Algorithmes pour la vérification de formules LTL par l'approche automate

Alexandre DURET-LUTZ
Alexandre.Duret-Lutz@lip6.fr

16 mars 2004

Plan

I. Logique temporelle et automates de Büchi
– Logique propositionnelle
– Logique monadique du premier ordre
– LTL
– Automates de Büchi généralisés
– Logique monadique du second ordre

II. Traduction de formules LTL en automates de Büchi généralisés
– Méthode des tableaux pour la logique propositionnelle
– Méthodes des tableaux pour LTL

III. Diagrammes de décision binaires

IV. Traduction symbolique de formules LTL en automates de Büchi généralisés

V. Test de vacuité d'un automate de Büchi généralisé

Partie I
Logique temporelle et automates Büchi

Logique propositionnelle : l'instant présent

La logique propositionnelle peut être utilisée pour caractériser un instant.

r : feu rouge allumé

o : feu orange allumé

v : feu vert allumé

$r \wedge o \wedge v =$  , $r \wedge \neg o \wedge \neg v =$  , $\neg r \wedge \neg o \wedge v =$  , $\neg r \wedge \neg o \wedge \neg v =$  .

Comment dire que  précède  ? Comment dire que le système ne reste pas toujours sur  ?

$\Rightarrow$ besoin de faire apparaître le temps

Logique monadique du premier ordre à un successeur

Les variables propositionnelles deviennent des prédicats unaires, paramétrés par le temps.

$r(t)$, $o(t)$, $v(t)$: feux allumés à l'instant t
$t+1$: instant successeur immédiat
$t \leq u$: ordre total sur les instants
$\exists t$, $\forall t$: quantificateurs du premier ordre

$\neg\forall t.(r(t) \wedge \neg o(t) \wedge \neg v(t))$: le système ne reste pas tout le temps dans la configuration  .

$\forall t.((\neg r(t) \wedge o(t) \wedge \neg v(t)) \Rightarrow (r(t+1) \wedge \neg o(t+1) \wedge \neg v(t+1)))$:

toute configuration  est immédiatement suivie de  .

$\forall t.\exists u.(t \leq u) \wedge (\neg r(u) \wedge \neg o(u) \wedge v(u))$:

le système passe infiniment souvent par la configuration  .

Logique Temporelle Linéaire

Next	$\mathsf{X}\, f$	f est vraie à l'instant suivant
Always	$\mathsf{G}\, f$	f est vraie a tout instant
Eventually	$\mathsf{F}\, f$	f sera vraie à un instant (présent ou futur)
Until	$f \,\mathsf{U}\, g$	f est toujours vraie jusqu'à ce que g le soit

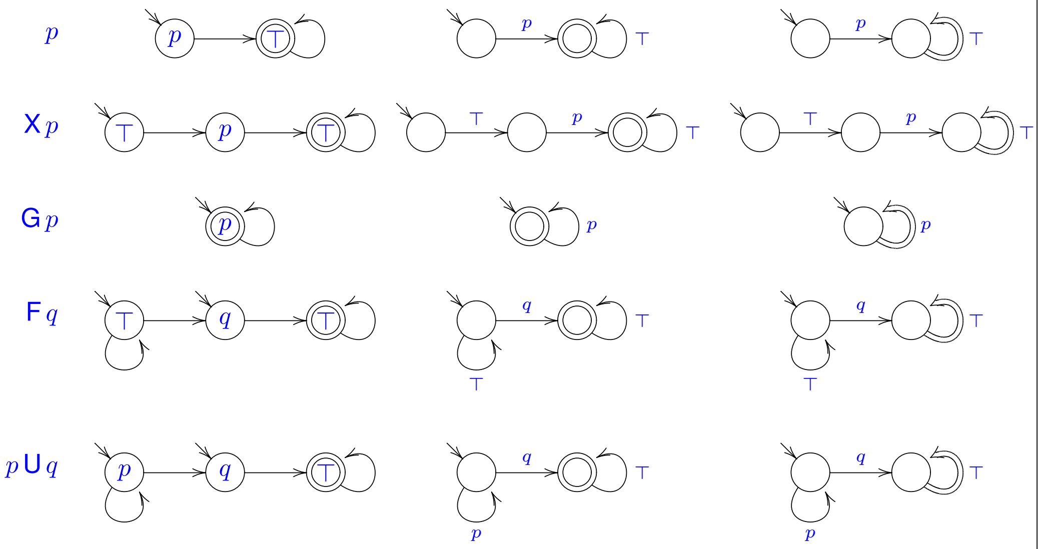
Équivalente à la logique monadique du premier ordre à un successeur.

$\neg\,\mathsf{G}(r \wedge \neg o \wedge \neg v)$: le système ne reste pas tout le temps dans la configuration  .

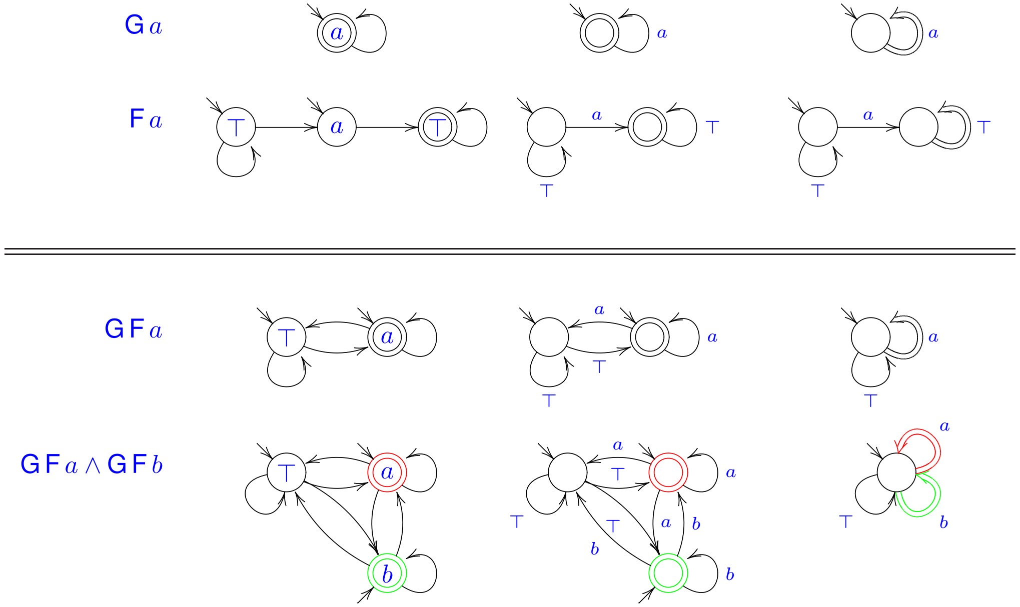
$\mathsf{G}((\neg r \wedge o \wedge \neg v) \Rightarrow \mathsf{X}(r \wedge \neg o \wedge \neg v))$: toute configuration  est immédiatement suivie de  .

$\mathsf{G}\,\mathsf{F}(\neg r \wedge \neg o \wedge v)$: le système passe infiniment souvent par la configuration  .

Vision « automate » des opérateurs LTL



Ces automates ne sont pas composables



Notations (1/2)

On note traditionnellement AP (*atomic propositions*) l'ensemble des variables propositionnelles utilisées dans la vérification d'un système.

Un *minterm* est une conjonction dans laquelle toutes les variables apparaissent sous forme positive ou négative.

Si $AP = \{p, q\}$, chaque état du système vérifie exactement l'un des quatre minterms suivants : $p \wedge q$, $p \wedge \neg q$, $\neg p \wedge q$, $\neg p \wedge \neg q$.

2^{AP} désigne l'ensemble des parties de AP, soit $2^{AP} = \{\{p,q\}, \{p\}, \{q\}, \emptyset\}$.

Il existe une bijection entre 2^{AP} et l'ensemble des minterms sur AP.

$$\begin{array}{rcl} \{p,q\} & \leftrightarrow & p \wedge q \\ \{p\} & \leftrightarrow & p \wedge \neg q \\ \{q\} & \leftrightarrow & \neg p \wedge q \\ \emptyset & \leftrightarrow & \neg p \wedge \neg q \end{array}$$

Notations (2/2)

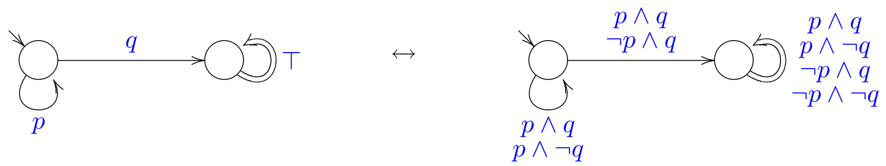
Toute formule propositionnelle sur AP peut être vue comme une disjonction de minterms.
$p \equiv (p \wedge q) \vee (p \wedge \neg q)$.

De même, une formule booléenne peut être représentée par un ensemble d'éléments de 2^{AP}.
$p \leftrightarrow \{\{p\}, \{p,q\}\}$

Nous désignerons par $2^{2^{AP}}$ l'ensemble des formules propositionnelles sur AP.

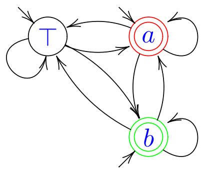
Les automates que l'on considère reconnaissent des ω-mots de 2^{AP}, i.e., les lettres sont des minterms et les mots sont infinis.

On simplifie les automates en représentant non pas des minterms, mais des formules propositionnelles sur les arcs.



Automates de Büchi généralisés (étiquetés sur les états)

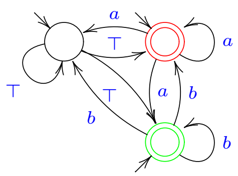
Définition 1. *Un automate de Büchi généralisé (étiqueté sur les états) est un sextuplet* $A = \langle \Sigma, \mathcal{Q}, l, \delta, \mathcal{Q}_0, \mathcal{F} \rangle$ *où*
- Σ *est un ensemble fini de lettres (pour nous c'est* $\Sigma = 2^{AP}$*, l'ensemble des minterms de* AP*),*
- $\mathcal{Q}$ *est un ensemble fini d'états,*
- $l : \mathcal{Q} \mapsto 2^{\Sigma} \setminus \{\emptyset\}$ *est la fonction d'étiquetage des états (par des formules propositionnelles),*
- $\delta \subseteq \mathcal{Q} \times \mathcal{Q}$ *est la relation de transition de l'automate,*
- $\mathcal{Q}_0 \subseteq \mathcal{Q}$ *est l'ensemble des états initiaux,*
- $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ *est un ensemble d'ensembles d'états d'acceptation :* $\forall i, F_i \subseteq \mathcal{Q}$.



Les ω-mots reconnus par l'automate sont acceptés uniquement s'ils traversent infiniment souvent un état de chaque ensemble d'acceptation.

Automates de Büchi généralisés étiquetés sur les transitions

Définition 2. *Un automate de Büchi généralisé étiqueté sur les transitions est un quintuplet* $A = \langle \Sigma, \mathcal{Q}, \delta, \mathcal{Q}_0, \mathcal{F} \rangle$ *où*
- Σ *est un ensemble fini de lettres (pour nous c'est* $\Sigma = 2^{AP}$*, l'ensemble des minterms de* AP*)*
- $\mathcal{Q}$ *est un ensemble fini d'états,*
- $\delta \subseteq \mathcal{Q} \times (2^{\Sigma} \setminus \{\emptyset\}) \times \mathcal{Q}$ *est la relation de transition de l'automate (chaque transition est étiquetée par une formule propositionnelle),*
- $\mathcal{Q}_0 \subseteq \mathcal{Q}$ *est l'ensemble des états initiaux,*
- $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ *est un ensemble d'ensembles d'états d'acceptation :* $\forall i, F_i \subseteq \mathcal{Q}$.

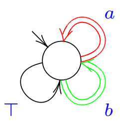


Les ω-mots reconnus par l'automate sont acceptés uniquement s'ils traversent infiniment souvent un état de chaque ensemble d'acceptation.

Automates de Büchi généralisés avec transitions d'acceptation

Définition 3. *Un automate de Büchi généralisé avec transitions d'acceptations est un quintuplet* $A = \langle \Sigma, \mathcal{Q}, \delta, Q_0, \mathcal{F} \rangle$ *où*

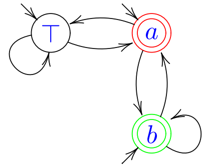
– Σ *est un ensemble fini de lettres (pour nous c'est* $\Sigma = 2^{AP}$*, l'ensemble des minterms de* AP*)*
– $\mathcal{Q}$ *est un ensemble fini d'états,*
– $\delta \subseteq \mathcal{Q} \times (2^{\Sigma} \setminus \{\emptyset\}) \times \mathcal{Q}$ *est la relation de transition de l'automate (chaque transition est étiquetée par une formule propositionnelle),*
– $Q_0 \subseteq \mathcal{Q}$ *est l'ensemble des états initiaux,*
– $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ *est un ensemble d'ensembles de transitions d'acceptation :* $\forall i, F_i \subseteq \delta$.



Les ω-mots reconnus par l'automate sont acceptés uniquement s'ils traversent infiniment souvent une transition de chaque ensemble d'acceptation.

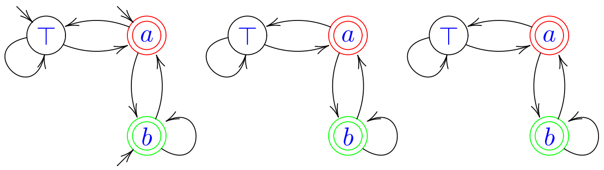
Dégénéralisation d'automates de Büchi

On souhaite transformer un automate de Büchi avec n ensembles d'acceptations en un automate avec un seul ensemble d'acceptation.



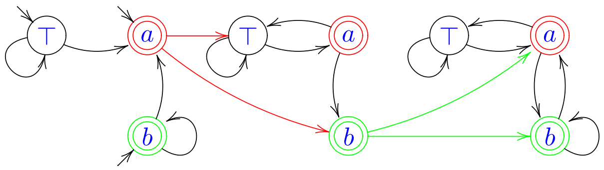
Dégénéralisation d'automates de Büchi

Première étape : on duplique l'automate $n + 1$ fois, en laissant les états initiaux sur la première copie.



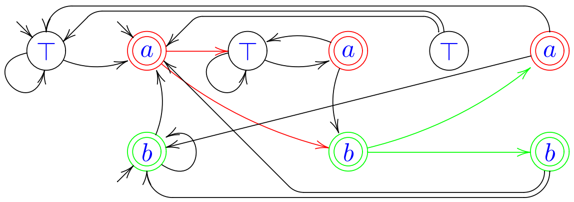
Dégénéralisation d'automates de Büchi

Seconde étape : dans la i^{e} copie de l'automate, les transitions sortantes des états appartenant à l'i^{e} ensemble d'acceptation ($\mathcal{F}_i$) sont redirigées vers la copie suivante.



Dégénéralisation d'automates de Büchi

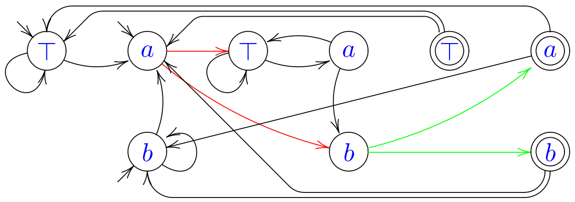
Troisième étape : les transitions sortantes de tous les états de la dernière copie de l'automate sont redirigées vers la 1re copie.



Dégénéralisation d'automates de Büchi

Enfin : l'ensemble d'acceptation est constitué de tous les états de la dernière copie de l'automate.

Notes :
- on peut faire mieux
- cela marche aussi pour les automates avec transitions d'acceptation



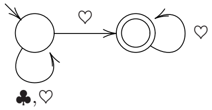
Propriétés des automates de Büchi

Les automates de Büchi, étiquetés sur les états ou les transitions, avec états ou transitions d'acceptation, généralisés ou non, sont tous aussi expressifs. I.e., ils peuvent reconnaître les mêmes langages (pas forcement avec autant d'états ou de transition).

D'autre part les langages reconnaissables par des automates de Büchi
- sont clos par union (évident)
- sont clos par intersection (produit synchronisé)
- sont clos par complémentation (difficile à montrer)
  [Construction connue en c^{n^2} avec $c > 1$ et n le nombre d'états de l'automate à complémenter.]
- ont leur vide décidable

Un automate de Büchi n'est pas toujours déterminisable.

Un automate de Büchi n'est pas toujours déterminisable

A = 

$\mathscr{L}(A) = (\clubsuit + \heartsuit)^\star \heartsuit^\omega$

Supposons qu'il existe un automate déterministe $B = \langle \{\clubsuit, \heartsuit\}, \mathcal{Q}, \delta, \{q_0\}, F \rangle$ avec un seul ensemble d'acceptation, tel que $\mathscr{L}(B) = \mathscr{L}(A)$.

$u_0 = \heartsuit^\omega \in \mathscr{L}(A)$, donc il existe un préfixe fini v_0 de u_0 qui amène l'automate B dans F.
$u_1 = v_0\clubsuit\heartsuit^\omega \in \mathscr{L}(A)$, donc il existe un préfixe fini $v_0\clubsuit v_1$ de u_1 qui amène l'automate B dans F.
$\vdots$
$u_n = v_{n-1}\clubsuit\heartsuit^\omega \in \mathscr{L}(A)$, donc il existe un préfixe fini $v_0\clubsuit v_1\clubsuit\cdots\clubsuit v_n$ de u_n qui amène l'automate B dans F.

Puisque $\mathcal{Q}$ est fini, il existe i et j, $0 \leqslant i < j$, tels que les mots $v_0\clubsuit v_1\clubsuit\cdots\clubsuit v_i$ et $v_0\clubsuit v_1\clubsuit\cdots\clubsuit v_i\clubsuit\cdots\clubsuit v_j$ conduisent au même état.

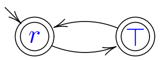
Donc $m = v_0\clubsuit v_1\clubsuit\cdots\clubsuit v_i(\clubsuit\cdots\clubsuit v_j)^\omega$ est accepté par B.
Or m contient une infinité de $\clubsuit$, il ne peut pas appartenir à $\mathscr{L}(A)$!

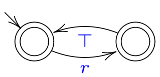
S1S : Logique monadique du second ordre à un successeur

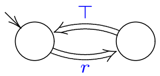
$r(t)$, $o(t)$, $v(t)$: feux allumés à l'instant t
0 : instant initial
$t+1$: instant successeur immédiat
$t \leq u$: ordre total sur les instants
$\exists t$, $\forall t$: quantificateurs du premier ordre
$\exists^2 X$, $\forall^2 X$: quantificateurs du second ordre
$t \in X$: appartenance d'une variable du premier ordre a une variable du second

$$\exists^2 X.\underbrace{(0 \in X \wedge (\forall t.(t \in X \Rightarrow (\neg(t+1 \in X) \wedge (t+1+1 \in X)))))}_{Pair(X)}$$

$\exists^2 X.Pair(X) \wedge \forall t.(t \in X \Rightarrow r(t))$: le feu rouge doit toujours être allumé aux instants pairs.







Lien entre LTL et automates de Büchi

LTL		Automates de Büchi
$=$		$=$
Logique monadique du premier ordre	$\subsetneq$	Logique monadique du second ordre

Model checking : approche automate

modèle M de haut niveau <····vérification····> propriété temporelle φ

traduction ↓ ↓ traduction

ω-automate A_M
$\mathscr{L}(A_M)$

ω-automate $A_{\neg\varphi}$
$\mathscr{L}(A_{\neg\varphi})$

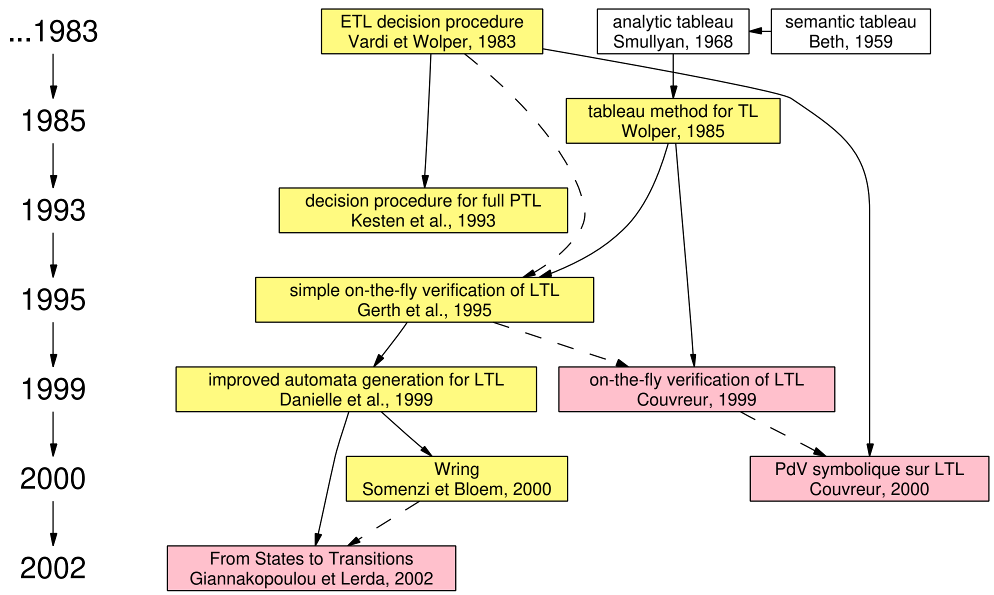
produit synchronisé $A_M \otimes A_{\neg\varphi}$
$\mathscr{L}(A_M \otimes A_{\neg\varphi}) = \mathscr{L}(A_M) \cap \mathscr{L}(A_{\neg\varphi})$

↓

φ vérifiée $M \models \varphi$ ← oui — emptiness check $\mathscr{L}(A_M \otimes A_{\neg\varphi}) \stackrel{?}{=} \emptyset$ — non → contre-exemple $\omega \in \mathscr{L}(A_M \otimes A_{\neg\varphi})$

Partie II
Traduction de formules LTL en automates de Büchi généralisés

Généalogie des méthodes par tableau pour la traduction LTL



Méthode des tableaux pour la logique propositionnelle

Une formule propositionnelle φ est-elle vraie ?

Idée : mettre $\neg\varphi$ sous forme normale disjonctive. φ est vraie si $\neg\varphi$ n'est pas satisfiable.

formule	1[er] fils	2[e] fils
$\neg\neg f$	$\{f\}$	
$\neg\top$	$\{\bot\}$	
$\neg\bot$	$\{\top\}$	
$f \wedge g$	$\{f, g\}$	
$f \vee g$	$\{f\}$	$\{g\}$
$\neg(f \wedge g)$	$\{\neg f\}$	$\{\neg g\}$
$\neg(f \vee g)$	$\{\neg f, \neg g\}$	

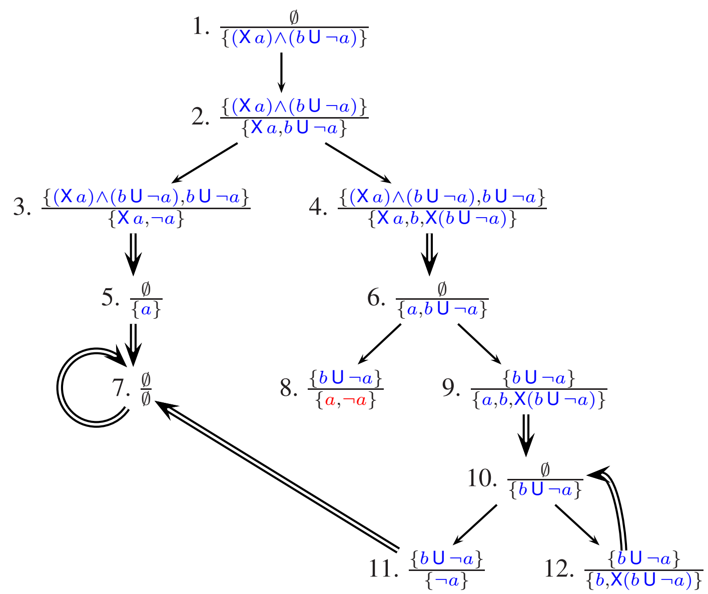
Tableau de $\neg\varphi$ pour prouver $\varphi = \neg(\neg A \vee B) \vee (\neg(A \wedge C) \vee (B \wedge C))$

$$\{\neg(\neg(\neg A \vee B) \vee (\neg(A \wedge C) \vee (B \wedge C)))\}$$
$$\downarrow$$
$$\{\neg\neg(\neg A \vee B), \neg(\neg(A \wedge C) \vee (B \wedge C))\}$$
$$\downarrow$$
$$\{\neg A \vee B, \neg(\neg(A \wedge C) \vee (B \wedge C))\}$$
$$\downarrow$$
$$\{\neg A \vee B, \neg\neg(A \wedge C), \neg(B \wedge C)\}$$
$$\downarrow$$
$$\{\neg A \vee B, A \wedge C, \neg(B \wedge C)\}$$
$$\downarrow$$
$$\{\neg A \vee B, A, C, \neg(B \wedge C)\}$$
$$\swarrow \qquad \searrow$$
$$\{\neg A, A, C, \neg(B \wedge C)\} \qquad \{B, A, C, \neg(B \wedge C)\}$$
$$\swarrow \qquad \searrow$$
$$\{B, A, C, \neg B\} \qquad \{B, A, C, \neg C\}$$

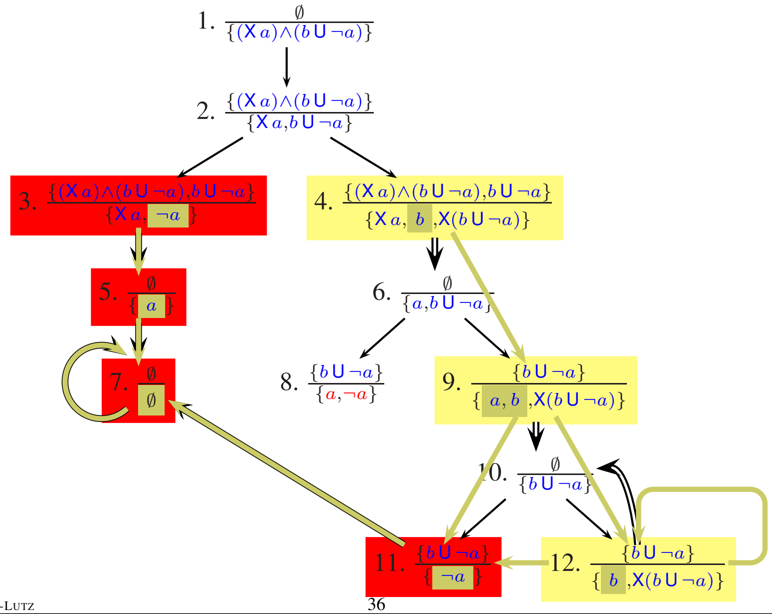
Méthode des tableaux pour LTL

formule	1[er] fils	2[e] fils
$\neg\neg f$	$\{f\}$	
$\neg\top$	$\{\bot\}$	
$\neg\bot$	$\{\top\}$	
$f \wedge g$	$\{f, g\}$	
$f \vee g$	$\{f\}$	$\{g\}$
$\neg(f \wedge g)$	$\{\neg f\}$	$\{\neg g\}$
$\neg(f \vee g)$	$\{\neg f, \neg g\}$	
$\neg \mathsf{X} f$	$\{\mathsf{X} \neg f\}$	
$f \mathsf{U} g$	$\{g\}$	$\{f, \mathsf{X}(f \mathsf{U} g)\}$
$\neg(f \mathsf{U} g)$	$\{\neg f, \neg g\}$	$\{\neg g, \mathsf{X} \neg(f \mathsf{U} g)\}$
...	...	

Tableau pour $(\mathbf{X}\, a) \wedge (b\, \mathbf{U}\, \neg a)$



ω-Automate sur états pour $(\mathbf{X}\, a) \wedge (b\, \mathbf{U}\, \neg a)$



ω-Automate sur transitions pour $(\mathbf{X}\, a) \wedge (b\, \mathbf{U}\, \neg a)$

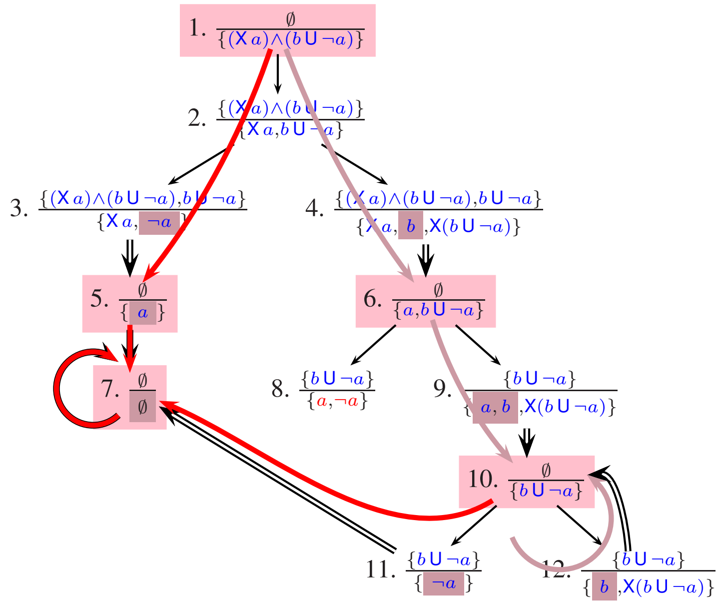


Schéma de traduction en trois phases

Phase I : Pré-traitements de la formule.

- Optimisation de la formule.
  Par ex. $a\, \mathbf{U}\, \mathbf{F}\, b \rightsquigarrow \mathbf{F}\, b$.
  Réécriture à faible coût, qui simplifie le travail de traduction par la suite.
- Mise sous forme normale négative (aussi appelée forme normale positive).
  On pousse les négations devant les propositions atomiques :
  par ex. $\neg\, \mathbf{G}\, \mathbf{F}\, b \rightsquigarrow \mathbf{F}\, \mathbf{G}\, \neg b$.
  Nécessaire pour certains algorithmes de traduction.
- …

Phase II : La traduction elle-même.
À partir d'une formule LTL, on produit un automate.

Phase III : Post-traitements de l'automate.

- Simplifications de l'automate (par ex. tests de simulation).
- Conversion de l'automate (par ex. dégénéralisation) .
- …

Partie III
Diagrammes de décision binaires

Formes normales disjonctive et conjonctive

Toute formule booléenne φ peut se mettre sous

– forme normale disjonctive (FND) :

$$\varphi = \bigvee_i \bigwedge_j l_{ij}$$

– forme normale conjonctive (FNC) :

$$\varphi = \bigwedge_i \bigvee_j l_{ij}$$

Ces formes normales ne sont pas uniques :

$$\begin{aligned}(c \Leftrightarrow b) \Rightarrow (a \Leftrightarrow b) &= (a \wedge b) \vee (\neg a \wedge c) \vee (\neg b \wedge c) && \text{(FND)}\\ &= (\neg a \wedge \neg b) \vee (a \wedge c) \vee (b \wedge \neg c) && \text{(FND)}\\ &= (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c) && \text{(FNC)}\end{aligned}$$

Forme normale if-then-else

Soit l'opérateur ternaire suivant :

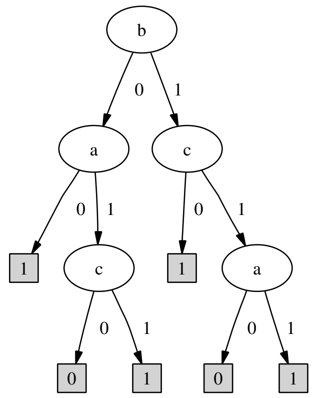
$$\mathsf{ITE}(condition, alors, sinon) = (condition \wedge alors) \vee (\neg condition \wedge sinon)$$

Tous les opérateurs classiques peuvent s'exprimer à l'aide de cet opérateur (et des variables ainsi que $\top$ et $\bot$). On peut de plus imposer que les conditions soient toutes des variables (positives), et que les variables n'apparaissent pas en dehors des conditions.

$$\begin{aligned}p &= \mathsf{ITE}(p, \top, \bot)\\ \neg p &= \mathsf{ITE}(p, \bot, \top)\\ p \Leftrightarrow q &= \mathsf{ITE}(p, \mathsf{ITE}(q, \top, \bot), \mathsf{ITE}(q, \bot, \top))\\ p \vee q &= \mathsf{ITE}(p, \top, \mathsf{ITE}(q, \top, \bot))\\ (c \Leftrightarrow b) \Rightarrow (a \Leftrightarrow b) &= \mathsf{ITE}(a, \mathsf{ITE}(b, \top, \mathsf{ITE}(c, \top, \bot)), \mathsf{ITE}(c, \mathsf{ITE}(b, \bot, \top), \top))\\ &= \mathsf{ITE}(b, \mathsf{ITE}(c, \mathsf{ITE}(a, \bot, \top), \top), \mathsf{ITE}(a, \mathsf{ITE}(c, \top, \bot), \top))\end{aligned}$$

Arbre de décision

$$(c \Leftrightarrow b) \Rightarrow (a \Leftrightarrow b) = \mathsf{ITE}(b, \mathsf{ITE}(c, \mathsf{ITE}(a, \bot, \top), \top), \mathsf{ITE}(a, \mathsf{ITE}(c, \top, \bot), \top))$$



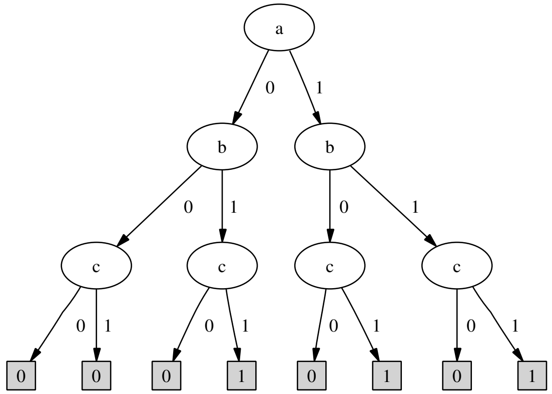
BDD : binary decision diagrams

Les BDD sont des graphes acycliques correspondant à des formes normales if-then-else avec les contraintes suivantes :

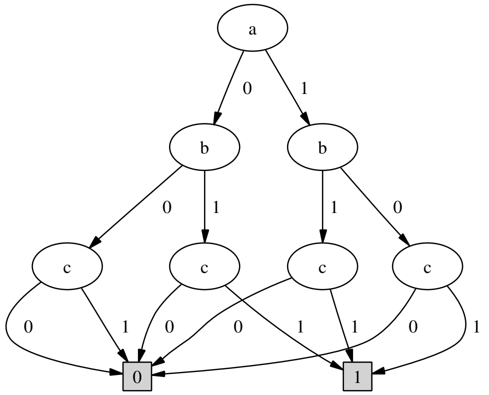
– les variables sont testées dans un ordre fixé
– les sous-graphes identiques sont identifiées
– aucun test inutile ($\mathsf{ITE}(t, f, f)$) n'est effectué

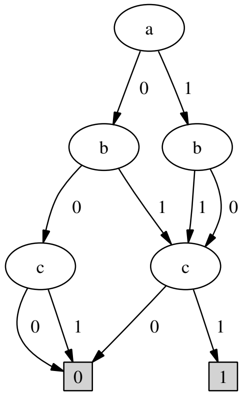
Exemple (1/3) : représentation de $f = (a \vee b) \wedge c$

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



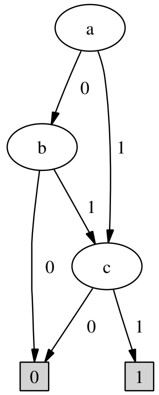
Exemple (2/3) : identification des sous-graphes identiques





Exemple (3/3) : suppression des tests inutiles

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



Canonicité

Pour un ordre de variables donné, toute formule propositionnelle possède une représentation BDD unique.

Tester l'équivalence de deux formules propositionnelle revient à tester l'égalité de leurs représentation BDD.

En particulier, il existe une unique représentation de la formule $\top$: le BDD constitué du seul nœud $\boxed{1}$.

Une formule est une tautologie si son BDD est $\boxed{1}$.

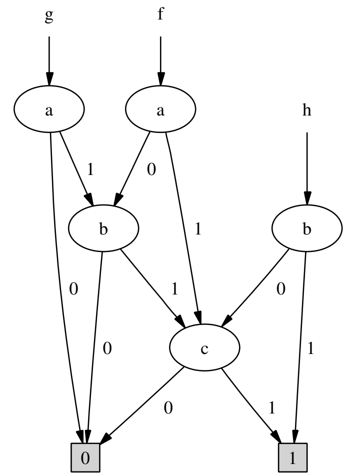
Partage inter-formules

Un même graphe peut servir de support à plusieurs formules.

$f = (a \vee b) \wedge c$

$g = a \wedge b \wedge c$

$h = b \vee c$



Tester l'équivalence de deux BDD se fait en temps constant.

Manipulation des BDD

On sait définir les constantes et opérateurs suivants directement sur les BDD.

$\top, \bot$

$v, \neg v$

$f \wedge g, f \vee g, f \Leftrightarrow g, f \Rightarrow g, \ldots$

$f[0/v], f[1/v]$ (substitution d'une variable par une valeur)

$f[g/v]$ (composition)

$\exists v.f = f[0/v] \vee f[1/v]$ (suppression d'une variable par quantification existencielle)

$\forall v.f = f[0/v] \wedge f[1/v]$ (suppression d'une variable par quantification universelle)

$onesat(f)$ (retourne une conjonction qui satisfait f)

Et bien plus...

Ces opérations se définissent récursivement sur la structure de graphe des BDD.

Utilisation **d'un cache d'opérations**.

Complexités (pire cas)

formules booléennes
satisfiabilité : $O(2^{|f|})$
tautologie : $O(2^{|f|})$

$\xrightarrow{O(2^{|f|})}$ **FND**
satisfiabilité : $O(|f|)$
tautologie : $O(2^{|f|})$

$\xrightarrow{O(2^{|f|})}$ **FNC**
satisfiabilité : $O(2^{|f|})$
tautologie : $O(|f|)$

$\xrightarrow{O(2^{|f|})}$ **BDD**
satisfiabilité : $O(1)$
tautologie : $O(1)$

Autres opérations sur les BDD :

opération	complexité
$f_1 \langle binop \rangle f_2$	$O(\lvert\mathcal{G}_{f_1}\rvert \times \lvert\mathcal{G}_{f_2}\rvert)$
$f[0/v], f[1/v]$	$O(\lvert\mathcal{G}_f\rvert)$
$f_1[f_2/v]$	$O(\lvert\mathcal{G}_{f_1}\rvert^2 \times \lvert\mathcal{G}_{f_2}\rvert)$
$\neg f$	$O(1)$
$onesat(f)$	$O(\lvert\mathcal{G}_f\rvert)$

$|\mathcal{G}_f|$, la taille du graphe représentant la formule f, est sensible à l'ordre de variables choisi.

Intérêt ?

De très nombreux problèmes peuvent se ramener à des manipulations de fonctions booléennes.
- minimisation de formules
- optimisation de circuits
- représentation d'ensembles
- programmation par contrainte
- relation de transitions
- ...

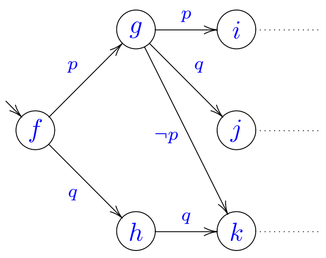
Les BDD fournissent :
- une forme canonique, qui permet d'effectuer des tests d'équivalence en temps constant
- une représentation mémoire favorisant le partage entre les formules manipulées (reste le problème du choix de l'ordre)
- un cache d'opérations, qui accélère les calculs répétitifs

De nombreuses variantes existent.

Partie IV
Traduction symbolique de formules LTL en automates de Büchi généralisés (Couvreur 99)

Avant-propos

- On va générer un automate de Büchi avec transitions d'acceptation (puisqu'ils sont plus petits).
- Les règles de tableau pour la logique propositionnelle seront replacées par une utilisation des BDD (avec les BDD, décider si une formule est satisfiable est immédiat).
- Un calcul d'impliquants premiers (lors de la mise sous forme normale disjonctive) permet de limiter le nombre de successeurs d'un état.
- On conserve les règles de tableau pour LTL, mais modifées légèrement pour représenter les promesses explicitement.



$$f = (p \wedge \mathsf{X}\, g) \vee (q \wedge \mathsf{X}\, h)$$
$$g = (p \wedge \mathsf{X}\, i) \vee (q \wedge \mathsf{X}\, j) \vee (\neg p \wedge \mathsf{X}\, k)$$
$$h = q \wedge \mathsf{X}\, k$$

Système de réécritures

La formule à traduire en BDD est supposée sous forme normale négative.

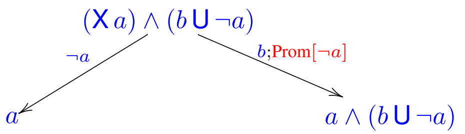
$$\begin{aligned}
rewrite(\top) &= \top \\
rewrite(\bot) &= \bot \\
rewrite(p) &= \mathrm{Var}[p] \\
rewrite(\neg p) &= \neg\mathrm{Var}[p] \\
rewrite(f \vee g) &= rewrite(f) \vee rewrite(g) \\
rewrite(f \wedge g) &= rewrite(f) \wedge rewrite(g) \\
rewrite(\mathsf{X}\, f) &= \mathrm{Next}[f] \\
rewrite(f \,\mathsf{U}\, g) &= rewrite(g) \vee (\mathrm{Prom}[g] \wedge rewrite(f) \wedge \mathrm{Next}[f \,\mathsf{U}\, g]) \\
rewrite(\mathsf{F}\, g) &= rewrite(g) \vee (\mathrm{Prom}[g] \wedge \mathrm{Next}[\mathsf{F}\, g]) \\
rewrite(f \,\mathsf{R}\, g) &= rewrite(g) \wedge (rewrite(f) \vee \mathrm{Next}[f \,\mathsf{R}\, g]) \\
rewrite(\mathsf{G}\, g) &= rewrite(g) \wedge \mathrm{Next}[\mathsf{G}\, g]
\end{aligned}$$

Exemple de réécriture

$$
\begin{aligned}
rewrite((\mathsf{X}\, a) \wedge (b \,\mathsf{U} \neg a)) &= rewrite(\mathsf{X}\, a) \wedge rewrite(b \,\mathsf{U} \neg a) \\
&= \mathrm{Next}[a] \wedge (rewrite(\neg a) \vee (\mathrm{Prom}[\neg a] \wedge rewrite(b) \wedge \mathrm{Next}[b \,\mathsf{U} \neg a])) \\
&= \mathrm{Next}[a] \wedge (\neg \mathrm{Var}[a] \vee (\mathrm{Prom}[\neg a] \wedge \mathrm{Var}[b] \wedge \mathrm{Next}[b \,\mathsf{U} \neg a]))
\end{aligned}
$$

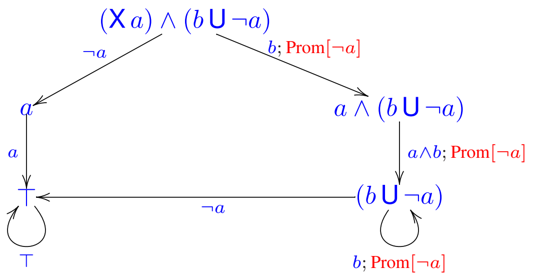
Mise sous forme normale disjonctive (en pratique, calcul d'impliquants premiers sur le BDD représentant la formule).

$$rewrite((\mathsf{X}\, a) \wedge (b \,\mathsf{U} \neg a)) = (\neg \mathrm{Var}[a] \wedge \mathrm{Next}[a]) \vee (\mathrm{Prom}[\neg a] \wedge \mathrm{Var}[b] \wedge \mathrm{Next}[a] \wedge \mathrm{Next}[b \,\mathsf{U} \neg a])$$

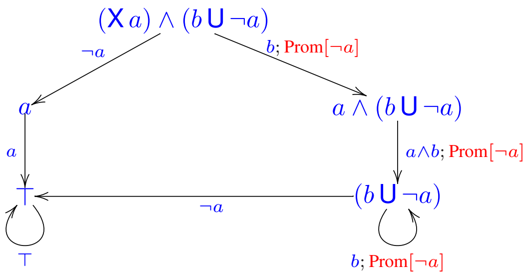


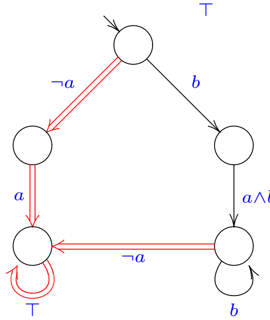
Automate récrit complètement

$$
\begin{aligned}
rewrite((\mathsf{X}\, a) \wedge (b \,\mathsf{U} \neg a)) &= (\neg \mathrm{Var}[a] \wedge \mathrm{Next}[a]) \vee (\mathrm{Prom}[\neg a] \wedge \mathrm{Var}[b] \wedge \mathrm{Next}[a] \wedge \mathrm{Next}[b \,\mathsf{U} \neg a]) \\
rewrite(a) &= \mathrm{Var}[a] \\
rewrite(\top) &= \top \\
rewrite(a \wedge (b \,\mathsf{U} \neg a)) &= \mathrm{Prom}[\neg a] \wedge \mathrm{Var}[a] \wedge \mathrm{Var}[b] \wedge \mathrm{Next}[b \,\mathsf{U} \neg a] \\
rewrite(b \,\mathsf{U} \neg a) &= \neg \mathrm{Var}[a] \vee (\mathrm{Prom}[\neg a] \wedge \mathrm{Var}[b] \wedge \mathrm{Next}[b \,\mathsf{U} \neg a])
\end{aligned}
$$



Automate de Büchi généralisé correspondant





Pour chaque promesse $Prom[f]$, un ensemble d'acceptation F_f est créé. Il contient toutes les transitions non étiquetées par $Prom[f]$.

Algorithme de traduction

```
ltl_to_tgba_fm(f)
    todo ← {f}; all_acc ← ∅; a ← new automaton; a.set_initial_state(f)
    while (todo ≠ ∅)
        here ← todo.remove_one()
        forall i ∈ prime_implicants_of(rewrite(here))
            Put i as ⋀_{v∈V} Var[v] ∧ ⋀_{v∈V'} ¬Var[v] ∧ ⋀_{a∈A} Prom[a] ∧ ⋀_{n∈N} Next[n]
            dest ← ⋀_{n∈N} n
            if ¬ a.has_state(dest)
                todo.insert(dest)
            a.add_transition(source: here, destination: dest,
                             cond: ⋀_{v∈V} v ∧ ⋀_{v∈V'} ¬v, promises: A)
            all_acc ← all_acc ∪ A
    forall t in a.transitions()
        t.acceptance_conditions ← all_acc \ t.promises
    return a
```

Remarques

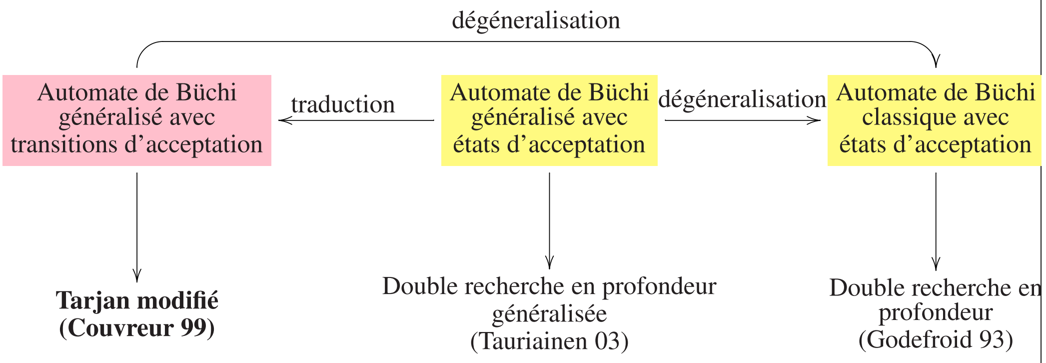
Au lieu d'identifier les états de l'automate avec des formules LTL f ou g, il est préférable de les identifier avec leurs réécritures BDDesque ($rewrite(f)$ et $rewrite(g)$), qui ont une forme canonique.

Cette construction peut être faite à la volée : on construit les états de l'automate au fur et à mesure des besoins.

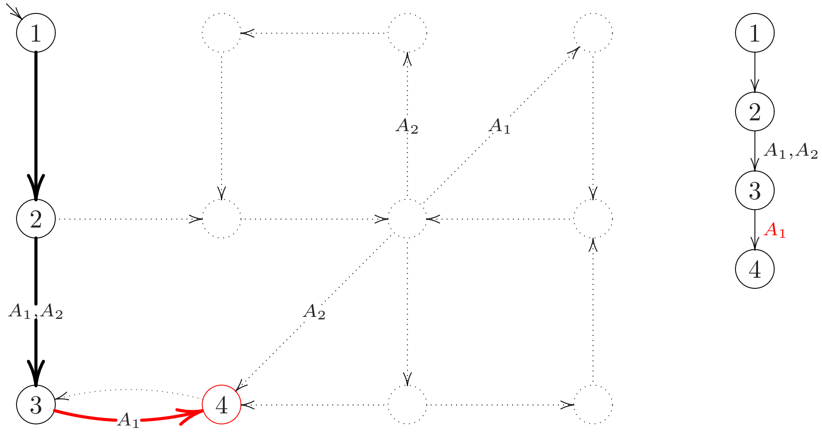
Partie V
Test de vacuité d'un automate de Büchi généralisé

Emptiness-check

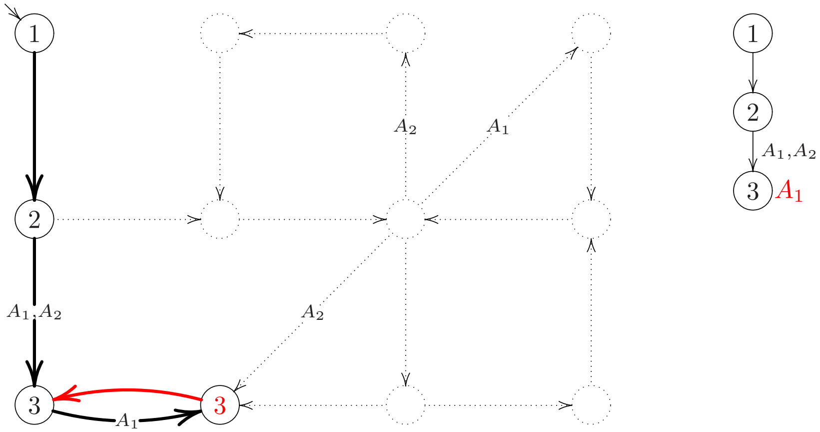
Différents algorithmes pour différents types d'automates.



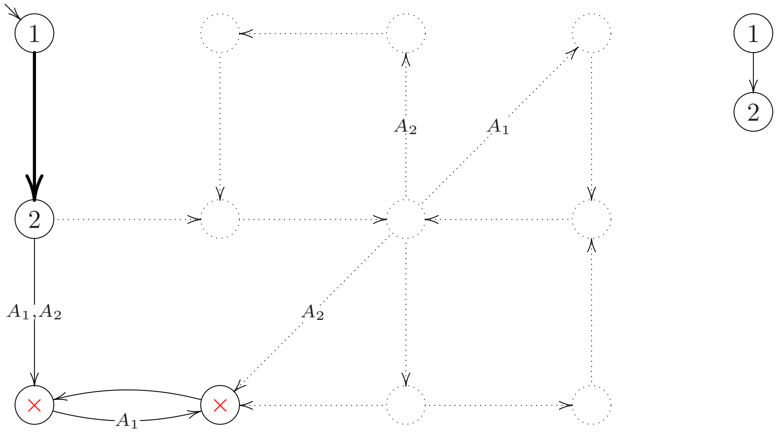
Exemple de recherche de CFC acceptante



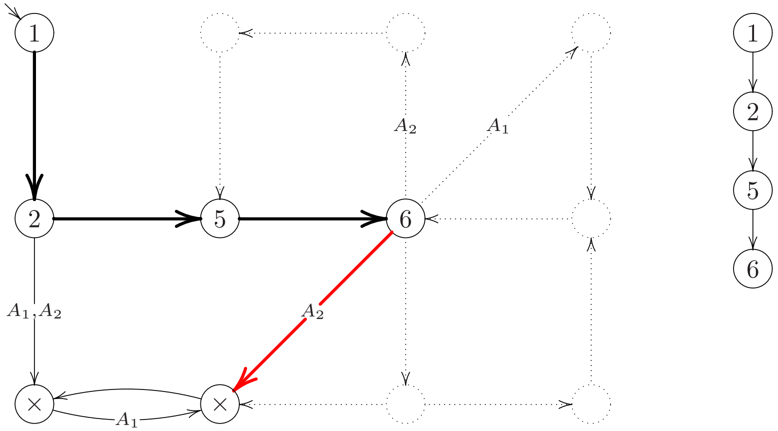
Exemple de recherche de CFC acceptante



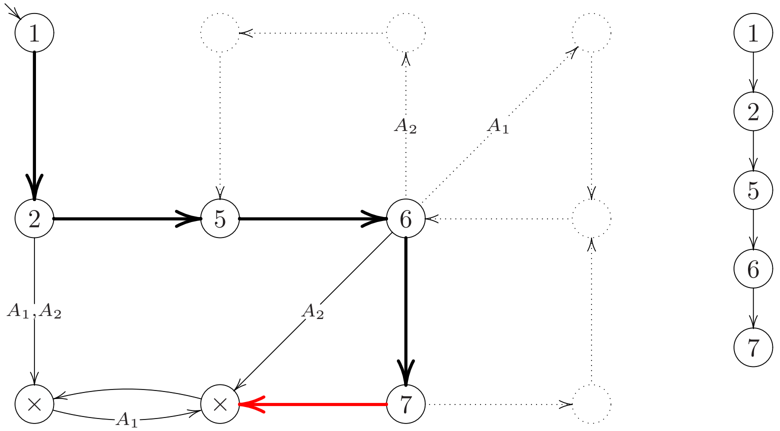
Exemple de recherche de CFC acceptante



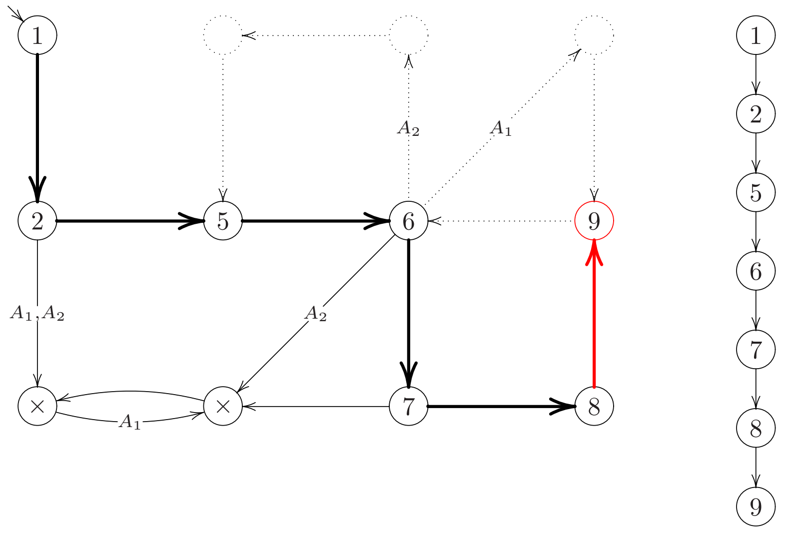
Exemple de recherche de CFC acceptante



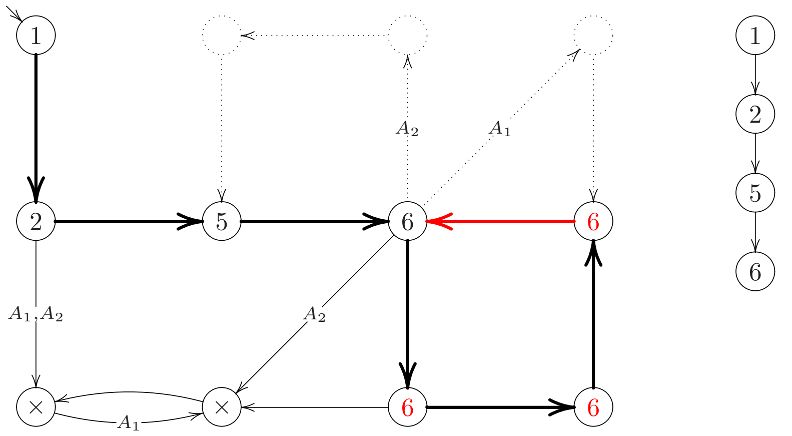
Exemple de recherche de CFC acceptante



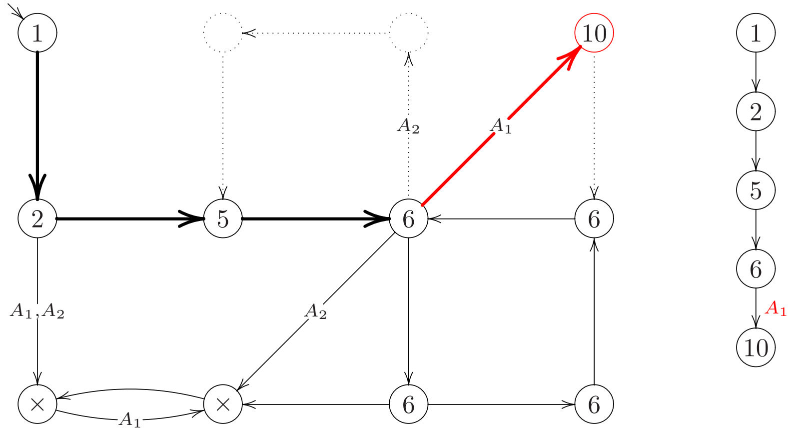
Exemple de recherche de CFC acceptante



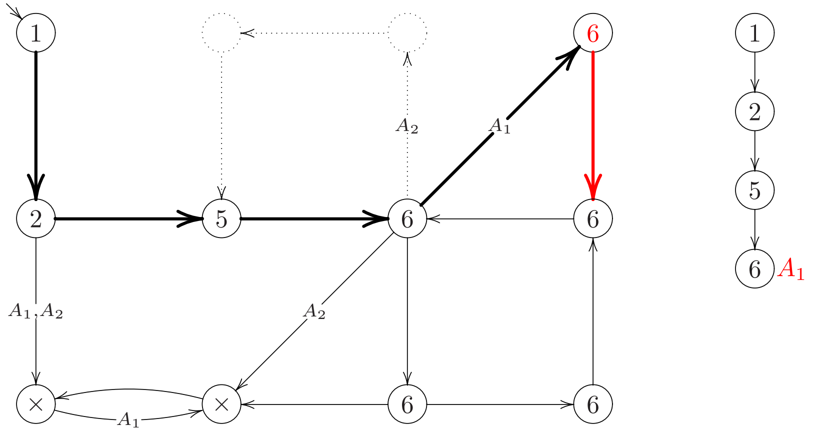
Exemple de recherche de CFC acceptante



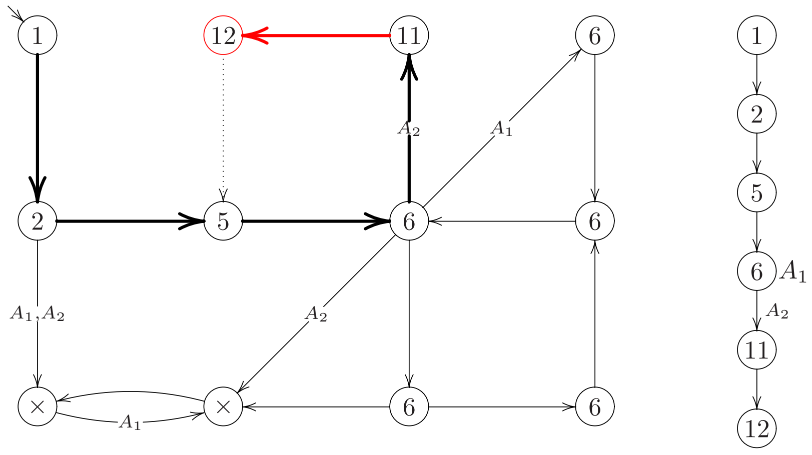
Exemple de recherche de CFC acceptante



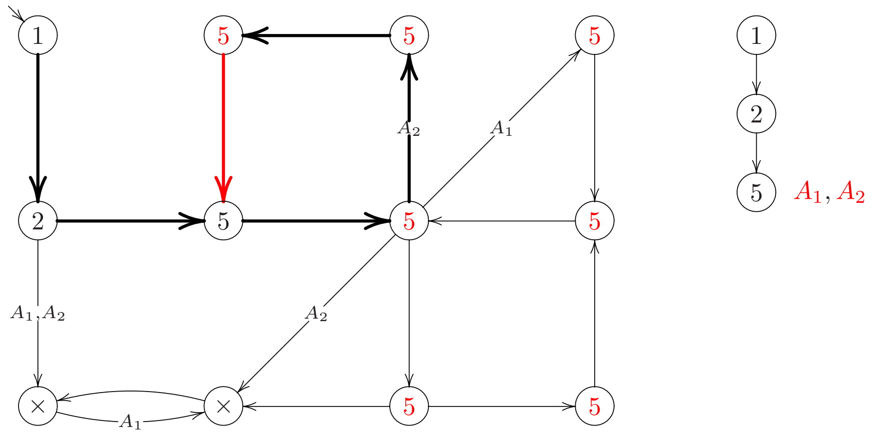
Exemple de recherche de CFC acceptante



Exemple de recherche de CFC acceptante



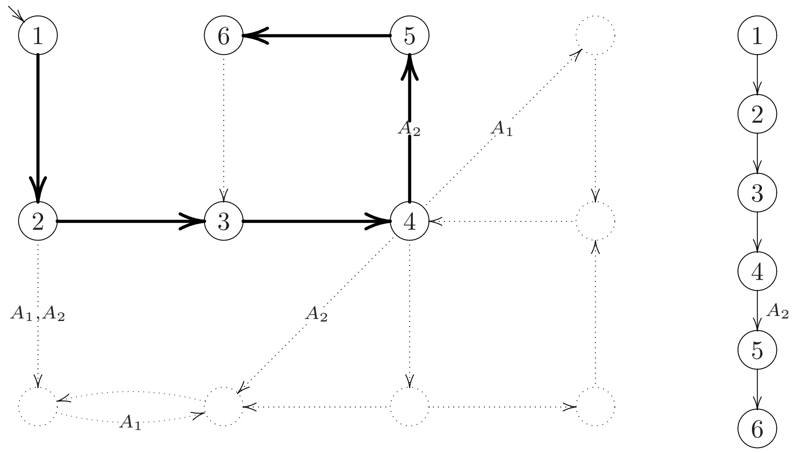
Exemple de recherche de CFC acceptante



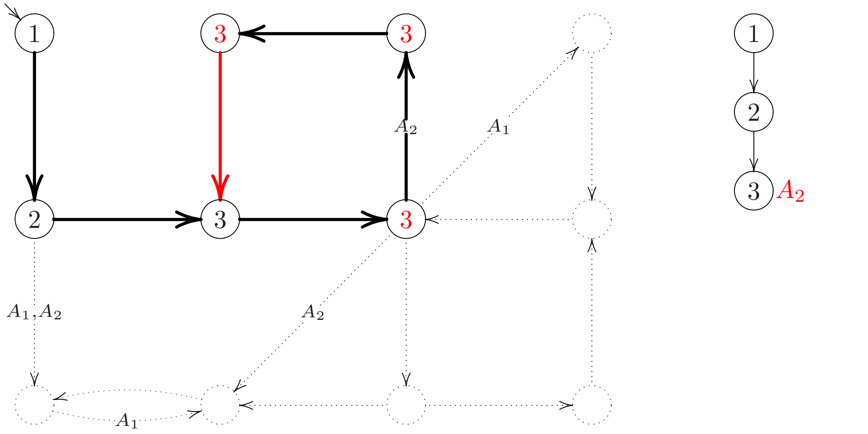
Exemple de recherche de CFC acceptante : Remarques

- Nous avons montré qu'il existe une composante fortement connexe acceptante et accessible.
- Donc l'automate est satisfiable.
- Nous n'avons pas de contre-exemple explicite.
- Dans le pire cas, l'algorithme traverse une seul fois chaque transition.
- Avec un autre ordre de parcours, nous aurions pu répondre plus vite.

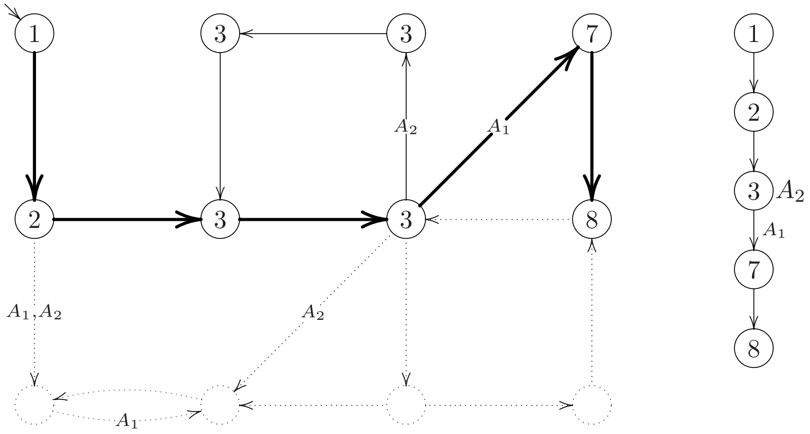
Exemple de recherche de CFC acceptante



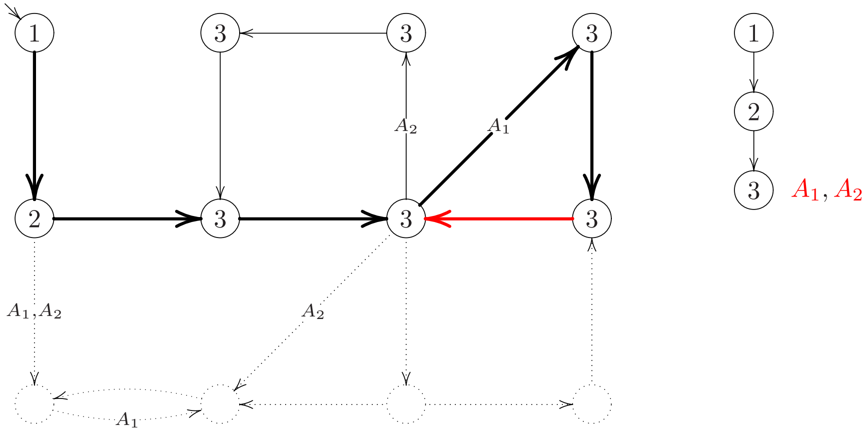
Exemple de recherche de CFC acceptante



Exemple de recherche de CFC acceptante



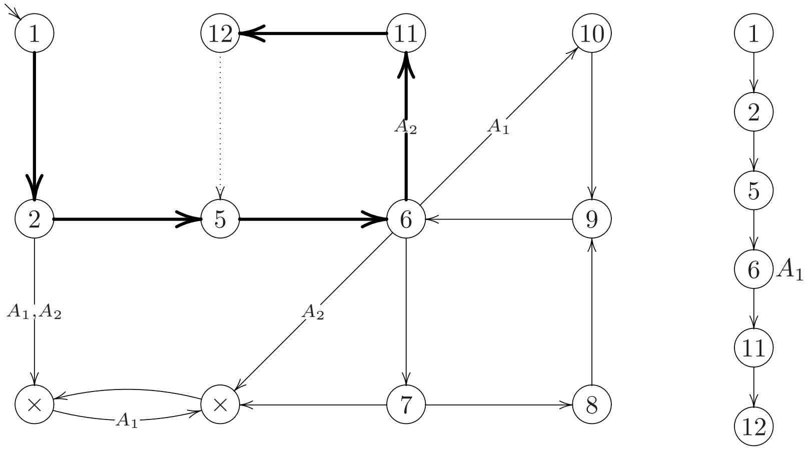
Exemple de recherche de CFC acceptante



Exemple de recherche de CFC acceptante

En pratique il n'est utile de renuméroter que les états des CFC mortes.

Toute étiquette $6 \leqslant x < 11$ désigne un état de la CFC 6.



Algorithme de recherche de CFC acceptante (1/3)

On ne travaille pas sur les propositions des arcs, mais sur leur appartenance aux ensembles d'acceptation.
Plutôt que $A = \langle \Sigma, Q, \delta, Q_0, \mathcal{F} \rangle$ avec $\delta \subseteq \mathcal{Q} \times 2^\Sigma \times \mathcal{Q}$,
on utilisera $A = \langle Q, \delta', Q_0, \mathcal{F} \rangle$ avec $\delta' \subseteq \mathcal{Q} \times 2^\mathcal{F} \times \mathcal{Q}$.

emptiness_check($\langle Q, \delta', q, \mathcal{F} \rangle, H, SCC, ACC$) $\mapsto \{\top, \bot\}$

$\langle Q, \delta', q, \mathcal{F} \rangle$ est l'automate à explorer. Attention, il n'y a qu'un seul état initial, q.

H est une table de hachage des états visités :

- $q \in H$ si q est visité, et dans ce cas
- $H[q]$ est l'étiquette ($\in \mathbb{N}$) associée à l'état q (numéro de visite > 0, ou 0 pour les états morts)
- H.size $\in \mathbb{N}$ est le nombre d'états visités

SCC est la pile de CFC traversées, elle contient des paires $\langle n, a \rangle \in \mathbb{N} \times 2^\mathcal{F}$

$\langle n, a \rangle \leftarrow SCC$.pop() $\quad SCC$.push($\langle n, a \rangle$) $\quad \langle n \rangle \leftarrow SCC$.top_n()

ACC est la pile des conditions d'acceptation sur les arcs entre chacun des éléments de SCC

$a \leftarrow ACC$.pop() $\quad ACC$.push(a)

Algorithme de recherche de CFC acceptante (2/3)

```
emptiness_check(⟨Q, δ', q, F⟩, H, SCC, ACC)
  if q ∉ H // Nouvel etat
      H[q] ← H.size + 1
      SCC.push(⟨H[q], ∅⟩)
      forall ⟨a, s⟩ : ⟨q, a, s⟩ ∈ δ'
          ACC.push(a)
          if emptiness_check(⟨Q, δ', s, F⟩, H, SCC, ACC) = ⊥
              return ⊥
      if SCC.top_n() = H[q]
          SCC.pop()
          mark_reachable_states_as_dead(⟨Q, δ', q, F⟩, H)
      return ⊤
  else
      ...
```


Algorithme de recherche de CFC acceptante (3/3)

```
emptiness_check(⟨Q, δ', q, F⟩, H, SCC, ACC)
  if q ∉ H // Nouvel etat
      ...
  else
      if H[q] = 0 // etat mort
          ACC.pop()
          return ⊤
      else // etat d'une CFC de la pile
          all ← ∅
          do
              ⟨n, a⟩ ← SCC.pop()
              all ← all ∪ a ∪ {ACC.pop()}
          until n ⩽ H[q]
          SCC.push(⟨n, all⟩)
          return all ≠ F
```


Trois références

[1] Henrik Reif Andersen. An introduction to binary decision diagrams. Lecture notes, October 1997. URL http://www.itu.dk/people/hra/bdd97-abstract.html.

[2] Jean-Michel Couvreur. On-the-fly verification of temporal logic. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, Toulouse, France, September 1999. Springer-Verlag. ISBN 3-540-66587-0.

[3] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Proceedings of the 8th Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996. ISBN 3-540-60915-6.