# Cours de Model Checking

## Leçon 3 : Algorithmes Efficaces de Model Checking



Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

sebastien.bardin@cea.fr
http://sebastien.bardin.free.fr/

# Plan

# Rappels



## Model Checking

Technique de vérification automatique de systèmes réactifs

### Ingrédients

- $\mathcal{M}$ = système de transitions
- $\varphi$ = formule de logique temporelle (LTL, CTL, CTL*, etc.)
- MC = est-ce que $\mathcal{M} \models \varphi$ ?

### Algorithmes déjà vus

- (cas simples : accessibilité, invariance, sûreté)
- CTL : technique de marquage des états
- LTL : transformation de la formule en automate de Büchi
- Fair CTL : marquage de CTL + détection des états “fair”

Problèmes d’efficacité car $\mathcal{M}$ doit être construit entièrement !

# Rappel : la difficulté du model checking

**Model Checking = gérer l'explosion d'états**



## Deux raisons principales

- variables (domaine fini) : $|\mathcal{M}|$ exponentiel en #Var
- concurrence : $|\mathcal{M}|$ exponentiel en #Composants
  - produit asynchrone : entrelacements

# Plan

# Rappel : process du model checking



1. Modéliser le système par $\mathcal{M}$ et la propriété par $\varphi$
2. $\mathcal{M} \models \varphi$ ? Si non, contre-exemple $\sigma$.
3. Analyser les résultats
   - Si oui, **ok**. Attention au lien avec système réel
   - Si non, rejouer $\sigma$ sur système réel.
     - Si vrai bug, alors **erreur**.
     - Sinon goto (1) en raffinant $\mathcal{M}$ grâce à $\sigma$.

# Rappel : process du model checking



1. Modéliser le système par $\mathcal{M}$ et la propriété par $\varphi$
2. $\mathcal{M} \models \varphi$ ? Si non, contre-exemple $\sigma$.
3. Analyser les résultats
   - Si oui, **ok**. Attention au lien avec système réel
   - Si non, rejouer $\sigma$ sur système réel.
     - Si vrai bug, alors **erreur**.
     - Sinon goto (1) en raffinant $\mathcal{M}$ grâce à $\sigma$.

Si $\mathcal{M}$ trop large, prendre un plus petit !

# Simplifications du modèle



## Quelques idées

- enlever des variables inutilisées
  - ex : variables ni dans gardes ni dans prédicats atomiques
- utilisation de symmétries
  - ex : symmétrie en $(x, y)$, alors explorer seulement $\mathcal{M}_{(x \leq y)}$
  - ex : travailler modulo permutation des *process_id*
- limiter le nombre d'états explorés
- fusionner certains états entre eux
  - états ayant mêmes ensembles de traces
  - états avec même signatue (hash)

# Simplifications du modèle



## Quelques idées

- enlever des variables inutilisées
  - ex : variables ni dans gardes ni dans prédicats atomiques
- utilisation de symmétries
  - ex : symmétrie en $(x, y)$, alors explorer seulement $\mathcal{M}_{(x \leq y)}$
  - ex : travailler modulo permutation des *process_id*
- limiter le nombre d'états explorés
- fusionner certains états entre eux
  - états ayant mêmes ensembles de traces
  - états avec même signatue (hash)

- quelles garanties ?
- simplifications systématiques, dépendantes de $\mathcal{M}$, ou dépendantes de $(\mathcal{M}, \varphi)$ ?

# Correction ?



Lien entre le model initial $\mathcal{M}$, et le nouveau modèle $\mathcal{M}^{\#}$ tq $|\mathcal{M}^{\#}| << |\mathcal{M}|$

- équivalence : $\mathcal{M}^{\#} \models \varphi$ ssi $\mathcal{M} \models \varphi$
- sur-approximation : si $\mathcal{M}^{\#} \models \varphi$ alors $\mathcal{M} \models \varphi$
- sous-approximation : si $\mathcal{M}^{\#} \not\models \varphi$ alors $\mathcal{M} \not\models \varphi$

Remarque : pour une $\varphi$ donnée, ou pour toute $\varphi \in \mathcal{L}$ ?

# Simplifications du modèle



## Quelques idées

- enlever des variables inutilisées [équivalent]
- utilisation de symmétries [équivalent]
- limiter le nombre d'états explorés
  [ sous-approx pour AGp, surapprox pour EFp]
- considérer comme égaux des états avec même hash (et les couper)
  [ idem]
- fusionner des états entre eux
  [ surapprox pour AGp, sous-approx pour EFp]

# Cadre général



Relation d'équivalence $\equiv$ entre états de $\mathcal{M}$
(hyp : $s \equiv s \Rightarrow l(s) = l(s')$)

- notion de classe d'équivalence de $s$ : $s_{\equiv} = \{s' | s \equiv s'\}$
- travailler sur le système quotient $\mathcal{M}_{\equiv}$
  $\mathcal{M}_{\equiv} = \langle Q_{\equiv}, \rightarrow_{\equiv}, \mathrm{AP}, l_{\equiv}, s_{0\equiv} \rangle$
- $s_{\equiv} \rightarrow_{\equiv} s'_{\equiv}$ ssi il existe $s_1 \in s_{\equiv}$ et $s'_1 \in s'_{\equiv}$ tq $s_1 \rightarrow s'_1$
- $l_{\equiv}(s_{\equiv}) = l(s)$

# Cadre général (2)



La correction dépend de la relation $\equiv$

- bisimilarité : $\mathcal{M}$ et $\mathcal{M}_{\equiv}$ équivalents pour CTL*
- égalité de $w$-languages : $\mathcal{M}$ et $\mathcal{M}_{\equiv}$ équivalents pour LTL
- équivalence de simulation : $\mathcal{M}$ et $\mathcal{M}_{\equiv}$ équivalents pour ACTL*
- égalité de $w$-languages modulo bégaiement : $\mathcal{M}$ et $\mathcal{M}_{\equiv}$ équivalents pour LTL-X
- ...

# Cadre général (2)



La correction dépend de la relation $\equiv$

- bisimilarité : $\mathcal{M}$ et $\mathcal{M}_{\equiv}$ équivalents pour CTL*
- égalité de $w$-languages : $\mathcal{M}$ et $\mathcal{M}_{\equiv}$ équivalents pour LTL
- équivalence de simulation : $\mathcal{M}$ et $\mathcal{M}_{\equiv}$ équivalents pour ACTL*
- égalité de $w$-languages modulo bégaiement : $\mathcal{M}$ et $\mathcal{M}_{\equiv}$ équivalents pour LTL-X
- ...

# Équivalences comportementales : bisimilarité



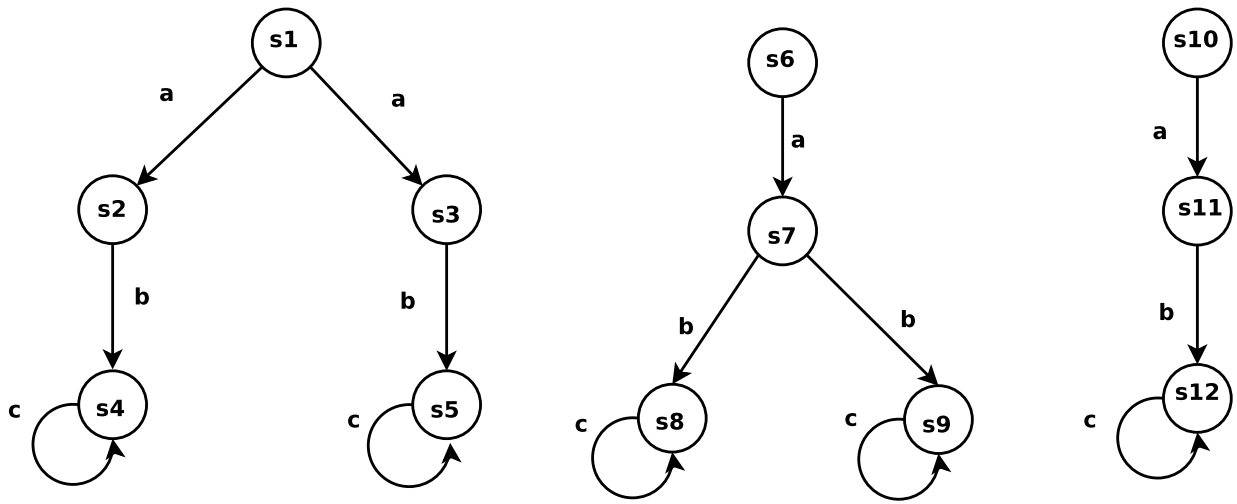
Bisimulation : relation $R \subseteq Q \times Q$ tq si $sRs'$ alors

- $l(s) = l(s')$
- si $s \rightarrow s_1$ alors il existe $s'_1$ tq : $s' \rightarrow s'_1$ et $s_1 R s'_1$
- si $s' \rightarrow s'_1$ alors il existe $s_1$ tq : $s \rightarrow s_1$ et $s_1 R s'_1$

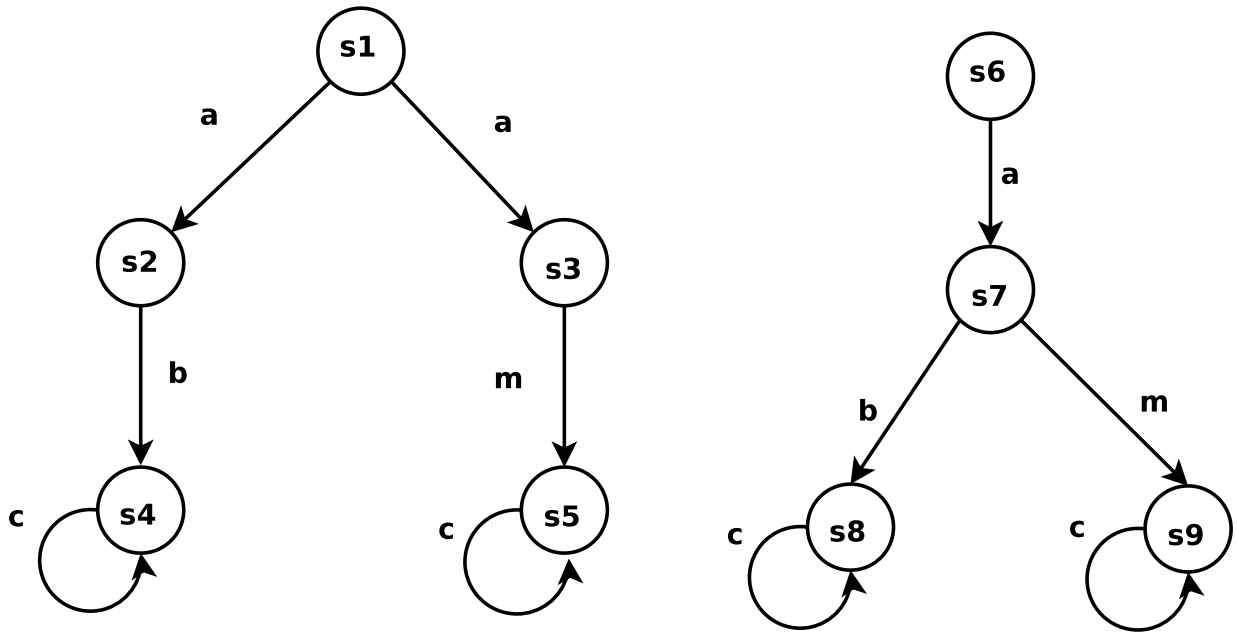
Bisimilarité : $s$ et $s'$ sont bisimilaires si il existe une bisimulation $R$ tq $sRs'$

- conserve CTL*

# Exemples de systèmes bisimilaires



# Exemples de systèmes non bisimilaires



# Équivalences comportementales : traces



Équivalence de traces : $s$ et $s'$ sont trace-équivalents si $\mathcal{L}(s) = \mathcal{L}(s')$

- conserve LTL

Liens : $s$ et $s'$ bisimilaires $\Rightarrow \mathcal{L}(s) = \mathcal{L}(s')$, mais pas l'inverse
(cf slide précédent)

# Remarque



On peut procéder de manière similaire avec une relation d'ordre $\preceq$

- simulation : $\mathcal{M}_{\preceq}$ surapprox $\mathcal{M}$ pour ACTL*
- inclusion de $w$-languages : $\mathcal{M}_{\preceq}$ surapprox $\mathcal{M}$ pour LTL
- ...

# Plan

# Calcul à la volée



Remplacer

1. Calculer $\mathcal{M}$
2. Transformer $\varphi_p$ en automate $B_{\neg\varphi_p}$
3. Calculer $B_\otimes$ reconnaissant $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(B_{\neg\varphi_p})$
4. Tester si $\mathcal{L}(B_\otimes) = \emptyset$

par

1. Transformer $\varphi_p$ en automate $B_{\neg\varphi_p}$
2. Calculer $B_\otimes$ et vérifier à la volée si $\mathcal{L}(B_\otimes) = \emptyset$

# Calcul à la volée (2)



Intérêt : ne calcule pas $\mathcal{M}$ en entier

- propriété fausse : s'arrête dès que cex trouvé !
- cas général : $B_{\otimes}$ peut ne contenir qu'une petite partie de $\mathcal{M}$
  ex : $\varphi : A(p \Rightarrow ...)$, et $p$ faux sur $s_0$.

# Calcul à la volée (3)



pb = détection on-the-fly de cycles acceptants (test du vide de Büchi)

solution naive : une DFS par noeud (arghh)

nested DFS : chaque noeud exploré $\leq$ 2 fois

# Rappel : DFS



Entrées : structure $\mathcal{M}$ avec état initial $s_0$
Sorties : ensemble d'accessibilité $\text{post}^*(s_0)$

$R := \emptyset$ /* *états atteints et traîtés* */
$Q := push(s_0, \bot)$ /* *pile des états atteints non traîtés* */
**Tant que** $Q$ non vide **faire**
  $q := top(Q)$;
  **Essayer de choisir** $q'$ **tq** $q \rightarrow q'$ et $q' \notin (R \cup Q)$
    . **si possible faire** $Q := push(q', Q)$
    . **sinon faire** $R := R + \{q\}$ ; $Q := pop(Q)$
  **Fin essai**
**Fin Tant que**
**retourner** $R$

# Détection de cycles : nested DFS naïf



## Deux types de DFS imbriquées

- 1 dfs externe pour calculer les états accessibles
- $|\mathcal{M}|$ dfs internes pour vérifier si $s \rightarrow^+ s$ (cycle)
  - . $\text{dfs}^+$ : comme dfs, mais retourne les successeurs stricts

simple, correct, mais quadratique dans le nombre d'états accessibles

Entrées : structure $\mathcal{M}$ avec état initial $s_0$
Sorties : oui ssi il existe un cycle acceptant

*cycle* := *false* ;
$R := \emptyset$ /* *dfs externe : ...* */
$Q := push(s_0, \bot)$ /* *dfs externe : ...* */
**Tant que** $Q$ non vide ET $\neg$*cycle* **faire** /* *dfs externe ...* */
  $q := top(Q)$;
  **Si** $q \in dfs^+(\mathcal{M}, q)$ **alors** *cycle* := *true* **Fin Si** /*dfs interne*/
  **Essayer de choisir** $q'$ **tq** $q \rightarrow q'$ et $q' \notin (R \cup Q)$
    . **si possible faire** $Q := push(q', Q)$
    . **sinon faire** $R := R + \{q\}$ ; $Q := pop(Q)$
  **Fin essai**
**Fin Tant que**
**retourner** *cycle*

# Détection de cycles : nested DFS



voir : [Courcoubetis-Vardi-Wolper-Yannakakis : 1990]

Les deux classes de DFS sont partagées

- chaque état exploré au plus 2 fois (externe, interne)
- efficace :
  - . temps $\leq 2 \times$ temps(dfs)
  - . space $\approx$ space(dfs) [2 bits en $+$ par état]
- optim en plus : stop quand la dfs interne rencontre un état sur la stack de la dfs externe
  - . juste une optim, correct mais pas complet

attention

- assez tricky (la complétude dépend de bcp de détails)
- pas compatible avec les ordres partiels (cf après)
  - . une version compatible existe

# Automates de formule plus petits



Les automates de Büchi n'ont pas de forme minimale : essayer de réduire la taille

- enlever les noeuds non accessibles de $q_0$
- fusionner les noeuds $n$ tq $L(n) = \emptyset$ dans un noeud $KO$
- fusionner les noeuds $n$ tq $L(n) = \top$ dans un noeud $OK$

  . + gestion ad hoc des noeuds OK et KO
- fusionner les noeuds bisimilaires et/ou L-équivalents
- ... (cf LTL2BA, Spot)

Automates plus complexes

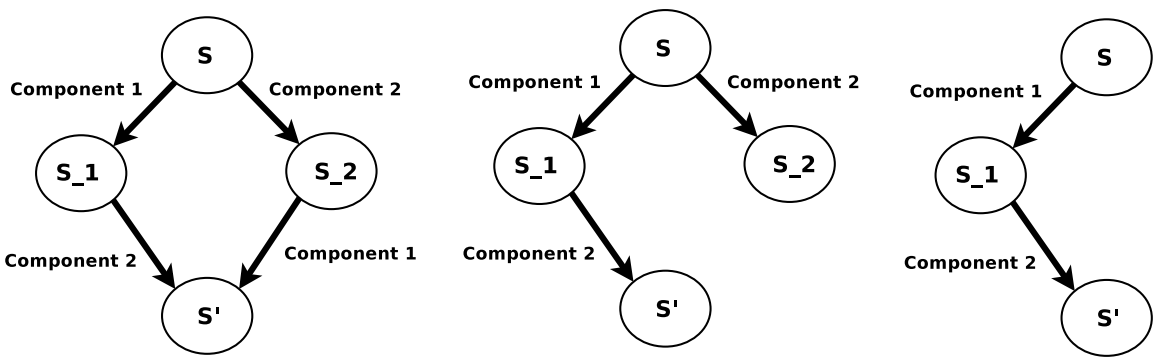
- encodage de la formule LTL plus concis, mais opérations d'automates plus complexes
- ex1 : équité faible (justice) $\mathbf{G}^\infty \Rightarrow \mathbf{F}^\infty$ : Generalized Büchi Automata
- ex2 : équité forte (compassion): $\mathbf{F}^\infty \Rightarrow \mathbf{F}^\infty$ : Streett automata

# Entrelacement et ordres partiels



Ordres partiels (Valamari, Peled, Godefroid)

- certains états/transitions sont redondants pour certaines propriétés
- pas besoin de construire tout $\mathcal{M}$
- très adapté aux systèmes asynchrones



# Plan



- Introduction
- Meilleure modélisation
- Optimisations pour LTL
- Optimisations pour CTL et model checking symbolique
- Disgression

# Model Checking Symbolique



Cible originelle : hardware

- très grand nombre de variables booléennes

Idée générale (McMillan 90)

- Ensembles d'états représentés par des équations
- Bien plus concis : $n$ variables $a_1, \ldots, a_n$, alors la formule $\top$ représente $2^n$ n-uplets possibles
- Les BDD implantent très efficacement les équations booléennes
- Très adapté à CTL pour circuits synchrones
- Pas parfait : entiers, concurrence, autre logique

# Model Checking Symbolique (2)



## Ingrédients

- domaine de valeurs symboliques $D$ représentant des *ensembles* de configuration
  - $[\![]\!] : D \mapsto 2^Q$
- calculs symboliques sur $D$ copiant les calculs ensemblistes sur $2^Q$
  - $\bot$ et $\top$ tels que $[\![\bot]\!] = \emptyset$ et $[\![\top]\!] = 2^Q$
  - $[\![d \sqcup d']\!] = [\![d]\!] \cup [\![d']\!]$
  - $[\![d \sqcap d']\!] = [\![d]\!] \cap [\![d']\!]$
  - $[\![\neg d]\!] = 2^Q \setminus [\![d]\!]$
  - $d \sqsubseteq d'$ ssi $[\![d]\!] \subseteq [\![d']\!]$
  - $d = \bot$ ssi $[\![d]\!] = \emptyset$
- calcul de pre et post symboliques (ici : toutes les transitions d'un coup)
  - $[\![post^\#(d)]\!] = post([\![d]\!])$

Avec ça : on refait le calcul d'accessibilité, version ensembliste

# Model Checking Symbolique (2)



## Ingrédients

- domaine de valeurs symboliques $D$ représentant des *ensembles* de configuration
  - $[\![]\!] : D \mapsto 2^Q$
- calculs symboliques sur $D$ copiant les calculs ensemblistes sur $2^Q$
  - $\bot$ et $\top$ tels que $[\![\bot]\!] = \emptyset$ et $[\![\top]\!] = 2^Q$
  - $[\![d \sqcup d']\!] = [\![d]\!] \cup [\![d']\!]$
  - $[\![d \sqcap d']\!] = [\![d]\!] \cap [\![d']\!]$
  - $[\![\neg d]\!] = 2^Q \setminus [\![d]\!]$
  - $d \sqsubseteq d'$ ssi $[\![d]\!] \subseteq [\![d']\!]$
  - $d = \bot$ ssi $[\![d]\!] = \emptyset$
- calcul de pre et post symboliques (ici : toutes les transitions d'un coup)
  - $[\![post^\#(d)]\!] = post([\![d]\!])$

Avec ça : on refait le calcul d'accessibilité, version ensembliste

On peut en fait faire tout CTL*

# Calcul symbolique de $post^*(Q_0)$



Calcul symbolique d'accessibilité ($d_I$ tq $[\![d_I]\!]$ = états initiaux)

$d := d_I$
**Tant que** $post^{\#}(d) \not\sqsubseteq d$ **faire**
  $d := post^{\#}(d) \sqcup d$
**Fin Tant que**
**retourner** $d$

Remarque : ici juste besoin de $\sqcup, post^{\#}, \sqsubseteq$
. CTL et CTL* ont besoin des autres opérateurs

# Model Checking Symbolique (3)



Quelle représentation symbolique D ? (pour ensemble sur $\mathcal{B}^N$)

- tables de vérités :
  . taille = toujours $2^N$, indépendamment #éléments
- ensembles explicites :
  . taille = #elts (noté $\#e$), au pire $2^N$ (ex : $\top$)
  . opérations binaires en $\#e \cdot ln(\#e)$
  . test du vide en temps constant
  . complément en $2^N$, post en $\#e \cdot max - \#succ$
- forme logique normale CNF (DNF même genres de problèmes)
  . taille au pire $2^{\#e}$ (mais $\top$ concis)
  . $\sqcap$ constant ($\wedge$)
  . autres opérations coûteuses (empty? : NP-hard, $\sqsubseteq$ : coNP-hard, $\sqcup$ : exponentiel)
- arbres de décision binaire (binary decision tree, bdt)
  . taille = toujours $2^{N+1} - 1$, indépendamment #éléments

# Model Checking Symbolique : BDDs



## BDD : Binary Decision Diagram

- représentation canonique concise, sous forme de DAG

  . idée : arbre des valuations, + partage des sous-arbres, + élimination des noeuds inutiles

  . gain possiblement exponentiel
- algorithmes ensemblistes efficaces
- forme normale : $= \emptyset$? ou $= \top$? en $O(1)$
- possibilité d'avoir un cache : $d = d'$ en $O(1)$

Attention :

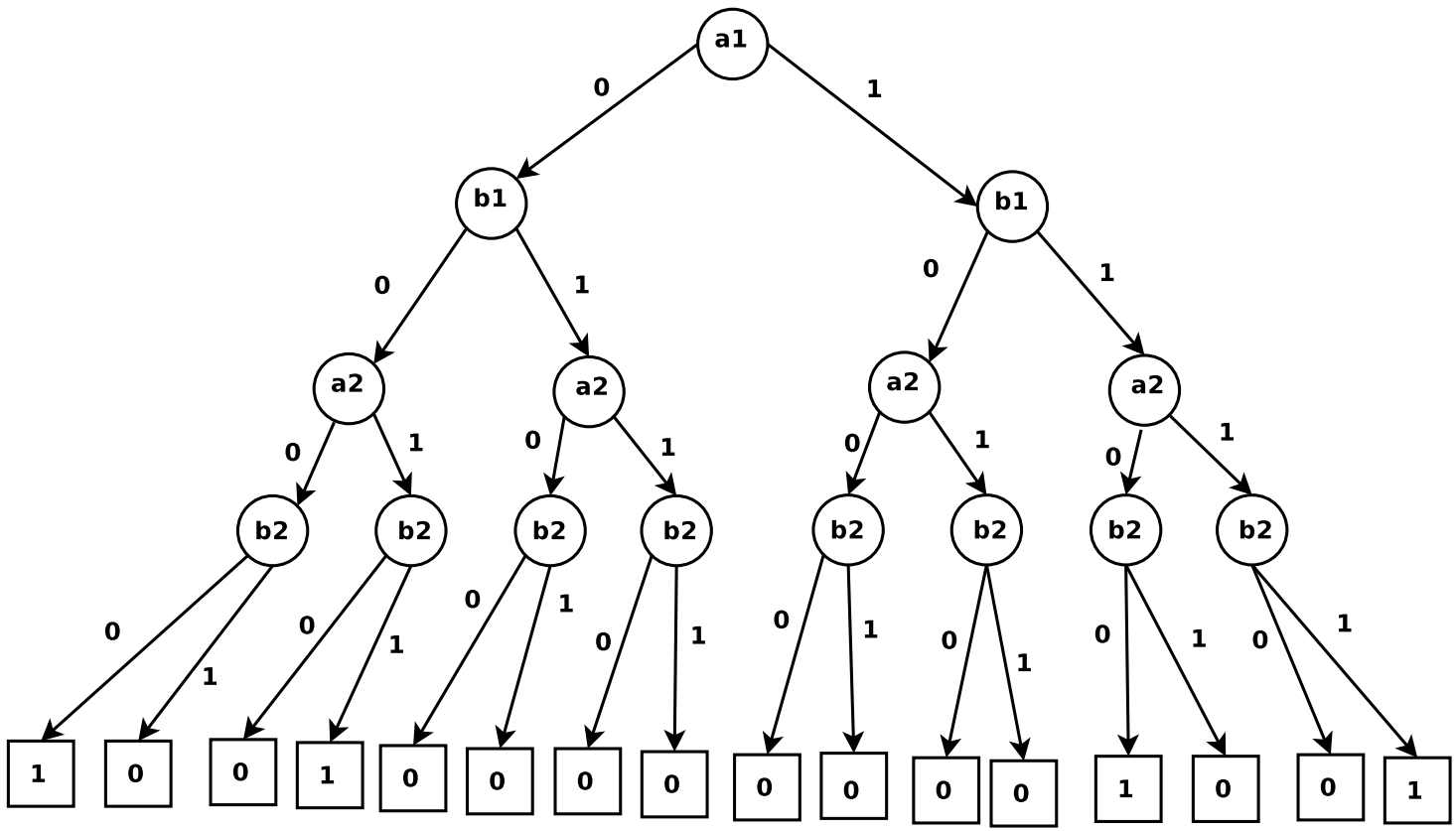
- le BDD peut exploser (représentation finale)
- les calculs intermédiaires du BDD peuvent exploser
- taille du BDD dépend fortement de l'ordre des variables

# Exemple : Binary Decision Tree



## Comparateur sur 2 bits

- entrée $a = (a_1, a_2)$ et $b = (b_1, b_2)$
- répond vrai ssi $a = b$



# Des BDT aux BDD



BDD = BTT tel que :

- les sous-arbres isomorphes sont fusionnés
- les noeuds avec des fils isomorphes sont éliminés

Remarques :

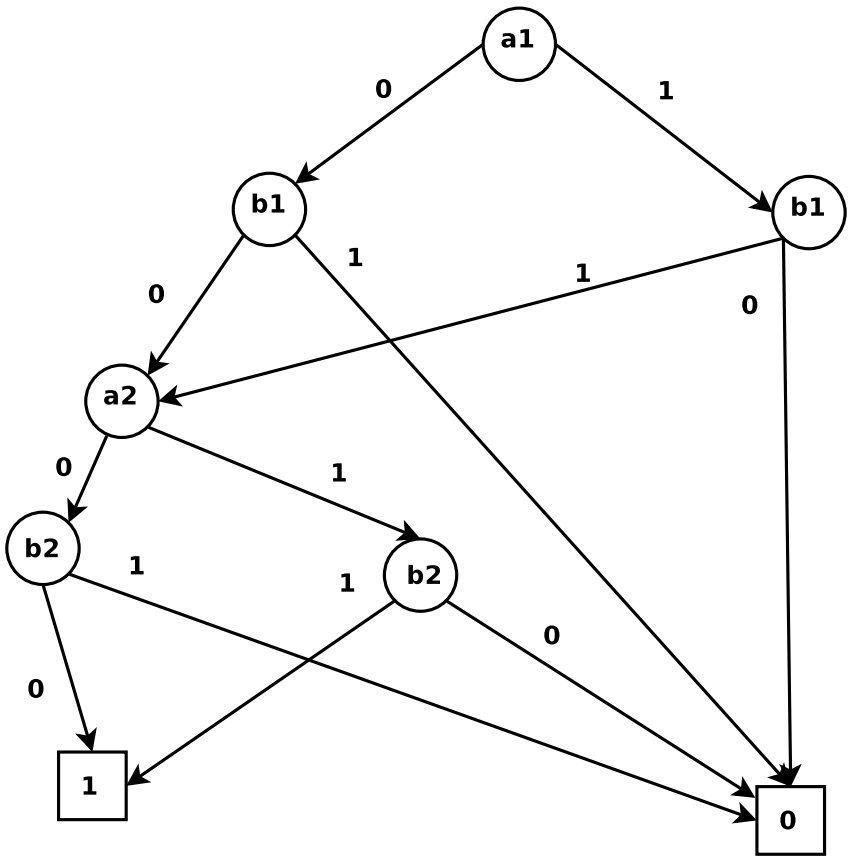
. donne une forme normale

. donne aussi un algo de normalisation (bottom-up)

# Exemple : Binary Decision Diagram



## Comparateur sur 2 bits



# Remarque sur l'ordre



pour l'exemple d'un n-bit comparateur :

- ordre $a_1 < b_1 < \ldots a_n < b_n$ : taille(bdd) $= 3n+2$
- ordre $a_1 < \ldots a_n < b_1 < \ldots < b_n$ : taille(bdd) $= 3.2^n$-1

# Plan



- Introduction
- Meilleure modélisation
- Optimisations pour LTL
- Optimisations pour CTL et model checking symbolique
- Disgression

# Quelques Model Checkers



## Outil SMV de CMU (McMillan)

- CTL sur systèmes synchrones avec variables booléennes
- représentation symbolique par BDD
- autres : assume-garantee, abstraction
- $10^{20}$ états couramment, des études ‘a $10^{800}$

## Outil SPIN du Bell Labs (Holzmann et Peled)

- LTL sur systèmes asynchrones avec variables non booléennes
- énumération concrète optimisée
- ordres partiels, génération à la volée, memory compression
- $10^{10}$ états couramment, jusqu’à $10^{30}$

# Quelques Model Checkers



## Outil SMV de CMU (McMillan)

- CTL sur systèmes synchrones avec variables booléennes
- représentation symbolique par BDD
- autres : assume-garantee, abstraction
- $10^{20}$ états couramment, des études ‘a $10^{800}$

## Outil SPIN du Bell Labs (Holzmann et Peled)

- LTL sur systèmes asynchrones avec variables non booléennes
- énumération concrète optimisée
- ordres partiels, génération à la volée, memory compression
- $10^{10}$ états couramment, jusqu’à $10^{30}$

## Et aussi : SPOT, Paris 6 et Epita

- model checking de LTL, automates très optimisés

# Sur le model checking symbolique



Le model checking symbolique ouvre de nombreuses possibilités !!

Les ensembles peuvent être infinis $\{(x, y, z).x + y \geq z\}$

- model checking de systèmes infinis
- quelques beaux résultats (pushdown systems, automates temporisés, réseaux de Petri)

Les ensembles peuvent être approchés $x \in [0..100]$ pour coder $\{0, 65, 100\}$

- liens avec l'interprétation abstraite
- un exemple utile : predicate abstraction

# Sur le model checking symbolique



Le model checking symbolique ouvre de nombreuses possibilités !!

Les ensembles peuvent être infinis $\{(x, y, z).x + y \geq z\}$

- model checking de systèmes infinis
- quelques beaux résultats (pushdown systems, automates temporisés, réseaux de Petri)

Les ensembles peuvent être approchés $x \in [0..100]$ pour coder $\{0, 65, 100\}$

- liens avec l'interprétation abstraite
- un exemple utile : predicate abstraction

Le software model checking se base extensivement sur ces idées