, alors uv(la juxtaposition des termes) est un λ-terme, nous appellons un tel λ-terme une *application*.

On impose souvent au λ-calcul un système de types. Il s'agit d'associer aux termes du λ-calcul un type. On impose ensuite de ne construire de termes que si on peut leur associer un type.

Citons, par exemple, le *λ-calcul simplement typé*. On part d'un ensemble de types atomiques et on type les termes du λ-calcul à l'aide des règles suivantes :

- $\dfrac{\Gamma,\ x{:}U\ \vdash\ v{:}V}{\Gamma\ \vdash\ \lambda x.v{:}U{\to}V}$ (Abs)
- $\dfrac{\Gamma\ \vdash\ u{:}V{\to}T, \Gamma\ \vdash\ v{:}V}{\Gamma\ \vdash\ uv{:}T}$ (App)

$U \to V$ est le type des fonctions d'un type U vers un type V.

- La règle d'abstraction (Abs) autorise à former une fonction $x.v$ d'un type U vers un type V si on peut créer un terme v de type V en utilisant le terme x supposé de type U.
- La règle d'application (App) autorise à appliquer un argument de type V à une fonction du type V vers le type T. On obtient alors un terme de type T.

1.4 La logique constructive(intuitionniste)

La logique classique, classique par opposition à la logique intuitionniste, ne construit pas les vérités qu'elle démontre, elle les découvre. Elle pré-suppose que les choses sont soit vraies, soit fausses. Cela se traduit par l'axiome du tiers-exclu :

A $\vee \neg$A dont le sens littéral est qu'étant donnée une proposition, celle ci est vraie ou bien sa négation est vraie.

Les intuitionnistes rejettent le caractère non constructif de la logique classique qui est matérialisé par l'axiome du tiers-exclu. Dans la logique intuitionniste(ou logique constructive), la vérité n'est pas découverte grâce à la preuve mais construite avec la preuve. Ainsi dire qu'une formule A est soit vraie soit fausse n'a de sens que si l'on a prouvé qu'elle était vraie ou bien si l'on a prouvé qu'elle était fausse. D'où le rejet de l'axiome du tiers-exclu.

1.5 Heyting-Kolmogorov et Curry-Howard

1.5.1 La sémantique de Heyting et Kolmogorov

Les intuitionnistes *Heyting et Kolmogorov* définissent ce qu'est une preuve. Cette définition des preuves est appelée la *sémantique des preuves intuitionnistes de Heyting-Kolmogorov* [16] [21]. Les preuves sont définies en termes de processus effectivement calculables au même titre que les entiers ou les fonctions.

La sémantique de *Heyting* et *Kolmogorov* se définit comme suit :

- La preuve d'une négation $\neg A$, i.e. *non* A, est une fonction qui transforme toute preuve de A en une preuve d'une contradiction.
- La preuve d'une conjonction $A \wedge B$, i.e. A *et* B, est formée d'une preuve de A et d'une preuve de B.
- La preuve d'une disjonction $A \vee B$, i.e. A *ou* B, est formée d'une preuve de A ou d'une preuve de B plus une indication permettant de savoir lequel a été prouvé.
- La preuve d'une implication, $A \rightarrow B$, i.e. A *implique* B, est une fonction qui construit, à partir d'une preuve (quelconque) de A, une preuve de B.
- La preuve d'une quantification existentielle, $\exists x.A(x)$, est formée d'un élément du domaine de quantification et d'une preuve de $A(t)$.
- Une preuve d'une quantification universelle, $\forall xA.P(x)$, est une fonction qui prend en argument un élément quelconque de A et rend une preuve de $P(x)$.

1.5.2 L'isomorphisme de Curry-Howard

L'isomorphisme de Curry-Howard établit la relation entre les λ-termes simplement typés et les démonstrations en déduction naturelle. Pour illustrer celà, nous mettons les règles d'inférence dans le système de déduction naturelle et les règles de typage dans le λ-calcul côte à côte comme suit :

Déduction naturelle

$$\frac{U \in \Gamma}{\Gamma \vdash U}(\text{Axiome})$$

$$\frac{\Gamma, U \vdash V}{\Gamma \vdash U \Rightarrow V}(\text{Introduction})$$

$$\frac{\Gamma \vdash V \Rightarrow T, \Gamma \vdash V}{\Gamma \vdash T}(\text{Elimination})$$

Lambda calcul simplement typé

$$\frac{(x{:}U) \in \Gamma}{\Gamma \vdash x{:}U}(\text{Var})$$

$$\frac{\Gamma, x{:}U \vdash v{:}V}{\Gamma \vdash \lambda x.v{:}U \rightarrow V}(\text{Abs})$$

$$\frac{\Gamma \vdash u{:}V \rightarrow T, \Gamma \vdash v{:}V}{\Gamma \vdash uv{:}T}\ (\text{App})$$

Nous constatons que les règles de typage du λ-calcul simplement typé correspondent précisément, si on efface les termes pour ne conserver que les types, aux règles de la déduction naturelle. Autrement dit, si l'on utilise le λ-calcul pour interpréter la logique propositionnelle alors une proposition est interprétée par un type. Ainsi, on a le paradigme *proposition = type, preuve = fonction(programme)* appelé *isomorphisme de Curry-Howard.*

1.6 L'assistant de preuve Coq

Coq [07] [22] est un assistant de preuves interactif. Il est développé [23] à l'INRIA(Institut National de Recherche en Informatique et Automatique), à l'École polytechnique et à l'université de Paris XI, dans le cadre du projet TypiCal. La première implémentation [24] réalisée en CAML par *Gérard Huet* et *Thierry Coquand* est commencée en 1984 et distribuée en 1989. Coq est bâti directement sur la correspondance de Curry-Howard : toutes les preuves sont en interne représentées comme des termes du *Calcul des Constructions Inductives(CCI)*, une dérivé du λ-calcul. Les termes du CCI sont peu différents des expressions mathématiques courantes, dans leur contenu et dans leur syntaxe. Mais ils sont représentés avec une syntaxe plutôt exotique en Coq, en raison de son interface ASCII.

Coq peut être vu comme trois sous systèmes, le premier étant un langage de spécification appelé *GALLINA*, le second un assistant de preuves pour les démonstrations des formules de la logique d'ordre supérieur et le troisième, un outil d'extraction de programmes certifiés sans erreurs et qui utilise les services des deux premiers systèmes.

1.6.1 Le langage de spécification

Les preuves développées par les utilisateurs se portent sur des objets mathématiques de différentes natures : prédicats, ensembles, variables .... Donc avant d'entamer une preuve, il faut tous d'abord définir tous les objets sur lesquels se portera celle-ci. Le système Coq dispose d'une interface via laquelle l'utilisateur introduit toutes définitions. Ceci en utilisant un langage précis, comportant un certain nombre de commandes et de conventions syntaxiques. Les développeurs du système Coq ont choisi *GALLINA* pour nom de ce langage de commandes. La syntaxe de ce langage est détaillée dans le manuel de Coq [07]. Nous nous contentons d'aborder ici certains points liés à notre travail, à travers une suite d'exemples.

Types et sortes

Dans le CCI, on appelle *sorte* le type d'un type(considéré en tant que terme). Coq distingue deux sortes de base *Set* et *Prop* :

- ***Set*** : la sorte des types de données destinée à contenir tous les objets ayant un contenu calculatoire,

- ***Prop*** : la sorte des propriétés, pour rôle de contenir tout ce qui a trait à la logique pure, comme par exemple les justifications diverses, pré- et post-conditions, autrement dit tout ce qui peut être ignoré pendant des calculs.

La commande `"Check"` permet d'obtenir le type d'un objet en Coq :

```
Check nat.
nat
   :set

Check le.
le
   :nat -> nat -> Prop
```

Par exemple, la sorte de `nat` (type des naturels) est `Set` tandis que celui de la relation `le` (inférieur ou égal à) entre deux naturels est `Prop`.

```
Check Set.
Set
   :Type

Check Prop.
Prop
    :Type

Check Type.
Type
    :Type
```

Le type de `Set` et `Prop` est `Type`. D'une manière précise, il existe une suite infinie de *types{Type(i)/i∈N}* telle que *Prop : Type(0), Set : Type(0) et Type(i) : Type(j)* pour $i < j$.

Les types inductifs

Les *types inductifs* permettent de définir par récurrence des ensembles de termes. A chaque type de données inductif correspond une structure de calcul, basée sur le filtrage et la récursion. Nous citons ci-dessous quelques exemples de types inductifs prédéfinis en Coq.

Exemple 1.6.1. *Les booléens.*

```
Inductive bool: Set := true: bool | false: bool.
  bool is defined
  bool_ind is defined
  bool_rec is defined
  bool_rect is defined
```

Cette déclaration crée un nouveau type, nommé `bool`, ainsi que deux constructeurs `true` et `false` de type `bool`. Et à partir de cette définition, Coq génère automatiquement trois principes de récurrence `bool_ind, bool_rec, bool_rect` respectivement pour les sortes *Prop*, *Set* et *Type*.

Exemple 1.6.2. *Les entiers de Peano.*

```
Inductive nat: Set := O: nat | S: nat -> nat.
  nat is defined
  nat_ind is defined
  nat_rec is defined
  nat_rect is defined
```

Cette définition introduit le type *nat* dans la sorte *Set*. Ce type possède deux constructeurs qui traduisent les deux manières dont sont construits les nombres naturels : soit on prend le nombre 0 (le constructeur `O`), soit on applique la fonction successeur à un nombre déjà construit (le constructeur `S`) et c'est dans le deuxième constructeur ou l'on trouve la récursivité. Par exemple le nombre naturel *(S(S 0))* contient le fragment *(S 0)* qui est lui même un nombre naturel.

Exemple 1.6.3. *Les listes*

```
Inductive list(A: Set): Set :=  nil: list A
                              | cons: A -> list A -> list A.
  list is defined
  list_ind is defined
  list_rec is defined
  list_rect is defined
```

La déclaration de paramètre `(A: Set)` introduit un type `A` dans la sorte *Set* et on peut donc utiliser l'identificateur dans la description des types pour les deux constructeurs `nil` et `cons`.

Les fonctions récursives

Pour définir une fonction récursive dans Coq, l'utilisateur doit déterminer les valeurs prises par la fonction dans un ordre précis qui suit l'ordre dans lequel sont construits les termes des types inductifs. Pour les nombres naturels, par exemple, on est obligé de construire 0 avant *(S 0)*, *(S 0)* avant *(S (S 0))* et ainsi de suite.

Pour la définition d'une fonction récursive sur les naturels, on utilise la valeur *(f 0)* pour définir la valeur *(f (S 0))*, la valeur *(f (S 0))* pour définir la valeur *(f (S (S 0)))* et de manière générale, on utilise la valeur *(f n)* pour définir la valeur *(f (S n))*.

La construction des fonctions récursives se fait dans Coq avec la commande `"Fixpoint"`.

Exemple 1.6.4. *Soit la fonction récursive* ***plus*** *qui calcule la somme de deux nombres naturels* ***n*** *et* ***m*** *:*

```
Fixpoint plus (n m: nat) struct n: nat :=
  match n with
    |O   => m
    |S p => S (plus p m)
end.
```

La syntaxe `Fixpoint` permet de définir un terme avec la construction *fix* du CCI. La représentation interne d'un objet en Coq peut être obtenue en utilisant la commande `"Print"` :

```
Print plus.
  plus =
    fix plus (n m : nat) : nat := match n with
      |O => m
      |S p => S (plus p m)
  end
      : nat -> nat -> nat
```

Les constantes

Comme la taille des spécifications en Coq augmente relativement vite, il est souhaitable de pouvoir factoriser les parties communes de la spécification dans un alias. Cet alias est une constante dans le système et est définie en Gallina par la syntaxe suivante :

```
Definition c := t.
  c is defined.
```

1.6.2 L'assistant de preuves

L'objectif est d'avoir un outil qui permet entre autres la démonstration automatique de théorèmes dans certains cas, mais aussi de dégager les logiciens et mathématiciens de la manipulation fastidieuse des formules logiques, des listes d'hypothèses ou des séquents, des symboles logiques et autres durant le développement de preuves.

Preuves et tactiques

Les preuves sont construites de manière interactive. La première étape de la construction d'une preuve consiste à entrer l'énoncé du théorème à prouver, et déclencher l'ouverture du mode d'édition de preuves de Coq. Celui-ci présente à l'utilisateur le but à prouver, en séparant les hypothèses de la conclusion par une barre horizontale. L'utilisateur construit sa preuve en partant de la conclusion, et en revenant aux hypothèses.

L'arbre de preuve est construit pas à pas, par applications de *tactiques* ; une *tactique* peut être définie comme une fonction qui à tout but b associe une suite (éventuellement vide) de nouveaux sous buts $b_1,\ldots,b_n$. De plus, une tactique doit permettre, à partir des solutions des b_i, de construire une solution pour b. Notons qu'une tactique peut échouer sur un but, soit par impossibilité de le décomposer en sous-buts, soit dans la phase de reconstruction de la solution.

Une fois la preuve terminée, elle est vérifiée par le système, puis, si elle est correcte, elle est sauvegardée et ajoutée à l'environnement courant. Le théorème ainsi prouvé peut désormais être utilisé dans des preuves ultérieures.

Quelques tactiques de base

Le système Coq est fourni avec un grand nombre de tactiques, nous présentons ci-dessous quelques une de ces tactiques :

- ***intro*** : la tactique `intro` permet d'introduire des hypothèses dans le contexte du but courant et de transformer un but quantifié universellement en un but sans quantification universelle.
- ***apply terme :*** cette tactique essaie de faire correspondre le but courant avec la conclusion du type de *term* (un terme dans le contexte local). Si elle réussit, elle retourne autant de sous buts qu'il y a de prémisses dans *term*.
- ***split*** : la tactique qui permet de décomposer un but dont l'énoncé est une conjonction est la tactique `split`. Elle permet de passer d'un but de la forme $A \wedge B$ à deux buts plus simple de la forme A et B.
- ***left*** : lorsque l'on cherche à prouver une expression de la forme $A \vee B$ il suffit parfois de prouver A. C'est cette étape de raisonnement qui est fournie par la tactique `left`. Cette tactique permet de remplacer un but de la forme $A \vee B$ par un seul but de la forme A.
- ***right*** : le comportement de cette tactique est exactement symetrique de celui de la tactique `left`. Elle permet de transformer un but de la forme $A \vee B$ en un but de la forme B.
- ***assumption*** : l'activation de la tactique `assumption` contrôle la convertibilité entre l'hypothèse et l'énoncé à prouver.
- ***genaralize*** : elle permet d'ajouter au but courant une quantification universelle. Si le but est de la forme *(p x)* et que x est un objet défini de type T, alors, `generalize` x transforme le but en : *forall x : t, (P x)*.
- ***exists :*** lorsque le but est de la forme exists x : T, A x, la tactique `exists` M (où M est un terme de type T) remplace le but courant par un but de la forme A M.
- ***Reflexivity :*** cette tactique résout tout but de la forme M1 = M2, où M1 et M2 sont deux termes convertibles au sens des règles de calcul de Coq, comme par exemple les termes 2 + 2 et 4. En particulier, elle résout tous les buts de la forme M = M.
- ***Absurd term*** : cette tactique permet de déduire une contradiction (*term* est de type *Prop*) et génère deux sous buts : *term* et *term*. En général, l'un de ces deux énoncés est une des hypothèses du but courant. Elle est utile dans les preuves par cas ou certains cas sont impossibles.
- ***induction term :*** elle permet de faire une preuve par induction. L'argument term doit être une constante inductive. Cette tactique génère un sous but pour chaque constructeur du type inductif concerné et remplace chaque occurrence de term dans la conclusion et les hypothèse du but en rajoutant au contexte local des hypothèses d'induction.
- ***auto :*** le principe de cette tactique est simple, `auto` utilise une base de données de tactiques(Hints), lesquelles sont appliquées au but initial, ainsi qu'à tous les sous buts engendrés, et ce répétitivement, jusqu'à la résolution de tous les buts. Si ces buts ne peuvent être tous résolus, `auto` annule tous ses efforts et laisse le but initial inchangé.
- ***unfold*** : la tactique `unfold` *id* remplace dans le but courant toutes les

occurrences de l'identificateur *id* par le corps de sa définition dans l'environnement courant.
- ***Qed*** : après avoir terminé la construction d'une preuve on doit l'enregistrer dans l'environnement pour des utilisations ultérieures. Cette tache est justement celle de la tactique `Qed`

Composition de tactiques

La construction interactive d'une preuve pourrait être très longue si elle devait se réduire à une suite d'activations de tactiques de bases. Coq propose une collection d'opérateurs permettant de composer les tactiques préexistantes pour en former de nouvelles. Ces opérateurs sont appelés tacticielles. On distingue deux variantes de composition :
- Composition simple : la composition simple permet d'enchainer l'application de plusieurs tactiques sans s'arrêter aux sous-buts intermédiaires, sous la forme $tac_1; tac_2; ...; tac_n$.
- Composition généralisée : la composition généralisée notée `tac;` [$tac_1|tac_2|...|tac_n$`]` s'apparente à la composition simple, excepté le fait que la tactique tac_i s'applique au i-éme sous-but(en supposant que la tactique `tac` engendre exactement n sous-buts)

Un exemple détaillé

Considérons la proposition $(A \to B) \to (B \to C) \to A \to C$, qui énonce que l'implication est transitive dans la logique propositionnelle. La démonstration de cette proposition dans Coq s'effectue en suivant les étapes suivantes :
- **Déclaration des variables propositionnelles** : on définit trois identificateurs de type `Prop`.

```
Variables A B C: Prop.
  A is assumed
  B is assumed
  C is assumed
```

- **Activation du système de buts** : la commande suivante annonce que l'on souhaite prouver un théorème de nom `imp_trans` précisant son énoncé. Un but initial est alors crée, composée du contexte courant et de la formule à prouver. L'affichage du but courant se fait en séparant la présentation du contexte de celle de l'énoncé à prouver par une barre horizontale.

```
Theorem imp_trans: (A -> B) -> (B -> C) -> A -> C.

  1 subgoal
 ----------------------------------    (1/1)
  (A -> B) -> (B -> C) -> A -> C
```

Après avoir définit les identificateurs et le but à prouver, on peut commencer l'élaboration de la preuve. La commande `Proof` est facultatif, et a pour but de signaler le début de la preuve.

- **Introduction des hypothèses** : comme vu précédemment, la tactique `intros` permet de réduire la tache de construction d'une preuve de la proposition considérée à celle d'une preuve de `C`, dans un contexte augmenté de trois hypothèses. En argument d'`intros`, nous précisons le nom que nous souhaitons donner à chacune des trois hypothèses.

```
intros D E G.
```

```
1 subgoal
A: Prop
B: Prop
C: Prop
D: A -> B
E: B -> C
G: A
------------------------------------    (1/1)
C
```

- **Application d'une hypothèse :** l'examen du but courant tel qu'affiché par Coq nous montre que l'énoncer à prouver est la conclusion de l'hypothèse `E`. La tactique `apply E` a pour effet de donner comme nouveau but courant la prémisse de `E`, c'est à dire la proposition `C`.

```
apply E.
```

```
1 subgoal
A: Prop
B: Prop
C: Prop
D: A -> B
E: B -> C
G: A
------------------------------------    (1/1)
B
```

- **Application d'une hypothèse :** le même raisonnement s'applique à la proposition `B` et à l'hypothèse `D`.

```
apply D.
```

```
1 subgoal
A: Prop
B: Prop
C: Prop
D: A -> B
```

```
E: B -> C
G: A
------------------------------------    (1/1)
A
```

- **Application d'une hypothèse directe :** on remarque que l'énoncé à prouver dans le but courant est exactement l'énoncé de l'hypothèse G. La tactique `assumption` reconnait ce fait, et réussit sans engendrer aucun nouveau sous-but, avec pour solution le terme G. Il ne reste plus aucun but à résoudre, la preuve est alors terminée, ce qui est signalé par le message `Proof completed`.

```
assumption.

 Proof completed.
```

- **Calcul et sauvegarde du terme de preuve :** une preuve termine par la commande `Qed`. Cette commande a pour effet de calculer le terme de preuve associé à l'enchaînement de tactiques ci-dessus, de vérifier que son type est bien l'énoncé à prouver, et de sauvegarder le nouveau théorème sous la forme d'une définition reliant le nom du théorème, son énoncé(son type), et le terme de preuve construit.

```
Qed.

 intros D E G.
 apply E.
 apply D.
 assumption.

 imp_trans is defined.
```

D'autres exemples

Exemple 1.6.5. $\forall A,\ \neg\neg\neg A \rightarrow \neg\ A$

```
•Parameter A : Prop.
  is assumed

•Theorem exemple1: (not (not (not A))) -> (not A).

   1 subgoal
  ------------------------------------    (1/1)
   ¬¬¬ A -> ¬A

•intro.
```

```
   1 subgoal
   A : Prop
   H : ¬¬¬A
  ------------------------------------    (1/1)
   ¬ A

•intro.
   1 subgoal
   A : Prop
   H : ¬¬¬A
   H0: A
  ------------------------------------    (1/1)
   False

•apply H.
   1 subgoal
   A : Prop
   H : ¬¬¬A
   H0: A
  ------------------------------------    (1/1)
   ¬¬A

•intro.
   1 subgoal
   A : Prop
   H : ¬¬¬A
   H0: A
   H1: ¬A
  ------------------------------------    (1/1)
   False

•apply H1.
   1 subgoal
   A : Prop
   H : ¬¬¬A
   H0: A
   H1: ¬A
  ------------------------------------    (1/1)
   A

•auto.
 Proof completed.

•Qed.
 intro.
 intro.
```

```
apply H.
intro.
apply H1.
intro.
auto.

exemple1 is defined
```

Exemple 1.6.6. *(x + y)(x - y) = x^2 - y^2.*

```
•Lemma exemple2: forall x y: Z, (x + y) * (x - y) = x * x - y * y.

  1 subgoal
  ----------------------------------    (1/1)
  forall x y : Z, (x + y) * (x - y) = x * x - y * y

•intros.

  1 subgoal
  x: Z
  y: Z
  ----------------------------------    (1/1)
  (x + y) * (x - y) = x * x - y * y

•ring.
 Proof completed.

•save.
 intros.
 ring.

 exemple2 is defined.
```

1.6.3 Extraction de programmes

L'autre grande application de la correspondance de Curry-Howard est l'extraction de programmes [25] . L'isomorphisme de Curry-Howard permet de traiter les preuves comme des objets de calcul et d'en extraire ainsi une partie algorithmique donc un programme. L'extraction des programmes est entièrement prise en charge par le systeme Coq. Elle s'effectue [16] en deux étapes :

- Extraction du λ-terme correspondant à la preuve après la suppression de l'information logique(qui nous assure que la démonstration est correcte).
- Traduction de ce λ-terme en un programme écrit en langage fonctionnel (Caml, Ocaml ou Haskell).

Chapitre 2

La logique temporelle

Ce chapitre est construit sur la base de [26]. Nous définissons en premier lieu ce qui est la logique modale puis nous présentons la logique temporelle qui est un type spécial de la logique modale. Dans ce chapitre nous nous intéressons essentiellement à la logique temporelle linéaire propositionnelle ($\mathcal{LTLP}$), une extension du calcul propositionnel, pour cela nous donnons un petit rappel de ce dernier . Nous décrivons la sémantique des opérateurs temporels et nous présentons quelques-unes de leurs propriétés.

2.1 La logique modale

La logique classique, c'est-à-dire la logique propositionnelle et la logique du premier ordre, sert à exprimer des énoncés auxquels on attribue une valeur de vérité : un énoncé est soit "vrai" soit "faux" et il n'y a pas d'autre valeur possible. Par exemple quand je dis que « les triangles sont des polygones à trois côtés » on peut sans problème attribuer la valeur "vrai" à cet énoncé.

Dans d'autres cas les propositions que l'on doit manipuler n'ont pas le même statut vis-à-vis du couple (vrai, faux). Ainsi la proposition « l'attaque de P. H. a entrainé la défaite des puissances de l'axe » utilise des éléments (l'attaque, la défaite) dont la véracité n'est pas en cause mais l'implication qu'elle affirme est du domaine de l'argumentation : les historiens peuvent l'admettre ou la contester ; il n'y a pas de sens à la déclarer "vraie" ou "fausse".

La logique classique est une logique frustre, d'utilisation restreinte et de pouvoir expressif très limité. Certaines propriétés des systèmes s'expriment difficilement en logique classique. Une des manières d'étendre le champs d'application de cette logique consiste à lui adjoindre des opérateurs de modalité. Ainsi complétée, la logique propositionnelle se transforme en *logique modale*.

La logique modale est l'étude des déductions possibles à partir d'expressions comme « *il est nécessaire que...* » ou « *il est possible que...* » [27]. Elle regroupe plusieurs logiques, comme la logique epistémique (« *il est connu que...* », « *il est exclu que...* », ...)., la logique déontique (« *il est obligatoire que...* », « *il est autorisé que...* », ...) ou la logique temporelle(« *Un jour dans le futur...* », « *Toujours dans le futur...* »).

Dans les logiques classiques, la valeur de vérité d'un énoncé est définie par

un état du monde. Pour savoir, par exemple, si l'énoncé « *Lila habite à Tizi Ouzou*» est vrai ou faux, il suffit de considérer l'état du monde où on se place. Dans les logiques modales, la valeur de vérité des énoncés modaux est définie par un ensemble d'états du monde. Par exemple, pour savoir si «*Lila a toujours habité à Tizi Ouzou* » est vrai ou faux, on doit considérer tous les états du monde auxquels réfère la modalité temporelle « *toujours* ». Pour chaque type de modalité cet ensemble de mondes doit être clairement défini. Ici, par exemple, on pourrait convenir que c'est l'ensemble des états du monde depuis la naissance de *Lila* jusqu'à l'instant où on se pose la question ; la valeur de vérité peut donc changer en fonction de cet instant.

Les logiques modales sont essentielles à l'intelligence artificielle et à la formalisation de domaines comme le droit, la temporalité, la connaissance, la croyance etc.

2.2 La logique temporelle

La logique temporelle [27] a été introduite par *Arthur Prior* en 1957 afin de traiter des informations temporelles dans un cadre logique proche de la logique modale. Techniquement parlant, la logique temporelle est fermement liée à la logique modale. Dans la logique temporelle, les mondes possibles deviennent des contextes temporels (qui peuvent être, soit des instants, soit des intervalles de temps, selon l'option sémantique choisie).

Les logiques temporelles sont utilisées dans la description de systèmes informatiques afin de spécifier des propriétés concernant le comportement dynamique du système. En ajoutant aux connecteurs de la logique propositionnelle des opérateurs temporels, elles permettent d'énoncer formellement des propriétés sur les enchaînements d'états, c'est-à-dire sur les exécutions du système étudié.

Les diverses logiques temporelles se distinguent selon qu'elles sont propositionnelles, du premier ordre ou d'ordre supérieur, qu'elles incluent ou non des opérateurs passés, qu'elles sont linéaires ou à branchement, que la représentation du temps est discrète ou continue.

On distingue principalement deux grandes familles selon la structure du temps considérée :

- ***Les logiques du temps linéaire*** [28] qui ont pour origine les travaux de *A. Prior (1960)*. Dans le cadre de ces logiques, le temps se déroule, comme son nom l'indique, linéairement : une succession d'états. Chaque état est caractérisé par l'ensemble des propositions atomiques vérifiées dans cet état. Autrement dit, on spécifie le comportement attendu d'un système sur les chemins individuels (issus de l'état initial) sachant que tout instant possède un et un seul successeur immédiat.
  Parmi les logiques temporelles du temps linéaire , on cite la Logique LTL(Linear Temporal Logic).
- ***Les logiques du temps arborescent*** [29] : contrairement aux logiques linéaires qui supposent que chaque instant dans le temps n'a qu'un seul futur, les logiques arborescentes supposent que plusieurs futurs sont possibles à chaque instant. Une formule de ces logiques est donc évaluée sur l'arbre

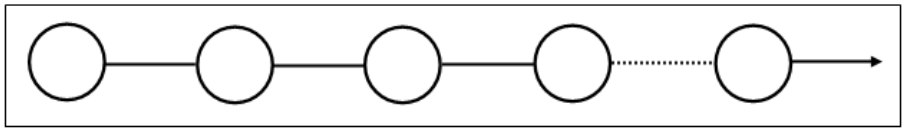

Fig 2.1 – Le déroulement temporel d'un système dans une logique linéaire

des exécutions du système à vérifier. Un état du système à vérifier satisfait une formule si l'arbre des exécutions possibles à partir de cet état satisfait la formule.
Parmi les logiques temporelles du temps arborescent, la plus connue et la plus étudiée est CTL(Computational Tree Logic).

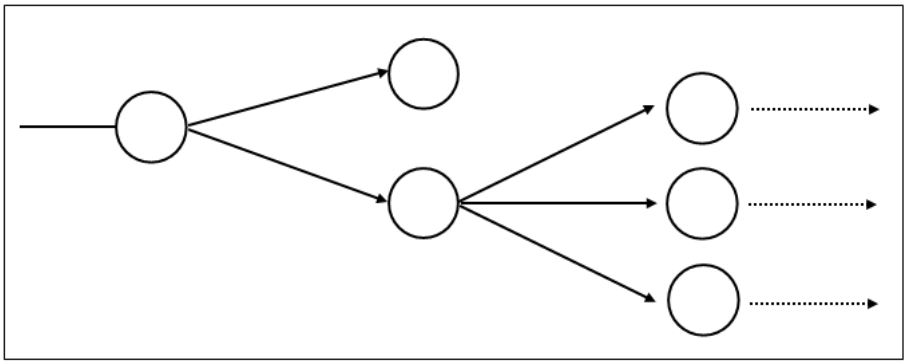

Fig 2.2 – Le déroulement temporel d'un système dans une logique arborescente

Dans notre travail, nous nous intéressons seulement à la logique temporelle linéaire propositionnelle $\mathcal{LTLP}$ (*Proposional Linear Temporal logic)*, qui restreint la construction de formules LTL à l'utilisation de propositions reliées par des connecteurs et des opérateurs de la logique temporelle.

2.3 La Logique Temporelle Linéaire Propositionnelle

La logique temporelle linéaire propositionnelle($\mathcal{LTLP}$) [26] [30] est une extension de la logique propositionnelle classique. Elle permet d'exprimer des propriétés sur l'ordre d'apparition de certains événements au cours d'une exécution d'un système. Elle utilise pour ce faire des modalités, afin de parler d'un ou plusieurs moments de l'exécution. Le temps utilisé par $\mathcal{LTLP}$ est discret et modélisé par un nombre entier. Rien n'existe entre l'instant j et l'instant $j+1$. La date des instants n'a pas d'importance, c'est leur ordre qui compte.

2.3.1 Rappel : le calcul propositionnel

Le calcul propositionnel (CP)(ou la logique propositionnelle) [26] [31] est une théorie logique qui définit les lois formelles du raisonnement. C'est la première étape dans la construction des outils de la logique mathématique.

Dans la logique propositionnelle, on étudie les relations entre des énoncés, que l'on va appeler propositions ou encore des formules. Ces relations peuvent être exprimées par l'intermédiaire de connecteurs logiques qui permettent, par composition, de construire des formules syntaxiquement correctes. Dans cette logique on prend seulement en compte l'assemblage de propositions, sans analyser plus avant le contenu de celles-ci.

Syntaxe du CP

S'intéresser à la syntaxe du CP, c'est considérer les formules qui sont bien écrites(ou bien formées). Pour cela, on se donne un alphabet, i.e. un ensemble de symboles répartis, avec :

- Un ensemble *Prop = {p, q, r,...}* dénombrable de lettres appelées variables propositionnelles qui sont des énoncés dont on ne connaît pas la signification,
- Un ensemble (fini) de connecteurs logiques. Les plus courants de ces connecteurs sont : négation($\neg$), conjonction($\wedge$), disjonction($\vee$), implication($\rightarrow$), équivalence($\equiv$).
  Les connecteurs ont une priorité. Ils sont donnés ici dans l'ordre des priorités décroissantes : $\neg, \wedge, \vee, \equiv$ et $\rightarrow$. Si on veut contrecarrer les priorités on utilise des parenthèses.
- Les parenthèses : (et).

Alors l'ensemble des formules bien formées (*fbf*) du calcul propositionnel est défini inductivement par :

- Les atomes sont des formules bien formées,
- Si A et B sont des formules bien formées alors, $\neg$A, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \equiv B)$ sont des formules bien formées,
- Toute formule bien formée est obtenue par application des règles précédentes un nombre fini de fois.

Exemple 2.3.1. *Soient p,q,r des variables propositionnelles.*
Les formules $(\neg p \vee \neg q)$, $(p \rightarrow q) \vee (p \rightarrow r)$,$\neg\neg p$ *sont des formules bien formées du calcul propositionnel.*

Sémantique du CP

Pour interpréter une formule du CP, il suffit de donner une valeur de vérité à ses constantes propositionnelles, pour calculer ensuite la valeur de vérité de la formule.

Dans la logique propositionnelle, il n'y a que deux valeurs de vérité, le "vrai" et le "faux", auxquels réfèrent respectivement les signes 1 et 0.

Définition 2.3.1. ***(Valuation)***
Une valuation (ou assignation) v est une fonction qui associe des valeurs de vérité (vrai ou faux) aux variables propositionnelles.

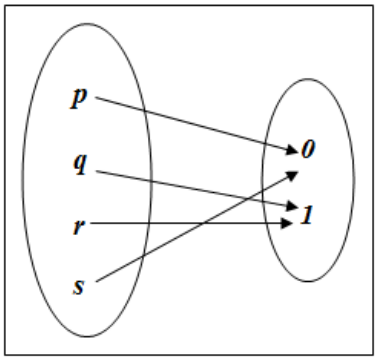

Fig 2.3 – Exemple de valuation *p=0, q=1, r=1, s=0*

Définition 2.3.2. ***(Interprétation)***
Une interprétation I d'une formule A est une valuation des variables de A. On associe une (et une seule à la fois) valeur de vérité aux variables propositionnelles au moyen d'une fonction d'interprétation.

Définition 2.3.3. ***(Fonction d'interprétation)***
On appelle fonction d'interprétation, une fonction qui évalue la valeur de vérité d'une formule en fonction des connecteurs logiques ($\neg, \wedge, \vee, \rightarrow, \equiv$) et d'une interprétation de la formule.

p	q	$\neg\ p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \equiv q$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Tab 2.1 – Interprétation des connecteurs booléens

Exemple 2.3.2. *Soit $A = ((p \vee q) \rightarrow (q \wedge r))$ une formule de la logique propositionnelle, soit $v = \{p = 1, q = 1, r = 0\}$ une valuation. L'interprétation de A pour v notéé par I(A) est :*

1. $A_1 = (p \vee q), I(A_1) = 1,$
2. $A_2 = (q \wedge r), I(A_2) = 0,$
3. $A = (A_1 \rightarrow A_2), I(A) = 0.$

2.3.2 Syntaxe de la $\mathcal{LTLP}$

Le langage

Le langage de la logique temporelle $\mathcal{LTLP}$ est constitué de quatre groupes de symboles différents : les variables propositionnelles, les connecteurs logiques, les opérateurs temporels et les parenthèses.

- **Les variables propositionnelles** : tout comme dans le cadre du calcul propositionnel, les variables proposionnelles sont représentées par des symboles non ambiguës qui les distingues des connecteurs, des opérateurs temporels et des parenthèses. Elles sont utilisées pour caractériser les états qui apparaissent dans les comportements. Ces propositions sont regroupées dans l'ensemble de propositions $Prop = \{p_1, p_2, \ldots\}$. Une proposition p_i est vraie dans un état e_i. Ainsi les états comprennent un ensemble de propositions élémentaires vraies,
- **Les combinateurs logiques :** des formules complexes sont construites avec les connecteurs de la logique propositionnelle $\{\neg, \wedge, \vee, \rightarrow, \equiv\}$.
- **Les opérateurs temporels :** il s'agit de symboles opérants comme des connecteurs unaires. Ils permettent de parler d'une séquence d'état (exécution) et non simplement d'un état pris individuellement. La logique temporelle $\mathcal{LTLP}$ utilise les opérateurs temporels du futur suivants [26] :
  - toujours(always)$\Box$
  - prochain(next)$\bigcirc$
  - éventuellement(Eventually)$\Diamond$.
- **Les parenthèses :** il s'agit simplement de la parenthèse ouvrante"(" et de la parenthèse fermente")" .

Définition 2.3.4. ***(Les formules bien formées)***
Une formule de $\mathcal{LTLP}$ est définie inductivement comme suit :

- *Toute proposition atomique p est une formule de la $\mathcal{LTLP}$,*
- *Si A et B sont des formules de la $\mathcal{LTLP}$ alors $\neg A$, $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \equiv B$, $\Box A$, $\bigcirc A$, $\Diamond A$ sont aussi des formules de la $\mathcal{LTLP}$.*

Exemple 2.3.3. *Soient p et q deux propositions de la $\mathcal{LTLP}$. Les formules*
- $p \vee q$
- $\Box p$
- $\neg\Diamond p \vee \Box\neg q$
- $\Diamond(p \vee q) \rightarrow \Diamond p$
- $\Box\Box(p \rightarrow p)$

sont des formules de la $\mathcal{LTLP}$.

2.3.3 Axiomatique de la $\mathcal{LTLP}$

Nous appellons *axiomatique* le système constitué par un ensemble de propositions admises comme vraies sans démonstration, appelées *axiomes* et un ensemble de règles logiques (appelées *règles d'inférence* ou *règles de déduction*). En appliquant une ou plusieurs fois des règles d'inférence à des axiomes de notre axiomatique, nous pouvons déduire des propositions vraies (appelées *théorèmes*). Et les théorèmes démontrés s'ajoutant aux axiomes admis, nous pouvons en leur appliquant de nouveau des règles d'inférence déduire de nouveaux théorèmes, etc ...

Soit *A,B,C* des formules de la $\mathcal{LTLP}$, le système formel de la $\mathcal{LTLP}$ est composé de :

- **Les axiomes** : les axiomes de la $\mathcal{LTLP}$ sont en nombre de huit, les trois premiers sont ceux du CP :

- *Axiome 1 :* $A \rightarrow (B \rightarrow A)$;
- *Axiome 2 :* $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$;
- *Axiome 3 :* $(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$;
- *Axiome 4 :* $\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$;
- *Axiome 5 :* $\bigcirc(A \rightarrow B) \rightarrow (\bigcirc A \rightarrow \bigcirc B)$;
- *Axiome 6 :* $\Box A \rightarrow (A \wedge \bigcirc A \wedge \bigcirc \Box A)$;
- *Axiome 7 :* $\Box(A \rightarrow \bigcirc A) \rightarrow (A \rightarrow \Box A)$;
- *Axiome 8 :* $\bigcirc A \equiv \neg \bigcirc \neg A$.

– **Les règles d'inférence** : la $\mathcal{LTLP}$ utilise deux règles d'inférence :

- *Le Modus Ponens :* $\dfrac{\vdash A \rightarrow B \quad \vdash A}{\vdash B}$;
- *Généralisation :* $\dfrac{\vdash A}{\vdash \Box A}$.

Exemple 2.3.4. *Utilisons l'axiomatique de $\mathcal{LTLP}$ pour démontrer le théorème suivant :*

Théorème 1. $\models \Box(p \rightarrow q) \rightarrow \Box(q \rightarrow r) \rightarrow \Box(p \rightarrow r)$

Démonstration. On pose $A = p \rightarrow q$, $B = q \rightarrow r$, $C = p \rightarrow r$.

1.	$\models \Box A$	Hypothèse
2.	$\models \Box B$	Hypothèse
3.	$\models A \rightarrow (B \rightarrow C)$	Théorème du CP
4.	$\models \Box(A \rightarrow (B \rightarrow C))$	3 + Géné
5.	$\models \Box(A \rightarrow (B \rightarrow C)) \rightarrow \Box A \rightarrow \Box(B \rightarrow C))$	Axiome 4
6.	$\models \Box A \rightarrow \Box(B \rightarrow C))$	4 + 5 + MP
7.	$\models \Box(B \rightarrow C)$	1 + 6 + MP
8.	$\models \Box(B \rightarrow C) \rightarrow (\Box B \rightarrow \Box C)$	Axiome 4
9.	$\models \Box B \rightarrow \Box C$	7 + 8 + MP
10.	$\models \Box C$	2 + 9 + MP

□

2.3.4 Sémantique de la $\mathcal{LTLP}$

Les formules de la logique temporelle $\mathcal{LTLP}$ expriment des propriétés sur une exécution d'un système. Une telle exécution est représentée par la suite des états dans les quels se trouvent ces systèmes au cours du temps.

Interprétation et valuation

L'interprétation en $\mathcal{LTLP}$ peut être représentée graphiquement par un *automate*(figure 2.4) (appelé aussi *diagramme état transition*) où, les cercles représentent les états et les arcs indiquent les transitions possibles entre les états. Chaque état est étiqueté par un ensemble de propositions, qui sont les propositions vraies dans cet état.

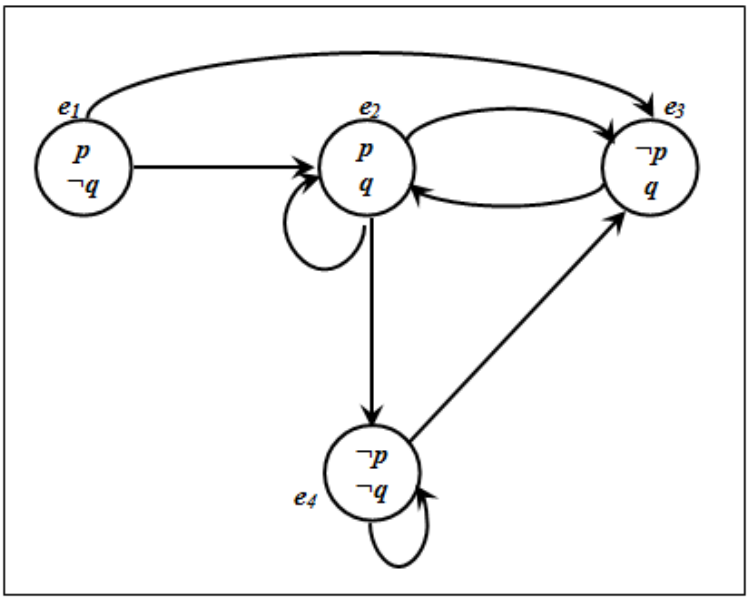

Fig 2.4 – Interprétation dans la $\mathcal{LTLP}$

Définition 2.3.5. *(**Interprétation**)*

Formellement, une interprétation I d'une formule A est un triplet(E,W,ρ) où :

- $E = \{e_1, e_2, \ldots, e_n\}$ *est un ensemble non vide appelé l'ensemble des états.*
- $W = \{w_1, w_2, \ldots, w_n\}$ *est un ensemble d'étiquettes (alphabet) qui dénote les événements observables du système, qui associe donc à chaque état un ensemble particulier de variables propositionnelles.*
- *ρ est une relation de transition.*

La valeur de vérité de la formule temporelle A à l'état e, notée $v_e(A)$ *, est définie inductivement :*

- *Si A est une variable atomique alors* $v_e(A) = w_e(A)$.
- *Si A est de la forme* $\neg A'$ *alors* $v_e(A) = 1$ *ssi* $v_e(A') = 0$.
- *Si A est de la forme* $A' \wedge A''$ *alors* $v_e(A) = 1$ *ssi* $v_e(A') = 1$ *et* $v_e(A'') = 1$.
- *Si A est de la forme* $A' \vee A''$ *alors* $v_e(A) = 1$ *ssi* $v_e(A') = 1$ *ou* $v_e(A'') = 1$.
- *Si A est de la forme* $A' \rightarrow A''$ *alors* $v_e(A) = 0$ *ssi* $v_e(A') = 1$ *et* $v_e(A'') = 0$.
- *Si A est de la forme* $A' \equiv A''$ *alors* $v_e(A) = 1$ *ssi* ($v_e(A')$= 1 *et* $v_e(A'') = 1$) *ou* ($v_e(A')$= 0 *et* $v_e(A'') = 0$).
- *si A est de la forme* $\Box A'$ *alors* $v_e(A) = 1$ *ssi* $v_{e'}(A') = 1$ *pour tous les états* e' *tel que* (e,e') $\in \rho$.
- *si A est de la forme* $\Diamond A'$ *alors* $v_e(A) = 1$ *ssi* $v_{e'}(A') = 1$ *pour un état* e' *tel que* (e,e') $\in \rho$.
- *si A est de la forme* $\bigcirc A'$ *alors* $v_e(A) = 1$ *ssi* $v_{e'}(A') = 1$ *pour l'état* e' *tel que* (e,e') $\in \rho$ *et* $e' = e + 1$.

Exemple 2.3.5. *Soit* $A = \Box p \vee \Box q$ *une formule de la logique temporelle. Dans la figure 2.4, la valeur de vérité de A dans chaque état est comme suit :*

- *A l'état e_1, $v_{e_1}(A)$=1 car e_2 et e_3 sont accessibles de e_1, et q est vrai dans les deux états.*
- *A l'état e_2, $v_{e_2}(A)$=0 car e_4 est accessibles de e_1, mais ni p ni q sont vrais dans cet état.*
- *A l'état e_3, $v_{e_3}(A)$=1 car le seul état accessibles de e_1 est e_2 où p et q sont tous les deux vrais.*
- *A l'état e_4, $v_{e_4}(A)$=0 car e_4 est uniquement accessibles par soit même, alors que ni p ni q sont vrais dans cet état.*

Interprétation des opérateurs temporels

Pour chaque opérateur et chaque formule temporelle, nous présentons une définition [30] de leur interprétation dans un modèle donné. Le modèle σ sur lequel on raisonne est une suite infinie de valuations sur des propositions atomiques.

Définition 2.3.6. ***(Modèle)***
Un modèle(ou une éxécution) est une fonction σ associant à chaque entier positif j l'ensemble des propositions atomiques vraies à l'instant j. Si une formule p de la $\mathcal{LTLP}$ est vraie l'insatnt j dans le modèle σ, on dit que σ satisfait p en j, et on note $(\sigma,j) \models p$ et qu'on peut schématiser par :

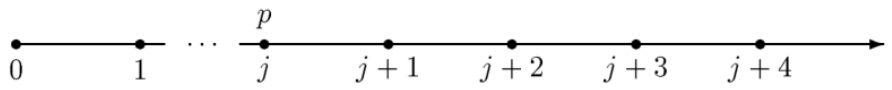

Fig 2.5 – Sémantique de la $\mathcal{LTLP}$

– ***L'opérateur toujours(always)*** $\Box$ ***:*** l'opérateur *toujours* exprime qu'une propriété p est satisfaite par tous les sufixes d'une exécution, c'est-à-dire à chaque instant. On note

$$(\sigma,j) \models \Box p \text{ ssi } \forall k \geq j : (\sigma, k) \models p$$

La formule $\Box p$ est vraie à la position j ssi p est vraie en toute position supérieur ou égale à j. Nous pouvons la schématiser par :

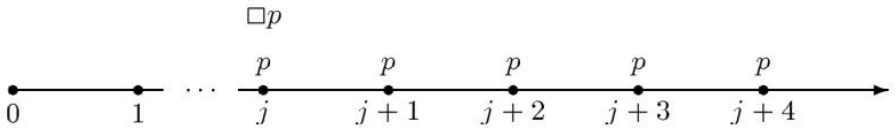

Fig 2.6 – Sémantique de l'opérateur *Always*

– ***L'opérateur prochain (next)*** $\bigcirc$: cet opérateur exprime qu'une propriété p va être satisfaite à l'instant suivant. On note

$$(\sigma,j) \models \bigcirc p \text{ ssi } (\sigma, j+1) \models p$$

La formule $\bigcirc p$ est vraie à la position j ssi p est vraie à la position $j+1$. Nous pouvons la schématiser par :

– ***L'opérateur éventuellement (Eventually)*** $\Diamond$: l'opérateur *éventuellement* exprime que la propriété p sera satisfaite au bout d'un temps fini. On note

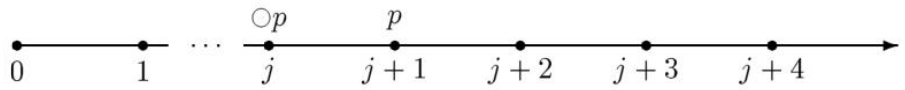

Fig 2.7 – Sémantique de l'opérateur *Next*

$(\sigma,j) \models \Diamond p$ ssi $\exists k \geq j : (\sigma, k) \models p$.

La formule $\Diamond p$ est vraie à la position j ssi p est vraie en une certaine position k supérieur ou égale à j. Nous pouvons la schématiser par :

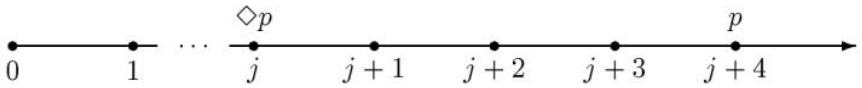

Fig 2.8 – Sémantique de l'opérateur *Eventually*

La Sémantique complète

Soit σ un modèle, j un entier naturel, p et q variables propositionnelles(propriétés). On définit la sémantique de la $\mathcal{LTLP}$ par :

- $(\sigma,j) \models p$ ssi p est vraie à la position j,
- $(\sigma,j) \models \neg p$ ssi il est faux que $(\sigma,j) \models p$,
- $(\sigma,j) \models p \wedge q$ ssi $(\sigma,j) \models p$ et $(\sigma,j) \models q$,
- $(\sigma,j) \models p \vee q$ ssi $(\sigma,j) \models p$ ou $(\sigma,j) \models q$,
- $(\sigma,j) \models p \rightarrow q$ ssi $(\sigma,j) \models \neg p$ ou $(\sigma,j) \models q$,
- $(\sigma,j) \models p \equiv q$ ssi $(\sigma,j) \models p \rightarrow q$ et $(\sigma,j) \models q \rightarrow p$,
- $(\sigma,j) \models \Box p$ ssi $\forall k \geq j : (\sigma, k) \models p$
- $(\sigma,j) \models \Diamond p$ ssi $\exists k \geq j : (\sigma, k) \models p$
- $(\sigma,j) \models \bigcirc p$ ssi $(\sigma, j+1) \models p$

2.3.5 Satisfaction et validité

Soit A une formule de la logique temporelle $\mathcal{LTLP}$.

Définition 2.3.7. ***(Satisfaction d'une formule)***

A est dite satisfiable(consistante), s'il existe une interprétation I = (E,W,ρ) de A et un état(monde) e ∈ E telle que $v_e(A) = 1$ (noté $e \models A$). Autrement, il existe un modèle σ qui satisfait la formule A à un instant j($(\sigma,j) \models A$)

Définition 2.3.8. ***(Validité d'une formule)***

on dit que A est valide(tautologie), et on note $\models A$, si et seulement si pour toute interprétation I = (E,W,p) et pour tout état(monde) e ∈ E, A est vraie .

Exemple 2.3.6. *$\Box p \rightarrow \neg\Diamond\neg p$ est une formule valide de la $\mathcal{LTLP}$*

Démonstration.

Soit $I = (E,W,p)$ une interprétation quelconque et e un état quelconque. Nous admettons que

$$e \models \Box p$$

et nous montrons que

$$e \models \neg\Diamond\neg p$$

Supposons que

$$e \models \Diamond\neg p$$

alors il existe un état $e^{'} \in E$ tel que

$$(e,e^{'}) \in \rho \text{ et } e^{'} \models \neg p$$

Or

$$e \models \Box p$$

pour tous les états accessibles pour e par ρ , et en particulier pour l'état $e^{'}$, d'où

$$e^{'} \models p(\text{contradiction})$$

Donc la supposition

$$e \models \Diamond\neg p$$

est fausse alors

$$e \models \Diamond\neg p$$

d'où

$$\Box p \rightarrow \neg\Diamond\neg p$$

est une formule *valide* de la $\mathcal{LTLP}$ □

Définition 2.3.9. ***(Contradiction)***
Si A est fausse dans toute interprétation dans tout état, A est dite insatisfiable ou contradiction.
Si A est une tautologie, alors $\neg A$ est une formule insatisfiable et inversement.

Définition 2.3.10. ***(Satisfaction d'un ensemble de formules)***
Un ensemble de formules $U = \{A_1, A_2, ..., A_n\}$ est satisfiable, s'il existe une interprétation I de U telle que toutes les formules de U soient satisfaites par I.

2.3.6 Propriétés des opérateurs temporels

Dans la suite p, q désignent des formules de la $\mathcal{LTLP}$.

Dualité : Chaque opérateur temporel a son dual :
- $\neg\Box p \equiv \Diamond\neg p$
- $\neg\Diamond p \equiv \Box\neg p$
- $\neg\bigcirc p \equiv \bigcirc\neg p$

Implication : Les formules valides suivantes décrivent des implications dans un seul sens :
- $\Box p \Rightarrow p$,
- $\Box p \Rightarrow \Diamond p$,

- $\Box p \Rightarrow \bigcirc p$,
- $\Box p \Rightarrow \bigcirc \Box p$,
- $\bigcirc p \Rightarrow \Diamond p$,
- $p \Rightarrow \Diamond p$,

Idempotence : Un opérateur est dit *idempotent* si une double application donne le même résultat qu'une seule application.

- $\Box\Box p \equiv \Box p$,
- $\Diamond\Diamond p \equiv \Diamond p$,

Absorption : Voici deux formules d'absorption :

- $\Diamond\Box\Diamond p \equiv \Box\Diamond p$,
- $\Box\Diamond\Box p \equiv \Diamond\Box p$.

Distributivité : Nous avons les relations de distributivité suivantes :

- $\Box$ (p $\wedge$ q)$\equiv \Box$p $\wedge \Box$q,
- $\Diamond$(p $\vee$ q)$\equiv \Diamond$p $\vee \Diamond$q,
- $\bigcirc$(p $\wedge$ q)$\equiv \bigcirc$p $\wedge \bigcirc$q,
- $\bigcirc$(p $\vee$ q)$\equiv \bigcirc$p $\vee \bigcirc$q,
- $\bigcirc$(p $\Rightarrow$ q)$\equiv \bigcirc$p $\Rightarrow \bigcirc$q,
- $\bigcirc$(p $\Leftrightarrow$ q)$\equiv \bigcirc$p $\Leftrightarrow \bigcirc$q,

2.4 Propriétés de la logique temporelle

La $\mathcal{LTLP}$ permet d'exprimer de nombreuses propriétés que l'on peut classer en deux catégories principales [13][32][33] :

2.4.1 Propriétés d'invariance (safety property)

Également appelées les propriétés de *sûreté*. Elles énoncent des conditions qui doivent toujours être vérifiées c'est à dire, maintenues à vrai. Cela équivaut à exprimer les mauvaises choses ne peuvent (ne doivent) pas se produire. En général, elles sont exprimées par des formules telles que : $\Box p$ *ou* $\Box(p \rightarrow q)$, qui peuvent être lues respectivement comme : "toute exécution du système vérifie q" et "si p est vraie à l'état initial, alors q est immédiatement vraie et elle restera vrai pendant le reste de l'exécution du système".

Des exemples de telles propriétés sont : l'atteignabilité (d'un état), l'absence de blocage, l'exclusion mutuelle, etc.

2.4.2 Propriétés de vivacité (liveness property)

Elles expriment qu'une situation particulière se produira. Les propriétés de vivacité assurent que les bonnes choses vont effectivement se produire. En général, elles sont exprimées par des formules telles que : $(p \rightarrow \Diamond q)$ *ou* $\Box(p \rightarrow \Diamond q)$, qui peuvent être lues respectivementcomme : "si p est vraie à l'état initial, alors q sera vraie dans le futur" et "toutes les fois où p est vraie, alors q sera vraie dans

le futur".

Des exemples de telles propriétés sont : la disponibilité, l'absence de famine, ou la garantie de service, etc.

Les autres propriétés, qui ne sont ni des propriétés de sûreté ni des propriétés de vivacité peuvent se ramener à la conjonction d'une propriété de sûreté et d'une propriété de vivacité

Chapitre 3

La méthode des tableaux sémantiques

Dans le présent chapitre, nous allons présenter une méthode dite *méthode des tableaux sémantiques*[10]. Nous présentons les différentes évolutions de cette méthode. A partir d'une des premières formalisations de la méthode pour la logique propositionnelle, on arrive à son application à la logique temporelle linéaire propositionnelle ($\mathcal{LTLP}$) [26][34][35]. La méthode des tableaux est développée par *Smullyan* pour la logique classique(vers 1968) et elle a été implémentée la première fois par *Fujita* en 1985.

3.1 Définitions

Définition 3.1.1. ***(Littéral)***.
On appelle littéral une constante propositionnelle(atome) ou une négation de constante propositionnelle.

Définition 3.1.2. ***(Complémentaire)***.

- *Si p est un atome alors {p,¬p} est une paire de littéraux complémentaires,*
- *Si A est une formule logique alors {A,¬A} est une paire de formules complémentaires. A est le complément de ¬A et ¬A est le complément de A.*

3.2 Tableaux pour la logique classique propositionnelle

Un tableau sémantique est une façon de prouver qu'une formule de logique propositionnelle A est satisfiable. Le plus souvent un tableau est utilisé pour des raisonnements par *réfutation*, c'est-à-dire que pour prouver que A est une tautologie, on montre avec un tableau que $\neg A$ n'est pas satisfiable.

3.2.1 Rappel : les tables de vérité

Les tables de vérité[26], permettent de décider, à propos de toute proposition, si celle-ci est satisfiable, une tautologie ou une contradiction.

Le principe de cette méthode est très simple. Toute formule contient un nombre fini d'atomes et admet donc un nombre fini d'interprétations. En conséquence, on peut déterminer la valeur de vérité de la formule pour toutes ses interprétations. On présente souvent le résultat sous forme d'une *table de vérité*, appelée aussi *tableau matriciel*.

Exemple 3.2.1. *Soit $F = (p \wedge q) \rightarrow \neg r \equiv (p \rightarrow (q \rightarrow \neg r))$ une formule du calcul propositionnel. La table de vérité correspondante à F est présentée à la figure 3.1.*

p	q	r	$\neg r$	$p \wedge q$	$q \rightarrow \neg r$	$(p \wedge q) \rightarrow \neg r)$	$p \rightarrow (q \rightarrow \neg r)$	F
0	0	0	1	0	1	1	1	1
0	0	1	0	0	1	1	1	1
0	1	0	1	0	1	1	1	1
0	1	1	0	0	0	1	1	1
1	0	0	1	0	1	1	1	1
1	0	1	0	0	1	1	1	1
1	1	0	1	1	1	1	1	1
1	1	1	0	1	0	0	0	1

Tab 3.1 – Table de vérité de la formule $F = (p \wedge q) \rightarrow \neg r \equiv (p \rightarrow (q \rightarrow \neg r))$

La dernière colonne de la table ne contient que des 1, cela signifie que la formule est vraie quelles que soient les valeurs des variables p, q et r, donc F est une formule valide du calcul propositionnel.

On voit immédiatement que la méthode est très inefficace ; si la formule à analyser contient n atomes distincts, elle admet 2^n interprétations. L'algorithme est donc exponentiel (en temps et espace) en fonction du nombre de propositions intervenant dans la formule.

On peut améliorer la méthode des tables de vérité en utilisant diverses simplifications. La plus importante consiste à ne pas attendre la fin de la construction de la table pour tirer une conclusion.

3.2.2 Présentation informelle de la méthode des tableaux

La table de vérité d'une formule permet à la fois de savoir si celle-ci est une tautologie et si elle est satisfiable. Si l'on rompt la symétrie, en s'intéressant seulement à l'un des deux problèmes, on peut espérer arriver plus rapidement au résultat.

Pour savoir si une formule est satisfiable, il peut être plus rapide d'analyser celle-ci, en cherchant à quelle condition elle est satisfaite, plutôt que d'énumérer toutes les valuations jusqu'à en trouver une convenable.

Soit la formule du calcul proposionnel $A = p \wedge (\neg q \vee \neg p)$. Supposons que l'on souhaite savoir si A est satisfiable ?
Soit v une interprétation de A :

- $v(A) = 1$ si et seulement si $v(p) = 1$ et $v(\neg q \vee \neg p) = 1$,
- Donc, $v(A) = 1$ si et seulement si

1. *v(p) = 1 et v(¬q) = 1*, ou
2. *v(p) = 1 et v(¬p) = 1.*

Donc, A est satisfiable si et seulement s'il existe une interprétation tel que (1) est vérifié, ou une interprétation tel que (2) est vérifié. Ainsi nous avons réduit la question de satisfiabilité de A en la satisfiabilité d'un ensemble de littéraux.

Définition 3.2.1. ***(Satisfaction d'un ensemble de littéraux)***
Un ensemble de littéraux est satisfiable si et seulement si cet ensemble ne contient pas simultanément un littéral et son complémentaire. [26]

Dans l'exemple précédent, le premier ensemble de littéraux $\{p,\neg q\}$ ne contient pas de littéraux complémentaires, alors cet ensemble est *satisfiable*, de là on conclut que A est aussi *satisfiable*. En outre nous pouvons lire rapidement un modèle de la formule A :

$$v(p) = 1 \text{ et } v(q) = 0$$

Cette méthode permet également de savoir si une formule est ou non une tautologie. L'exemple précédent nous permet par exemple de conclure que la formule $\neg A$ *n'est pas une tautologie.*

On considère maintenant la formule $B = (p \vee q) \wedge (\neg p \wedge \neg q)$

- $v(B) = 1$ si et seulement si $v(p \vee q) = 1$ et $v(\neg p \wedge \neg q) = 1$.
- Donc, $v(B) = 1$ si et seulement si $v(p \vee q) = 1$, $v(\neg p) = 1$ et $(\neg q) = 1$
- Donc, $v(B) = 1$ si et seulement si :
  1. $v(p) = v(\neg p) = v(\neg q) = 1$, ou
  2. $v(q) = v(\neg p) = v(\neg p) = 1$

B est satisfiable si et seulement s'il existe une interprétation tel que (1)est vérifié, ou une interprétation tel que (2) est vérifié, or les deux ensembles de littéraux $\{p, \neg p, \neg q\}$ et $\{q, \neg p, \neg q\}$ contiennent des littéraux complémentaires donc insatisfiables. De là on conclut qu'on ne peut trouver de modèle pour la formule B, donc $(p \vee q) \wedge (\neg p \vee \neg q)$ est insatisfiable donc $\neg B$ est une tautologie.

On ne donne pas de présentation systématique de cette méthode, qui a d'ailleurs plusieurs variantes. Une façon synthétique de procéder est de la présenter sous forme d'arbre, un embranchement correspond à l'étude de plusieurs possibilités. Une formalisation de cette méthode est connue sous le nom de *méthode des tableaux sémantiques.*

Le tableau sémantique correspondant à la formule A est représenté à la figure 3.1 et, le tableau correspondant à la formule B est représenté à la figure 3.2.

Une formule peut donner lieu à plusieurs tableaux sémantiques différents suivant l'ordre d'application des règles de construction, mais tous conduisent à la même conclusion concernant la consistance de la formule.

La signification commune des deux tableaux de la figure 3.3 est : une interpretation est un modèle de la formule $(p \rightarrow q) \wedge \neg(p \rightarrow r)$ si et seulement si c'est un modèle de l'un des ensembles $\{\neg p, p, \neg r\}$, $\{q, p, \neg r\}$.

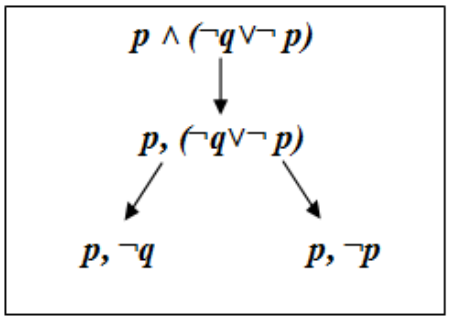

Fig 3.1 – Tableau sémantique de $A = p \wedge(\neg q \vee \neg\ p)$

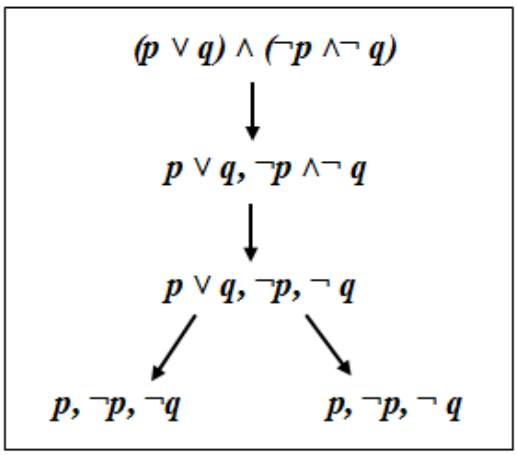

Fig 3.2 – Tableau sémantique de $B = (p \vee q) \wedge (\neg p \vee \neg\ q))$

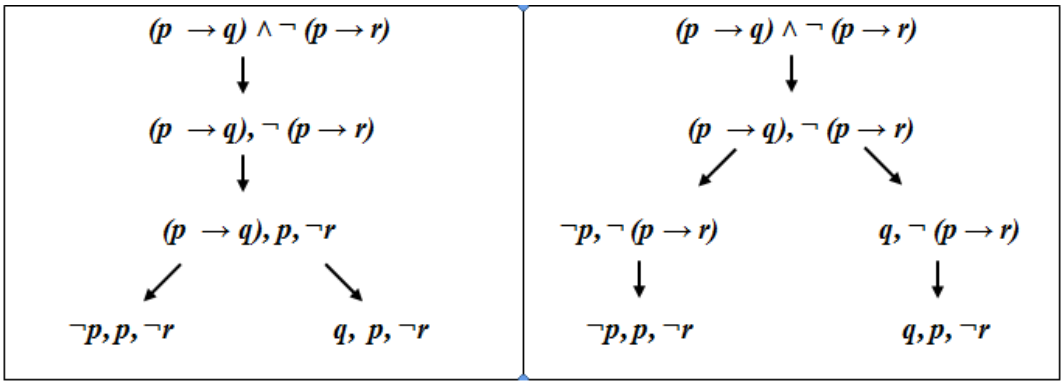

Fig 3.3 – Deux tableaux sémantiques de $(p \rightarrow q) \wedge \neg(p \rightarrow r)$

3.2.3 Les règles de décomposition des formules

La preuve par tableau prend la forme d'un arbre dont les noeuds sont étiquetés par un ensemble de formules logiques. Dans le cadre de la logique propositionnelle, cet arbre peut être vu comme une mise sous forme normale disjonctive : les feuilles de l'arbre représentent des conjonctions de sous-formules atomiques à satisfaire, tandis que l'arbre lui-même représente la disjonction de ces conjonctions. Une branche peut être vue comme une suite d'implications, des feuilles vers la racine ;

c'est-à-dire que la satisfiabilité d'une feuille implique celle de la formule.

La construction des tableaux sémantiques pour le CP est basée sur la partition des formules en trois catégories :

– Les littéraux ;
– Les formules conjonctives ;
– Les formules disjonctives.

Par exemple la formule $\neg(\mathrm{A} \rightarrow \mathrm{B})$ est conjonctive car elle est équivalente à la conjonction des deux formules A et $\neg$B, alors que la formule $\mathrm{A} \rightarrow \mathrm{B}$ est disjonctive car elle équivalente à la disjonction de $\neg$A et B.

La construction est basée sur deux types de règles de décomposition de formules appelés α et β règles. Dans le contexte des tableaux sémantiques on utilise les règles α pour les formules conjonctives, les règles β pour les formules disjonctives. Les deux tables 3.2 et 3.3 répertorient les règles de décomposition de type α et β respectivement.

α	α_1	α_2
$\neg\neg A$	A	
$A_1 \wedge A_2$	A_1	A_2
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$
$\neg(A_1 \rightarrow A_2)$	A_1	$\neg A_2$
$A_1 \equiv A_2$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$

Tab 3.2 – Les règles de décomposition de type α

β	β_1	β_2
$A_1 \vee A_2$	A_1	A_2
$\neg(A_1 \wedge A_2)$	$\neg A_1$	$\neg A_2$
$(A_1 \rightarrow A_2)$	$\neg A_1$	A_2
$\neg(A_1 \equiv A_2)$	$\neg(A_1 \rightarrow A_2)$	$\neg(A_2 \rightarrow A_1)$

Tab 3.3 – Les règles de décomposition de type β

3.2.4 Construction du tableau sémantique

Le processus général de construction d'un tableau sémantique pour une formule du calcul propositionnel donnée C est décrit à la figure 3.4. Ce tableau est un arbre dont chaque noeud est étiqueté par un ensemble de formules. Un noeud est *terminal* quand sont étiquette ne comporte que des littéraux. Quand la construction du tableau est achevée, toutes les feuilles sont des noeuds terminaux. On convient de marquer un noeud terminal par '$\bigcirc$' si l'étiquette est consistante(feuille *ouverte*), et par '$\times$' si l'étiquette est inconsistante(feuille *fermée*).

- *Initialisation*. On crée une racine étiquetée {C}.

- *Itération*. On sélectionne une feuille non marquée l, d'étiquette U(l).

  - Si U(l) est un ensemble de littéraux alors
    -si U(l) contient une paire de littéraux complémentaires alors
      marquer la feuille l fermée '×';
    -sinon, marquer la feuille l ouverte '◯';

  - Si U(l) n'est pas un ensemble de littéraux, choisir une formule,A, dans U(l),
    -si A est une α-formule alors
      créer un fils du noeud l, soit l';
      étiqueter l' avec U(l') = (U(l) - {A}) ∪ {α1,α2}
    -si A est une β-formule alors
      créer deux fils du noeud l, soit l' et l'';
      étiqueter l' avec U(l') = (U(l) - A) U {β1}, et
      étiqueter l'' avec U(l'') = (U(l) - A) U {β2} ;

- *Terminaison*. La construction est achevée quand toutes les feuilles sont marquées '×' ou '◯'.

Fig 3.4 – Algorithme de construction d'un tableau sémantique dans la CP

On utilise parfois des *assertions* plutôt que des formules, une assertion étant l'attribution d'une valeur de vérité à une formule. La figure 3.5 comporte un tableau classique, en notation'formule', et sa variante *signée*, en notation 'assertion'.

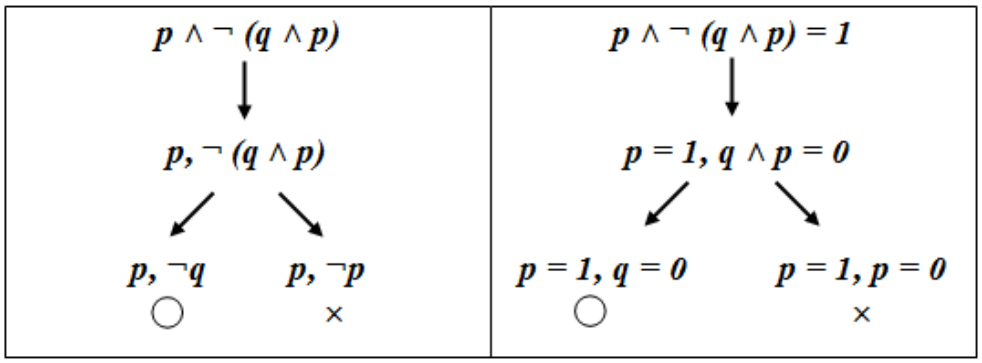

Fig 3.5 – Tableau classique et tableau signé

3.2.5 Consistance et validité

Définition 3.2.2. *(**Tableau complet**)*
Un tableau sémantique est dit complet si toutes ses feuilles sont marquées '×' ou '◯'.

Définition 3.2.3. *(**Tableau fermé/ tableau ouvert**)*
Un tableau complet est dit fermé si toutes ses feuilles sont marquées '×', sinon(c'est-à-dire le tableau contient des feuilles marquées "◯") le tableau est dit ouvert.

Définition 3.2.4. ***(Formule consistante et formule valide)***
Soit A une formule de la logique propositionnelle, et soit T le tableau sémantique complet de A. On dit que :

- *A est consistante(ou satisfiable) si et seulement si T est ouvert ;*
- *A est inconsistante(ou insatisfiable) si et seulement si T est fermé ;*
- *A est valide(ou tautologie) si et seulement si le tableau de $\neg A$ est fermé.*

3.2.6 Exemples de tableaux sémantiques

Exemple 3.2.2. $A_1 = (p \wedge \neg(q \rightarrow p))$.

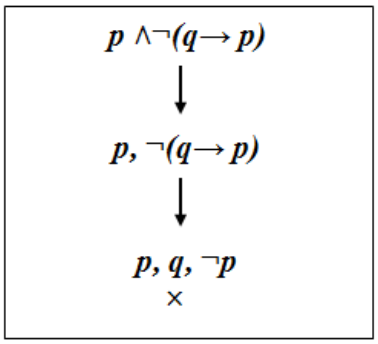

Fig 3.6 – Tableau sémantique de $(p \wedge \neg(q \rightarrow p))$

Le tableau sémantique (figure 3.6) de A_1 a une seule branche, cette branche est marquée '×' donc fermée, ce qui implique que A_1 est *inconsistante*. Il n'existe donc pas de modèle qui satisfait cette formule. Donc A_1 est une *contradiction*.

Exemple 3.2.3. *Soit* $A_2 = ((p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r)))$.
La figure 3.7 démontre que A est une tautologie(valide) en utilisant un tableau sémantique.
La racine de l'arbre est $\neg A_2$, la négation de la formule à prouver. L'arbre est construit en appliquant des règles de tableau successivement jusqu'à obtenir une contradiction ou à ne plus pouvoir appliquer de règles. Dans cet exemple toutes les branches sont fermées, cela implique que $\neg A_2$ n'est pas satisfiable, et donc la formule A_2 est une tautologie.

Exemple 3.2.4. *Soit* $A_3 = ((p \rightarrow q) \wedge (q \rightarrow r)) \vee ((r \rightarrow q) \wedge (q \rightarrow p))$.
Le tableau (figure 3.8) construit pour la formule proposionnelle A_3 possède des branches ouvertes et d'autres fermées. Donc A_3 possède au moins un modèle, ce qui implique que A_3 est satisfiable mais elle n'est pas une tautologie. Les modèles qui satisfont cette formule sont :

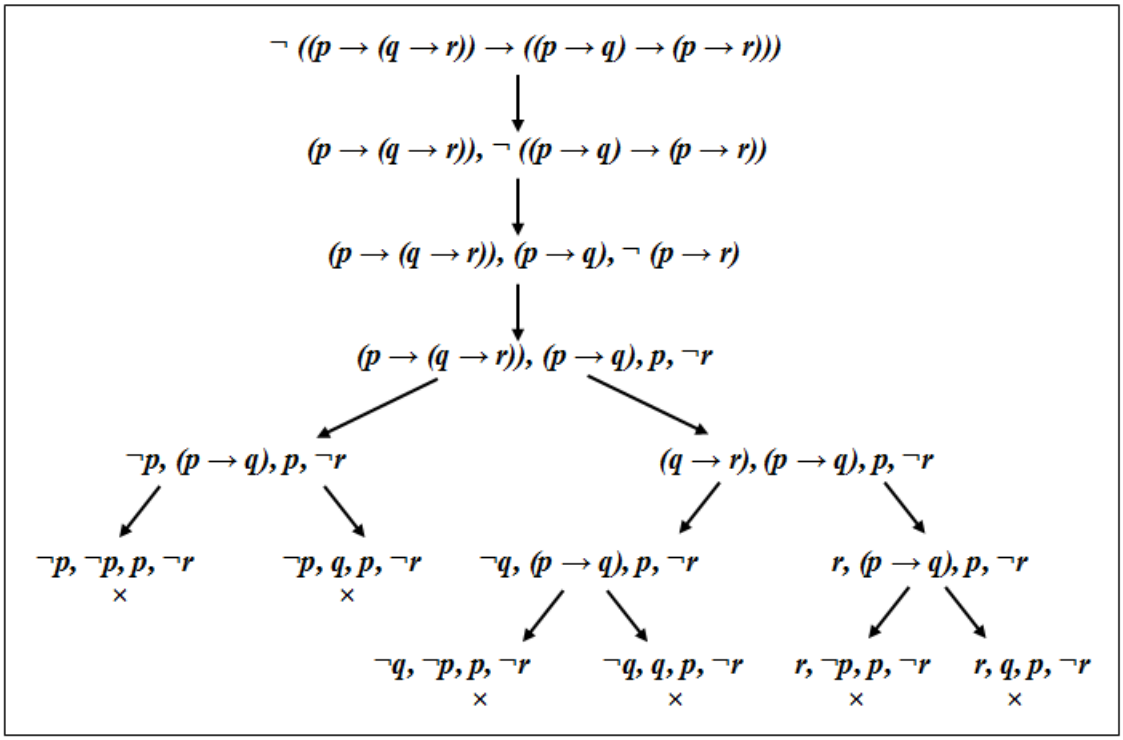

Fig 3.7 – Tableau de $\neg((p \to (q \to r)) \to ((p \to q) \to (p \to r)))$

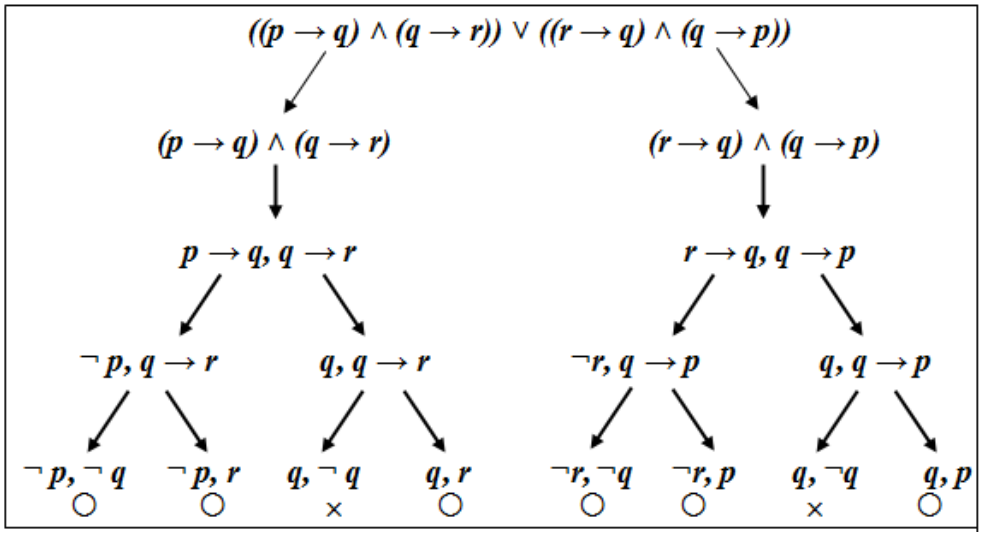

Fig 3.8 – Tableau de $((p \to q) \wedge (q \to r)) \vee ((r \to q) \wedge (q \to p))$

-$v(p) = 0$, $v(q) = 0$
-$v(p) = 0$, $v(r) = 1$
-$v(q) = 1$, $v(r) = 1$
-$v(p) = 0$, $v(q) = 0$
-$v(p) = 1$, $v(r) = 0$
-$v(q) = 1$, $v(r) = 1$

3.3 Tableaux pour la $\mathcal{LTLP}$

3.3.1 Les règles de décomposition des formules

La méthode des tableaux sémantiques pour la logique propositionnelle a été adaptée à une logique plus expressive [33], la logique temporelle linéaire propositionnelle(voir chapitre 2). Nous ajoutons les règles présentées dans les tables 3.4- 3.6 aux règles α et β définies pour la logique propositionnelle.

α	α_1	α_2
$\Box A$	A	$\bigcirc\Box$ A
$\neg\Diamond$ A	$\neg$A	$\neg \bigcirc \Diamond$ A

Tab 3.4 – Règles de décomposition de type α

β	β_1	β_2
$\Diamond$A	A	$\bigcirc\Diamond$A
$\neg\Box A$	$\neg A$	$\neg\Box\bigcirc$A

Tab 3.5 – Règles de décomposition de type β

X	X_1
$\bigcirc$A	A
$\neg\bigcirc$A	$\neg$A

Tab 3.6 – Règles de décomposition de type X

3.3.2 Construction du tableau sémantique

L'idée principale qui permet d'étendre les preuves par tableau aux logiques temporelles à temps linéaire [34] est de séparer une formule en deux parties : d'un côté les sous-formules à vérifier dans l'instant présent, et de l'autre celles à vérifier à l'instant suivant. Le processus général de construction du tableau est construit inductivement comme suit :

- *Initialisation*. On crée une racine étiquetée {C}.

- *Itération*. On sélectionne une feuille non marquée l, d'étiquette U(l).

  - Si U(l) est un ensemble de littéraux alors
    -si U(l) contient une paire de littéraux complémentaires alors
      marquer la feuille l fermée $'\times'$;
    -sinon, marquer la feuille l ouverte $'\bigcirc'$;

  - Si U(l) n'est pas un ensemble de littéraux, choisir une formule,A, dans U(l),

-si A est une α-formule alors
créer un fils du noeud l, soit l';
étiqueter l' avec U(l') = (U(l) - {A}) $\cup$ {α1,α2}
-si A est une β-formule alors
créer deux fils du noeud l, soit l' et l'';
étiqueter l' avec U(l') = (U(l) - A) U {β1}, et
étiqueter l'' avec U(l'') = (U(l) - A) U {β2} ;

∘ Si U(l) est un ensemble de littéraux et de X-formules {◯A1,...,◯An} alors
créer un fils du noeud l, soit l' ;
étiqueter l' avec U(l') = {A1,..., An}
-si U(l') est égal à un noeud qui a été déjà développé alors le nouveau fils ne sera pas construit mais le père l sera marqué existe '*'.

•*Terminaison.* La construction est achevée quand toutes les feuilles sont marquées '×', '◯' ou '*'.

Fig 3.9 – Algorithme de construction d'un tableau sémantique dans la $\mathcal{LTLP}$

3.3.3 Consistance et validité

Définition 3.3.1. *(**Tableau complet**)*
Un tableau sémantique est dit complet si toutes ses feuilles sont marquées '×', '◯' ou ''.*

Définition 3.3.2. *(**Tableau fermé/Tableau ouvert**)*
Un tableau complet est fermé si toutes ses feuilles sont marquées '×', sinon(c'est-à-dire le tableau contient des feuilles marquées '◯' ou '') le tableau est ouvert.*

Définition 3.3.3. *(**Formule consistance et formule valide**)*
Soit A une formule de la $\mathcal{LTLP}$, et soit T le tableau sémantique complet de A. On dit que :

- *A est inconsistante(ou insatisfiable) si et seulement si T est fermé ;*
- *A est valide(ou tautologie) si et seulement si le tableau de $\neg A$ est fermé.*
- *Cependant dans le cas où le tableau T est ouvert, nous ne pouvons conclure directement que la formule A est satisfiable.*

3.3.4 Exemples de tableaux

Exemple 3.3.1. *Soit $F_1 = (\bigcirc(p \to q) \to (\bigcirc p \to \bigcirc q))$ une formule de la logique temporelle linéaire propositionnelle. Pour prouver que F_1 est une tautologie, on montre avec un tableau sémantique que $\neg F_1$ est insatisfiable.*

Toutes les feuilles du tableau sémantique de $\neg F_1$ (figure 3.10) sont fermées, ce qui implique que $\neg F_1$ est insatisfiable. Donc F_1 est une tautologie.

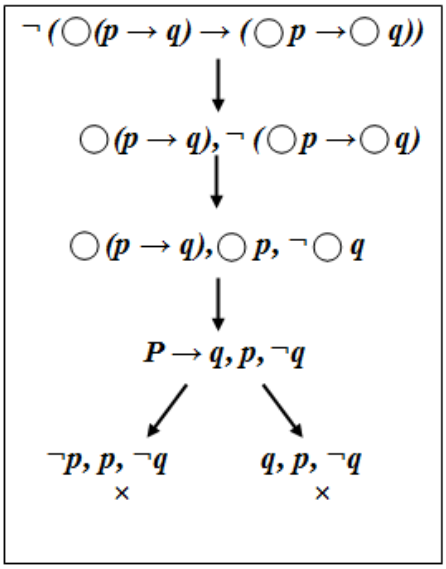

Fig 3.10 – Tableau sémantique de $\neg(\bigcirc(p \to q) \to (\bigcirc p \to \bigcirc q))$

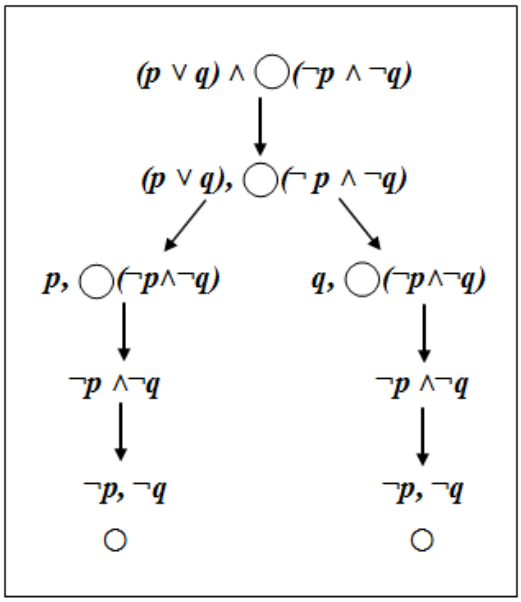

Fig 3.11 – Tableau sémantique de $((p \vee q) \wedge \bigcirc(\neg p \wedge \neg q))$

Exemple 3.3.2. *Soit* $F_2 = ((p \vee q) \wedge \bigcirc(\neg p \wedge \neg q))$.

Dans cet exemple, on remarque que toutes les feuilles du tableau (figure 3.11) sont marquée '$\bigcirc$*', donc le tableau de* F_2 *est ouvert. De là, on ne peut conclure que* F_2 *est satisfiable ou pas.*

Exemple 3.3.3. *Soit* $F_3 = \Box(\Diamond(p \wedge q) \wedge \Diamond(\neg p \wedge q) \wedge \Diamond(p \wedge \neg q))$.

Le tableau sémantique de F_3 *est montré dans la figure 3.12. Ce tableau est ouvert, il possède des feuilles ouvertes et d'autres fermées.*

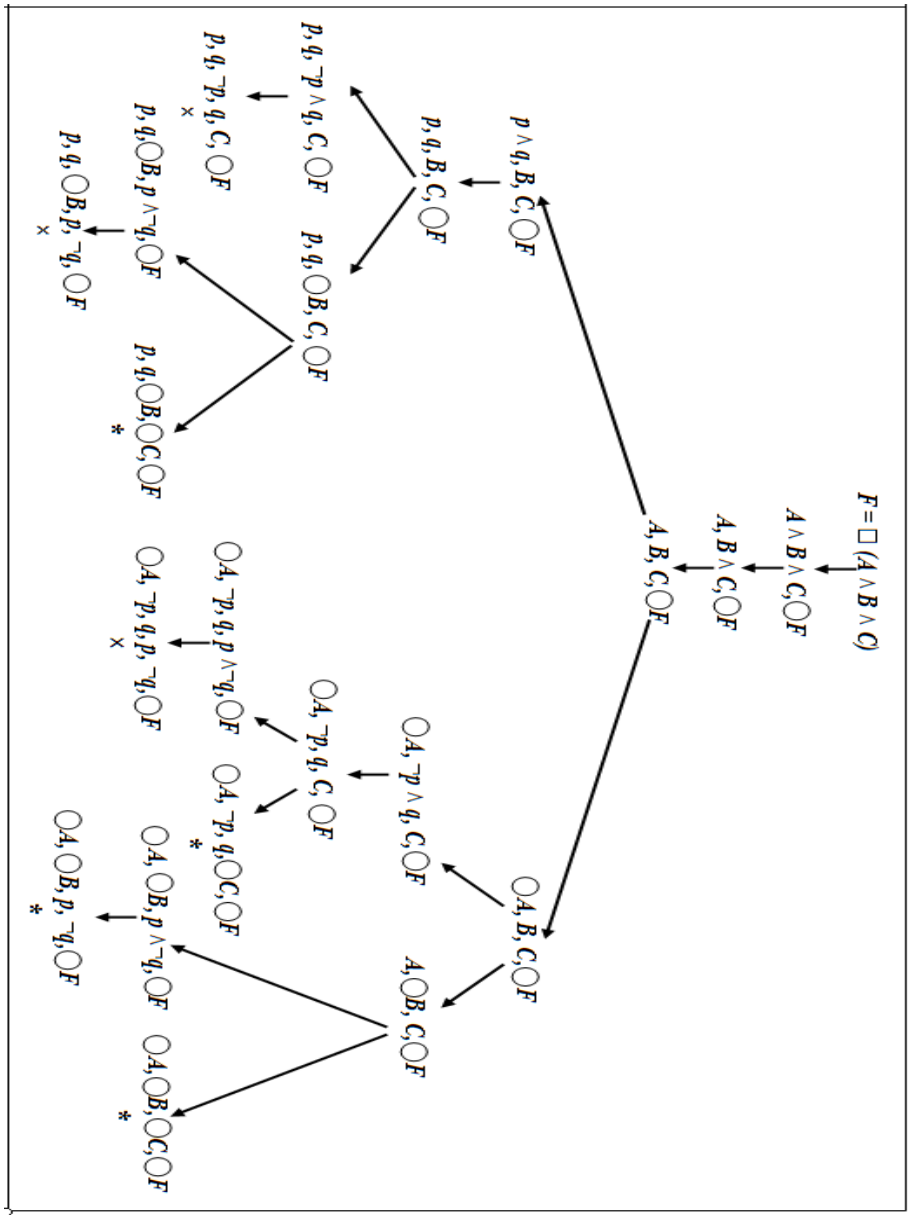

Fig 3.12 – Tableau sémantique de $\Box(\Diamond(p \wedge q) \wedge \Diamond(\neg p \wedge q) \wedge \Diamond(p \wedge \neg q))$

Dans le cas d'un tableau sémantique *ouvert*, pour décider si une formule donnée est satisfiable, une seconde étape d'analyse après la construction du tableau T est nécessaire. Cette étape est la *construction de la structure d'Hintikka* correspondante au tableau sémantique T.

3.4 La structure d'Hintikka

3.4.1 Principe

Les méthodes de la logique sont en général constructives. En particulier, si une formule A est consistante, non seulement tout tableau sémantique pour cette formule doit être ouvert, mais en outre il doit être possible de construire un modèle de A à partir de ce tableau.

On reprend la formule $A = ((p \vee q) \wedge \bigcirc(\neg p \wedge \neg q))$ vue à la section précédente.

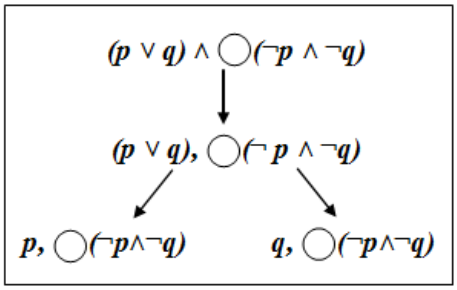

Fig 3.13 – Tableau sémantique de $((p \vee q) \wedge \bigcirc(\neg p \wedge \neg q))$

Si nous appliquons uniquement les $\alpha - r\grave{e}gles$ et $\beta - r\grave{e}gles$ à la formule A (figure 3.13), nous pouvons déduire que dans un modèle de A, l'état initial est

$$e_0 \models p$$

ou

$$e_0 \models q$$

Mais pour satisfaire la formule A dans la $\mathcal{LTLP}$, quelque chose doit être aussi vrai à l'état suivant e_1. Nous utilisons une règle de décompostion de type X pour construire le nouvel état e_1. Donc, par la X-règle

$$e_1 \models \neg p \wedge \neg q$$

doit être vrai et une application d'une α-règle complète le tableau(voir figure 3.11. Donc tout modèle de A doit être de la forme d'une des structures montrées à la figure 3.14. Dans ce cas, il n y a pas d'interprétations mais des *structures d'Hintikka* [36] qui peuvent être étendues à des interprétations en spécifiant les valeurs de toutes les formules pour chaque état.

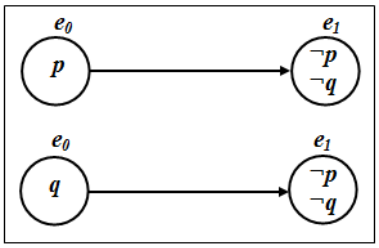

Fig 3.14 – La structure d'un modèle de $((p \vee q) \wedge \bigcirc(\neg p \wedge \neg q))$

3.4.2 Structure

Une structure est utilisée pour représenter les liens entre les états du système à vérifier. Il s'agit d'un graphe orienté dans lequel les noeuds, appelés états, sont étiquetés par les valuations de propriétés atomiques observées sur le système.

Définition 3.4.1. ***(Structure)***
Une structure est un triplet (E, U, ρ) où

- $E = \{e_1, e_2, \ldots, e_n\}$ *est un ensemble d'états.*
- $U = \{U_1, U_2, \ldots, U_n\}$ *est un ensemble d'ensembles des formules* $\mathcal{LTLP}$*, un ensemble est associé à chaque état.*
- ρ *est une relation binaire sur les états.*

Construction d'une structure

Considérons un tableau sémantique *ouvert* T d'une formule de la $\mathcal{LTLP}$, la structure correspondante au tableau T est obtenue de la manière suivante :

- Les états correspondent aux X-noeuds, c'est-à-dire les noeuds où les X-règles sont appliquées simultanément sur toutes les X-formules.
- Considérons deux états e_1 et e_2, $(e_1, e_2) \in \rho$ si et seulement si un chemin existe entre e_1 et e_2 dans le tableau, ou le X-noeud correspondant à e_1 est marqué "*" parce qu'il est identique à un noeud précèdent k et il existe un chemin de k à e_2.
- A $\in U_i$ si et seulement si A parait dans *U(n)* pour tout noeud n du chemin allant de e_j à e_i ne traversant pas un autre chemin.
  Formellement, soit
  $$(e_j = n_1, e_2, \ldots, n_k = e_i)$$
  un chemin, alors
  $$U_i = U(n_2) \cup U(n_3) \cup \ldots \ ,U(n_k).$$

Exemple

Soit $F = \Box(\Diamond(p \wedge q) \wedge \Diamond(\neg p \wedge q) \wedge \Diamond(p \wedge \neg q))$ une formule de $\mathcal{LTLP}$. Le tableau sémantique de F est montré à la figure 3.12. La structure obtenue du tableau sémantique de F est représentée à la figure 3.15. Seuls les variables propositionnelles de U_i sont représentées.

On pose $A \equiv \Diamond(p \wedge q)$, $B \equiv \Diamond(p \wedge q)$, $C \equiv \Diamond(p \wedge q)$. La structure correspondante au tableau sémantique de F est définit comme suit :

- L'ensemble des états $E = \{e_0, e_1, e_2\}$
- $U_i = \{U_0, U_1, U_2\}$ tel que :
  - $U_0 = \{\Box(A \wedge B \wedge C), A \wedge B \wedge C, \bigcirc\Box(A \wedge B \wedge C), A, B \wedge C, B, C, p \wedge q, \bigcirc\Diamond(\neg p \wedge q), \bigcirc\Diamond(p \wedge \neg\ q),\ p,\ q\}$;
  - $U_1 = \{\Box(A \wedge B \wedge C), A \wedge B \wedge C, \bigcirc\Box(A \wedge B \wedge C), A, B \wedge C, B, C, \bigcirc\Diamond(p \wedge q), \neg p \wedge q, \bigcirc\Diamond(p \wedge \neg q)), \neg\ p,\ q\}$;
  - $U_2 = \{\Box(A \wedge B \wedge C), A \wedge B \wedge C, \bigcirc\Box(A \wedge B \wedge C), A, B \wedge C, B, C, \bigcirc\Diamond(p \wedge q), \bigcirc\Diamond(\neg p \wedge q), p \wedge \neg q, p, \neg\ q\}$.
- $\rho = \{(e_0, e_0),\ (e_0, e_1),\ (e_0, e_2),\ (e_1, e_1),\ (e_1, e_0),\ (e_1, e_2),\ (e_2, e_2),\ (e_2, e_0),\ (e_2, e_1)\}$

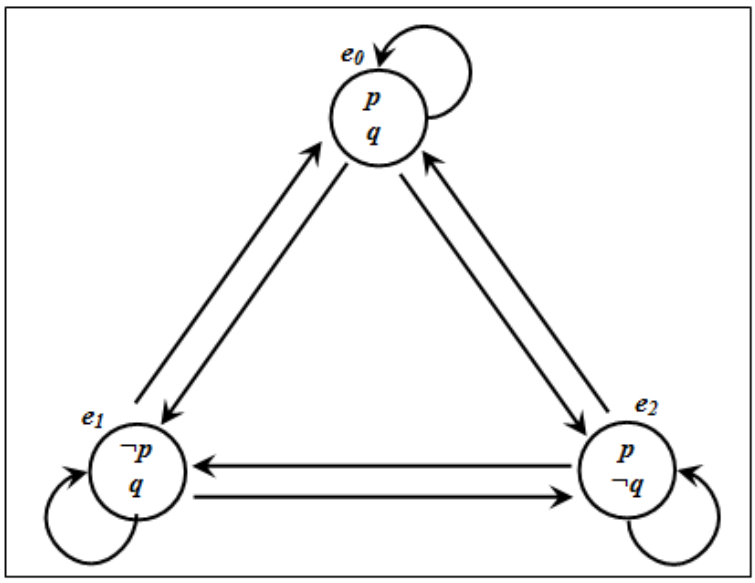

Fig 3.15 – Structure du tableau sémantique de $\Box(\Diamond(p \wedge q) \wedge \Diamond(\neg p \wedge q) \wedge \Diamond(p \wedge \neg q))$

Cette structure n'est pas linéaire alors elle ne peut pas être étendue directement à un modèle.

3.4.3 Structure Hintikka

Définition 3.4.2. ***(Structure d'Hintikka)***

Soit $H = (E, U, \rho)$ une structure. H est une structure d'Hintikka si et seulement si pour tout état e_i et toute formule $A \in U_i$:

1. *Si A est un atome p, alors $\neg p \notin U_i$*
2. *Si A est une α-formule, alors $\alpha_1 \in U_i$ et $\alpha_2 \in U_i$*
3. *Si A est une β-formule, alors $\beta_1 \in U_i$ ou $\beta_2 \in U_i$*
4. *Si $A = \bigcirc A_1$ est une X-formule, alors pour tout état e_j tel que $(e_i, e_j) \in \rho$, $A_1 \in U_j$*

Théorème 2. *La structure crée pour un tableau sémantique est une structure Hintikka [26].*

Exemple 3.4.1. *La structure construite pour la formule $F = \Box(\Diamond(p \wedge q) \wedge \Diamond(\neg p \wedge q) \wedge \Diamond(p \wedge \neg q))$, illustrée dans la figure 3.15, est une structure d'Hintikka.*

3.4.4 Lemme d'*Hintikka* pour la $\mathcal{LTLP}$

Définition 3.4.3. ***(Structure d'Hintikka Linéaire)***

Soit H une structure d'Hintikka. H est une structure d'Hintikka linéaire si ρ est linéaire, c'est-à-dire, pour tous les états e_i il y a au plus un état e_j tel que $(e_i, e_j) \in \rho$.

Définition 3.4.4. ***(La clôture réflexive-transitive)***

ρ^* est la clôture réflexive transitive de ρ, tel que

- *Si$(e_1, e_2) \in \rho$ alors $(e_1, e_2) \in \rho^*$;*
- *$(e_i, e_i) \in \rho^*$, $\forall\ e_i \in E$;*
- *Si $(e_1, e_2) \in \rho^*$ et (e_2, e_3) ρ^* alors $(e_1, e_3) \in \rho^*$*

Définition 3.4.5. ***(structure d'Hintikka complète"fulfilling Hintikka structure")***

Soit H une structure d'Hintikka. H est une structure d'Hintikka complète si pour tous les états e_i et pour toutes les formules futurs $A = \Diamond A_1$:

$$\textit{Si } A \in U_i, \textit{ alors il existe } e_j \in \rho^* \textit{ tel que } A_1 \in U_i.$$

Théorème 3. ***(Lemme d'Hintikka pour la $\mathcal{LTLP}$)***

Soit H une structure d'Hintikka complète et linéaire de A. Alors A est satisfiable [26].

Exemple 3.4.2. *Un modèle construit pour la structure de la figure 3.15 est illustré dans la figure ci-dessous*

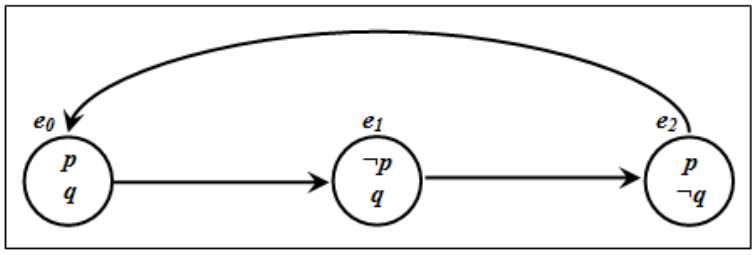

Fig 3.16 – Une structure d'Hintikka complète et linéaire

Chapitre 4

La terminaison

Nous allons introduire dans ce chapitre la notion de système de réécriture des termes [37]. Nous aborderons tout d'abord la notion de terme et de signature, avant de donner la définition des systèmes de réécriture. Ensuite nous allons présenter une propriété importante des systèmes de réécriture, la terminaison, ainsi quelques méthodes qui permettent de prouver cette propriété.

4.1 Les termes

Définition 4.1.1. *(**Signature**)*
Une signature Σ est un ensemble de symboles pour lequel chaque symbole est associé à une valeur entière positive appelée arité(nombre d'arguments attendu par le symbole).

Définition 4.1.2. *(**Termes**)*
Soit $\Sigma = \{f_1, ..., f_n\}$ une signature. Soit V un ensemble de variables. On peut alors définir l'ensemble T(Σ,V) des termes construits à partir de Σ et V par les clauses suivantes :

- *une variable est un terme ;*
- *pour tout symbole $f \in \Sigma$ d'arité n, si $t_1,\ldots,t_n$ sont des termes, alors $f(t_1,\ldots,t_n)$ est un terme.*

Exemple 4.1.1. *Soit V = {x,y} un ensemble de variables et Σ = {0,Succ,Plus} une signature composée de trois fonctions d'arité 0,1,2 respectivement. Alors :*
0, Succ(0), Succ(Succ(0)), Plus(x,Succ(0)), Succ(Plus(Succ(y), Plus(0,x)))
sont des termes

4.2 Règles et système de réécriture sur les termes

La réécriture est un formalisme de calcul basé sur la donnée de règles de réécriture.

Définition 4.2.1. *(**Règle de réécriture**)*
Une règle de réécriture est une paire orientée de termes et est notée $t \to t'$ *où t,* $t' \in T(\Sigma, V)$*. On appelle t et* t' *respectivement le membre gauche et le membre droit de la règle.*

Deux restrictions sont souvent imposées sur une règle de réécriture t $\to t'$:

1. t $\notin$ V (le membre gauche de la règle n'est pas une variable) ;
2. V(t') $\subseteq$ V(t)(toutes les variables du membre droit apparaissent dans le membre gauche)

Exemple 4.2.1. *Plus(x,Succ(y)) → Succ(plus(x,y)) et Plus(x,0) → x sont des règles de réécritures valides*

Maintenant qu'on a défini ce qu'est une règle de réécriture, on peut poser la définition des systèmes de réécriture :

Définition 4.2.2. *(**Système de réécriture**)*
Un système de réécriture $\Re$ *sur les termes est un ensemble fini de règles de réécriture.*

Exemple 4.2.2. *On reprend la signature et l'ensemble de variables définis dans l'exemple 4.1.1.*
L'ensemble des règles de réécriture ci-dessous définissent un système de réécriture $\Re$*.*

$$\Re = \begin{cases} Plus(x, 0) \to x \\ Plus(x, Plus(y, z)) \to (Plus(Plus(x, y), z) \\ Plus(x, Succ(y)) \to Succ(plus(x, y)). \end{cases}$$

La première règle dit que

$$x + 0 = x$$

la seconde dit que

$$x + (y + z) = (x + y) + z$$

et la dernière dit que

$$x + (y + 1) = (x + y) + 1$$

Vérifions par ce système que 2 + 2 = 4.
2 + 2 = Plus((Succ(Succ(0)),(Succ(Succ(0))) → Succ(Plus((Succ(Succ(0)),(Succ(0)))
→ Succ(Succ(Plus(Succ(Succ(0))),0))
→ Succ(Succ(Succ(Succ(0)))) = 4.

Exemple 4.2.3. *Soit* $\Sigma = \{\neg, \vee, \wedge\}$ *une signature composée de trois fonctions d'arité 1,2,2 respectivement. Soit V={x,y,z} l'ensemble des variables. L'ensemble*

des règles de réécriture ci-dessous définissent un système de réécriture $\Re'$.

$$\Re' = \begin{cases} \neg(\neg(x)) & \to x \\ \neg(\vee(x,y)) & \to \wedge(\neg(x),\neg(y)) \\ \neg(\wedge(x,y)) & \to \vee(\neg(x),\neg(y)) \\ \wedge(x,\vee(y,z)) & \to \vee(\wedge(x,y),\wedge(x,z)) \\ \wedge(\vee(x,y),z) & \to \vee(\wedge(x,y),\wedge(y,z)). \end{cases}$$

Le système $\Re'$ calcule la *forme normale disjonctive* d'une formule du calcul proposionnel.

4.3 Contextes, substitutions et relations de réécriture

Avant de définir la notion de relation de réécriture associée à un système de réécriture, on doit définir celles de contextes et de substitutions.

Définition 4.3.1. *(**Contexte**)*
Un contexte est un terme avec exactement un emplacement de libre. Plus formellement les contextes se définissent par induction :
- *le contexte vide noté[ ] est un contexte ;*
- *pour tout contexte C, pour tout symbole f d'arité n, pour tout i, si $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ sont des termes, alors $f(t_1, \ldots, t_{i-1}, C, t_{i+1}, \ldots, t_n)$ est un contexte.*

Si C est un contexte et t un terme, on note $C[t]$ le terme obtenu en remplissant l'emplacement libre de C avec le terme t.

Exemple 4.3.1. *Si C est le contexte vide, C[t] représente le terme t lui même.*

Définition 4.3.2. *(**Substitution**)*
Une substitution est une application $\sigma : V \to T(\Sigma, V)$. Une fois définie sur les variables, cette application peut être définie sur tous les termes. L'application d'une substitution σ à un terme t, notée $t\sigma$, consiste à remplacer chaque occurrence d'une variable v apparaissant dans t par le terme $\sigma(v)$

On peut maintenant définir la notion de relation de réécriture associée à un système de réécriture $\Re$.

Définition 4.3.3. *(**Relation de réécriture**)*
Étant donné une signature Σ et un système de réécriture $\Re$ défini sur Σ. La relation de réécriture associée à $\Re$ sur Σ est notée $\to_{\Re}$ et est définie comme suit :
$t_1 \to_{\Re} t_2$ s'il existe une règle de réécriture $t \to t'$ de $\Re$ ainsi qu'un contexte C et une substitution σ tels que $t_1 = C[t\sigma]$ et $t_2 = C[t'\sigma]$

4.4 Terminaison des systèmes de réécriture

La relation de réécriture $\rightarrow_{\Re}$ associée à un système de réécriture $\Re$ est une relation binaire sur l'ensemble des termes *T(Σ, V)*. Rappelons ci-dessous quelques définitions importantes.

Définition 4.4.1. ***(Relation binaire)***
Une relation binaire $\Re$ sur des ensembles E_1, E_2 est un sous ensemble du produit cartésien de E_1, E_2. $\Re \subseteq E_1 X E_2$.
On note x $\Re$ y si (x,y) $\in \Re$.

Définition 4.4.2. ***(Relation d'ordre)***
Une relation binaire $\Re$ sur un ensemble E est une relation d'ordre ssi elle satisfait les propriétés suivantes :

- *Réfléxivité : $\forall x \in E$, x $\Re$ x*
- *Antisymétrie : $\forall x,y \in E$, x $\Re$ y et y $\Re$ x $\Rightarrow x = y$*
- *Transitivité : $\forall x,y,z \in E$, x $\Re$ y et y $\Re$ z $\Rightarrow$ x $\Re$ z*

Notation : $\geq$

Exemple 4.4.1. *E_1 = {(2,2),(3,3),(4,4),(2,3),(3,4)} est un ordre.*
E_2 = {(2,2),(3,3),(4,4),(3,2),(2,3),(2,4),(3,4)} n'est pas un ordre car (3,2), (2,3) mais $2 \neq 3$.

Définition 4.4.3. ***(Ordre strict)***
Une relation binaire $\Re$ sur un ensemble E est une relation d'ordre strict si et seulement si elle est transitive, anti-symétrique et vérifie :

$$\forall x \in E, \; non(x \; \Re \; x).$$

Notation : >

Exemple 4.4.2. *> sur les entiers.*

Définition 4.4.4. ***(Ordre bien fondé)***
Une relation d'ordre strict $\Re$ est dite bien fondée s'il n'existe pas de suite infinie$(u_i)_{i \in N}$ vérifiant $\forall_i \in N$, $u_i > u_{i+1}$.

Exemple 4.4.3. *> sur les entiers naturels est bien fondé mais > sur tous les entiers n'est pas bien fondé.*

On peut définir la propriété suivante pour la relation binaires $\rightarrow_{\Re}$ associée à un système de réécriture $\Re$:

Définition 4.4.5. ***(Relation binaire terminante)***
Une relation binaire $\rightarrow_{\Re}$ sur un ensemble T(Σ, V) de termes est terminante s'il n'existe pas de chaîne de réduction infinie $t_1 \rightarrow_{\Re} t_2 \rightarrow_{\Re} t_3 \ldots$ de termes de T(Σ, V). En d'autres termes, la relation $\rightarrow_{\Re}$ termine si et seulement si elle est bien fondée.

Théorème 4. *On dit qu'un système de réécriture $\Re$ termine si la relation de réécriture qui lui est associée est terminante.*

Le problème consistant à déterminer si un système de réécriture donné termine est indécidable [38]. Néanmoins, il existe des méthodes pour décider cette propriété pour des classes de systèmes de réécriture spécifiques.

Les techniques utilisées pour établir la terminaison d'un système de réécriture peuvent être classées [39] selon différentes catégories :

- ***Les méthodes incrémentales et modulaires*** : elles consistent à appliquer un principe de type "divide and conquer", c'est-à-dire à prouver séparément la terminaison de plusieurs relations de réécriture dont la relation de départ est l'union ;
- ***Les méthodes transformationnelles*** : elles consistent à transformer le système de réécriture de sorte que sa terminaison puisse se déduire de celle du système transformé. L'une des méthodes de cette catégorie est celle des *paires de dépendances*[40] ;
- ***Les méthodes syntaxiques*** : elles consistent à construire un ordre bien fondé sur les termes, et à montrer que la relation $R^{'}$engendrée par un système de réécriture R est incluse dans cet ordre, ce qui prouve que $R^{'}$ est bien fondée, et donc que R termine. La plus connue de ces méthodes est *RPO(Recursive Path Ordering)* [41].
- ***Les méthodes sémantiques*** : elles consistent à interpréter chaque symbole f d'arité n par une fonction n-aire sur un domaine D à préciser, et à comparer les termes en les interprétant et en comparant leurs interprétations grâce à une relation bien fondée sur D. L'une des méthodes les plus connues de cette catégorie est celle par *interprétations polynomiales* [39] [42].

.

4.4.1 Les ordres de réduction sur les termes

Une première façon de montrer qu'un système de réécriture termine consiste à trouver un ordre strict $>$ sur l'ensemble des termes $T(\Sigma, V)$ qui soit bien fondé.

Mais cette première façon présente l'énorme inconvénient de requérir le test pour l'ensemble des couples t $\rightarrow_{\Re} t^{'}$, c'est-à-dire, vérifier $t > t^{'}$ pour tous les t, $t^{'}$ tel que t $\rightarrow_{\Re} t^{'}$, potentiellement infini.

Cependant, il existe d'autres façons de prouver que le système de réécriture $\Re$ termine, qui recquièrent seulement la vérification de la propriété $t > t^{'}$ pour uniquement les règles $t \rightarrow t^{'} \in \Re$ (un nombre fini de règles). Une de ces façons consiste à mettre en évidence l'existence d'un ordre de réduction [38] [39].

Définition 4.4.6. ***(Ordre de réduction)***

On considère les termes construit à partir d'une signature Σ et d'un ensemble de variables V. Un ordre strict $>$ sur l'ensemble des termes $T(\Sigma, V)$ est un ordre de réduction si et seulement si cet ordre est [43] :

- *bien fondé ;*
- *clos par contexte : $t > t^{'}$ alors $f(\ldots,t,\ldots) > f(\ldots,t^{'},\ldots)$;*

– *clos par substitution : $t > t'$ alors pour toute substitution σ,(σt) > $\sigma(t')$.*

La recherche d'un ordre de réduction est motivée par le résultat suivant :

Théorème 5. *Un système de réécriture $\Re$ termine si et seulement s'il existe un ordre de réduction > satisfaisant $t > t'$ pour toute règle $t \to t' \in \Re$ [38] [39].*

4.4.2 Méthode des interprétations polynomiales

La méthode des interprétations [39] est classiquement utilisée pour montrer la terminaison de système de réécriture. Cette méthode permet de construire des ordres de réductions non pas sur l'ensemble des termes *T(Σ, V)* mais elles étudient le comportement de l'image de ces termes par une famille de fonctions évaluant les termes vers un ensemble où il existe un ordre bien fondé.

Au rang des méthodes d'interpretations, on peut citer les méthodes d'*interprétations polynomiales* où les termes sont interprétés par des fonctions polynomiales sur les entiers. Ces méthodes sont introduites par Lankford.

Définition 4.4.7. ***(Interprétation polynomiale)***
Une interprétation polynomiale P sur une signature Σ de domaine A est la donnée d'un polynôme $P_f(x_1,\ldots,x_n)$ monotone(c'est à dire qu'il dépend de toutes ses indeterminées)pour chaque symbole f d'arité n de Σ.

S'il on se donne une indeterminée pour chaque variable, on peut définir par induction un polynôme P_t pour chaque terme t :

$$P_t = \begin{cases} x & \text{si } t = x \\ P_f(P_{t_1}, P_{t_2}, \ldots, P_{t_n}) & \text{si } t = f(t_1, t_2, \ldots, t_n). \end{cases}$$

Cette interpretation polynomiale fournit un ordre sur les termes qui est défini comme suit :

Définition 4.4.8. ***(Ordre polynomial)***
L'ordre polynomial $>_P$ est défini sur T(Σ, V) par :
$t_1 >_P t_2$ ssi $P_{t_1}(a_1,\ldots,a_n) > P_{t_2}(a_1,\ldots,a_n)$ pour tout $a_1,\ldots,a_n \in A$

Exemple 4.4.4. *Soit $\Sigma = \{\oplus,\odot\}$ une signature composée de deux fonctions d'arité 2. Nous définissons le système de réécriture $\Re$ composé d'une seule règle :*

$$\Re = \{x \odot (y \oplus z) \to (x \odot y) \oplus (x \odot y)\}$$

Soit l'interpretation polynomiale suivante :
– *A = N-{0,1}*
– *$P_\oplus$ = 2X + Y + 1, $P_\odot$ = XY*
Cette interprétation polynomiale est monotone et fournit donc un ordre de réduction.

Démonstration. On note la seule règle de $\Re$ par $t_1 \to t_2$.

$$\begin{aligned} P_{t_1} &= P_\odot(X, P_\oplus(Y,Z)) \\ &= P_\odot(X, 2Y + Z + 1) \end{aligned}$$

$= X(2Y + Z + 1)$
$= 2XY + XZ + X$

$P_{t_2} = P_{\oplus}(P_{\odot}(X,Y),P_{\odot}(X,Z))$
$= P_{\oplus}(XY,P_{\odot}(X,Z))$
$= P_{\oplus}(XY,XZ)$
$= 2XY + XZ + 1$

Étant donné que tous les éléments de A sont supérieurs à 1 alors,

$$2XY + XZ + X > 2XY + XZ + 1$$

D'où $t_1 >_P t_2$ ce qui assure la terminaison du système de réécriture $\Re$. □

Exemple 4.4.5. *Soit $\Re'$ un système de réécriture défini sur la signature $\Sigma' = \{f,o\}$ composée de deux fonctions d'arité 1 et 2 respectivement*

$$\Re' = \begin{cases} f(xoy) \to f(x)of(y) \\ (xoy)oz \to xo(yoz). \end{cases}$$

Soit l'interpretation polynomiale suivante :
– A = N-{0,1}
– $P_f = X^2$, $P_o = XY + X$
Cette interprétation polynomiale est monotone et fournit donc un ordre de réduction.

Démonstration. On note la première règle de $\Re$ par $t_1 \to t_2$.
$P_{t_1} = f(XY + X)$
$= (XY + X)^2$
$= X^2Y^2 + X^2 + 2X^2Y$

$P_{t_2} = X^2oY^2$
$= X^2Y^2 + X^2$

$X^2Y^2 + X^2 + 2X^2Y > X^2Y^2 + X^2$
D'où $t_1 >_P t_2$
On note la deuxième règle de $\Re$ *par* $t_3 \to t_4$.
$P_{t_3} = (XY + X)Z + XY + X$
$= XYZ + XZ + XY + X$

$P_{t_4} = X(Y\ o\ Z) + X$
$= X(YZ + Y) + X$
$= XYZ + XY + X$

$XYZ + XZ + XY + X > XYZ + XY + X.$
D'où $t_3 >_P t_4$.

$$f(xoy) >_A f(x)of(y) \text{ et } (xoy)oz >_A xo(yoz)$$

Comme $>_A$ est un ordre de réduction, $\Re'$ est terminant. □

4.4.3 Terminaison à l'aide d'une mesure

Une technique utilisée pour prouver qu'un système de réécriture donné termine, consiste à définir une fonction mathématique, appelée *mesure*, qui calcule la taille du membre droit et membre gauche de chaque règle du système de réécriture dans un ensemble disposant d'une relation bien fondée, puis de démontrer que cette mesure décroit pour chaque réduction par le système de réécriture [09][43].

Exemple 4.4.6. *Soit $\Re$ le système de réécriture qui représente les règles de décomposition de l'algorithme des tableaux sémantiques pour le calcul propositionnel :*

$$\Re = \begin{cases} \{\neg\neg A\} \to \{A\} \\ \{A_1 \wedge A_2\} \to \{A_1, A_2\} \\ \{\neg(A_1 \vee A_2)\} \to \{\neg A_1, \neg A_2\} \\ \{\neg(A_1 \to A_2)\} \to \{A_1, \neg A_2\} \\ \{(A_1 \equiv A_2)\} \to \{A_1 \to A_2, A_2 \to A_1\} \\ \{A_1 \vee A_2\} \to \{A_1\}, \{A_2\} \\ \{\neg(A_1 \wedge A_2)\} \to \{\neg A_1\}, \{\neg A_2\} \\ \{A_1 \to A_2\} \to \{\neg A_1\}, \{A_2\} \\ \{\neg(A_1 \equiv A_2)\} \to \{\neg(A_1 \to A_2)\}, \{\neg(A_2 \to A_1)\} \end{cases}$$

*On définit une fonction de mesure "**w**" qui associe à chaque ensemble de formules un nombre naturel ; on note **w(l)** la musure de l'étiquette du noeud **l** de l'arbre.*

*On montre que si l' est un fils de **l** alors*

$$\boldsymbol{w(l') < w(l)}$$

***Définition de la fonction w(F)**. On définit d'abord cette fonction pour une formule **f** par induction sur la structure de la formule.*

- *w(p) = 1 ;*
- *w(¬ f) = 1 + w(f) ;*
- *$w(f_1 \wedge f_2)$ = 2 + $w(f_1)$ + $w(f_2)$;*
- *$w(f_1 \vee f_2)$ = 2 + $w(f_1)$ + $w(f_2)$;*
- *$w(f_1 \to f_2)$ = 2 + $w(f_1)$ + $w(f_2)$;*
- *$w(f_1 \equiv f_2)$ = 2 * ($w(f_1)$ + $w(f_2)$) + 5 ;*

et finalement ,

$$\boldsymbol{w(F) = \sum_{f \in F} w(f)}$$

*Quelle que soit la règle de réécriture, de $\Re$, utilisée, toute étape de la construction crée un nouveau noeud l' ou deux nouveaux noeuds l', l'' tels que **w(l')<w(l)** et **w(l'')<w(l)**. Or, la fonction **w()** prend ses valeurs dans l'ensemble des entiers naturels **N** ; il ne peut donc y avoir de branche infinie.*

Par exemple, pour la règle $\{\neg(A_1 \vee A_2)\} \to \{\neg A_1, \neg A_2\}$, on a

$$w(l') = 2 + k < 3 + k = w(l).(k = w(A_1) + w(A_2)$$

Donc le système de réécriture $\Re$ termine.

Chapitre 5

Formalisation de la logique temporelle $\mathcal{LTLP}$ dans Coq

Dans ce chapitre, nous allons aborder la partie centrale de notre travail, à savoir formaliser la logique temporelle $\mathcal{LTLP}$ dans le système de preuves Coq. Cette formalisation est présentée dans la première partie de ce chapitre. En seconde partie, nous présentons une spécification des règles de décompositions de l'algorithme de la méthode des tableaux pour la logique temporelle $\mathcal{LTLP}$ et une preuve formelle de la terminaison de cet algorithme dans Coq.

5.1 Formalisation de la logique temporelle $\mathcal{LTLP}$

5.1.1 Syntaxe

Pour spécifier la définition de la syntaxe de $\mathcal{LTLP}$, présentée au chapitre 2, dans Coq et prouver formellement les propriétés de cette logique, nous devrions, en premier lieu, déclarer le type P qui représente l'ensemble des symboles propositionnels. Sa spécification correspondante en Coq est :

```
Parameter P: Set.
```

L'ensemble des formules de la logique temporelle $\mathcal{LTLP}$ peut être défini inductivement dans Coq comme suit :

```
Inductive Lterm:=
  |Tatom   : P     -> Lterm
  |Tneg    : Lterm -> Lterm
  |Tand    : Lterm -> Lterm -> Lterm
  |Tor     : Lterm -> Lterm -> Lterm
  |Timp    : Lterm -> Lterm -> Lterm
  |Tequiv  : Lterm -> Lterm -> Lterm
  |Diamond : Lterm -> Lterm
  |Next    : Lterm -> Lterm
  |Box     : Lterm -> Lterm
```

La commande *Inductive* est utilsée pour définir les types inductifs (voir chapitre 1). Le nom `Lterm` est le nom de l'objet inductivement défini et est de sorte `Type`. Les noms `Tatom, Tneg,..., Box` sont les noms des constructeurs et `P -> Lterm, Lterm -> Lterm, ...` sont leurs types.

Le terme `Tatom` introduit les variables propositionnels(atomes) comme des formules, `Tneg` indique que la négation d'une formule est une formule, `Tand` représente la conjonction de formules, `Tor` représente la disconjonction de formules, `Timp` représente l'implication de formules, *Tequiv* représente l'équivalence de formules, `Diamond` représente l'éventualité et `Next` et `Box` représentent les opérateurs prochainement et toujours.

Pour réduire la complexité des preuves (surtout les preuves inductives), on s'intéresse à limiter au minimum le nombre des constructeurs du langage. Pour se faire, les opérateurs $\vee, \rightarrow, \equiv et \Box$ sont redéfinis comme suit :

- $A_1 \vee A_2 = \neg(\neg A_1 \wedge \neg A_2)$;
- $A_1 \rightarrow A_2 = \neg A_1 \vee A_2$;
- $A_1 \equiv A_2 = (A_1 \rightarrow A_2) \wedge (A_2 \rightarrow A_1)$;
- $\Box A = \neg\Diamond\neg A$

Leurs définitions correspondantes en Coq est :

```
Definition Tor A1 A2    := Tneg(Tand(Tneg A1)(Tneg A2)).
Definition Timp A1 A2   := Tor (Tneg A1) A2.
Definition Tequiv A1 A2 := Tand(Timp A1 A2)(Timp A2 A1).
Definition Box A        := Tneg(Diamond(Tneg A)).
```

Donc `Lterm` peut être défini avec uniquement cinq constructeurs comme suit :

```
Inductive Lterm:=
  |Tatom   : P     -> Lterm
  |Tneg    : Lterm -> Lterm
  |Tand    : Lterm -> Lterm -> Lterm
  |Diamond : Lterm -> Lterm
  |Next    : Lterm -> Lterm
```

Cette déclaration crée un nouveau type, nommé `Lterm`, ainsi que cinq constructeurs `Tatom, Tneg, Tand, Diamond et Next` de `type P-> Lterm, Lterm -> Lterm, Lterm -> Lterm -> Lterm, Lterm -> Lterm, Lterm -> Lterm` respectivement. Et à partir de cette définition, le système Coq génère automatiquement trois principes de récurrence `Lterm_ind, Lterm_rec, Lterm_rect` respectivement pour les sortes `Prop`, `Set` et `Type`.

5.1.2 Sémantique

La sémantique complète de la logique temporelle $\mathcal{LTLP}$ est définie au chapitre 2. Après avoir réduit, dans la section précédente, le nombre de constructeurs du langage, nous définissons la sémantique de cette logique comme suit :

- *(σ,j)* $\models p$ ssi p est vraie à la position j,
- *(σ,j)* $\models \neg A$ ssi il est faux que *(σ,j)* $\models A$,
- *(σ,j)* $\models A_1 \wedge A_2$ ssi *(σ,j)* $\models A_1$ et *(σ,j)* $\models A_2$,
- *(σ,j)* $\models \Diamond A$ ssi $\exists k \geq j : (\sigma, k) \models A$
- *(σ,j)* $\models \bigcirc A$ ssi $(\sigma,$ *j+1*$) \models A$

L'interprétation des termes à l'instant `j` est définie, dans Coq, par récursion sur la structure des formules `F` :

```
Fixpoint  interp  (F: Lterm) (j: nat)(Inter_var: P -> Prop){struct F}: Prop :=
  match F with
    |Tatom p    => Inter_var p
    |Tneg A     => not (interp  A  j Inter_var)
    |Tand A1 A2 => (interp  A1  j Inter_var) ∧ (interp  A2  j Inter_var)
    |Diamond  A => (exists k , k >= j  ∧  (interp  A  k Inter_var))
    |Next  A    => (interp  A  (j + 1) Inter_var)
  end.
```

5.1.3 Quelques preuves des propriétés des opérateurs temporels

Maintenant que nous avons défini la sémantique de logique temporelle linéaire propositionnelle, nous pouvons prouver formellement dans l'assistant de preuve Coq les propriétés des opérateurs temporels (dualité, idempotence, absorption...) vues antérieurement au chapitre 2. Ci dessous, nous citons quelques preuves éffectuées :

Exemple 5.1.1. $\neg\Diamond p \equiv \Box\neg p$ *(Dualité)*

1$^{\text{ère}}$ ***étape :*** $\neg\Diamond p \rightarrow \Box\neg p$

```
Lemma Dual_c1: forall p j Inter_var,
     ((interp (Tneg (Diamond p)) j Inter_var))
        -> ((interp  (Box (Tneg p)) j Inter_var)).
```

```
Proof.
simpl; intros.
intro.
apply H; clear H.
inversion_clear H0.
inversion_clear H.
exists x; split.
omega.
apply NNPP.
assumption.
Qed.
```

2ème ***étape :*** $\Box\neg p \rightarrow \neg\Diamond p$

```
Lemma Dual_c2: forall p j Inter_var,
     ((interp  (Box (Tneg p)) j Inter_var))
        -> ((interp (Tneg (Diamond p)) j Inter_var)).
```

```
Proof.
simpl; intros.
intro.
apply H; clear H.
inversion_clear H0.
inversion_clear H.
exists x; split.
omega.
intro.
apply H; clear H.
assumption.
Qed.
```

3ème ***étape :*** *Démontrons maintenant l'équivalence* $\neg\Diamond p \equiv \Box\neg p$

```
Lemma Dual_C: forall p j Inter_var,
     ((interp (Tneg (Diamond p)) j Inter_var))
        <-> ((interp  (Box (Tneg p)) j Inter_var)).
```

```
Proof.
intros.
split.
apply Dual_c1.
apply Dual_c2.
Qed.
```

Exemple 5.1.2. $\Box p \rightarrow \Diamond p$*(Implication)*

```
Lemma Imp_d: forall p j Inter_var,
     (interp(Box p) j Inter_var)
        -> (interp (Next p) j Inter_var).
```

```
Proof.
simpl;intros.
apply NNPP.
intro.
apply H; clear H.
exists (j + 1); split.
omega.
assumption.
Qed.
```

Exemple 5.1.3. $\bigcirc(p \vee q) \equiv \bigcirc p \vee \bigcirc q$*(Distributivité)*
1**ère* ***étape : $\bigcirc(p \vee q) \rightarrow \bigcirc p \vee \bigcirc q$

```
Lemma Dis_d1: forall p q j Inter_var,
     (interp(Next(Tor p q)) j Inter_var)
        -> (interp(Tor(Next p)(Next q)) j Inter_var).

Proof.
simpl; intros.
exact H.
Save.
```

2**ème* ***étape : $\bigcirc p \vee \bigcirc q \rightarrow \bigcirc(p \vee q)$

```
Lemma Dis_d2: forall p q j Inter_var,
     (interp(Tor(Next p)(Next q)) j Inter_var)
        -> (interp(Next(Tor p q)) j Inter_var).

Proof.
simpl; intros.
exact H.
Save.
```

3**ème* ***étape : *Démontrons maintenant l'équivalence* $\bigcirc(p \vee q) \equiv \bigcirc p \vee \bigcirc q$

```
Lemma Dis_D: forall p q j Inter_var,
     (interp(Next(Tor p q)) j Inter_var)
        <-> (interp(Tor(Next p)(Next q)) j Inter_var).

Proof.
intros.
split.
apply Dis_d1.
apply Dis_d2.
Save.
```

Exemple 5.1.4. $\Diamond\Diamond p \equiv \Diamond p$*(Idempotence)*
1**ère* ***étape : $\Diamond\Diamond p \rightarrow \Diamond p$

```
Lemma Idemp_b1: forall p j Inter_var,
     ((interp (Diamond(Diamond p)) j Inter_var))
         -> (interp (Diamond p) j Inter_var).

Proof.
simpl;intros.
inversion_clear H.
inversion_clear H0.
```

```
inversion_clear H1.
inversion_clear H0.
exists x0; split.
omega.
auto.
Qed.
```

2ème ***étape :*** $\Diamond p \rightarrow \Diamond\Diamond p$

```
Lemma Idemp_b2: forall p j Inter_var,
     (interp (Diamond p) j Inter_var)
         -> (interp(Diamond (Diamond p)) j Inter_var).

Proof.
simpl; intros.
inversion_clear H.
inversion_clear H0.
exists x; split.
omega.
exists x; split.
omega.
auto.
Save.
```

3ème ***étape :*** *Prouvons maintenant l'équivalence* $\Diamond\Diamond p \equiv \Diamond p$

```
Lemma Idemp_B: forall p j Inter_var,
     ((interp (Diamond(Diamond p)) j Inter_var))
        <->(interp (Diamond p) j Inter_var).

Proof.
intros; split.
apply Idemp_b1.
apply Idemp_b2.
Save.
```

5.2 Preuve de la terminaison de la méthode des tableaux sémantiques dans Coq

Formaliser une preuve mathématique dans un assistant à la preuve consiste à développer cette preuve pour qu'elle soit compréhensible pour un ordinateur. Pour arriver à cet objectif deux difficultés sont à surmonter. En premier lieu, il faut expliciter les parties de la preuve qui sont implicites ou "triviales" pour un mathématicien. Paradoxalement, l'implémentation sur ordinateur de ces parties, qui n'apparaissent pas dans la preuve, est la tâche la plus complexe du travail de

formalisation. En second lieu, il faut avoir des énoncés compréhensibles pour un mathématicien. L'intérêt n'est pas simplement de faire des preuves sur ordinateurs mais il faut que les énoncés de ces preuves soient le plus proche possible de ceux utilisés dans la littérature mathématique. Ceci facilitera la réutilisation de ces preuves dans d'autres développements.

5.2.1 Les règles de décomposition

L'algorithme de la méthode des tableaux pour la logique temporelle $\mathcal{LTLP}$ (voir chapitre 3) est constitué d'un ensemble de règles de décomposition de formules, chaque règle de décomposition peut être représentée sous forme d'une règle de réécriture. L'ensemble de ces règles constitue un système de réécriture.

Soit $\Re$ le système de réécriture correspondant à l'algorithme de construction de tableaux sémantiques. Chaque formule est représentée par un ensemble. Le système $\Re$ est défini sur la signature $\Sigma = \{\neg, \wedge, \Diamond, \bigcirc\}$:

$$\Re = \begin{cases} \{\neg\neg A\} \rightarrow \{A\} \\ \{A_1 \wedge A_2\} \rightarrow \{A_1, A_2\} \\ \{\neg\Diamond A\} \rightarrow \{\neg A, \bigcirc\Diamond A\} \\ \{\neg(A_1 \wedge A_2)\} \rightarrow \{\neg A_1\}, \{\neg A_2\} \\ \{\Diamond A\} \rightarrow \{A\}, \{\bigcirc\Diamond A\} \\ \{\bigcirc A\} \rightarrow \{A\} \\ \{\neg \bigcirc A\} \rightarrow \{\neg A\} \end{cases}$$

La formalisation que nous proposons est une formalisation modulaire : chaque règle de réécriture est codée indépendamment des autres règles, et les preuves sont aussi modulaires. Cela permet d'écourter les preuves et de les rendre plus lisibles ; ainsi pour pouvoir réutiliser cette formalisation pour d'autres logiques.

L'application d'une règle de transformation est sujette à des conditions d'applicabilité. Par exemple dans la règle $\{\neg\neg A\} \rightarrow \{A\}$, il faut s'assurer que le terme A n'est pas présent dans le tableau actuel. Si les conditions sont satisfaites, on exécute une action, qui consiste à rajouter des éléments au tableau actuel.

Dans Coq, chaque règle est définie comme un prédicat binaire, codé comme prédicat inductif avec un seul constructeur. Les règles de type α et les règles de type X sont déterministes alors que les règles de type β sont indéterministes. La spécification correspondante des règles dans Coq est :

Les règles de type α

```
Inductive not_not_rule (S: ltlset)(S1: ltlset): Prop :=
  Make_not_not_rule :
    forall A, In (Tneg (Tneg A)) S -> ~In A S ->
    S1 = (add A S) ->
    not_not_rule (S: ltlset)(S1: ltlset).

Inductive and_rule (S: ltlset)(S1: ltlset): Prop:=
  Make_and_rule:
    forall A1 A2, In ( Tand A1  A2) S  -> ~In A1 S ∨ ~In A2 S ->
```

```
    S1 = (add A1 (add A2 S)) ->
    and_rule (S: ltlset)(S1: ltlset).

Inductive not_eventually_rule (S: ltlset)(S1: ltlset): Prop:=
  Make_not_eventually_rule:
    forall A, In (Tneg(Diamond A)) S -> ~In A S ->
    S1 = (add (Tneg A)(add (Tneg(Next(Diamond A))) S)) ->
    not_eventually_rule (S: ltlset)(S1: ltlset).
```

On définit l'ensemble des règles de type α pour la logique temporelle $\mathcal{LTLP}$ comme suit :

```
Definition alpha_rules S S1 :=
   (not_not_rule S S1) ∨ (and_rule S S1) ∨ (not_eventually_rule S S1).
```

Les règles de type β

Les règles de type β sont des règles indéterministes. Pour que l'on puisse les coder, on les rend déterministes par l'introduction de deux règles pour chaque règle : la règle (*Left*) introduit l'élément gauche de la disjonction tandis que la règle (*Right*) introduit l'élément droit de la disjonction.

```
Inductive not_and_rule_left (S: ltlset)(S1: ltlset): Prop:=
  Make_not_and_rule_left:
    forall A1  A2,In (Tneg (Tand A1  A2)) S -> ~In A1 S ∧ ~In A2 S ->
    S1 = (add (Tneg  A1) S) ->
    not_and_rule_left (S: ltlset)(S1: ltlset).

Inductive not_and_rule_right (S: ltlset)(S1: ltlset): Prop:=
  Make_not_and_rule_right:
    forall A1  A2, In (Tneg (Tand A1  A2)) S -> ~In A1 S ∧ ~In A2 S ->
    S1 = (add (Tneg A2) S) ->
    not_and_rule_right (S: ltlset)(S1: ltlset).

Inductive eventually_rule_left (S: ltlset)(S1: ltlset): Prop:=
  Make_eventually_rule_left:
    forall A, In(Diamond A) S ->  ~In A S ->
    S1 = (add A S) ->
    eventually_rule_left (S: ltlset)(S1: ltlset).

Inductive eventually_rule_right (S: ltlset)(S1:ltlset): Prop :=
  Make_eventually_rule_right:
   forall A, In(Diamond A) S ->  ~In A S ->
   S1 = (add (Next(Diamond A)) S) ->
   eventually_rule_right (S: ltlset)(S1: ltlset).
```

L'ensemble des règles de type β pour la logique $\mathcal{LTLP}$ est défini dans Coq comme suit :

```
Definition beta_rules S S1 :=
   (not_and_rule_right S S1) ∨ (not_and_rule_right S S1)
      ∨(eventually_rule_left S S1) ∨ (eventually_rule_right S S1).
```

Les règles de type X

```
Inductive next_rule (S: ltlset)(S1: ltlset): Prop:=
  Make_next_rule:
    forall A, In (Next A) S -> ~In A S ->
    S1 = (add A S) ->
    next_rule (S: ltlset)(S1: ltlset).

Inductive not_next_rule (S: ltlset)(S1: ltlset): Prop:=
  Make_not_next_rule:
    forall A, In (Tneg(Next A)) S ->
    S1= (add (Tneg A) S) ->
    not_next_rule (S: ltlset)(S1: ltlset).
```

On définit l'ensemble des règles de type X pour la logique $\mathcal{LTLP}$ comme suit :

```
Definition X_rules S S1 :=
   (next_rule S S1) ∨ (not_next_rule S S1).
```

Après avoir défini les differentes types de règles de décomposition, nous définissons l'ensemble des règles de la logique temporelle $\mathcal{LTLP}$ dans Coq ccmme suit :

```
Definition LTLP_rule S1 S2 :=
   (X_rules S1 S2) ∨ (beta_rules S1 S2) ∨ (alpha_rules S1 S2).
```

5.2.2 Les multi-ensembles

Les multi-ensembles sont des collections analogues aux ensembles mais pouvant contenir des doubles.

Définition 5.2.1. ***(Multi-ensemble)***
Un multi-ensemble M d'éléments dans un ensemble E est une fonction de E dans l'ensemble des entiers naturels, qui fait correspondre à chaque élément x de E le nombre M(x) d'occurrences de x dans M.

Définition 5.2.2. ***(Multi-ensemble fini)***
Un multi-ensemble est fini si son support est fini, i.e. l'ensemble des x avec image non nulle est fini. On note M(A) l'ensemble des multi-ensembles finis sur A.

Exemple 5.2.1. *M = {{0,0,2,2,2,3}} est un multi-ensemble fini d'entiers. Le nombre d'occurrences de 2 dans M est 3 (M(2) = 3), celui de 3 est 1 (M(3) = 1).*

Définition 5.2.3. *(**Les ordres multi-ensembliste**)*
$M >_{mul} N$ ssi N s'obtient à partir de M en appliquant la règle suivante un nombre fini de fois : enlever un élément x de M et le remplacer par un nombre fini d'éléments plus petits que x (par rapport à l'ordre >).

Exemple 5.2.2. *{{5,3,1,1}} $>_{mul}$ {{4,3,3,1}}.*

Théorème 6. *Si $>_A$ est un ordre strict bien fondé sur A, alors $>_{mul}$ est un ordre strict bien fondé sur les multi-ensembles de base A.*

Nous nous sommes appuyées sur une formalisation de l'ordre multi-ensembliste dans l'assistant Coq élaborée dans le projet ***CoLoR*** [44].

5.2.3 Preuve de terminaison à l'aide d'une mesure

Démontrer la terminaison de la méthode des tableaux sémantiques pour la logique temporelle $\mathcal{LTLP}$ reviens à démontrer la terminaison du système de réécriture $\Re$. Peut-on démontrer que $\Re$ se termine toujours ?

Théorème 7. *La construction d'un tableau sémantique pour la logique temporelle $\mathcal{LTLP}$ termine [26]*

Nous avons vu dans le chapitre 4 que pour prouver la terminaison d'un système de réécriture, nous pouvons utiliser une méthode qui consiste à exhiber un ordre bien fondé sur les termes, tel que, si t se réécrit en t' alors $t \gg t'$.

Formellement, on doit prouver que la relation *successeur* est une relation d'ordre bien fondé [09], c'est à dire nous devons atteindre le résultat suivant :

Théorème 8. *wellfounded succ.*

Pour prouver le théorème précédent dans Coq, on associe à chaque terme une mesure. Si cette mesure décroît pour toute règle dans un ordre bien fondé(l'ordre des entiers naturels par exemple), la terminaison est assurée.

La fonction de mesure

Principe de la démonstration : à chaque étape de la construction, les formules(termes) deviennent plus simples, mais aussi plus nombreuses ; le second fait ne compense-t-il pas le premier, et la diminution de complexité est-elle réelle ? Pour répondre avec rigueur à cette question, il faut donner une définition précise de la notion de complexité d'une formule ou d'un ensemble de formules. Plusieurs possibilités existent, par exemple : la complexité d'une formule est le nombre d'opérateurs unaires qu'elle contient, plus deux fois le nombre d'opérateurs binaires ; de plus, la complexité d'un ensemble de formules est la somme des complexités des éléments. On voit immédiatement que la complexité est un entier naturel, et que la complexité d'un noeud fils est moindre que celle du noeud père.

Formellement, la fonction `Mesure` calcule la taille d'un terme. Elle est définie comme le nombre de constructeurs de ce terme.

- Le cas où le terme T est un atome, on lui associe la valeur 0 ;

- Les cas où T est une négation ($\neg P1$) ou un opérateur temporel($\Diamond P1$ ou $\bigcirc P1$), on lui associe la valeur 1 + la mesure de P1
- Le cas où T est une conjonction($P1 \wedge P2$), on lui associe la valeur 2 plus la somme des mesures de P1 et P2.

Cette fonction est défini dans Coq comme suit :

```
Fixpoint  Mesure  T: nat :=
  match  T with
    |Tatom  x   =>  0
    |Tneg P1    => 1 + mesure P1
    |Tand P1 P2 => 2 + mesure P1 + mesure P2
    |Diamond P1 => 1 + mesure P1
    |Next P1    => 1 + mesure P1
end.
```

Pour un terme `T`, la mesure de `T` dans l'ensemble `S` est codée dans Coq par la fonction `Mesure_terme` est définie par filtrage sur la structure du terme.

Avant de donner la définition de `Mesure_terme`, nous devons définir ce qui est une règle de réécriture applicable(ou réductible).

Définition 5.2.4. *(**Règle applicable**)*

Une règle r est dite applicable s'il existe un terme t tel que se terme se réduit par cette règle en un terme t'

Nous avons défini l'applicabilité pour chaque règle de décomposition dans Coq comme suit :

- Les règles de type α

```
Inductive not_not_rule_app T (S: ltlset):Prop :=
  Make_not_not_rule_app:
     forall A, T = (Tneg (Tneg A)) -> In  T S -> ~In A S ->
        not_not_rule_app  T (S: ltlset).

Inductive and_rule_app  T (S: ltlset): Prop:=
   Make_and_rule_app :
      forall A1 A2,T = (Tand A1 A2) -> In (Tand A1 A2) S ->
         ~In A1 S ∨ ~In A2 S ->
            and_rule_app  T S.

Inductive not_eventually_rule_app T (S : ltlset): Prop :=
   Make_not_eventually_rule_app :
      forall A, T = (Tneg(Diamond A)) -> In T S -> ~In A S ->
         not_eventually_rule_app T S.
```

- Les règles de type β

```
Inductive not_and_rule_app T (S: ltlset): Prop :=
  Make_not_and_rule_app:
     forall A1 A2,T = Tneg (Tand A1 A2) ->
       In (Tneg (Tand A1 A2)) S -> ~In A1 S ∧ ~In A2 S ->
          not_and_rule_app T S.
```

```
Inductive eventually_rule_app T (S: ltlset): Prop :=
  Make_eventually_rule_app :
     forall A,T = Diamond A ->
        In(Diamond A) S ->  ~In A S ->
           eventually_rule_app T S .
```

- Les règles de type X

```
Inductive next_rule_app T (S: ltlset): Prop:=
  Make_next_rule_app :
     forall A,T = Next A -> In (Next A) S -> ~In A S ->
        next_rule_app T S.

Inductive not_next_rule_app T (S: ltlset): Prop :=
  Make_not_next_rule_app :
     forall A,T = Tneg(Next A) -> In (Tneg(Next A)) S ->
        not_next_rule_app T S.
```

Après la définition de l'applicablité de chaque règle de décomposition, nous définissons l'applicabilité d'une règle en général :

```
Definition rule_app T S :=
   not_not_rule_app T S        ∨
   and_rule_app  T S           ∨
   not_eventually_rule_app T S ∨
   not_and_rule_app T S        ∨
   eventually_rule_app T S     ∨
   next_rule_app T S           ∨
   not_next_rule_app T S.
```

Nous démontrons maintenant la décidabilité de l'applicabilité de chaque règle, c'est à dire si une règle donnée est applicable ou non.

Par exemple, la décidabilité de l'apllicabilité de la règle `not_not_rule_app` est défini dans Coq par le lemme suivant :

```
Lemma Dec_not_not_rule_app:
   forall A S, not_not_rule_app A S + ~not_not_rule_app A S.
```

La décidabilité d'une règle de décomposition est alors défini par le lemme suivant

```
Lemma Dec_rule_app: forall T S, rule_app T S + ~rule_app T S.
```

Maintenant que nous avons défini les deux notions d'applicabilité et de décidabilité pour chaque règle, nous pouvons donner la définition de Mesure_terme : si le terme T est réductible sa mesure est calculée par la fonction `Mesure`, sinon on lui attribue la valeur 0.

Sa spécification dans coq est :

```
Fixpoint  Mesure_terme T  S: nat :=
match Dec_rule_app T S with
|left _  => Mesure   T
|right _ => 0
end.
```

Exemple 5.2.3. *Soit T = {$p \wedge q$,$\Diamond\Diamond p$,$\neg(p \vee q)$, p} un terme(formule) de la logique temporelle $\mathcal{LTLP}$.*

- *Le terme $(p \wedge q)$ est réductible, sa mesure est :*
  Mesure_terme($p \wedge q$) $S = 2$.
- *Le terme $\Diamond\Diamond p$ est aussi réductible, sa mesure est :*
  Mesure_terme($\Diamond\Diamond p$) $S = 2$.
- *Le terme $\neg(p \vee q)$ est aussi réductible, sa mesure est :*
  Mesure_terme($\neg(p \vee q)$) $S = 3$.
- *Le terme $\neg p$ est irréductible, sa mesure est :*
  Mesure_terme(p) $S = 0$.

La mesure d'un ensemble de termes `Mesure_ltlset` est un multi-ensemble qui contient les mesures des termes.

```
Definition construct_Multiset (A: ltlset) (T: Lterm) M: Multiset :=
  insert (Mesure_terme T  A) M.

Definition Mesure2 A1 A2: Multiset :=
  (fold  (construct_Multiset  A1) A2 MSetCore.empty).

Definition Mesure_ltlset LT := Mesure2 LT LT.
```

Exemple 5.2.4. *Soit p, q deux variables atomiques et et soit T un ensemble de termes tel que T = {$p \wedge q$,$\Diamond\Diamond p$,$\neg(p \vee q)$, p}.*

La mesure de T est le multi-ensemble : `Mesure_ltlset T` = {{2,2,3,0}} où 2 est la mesure de $p \wedge q$, 2 est la mesure de $\Diamond\Diamond p$,3 est la mesure de $\neg(p \vee q)$ et 0 est la mesure de p.

La preuve

Le principe de la preuve est similaire pour toutes les règles : pour chaque preuve, si t se réécrit en t' et si la taille de t est strictement inférieur à t' alors la règle $t \rightarrow t'$ est terminante. Donc pour chaque application d'une règle, on doit montrer que la mesure décroît. Les lemmes suivants démontrent que la mesure décroît pour les différentes règles de décompositions :

Les règles de type α

```
Lemma not_not_decrease:
   forall S S1,not_not_rule S1 S ->
      MultisetLT gtA (Mesure2 S S) (Mesure2 S1 S1).
```

```
Lemma and_decrease:
   forall S S1, and_rule S S1 ->
      MultisetLT gtA (Mesure2 S S) (Mesure2 S1 S1).

Lemma not_eventually_decrease :
   forall  S S1, not_eventually_rule S S1
     -> MultisetLT gtA (Mesure2 S S) (Mesure2 S1 S1).
```

Les règles de type β

```
Lemma not_and_rule_left_decrease:
   forall  S S1, not_and_rule_left_rule S S1 ->
      MultisetLT gtA (Mesure2 S S) (Mesure2 S1 S1).

Lemma not_and_rule_right_decrease:
   forall  S S1, not_and_rule_right_rule S S1 ->
      MultisetLT gtA   (Mesure2 S S) (Mesure2 S1 S1).

Lemma eventually_rule_left_decrease:
   forall  S S1, eventually_rule_left_rule S S1 ->
      MultisetLT gtA   (Mesure2 S S) (Mesure2 S1 S1).

Lemma eventually_rule_right_decrease:
   forall  S S1, eventually_rule_right_rule S S1 ->
      MultisetLT gtA   (Mesure2 S S) (Mesure2 S1 S1).
```

Les règles de type X

```
Lemma next_decrease:
   forall  S S1, next_rule S1 S ->
      MultisetLT gtA   (Mesure2 S S) (Mesure2 S1 S1).

Lemma not_next_decrease:
   forall  S S1, not_next_rule S S1 ->
      MultisetLT gtA   (Mesure2 S S) (Mesure2 S1 S1).
```

Une fois cette propriété démontrée pour chaque règle, on peut généraliser cette propriété sur la relation *succ* :

```
Lemma succ-decrease:
   forall S S1, succ S S1 ->
      MultisetLT gtA (Mesure_ltlset S S) (Mesure_ltlset S1 S1).
```

La relation $<_{mul}$ est une relation bien fondée :

```
Lemma wf_multiLT: well_founded (MultisetLT gt).
```

Enfin, on peut déduire facilement que la relation `succ` est une relation d'ordre bien fondé :

```
Lemma wf_succ: well_founded succ.
```

Conclusion et perspectives

Dans notre travail, nous avons présenté une formalisation de la logique temporelle linéaire propositionnelle $\mathcal{LTLP}$ qui adopte une approche modulaire. Cette formalisation en Coq s'articule sur plusieurs modules :

- La spécification de la syntaxe et de la sémantique de $\mathcal{LTLP}$,
- Preuve de certaines propriétés de $\mathcal{LTLP}$
- La formalisation des règles de décomposition du tableau sémantique pour la $\mathcal{LTLP}$,
- Preuve de la terminaison du tableau sémantique qui nécessite la définition d'une mesure pour chaque terme et pour chaque ensemble de termes. Nous avons montré que cette mesure décroit pour chaque application d'une règle

Nous envisagerons plusieurs extensions pour ce travail, parmi les quelles on peut citer :

- Preuve de l'adéquation(soundness) de la méthode des tableaux sémantiques pour $\mathcal{LTLP}$,
- Preuve de la complétude(completeness) de la méthode des tableaux sémantiques pour $\mathcal{LTLP}$,
- L'extraction d'un raisonneur exécutable certifié,
- Extensions de $\mathcal{LTLP}$ en rajoutant par exemple des opérateurs pour le passé [27].
- Effectuer le même travail avec d'autres logiques temporelles, telles que la logique temporelles du temps arborescent.

Bibliographie

[01] Jacques-Louis Lions et al. Ariane 5 flight 501 failure report by the inquiry board. Technical report, European Space Agency, Paris, France, 1996.

[02] Information Management and Technology Division. Patriot missile defense : software problem led to system failure at Dhahran, Saudi Arabia. Report B-247094, United States General Accounting Office, 1992. http ://www.fas.org/spp/starwars/gao/im92026.htm.

[03] Debbie Gage and John McCormick. We did nothing wrong. Baseline. http ://www.baselinemag.com/

[04] Christel Baier and Joost-Pieter Katoen. Principles of Model Checking. The MIT Press. Cambridge, Massachusetts, London, England.

[05] Francisco José CHÁVES ALONSO. Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves. École Normale Supérieure de Lyon. Septembre 2007.

[06] The HOL Light theorem prover. http ://www.cl.cam.ac.uk/users/jrh/hol-light

[07] The Coq Proof Assistant Reference Manual - Version 8.2. Technical report, INRIA, September 2008. http ://coq.inria.fr.

[08] Julien NARBOUX. Formalisation et automatisation du raisonnement géométrique en Coq. Thèse de doctorat. Université Paris XI Orsay. Septembre 2006.

[09] M.CHAABANI, M.MEZGHICHE et M.STRECHER. Formalisation de la logique de description ALC dans l'assistant de preuve Coq. JFO 2009, Poitiers, France. Décembre 2009.

[10] M. D'AGOSTINO, D.M. GABBAY, R. HÄHNLE and J.POSEGGA,eds. Handbook of Tableau Methods. Ferrara, London, Karlsruhe and Darmstadt, March 1998.

[11] Nguyen Quang Huy. Calcul de réécriture et automatisation du raisonnement dans les assistants de preuve. Thèse de Doctorat. Université Henri Poincaré-Nancy 1. Octobre 2002.

[12] Pierre-Alain MASSON. Vérification par model-checking modulaire de proporiétés dynamiques PLTL exprimées dans le cadre de spécification B événementielles. Thèse de Doctorat. UFR des sciences et techniques. Université de Franche-Comté, 2001.

[13] L. MOHAND OUSSAID. Vérification formelle des propriétés de sécurité des logiciels. Mémoire de magister. Institut National de Formation en Informatique,(INI) Oued Smar, 2006.

[14] Norbert SCHIRMER. Verification of Sequential Imperative Programs in Isabelle/HOL. Thèse de doctorat. Université de MUNICH, 2005.

[15] Ricky Butler, Jeff Maddalon, Alfons Geser, and César Muñoz. Formal verification of a conflict resolution and recovery algorithm. Technical Paper NASA/TP-2004-213015, NASA Langley Research Center, Hampton, VA,2004.

[16] M.CHAABANI. Specification, Preuve et Extraction Automatique des programmes en Coq. Mémoire de magister, Université M'hamed BOUGARA de Boumerdes, 2003.

[17] Razika LOUNAS. Preuve en Coq de propriétés de programmes numériques partant du code en C. Mémoire de magister, Université M'hamed BOUGARA de Boumerdes, 2009.

[18] Stéphane GLONDU. Extraction certifiée dans Coq-en-Coq. Laboratoire Preuves, Programmes et systèmes.Université Paris Diderot - Paris 7. Journées Francophones des Langages Applicatifs. Janvier 2009.

[19] Tomasz BLANC. Propriétés de sécurité dans le λ-calcul. Thèse de doctorat. Ecole polytechnique. Novembre 2006.

[20] Nicolas OURY. Egalité et filtrage avec types dépendants dans le calcul des constructions inductives. Thèse de doctorat. Université de Paris XI Orsay. Sepembre 2006.

[21] Bruno BARRAS. Auto-validation d'un système de preuves avec familles inductives. Thèse de doctorat. Université Paris 7 - Denis Diderot. Novembre 1999.

[22] Yves BERTOT, Pierre CASTERAN. Interactive Theorem Proving and Program Development, Coq'Art : the calculus of inductive constructions. Springer-Verlag , 2004..

[23] Naim ABER. Vérification d'un protocole Stop-And-Wait : Combiner Model-Checking et Preuve de théorèmes. Rapport de stage. Université de versailles Saint-Quentin-En-Yvelines. 2008/2009.

[24] David DELAHAYE. Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve. Thèse de doctorat. Université Paris 6 - Pierre et Marie CURIE, 2001.

[25] Alexandre MIQUEL. De la formalisation des preuves à l'extraction de programmes. Mémoire d'habilitation à diriger des recherches. L'université Paris Dideros - Paris 7. Décembre 2009.

[26] M.BEN-ARI. Mathematical Logic For Computer Science. Technion-Israel Institute of technology. Prentice Hall International(UK)Ltd,1993.

[27] Nicolas MARKEY. Logiques temporelles pour la vérification : expressivité, complexité, algorithmes. Thèse de doctorat. Université d'Orléans. Avril 2003.

[28] Alexandre DURET-LUT, Rachid REBIHA. SPOT : une bibliothèque devérification de propriétés de logique temporelle à temps linéaire. Mémoire de DEA. LIP6,Paris/LIFO,Orléans. Septembre 2003.

[29] Susanne GRAF. Logiques du temps arborescent pour la specification et la preuve de programmes. Thèse de doctorat. Institut national poluthechnique de Grenoble. Février 1984.

[30] Imed JARRAS. Vérification et synthèse d'un réseau de petri relativement à une spécificaion logique temporelle. Mémoire. Université Laval. Juillet 1995.

[31] Vincent ARAVANTINOS. Formulas and proofs schemas in propositional logic.Thèse de doctorat. Université de Grenoble. Septembre 2010.

[32] Dario VIEIRA et Ana CAVALLI. Vérification et analyse de performance d'un protocole de gestion de session. Laboratoire Samovar(CNRS)andGET/INT. 2006.

[33] Alexandre DURET-LUTZ. Contributions à l'approche automate pour la vérification de propriétés de systèmes concurrents. Thèse de Doctorat. Université Pierre et Marie CURIE, paris 6. Juillet 2007.

[34] J. GAINTZARAIN, M. HERMO, P. LUCIO and M. NAVARRO. Systematic Semantic Tableaux for PLTL. Spanish Project TIN2004-079250-C03. www.elsevier.nl/locate/entcs. 2004.

[35] Akash DESHPANDE and Pravin VARAIYA. Semantic tableau for control of PLTL Formulae. University of california at Berkley, Berkley, Californmnia 94720.

[36] Bettina Kemme. Algoritmos de satisfactibildad y Model-Checking para la ogica Temporal Proposicional : Comparacion y Aplicacion para la Representacion de Conocimientos temporales. Université de Sevilla.

[37] Yves GUIRAUD. Présentations d'opérades et systèmes de réécriture. Université Montpellier II. Thèse de doctorat. Mai 2004.

[38] Florent GARNIER. Terminaison en temps moyen fini de systèmes de règles probabilistes. Institut National Polytechnique de Lorraine. Thèse de doctorat. Septembre 2007.

[39] Sébastien HINDERER. Certification des preuves de terminaison par interprétations polynomiales. Rapport de stage. Université Hené Poincaré, Nancy. 2004.

[40] Léo DUCAS. Certification de preuves de terminaison basées sur la décomposition du graphe des paires de dépendance. Stage de fin de licence. Laboratoire du Loria, Nancy. Juillet-Août 2007.

[41] CHeikh.SALMI. Confluence et préservation de normalisation forte dans un système de réécriture d'ordre supérieur . Mémoire de magister, Université M'hamed BOUGARA de Boumerdes, 2006.

[42] Ahlem BEN CHERIFA and Pierre LESCANNE. Termination of rewriting systems by polynomial interpretations and its implementation. INRIA. Rapport de recherche. Juin 1987.

[43] Claude MARCHE. Preuves mécanisées de propriétés de programmes. Thèse d'habilitation à diriger des recherches. Université Paris XI. Décembre 2005.

[44] A Coq Library on Rewriting and Termination. http ://color.inria.fr