Sémantique et analyse de programmes *model-checking* booléen

David Monniaux

CNRS / VERIMAG

15 mars 2011

Plan

États de départ + transitions

État du programme / de la machine = valeur des variables, registres, mémoires... pris dans un certain ensemble Σ.

Par exemple :

- si état du système = valeur de 17 bits, alors $\Sigma = \{0,1\}^{17}$;
- si état du système = 3 entiers non bornés, alors $\Sigma = \mathbb{Z}^3$;
- si automate fini, Σ est l'ensemble (fini) des états ;
- si automate à pile, état total = couple (état de l'automate, contenu de la pile), donc $\Sigma = \Sigma_S \times \Sigma_P^*$.

Relation de transition $\rightarrow$: $x \rightarrow y$ = « si je suis en x alors je peux passer en y au coup suivant »

Propriétés de sécurité

Montrer qu'un programme n'atteint pas un **état indésirable** (plantage, erreur, hors spécification). Ensemble W d'états indésirables.

Autrement dit : montrer qu'il n'existe pas $n \geq 0$ et $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \sigma_n$, σ_0 état initial (= reset), $\sigma_n \in W$ (trace de longueur $n+1$ qui emmène à l'indésirable).

Autrement noté : $\sigma_0 \rightarrow^* \sigma_n \in W$. $\rightarrow^*$ **clôture transitive** de $\rightarrow$.

États accessibles

Soit $\Sigma_0 \subseteq \Sigma$ l'ensemble des états initiaux. L'ensemble des **états accessibles** A du système est l'ensemble des σ tels que

$$\exists \sigma_0 \in \Sigma_0 \ \sigma_0 \rightarrow^* \sigma \tag{1}$$

On veut montrer que $A \cap W = \emptyset$.

Calcul comme limite

X_n est l'ensemble des états accessibles en n coups de $\rightarrow$:
$X_0 = \Sigma_0$, $X_1 = \Sigma_0 \cup R(\Sigma_0)$, $X_2 = \Sigma_0 \cup R(\Sigma_0) \cup R(R(\Sigma_0))$,
etc.

avec $R(X) = \{y \in \Sigma \mid \exists x \in X \; x \rightarrow y\}$.

La suite X_k est croissante pour l'ordre $\subseteq$. La limite de la suite (= l'union de tous les termes) est l'**ensemble des états accessibles** tout court.

Calcul itératif

Remarquer que $X_{n+1} = \Sigma_0 \cup R(X_n)$.

Détails en TD. Intuition : pour atteindre en au plus $n+1$ coups

- soit on peut atteindre en 0 coups, autrement dit, on est sur un état initial : Σ_0
- soit on peut atteindre en $0 < k \leq n+1$ coups, autrement dit on fait déjà au maximum n coups (X_n), puis un coup supplémentaire.

Mais comment **calculer effectivement** les X_n ? Et la limite ?

Model-checking explicite

On représente explicitement les ensembles X_n en mémoire (de façon algorithmiquement astucieuse ?).

Si Σ est fini, la suite X_n converge en au maximum $|\Sigma|$ itérations ($|A|$ note le cardinal de l'ensemble A).

Raison :

- Soit $X_n = X_{n+1}$. Donc reste constant pour tous les n suivants.
- Soit $X_n \subsetneq X_{n+1}$, mais alors $X_{n+1} \setminus X_n$ contient au moins 1 état. On ne peut pas faire cela plus de $\|\Sigma\|$ fois.

Coût élevé

Si $\Sigma = \{0,1\}^n$ (n bits de mémoire), $|\Sigma| = 2^n$.

Tendance à explosion en temps et mémoire.

Plan

Le problème

Représenter de façon **compacte** des ensembles d'états booléens.

Remarque : un ensemble de vecteurs de n booléens = une fonction de $\{0,1\}^n$ dans $\{0,1\}$.

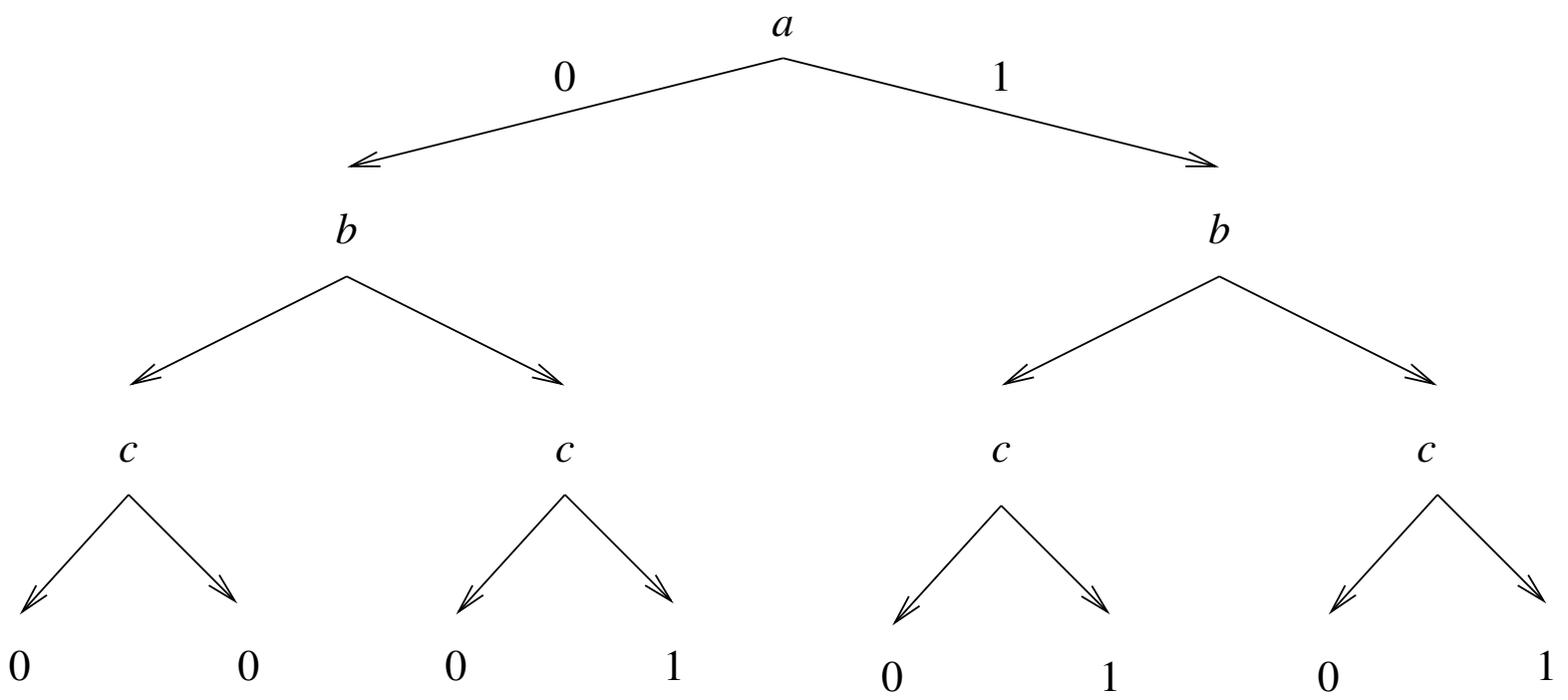
Exemple : $\{(0,0,0),(1,1,0)\}$ représenté par la fonction qui envoie $(0,0,0) \mapsto 1$, $(1,1,0) \mapsto 1$ et 0 partout ailleurs.

Pour le moment, juste de la paraphrase...

BDD expansé

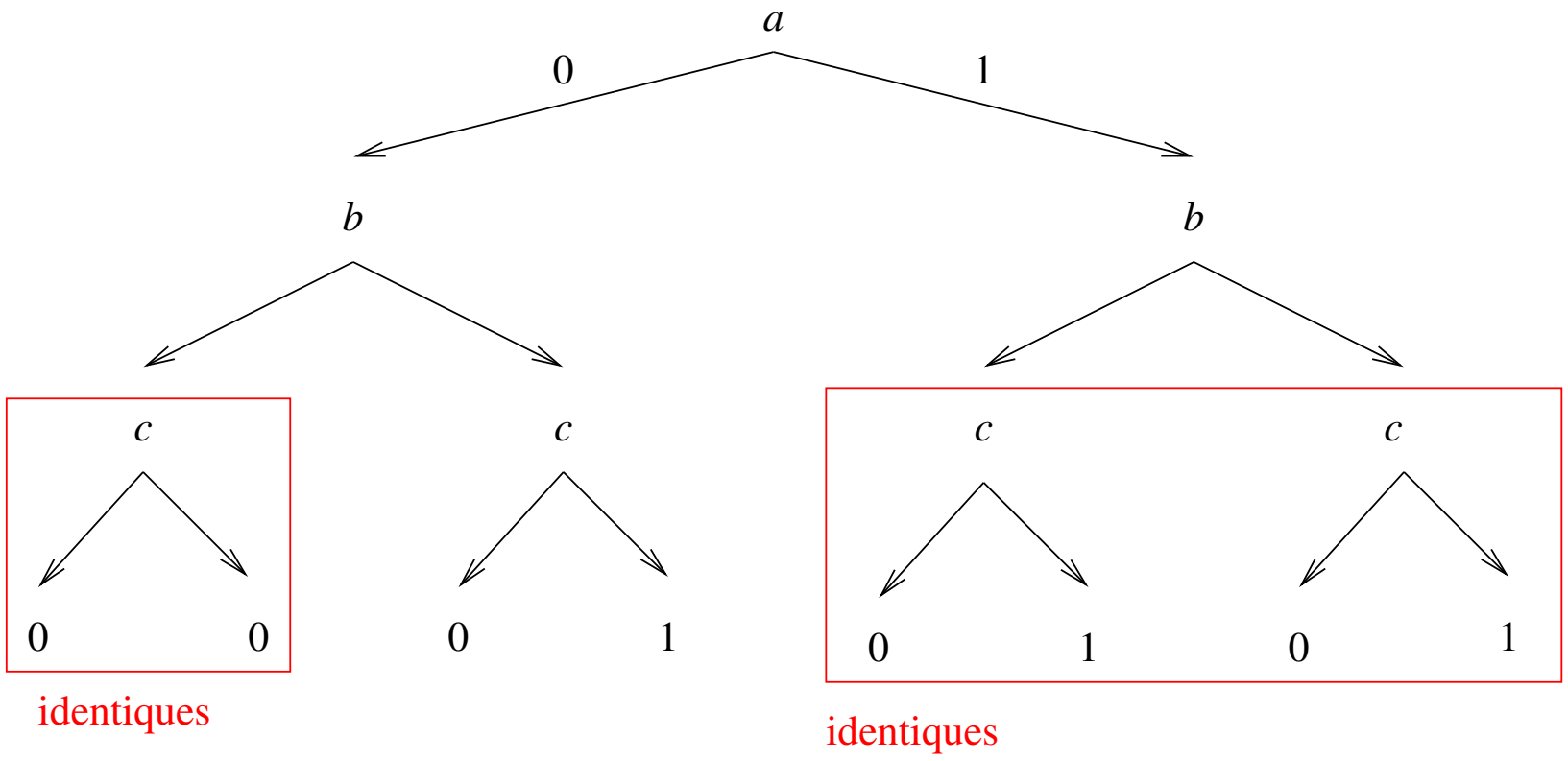
Binary decision diagrams

Étant donné un ordre de variables booléennes (a, b, c), représenter $(a \wedge c) \vee (b \wedge c)$:

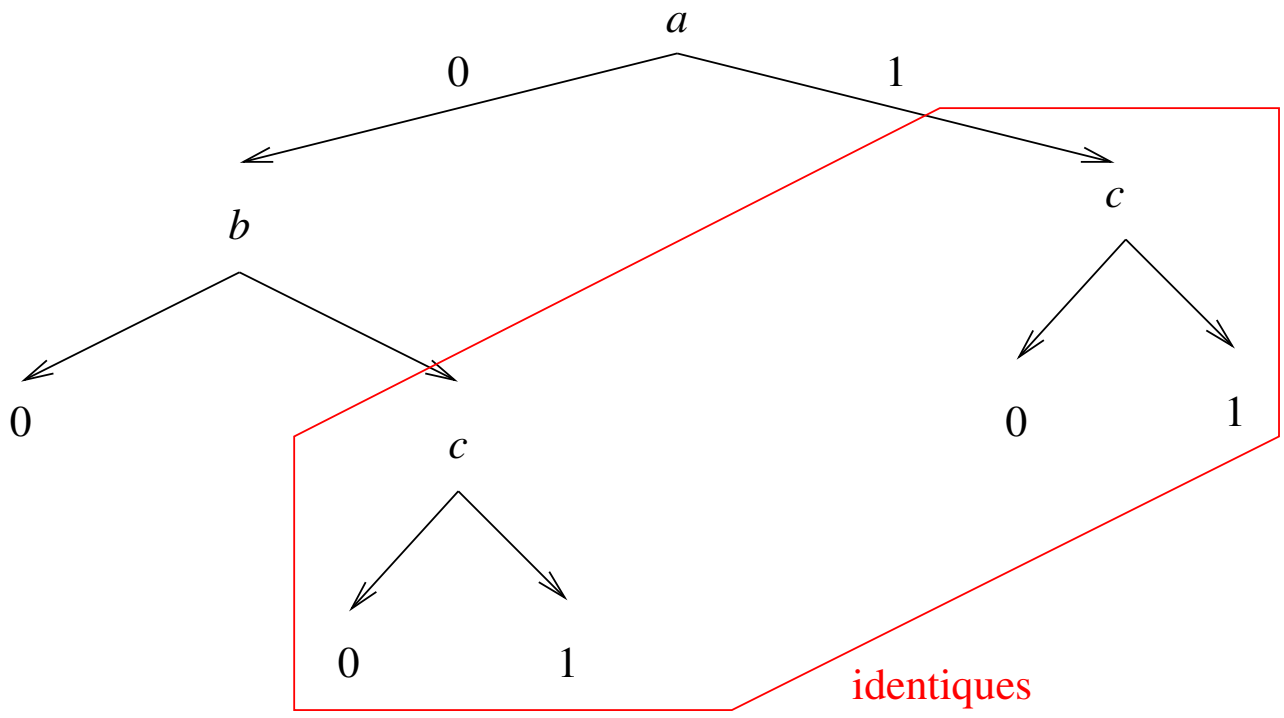


Supprimer les nœuds inutiles

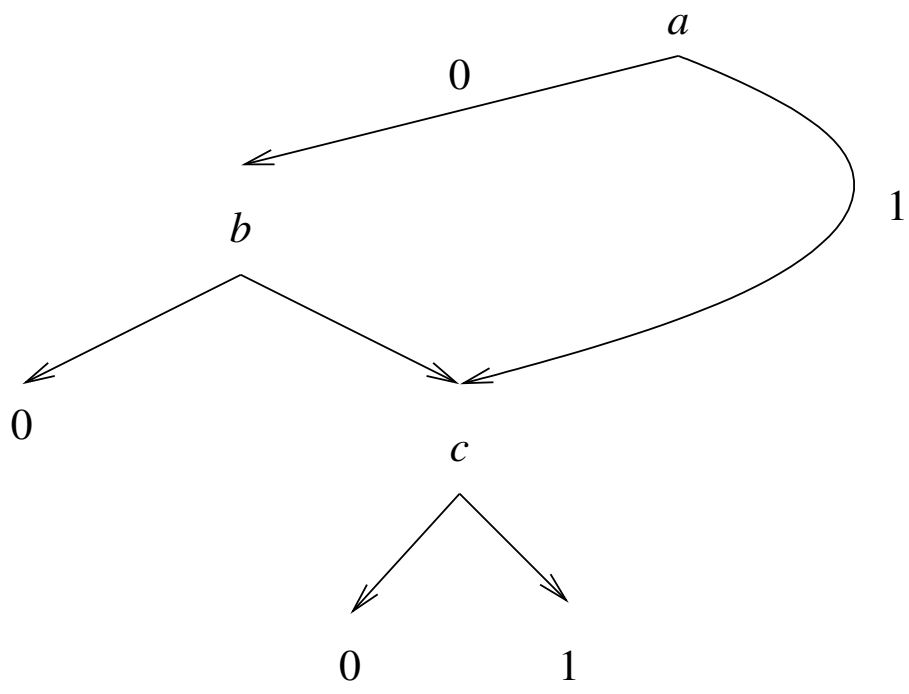
Il est bête de garder deux sous-arbres identiques :



Compression



BDD réduit



Idée : transformer l'arbre original en DAG (graphe orienté acyclique et connexe = arbre avec partage des sous-arbres) avec **partage maximal**.

On se débrouille pour ne jamais créer deux sous-arbres différents. **Canonicité** : un ensemble est toujours codé par le même DAG.

Implémentation : *hash-consing*

Attention : technique d'implémentation que vous pourrez utiliser dans d'autres contextes.

"Consing" à partir de "constructor" (cf Lisp : `cons`).

On garde une **table de hachage** de tous les objets (nœuds d'arbres) déjà créés, avec le hachage $H(x)$ d'un nœud portant une variable v et deux branches b_0 et b_1 calculé à partir de $v, H(b_0), H(b_1)$.

Si un objet correspondant à (v, b_0, b_1) existe déjà dans la table, on retourne cet objet.

Garbage-collection : nécessite des *weak pointers*, cf classe `WeakHashMap` (le pointeur depuis la table de hachage ne doit pas empêcher le GC).

Le *hash-consing* c'est magique

Garantit :

- **partage maximal** : jamais deux objets à contenu identiques créés à deux endroits différents en mémoire
- test d'égalité en $O(1)$: il suffit de **comparer les pointeurs**.

Et surtout une fois qu'on l'a, on peut implémenter facilement des BDD naïfs.

Opérations sur les BDD

Une fois un ordre choisi sur les variables :

- Créer BDD false, true.
- Créer BDD correspondant à formule v, v une variable.
- Opérations $\wedge$, $\vee$, etc.

Opérations $\wedge$, $\vee$: par descente récursive sur les deux arbres, en **programmation dynamique** (si on calcule $f(a, b)$ avec $f(a, b)$ déjà calculé, on ressort la valeur déjà calculée, pour une opération f).

Quantificateurs

On a un BDD pour une formule F sur les variables booléennes x, y, z. On cherche un BDD pour la formule $\exists x\ F$ sur les variables y et z. Ça marche car $[\exists x\ F](y, z) \equiv F(0, y, z) \vee F(1, y, z)$, autrement dit on calcule $F[0/x] \vee F[1/x]$ ($F[b/x]$ est F où x remplacée par b).

Pareil pour $\forall$ sauf qu'on met un $\wedge$.

Autrement dit, on **élimine les quantificateurs**.

Revenons à nos moutons de transition

- On a un ensemble Σ_0 d'états initiaux, donnés par l'utilisateur comme une formule sur les variables d'état booléennes $x_1, \ldots, x_n \Rightarrow$ un BDD sur n variables.
- On a une relation de transition donnée comme une formule T sur les variables d'état booléennes $x_1, \ldots, x_n, x'_1, \ldots, x'_n$ ($x' =$ nouvelle valeur de x après la transition) $\Rightarrow$ un BDD sur $2n$ variables.

Rappelons l'opérateur $\phi(X) = \Sigma_0 \cup R(X)$, autrement dit en formules

$$\phi(X) = \Sigma_0 \vee (\exists x_1, \ldots, x_n (X \wedge T))[x'_1/x_1, \ldots, x'_n/x_n] \qquad (2)$$

toutes opérations que nous savons faire sur des BDD.

Calcul itératif sur les BDD

On calcule la suite $X_0, \ldots$ avec $X_0 = \Sigma_0$ et $X_{n+1} = \phi(X_n)$ et on s'arrête quand $X_n = X_{n+1}$ (test d'égalité = test de pointeur !).

Ça a l'air tout simple...
mais en fait il y a énormément d'optimisations possibles et implémenter une bibliothèque performantes de BDD est un travail énorme !

En pratique, on utilise aussi des opérateurs BDD "constrain", "restrict", des BDD « signés », et des variantes des BDD. Nous n'en parlerons pas dans ce cours.

On peut aussi s'arrêter dès qu'on touche un des états indésirables W.

Plan

Analyse en arrière

On a voulu calculer l'ensemble des états accessibles par $\rightarrow$ depuis les états initiaux Σ_0 pour savoir s'il touche un ensemble d'états indésirables W

On aurait pu calculer l'ensemble des états co-accessibles par $\rightarrow$ depuis les états W (= les états accessibles par la relation inverse $\leftarrow$) pour savoir s'ils touchent un état initial W.

σ co-accessible pour W = il existe une trace d'exécution de σ jusqu'à un état dans W.

Rapport avec le calcul de WP

W états indésirables, $\neg W$ = états corrects.

Weakest precondition de $\neg W$ = états qui ne peuvent arriver que dans $\neg W$.

Autrement dit : le complémentaire des états co-accessibles de W.

Rapport avec les invariants

Le plus petit invariant [inductif] au sens du calcul de WP = l'ensemble des états accessibles de la boucle.

Pourquoi ?

- Un invariant doit contenir Σ_0.
- Un invariant doit être stable par $\rightarrow$.

Autrement dit, notant $\phi(X) = \Sigma_0 \cup R(X)$,
$R(X) = \{y \in \Sigma \mid \exists x \in X \; x \rightarrow y\}$: $\phi(X) \subseteq X$.

Invariants

Un invariant c'est un ensemble X d'états tel que $\phi(X) \subseteq X$.

Si X et Y sont deux invariants, alors $X \cap Y$ aussi.

ϕ **monotone** pour l'ordre d'inclusion (si $X \subseteq Y$, alors $\phi(X) \subseteq \phi(Y)$) : pas surprenant, plus on a d'états au début, plus on peut en atteindre.

$\phi(X \cap Y) \subseteq \phi(X) \subseteq X$, idem pour Y, donc $\phi(X \cap Y) \subseteq X \cap Y$.

Ça marche aussi pour des intersections infinies.

Le plus petit invariant

Si je prends l'intersection de tous les invariants, j'obtiens **le plus petit invariant** (*strongest invariant*).

On montre que ce plus petit invariant vérifie $\phi(X) = X$ et que c'est le **plus petit point fixe** de ϕ.

Plan

Pour le *hardware*

Matériel = souvent spécifiable par état au *reset* + relation logique de transition sur des vecteurs de booléens.

Vérification de propriétés au moment du design de circuits (EDA) avant transfert vers le silicium (bug en production = extrêmement coûteux).

Outils comme Cadence-SMV (Kenneth McMillan, un des principaux auteurs des BDD, travaille chez Cadence).

Bounded model checking

C'est compliqué de calculer des invariants avec des BDD et ça prend beaucoup de mémoire (exponentiel dans le pire cas).

BMC : tester si on atteint les états illégaux en k coups.

Dérouler $\sigma_0 \to \sigma_1 \dots \sigma_k$ sous forme de formule logique F reliant les valeurs des booléens $x_1, \dots, x_n$ à toutes les étapes : formule dont les variables sont $(x_i^{(j)})$ pour $1 \leq i \leq n$ et $1 \leq j \leq k$.

Si on trouve une solution de $F \wedge \Sigma_0[x_1^{(1)}/x_1, \dots, x_n^{(1)}/x_n] \wedge W[x_1^{(k)}/x_1, \dots, x_n^{(k)}/x_n]$ on a une **trace d'exécution** en exactement k coups démarrant dans Σ_0 et finissant dans W.

SAT

Trouver une trace incorrecte = trouver une solution d'une formule logique (= valeur des variables qui rendent vraie la formule).

Problème **SAT** = problème **NP-complet** canonique.
Conjecture : impossible en temps polynomial en la taille de la formule.

En pratique : algorithmique souvent efficace (algorithmes DPLL avec *learned clauses*).

Teaser prochain cours

Et si les variables ne sont pas booléennes ?