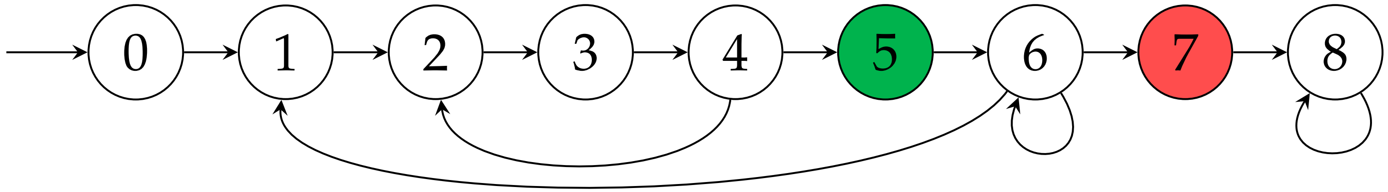
Plan

Vérification explicite et symbolique

- Vérification par énumération explicite
  - calculer le graphe des états accessibles du système
  - problème de passage à l'échelle à cause de l'explosion combinatoire
- Techniques « symboliques » de vérification
  - manipuler des ensembles d'états plutôt que les états individuels
  - représentation implicite d'ensembles d'états par des prédicats
- Mise en œuvre de la vérification symbolique
  - BDD (binary decision diagram)
    $\Rightarrow$ structure de données compacte
    $\Rightarrow$ forme canonique pour le calcul de points fixes
  - bounded model checking
    $\Rightarrow$ décrire les contre-exemples en logique propositionnelle
    $\Rightarrow$ bénéficier des progrès spectaculaires de techniques SAT/SMT

Bounded model checking : idée générale

- Considérer des préfixes d'exécutions de taille fixe k
  - existe-t-il un chemin de k états qui mène à un état rouge ?
  - existe-t-il un « lasso » de k états qui contient un état vert ?



- Mise en œuvre
  - coder l'existence de chemins en logique propositionnelle
  - faire appel à un solveur SAT pour déterminer si une solution existe
  - un modèle correspond à un chemin qui vérifie la propriété
  - si non satisfiable : il n'existe pas de chemin de taille k
    $\Rightarrow$ augmenter la borne k
  - technique utile si les chemins recherchés sont courts

Plan

1. Introduction
2. Systèmes de transitions et propriétés LTL
3. Bounded model checking
   - Satisfiabilité en logique propositionnelle
   - Vérifier l'accessibilité dans un système de transitions
   - Vérification de propriétés

Le problème SAT

- Rappel : logique propositionnelle
  - formules construites à partir de propositions et opérateurs booléens
  - interprétation : évaluation booléenne de propositions atomiques
  - φ satisfiable φ est vrai dans une certaine interprétation
  - φ valide φ est vrai dans toutes les interprétations
- Le problème SAT est décidable
  - méthode naïve : construire une table de vérité
  - complexité : $O(2^n)$ pour une formule contenant n propositions
  - le problème est NP-complet, tous les algorithmes connus sont exponentiels
- Mais : les solveurs SAT modernes sont très rapides en pratique

Digression : SAT pour sudoku (1)

SuDoku Puzzle

4				3	1		9	
			2			8	4	
	1	6					3	
	8	9			4			6
	7			6				9
			9	8		5		
5				7		1		4
7			6					2
		3			5		7	

SuDoku Puzzle

4	5	2	8	3	1	6	9	7
9	3	7	2	5	6	8	4	1
8	1	6	7	4	9	2	3	5
3	8	9	5	2	4	7	1	6
2	7	5	1	6	3	4	8	9
1	6	4	9	8	7	5	2	3
5	9	8	3	7	2	1	6	4
7	4	1	6	9	8	3	5	2
6	2	3	4	1	5	9	7	8

La solution est trouvée «instantanément» par un solveur SAT

- `http://www.cs.qub.ac.uk/~I.Spence/SuDoku/SuDoku.html`
- solveur SAT pour Java : `http://www.sat4j.org/`

Digression : SAT pour sudoku (2)

- Représenter le contenu de chaque case par une proposition
  - v_{ijk} vrai : la case (i, j) contient le chiffre k $(i, j, k = 1, \ldots, 9)$
- Clauses pour représenter les règles Sudoku
  - chaque case contient un chiffre unique

    $$\begin{array}{rl} v_{ij1} \vee \ldots \vee v_{ij9} & (i, j = 1, \ldots, 9) \\ \neg v_{ijk} \vee \neg v_{ijk'} & (i, j = 1, \ldots, 9,\ 1 \leq k < k' \leq 9) \end{array}$$

  - chaque chiffre apparaît une (seule) fois à chaque ligne

    $$\begin{array}{rl} v_{i1k} \vee \ldots \vee v_{i9k} & (i, k = 1, \ldots, 9) \\ \neg v_{ijk} \vee \neg v_{ij'k} & (i, k = 1, \ldots, 9,\ 1 \leq j < j' \leq 9) \end{array}$$

  - des clauses similaires pour représenter les autres contraintes
  - représenter les cases pré-remplies en énonçant que v_{ijk} doit être vrai
  - 729 variables, environ 12000 clauses

$\Rightarrow$ L'exemple atteste de l'efficacité des solveurs SAT modernes

Plan

Coder l’existence de chemins finis en logique propositionnelle

Un berger veut traverser une rivière avec un loup, une chèvre et un chou.
Il a une petite barque dans laquelle il peut transporter un seul animal ou le chou.

- Le loup mangera la chèvre si le berger n’est pas présent.
- La chèvre mangera le chou si le berger n’est pas présent.

Le berger, avec le loup, la chèvre et le chou, peuvent-ils traverser la rivière ?

Questions :

1. Coder ce problème par un système de transitions.
2. Définir une formule *danger* qui décrit les états indésirables.
3. Expliquer comment se servir d’un solveur SAT pour trouver la solution.

Encodage propositionnel du système de transitions

- Représenter l'état courant par 4 variables booléennes
  - p, w, g, c : position du berger, du loup, de la chèvre et du chou
  - vrai : rive de départ, faux : rive d'arrivée
- Coder des ensembles d'états par des formules propositionnelles
  - état initial : $\mathit{init}(p,g,w,c) \triangleq p \land g \land w \land c$
  - état cible : $\mathit{final}(p,g,w,c) \triangleq \neg p \land \neg g \land \neg w \land \neg c$
  - états dangéreux : $\mathit{danger}(p,g,w,c) \triangleq \begin{array}{l} \lor\ (w \Leftrightarrow g) \land (w \Leftrightarrow \neg p) \\ \lor\ (g \Leftrightarrow c) \land (g \Leftrightarrow \neg p) \end{array}$
- Transitions : formules sur deux copies des variables

$$\mathit{trans}(p,g,w,c,p',g',w',c') \triangleq \begin{array}{l} \land\ p' \Leftrightarrow \neg p \\ \land\ \begin{array}{l} \lor\ (g' \Leftrightarrow g) \land (w' \Leftrightarrow w) \\ \lor\ (g' \Leftrightarrow g) \land (c' \Leftrightarrow c) \\ \lor\ (w' \Leftrightarrow w) \land (c' \Leftrightarrow c) \end{array} \end{array}$$

Utiliser un solveur SAT pour trouver une solution

- Il existe une solution en k étapes s'il existe des états $s_0, \ldots s_k$ avec :
  - s_0 est un état initial, s_k est un état final,
  - aucun état s_i n'est dangéreux et
  - les états successifs (s_i, s_{i+1}) sont reliés par une transition

- Cette question peut être résolue pour des valeurs fixes de k
  - introduire $k+1$ copies p_i, g_i, w_i, c_i des variables propositionnelles
  - vérifier la satisfiabilité de la formule

$$
\begin{array}{l}
\wedge\ init(p_0, g_0, w_0, c_0) \wedge final(p_k, g_k, w_k, c_k) \\
\wedge\ \neg danger(p_0, g_0, w_0, c_0) \wedge \ldots \wedge \neg danger(p_k, g_k, w_k, c_k) \\
\wedge\ trans(p_0, g_0, w_0, c_0, p_1, g_1, w_1, c_1) \\
\wedge\ \ldots \\
\wedge\ trans(p_{k-1}, g_{k-1}, w_{k-1}, c_{k-1}, p_k, g_k, w_k, c_k)
\end{array}
$$

  - non satisfiable pour $k \leq 5$, mais satisfiable pour $k = 7, 9, \ldots$

Cas général : encoder les exécutions d'un système de transitions

- Encoder les états et les ensembles d'états
  - représenter les états par une liste $\vec{x}$ de variables propositionnelles
  - représenter un ensemble S d'états par une formule
  - exemples : condition initiale, invariants, contraintes, ...
- Encoder la relation de transition
  - définir un prédicat $\delta(\vec{x}, \vec{x}')$ sur deux copies de variables
- Cet encodage est souvent très naturel
  - exemple : codage d'un circuit par une formule propositionnelle
  - représenter des structures de données peut être compliqué

Plan

Bounded Model Checking : vérification d'invariants

- Existe-t-il un contre-exemple à l'invariant en k transitions ?
  - utiliser $k+1$ copies $\vec{x}_0, \ldots, \vec{x}_k$ des variables d'état
  - faire appel à un solveur SAT pour déterminer si la formule

    $$Init(\vec{x}_0) \land \delta(\vec{x}_0, \vec{x}_1) \land \ldots \land \delta(\vec{x}_{k-1}, \vec{x}_k) \land \neg Inv(\vec{x}_k)$$

    est satisfiable — si oui, le modèle fournit un contre-exemple

- Vérification de systèmes
  - effectuer la recherche de contre-exemples pour $k = 0, 1, \ldots$
  - taille maximale à considérer : « diamètre du système »

    $\Rightarrow$ longueur de chemin maximal dans le graphe sans répétition
  - en pratique : augmenter k jusqu'à l'épuisement des ressources (ou vérifier que le diamètre du système est atteint)

Exemple : algorithme de Peterson pour 2 processus (1)

algorithm Peterson {
  variables $turn \in \{0,1\}, req = [p \in \{0,1\} \mapsto \text{FALSE}]$;
  process $(proc \in \{0,1\})$ {
  nc : while (TRUE) {
      skip;
  set : $req[self] := \text{TRUE}; turn := 1 - self$;
  try : await $(turn = self) \lor \neg req[1 - self]$;
  cs : $req[self] := \text{FALSE}$;
} }

- Représenter l'état par des variables propositionnelles

$$\vec{x} \triangleq turn, req0, req1, nc0, set0, try0, cs0, nc1, set1, try1, cs1$$

- Encodage des états initiaux

$$\begin{aligned} Init(\vec{x}) \triangleq\ & \land \neg req0 \land \neg req1 \\ & \land nc0 \land \neg set0 \land \neg try0 \land \neg cs0 \\ & \land nc1 \land \neg set1 \land \neg try1 \land \neg cs1 \end{aligned}$$

Exemple : algorithme de Peterson pour 2 processus (2)

- Codage de la relation de transition

$$
\begin{array}{l}
\mathit{Next}(\vec{x},\vec{x}') \triangleq \\
\quad \lor \land nc0 \land \neg nc0' \land set0' \\
\quad \land \textsc{unchanged}(turn, req0, req1, try0, cs0, nc1, req1, try1, cs1) \\
\quad \lor \land set0 \land req0' \land turn' \land \neg set0' \land try0' \\
\quad \land \textsc{unchanged}(req1, nc0, cs0, nc1, set1, try1, cs1) \\
\quad \lor \land try0 \land (\neg turn \lor \neg req1) \land \neg try0' \land cs0' \\
\quad \land \textsc{unchanged}(turn, req0, req1, nc0, req0, nc1, set1, try1, cs1) \\
\quad \lor \land cs0 \land \neg req0' \land \neg cs0' \land nc0' \\
\quad \land \textsc{unchanged}(turn, req1, set0, try0, nc1, set1, try1, cs1) \\
\quad \lor \text{(* symétrique pour les transitions du processus 1 *)}
\end{array}
$$

NB : $\textsc{unchanged}(v, \ldots, w)$ représente $v' \Leftrightarrow v \land \cdots \land w' \Leftrightarrow w$

- Invariant $\quad \mathit{Inv}(\vec{x}) \triangleq \neg(cs0 \land cs1)$
- Vérification pour $k = 10$: décider la satisfiabilité de

$$\mathit{Init}(\vec{x}_0) \land \mathit{Next}(\vec{x}_0, \vec{x}_1) \land \ldots \land \mathit{Next}(\vec{x}_9, \vec{x}_{10}) \land \neg \mathit{Inv}(\vec{x}_{10})$$

Vérification d'invariants inductifs

- Rappel : invariant inductif *Inv*
  - *Inv* est vrai dans tous les états initiaux …
  - … et préservé par toute transition du système
  - de tels invariants sont à la base de la vérification déductive

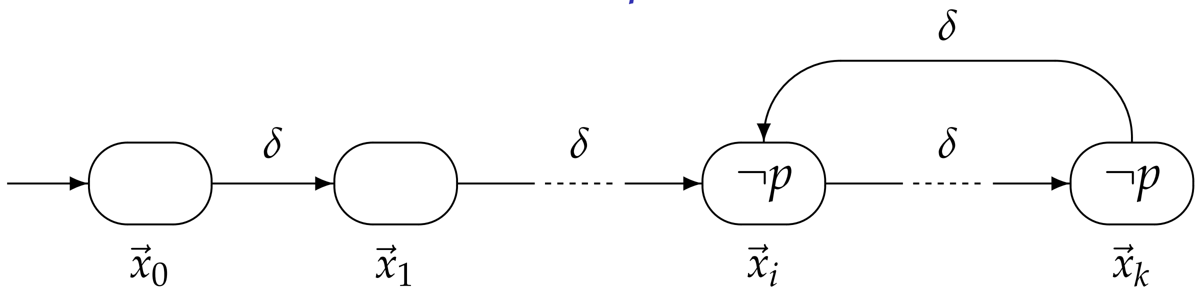
- Vérification automatique (sur des modèles finis)
  - déterminer que les formules suivantes ne sont pas satisfiables :

    $Init(\vec{x}) \land \neg Inv(\vec{x})$ et $Inv(\vec{x}) \land Next(\vec{x}, \vec{x}') \land \neg Inv(\vec{x}')$

- Ceci établit la validité de *Inv* dans tous les états accessibles
  - le prédicat reste vrai pendant toute exécution, même infinie
  - la vérification ne nécessite que deux copies de variables d'états
  - technique utile de validation avant une preuve interactive

Vivacité : vérification de **G F** p

- Bounded model checking de formules temporelles, p.ex. **G F** p
- Idée : recherche d'un «lasso» vérifiant **F G** $\neg p$



- Encodage en logique propositionnelle
  - déterminer la satisfiabilité de la formule suivante :

$$\begin{array}{ll} & Init(\vec{x}_0) \wedge \delta(\vec{x}_0, \vec{x}_1) \wedge \ldots \wedge \delta(\vec{x}_{k-1}, \vec{x}_k) \\ \wedge & \displaystyle\bigvee_{i=0}^{k} \delta(\vec{x}_k, \vec{x}_i) \wedge \neg p(\vec{x}_i) \wedge \neg p(\vec{x}_{i+1}) \wedge \ldots \wedge \neg p(\vec{x}_k) \end{array}$$

  - le paramètre k détermine la taille du «lasso»
- Cette idée peut être généralisée à la vérification de formules LTL

Résumé : bounded model checking

- Références et outils
  - E.M. Clarke, A. Biere, R. Raimi, Y. Zhu : *Bounded Model Checking Using Satisfiability Solving*. Formal Methods in System Design 19(1) : 7-34 (2001)
  - nuXmv : `http://nuxmv.fbk.eu/` $\Rightarrow$ hardware, systèmes synchrones
  - Apalache : `https://apalache.informal.systems` $\Rightarrow$ spécifications TLA$^+$
  - CBMC : `http://www.cprover.org/cbmc/` $\Rightarrow$ vérification de programmes C
- Etat de l'art
  - technique utilisée très couramment pour la vérification de circuits
  - des compétitions annuelles d'outils (SAT/SMT et vérification)
  - peut fonctionner jusqu'à des centaines de milliers de variables, profondeur typique : quelques dizaines de transitions