UNIVERSITÉ PARIS DIDEROT
UFR Informatique
Département de formation de Licence L3

La logique temporelle LTL

Mémoire présenté par :
Félix DESMARETZ
Yuchen BAI

Devant le jury :
François Laroussinie

05/2019 - 06/2019

Table des matières

Table des matières

1. Introduction générale

1.1 Introduction

La logique temporelle linéaire (LTL) a beaucoup évolué depuis son introduction en informatique dans les années 1970. De nos jours, il est largement utilisé dans divers domaines tels que la vérification et l'analyse de programmes, la synthèse de programmes, la base de données et l'intelligence artificielle.

LTL est une sorte de langage logique dont le principal problème est le problème de la satisfiabilité. Pendant ce temps, le problème de la satisfiabilité est également important dans l'industrie car, en termes de vérification de programme, de synthèse de programme et d'intelligence artificielle, une formule non satisfaisante n'a pas de sens.

Pour la programmation réactive, l'exactitude dépend non seulement de l'entrée et de la sortie de l'opération, mais également de la séquence d'exécution du système. La logique temporelle peut décrire la temporalité de la logique et décrire la séquence d'exécution infinie de système en étendant la logique propositionnelle et la logique des prédicats. En général, la logique temporelle est une logique qui contient un facteur temps, un facteur qui ajoute du temps à la logique. À la fin des années 1950, Prior a proposé deux types d'opérateurs temporels, en utilisant "F" et "P" pour indiquer "dans le futur (Future)" et "dans le passé (Past)".

La logique temporelle a deux branches, la logique temporelle linéaire (LTL: Linear temporal logic) et la logique du temps arborescent (CTL: Computation tree logic). La vue linéaire considère qu'en même temps, un état n'a qu'un seul successeur, mais le point de vue est qu'il existe une structure de branche en forme d'arbre et qu'un état a plusieurs successeurs ultérieurs possibles. À l'heure actuelle, les deux branches de la logique temporelle ont beaucoup progressé. Nous ne cherchons pas à savoir si l'essence du temps est linéaire ou ramifiée. Dans cet article, nous nous concentrons uniquement sur la logique temporelle linéaire.

À la fin des années 1970, Pnueli a introduit la logique temporelle dans le domaine de l'informatique et l'a utilisé comme langage de spécification formalisé pour la spécification et la vérification d'outils concurrents en informatique. Il a donc acquis la communauté informatique en 1996. La plus haute récompense - la récompense de

Turing. En 1977, Pnueli proposa la "Logique temporelle linéaire future", désormais la logique temporelle linéaire standard, qui inclut deux nouveaux opérateurs temporels: X (Next) et U (Until).
La capacité d'expression de LTL est équivalent à la logique des prédicats de premier ordre, mais la difficulté de résolution dans le problème de la satisfiabilité et de la Raisonnement déductif est très différente. La complexité de la logique des prédicats de premier ordre pour ces problèmes est infiniment grande, mais pour LTL, la difficulté de résolution de ces problèmes est PSPACE-complete. C'est pourquoi LTL peut être largement utilisé.

1.2 Vérification

La première fois que la logique temporelle a été introduite dans le domaine de l'informatique a été de la décrire comme un langage de spécification comportemental. De nombreux outils de vérification de modèle utilisent LTL comme langage de spécification, par exemple SPIN.

Dans cet article, on va utiliser l'algorithme de model-checking pour vérifier le modèle d'un système informatique satisfait une propriété ou non. Par exemple, on souhaite vérifier qu'un programme ne se bloque pas, qu'une variable n'est jamais nulle, etc. Généralement, la propriété est écrite dans un langage, souvent en logique temporelle. La vérification est généralement faite de manière automatique.

Supposons qu'on a un modèle M et une propriété φ, ¬ φon prend en un modèle exprimé dans un formalisme imposé, et une spécification qui exprime la propriété φ que doivent vérifier certaines données du modèle. Il effectue ensuite un calcul à partir de ces données, et peut produire deux résultats différents : soit toutes les exécutions du modèle satisfont la spécification, soit au moins une exécution du modèle ne satisfait pas la spécification, et dans ce cas le résultat est négatif et l'automate donne la exécution comme un contre-exemple d'exécution non satisfaisante.

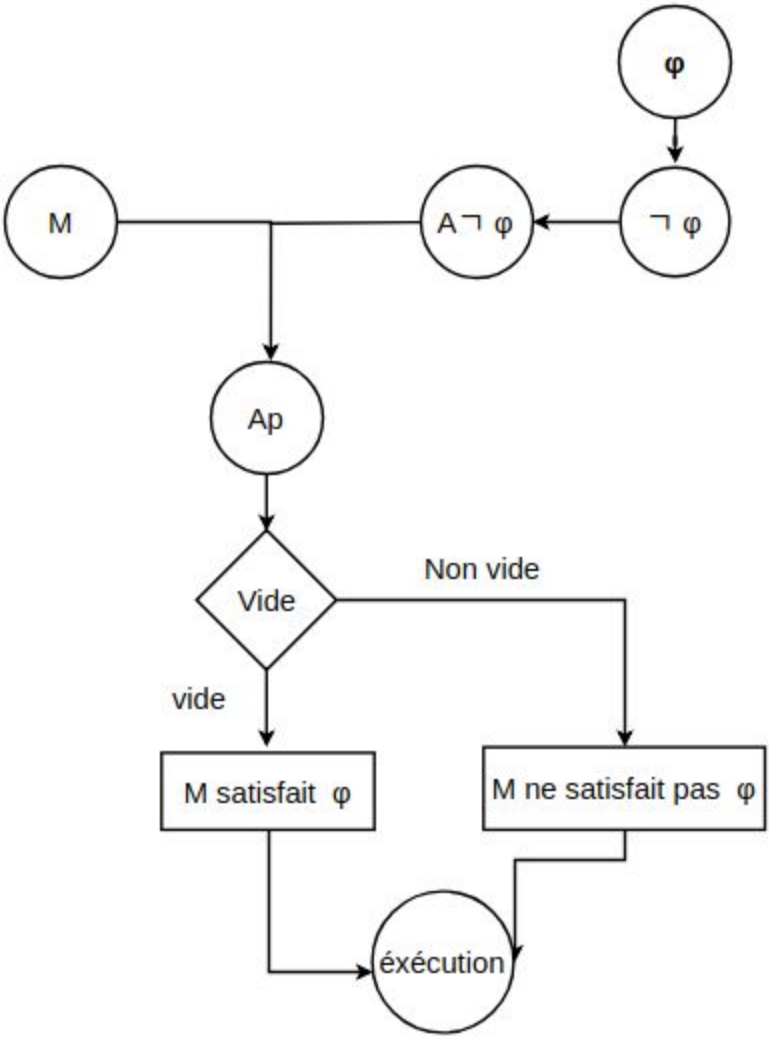


FIG. 1.1

La complexité de l'utilisation de cette méthode pour résoudre le problème de vérification du modèle dépend de la taille de l'automate A. Si l'espace de recherche de A est très grand, il faut beaucoup de temps pour déterminer si l'exécution est vide ou non.

2. Logique temporelle linéaire

Dans ce partie, on va parler comment modéliser une application, exprimer des spécifications et détailler les algorithmes de model-checking.

2.1 Notation de base

On note AP un ensemble de propositions atomiques, et soit Σ = 2^{AP} . Nous notons Σ^{ω} l'ensemble des mots infinis sur l'alphabet Σ.

2.1.1 Syntaxe

Une syntaxe est l'opérateurs de logique classique plus des opérateurs du futur et du passé.

Les formules LTL sont définies ainsi :

– ⊥ (false), et tous les éléments p dans AP sont des formules LTL,
– si ϕ et ψ sont des formules LTL, alors ¬ϕ (not ϕ), $\phi \vee \psi$ (ϕ or ψ), X ϕ (next ϕ) et ϕ U ψ (ϕ until ψ) sont des formules de LTL.

La sémantique de LTL définit si une exécution d'un système donné satisfait une formule.

- **X** est lu comme *suivant* (ne**x**t en anglais)
- **U** est lu comme *jusqu'à* (**u**ntil en anglais)
- **G** pour toujours (**g**lobalement (***g****lobally*** en anglais))
- **F** pour éventuellement (dans le **f**utur (*in the **f**uture* en anglais))
- **R** pour libération (***r****elease* en anglais)
- **W** pour faible jusqu'à (***w****eakly until* en anglais)

2.1.2 Sémantique

Une sémantique est domaine des objets (appelés modèles) sur lesquels on va tester la validité des formules, plus l'interprétation des opérateurs.

Une formule de LTL peut être satisfaite par une suite infinie d'évaluations de vérité des variables dans *AP*. Soit $w = a_0,a_1,a_2,...$ tel un mot-ω. Soit $w(i) = a_i$. Soit $w^i = a_i,a_{i+1},...$, qui

est un suffixe de *w*. Formellement, la relation de satisfaction $\models$ entre un mot et une formule de LTL est définie comme suit:

- $w \models p$ si $p \in w(0)$
- $w \models \neg\psi$ si $w \models \psi$
- $w \models \varphi \vee \psi$ si $w \models \varphi$ ou $w \models \psi$
- $w \models \mathbf{X}\,\psi$ si $w^1 \models \psi$ (ψ doit être vrai à l'étape *suivante*)
- $w \models \varphi\ \mathbf{U}\ \psi$ s'il existe $i \geq 0$ tel que $w^i \models \psi$ et pour tout $0 \leq k < i$, $w^k \models \varphi$ (φ doit rester vrai *jusqu'à ce que* ψ devienne vrai)

On dit qu'un mot-ω *w* satisfait une formule LTL ψ quand $w \models \psi$.

Les opérateurs logiques supplémentaires sont définis comme suit:

- $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$
- $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
- **vrai** $\equiv p \vee \neg p$, où $p \in AP$
- **faux** $\equiv \neg$**vrai**

Les opérateurs temporels supplémentaires **R**, **F** et **G** sont définis comme suit:

- $\varphi\ \mathbf{R}\ \psi \equiv \neg(\neg\varphi\ \mathbf{U}\ \neg\psi)$
- $\mathbf{F}\,\psi \equiv$ **vrai** $\mathbf{U}\ \psi$ (ψ devient éventuellement vrai)
- $\mathbf{G}\,\psi \equiv$ **faux** $\mathbf{R}\ \psi \equiv \neg\mathbf{F}\ \neg\psi$ (ψ reste toujours vrai)

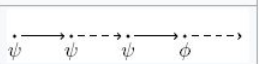
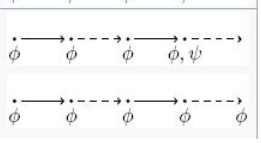
Opérateur	Symbole alternatif	Explication	Diagramme
Opérateurs unaires:			
$\mathbf{X}\,\phi$	$\bigcirc\phi$	suivant: ϕ doit être satisfaite dans l'état suivant.	
$\mathbf{G}\,\phi$	$\Box\phi$	Globalement: ϕ doit être satisfaite dans l'intégralité des états futurs.	
$\mathbf{F}\,\phi$	$\Diamond\phi$	Finalement: ϕ doit être satisfaite dans un état futur.	
Opérateurs binaires:			
$\psi\ \mathbf{U}\ \phi$	$\psi\ \mathcal{U}\ \phi$	Jusqu'à : ψ doit être satisfaite dans tous les états jusqu'à un état ou ϕ est satisfaite, non inclus.	
$\psi\ \mathbf{R}\ \phi$	$\psi\ \mathcal{R}\ \phi$	Réaliser : ϕ doit être satisfaite dans tous les états tant que ψ n'est pas satisfaite. Si ψ n'est jamais satisfaite, ϕ doit être vraie partout.	

FIG. 2.1

2.1.3 Modèle

Une formule LTL se rapporte toujours à une trace donnée σ d'un système. Les traces constituent les modèles de cette logique.

Note: au lieu de l'*état*, on parle aussi d'*instant*.

2.1.4 taille temporelle d'une formule

La taille temporelle $|\phi|$ mesure le nombre d'opérateurs temporels d'une formule ϕ. Elle est définie par induction comme suit :

– $|>| = |\bot| = |p| = 1$ pour tout p dans AP,

– $|\neg\phi| = |\phi|$, $|\phi \vee \psi| = |\phi| + |\psi|$, $|X\ \phi| = 1 + |\phi|$, $|\phi\ U\ \psi| = 1 + |\phi| + |\psi|$.

2.2 Normalisation et simplification

2.2.1 Normalisation

Avant d'effectuer d'autres opérations sur la formule, nous devons normaliser la formule pour faciliter les autres opérations.
L'opération de normalisation d'une formule comprend les deux types d'opérations suivants:

1. Supprimez les opérateurs <->, ->, G et F, et nous les réécrivons en appliquant les règles suivantes:
   a. `a -> b ≡ ¬a ∨ b`
   b. `a <-> b ≡ (¬a ∨ b) ∧ (¬b ∨ a)`
   c. `G a ≡ False R a`
   d. `F a ≡ True U a`

2. Pour s'assurer que l'opérateur ¬ n'apparaisse que devant l'atome, nous le réécrivons en appliquant les règles suivantes:
   a. `¬True ≡ False`
   b. `¬False ≡ True`
   c. `¬¬a ≡ a`

d. ¬(Xa) ≡ X(¬a)
e. ¬(G a) ≡ True U ¬a
f. ¬(F a) ≡ False R ¬a
g. ¬(a U b) ≡ ¬a R ¬b
h. ¬(a R b) ≡ ¬a U ¬b
i. ¬(a ⋀ b) ≡ ¬a ⋁ ¬b
j. ¬(a ⋁ b) ≡ ¬a ⋀ ¬b
k. ¬(a->b) ≡ a ⋀ ¬b
l. ¬(a<->b) ≡ (a ⋀ ¬b) ⋁ (¬a ⋀ b)

Par exemple, si on a la formule : ¬((a -> Xb) U (Ga)), on peut le transfert par les règles ci-dessus:

1. ¬((a -> Xb) U (Ga)) = ¬(¬a ⋁ Xb) R ¬(False R a)
2. ¬(¬a ⋁ Xb) R ¬(False R a) = (¬¬a ⋀ ¬Xb) R (¬False U ¬a)
3. (¬¬a ⋀ ¬Xb) R (¬False U ¬a) = (a ⋀ X(¬b)) R (True U ¬a)

2.2.2 Simplification

La simplification permet de raccourcir la longueur de la formule. Ainsi, une fois la formule normalisée, nous utilisons les règles suivantes pour simplifier la formule:

1. Simplification de AND (⋀):
   a. True ⋀ a ⋀ ... ≡ a ⋀ ... (S1.1)
   b. False ⋀ a ⋀ ... ≡ False (S1.2)
   c. a ⋀ a ⋀ ... ≡ a ⋀ ... (S1.3)

2. Simplification de OR (V):
   a. False ⋁ a ⋁ ... ≡ a ⋁ ... (S2.1)
   b. True ⋁ a ⋁ ... ≡ True (S2.2)
   c. a ⋁ ¬a ⋁ ... ≡ True (S2.3)
   d. a ⋁ a ⋁ ... ≡ a ⋁ ... (S2.4)
   e. a ⋁ b U (¬a ⋁ ...) ≡ True (S2.5)
   f. b U (a ⋁ ...) ⋁ c U (¬a ⋁ ...) ≡ True (S2.6)
   g. a ⋁ b R a ⋁ ... ≡ a ⋁ ... (S2.7)
   h. a ⋁ b U a ⋁ ... ≡ b U a ⋁ ... (S2.8)

3. Simplification de NEXT(X):
   a. X True ≡ True (S3.1)
   b. X False ≡ False (S3.2)

4. Simplification de UNTIL(U):
    a. `False U a ≡ a` (S4.1)
    b. `a U False ≡ False` (S4.2)
    c. `a U True ≡ True` (S4.3)
    d. `a U (a ∨ ...) ≡ a ∨ ...` (S4.4)
    e. `a U (a U b) ≡ a U b` (S4.5)
    f. `a U (b U a) ≡ b U a` (S4.6)
    g. `a U (b R a) ≡ b R a` (S4.7)
    h. `(b R a) U a ≡ a` (S4.8)
    i. `(a U b) U a ≡ b U a` (S4.9)
    j. `(b U a) U a ≡ b U a` (S4.10)
    k. `X a U a ≡ X a ∨ a` (S4.11)
    l. `X a U X b ≡ X ( a U b )` (S4.12)

5. Simplification de RELEASE(R):
    a. `True R a ≡ a` (S5.1)
    b. `a R False ≡ False` (S5.2)
    c. `a R True ≡ True` (S5.3)
    d. `a R (a ∧ ...) ≡ a ∧ ...` (S5.4)
    e. `(a ∨ ...) R a ≡ a` (S5.5)
    f. `a R (a R b) ≡ a R b` (S5.6)
    g. `a R (b R a) ≡ b R a` (S5.7)
    h. `a R (b U a) ≡ b U a` (S5.8)
    i. `(b U a ∨ ... ) R a ≡ a` (S5.9)
    j. `(a R b) R a ≡ b R a` (S5.10)
    k. `(b R a) R a ≡ b R a` (S5.11)
    l. `X a R X b ≡ X ( a R b )` (S5.12)
    m. `¬a R a ≡ False R a` (S5.13)
    n. `(b R (¬a ∧ ...) ∧ ...) R a ≡ False R a` (S5.14)

Par exemple, si nous avons la formule (((a ∨ b) R (c R (a ∨ b)) U (a ∨ b)) R a) ∧ a, alors nous pouvons le simplifier par les règles ci-dessus:

1. Par S5.7 : (((a ∨ b) R (c R (a ∨ b)) U (a ∨ b)) R a) ∧ a = ((c R (a ∨ b)) U (a ∨ b) R a) ^ a ;
2. Par S4.8 : ((c R (a ∨ b)) U (a ∨ b) R a) ^ a = ((a ∨ b) R a) ^ a ;
3. Par S2.7 et N2.8 : ((a ∨ b) R a) ^ a = a ^ a ;
4. Par S1.3 : a ^ a = a.

2.3 Exemple : Spécification d’un ascenseur

«Up and Down the Temporal Way» était un article publié par Howard Barringer dans les années 1980 qui utilisait des logiques temporelles pour spécifier formellement une spécification d’un ascenseur.

2.3.1 Spécification

On a les hypothèse suivantes:

- Une porte est ouverte ou fermée.
- Un bouton est enfoncé ou relâché.
- Un voyant est allumé ou éteint.
- La cabine est présente à l'étage i ou il est absente.

Et nous posons:

- $D : array\ (1..m)\ of\ DoorState\ \ avec\ DoorState\ =\ OpenDoor|ClosedDoor$
- $C : array\ (1..m)\ of\ Boolean\ avec\ S\ repr$ésente appeler un ascenseur
- $S : array\ (1..m)\ of\ Boolean\ avec\ C\ repr$ésente envoyer un ascenseur
- $LC : array\ (1..m)\ of\ Boolean$ avec LC représente le voyant associé le bouton appelé
- $LS : array\ (1..m)\ of\ Boolean$ avec LS représente le voyant associé le bouton d’envoie

2.3.2 Exemples

(1) Si l'un des boutons d'appel est enfoncé, il reste allumé et reste allumé sauf si la demande est traitée. L'ascenseur doit être présent et pouvoir accueillir des passagers.
$G\ (\ \wedge_i\ (\ C_i\ \Rightarrow\ (\ LC_i\ U\ service\ at\ i\ )\ )\ )$ quand $service\ at\ i\ = D_i\ \in\ OpenDoors$

(2) Si le bouton d’envoi est enfoncé, il est allumé ou l’ascenseur est à l’étage et peut fournir un service immédiat.
$G\ (\ \wedge_i\ (\ S_i\ \Rightarrow\ (\ LS_i\ U\ service\ at\ i\ )\ )\ )$

(3) Si jamais un voyant d'appel est éteinte, elle doit rester éteinte sauf si vous appuyez sur le bouton d'appel.
$G\,(\,\wedge_i\,(\,\neg LC_i\;\Rightarrow\,(\,\neg LC_i\;U\;C_i\,)\,)\,)$

(4) Si jamais un voyant d'envoie est éteinte, elle doit rester éteinte sauf si vous appuyez sur le bouton d'envoie.
$G\,(\,\wedge_i\,(\,\neg LS_i\;\Rightarrow\,(\,\neg LS_i\;U\;S_i\,)\,)\,)$

(5) Il est nécessaire d'exprimer que, si une lumière est allumée, il n'ya pas de service immédiat pour le moment.
$G\,(\,\wedge_i\,(\,(\,LC_i\;\Rightarrow\,\neg service\;at\;i\,)\,\wedge\,(LS_i\;\Rightarrow\,(\,\neg service\;at\;i\,))\,)$

(6) En réalité, les cinq propriétés ci-dessus peuvent être combinées en une propriété suivante.
$G\,(\,\wedge_i\,(\,C_i\;\Rightarrow\,(\,LC_i\;U\;service\;at\;i\,)\,)\;\wedge(\,(\,service\;at\;i\,\vee\neg LC_i\,)\;\Rightarrow\,(\neg LC_i\;U^{+}\;C_i)\,)\wedge$
$(\,S_i\;\Rightarrow\,(\,LS_i\;U\;service\;at\;i\,)\,)\,\wedge\,(\,(service\;at\;i\;\vee\neg LS_i\,)\;\Rightarrow\,(\neg LS_i\;U^{+}\;S_i)\,)$

(7) Soit toutes les portes sont correctement fermées, soit au plus une porte est ouverte.
$G\,(\,(\,\sum_i D_i\;\in\;OpenDoors\,)\leq 1)$

3 Automate

3.1 Automate de Büchi

En informatique théorique, un automate de Büchi est un automate fini opérant sur des mots infinis, avec une condition d'acceptation particulière : une chemin infini est réussie si et seulement si elle passe un nombre infini de fois par au moins un état acceptant. Un mot infini est accepté s'il est l'étiquette d'un calcul réussi.

3.1.1 Définition

Un automate de Büchi est un quintuplet B = (Q, Σ, δ, I, R) ou :
– Q est l'ensemble fini des états,
– Σ est l'alphabet,
– δ : Q × Σ → 2^Q est la fonction de transition,

– I ⊆ Q est l'ensemble des états initiaux,
– R ⊆ Q est l'ensemble des états répétés.

Soit u = u1u2 . . . ∈ Σ^{ω} . Une exécution ρ de B sur u est une suite infinie q0, q1, . . . d'éléments de Q telle que : – le premier état est initial : q0 ∈ I, – on passe d'un état au suivant en suivant la fonction de transition : ∀i ≥ 1, qi ∈ δ(qi−1, ui). Une ex´ecution ρ est acceptante si, de plus, une infinit´e de qi sont dans l'ensemble R. Le langage L(B) d'un automate de Büchi B est l'ensemble des mots de Σ ω sur lesquels il existe une exécution acceptante de B.

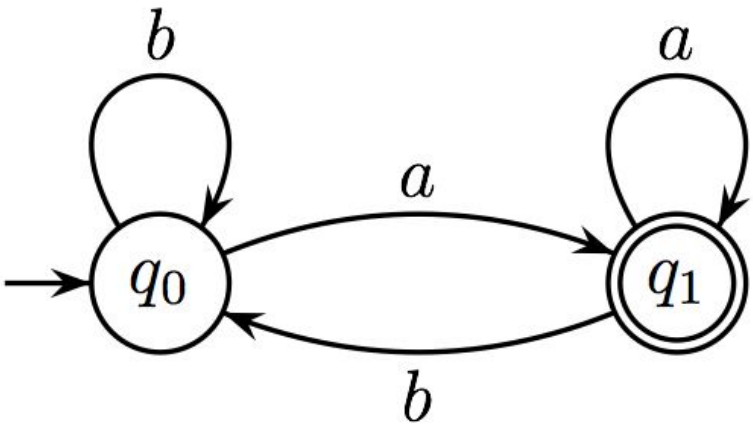


FIG 2.3.1
Automate de Büchi reconnaissant les mots infinis contenant un nombre infini de a.

3.2 Automate fini alternant

Dans les années 90, Moshe a proposé une méthode pour juger de la satisfiabilité de la formule LTL: en développant la formule LTL dans la forme GBA (Generalized Büchi automaton)puis en convertissant la GBA en Automate de Büchi, si la langue acceptée du Automate de Büchi est vide, la formule LTL n'est pas satisfaite, sinon la formule LTL est satisfaite. Nous découvrirons que si nous utilisons cette méthode pour évaluer la satisfaisabilité de la formule de LTL, nous devons construire un automate complet.

3.2.1 Algorithme de LTL à Büchi automate

La première étape produit un automate de Büchi généralisé (GBA) à partir d'une formule LTL, la seconde étape la convertissant en BA, ce qui implique une construction relativement simple.Le LTL étant strictement moins expressif que BA, la construction inverse n'est pas possible.

3.2.1.1 Construction déclarative

Avant de décrire la construction, nous devons présenter quelques définitions auxiliaires. Pour une formule LTL f, Soit cl (f) un plus petit ensemble de formules qui remplit les conditions suivantes:

- **true** $\in$ *cl*(f)
- f $\in$ *cl*(f)
- si $f_1 \in$ *cl*(f) alors *neg*(f_1) $\in$ *cl*(f)
- si **X** $f_1 \in$ *cl*(f) alors $f_1 \in$ *cl*(f)
- si $f_1 \wedge f_2 \in$ *cl*(f) alors f_1,$f_2 \in$ *cl*(f)
- si $f_1 \vee f_2 \in$ *cl*(f) alors f_1,$f_2 \in$ *cl*(f)
- si f_1 **U** $f_2 \in$ *cl*(f) alors f_1,$f_2 \in$ *cl*(f)
- si f_1 **R** $f_2 \in$ *cl*(f) alors f_1,$f_2 \in$ *cl*(f)

Nous définissons que cl(f) est la clôture de sous-formules de f sous négation. Notez que cl(f) peut contenir des formules qui ne sont pas sous forme normale de négation. Les sous-ensembles de cl(f) vont servir d'états de la GBA équivalente. Notre objectif est de construire la GBA de telle sorte que si un état correspond à un sous-ensemble M $\subset$ cl(f), alors la GBA a un parcours acceptant commençant à partir de l'état d'un mot iff si le mot satisfait chaque formule de M et viole chaque formule de cl (f) / M. Pour cette raison, nous ne considérerons pas chaque ensemble de formules M clairement incohérent ni subsumé par un super ensemble strictement M 'tel que M et M' soient équivalents. Un ensemble M $\subset$ cl (f) est au maximum cohérent s'il remplit les conditions suivantes:

- **true** $\in$ M
- $f_1 \in$ M iff $\neg f_1 \notin$ M
- $f_1 \wedge f_2 \in$ M iff $f_1 \in$ M and $f_2 \in$ M
- $f_1 \vee f_2 \in$ M iff $f_1 \in$ M or $f_2 \in$ M

Nous se mettrons cs(f) l'ensemble des sous-ensembles à la cohérence maximale de cl(f). Nous allons utiliser uniquement cs(f) comme états de l'GBA.

3.2.1.2 La construction de GBA

Un GBA équivalent à f est A = (I ∪ cs(f), 2^{AP} , Δ, I, F), où :

- Δ = Δ1 ∪ Δ2
  - (M, a, M ') Δ1 ssi (M' ∩ AP) ⊆ a ⊆ {p ∈ AP | ¬p ∉ M '} et:
    - **X** f1 ∈ M ssi f1 ∈ M ';
    - f1 **U** f2 M ssi f2 ∈ M ou (f1 ∈ M et f1 U f2 ∈ M ');
    - f1 **R** f2 M ssi f1 ∧ f2 ∈ M ou (f2 ∈ M et f1 **R** f2 ∈ M ')
  - Δ2 = {(I, a, M') | (M '∩AP) ⊆ a ⊆ {p ∈ AP | ¬p∉ M '} et f ∈ M'}
- Pour chaque f1 **U** f2 ∈ cl(f), {M ∈ cs (f) | f2 ∈ M ou ¬ (f1 U f2) ∈ M} ∈ F

Les trois conditions de la définition de Δ1 garantissent que toute exécution de A ne viole pas la sémantique des opérateurs temporels. Notez que F est un ensemble d'ensembles d'états. Les ensembles dans F sont définis pour capturer une propriété de l'opérateur U qui ne peut pas être vérifiée en comparant deux états consécutifs dans une exécution, c'est-à-dire, si f1 U f2 est vraie dans un état puis finalement f2 est vraie à un état ultérieur.

3.2.1.3 L'algorithme

L'algorithme suivant est dû à Gerth, Peled, Vardi et Wolper. La construction précédente créé de manière exponentielle de nombreux états et ceux-ci peuvent être inaccessibles. L'algorithme suivant évite cette construction initiale et comporte deux étapes. Dans la première étape, il construit progressivement un graphe dirigé. Dans un deuxième temps, il construit un automate de Büchi généralisé (GBA) étiqueté en définissant les nœuds du graphe en tant que l'états et les arêtes dirigées en transitions. Cet algorithme prend en compte l'accessibilité et peut produire un automate plus petit mais la complexité dans le pire des cas reste la même.

Les nœuds du graphe sont étiquetés par des ensembles de formules et sont obtenus en décomposant les formules en fonction de leur structure booléenne et en développant les opérateurs temporels afin de séparer immédiatement ce qui doit être vrai de ce qui doit être vrai. .

Par exemple, supposons qu'une formule LTL: f1 **U** f2 apparaisse dans l'étiquette d'un nœud. f1 **U** f2 est équivalent à f2 ∨ (f1 ∧ X (f1 **U** f2)). L'expansion équivalente suggère que f1 **U** f2 est vraie dans l'une des deux conditions suivantes.

1. f1 est en attente à l'heure actuelle et (f1 **U** f2) est en attente au pas de temps suivant, ou
2. f2 est valable au pas de temps actuel

Les deux cas peuvent être codés en créant deux états (nœuds) de l'automate et l'automate peut sauter de manière non déterministe à l'un d'eux. Dans le premier cas, nous avons déchargé une partie de la charge de la preuve dans le prochain pas de temps; nous avons donc également créé un autre état (nœud) qui comportera l'obligation du prochain pas de temps dans son libellé.

Nous devons également considérer l'opérateur temporel **R** qui peut provoquer une telle séparation de cas. (f1 **R** f2) est équivalent à (f1 ∧ f2) ∨ (f2 ∧ X (f1 **R** f2)) et cette extension équivalente suggère que f1 R f2 est vraie dans l'une des deux conditions suivantes.

1. f2 en attente à l'heure actuelle et (f1 **R** f2) en attente au pas de temps suivant, ou
2. (f1 ∧f2) est maintenu au pas de temps actuel.

Pour éviter de nombreux cas dans l'algorithme suivant, définissons les fonctions cur1, next1 et cur2 qui codent les équivalences ci-dessus dans le tableau suivant.

f	cur1(f)	next(f)	cur2(f)
f1 **U** f2	{f1}	{f1 **U** f2}	{f2}
f1 **R** f2	{f1}	{f1 R f2}	{f1,f2}
f1 ∨f2	{f2}	∅	{f1}

Nous avons également ajouté un cas de disjonction dans le tableau ci-dessus, car il provoque également une scission de cas dans l'automate.