Création et utilisation d'un algorithme de model-checking :

à l'aide de la logique temporelle LTL

Présentation de **Valentin TOMAS–TORTONI**
(numéro de candidat : 12291)

travail réalisé avec Tom LAHRIZIA

1. Définitions
Model-Checking
LGBA
Logique LTL
Objectifs

2. Implémentation LTL

3. Faire l'intersection entre deux automates de Büchi

4. Annexe

Objectifs du Model-Checking

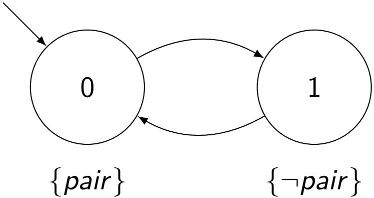
3 étapes clés :

$\rightarrow$ **Modéliser** : trouver une **bonne** modélisation pour le système pour lequel on souhaite vérifier une propriété

$\rightarrow$ **Définir** : la **propriété** que doit vérifier le modèle

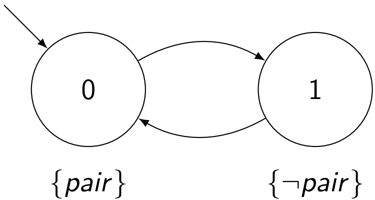
$\rightarrow$ **Tester** : si le modèle vérifie la propriété

Structure de Kripke



Structure de Kripke : $\mathcal{M} = (\mathcal{Q}, I, \rightarrow, \mathcal{L})$

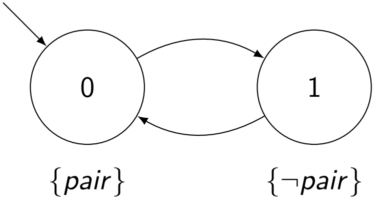
Structure de Kripke



Structure de Kripke : $\mathcal{M} = (\mathcal{Q}, I, \rightarrow, \mathcal{L})$

$$\mathcal{Q} = \{0, 1\}$$

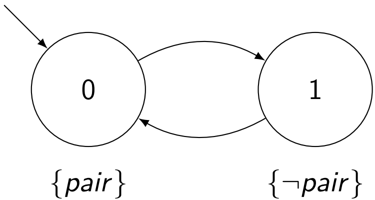
Structure de Kripke



Structure de Kripke : $\mathcal{M} = (\mathcal{Q}, I, \rightarrow, \mathcal{L})$

$$\begin{aligned} \mathcal{Q} &= \{0, 1\} \\ I \ (\subseteq \mathcal{Q}) &= \{0\} \end{aligned}$$

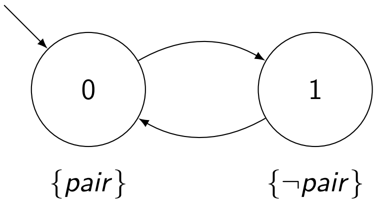
Structure de Kripke



Structure de Kripke : $\mathcal{M} = (\mathcal{Q}, I, \rightarrow, \mathcal{L})$

$$\begin{array}{ccc} \mathcal{Q} & = & \{0,1\} \\ I\ (\subseteq \mathcal{Q}) & = & \{0\} \\ \rightarrow\ (\subseteq \mathcal{Q} \times \mathcal{Q}) & = & \{(0,1),(1,0)\} \end{array}$$

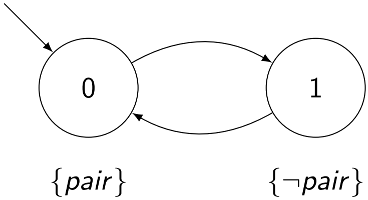
Structure de Kripke



Structure de Kripke : $\mathcal{M} = (\mathcal{Q}, I, \rightarrow, \mathcal{L})$

$$\begin{array}{ccc}
\mathcal{Q} & = & \{0,1\} \\
I\ (\subseteq \mathcal{Q}) & = & \{0\} \\
\rightarrow (\subseteq \mathcal{Q} \times \mathcal{Q}) & = & \{(0,1),(1,0)\} \\
\mathcal{L}\ (\subseteq \mathcal{Q} \times \mathcal{P}(AP \cup \neg AP)) & = & \{(0,\{pair\}),(1,\{\neg pair\})\}
\end{array}$$

Structure de Kripke



Structure de Kripke modélisant un compteur

Chemin sur $\mathcal{M}$: $S = (s_n)_{n \in \mathbf{N}}$ tq $\forall i \in \mathbf{N}$, $s_{2i} = 0$ et $s_{2i+1} = 1$

mot infini

LGBA

$$\begin{tikzcd}[column sep=small] & \text{mot infini} \Longrightarrow \omega\text{-}\textbf{mot} \arrow[dl] \arrow[dr] & \\ \omega\text{-}\textbf{automate} & & \omega\text{-langage} \end{tikzcd}$$

LGBA

mot infini $\Longrightarrow$ ω-**mot**

ω-**automate** ω-langage

Automate de Büchi Généralisé Étiquetté (LGBA)

$$\mathcal{B} = (\mathcal{Q}, I, F, \rightarrow, \mathcal{L})$$

éxécution acceptante

$$F = \{F_1, \ldots, F_n\} \subseteq (\mathcal{P}(\mathcal{Q}))^n$$

$\forall i, \exists q \in F_i$ tq on
passe par q infiniment souvent

LTL

Linear Temporal Logic (LTL)

LTL

Linear Temporal Logic (LTL)

Logique propositionnelle

$\bot$ $\top$ $\neg\varphi$ $\varphi_1 \wedge \varphi_2$

p $\varphi_1 \vee \varphi_2$

LTL

Linear Temporal Logic (LTL)

Logique propositionnelle $+$ Opérateurs temporels

$\bot$ $\quad \top$ $\quad \neg\varphi$ $\quad \varphi_1 \wedge \varphi_2$ $\qquad X\varphi$ $\quad F\varphi$ $\quad G\varphi$ $\quad \varphi_1 U \varphi_2$

p $\quad \varphi_1 \vee \varphi_2$ $\qquad \varphi_1 R \varphi_2$

LTL

Linear Temporal Logic (LTL)

Logique propositionnelle $+$ Opérateurs temporels

$\bot$ $\top$ $\neg\varphi$ $\varphi_1 \wedge \varphi_2$

p $\varphi_1 \vee \varphi_2$

$X\varphi$ $F\varphi$ $G\varphi$ $\varphi_1 U \varphi_2$

$\varphi_1 R \varphi_2$

$$\psi_1 \longrightarrow \varphi \longrightarrow \psi_2 \longrightarrow \psi_3$$

LTL

Linear Temporal Logic (LTL)

Logique propositionnelle + Opérateurs temporels

$\bot$ $\top$ $\neg\varphi$ $\varphi_1 \wedge \varphi_2$ p $\varphi_1 \vee \varphi_2$

$X\varphi$ $F\varphi$ $G\varphi$ $\varphi_1 U \varphi_2$ $\varphi_1 R \varphi_2$

$$\psi_1 \to \psi_2 \to \psi_3 \to \varphi$$

LTL

Linear Temporal Logic (LTL)

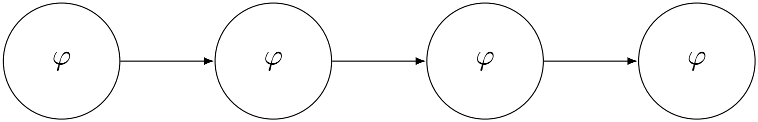
Logique propositionnelle $+$ Opérateurs temporels

$\bot$ $\top$ $\neg\varphi$ $\varphi_1 \wedge \varphi_2$

p $\varphi_1 \vee \varphi_2$

$X\varphi$ $F\varphi$ $G\varphi$ $\varphi_1 U \varphi_2$

$\varphi_1 R \varphi_2$



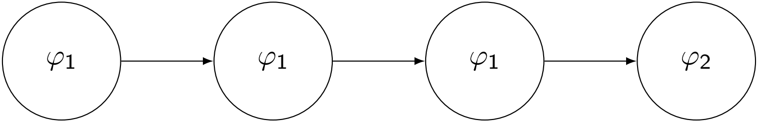
LTL

Linear Temporal Logic (LTL)

Logique propositionnelle + Opérateurs temporels

$\bot$ $\top$ $\neg\varphi$ $\varphi_1 \wedge \varphi_2$ p $\varphi_1 \vee \varphi_2$

$X\varphi$ $F\varphi$ $G\varphi$ $\varphi_1 U \varphi_2$ $\varphi_1 R \varphi_2$



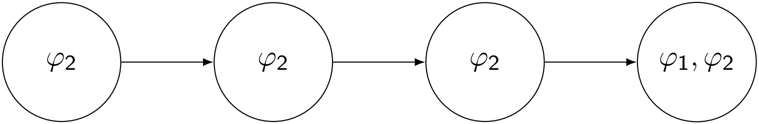
LTL

Linear Temporal Logic (LTL)

Logique propositionnelle + Opérateurs temporels

$\bot$ $\top$ $\neg\varphi$ $\varphi_1 \wedge \varphi_2$ p $\varphi_1 \vee \varphi_2$

$X\varphi$ $F\varphi$ $G\varphi$ $\varphi_1 U \varphi_2$ $\varphi_1 R \varphi_2$



Tester

But du Model-Checking : $\mathcal{M}$ satisfait φ ?

$$\boxed{(\mathcal{M}, \varphi)}$$

Tester

But du Model-Checking : $\mathcal{M}$ satisfait φ ?

$$\begin{tikzcd}
 & \boxed{(\mathcal{M}, \varphi)} \arrow[dl] \arrow[dr] & \\
\boxed{\mathcal{B}_{\mathcal{M}}} & & \boxed{\mathcal{B}_{\neg\varphi}}
\end{tikzcd}$$

Tester

But du Model-Checking : $\mathcal{M}$ satisfait φ ?

$$\begin{tikzcd} & (\mathcal{M},\varphi) \arrow[dl] \arrow[dr] & \\ \mathcal{B}_{\mathcal{M}} \arrow[dr] & & \mathcal{B}_{\neg\varphi} \arrow[dl] \\ & \mathcal{B}_{\mathcal{M}} \cap \mathcal{B}_{\neg\varphi} & \end{tikzcd}$$

Tester

But du Model-Checking : $\mathcal{M}$ satisfait φ ?

$$\begin{tikzcd}
& \boxed{(\mathcal{M}, \varphi)} \arrow[dl] \arrow[dr] & \\
\boxed{\mathcal{B}_{\mathcal{M}}} \arrow[dr] & & \boxed{\mathcal{B}_{\neg\varphi}} \arrow[dl] \\
& \boxed{\mathcal{B}_{\mathcal{M}} \cap \mathcal{B}_{\neg\varphi}} \arrow[d] & \\
& \boxed{L(\mathcal{B}_{\mathcal{M}} \cap \mathcal{B}_{\neg\varphi})} &
\end{tikzcd}$$

Tester

But du Model-Checking : $\mathcal{M}$ satisfait φ ?

$$
\begin{tikzcd}
 & (\mathcal{M}, \varphi) \arrow[dl] \arrow[dr] & \\
\mathcal{B}_{\mathcal{M}} \arrow[dr] & & \mathcal{B}_{\neg\varphi} \arrow[dl] \\
 & \mathcal{B}_{\mathcal{M}} \cap \mathcal{B}_{\neg\varphi} \arrow[d] & \\
 & L(\mathcal{B}_{\mathcal{M}} \cap \mathcal{B}_{\neg\varphi}) \arrow[dl] & \\
\neq \emptyset \arrow[d] & & \\
\mathcal{M} \not\models \varphi & &
\end{tikzcd}
$$

Tester

But du Model-Checking : $\mathcal{M}$ satisfait φ ?

$$\begin{tikzcd}
 & (\mathcal{M}, \varphi) \arrow[dl] \arrow[dr] & \\
\mathcal{B}_{\mathcal{M}} \arrow[dr] & & \mathcal{B}_{\neg\varphi} \arrow[dl] \\
 & \mathcal{B}_{\mathcal{M}} \cap \mathcal{B}_{\neg\varphi} \arrow[d] & \\
 & L(\mathcal{B}_{\mathcal{M}} \cap \mathcal{B}_{\neg\varphi}) \arrow[dl] \arrow[dr] & \\
\neq \emptyset \arrow[d] & & = \emptyset \arrow[d] \\
\mathcal{M} \not\models \varphi & & \mathcal{M} \models \varphi
\end{tikzcd}$$

1. Définitions

2. Implémentation LTL
LTL en ocaml
Mettre une formule sous forme normale négative
Transformation LTL $\rightarrow$ LGBA
Transformer f en LGBA

3. Faire l'intersection entre deux automates de Büchi

4. Annexe

Définition formelle de la logique LTL (linear temporal logic)

LTL :

- $\top$
- $\bot$
- p, p $\in$ AP
- $\neg\ \varphi$, $\varphi \in$ LTL
- $\varphi \wedge \psi$, $(\varphi, \psi) \in$ LTL $\times$ LTL
- $\varphi \vee \psi$, $(\varphi, \psi) \in$ LTL $\times$ LTL
- X φ, $\varphi \in$ LTL (Next)
- G φ, $\varphi \in$ LTL (**G**lobally)
- F φ, $\varphi \in$ LTL (**F**inally)
- φ U ψ, $(\varphi, \psi) \in$ LTL $\times$ LTL (**U**ntil)
- φ R ψ, $(\varphi, \psi) \in$ LTL $\times$ LTL (**R**elease)

```ocaml
type ltl =
    | Top
    | Bot
    | Atom of string
    | Not of ltl
    | Or of ltl*ltl
    | And of ltl*ltl
    | X of ltl
    | G of ltl
    | F of ltl
    | U of ltl*ltl
    | R of ltl*ltl
```

Définition d'une FNN (forme normale négative)

Une formule LTL est dite sous forme normale négative lorsque :

- $\rightarrow$ L'opérateur $\neg$ est appliqué uniquement aux propositions atomiques
- $\rightarrow$ Seules les opérateurs : $\vee$, $\wedge$, X, U, R peuvent apparaître devant les littéraux (On abandonne G, F)

Définition d'une FNN (forme normale négative)

Une formule LTL est dite sous forme normale négative lorsque :

- $\rightarrow$ L'opérateur $\neg$ est appliqué uniquement aux propositions atomiques
- $\rightarrow$ Seules les opérateurs : $\vee$, $\wedge$, X, U, R peuvent apparaître devant les littéraux (On abandonne G, F)

Exemple

$$(pUq) \wedge (X(\neg q))$$

$$\neg(pUq)$$

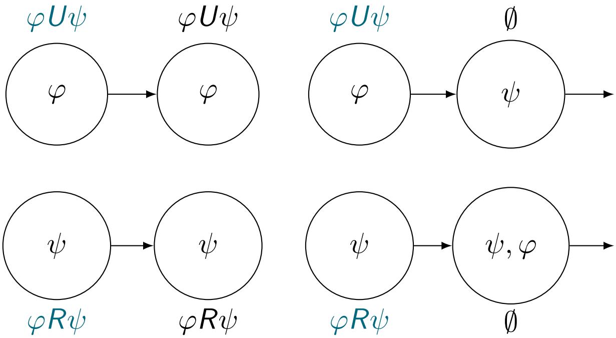
$$(Xp)U(F(\neg q))$$

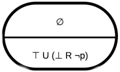
Les règles utilisées

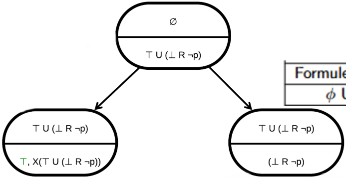
Loi de De Morgan	
Formule initiale	Formule équivalente
$\neg\ (\varphi \land \psi)$	$(\neg\ \varphi) \lor (\neg\ \psi)$
$\neg\ (\varphi \lor \psi)$	$(\neg\ \varphi) \land (\neg\ \psi)$

Règles utilisées

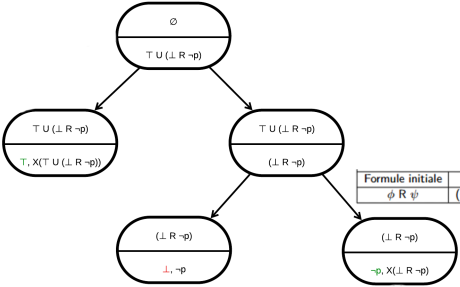
"Formules d'expansion"	
Formule initiale	Formule équivalente
φ U ψ	$\psi \vee (\varphi \wedge$ X$(\varphi$ U $\psi))$
φ R ψ	$(\psi \wedge \varphi) \vee (\psi \wedge$ X$(\varphi$ R $\psi))$



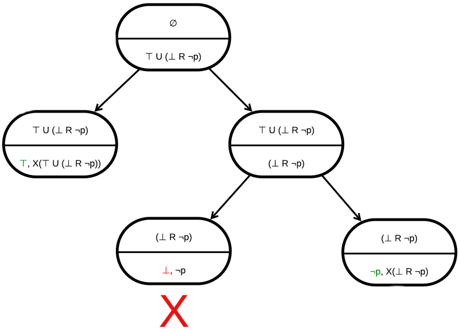

∅
⊤ U (⊥ R ¬p)

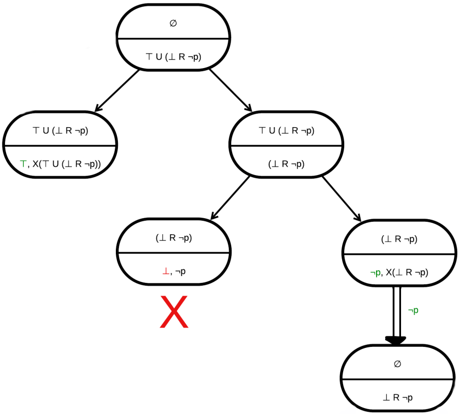


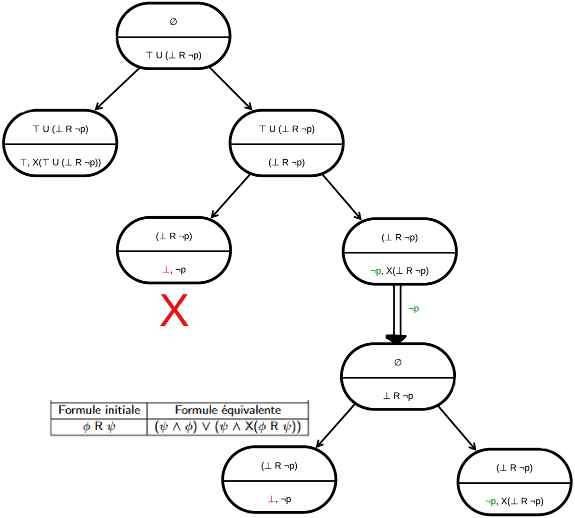
Formule initiale	Formule équivalente
ϕ U ψ	$\psi \vee (\phi \wedge$ X$(\phi$ U $\psi))$



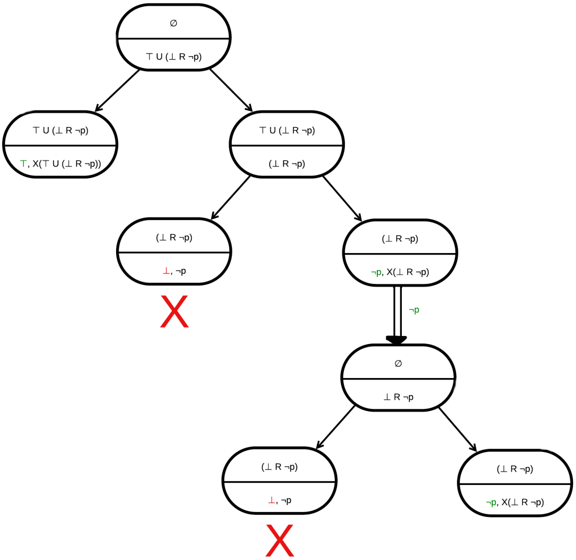
Formule initiale	Formule équivalente
ϕ R ψ	$(\psi \wedge \phi) \vee (\psi \wedge X(\phi \text{ R } \psi))$

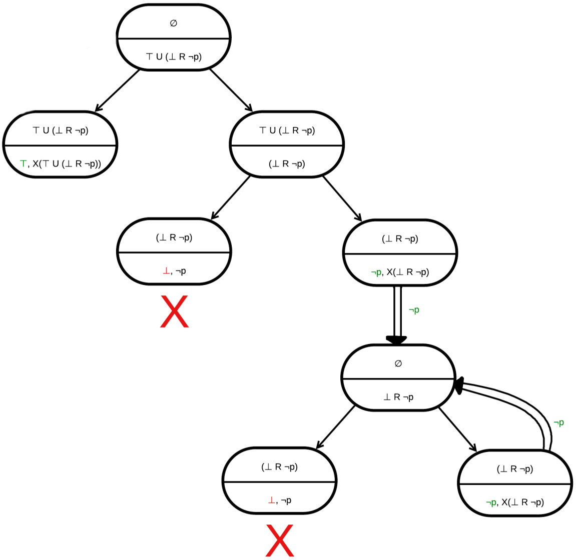

∅
⊤ U (⊥ R ¬p)
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
⊥, ¬p
X
(⊥ R ¬p)
¬p, X(⊥ R ¬p)

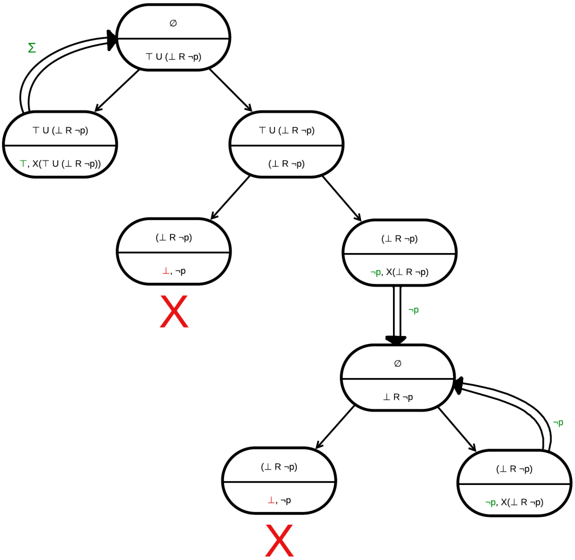

∅
⊤ U (⊥ R ¬p)
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
⊥, ¬p
X
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p
∅
⊥ R ¬p

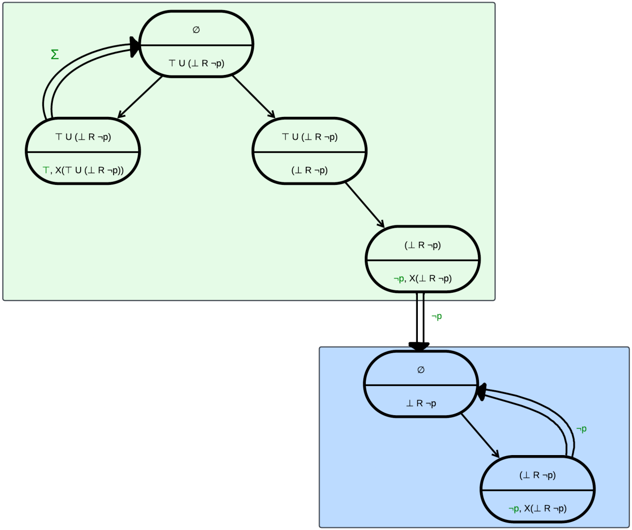


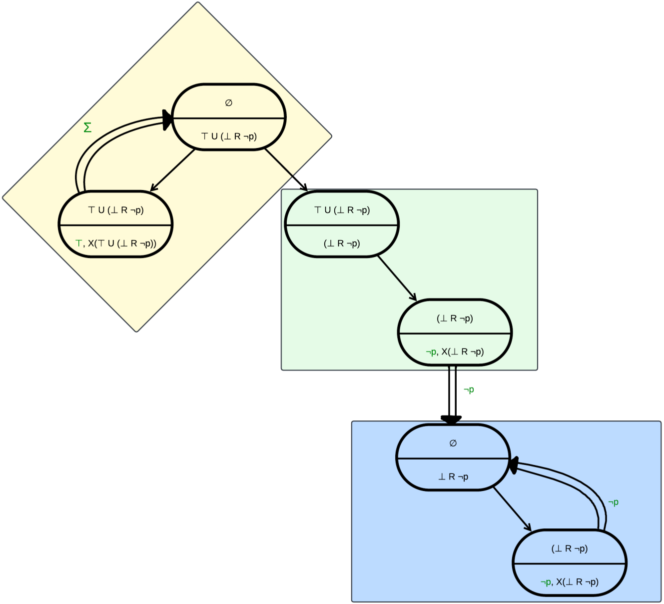
Formule initiale	Formule équivalente
ϕ R ψ	$(\psi \land \phi) \lor (\psi \land X(\phi \text{ R } \psi))$

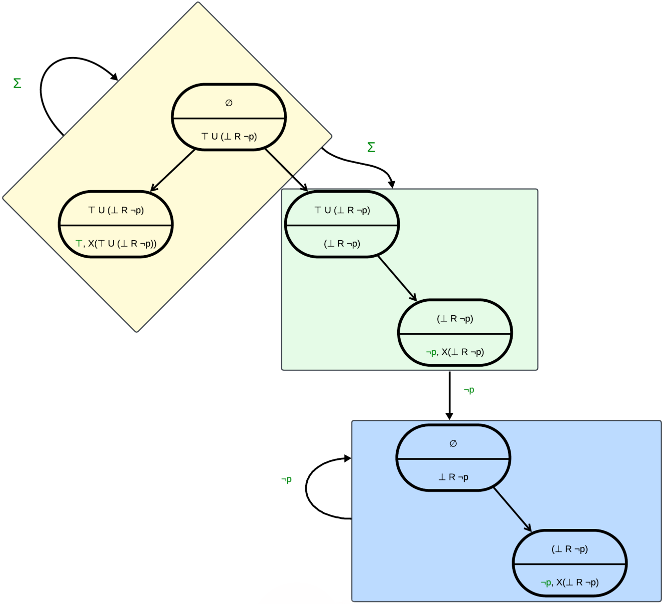

∅
⊤ U (⊥ R ¬p)
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
⊥, ¬p
X
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p
∅
⊥ R ¬p
(⊥ R ¬p)
⊥, ¬p
X
(⊥ R ¬p)
¬p, X(⊥ R ¬p)

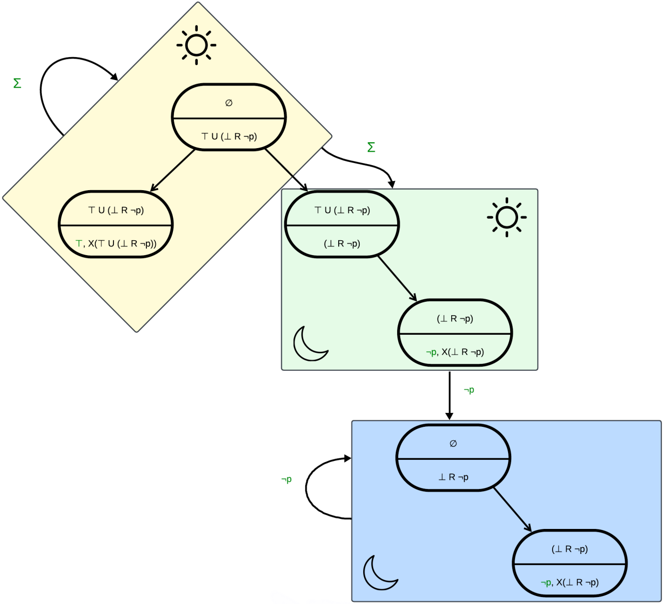

∅
⊤ U (⊥ R ¬p)
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
⊥, ¬p
X
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p
∅
⊥ R ¬p
(⊥ R ¬p)
⊥, ¬p
X
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p

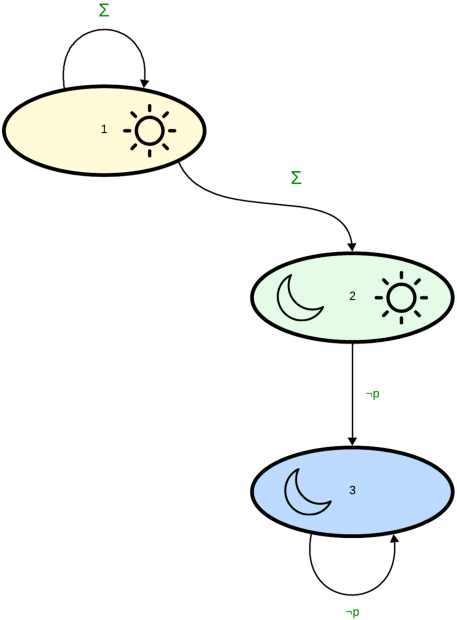

∅
⊤ U (⊥ R ¬p)
Σ
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
⊥, ¬p
X
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p
∅
⊥ R ¬p
(⊥ R ¬p)
⊥, ¬p
X
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p

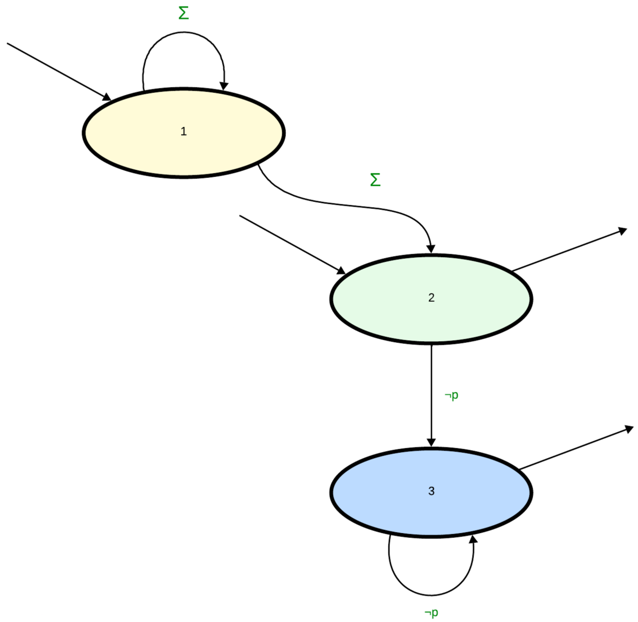

∅
⊤ U (⊥ R ¬p)
Σ
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p
∅
⊥ R ¬p
¬p
(⊥ R ¬p)
¬p, X(⊥ R ¬p)

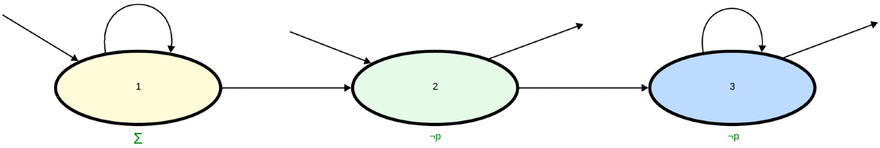

∅
⊤ U (⊥ R ¬p)
Σ
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p
∅
⊥ R ¬p
¬p
(⊥ R ¬p)
¬p, X(⊥ R ¬p)


Σ
∅
⊤ U (⊥ R ¬p)
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
Σ
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p
¬p
∅
⊥ R ¬p
(⊥ R ¬p)
¬p, X(⊥ R ¬p)


Σ
∅
⊤ U (⊥ R ¬p)
⊤ U (⊥ R ¬p)
⊤, X(⊤ U (⊥ R ¬p))
Σ
⊤ U (⊥ R ¬p)
(⊥ R ¬p)
(⊥ R ¬p)
¬p, X(⊥ R ¬p)
¬p
¬p
∅
⊥ R ¬p
(⊥ R ¬p)
¬p, X(⊥ R ¬p)


Σ
1
Σ
2
¬p
3
¬p


Σ
1
Σ
2
¬p
3
¬p


1
2
3
Σ
¬p
¬p

1. Définitions

2. Implémentation LTL

3. Faire l'intersection entre deux automates de Büchi
    Passer de structure de Kripke à LGBA
    Intersection de deux LGBA
    Test de vacuité

4. Annexe

Transformation d'une structure de Kripke vers un LGBA

Soit Σ un alphabet.
Soit K = (Q, I, $\rightarrow$, L) une structure de Kripke.
On pose A = (Σ ,Q', I', F, δ, L') un labelled generalized Büchi automaton (LGBA) où

- $\rightarrow$ Q' = Q
- $\rightarrow$ I' = I
- $\rightarrow$ F = Q
- $\rightarrow$ $\delta = \rightarrow$
- $\rightarrow$ L' = L

Intersection de deux LGBA

Soit $A_1 = (\Sigma, Q_1, I_1, F_1, \delta_1, \mathcal{L}_1)$ et
$A_2 = (\Sigma, Q_2, I_2, F_2, \delta_2, \mathcal{L}_2)$ deux LGBA.
On pose A_p l'automate produit $(\Sigma, Q, I, F, \delta, \mathcal{L})$ où

- $Q = Q_1 \times Q_2$
- $I = I_1 \times I_2$
- $F = F_1 \times F_2$
- $\delta = \{((\text{q,s}),(\text{q',s'})) \in Q \times Q \;/\; (\text{q,q'}) \in \delta_1$ et $(\text{s,s'}) \in \delta_2$ et $(\mathcal{L}_1(\text{q}) \cap \mathcal{L}_2(\text{q'})) \neq \emptyset \}$
- $\mathcal{L} = \{((\text{q,q'}), (\mathcal{L}_1(\text{q}) \cap \mathcal{L}_2(\text{q'})))\}$

Exemple

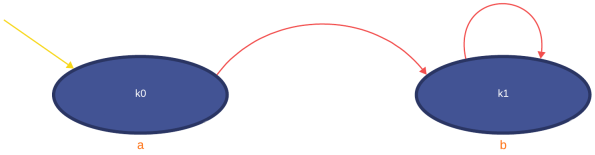


Figure – Structure de Kripke vérifiant G(F b)

Exemple

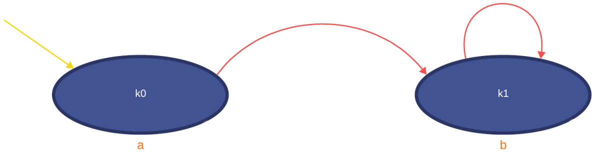


Figure – Structure de Kripke vérifiant G(F b)

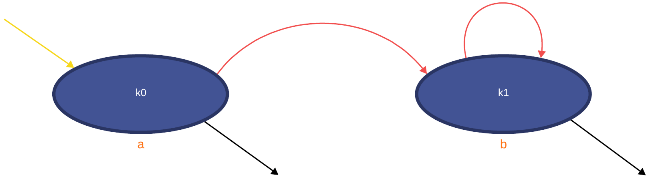


Figure – LGBA associé à la structure de Kripke

Exemple

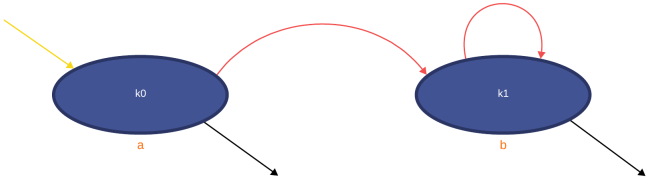


Figure – LGBA associé à la structure de Kripke

Exemple

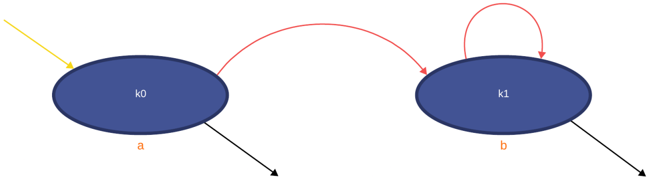


Figure – LGBA associé à la structure de Kripke

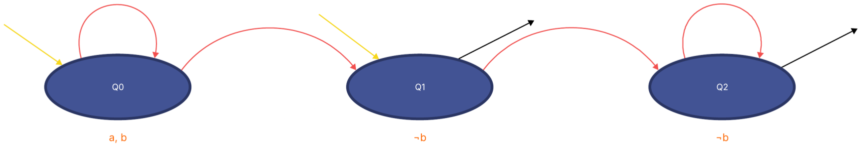


Figure – LGBA associé à la formule ¬G(F b)

Exemple

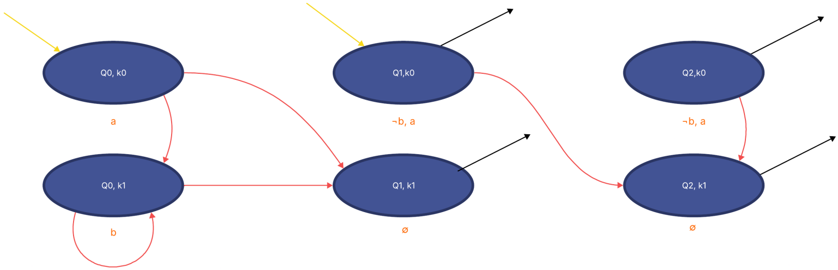
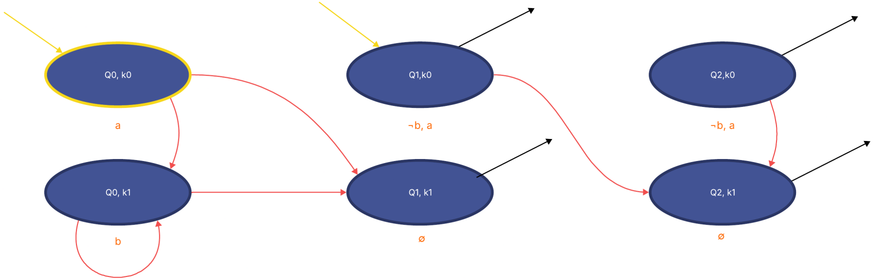


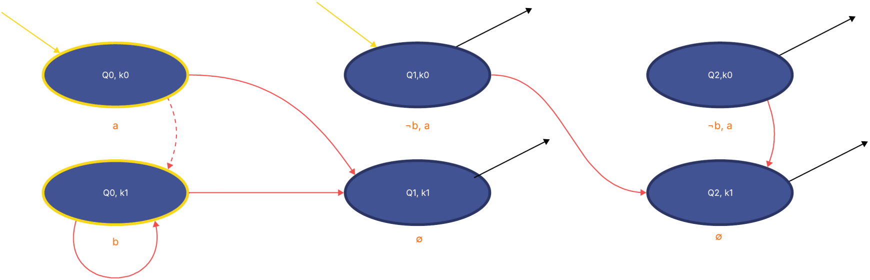
Figure – Intersection des deux LGBA

Exemple d'exécution



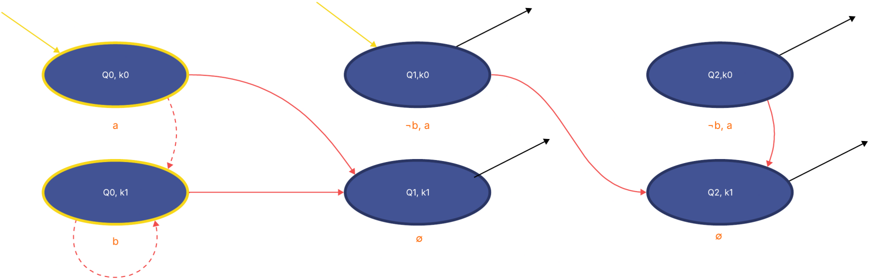
outerdfs Visités : [(Q0,k0)]

Exemple d'exécution



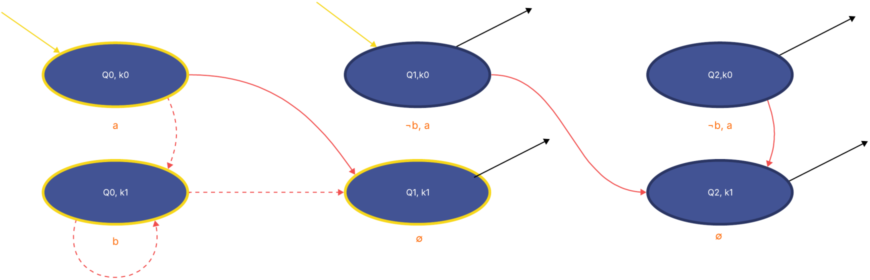
outerdfs Visités : [(Q0,k0), (Q0,k1)]

Exemple d'exécution



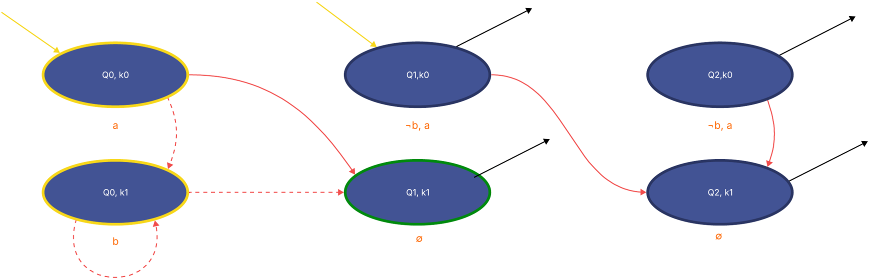
outerdfs Visités : [(Q0,k0), (Q0,k1)]

Exemple d'exécution



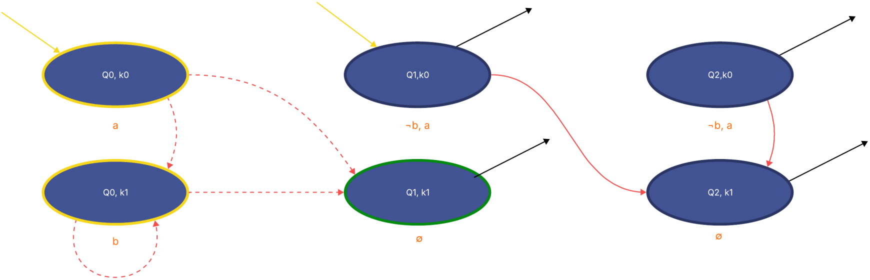
outerdfs Visités : [(Q0,k0), (Q0,k1), (Q1,k1)]

Exemple d'exécution



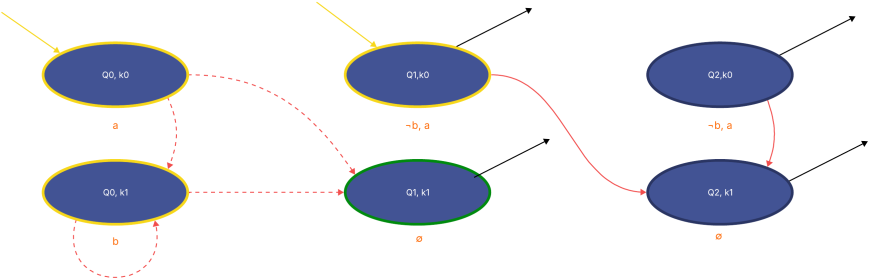
outerdfs Visités : [(Q0,k0), (Q0,k1), (Q1,k1)]
innerdfs1 VisitésIn1 : [(Q1,k1)]
innerdfs1 VisitésExt1 : [(Q0,k0), (Q0,k1), (Q1,k1)]

Exemple d'exécution



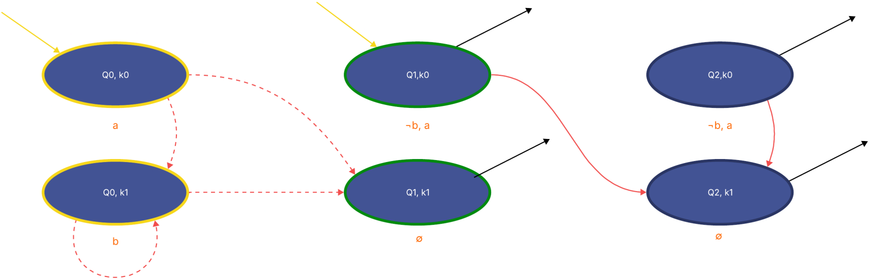
outerdfs Visités : [(Q0,k0), (Q0,k1), (Q1,k1)]
innerdfs1 VisitésIn1 : [(Q1,k1)]
innerdfs1 VisitésExt1 : [(Q0,k0), (Q0,k1), (Q1,k1)]

Exemple d'exécution



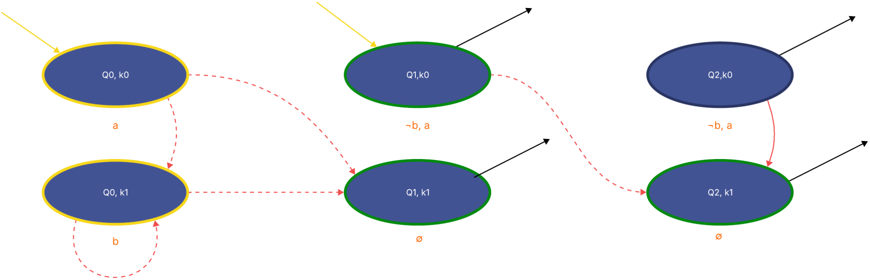
outerdfs Visités : [(Q1,k0)]
innerdfs1 VisitésIn1 : [(Q1,k1)]
innerdfs1 VisitésExt1 : [(Q0,k0), (Q0,k1), (Q1,k1)]

Exemple d'exécution



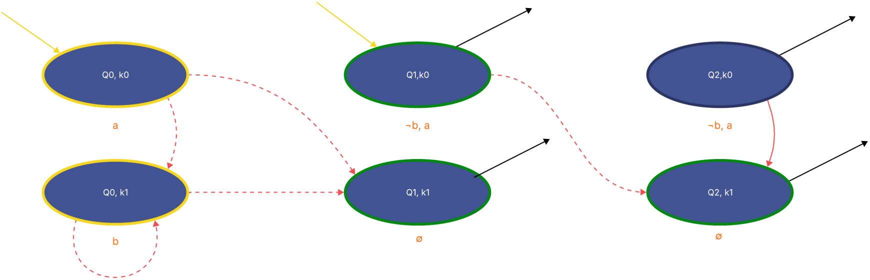
outerdfs Visités : [(Q1,k0)]
innerdfs1 VisitésIn1 : [(Q1,k1)]
innerdfs1 VisitésExt1 : [(Q0,k0), (Q0,k1), (Q1,k1)]
innerdfs2 VisitésIn2 : [(Q1,k0)]
innerdfs2 VisitésExt2 : [(Q1,k0)]

Exemple d'exécution



outerdfs Visités : [(Q1,k0)]
innerdfs1 VisitésIn1 : [(Q1,k1)]
innerdfs1 VisitésExt1 : [(Q0,k0), (Q0,k1), (Q1,k1)]
innerdfs2 VisitésIn2 : [(Q1,k0), (Q2,k1)]
innerdfs2 VisitésExt2 : [(Q1,k0)]

Exemple d'exécution



outerdfs Visités : [(Q1,k0)]
innerdfs1 VisitésIn1 : [(Q1,k1)]
innerdfs1 VisitésExt1 : [(Q0,k0), (Q0,k1), (Q1,k1)]
innerdfs2 VisitésIn2 : [(Q1,k0), (Q2,k1)]
innerdfs2 VisitésExt2 : [(Q1,k0)]
Estvide $\rightarrow$ Vrai

Conclusion

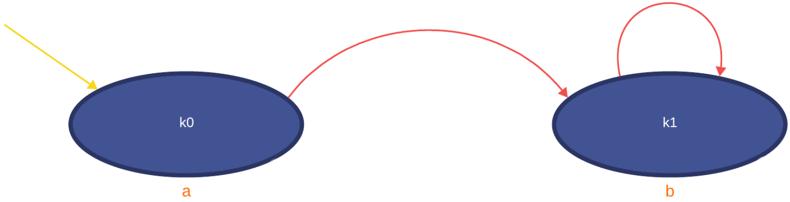


Figure – Structure de Kripke vérifiant G(F b)

1. Définitions

2. Implémentation LTL

3. Faire l'intersection entre deux automates de Büchi

4. Annexe
   Code
   Langage
   Algorithmes
   Règles pour transformer en FNN
   FNN
   Notions d'ensembles en Ocaml

Code du TIPE

```
type ltl =
  | Top
  | Bot
  | Atom of string
  | Not of ltl
  | Or of ltl*ltl
  | And of ltl*ltl
  | X of ltl
  | G of ltl
  | F of ltl
  | U of ltl*ltl
  | R of ltl*ltl

type noeud = {name : string}

type state = {etat : string}

type kripke_struct = {
  q : state list;
  init: state list;
  transition : (state * state) list;
  valuation : (state * (ltl list)) list ;
  (* une fonction qui à un état associe un ensemble de formules ltl*)
  (* Cf définition dans le diapo *)
}

```

Code du TIPE

```
type buchi_automate = {
  q : state list;
  init : state list;
  final : state list;
  transition : (state * state) list;
  valuation : (state * (ltl list)) list
  (* La valuation correspond au "L" dans la définition du LGBA (pour labelled generalised
  ↪  buchi automaton) *)
}

(* Fin des declarations de types *)

(* On va maintenant coder un système d'ensemble avec des listes *)
(* Exactement les mêmes que pour le model checker ctl *)
let rec prive (a: 'a list ) (b: 'a list) : 'a list =
  (* Réalise A\B *)
  match a with
  | [] -> []
  | x::q -> if (List.mem x b) then prive q b
  else x::(prive q b)

```

Code du TIPE

```
let rec union (a: 'a list) (b: 'a list) : 'a list =
  (* Réalise A U B (sans doublons si a et b sont sans doublons)*)
  match a with
  | [] -> b
  | x::q -> if (List.mem x b) then union q b
  else x::(union q b)

let rec inter (a: 'a list) (b:'a list) : 'a list =
  (* réalise a inter b *)
  match a with
  | [] -> []
  | x::q -> if (List.mem x b) then x::(inter q b)
  else inter q b

let appartient (x: 'a) (a: 'a list) :bool =
  List.mem x a

let inclus (a: 'a list) (b: 'a list) : bool =
  (* Test si A inclus dans B *)
  match (prive a b) with (* A\B est vide ssi A inclus dans B *)
  | [] -> true
  | _ -> false

exception Elem_pas_present
(* Supprime x de a, et lève une exception si x n'est pas dans a*)
let rec suppr (x:'a) (a: 'a list) : 'a list =
  match a with
  | [] -> raise Elem_pas_present
  | y::q -> if (x=y) then q
  else y::(suppr x q)
```

Code du TIPE

```
(* Fonction auxiliaire qui décide une égalité ensembliste entre a et b *)
let rec egal_ens_aux (a: 'a list) (b : 'a list) : bool =
  match a with
  | [] -> (b = [])
  | x::q -> egal_ens_aux q (suppr x b)

let egal_ens (a: 'a list) (b:'a list) : bool =
  try
    egal_ens_aux a b
  with
  | Elem_pas_present -> false

(* Fin de l'implémentation des ensembles *)

(* Renvoie une formule équivalente à f, ne contenant plus aucun F, G
cf wikipedia pour les règles utilisées*)
let suppr_fg (phi : ltl) : ltl =
  match phi with
  | F psi -> U (Top, psi)
  | G psi -> R (Bot, psi)
  | _ -> phi

(* Cette fonction nous servira a transformé notre formule en
automate de büchi généralisé, en effet, on doit d'abord transformer
notre formule f en forme normal négative *)
```

Code du TIPE

```
(* Transforme une formule f en une formule f' équivalente
mise sous forme normale négative *)
let rec fnn (psi:ltl) : ltl =
  match psi with
  | Top | Bot | Atom _ -> psi
  | Or (psi1, psi2) -> Or (fnn psi1, fnn psi2)
  | And (psi1, psi2) -> And (fnn psi1, fnn psi2)
  | U (psi1, psi2) -> U (fnn psi1, fnn psi2)
  | R (psi1, psi2) -> R (fnn psi1, fnn psi2)
  | X phi -> X (fnn phi)
  | F _ | G _ -> fnn (suppr_fg psi)
  | Not phi ->
    (* On va ici appliqué les lois de De Morgan *)
    (* Et les lois de dualités spécifiques à la logique LTL *)
    begin
      match phi with
      | Top ->  Bot
      | Bot -> Top
      | Atom _ -> Not phi
      | Not psi2 -> (fnn psi2) (* Elimination de la double négation*)
      | Or (psi1,psi2) -> And (fnn (Not psi1), fnn (Not psi2)) (* Loi de De Morgan *)
      | And (psi1,psi2) -> Or (fnn (Not psi1), fnn (Not psi2))
      | X psi2 -> X (fnn (Not psi2))
      | F _ | G _ -> fnn (Not (suppr_fg phi))
      | U (psi1, psi2) -> R (fnn (Not psi1), fnn (Not psi2))
      | R (psi1, psi2) -> U (fnn (Not psi1), fnn (Not psi2))
    end
```

Code du TIPE

```
(* On a ici des formules ne contenant pas de F, de G, et dont toutes les négations précèdes les litteraux
(mis sous forme normale négative) *)
let curr1 (phi: ltl) : ltl list =
  match phi with
  | U (psi1, psi2) -> [psi1]
  | R (psi1, psi2) -> [psi2]
  | Or (psi1, psi2) -> [psi2]
  | _ -> []

let next1 (phi: ltl) : ltl list =
  match phi with
  | U (psi1, psi2) -> [phi]
  | R (psi1 ,psi2) -> [phi]
  | _ -> []

let curr2 (phi : ltl): ltl list =
  match phi with
  | U (psi1, psi2) -> [psi2]
  | R (psi1, psi2) -> [psi1; psi2]
  | Or (psi1, psi2) -> [psi1]
  | _ -> []

let neg (phi : ltl) : ltl =
  match phi with
  | Top -> Bot
  | Bot -> Top
  | Atom p -> Not phi
  | Not (Atom p) -> Atom p
  | _ -> phi
```

Code du TIPE

```
(* Récupère l'ensemble d'éléments associé à un noeud dans set  *)
let rec get_set (set : (noeud * ('a list)) list) (q:noeud) : 'a list =
  match set with
  | [] -> failwith "From get_set : Valeur non présente dans set"
  | (x,l)::p -> if (x.name = q.name) then l else get_set p q

(* Renvoie incoming_u en version modifiée de tels manière que Incoming_u(q) := Incoming_u(q)
↪  union incoming *)
let rec edit_incoming (incoming_u : ((noeud * (noeud list)) list)) (incoming : noeud list)
↪  (q:noeud): (noeud * (noeud list)) list =
  match incoming_u with
  | [] -> [q,incoming]
  | (x,l)::p -> if (x.name = q.name) then (q, union l incoming)::p
  else (x,l)::(edit_incoming p incoming q)

```

Code du TIPE

```
(* Renvoie true si Il existe q appartient à  nodeset / q.next=node.next et q.old=node.old et dans ce cas la : modifie
q.incoming par q.incoming union node.incoming
Sinon renvoie false*)
let rec exists (next_u : (noeud * (ltl list)) list ) (next : ltl list) (now_u : (noeud * (ltl list)) list) (old: ltl list) (incoming: noeud list) (incoming_u : ((noeud * (noeud list)) list) ref) (nodeset: noeud list) : bool=
  match nodeset with
  | [] -> false
  | q::p -> let lnext = get_set next_u q in
  let lnow = get_set now_u q in
  if ((egal_ens lnext next) && (egal_ens lnow old)) then (
    let incoming_bis = edit_incoming (!incoming_u) incoming q in
    incoming_u := incoming_bis;
    let _ = exists next_u next now_u old incoming incoming_u p in true
  )
  else exists next_u next now_u old incoming incoming_u p

(* Renvoie le plus grand nombre présent dans nodeset *)
let rec max_name (l: noeud list) =
  match l with
  | [] -> 0
  | x::q -> max (int_of_string (x.name)) (max_name q)
```

Code du TIPE

```
let expand (curr: ltl list) (old : ltl list) (next : ltl list) (incoming : noeud list)  =
  let nodes_u = ref [] in
  let incoming_u = ref [] in
  let now_u = ref [] in
  let next_u = ref [] in
  let rec expand_aux (curr: ltl list) (old : ltl list) (next : ltl list) (incoming : noeud
  ↪  list) : unit =
    if curr = [] then (
      if (exists (!next_u) next (!now_u) old incoming incoming_u (!nodes_u)) then ()
      else (
        let q = {
          name = string_of_int ((max_name (!nodes_u))+1)
        } in
        nodes_u := union (!nodes_u) [q];
        incoming_u := union (!incoming_u) [(q, incoming)];
        now_u := union (!now_u) [(q,old)];
        next_u := union (!next_u) [(q,next)];
        (expand_aux next [] [] [q])
      )
    )
```

Code du TIPE

```
    else (
      let f = List.hd curr in
      let curr3 = List.tl curr in
      let old3 = union old [f] in
      match f with
      | Top | Bot | Atom _ | Not (Atom _) ->
        if ((f= Bot) || (appartient (neg f) old3)) then ()
        else (expand_aux curr3 old3 next incoming)
      | And (f1,f2) ->
        (expand_aux (union curr3 (prive [f1;f2] old3)) old3 next incoming)
      | X g ->
        (expand_aux curr3 old3 (union next [g]) incoming)
      | Or (f1,f2) | U (f1,f2) | R (f1,f2) ->
        (expand_aux (union curr3 (prive (curr1 f) old3)) old3 (union next (next1 f))
     ↪  incoming);
        (expand_aux (union curr3 (prive (curr2 f) old3)) old3 next incoming)
      | Not _ | G _ | F _ -> failwith "Formule non mise sous forme normale négative"
    )
    in (expand_aux curr old next incoming); (!nodes_u, !now_u, !incoming_u)
```

Code du TIPE

```
let create_graph (f :ltl) =
  let init = {name = "init"} in
  expand [f] [] [] [init]

(* Transforme une structure de kripke en un automate de buchi, (tout les états sont finaux)
On renomme ici les états par k_(nom de l'état) afin de pouvoir différencier les états provenants d'une structure
de kripke de ceux provenant d'un automate de buchi*)
let kripke_to_buchi (k: kripke_struct) : buchi_automate =
  {q = List.map (fun x -> {etat = "k_" ^ x.etat}) (k.q);
  init = List.map (fun x -> {etat = "k_" ^ x.etat}) (k.init);
  transition = List.map (fun (x,y) -> ({etat = "k_" ^ x.etat}, {etat = "k_" ^ y.etat})) (k.transition);
  final = List.map (fun x -> ({etat = "k_" ^ x.etat})) (k.q);
  valuation = List.map (fun (x,f) -> ({etat = "k_" ^ x.etat}, f)) (k.valuation)}
```

Code du TIPE

```
(* Construit Trans = {(q,q')| q,q' appartient à  Nodes and q appartient à  Incoming(q') } *)
let rec build_transition (nodes: noeud list) (incoming: (noeud * (noeud list)) list) :
↪ (state*state) list  =
  (* Renvoie  {(node,q')| node,q' appartient à  Nodes and node appartient à  l }*)
  let rec build_transition_with_list (nodes :noeud list) (node : noeud) (l: noeud list) :
  ↪ (state*state) list =
    match l with
    | [] -> []
    | q::p -> if (appartient q nodes) then ({etat = q.name},{etat =
    ↪ node.name})::(build_transition_with_list nodes node p)
    else (build_transition_with_list nodes node p)
  in
  match incoming with
  | [] -> []
  | (q',l)::p -> if ((appartient q' nodes)) then (build_transition_with_list nodes q'
  ↪ l)@(build_transition nodes p)
  else (build_transition nodes p)

(* Construit Q0 = { q appartient à  Nodes | init appartient à  Incoming(q) } *)
let rec build_init (nodes : noeud list) (incoming: (noeud*(noeud list)) list) (init: noeud) :
↪ state list =
  match incoming with
  | [] -> []
  | (q,l)::p -> if ((appartient init l) && (q.name <> "init")) then ({etat =
  ↪ q.name})::(build_init nodes p init)
  else (build_init nodes p init) (* on exclus ici init avec la condition q.name <> "init" car
  ↪ on veut les qappartient à  Nodes et init n'est pas
  dans Nodes *)
```

Code du TIPE

```
(* Extrait  l'ensemble des formules de now (qui est de type (noeud*ltl))*)
let rec extract_ltl (now: (noeud*ltl) list) : ltl list =
  match now with
  | [] -> []
  | (q,f)::p -> let res = extract_ltl p in
    if (appartient f res) then res
    else f::res

(* On va maintenant construire les états finaux *)

(* Construit Fg = { q appartient à  Nodes | g2 appartient à  Now(q) or g n'appartient pas à
↪  Now(q) } où g = g1 U g2*)
let rec build_fg (nodes: noeud list) (now: (noeud* (ltl list)) list) (g1:ltl) (g2: ltl) :
↪  state list =
  match now with
  | [] -> []
  | (q,l)::p -> let res = (build_fg nodes p g1 g2) in
    if ((appartient g2 l || not (appartient (U (g1,g2)) l)) && (not (appartient {etat =
    ↪  q.name} res)))
      then ({etat = q.name}::res)
  else res

(* Détermine si g est une sous formule de f *)
let rec est_sous_formule (g: ltl) (f: ltl) : bool =
  if (f=g) then true else
  match f with
  | Top | Bot | Atom _ -> (f=g) (*false donc*)
  | Not f' | X f' | F f' | G f' -> est_sous_formule g f'
  | Or (f1,f2) | And (f1,f2) | U (f1,f2) | R (f1,f2)  -> (est_sous_formule g f1) ||
 ↪  (est_sous_formule g f2)
```

Code du TIPE

```
(* Extrait les sous formule de f de la forme phi U psi  *)
let rec extrait_sous_formule (f: ltl) : ltl list =
  match f with
  | Atom p -> [U (Atom p, Atom p);U( Not (Atom p), Not (Atom p))]
  | Top -> [U (Top, Top)]
  | Bot -> []
  | U (f1,f2) -> f::(union (extrait_sous_formule f1) (extrait_sous_formule f2))
  | And (f1,f2) | Or (f1,f2) -> (union (extrait_sous_formule f1) (extrait_sous_formule f2))
  | R (f1,f2) ->
      (* R(f1,f2) équivaut à Not U (Not f1,Not f2) *)
      f::(union (extrait_sous_formule (Not f1)) (extrait_sous_formule (Not f2)))
  | X f1 | Not f1 -> extrait_sous_formule f1
  | G _ | F _ -> failwith"formule non mise sous forme normale négative"

(* Transforme un ensemble de noeud en un ensemble d'états *)
let rec nodes_to_state (nodes : noeud list) : state list =
  match nodes with
  | [] -> []
  | x::q -> {etat = x.name}::(nodes_to_state q)

let build_final (f: ltl) (nodes: noeud list) (now : (noeud* (ltl list)) list) : state list =
  let rec aux (set : ltl list) : state list list =
  (* Construit F = { Fg | g appartient à  cl( f ) } *)
  match set with
  | [] -> []
  | U(g1,g2)::q -> (build_fg nodes now g1 g2)::(aux q)
  | R(g1,g2)::q -> (build_fg nodes now (Not g1) (Not g2))::(aux q)
  | phi::q -> failwith"ne devrait pas arriver"
  in
  List.fold_left (fun x y -> inter x y) (nodes_to_state nodes) (aux (extrait_sous_formule f))
  (* On prend l'intersection de tout les ensembles obtenues avec aux *)
```

Code du TIPE

```
(* Extrait les littéraux d'un ensemble de couple (noeud, ensemble de formule)  *)
let rec litteraux (q: noeud) (now: (noeud* (ltl list)) list) : ltl list =
  let rec litteraux_aux (ltlset: ltl list) : ltl list =
    (* Récupère tout les littéraux d'un ensemble de formule ltlset *)
    match ltlset with
    | [] -> []
    | f::p -> let res = (litteraux_aux p) in
      begin
        match f with
        | Atom p  -> if (not (appartient f res)) then (Atom p)::res
        else res
        | Not (Atom p) -> if (not (appartient f res)) then (Not (Atom p))::res
        else res
        | _ -> res
      end
    in
  match now with
  | [] -> []
  | (x,l)::p -> if (x.name = q.name) then (
      (litteraux_aux l)@(litteraux q p)
  )
  else (litteraux q p)
```

Code du TIPE

```
(* Construit la valuation pour l'automate de buchi *)
let rec build_valuation (nodes : noeud list) (now : (noeud* (ltl list)) list) : (state * (ltl
↪ list)) list =
  match nodes with
  | [] -> []
  | q::p ->  ({etat = q.name}, litteraux q now)::(build_valuation p now)

(* Construit l'ensemble d'états *)
let rec build_state (nodes : noeud list) : state list =
  match nodes with
  | [] -> []
  | q::p -> {etat = q.name}::(build_state p)

(* Récupère le noeud initial dans nodes *)
let rec get_init (nodes : noeud list) : noeud =
  match nodes with
  | [] -> failwith"noeud initial non trouvé"
  | x::q -> if (x.name = "init") then x
  else get_init q
```

Code du TIPE

```
let build_lgba (f: ltl) : buchi_automate =
  let init = {name = "init"} in
  let (nodes,now,incoming) = create_graph f in
  {
    q = build_state nodes;
    init = build_init nodes incoming init;
    transition = build_transition nodes incoming ;
    final = build_final f nodes now;
    valuation = build_valuation nodes now
  }

(* On va donc maintenant coder l'intersection entre deux automates de buchi, l'un d'eux sera une
structure de kripke qui aura été transformé en automate de buchi *)

(* Réalise l'opération q1 x q2 (produit cartésien des états) *)
let rec prod_states (q1: state list) (q2: state list) : (state*state) list =
  let rec prod_cart (q: state) (l: state list) : (state*state) list =
    match l with
    | [] -> []
    | y::p -> (q,y)::(prod_cart q p)
  in
  match q1 with
  | [] -> []
  | x::l -> let res = prod_states l q2 in
    union (prod_cart x q2) res
```

Code du TIPE

```
(* Permet d'obtenir les états initiaux de l'automate produit, I = {(q,q') / q appartient à I1, q' appartient à  I2} *)
(* On va donc naturellement utilisé la fonction précédente *)
let prod_init (b1 : buchi_automate) (b2 : buchi_automate) : (state*state) list =
  prod_states (b1.init) (b2.init)

(* Un état (q,q') est final pour l'automate produit ssi q final pour b1 et q' final pour b2 *)
let prod_final (b1: buchi_automate) (b2:buchi_automate) : (state*state) list =
  prod_states (b1.final) (b2.final)

(* Récupère l'ensemble des transitions sortantes de q *)
let rec get_transition (q: state) (t: (state * state) list) : (state * state) list =
  match t with
  | [] -> []
  | (x,y)::p -> if (x=q) then  (x,y)::(get_transition q p)
  else (get_transition q p)
```

Code du TIPE

```
(* Fusionne deux ensembles de transitions : exemple
fusion [(q1,q1); (q1;q2)] [(s0,s0);(s0;s1)] renverra
[((q1,s0),(q1,s0)); ((q1,s0),(q1,s1)) ; ((q1,s0),(q2,s0)) ; ((q1,s0), (q2,s1))]*)
let rec fusion (t1: (state * state) list) (t2: (state * state) list) : ((state * state) *
↪  (state * state)) list =
  (* idem mais avec un seul état *)
  let rec fusion_bis (q: (state * state)) (t: (state * state) list) : ((state * state) *
  ↪  (state * state)) list =
    let (q1,q2) = q in
    match t with
    | [] -> []
    | (x,y)::p -> ((q1,x),(q2,y))::(fusion_bis q p)
  in
  match t1 with
  | [] -> []
  | q::p -> union (fusion_bis q t2) (fusion p t2)
```

Code du TIPE

```
(* Récupère l'ensemble des étiquettes de notre automate de buchi b *)
let get_alphabet (b:buchi_automate) : ltl list =
  (* récupère les litteraux de la forme Atom "p" d'un ensemble de formule fset *)
  let rec get_litteral (fset : ltl list) : ltl list =
    match fset with
    | [] -> []
    | (Atom p)::l | (Not (Atom p))::l -> (Atom p)::(get_litteral l)
    | _::l -> get_litteral l
  in
  let rec aux_alphabet (l: (state * (ltl list)) list) : ltl list =
    match l with
    | [] -> []
    | (q,fset)::p -> (get_litteral fset)@(aux_alphabet p)

  in aux_alphabet (b.valuation)

(* Récupère l'ensemble des formules atomiques de la forme Not (Atom "p") dans fset (On
↪ renverra une version épurée sans le Not) *)
let rec get_neg (fset: ltl list) : ltl list =
  match fset with
  | [] -> []
  | (Not (Atom p))::q -> (Atom p)::(get_neg q)
  | _::q -> get_neg q
```

Code du TIPE

```
(* Renvoie une version modifié de fset, si fset contient une formule de la forme Not (Atom
↪  "p"), alors on renvoie
alphabet\{Atom "p"}  *)
let rec modifie_ltlset (fset : ltl list) (alphabet : ltl list ) : ltl list =
    prive alphabet (get_neg fset)

(* Detecte si un ensemble de formule contient une absurdité du type [Not (Atom "p"); (Atom
↪  "p")] *)
let rec absurde (fset : ltl list) : bool =
  match fset with
  | [] -> false
  | (Atom p)::q -> if (appartient (Not (Atom p)) fset) then true
  else absurde q
  | (Not (Atom p))::q -> if (appartient (Atom p) fset) then true
  else absurde q
  (* On a ici pas besoin de "regarder en arrière" dans la liste car on la parcourt de gauche a
  ↪  droite *)
  | f::q -> absurde q
```

Code du TIPE

```
(* Réalise la fonction de transition pour l'automate produit *)
let rec prod_transition (b1: buchi_automate) (b2:buchi_automate) : ((state * state) * (state * state)) list =
  let states = prod_states (b1.q) (b2.q) in
  let res = ref [] in
  let alphabet1 = get_alphabet b1 in let alphabet2 = get_alphabet b2 in
  let alpha = union alphabet1 alphabet2 in
  List.iter (fun (q,q') ->
    (* Rappel : valuation correspond à l'étiquetage des transitions sortants d'un état, ainsi si
    deux transitions (q1,q2) et (q1',q2') correspondent en valuation_dans_autom1(q1) = valuation_dans_autom2(q2)
    alors on peut ajouter la transition ((q1,q1'),(q2,q2')) *)
    let v1 = List.assoc q (b1.valuation) in let v2 = List.assoc q' (b2.valuation) in
    (* Il y a ici un problème, lorsque la valuation étiquette (Not (Atom "p")), cela
    signifie que n'importe quelle lettre différente de p permet de prendre la transition *)
    let v1' = modifie_ltlset v1 alpha in let v2' = modifie_ltlset v2 alpha in
    let union_v1v2 = union v1 v2 in
    if ((inter v1' v2') <> [] && (not (absurde union_v1v2))) then (
      let t1 = get_transition q (b1.transition) in
      let t2 = get_transition q' (b2.transition) in
      let fus = fusion t1 t2 in
      res := union (!res) fus
    )
    else ()) states;
  !res
```

Code du TIPE

```
(* Réalise la fonction valuation pour l'automate produit *)
let rec prod_valuation (b1: buchi_automate) (b2:buchi_automate) : ((state * state) * (ltl list)) list =
  let states = prod_states (b1.q) (b2.q) in
  let res = ref [] in let alphabet1 = get_alphabet b1 in let alphabet2 = get_alphabet b2 in
  let alpha = union alphabet1 alphabet2 in
  List.iter (
    fun (q,q') ->
      let v1 = List.assoc q (b1.valuation) in
      let v2 = List.assoc q' (b2.valuation) in
      let v1' = modifie_ltlset v1 alpha in let v2' = modifie_ltlset v2 alpha in
      let union_v1v2 = union v1 v2 in
      if ((inter v1' v2') <> [] && not (absurde union_v1v2)) then (
        res := union (!res) ([((q,q'),(union_v1v2))])
      )
      else (res := union (!res) [((q,q'),[])])
  ) states;
  !res

type buchi_automate_product = {
  q : (state * state) list;
  init : (state * state) list;
  transition : ((state * state) * (state * state)) list;
  final : (state * state) list;
  valuation : ((state * state) * (ltl list)) list
}
```

Code du TIPE

```
let create_product (b1:buchi_automate) (b2:buchi_automate) : buchi_automate_product =
  {
    q = prod_states (b1.q) (b2.q);
    init = prod_init b1 b2;
    transition = prod_transition b1 b2;
    final = prod_final b1 b2;
    valuation = prod_valuation b1 b2;
  }

(* Début de l'emptiness check *)

(* Fonction donné par le doc : https://www.cs.cmu.edu/~15414/f17/lectures/18-ltl.pdf *)
(* Récupère l'ensemble des états atteints en un coup depuis q avec t(transition) *)
let rec get_state_set (q: ((state*state))) (t: ((state * state)*(state * state)) list) : (state*state) list =
  match t with
  | [] -> []
  | (x,y)::p -> if (x=q) then  y::(get_state_set q p)
  else (get_state_set q p)

exception Found
```

Code du TIPE

```
let rec innerdfs (q: (state*state)) (outervisited : (state*state) list) (innervisited : (state*state) list) (b: buchi_automate_product) : unit =
  let innervisited' = q::innervisited in
  List.iter (fun q' -> if (appartient q' outervisited) then (raise Found)
  else if (not (appartient q' innervisited' )) then (innerdfs q' outervisited innervisited' b)) (get_state_set q (b.transition))

let rec outerdfs (q:(state*state)) (visited : (state*state) list) (b: buchi_automate_product) : unit =
  let visited' = q::visited in
  List.iter (fun q' -> if (not (appartient q' visited')) then (outerdfs q' visited' b)) (get_state_set q (b.transition));
  if (appartient q (b.final)) then (innerdfs q visited' [] b)
  else ()

let is_empty (b: buchi_automate_product) : bool =
  let res = ref true in
  List.iter (fun q ->
    try
      outerdfs q [] b
    with
    | Found -> res := false
    ) (b.init);
  !res

let model_checker (k: kripke_struct) (f: ltl) : bool =
  let b2 = kripke_to_buchi k in
  let b1 = build_lgba (fnn (Not f)) in
  let product = create_product b1 b2 in
  ((is_empty product))
```

Langage jouet

```
<program> ::= "début" <listinstr> "fin"

 <listinstr> ::=
                <instr> <listinstr>
               | <instr>

 <instr> ::=
            "entier" <var> ";"
           | <var> " <- " <expr> ";"
           | "Tant que" "(" <condition> ")" "faire" <listinstr> "fin Tant
que" ";"
           | "Si" "(" <condition> ")" "faire" <listinstr> "fin Si" ";"
           | "Retourner" <expr>

 <expr> ::=
           <expr> "+" <expr>
          | <expr> "-" <expr>
          | <expr> "*" <expr>
          | "(" <expr> ")"
          | <expr0>

 <expr0> ::=
            <var>
           | <int>
           | "-" <int>


 <int> ::=
           <chiffre> <int>
          | <chiffre>

 <chiffre> ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

 <condition> ::= <expr> <compsymbol> <expr>

 <compsymbol> ::= ">" | "<" | ">=" | "<=" | "!=" | "="
```

Test de vacuité du langage d'un LGBA (Algo 1)

Algorithme 1 : outerdfs (Recherche externe de cycle)

Entrée : état q, liste *visités*, automate $A = (\Sigma, Q, I, F, \delta, L)$
Sortie : Recherche de cycle acceptant depuis q

visités$'$ $\leftarrow$ ajouter q à *visités*
pour tout $q' \in \delta(q)$ **faire**
    si $q' \notin$ *visités*$'$ **alors**
        `// Appel récursif`
        `outerdfs`(q', *visités*$'$, A)
si $q \in F$ **alors**
    `innerdfs`(q, *visités*$'$, [ ], A)

Test de vacuité du langage d'un LGBA (Algo 2)

Algorithme 2 : innerdfs (Recherche interne de cycle)

Entrée : état q, liste *visitesExt*, liste *visitesIn*, automate $A = (\Sigma, Q, I, F, \delta, L)$

Sortie : Recherche d'un cycle vers un état visité par `outerdfs`

visitesIn$'$ $\leftarrow$ ajouter q à *visitesIn*

pour tout $q' \in \delta(q)$ **faire**

  si $q' \in$ *visitesExt* **alors**

    lancer exception `Found` `// Cycle détecté`

  sinon si $q' \notin$ *visitesIn*$'$ **alors**

    appeler `innerdfs`(q', *visitesExt*, *visitesIn*$'$, A)

Test de vacuité du langage d'un LGBA (Algo 3)

Algorithme 3 : estVide (Test de vacuité)

Entrée : Automate $A = (\Sigma, Q, I, F, \delta, L)$
Sortie : Vrai si un cycle acceptant est trouvé, Faux sinon

pour tout $q_0 \in I$ **faire**
    essayer `outerdfs`$(q_0, [\ ], A)$ **avec**
        `Found` $\rightarrow$ **retourner** `Faux`
retourner `Vrai`

Pour "pousser" la négation	
Formule initiale	Formule équivalente
$\neg$ (X φ)	(X $\neg$ φ)
$\neg$ (φ U ψ)	($\neg$ φ) R ($\neg$ ψ)
$\neg$ (φ R ψ)	($\neg$ φ) U ($\neg$ ψ)

Pour supprimer F et G	
Formule initiale	Formule équivalente
F φ	$\top$ U φ
G φ	$\bot$ R φ

Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \land (X\ q))}$$

Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \wedge (X\ q))}$$

$$\neg(\mathbf{F}\ (\mathbf{p} \wedge (\mathbf{X}\ \mathbf{q}))$$

Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \wedge (X\ q))}$$

$$\neg(\mathbf{F}\ (\mathbf{p} \wedge (\mathbf{X}\ \mathbf{q}))$$

$$\downarrow$$

$$\neg(\top\ \mathbf{U}\ (\mathbf{p} \wedge (\mathbf{X}\ \mathbf{q})))$$

Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \wedge (X\ q))}$$

$$\neg(\mathbf{F}\ (p \wedge (\mathbf{X}\ q))$$

$$\downarrow$$

$$\neg(\top\ \mathbf{U}\ (p \wedge (\mathbf{X}\ q)))$$

$$\downarrow$$

$$(\neg\top\ \mathbf{R}\ \neg(p \wedge (\mathbf{X}\ q)))$$

Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \land (X\ q))}$$

$$\neg(\mathbf{F}\ (\mathbf{p} \land (\mathbf{X}\ \mathbf{q}))$$

$$\downarrow$$

$$\neg(\top\ \mathbf{U}\ (\mathbf{p} \land (\mathbf{X}\ \mathbf{q})))$$

$$\downarrow$$

$$(\neg\top\ \mathbf{R}\ \neg(\mathbf{p} \land (\mathbf{X}\ \mathbf{q})))$$

$$\swarrow$$

$$\neg\top$$

Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \wedge (X\ q))}$$

$$\begin{tikzcd}
 & \neg(\mathbf{F}\ (p \wedge (\mathbf{X}\ q)) \arrow[d] \\
 & \neg(\top\ \mathbf{U}\ (p \wedge (\mathbf{X}\ q))) \arrow[d] \\
 & (\neg\top\ \mathbf{R}\ \neg(p \wedge (\mathbf{X}\ q))) \arrow[dl] \\
\neg\top \arrow[d] & \\
\bot &
\end{tikzcd}$$

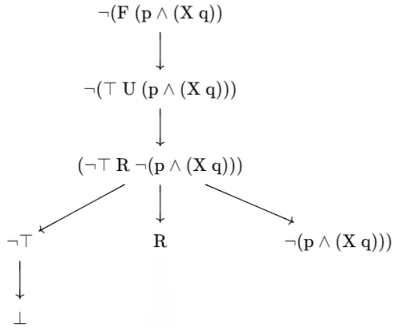
Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \wedge (X\ q))}$$

$$\begin{tikzcd}[column sep=small]
 & \neg(F\ (p \wedge (X\ q))) \arrow[d] \\
 & \neg(\top\ U\ (p \wedge (X\ q))) \arrow[d] \\
 & (\neg\top\ R\ \neg(p \wedge (X\ q))) \arrow[dl] \arrow[d] \\
\neg\top \arrow[d] & R \\
\bot &
\end{tikzcd}$$

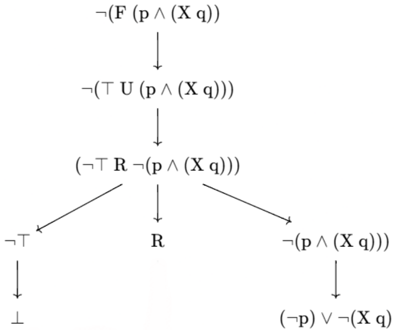
Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \wedge (X\ q))}$$



Exemple d'une transformation vers FNN

$$\boxed{f := \mathrm{F}\ (p \wedge (\mathrm{X}\ q))}$$



Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \land (X\ q))}$$

$$\begin{tikzcd}
 & \neg(F\ (p \land (X\ q)) \arrow[d] & \\
 & \neg(\top\ U\ (p \land (X\ q))) \arrow[d] & \\
 & (\neg\top\ R\ \neg(p \land (X\ q))) \arrow[dl] \arrow[d] \arrow[dr] & \\
\neg\top \arrow[d] & R & \neg(p \land (X\ q))) \arrow[d] \\
\bot & & (\neg p) \lor \neg(X\ q) \arrow[d] \\
 & & (\neg p) \lor X\ (\neg q)
\end{tikzcd}$$

Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \land (X\ q))}$$

$$\begin{tikzcd}
 & \neg(\mathrm{F}\ (\mathrm{p} \land (\mathrm{X}\ \mathrm{q})) \arrow[d] & \\
 & \neg(\top\ \mathrm{U}\ (\mathrm{p} \land (\mathrm{X}\ \mathrm{q}))) \arrow[d] & \\
 & (\neg\top\ \mathrm{R}\ \neg(\mathrm{p} \land (\mathrm{X}\ \mathrm{q}))) \arrow[dl] \arrow[d] \arrow[dr] & \\
\neg\top \arrow[d] & \mathrm{R} \arrow[ddd] & \neg(\mathrm{p} \land (\mathrm{X}\ \mathrm{q}))) \arrow[d] \\
\bot \arrow[ddr] & & (\neg\mathrm{p}) \lor \neg(\mathrm{X}\ \mathrm{q}) \arrow[d] \\
 & & (\neg\mathrm{p}) \lor \mathrm{X}\ (\neg\mathrm{q}) \arrow[dl] \\
 & \bot\ \mathrm{R}\ ((\neg\mathrm{p}) \lor \mathrm{X}\ (\neg\mathrm{q})) &
\end{tikzcd}$$

Exemple d'une transformation vers FNN

$$\boxed{f := F\ (p \wedge (X\ q))}$$

$$\boxed{FNN\ (\neg f) = \perp R\ (\neg p \vee (X\ \neg q))}$$

Ensembles en ocaml

$A \backslash B$

$A \cup B$

$A \cap B$

$a \in A$

$A \subseteq B$

$A = B$

```
prive : 'a list -> 'a list -> 'a list
union : 'a list -> 'a list -> 'a list
inter : 'a list -> 'a list -> 'a list
appartient : 'a -> 'a list -> bool
inclus : 'a list -> 'a list -> bool
exception Elem_pas_present
suppr : 'a -> 'a list -> 'a list
egal_ens : 'a list -> 'a list -> bool
```