Validation par analyse statique

Xavier Thirioux

ENSEEIHT

November 27, 2018

Le Model-Checking

- Vérification Automatique de Modèles:
- Qu’est-ce qu’un modèle ?
- Qu’est-ce qu’on vérifie ?
- Comment le vérifie-t’on ?

Termes du problème

formalisme opérationnel: système

- système de transitions
- automate (simple, à pile, ...)
- machine de Türing
- programme C

$\rightarrow$ structure de Kripke

formalisme logique: spécification

- (non-) atteignabilité
- observateurs
- LTL, CTL
- CTL*, μ-calcul
- (bi-)simulation

Structure de Kripke

Une structure de Kripke $\mathcal{S}$ est un quintuplet:

- S est l'espace d'états
- I $\subseteq$ S est l'ens. des états initiaux
- R $\subseteq$ S $\times$ S est la rel. de transition
- Prop est l'ens. des propositions atomiques
- L $\in$ S $\to$ 2^{Prop} est la fonction qui associe à chaque état les propriétés élémentaires qu'il vérifie

Approche sémantique

- pas de manipulation de formules (syntaxique)
- domaine sémantique (modèle) pour les exécutions:
  $\rightarrow$ états, traces (finies, infinies), arbres de calcul
- un système $\mathcal{S}$ représente un ens. de modèles $[\![\mathcal{S}]\!]$
- une spécification F représente un ens. de modèles $[\![\mathrm{F}]\!]$
- par un parcours de l'espace des états (co-) accessibles ...
- on vérifie que les exécutions de $\mathcal{S}$ sont permises par F:

$$[\![\mathcal{S}]\!] \subseteq [\![\mathrm{F}]\!]$$

Une nécessité ?

- 1981: bug dans la première navette spatiale (actuellement: 50M de lignes)
- 1989: panne géante du réseau téléphonique aux USA
- 1994: bug du pentium III
- 1996: explosion première Ariane 5
- 1997: perte de données Mars Pathfinder
- 1997: bug protocole audio/video Bang & Olufsen
- 2000: bug pile TCP/IP (OpenBsd, Windows)
- 2004: bug dans régulateur de vitesse Renault ?

Une nécessité !

- pour les systèmes embarqués / critiques, il y a mise en jeu:

→ de vies humaines, d'argent, d'informations rares

- certains bugs avaient été détectés mais impossibles à corriger !
- certains bugs n'avaient pas été encore découverts !

quelques outils

- JAVA PATHFINDER à la NASA
- SCADE / NP-PROVER chez Airbus
- UPPAAL chez Bang & Olufsen
- SLAM chez Microsoft
- SMV chez Cadence, Intel
- SPIN chez Nortel Network

Contraintes

l'algorithme de vérification doit être:

- (semi-) décidable
- raisonnable en mémoire
- raisonnable en temps
- bavard (suivi du processus de preuve)
- compréhensible (exploitation des résultats)

Problème de composition

- la composition (synchrone ou asynchrone) de systèmes est exponentielle:
  $\rightarrow$ N systèmes de taille $M \equiv$ un système de taille M^N !
- le model-checking n'est pas compositionnel:
  $\rightarrow$ on ne peut pas décomposer les preuves structurellement
- il faut valuer / expanser tous les paramètres symboliques
  $\rightarrow +$ facile quelquefois de vérifier pour N que pour 50 !

Problème de complexité

- la complexité des algorithmes de vérification varie entre polynomial / exponentiel
- en temps / mémoire
- une échelle de grandeur:
  500 bits $\sim 3 * 10^{150}$ états $\sim$ 62 octets $\implies 1 - 10$Go
- temps / mémoire très sensibles à la modélisation et aux algorithmes choisis:
  $\rightarrow$ abstraction, fusion, dépliage
  $\rightarrow$ ordre des variables
  $\rightarrow$ ordre du parcours

Solutions ?

- exploitation optimale des symétries du système
- représentations concises des ens. d'états
- décomposition des preuves manuellement
- conditions suffisantes (syntaxiques) de décomposition
- détermination des valeurs significatives des paramètres symboliques
  $\rightarrow$ par ex., si la spec est vérifiée pour $N = 5$, alors elle l'est aussi pour tout $N > 5$.

Explicite vs. symbolique

- explicite: un état est un élément d'un conteneur
- symbolique: représentation compacte d'ens. d'états
- en théorie: pas de gagnant
- en pratique: si l'espace d'états est grand, on préfère le symbolique

Explicite

- outil SPIN (logique LTL)
- adapté au parcours en profondeur (reconstruction des exécutions)
- peut-être très économe en mémoire / très gourmand en temps
- en général, table de hachage
- abstraction par "bit-state encoding"

Symbolique

- outil SMV (logique CTL)
- adapté au parcours en largeur (en couches d'oignon)
- un peu moins économe en mémoire / un peu moins gourmand en temps
- meilleur compromis en général
- se prête parfois à la représentation d'ens. infinis d'états
- un exemple célèbre: Binary Decision Diagrams (BDD)

Binary Decision Diagrams

Avantages

- travaux de Lee (1959), Bryant (1986), Coudert & Madre (1992), etc
  $\rightarrow$ technologie très bien maîtrisée
- utilisé dans beaucoup d'outils, dont SMV
- finalement, tous les programmes manipulent des bits

Inconvénients

- il faut tout transformer en programme booléen (circuit)
  $\rightarrow$ nombre de variables (et taille) parfois rédhibitoire
- opérations arithmétiques difficiles
- très sensible à l'encodage choisi
- calcul des transitions exponentiel

Définition

Un diagramme de décision binaire est:

- un arbre de décision binaire
- représentant une formule de logique propositionnelle
- où les variables sont toujours dans le même ordre
- où les sous-arbres communs sont partagés
- où les variables inutiles sont supprimées
- où une expression et sa négation sont partagées

Théorème de décomposition de Shannon

$$\forall f \in \mathrm{Bool}^n \to \mathrm{Bool} : \exists! f_0, f_1 \in \mathrm{Bool}^{n-1} \to \mathrm{Bool} :$$
$$f(x_1, \ldots, x_n) = (\neg x_1 \wedge f_0(x_2, \ldots, x_n)) \vee (x_1 \wedge f_1(x_2, \ldots, x_n))$$

Décomposition de Shannon

Pour toutes fonctions f et g, et toute var. de décomposition x:

- $f_0 = f[x \leftarrow \mathit{False}]$; $f_1 = f[x \leftarrow \mathit{True}]$
- $(\neg f)_0 = \neg(f_0)$; $(\neg f)_1 = \neg(f_1)$
- $(f \vee g)_0 = f_0 \vee g_0$; $(f \vee g)_1 = f_1 \vee g_1$
- $\exists x.f = f_0 \vee f_1$

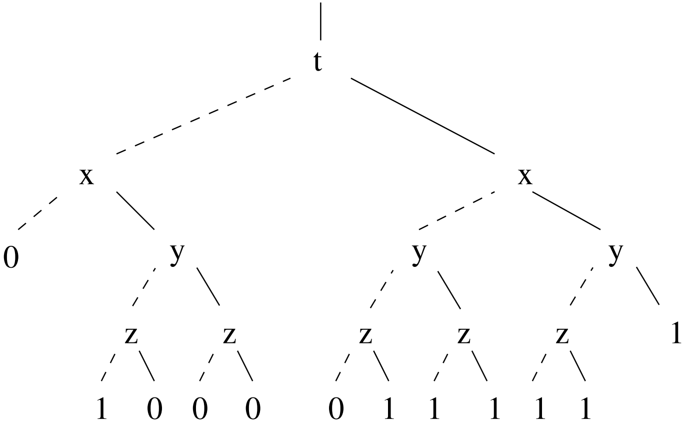
Démonstration !

Un exemple

- un arbre de décision binaire représentant:
  $f = (x \land \neg(y \lor z)) \lor (t \land y) \lor (z \land t)$
- on choisit (arbitrairement) l'ordre: $t < x < y < z$
- par rapport à t, on a:
  $f = (\neg t \land \underbrace{(x \land \neg(y \lor z))}_{f_0 = f[t \leftarrow False]}) \lor (t \land \underbrace{((x \land \neg(y \lor z)) \lor y \lor z)}_{f_1 = f[t \leftarrow True]})$
- on décompose récursivement f_0 et f_1
  en fonction de x, puis y, puis z
- donc:
  $f_0 = (\neg x \land \underbrace{False}_{f_{00}}) \lor (x \land \underbrace{\neg(y \lor z)}_{f_{01}})$
  $f_1 = (\neg x \land \underbrace{(y \lor z)}_{f_{10}}) \lor (x \land \underbrace{(\neg(y \lor z) \lor y \lor z)}_{f_{11}})$

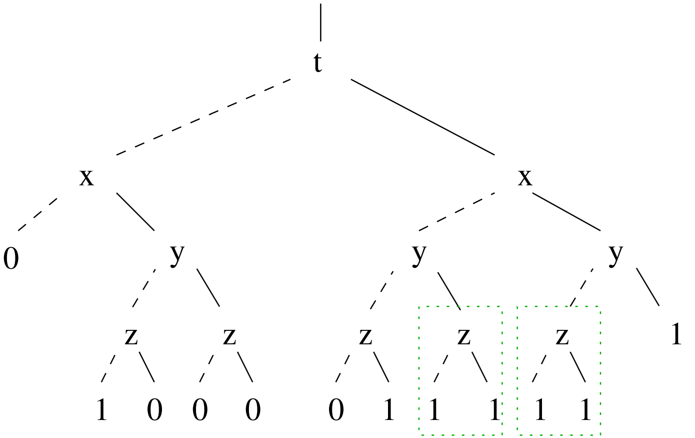
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Son arbre de décision binaire est (8 états sur 16):



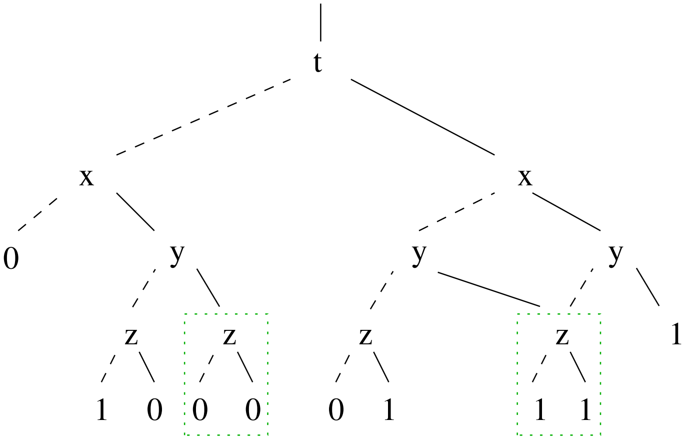
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Partage des sous-expressions en commun:



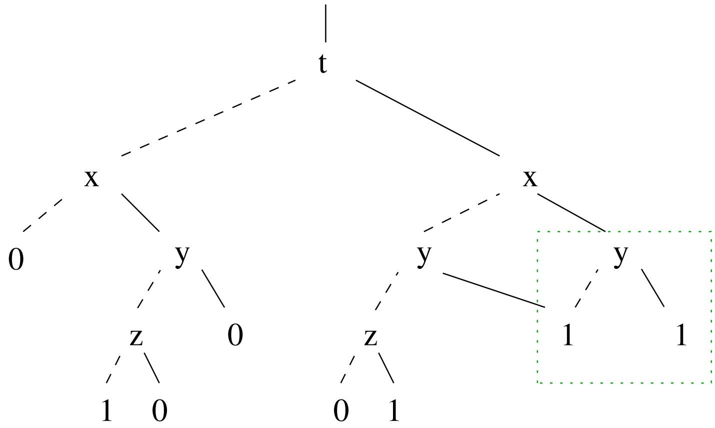
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Elimination des variables inutiles ($f_0 = f_1$):



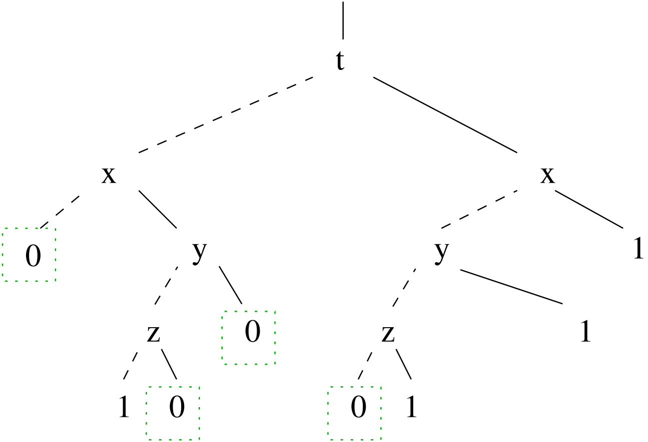
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Elimination des variables inutiles ($f_0 = f_1$):



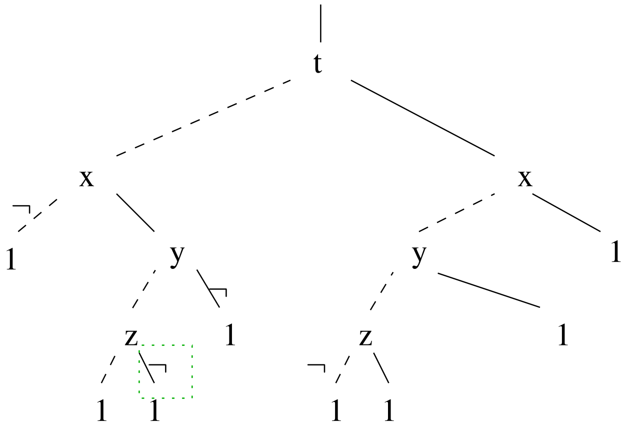
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Partage des négations (intro. bit de négation sur les branches):



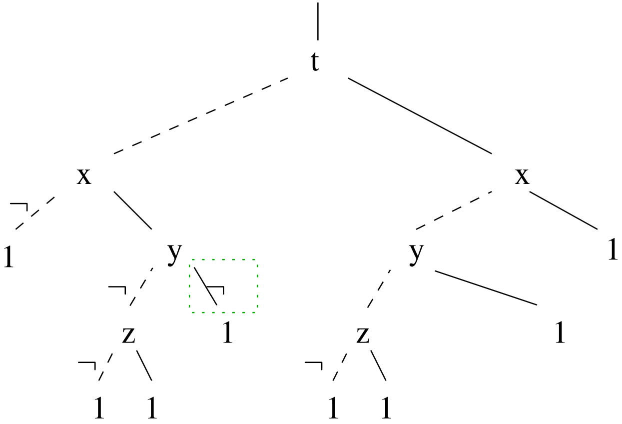
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Normalisation des négations (pas $\neg$ dans branches droites):



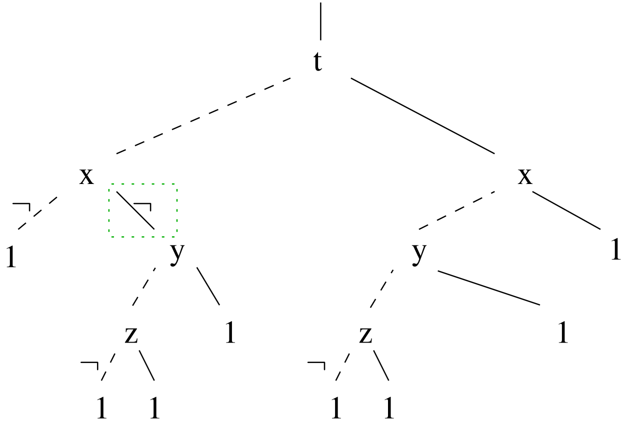
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Normalisation des négations (pas $\neg$ dans branches droites):



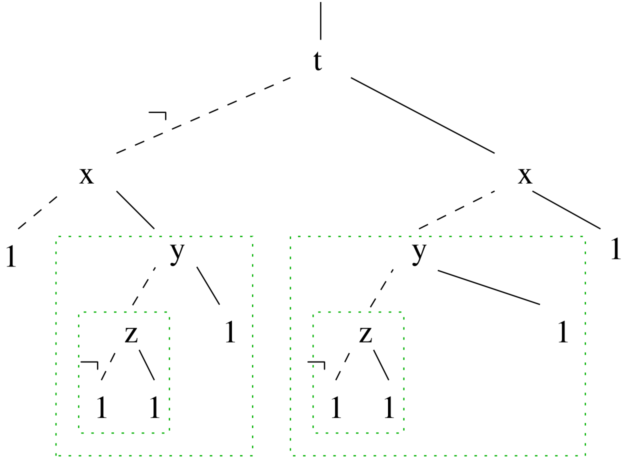
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Normalisation des négations (pas $\neg$ dans branches droites):



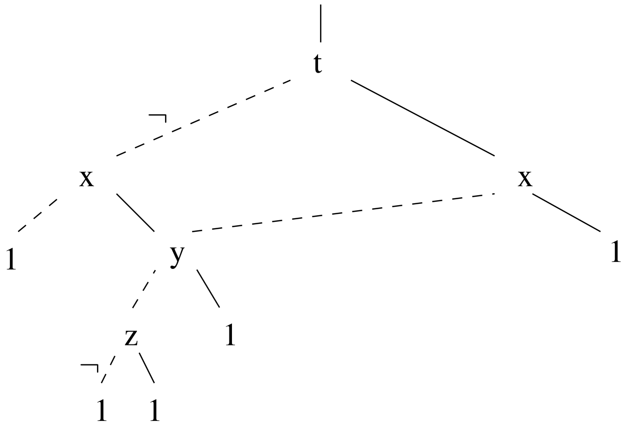
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Partage des sous-expressions en commun:



Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Diagramme final (retrouvez les 8 états !):



Opérations booléennes

- la négation est triviale: on change un bit à la racine !
- la proposition atomique est triviale
- la quantification existentielle: $\exists x.f = f_0 \vee f_1$
- la conjonction = disjonction + négation

$\rightarrow$ reste la disjonction $\vee$

- de complexité polynomiale !

Disjonction

Soient f et g suivantes. On cherche à calculer $f \vee g$:

- $f = (\neg x \wedge f_0) \vee (x \wedge f_1)$
- $g = (\neg y \wedge g_0) \vee (y \wedge g_1)$
- on peut avoir $x \neq y$

Cas $x = y$

On a: $f \vee g = (\neg x \wedge (f_0 \vee g_0)) \vee (x \wedge (f_1 \vee g_1))$

Cas $x < y$

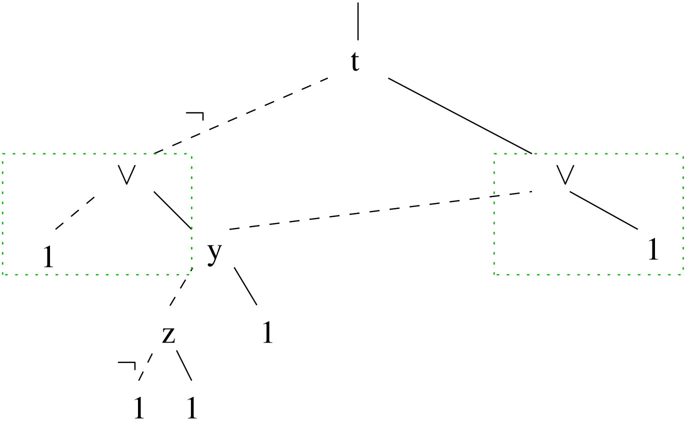
On a toujours: $g = (\neg x \wedge g) \vee (x \wedge g)$
On se ramène alors au premier cas

Cas $x > y$

On a toujours: $f = (\neg y \wedge f) \vee (y \wedge f)$
On se ramène alors au premier cas

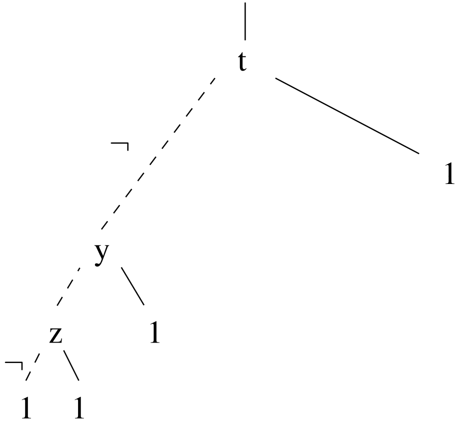
Formule $f = (x \wedge \neg(y \vee z)) \vee (t \wedge y) \vee (z \wedge t)$

Quantification existentielle: $\exists x : f$



Formule $\exists x : f$

Diagramme final: $t \vee (\neg t \wedge \neg y \wedge \neg z)$



Calcul des transitions

Hypothèses

- mise en forme (p registres, n entrées, m sorties):

$$\begin{cases} r'_i & = f_i(e_1, \ldots, e_n, r_1, \ldots, r_p) \quad \text{pour } i \in [1, p] \\ s_i & = g_i(e_1, \ldots, e_n, r_1, \ldots, r_p) \quad \text{pour } i \in [1, m] \end{cases}$$

- les variables sont les e_i et r_i, les BDD sont les f_i et g_i

Calcul

- à l'instant t: on a calculé l'ens. d'états a^t (variables r_i)
- à l'instant $t + 1$:

$$\begin{cases} a^{t+1} & = (\underbrace{\exists e_1, \ldots, e_n, r_1, \ldots, r_p}_{\text{exponentiel en } p,\, n} : a^t \bigwedge_{i \in [1,p]} (r'_i \Leftrightarrow f_i))[r'_i \leftarrow r_i] \\ s_i^{t+1} & = g_i \wedge a^{t+1} \end{cases}$$

Problème: ordre des variables

L'ordre des variables peut faire varier la taille des BDD.
Soient les variables $x_i, i \in [0, 2n-1]$:

Un cas linéaire

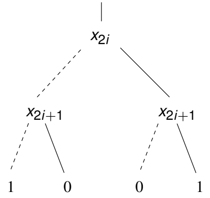
$$\bigwedge_{i \in [0,n-1]} (x_{2i} \Leftrightarrow x_{2i+1})$$

Un cas exponentiel

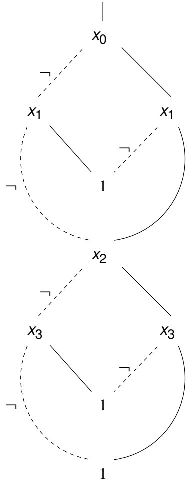
$$\bigwedge_{i \in [0,n-1]} (x_i \Leftrightarrow x_{n+i})$$

Cas linéaire

Arbre de $x_{2i} \Leftrightarrow x_{2i+1}$:

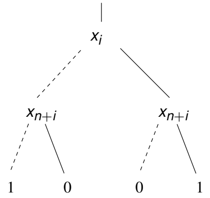


BDD pour $\bigwedge_{i \in [0,1]} (x_{2i} \Leftrightarrow x_{2i+1})$:

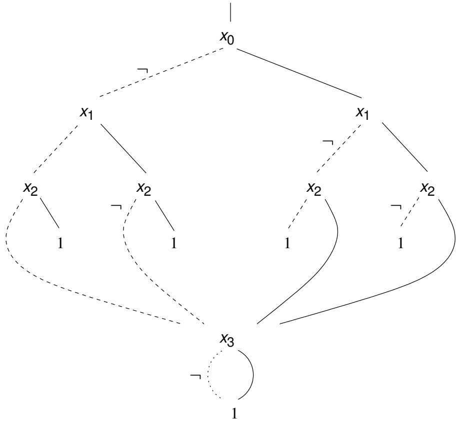


Cas exponentiel

Arbre de $x_i \Leftrightarrow x_{n+i}$:



BDD pour $\bigwedge_{i \in [0,1]} (x_i \Leftrightarrow x_{n+i})$:



Problème: ordre du parcours

L'ordre du parcours peut faire varier la taille des BDD.

- avoir une représentation dense:
  beaucoup d'états, peu de noeuds
- lorsqu'on accumule les états atteints
  par ex. calcul des états accessibles

solution 1

- il faut ordonner / partitionner les calculs d'image afin que les encodages des états sources et images soient "proches"
  par ex. au sens de la distance de Hamming
- utilisation du code de Gray

Problème: ordre du parcours

L'ordre du parcours peut faire varier la taille des BDD.

- avoir une représentation dense:
  beaucoup d'états, peu de noeuds
- lorsqu'on accumule les états atteints
  par ex. calcul des états accessibles

solution 2

- changer dynamiquement l'ordre des variables
- permutations de variables voisines (par ex. recuit simulé)

$\rightarrow$ algorithmiquement non trivial et grande complexité $O(n! \times 2^n)$

Problème: ordre du parcours

L'ordre du parcours peut faire varier la taille des BDD.

- avoir une représentation dense:
  beaucoup d'états, peu de noeuds
- lorsqu'on accumule les états atteints
  par ex. calcul des états accessibles

solution 3

- déterminer / fournir des invariants simples ("careset")
  sur les entrées et les registres
  statiquement ou bien à la volée
- par ex. x_i, $\neg x_i$, $x_i \Leftrightarrow x_j$, $x_i \Leftrightarrow \neg x_j$

$\rightarrow$ nouveaux opérateurs pour factoriser ces invariants
et diminuer ainsi la taille des BDD

Opérateurs de factorisation

- $factor(f, g) \wedge g \Leftrightarrow f \wedge g$
- $taille(factor(f, g)) < taille(f)$
- commute avec la décomposition de shannon (structurel)

restriction

- $f \wedge g \Rightarrow restrict(f, g) \Rightarrow f \vee \neg g$
- si $f \wedge g = False$, alors $restrict(f, g) = False$
- si $f \vee \neg g = True$, alors $restrict(f, g) = True$

$\rightarrow$ toujours + petit que f

cofacteur généralisé

- utilisation d'une projection de Hamming :
  $Proj(a, b) = \{x \in b \mid x = min_y \|y - z\|, z \in a\}$
- $constrain(f, g) \wedge \neg g = f \wedge Proj(\neg g, g)$

$\rightarrow$ pas toujours + petit, mais moins dépendant de f

Secrets d'implantation

le partage après coup est maladroit !

1. on stocke tous les noeuds créés (éviter les doublons) dans une table de hachage:
   $hash_f = Hachage(x, hash_{f_0}, hash_{f_1})$
2. on crée un noeud ssi il n'est pas déjà dans la table

$\rightarrow$ il faut une énorme table (ou bien accepter des doublons) !

comment faire pour allouer / libérer la mémoire ?

sans parcours récursif des pointeurs ou de la table !

1. par ex., un compteur de réf. par noeud (BDD acycliques)
2. un noeud `n` résultat d'une opération: `n.cpt++`
3. un noeud `n` libéré: `n.cpt--`
4. si `n.cpt` $= 0$, alors on peut désallouer `n` (`free(n)`) et libérer ses fils

test de SATisfiabilité

- outil qui teste la satisfiabilité d'une formule propositionnelle
- cherche une valuation des variables qui rend la formule vraie

Avantages / BDD

- pas de représentation de tous les modèles
- heuristiques de recherche de solution
  $\rightarrow$ test de satisfiabilité $+$ rapide

Inconvénients / BDD

- pas de représentation d'ensembles d'états
- pas de calcul d'images de transitions
  $\rightarrow$ pas de représentation de tous les modèles/solutions

Définition

Un observateur $\mathcal{O}$ d'un système $\mathcal{S}$ est:

- un automate / circuit / programme qui observe et mémorise les états et entrées / sorties de $\mathcal{S}$
- $\mathcal{O}$ n'interagit pas avec $\mathcal{S}$
- $\mathcal{O}$ possède un état "erreur", qu'il faut éviter d'atteindre
- $\mathcal{O}$ est composé de façon synchrone avec $\mathcal{S}$
- $\mathcal{O}$ reconnaît un sous-ens. régulier des exécutions de $\mathcal{S}$
- $\mathcal{O}$ peut représenter toutes les propriétés de sûreté de $\mathcal{S}$
- $\mathcal{O}$ est souvent plus simple qu'une formule temporelle équivalente

Applications

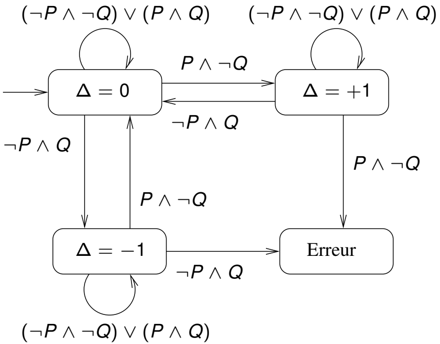
- très utilisé dans l'industrie (par défaut quelquefois)
- à coupler avec un algorithme d'exploration de l'espace d'états de $\mathcal{S}$:
  $\rightarrow$ parcours intelligent des exec. permises par $\mathcal{O}$ (cycles)
  $\rightarrow$ arrêt dès qu'une erreur est trouvée
- hardware: circuit $\mathcal{O}$ en parallèle avec $\mathcal{S}$, sur la même horloge
- software: plus compliqué !
  - on n'instrumente pas le code source (trop lourd)
  - on ne compose pas avec des threads (synchro. impossible)
  - on définit un interprète paramétré par $\mathcal{O}$
    qui vérifie en exécutant $\mathcal{S}$

Un exemple

- nbr. d'occ. de P = nbr. d'occ de Q à ± 1 près
- formule LTL équivalente:

$$\begin{aligned} &\Box((P \wedge \neg Q) \Rightarrow \mathrm{X}((\neg P \vee Q)\mathrm{W}(\neg P \wedge Q))) \\ \wedge &\Box((Q \wedge \neg P) \Rightarrow \mathrm{X}((\neg Q \vee P)\mathrm{W}(\neg Q \wedge P))) \end{aligned}$$

- observateur:



- Quelle que soit la logique, on se ramène à un parcours de l'espace d'états.
- Il y a des degrés de liberté énormes:
  1. sens du parcours (successeurs ou prédécesseurs) ;
  2. ordre du parcours (largeur, profondeur, autre) ;
  3. traitement de l'équité éventuelle (séquentiel, alterné) ;
  4. représentations mémoires.
- Nous étudierons seulement une technique pour LTL

Rappels de logique LTL

- LTL = Logique Temporelle Linéaire (comme en TLA)
- Les modèles de LTL sont les exécutions σ (en g^al infinies)
- On vérifie alors si: $\sigma \models ? F$
- Grammaire des formules F:

$$
\begin{array}{rcl}
F ::= & & \text{Proposition} \\
& | & F_1 \vee F_2 \\
& | & F_1 \wedge F_2 \\
& | & \neg F \\
& | & \mathrm{X}\, F \\
& | & F_1 \,\mathrm{U}\, F_2 \\
& | & F_1 \,\mathrm{W}\, F_2
\end{array}
$$

D'une formule à un observateur

Toute formule LTL se transforme en circuit (ou automate) t.q. :

la formule est un observateur

- les propositions atomiques correspondent à des entrées (transitions) de ce circuit ;
- des nouvelles entrées Oracle correspond aux disjonctions: $F_1 \vee F_2$, $F_1 \mathrm{U} F_2 \equiv F_2 \vee \mathrm{X}(F_1 \wedge F_1 \mathrm{U} F_2)$;
- une nouvelle entrée Active correspond aux instants d'activation ;
- la sortie Bug correspond à une violation de la spec. ;

la formule induit des contraintes d'équité

- les nouvelles entrées des opérateurs U doivent être vraies infiniment souvent (FairOracle).

C'est un automate de Büchi (généralisé) !

Automate de Büchi généralisé

Un automate de Büchi généralisé est défini par:

- un ens. d'états fini Q
- un ens. d'états initiaux $\mathrm{I} \subseteq \mathrm{Q}$
- un ens. d'étiquettes E
- une rel. de transition $\mathrm{R} \subseteq \mathrm{Q} \times \mathrm{E} \times \mathrm{Q}$
- une contrainte d'équité $\mathrm{F} = \{\mathrm{F}_1, \ldots, \mathrm{F}_\mathrm{n}\}$ avec $\mathrm{F}_\mathrm{i} \subseteq \mathrm{Q}$

Il reconnaît / engendre les exec. infinies $q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \ldots$ t.q. :

- $q_0 \in \mathrm{I}$
- $\forall i \in \mathbb{N} : q_i \xrightarrow{l_i} q_{i+1} \in \mathrm{R}$
- $\forall i \in [1..n] : \exists^\infty j : q_j \in F_i$

Génération du circuit

- Complexité: un registre par opérateur temporel:

$\rightarrow$ espace d'états potentiellement exponentiel !!

- Avantage: parcours récursif simple (et rapide) de formule.

$\rightarrow$ linéaire en temps !!

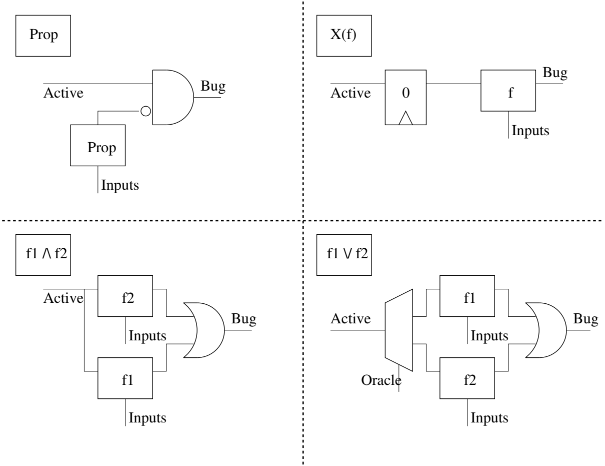
conséquence

- nécessité d'un compromis temps / mémoire
- génération du circuit améliorée
- simplification du circuit

$\rightarrow$ combinatoire (refactorisation, partage)

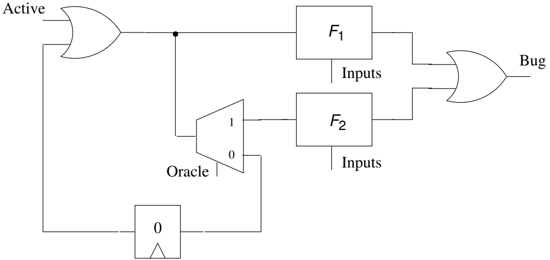
$\rightarrow$ séquentiel (retiming)

Opérateurs simples



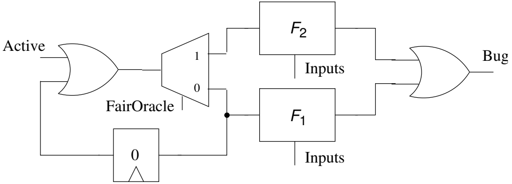
Weak Until

- $F_1 W F_2 \equiv F_1 \wedge (F_2 \vee X(F_1 W F_2))$
- un registre pour W
- un démux et un oracle pour $\vee$
- traduction en circuit:



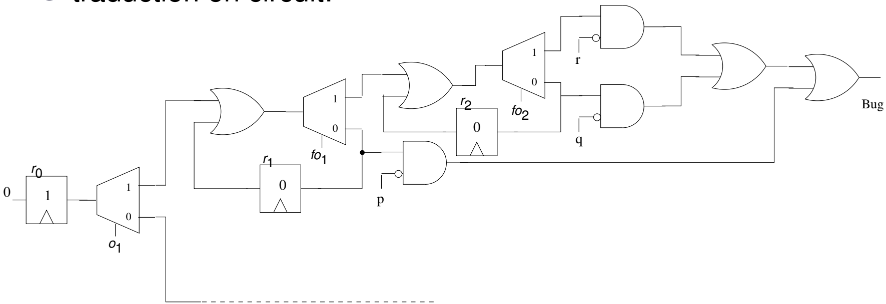
Until

- $F_1 \mathrm{U} F_2 \;\equiv\; F_2 \vee (F_1 \wedge \mathrm{X}(F_1 \mathrm{U} F_2))$
- un registre pour U
- un démux et un oracle pour $\vee$
- l'oracle est équitable !
- traduction en circuit:



Un exemple: $pU(qUr) \ \vee \ qU(rUp)$

- p, q et r sont des propositions atomiques.
- traduction en circuit:

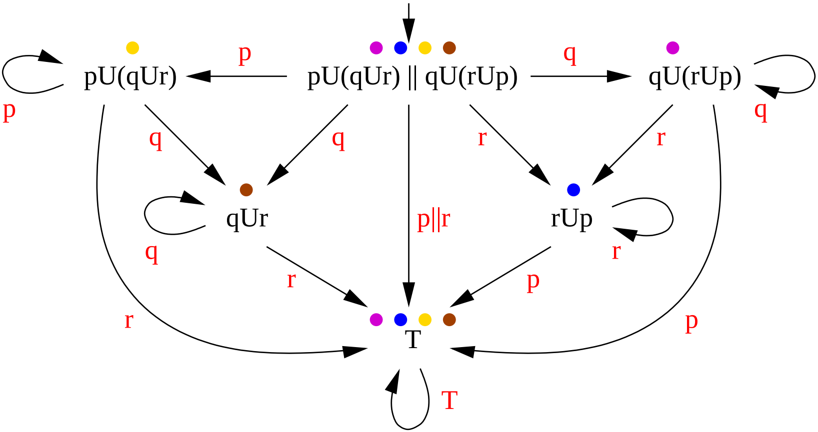


- correspondance registre $\rightarrow$ sous-formule:

$$\begin{cases} r_0 & \rightarrow & pU(qUr) \ \vee \ qU(rUp) \\ r_1 & \rightarrow & pU(qUr) \\ r_2 & \rightarrow & qUr \end{cases}$$

Un exemple

Représentation explicite:



Algorithme

1. on cherche une exécution qui invalide F: $\neg F$
2. on ramène les négations aux propositions atomiques:
   $\neg(F_1 \mathsf{U} F_2) \equiv \neg F_2 \mathsf{W}(\neg F_2 \wedge \neg F_1)$
3. on simplifie la formule si possible !
4. on crée un circuit $\mathcal{C}_{\neg F}$ qui reconnaît $[\![\neg F]\!]$.
5. on parcourt l'espace d'états de: $\mathcal{S} \times \mathcal{C}_{\neg F}$.
6. si on trouve une exécution σ telle que:
   $\sigma \models \Box\Diamond \mathit{FairOracle} \wedge \Box\neg \mathit{Bug}$
   alors la spécification F est violée.

Algorithme (version BDD)

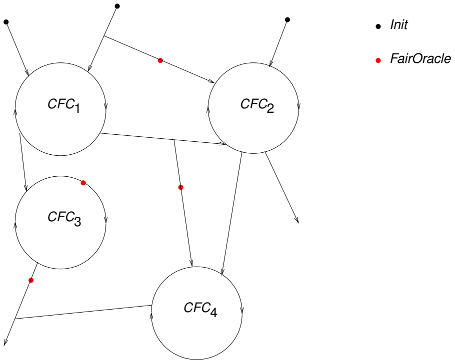
Pour trouver une exécution σ telle que:
$\sigma \models \Box\Diamond \mathit{FairOracle} \wedge \Box\neg \mathit{Bug}$
alors la spécification F est violée.

1. on calcule les états accessibles vérifiant $\neg \mathit{Bug}$:
   $\rightarrow$ plus petit point fixe de la rel. de trans. à partir des états initiaux: $\mathit{RSS} = \mu X.\neg \mathit{Bug} \wedge (\mathit{Init} \vee \mathit{Trans}(X))$
2. dans ces états, on cherche les cycles contenant au moins une fois *FairOracle*:
   $\rightarrow$ plus grand point fixe: $\mathit{Cycles} = \nu X.\mathit{RSS} \wedge \mathit{Trans}(X)$
   $\rightarrow$ on calcule en fait les cycles et leurs successeurs
3. on recommence à l'étape 1 en prenant $\mathit{Cycles} \wedge \mathit{FairOracle}$ comme nouveaux états "initiaux"
4. jusqu'à ce que *RSS* (et *Cycles*) soient inchangés
5. si $\mathit{Cycles} \neq \emptyset$, alors F est violée

Un exemple

Etats accessibles ($\mathcal{S} \times \mathcal{C}_{\neg F}$):

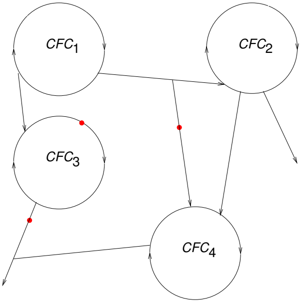
$$\begin{aligned} Init_0 &= Init \\ RSS_0 &= \mu X. \neg Bug \wedge (Init_0 \vee Trans(X)) \end{aligned}$$



Un exemple

Cycles et successeurs:

$$Cycles_0 = \nu X.RSS_0 \wedge Trans(X)$$



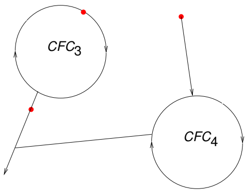
- *Init*
- *FairOracle*

Un exemple

Etats accessibles:

$$
\begin{aligned}
Init_1 &= Cycles_0 \land FairOracle \\
RSS_1 &= \mu X.\neg Bug \land (Init_1 \lor Trans(X))
\end{aligned}
$$

- *Init*
- *FairOracle*

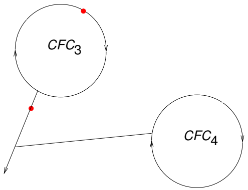


Un exemple

Cycles et successeurs:

$$Cycles_1 = \nu X.RSS_1 \wedge Trans(X)$$

- *Init*
- *FairOracle*

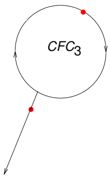


Un exemple

Etats accessibles:

$$
\begin{aligned}
Init_2 &= Cycles_1 \wedge FairOracle \\
RSS_2 &= \mu X. \neg Bug \wedge (Init_2 \vee Trans(X))
\end{aligned}
$$

- *Init*
- *FairOracle*

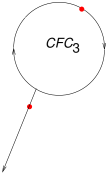


Un exemple

Cycles et successeurs:

$$Cycles_2 = \nu X.RSS_2 \wedge Trans(X)$$

- *Init*
- *FairOracle*

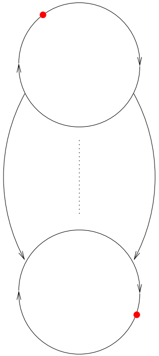


Un exemple

- le point fixe global est atteint,i.e.: $RSS_2 = Cycles_2$
- $Cycles_2 \neq \emptyset$, donc il existe une exécution σ t.q. : $\sigma \models \Box\Diamond FairOracle \wedge \Box\neg Bug$
- donc la spécification est ici fausse
- il reste à reconstruire un contre-exemple σ. Difficile avec des BDD

Un exemple

- on peut aussi combiner calculs d'images directes et inverses
- mais on ne peut jamais obtenir les états cycles seuls
- par ex.



Objectif:

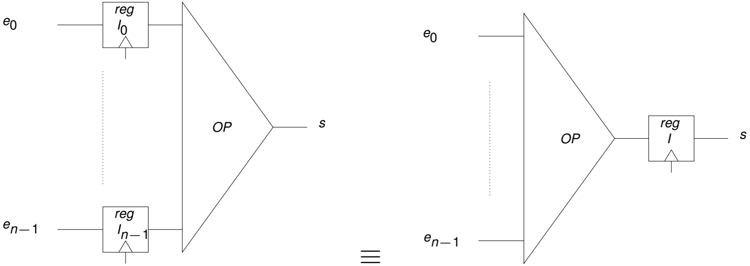
- réduire le nombre d'états / transitions à explorer
- pouvoir vérifier des systèmes plus gros
- sans abstraction, sans perte de précision
- relativement à la spécification
- bisimulation, conservation des propriétés

Introduction

- technique plutôt spécifique aux circuits
- déplacement / fusion de registres
- conserve le comportement du circuit de départ
- bisimulation
- objectif: diminuer le nombre de registres
- dans l'espace de tous les circuits équivalents, chercher l'optimum
- problème de graphe NP-complet
- heuristiques de recherche: recuit simulé, etc

Définition

Les registres *reg* commutent avec la logique combinatoire *OP*



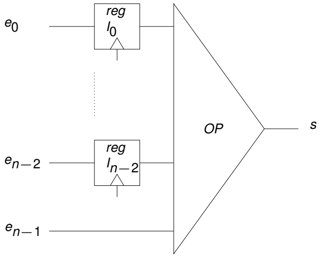
Gain de place !

Problématique

- les registres peuvent traverser la logique combinatoire an avant ou en arrière
- il faut parfois accepter un accroissement temporaire du nombre de registres
- technique trop limitée en général !
- on peut faire mieux: anti-registres

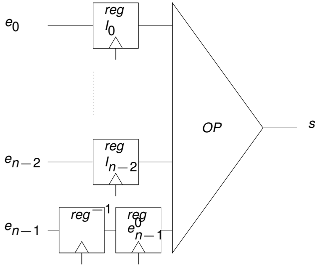
Problématique

- circuit presque bon pour le retiming
- on suppose l'ex. d'anti-registres



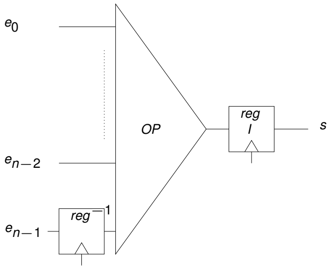
Problématique

- les anti-registres sont inverses des registres



Problématique

- un anti-registre en plus ?
- beaucoup de registres en moins !
- on doit avoir : $I = OP(I_0, \ldots, I_{n-2}, e^0_{n-1})$



Anti-registres

- un registre : $s = reg(e, X)$

e	e^0	e^1	e^2	e^3	$\ldots$
s	X	e^0	e^1	e^2	$\ldots$

- un anti-registre : $s = reg^{-1}(e)$

e	e^0	e^1	e^2	e^3	$\ldots$
s	e^1	e^2	e^3	e^4	$\ldots$

Anti-registres

Quelques propriétés :

- registre $\circ$ anti-registre : $s = reg(reg^{-1}(e), X)$

e	e^0	e^1	e^2	e^3	$\dots$
reg^{-1}	e^1	e^2	e^3	e^4	$\dots$
s	X	e^1	e^2	e^3	$\dots$

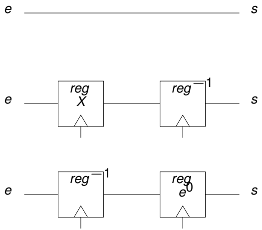
- anti-registre $\circ$ registre : $s = reg^{-1}(reg(e, X))$

e	e^0	e^1	e^2	e^3	$\dots$
reg	X	e^0	e^1	e^2	$\dots$
s	e^0	e^1	e^2	e^3	$\dots$

- les anti-registres commutent aussi avec la logique combinatoire !

Anti-registres

Graphiquement parlant:

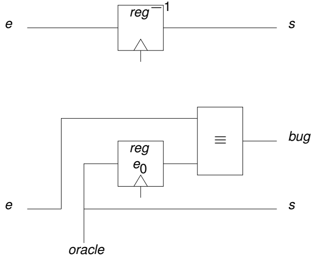


Anti-registres

- la sortie d'un anti-registre est le futur de son entrée !
  $\rightarrow$ peut-on implanter un anti-registre ?
- non, de façon déterministe (i.e. algorithmique, matérielle)
- oui, de façon non-déterministe (pour la vérification)
- l'entrée d'un anti-registre doit être le passé de sa sortie
  $s = reg^{-1}(e) \;\equiv\; e = reg(s, e^0)$
  $\rightarrow$ contrainte implantable (avec des registres) !

Anti-registres

- implantation: *oracle* est une entrée, *bug* une sortie



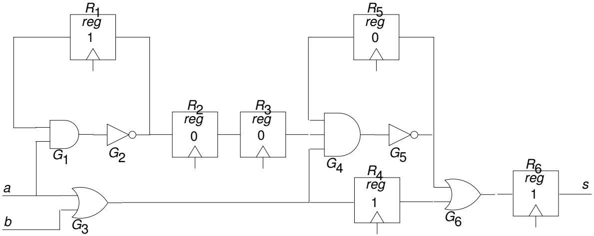
- soit $\mathcal{C}$ contenant des anti-registres :
  pour vérifier que $\mathcal{C}$ satisfait P
  1. on traduit reg_i^{-1} en $reg_i + oracle_i + bug_i$
  2. on vérifie : $(\Box\forall i : \neg bug_i) \Rightarrow P$
     (dans toutes les exécutions où *oracle* devine le futur de *e*
     P doit être vraie)

Anti-registres : conclusion

- les anti-registres sont très utiles
- beaucoup plus de liberté d'optimisation
- il existe d'autres problèmes et d'autres techniques

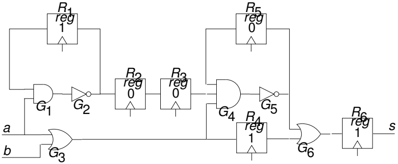
Application

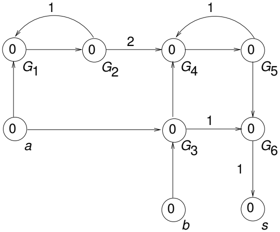
On souhaite réduire le nombre de (anti-)registres, afin de faciliter la vérification du circuit suivant :



Application

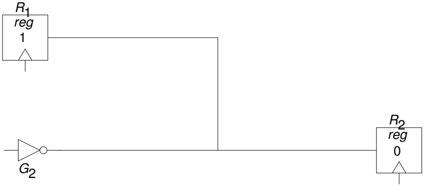
- on travaille sur un graphe plus simple (retiming graph)
- les noeuds sont les portes logiques, les entrées, les sorties
- les arcs comptent les registres entre ces portes
- on veut minimiser la somme des entiers sur les arcs (en valeur absolue)
- les entiers dans les noeuds comptent les traversées de registres effectuées





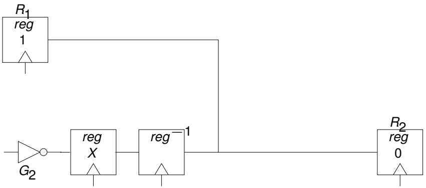
Problème de l'état initial

- l'annihilation de paires registre/anti-registre n'est pas toujours simple (désaccord sur l'état initial)
- un gros plan :

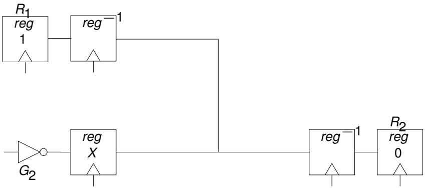


Problème de l'état initial

1



2



On doit avoir $X = 0 = 1$!

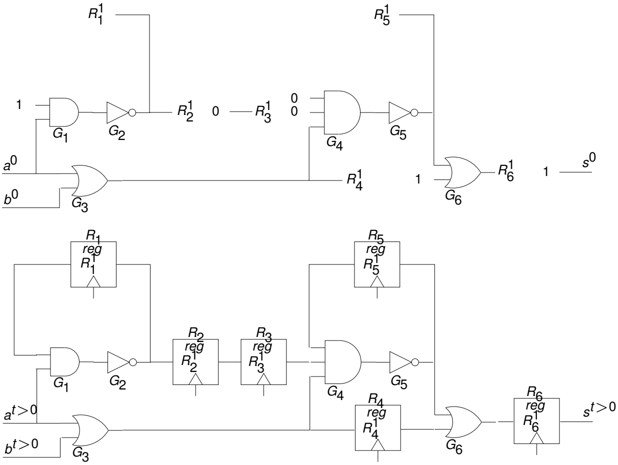
Problème de l'état initial

On résout le problème par dépliage du circuit :

1. on duplique le circuit
2. la première version calcule la transition $t = 0 \rightarrow t = 1$
   $\rightarrow$ on peut supprimer toute la combinatoire après les registres
3. la seconde version calcule les transitions $t > 0 \rightarrow t + 1$
   $\rightarrow$ la valeur initiale des registres est déterminée par le circuit précédent

Application

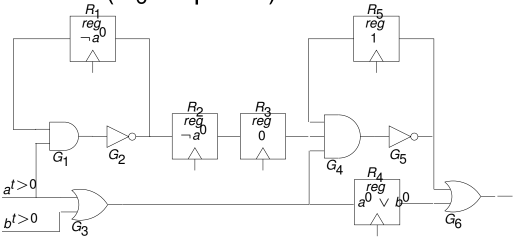
- le dépliage du circuit donne :



- on s'intéressera au second circuit

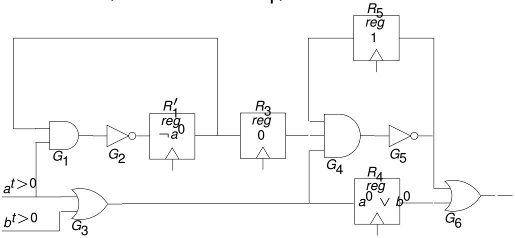
Application

- retiming périphérique
  $\rightarrow$ on retire les registres aux bornes du circuit
  $\rightarrow$ on reconstituera les traces avec le bon décalage temporel
- résultat (R_6 disparaît) :



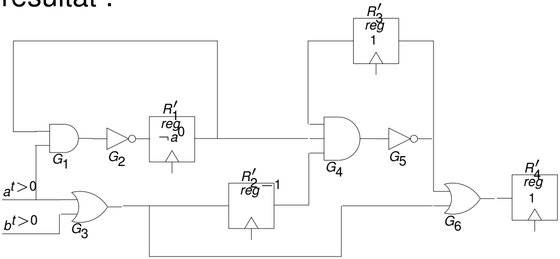
Application

- fusion de registres (de même état initial)
- résultat (R_1, $R_2 \rightarrow R'_1$) :



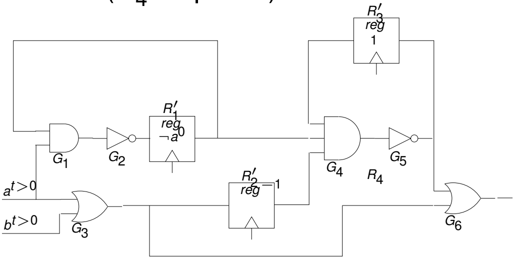
Application

- création d'une paire anti-registre/registre avant G_4 : R'_2
- prop. de reg. à travers G_4, G_5 : R_5, R_3, $R'_2 \rightarrow R'_3$
- prop. de reg. à travers G_6 : R'_3, $R_4 \rightarrow R'_4$
- résultat :



Application

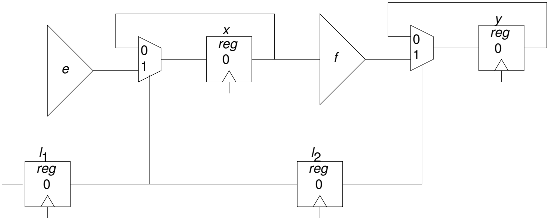
- retiming périphérique
- résultat (R'_4 disparaît) :



- il ne reste que 3 (sur 6) registres / anti-registres !

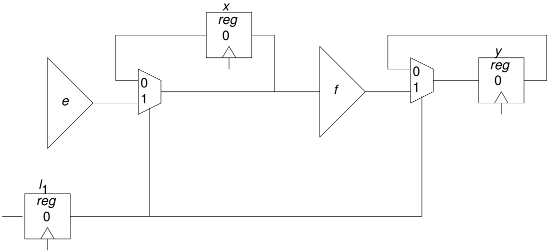
Retiming et logiciel

- le retiming dans le logiciel est possible
- correspond par ex. au réordonnancement d'instructions
- un exemple :

programme	circuit
$l_1 : x = e$; $l_2 : y = f(x)$;	

Retiming et logiciel

- x traverse f
- l_2, x et y traversent demux
- résultat :

programme	circuit
$l_1 : x = e \\ \| y = f(e)$;	

Symétrie et abstraction

constatations

- certaines transitions commutent
- certains états sont observationnellement identiques (bisimilaires)
- certains registres sont permutables

Automorphismes

Soit $\mathcal{K} = \langle S, AP, I, R, L \rangle$ une structure de Kripke.

σ est un automorphisme de $\mathcal{K}$ ssi :

- $\sigma \in S \rightarrow S$ est une permutation sur S
- σ préserve R : $\forall s, t \in S : (s, t) \in R \Leftrightarrow (\sigma(s), \sigma(t)) \in R$
- σ préserve I : $I = \sigma(I)$

$p \in Prop(AP)$ est un invariant pour σ ssi :

- σ préserve p : $\forall s \in S : L(s) \vDash p \Leftrightarrow L(\sigma(s)) \vDash p$

une formule temporelle f est un invariant pour σ ssi :

- soit MP l'ens. des sous-formules prop. maximales de f par ex. si $f = (a \vee \neg b)\mathrm{U}c$, alors $MP = \{a \vee \neg b, c\}$
- $\forall p \in MP : \forall s \in S : L(s) \vDash p \Leftrightarrow L(\sigma(s)) \vDash p$

Groupes et quotients

$G = \langle E, \circ, \_^{-1} \rangle$ est un groupe sur E ssi :

- G est fermé par $\circ$
- G est fermé par $\_^{-1}$

$\Theta(s) \subseteq S$ est une orbite de $s \in S$ pour G ssi :

- G est un groupe de permutations sur S
- $\Theta(s) = \{\sigma(s) \mid \sigma \in G\}$

$\mathcal{Q}$ est une structure quotient de $\mathcal{K}$ pour G ssi :

- G est un groupe d'automorphismes de $\mathcal{K}$
- $AP_{\mathcal{Q}} = AP_{\mathcal{K}}$
- $S_{\mathcal{Q}} = \{\Theta(s) \mid s \in S_{\mathcal{K}}\}$
- $R_{\mathcal{Q}} = \{(\Theta(s), \Theta(t)) \mid (s, t) \in R_{\mathcal{K}}\}$
- $L_{\mathcal{Q}}(\Theta(s)) = L_{\mathcal{K}}(\epsilon(\Theta(s)))$ (ϵ fonction de choix)

Propriétés

G est un groupe d'invariance de f sur $\mathcal{K}$ ssi :

- G est un groupe d'automorphismes sur $\mathcal{K}$
- f est invariant pour tout $\sigma \in G$, ou bien :
- si $f \in LTL$, alors G doit en fait être ($+$ souple) :
  - un groupe d'auto. sur l'auto. de Büchi de f
  - $\sigma \in G$ doit respecter l'équité : $\forall i : F_i = \sigma(F_i)$

préservation des propriétés

- soit G un groupe d'invariance de f sur $\mathcal{K}$
- soit $\mathcal{Q}$ une structure quotient de $\mathcal{K}$ pour G

$\rightarrow$ $L_{\mathcal{Q}}$ est indépendant de ϵ; donc $\mathcal{Q} = \mathcal{K}_{[G]}$ est unique

$\rightarrow$ $\mathcal{K} \models f \Leftrightarrow \mathcal{Q} \models f$

Recommandations

- les structures quotients sont toujours plus petites
- on doit chercher des groupes d'auto. de taille maximale :
  $\rightarrow$ au pire, $G = \{id\}$: $|\Theta(s)| = 1$; $\mathcal{S}_{\mathcal{Q}} = \mathcal{S}_{\mathcal{K}}$
  $\rightarrow$ au mieux, $G = \{\pi_i \mid i \in 1 \ldots n!\}$: $\Theta(s) = \mathcal{S}_{\mathcal{K}}$; $|\mathcal{S}_{\mathcal{Q}}| = 1$
- pour une spécification f, on s'intéressera aux structures quotients issues de groupes d'invariance de f

Trouver les symétries

- déterminer les symétries maximales d'un système est algorithmiquement difficile
- pas intéressant s'il faut d'abord construire l'espace d'états en entier
  $\rightarrow$ caractérisation syntaxique des systèmes exhibant des symétries

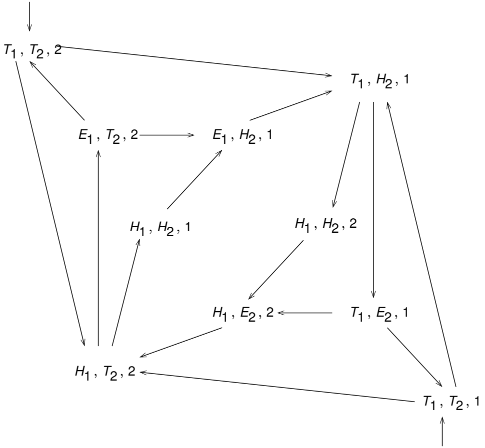
conditions suffisantes de symétrie

- système construits à partir de N composants identiques
- topologie de communication régulière (anneau, graphe complet, etc)
- types de données privés (scalarset) : $==$ et $:=$

Un exemple

Considérons l'algorithme de Peterson :

- protocole d'exclusion mutuelle à 2 proc. très symétrique
- graphe d'états :



Un exemple

spécification

- exclusion mutuelle : $\Box\neg(etat_1 = E \wedge etat_2 = E)$
- absence de famine : $\Box(etat_1 = H \Rightarrow \Diamond etat_1 = E) \wedge \Box(etat_2 = H \Rightarrow \Diamond etat_2 = E)$

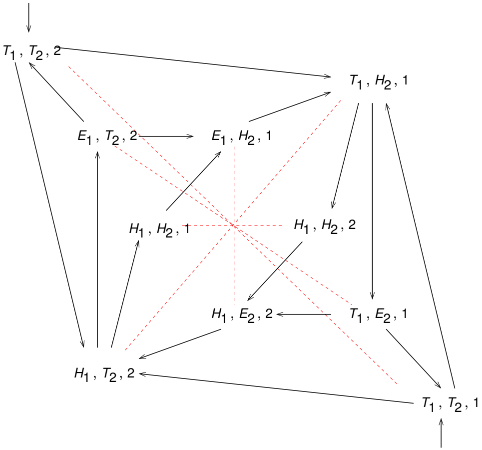
automorphismes invariants

- $(etat_1, etat_2, tour) \mapsto (etat_1, etat_2, tour)$
- $(etat_1, etat_2, tour) \mapsto (etat_2, etat_1, 3 - tour)$

Dessinons les automates de Büchi !

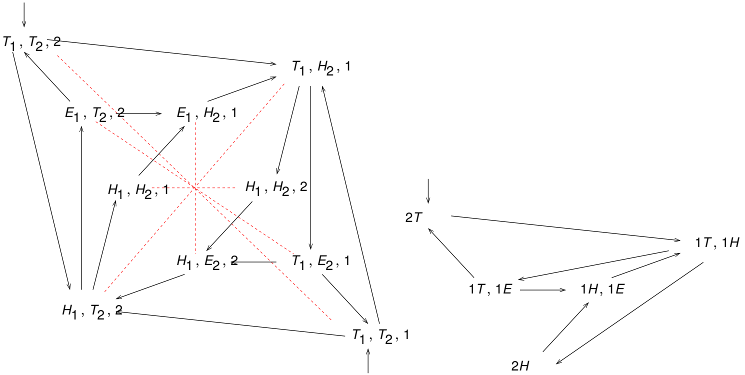
Un exemple

Les orbites sont en rouge :



Un exemple

Structure quotient, invariante pour la spécification :



Types privés

Soit N processus possédant une variable d'état : $etat[i]$

les expressions e sont de la forme :

- e ne contient pas de var. libres indices de proc.
- les indices de proc. dans e sont quantifiés par :
  $\forall i : \ldots$ ou bien $\exists i : \ldots$
- les indices de proc. dans e peuvent seulement être comparés entre eux : $i = j$

les transitions sont de la forme :

- $\forall i : etat[i] := e \texttt{ if } g$
- $\exists i : etat[i] := e \texttt{ if } g$
- e et g satisfont la contrainte précédente
  par ex. $\forall i : etat[i] \neq Eating$ est une garde acceptable

Types privés

- si ces conditions sont vérifiées, alors :

$\rightarrow$ l'ens. $[0..N-1]$ peut être vu comme un type privé

$\rightarrow$ les indices de proc. sont tous permutables

$\rightarrow$ ces permutations (N!) sont des automorphismes

- la spec. doit aussi être invariante (très fréquent)
- sinon, on peut essayer de partitionner les indices en sous-ensembles où les conditions seront vérifiés

Systèmes paramétrés

Ces résultats de symétrie peuvent s'étendre au cas paramétré.

- soit $\mathcal{S}(N)$ composé de N composants identiques, indexés par un type privé T

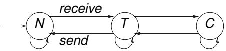
$\rightarrow$ $\exists K : \mathcal{S}(K)_{[\{\pi_T\}]} = \mathcal{S}(K+1)_{[\{\pi_T\}]}$

$\rightarrow$ on n'a besoin de considérer que K valeurs $\neq$ dans T pour être dans le cas général

Un exemple

Considérons le protocole d'exclusion mutuelle par jeton circulant sur un anneau.

- $etat[i] \in \{N, T, C\}$
- graphe d'état pour un proc. :



- synchronisation sur *receive* et *send* (correspond à $token := token \oplus 1$)
- les automorphismes sont les rotations
- graphe quotient :