Vérification des systèmes dynamiques finis

Pascal.Raymond@univ-grenoble-alpes.fr

3A SLE – Validation des systèmes embarqués

Sommaire

1. Introduction

Systèmes dynamiques

Qu'est-ce c'est ? / où les trouve-t-on ?

- Systèmes réactifs, contrôle/commande, embarqué

Comment sont-ils réalisés ?

- Architecture et OS spécialisés (souvent)
- Programmes séquentiels (C, assembleur), automates (petits systèmes)
- + Programmation multitâche/concurrente (gros systèmes)

Vérification (a priori)

Pourquoi faire ?

- Systèmes critiques
- Débogage difficile/impossible
- Méthodes de type ≪ essayer/corriger ≫ pas possibles (ni souhaitables)
- $\Rightarrow$ Besoin de vérification *a priori*, test, analyse statique

Qu'est-ce qu'on peut chercher à vérifier ?

Les exécutions seront toujours correctes :

- le système fonctionne correctement (pas de ≪ runtime error ≫),
- le système calcule les bonnes réactions (fonctionnalité),
- le système calcule assez vite (aspect temps-réel).

Toutes ces *qualités* dépendent :

- de la conception/programmation du système,
- mais aussi de l'exécutif (ressources disponibles en mémoire et calcul).

Précisons ...

(Au moins) 3 notions de correction

- propriétés génériques, e.g. non débordement arithmétique, déterminisme/absence de blocage, de famine (concurrence)
- propriétés spécifiques (i.e. prop. fonctionnelles), e.g. réalise le comportement attendu
- propriétés ≪ runtime ≫ : temps-réel, non débordement mémoire etc.
  dépend de l'exécutif (matériel/logiciel de base) ...
  ... mais aussi de la conception : utilisation de ressources non bornées (e.g. allocation dynamique)

Dans ce cours :

- *On s'intéresse uniquement aux propriétés fonctionnelles.*
- *(Mais les autres thèmes sont aussi très importants !)*
- *Pour raisonner sur la fonctionnalité, il faut un* modèle *mathématique du (fonctionnement du) système.*

Modèle fonctionnel

- S'affranchir du système concret pour raisonner sur ce qu'il fait (i.e. sémantique)
- L'exécution d'un système réactif vu par un observateur extérieur, est essentiellement une séquence de réactions entrées/sorties :
  - ↪ si indéterminisme, on a une relation e/s
  - ↪ si déterminisme, on a une fonction $e \rightarrow s$

Dans ce cours :

On se place dans un cas favorable où la fonctionnalité est bien définie : programmation concurrente déterministe (programmation synchrone, cf. le langage Lustre).

Remarque :

On pourrait aussi considérer des systèmes basés sur la concurrence asynchrone, mais il est alors (beaucoup) plus difficile d'extraire le modèle fonctionnel, si tant est qu'il existe (indéterminisme, blocage etc)

Plan du cours

- Présentation (rappel ?) du langage Lustre
- Exprimer/vérifier des propriétés dynamiques
  - ↪ quels sont les classes de propriétés ?
  - ↪ limitations théoriques ?
- Abstraction finie
  - ↪ comment aborder les systèmes complexes/infinis/de grande taille
  - ↪ abstraction conservative et preuve partielle
- Techniques d'exploration des systèmes finis
  - ↪ énumératif/symbolique
  - ↪ techniques de décision
- Des exemples, des exercices

2. Le langage Lustre

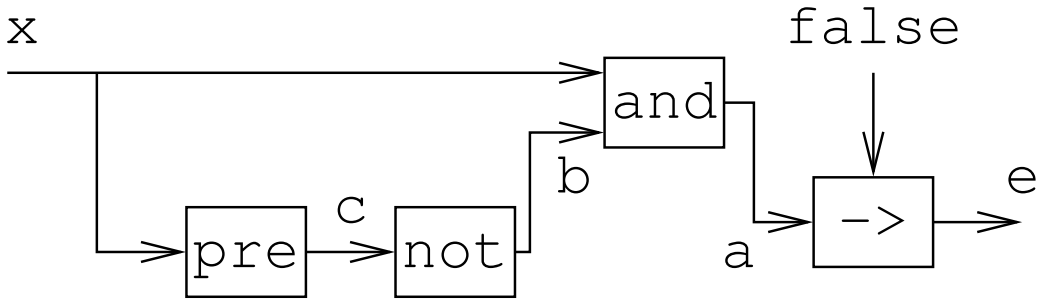
Pourquoi utiliser ce langage ?

- Langage relativement simple et sémantiquement ≪ propre ≫
- Représentatif de ce qui se fait dans les domaines critiques, avionique et nucléaire (cf. Scade)
- On dispose des outils (académiques) pour expérimenter

Exemple : détecteur de fronts montants

Version graphique

Style flot-de-données, très proche du modèle des circuits synchrones

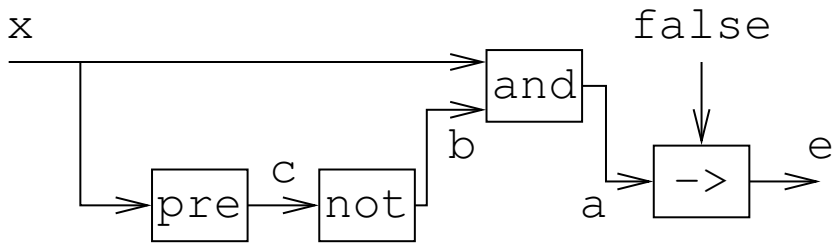


- des flots de valeurs circulent sur les fils
- rythmés par une horloge discrète ($t = 0, 1, 2 \cdots$)
- opérateurs classiques, e.g. $a_t = x_t \wedge b_t$
- mémoire (i.e. registre), e.g. $c_0 =$ nil et $\forall t \neq 0 \;\; c_t = x_{t-1}$
- initialisation, e.g. $e_0 =$ false et $\forall t \neq 0 \;\; e_t = a_t$

Remarque : similaire aux circuits séquentiels (ou synchrones).

Version textuelle (Lustre)

```
node edge(x:bool)returns(e:bool);
var a,b,c:bool;
let
   e = false -> a;
   a = x and b;
   b = not c;
   c = pre x;
tel
```



- ensemble d'équations (et non pas séquence d'affectations !)
- l'ordre des équations est indifférent
- on peut utiliser ou pas les variables intermédiaires, e.g. :

```
node edge(x : bool) returns (e : bool);
let
   e = false -> x and not pre x;
tel
```

Le langage en un transparent

Types et opérateurs

- types prédéfinis : **`bool`**, **`int`**, **`real`**
- tous les opérateurs arithmétiques et logiques classiques :
  `and`, **`or`**, **`not`**, **`+`**, **`-`**, **`*`**, **`/`**, **`if-then-else`** etc
- opérateur **`pre`** (previous = registre)
- opérateur **`->`** (flèche = initialisation)

Exemples de flots

```
p = false -> not pre p;
n = 0 -> pre n + 1;
m = i -> if (i < pre m) then i else pre m;
```

Exo. :

Donner les premières valeurs des flots ***p****,* ***n****,*

et du flot ***m*** *pour* ***i*** $= 4, 8, 5, -1, 0, 12, \cdots$ *?*

Exemple : l'automate à deux états

Brique de base très utile :

- une entrée **on**, une entrée **off**
- une sortie **s** :
  - ↪ passe de faux à vrai si **on**,
  - ↪ passe de vrai à faux si **off**
  - ↪ initialisée à **orig** à l'origine des temps

```
node switch(orig, on, off : bool) returns (s : bool);
var ps : bool;
let
   ps = orig -> pre s;
   s = if ps then not off else on;
tel
```

Exemple numérique : compteur plus/minus

- une entrée **`plus`**, une entrée **`minus`**
- une sortie **`cpt`** :
  - ↪ incrémentée si **`plus`**,
  - ↪ décrémentée si **`minus`**
  - ↪ initialisée à 0 à l'origine des temps

```
node counter(plus, minus : bool) returns (cpt : int);
let
   cpt = (0 -> pre cpt) +
         (if plus then 1 else 0) +
         (if minus then -1 else 0);
tel
```

Un exemple complet : compteur de balises

Spécification

Un train circule sur une voie où sont placées des balises fixes, de proche en proche. Un système informatique embarqué dans le train reçoit :

- un signal *balise* chaque fois que le train croise une balise,
- un signal *seconde* chaque fois qu'une seconde s'est écoulée.

Sachant que la vitesse ≪ idéale ≫est d'environ 1 balise par seconde, le système doit calculer (assez grossièrement) si le train est :

- à l'heure (on reçoit à peu près autant de secondes que de balises),
- en avance (on reçoit plus de balises que de secondes),
- en retard (on reçoit plus de secondes que de balises).

Hystérésis

Pour éviter les oscillations, on utilise un mécanisme classique *d'hystérésis*, c'est-à-dire de seuils de tolérance décalés :

- on est à l'heure tant que la différence balise/seconde n'excède pas 3
- pour repasser à l'heure, la différence doit être revenue à moins de 1

Exo. :

Proposer un programme Lustre qui réutilise les nœuds déjà vus.

Une solution

```
node speed(b,s:bool) returns (late, ontime, early: bool);
var cpt : int;
let
    cpt = counter(b,s);
    early = switch(false, cpt > 3, cpt <= 1);
    late = switch(false, cpt < -3, cpt >= -1);
    ontime = not (early or late);
tel
```

3. Définir et spécifier les propriétés

Qu'est-ce qu'une propriété fonctionnelle ?

Définition générale

- Définition générale : une propriété fonctionnelle $=$ un ensemble de comportements (corrects)
- Vérifier la propriété $=$ vérifier que l'ensemble des comportements possibles est inclus dans l'ensemble des comportements corrects.

Propriété et théorie des langages

- Problème proche de la théorie des langages : inclusion de langages
- Mais attention : langages infinitaires
  - $\hookrightarrow$ $\neq$ langages classiques finitaires
  - $\hookrightarrow$ e.g. $\ll$ il est inévitable que ... $\gg$ n'a de sens que pour des mots infinis.

Propriétés et théorie des langages (suite)

- Théorie et outils existent pour traiter les langages infinitaires :
  - ↪ expressions omega-régulières, automates de Büchi ...
  - ↪ mais c'est relativement complexe et coûteux (algorithmiquement)
- Peut-on s'en passer ?
  - ↪ Non si on veut toute la puissance d'expression (cf. ≪ inévitable ≫)
  - ↪ Oui, si on peut se contenter d'une classe de propriétés plus simple
  - ↪ Invariants de programmes ≪ aka ≫ propriétés de sûreté

Propriétés d'invariance (sûreté)

Définitions

Propriété d'état :

- *Une configuration (ou état) d'un système est une valuation particulière des entrées, sorties et mémoires internes du système.*
- *Une propriété d'état est une relation (i.e. un ensemble) de configurations.*

Propriété de sûreté :

- *Une propriété de sûreté exprime le fait qu'une propriété d'état est invariante au cours du temps.*
- *Ou, de manière équivalente, qu'une configuration (redoutée) n'arrive jamais*
- *n.b. le mot anglais est ≪ safety ≫*

Sûreté versus vivacité

Dans la théorie des propriétés temporelle, on distingue deux grandes classes :

- Sûreté (safety) : elles expriment l'impossibilité d'une configuration.

  Exemples sur le programme **`speed`** :

  ↪ *On ne peut pas être en retard et en avance.*

  ↪ *On ne peut pas passer directement d'en retard à en avance.*

  ↪ *On ne peut pas rester en retard un seul instant.*

- Vivacité (liveness) : elles expriment la *possibilité*, voire *l'inévitabilité* d'une configuration.

  Exemples sur le programme **`speed`** :

  ↪ *On peut rester en retard un seul instant* (n.b. c'est faux !).

  ↪ *Si le train ne roule pas, il finira par être en retard.*

Intérêt/spécificités des propriétés de sûreté

- Avantage par rapport à *l'inévitabilité* :
  - ↪ ne font jamais référence au futur non borné
  - ↪ pas besoin de raisonner sur des séquences *infinies*,
  - ↪ on peut raisonner sur des séquences *arbitrairement longues* (ce qui est tout à fait classique)
- Avantage par rapport à *la possibilité* :
  - ↪ vérifier l'impossibilité permet de faire des approximations *bien fondées*
  - ↪ notion d'abstraction conservative (on y reviendra précisément)

Dans ce cours :

On se limite volontairement à l'expression et la vérification de propriétés de sûreté, i.e. des invariants de programmes.

Exprimer les propriétés de sûreté

Relations simples

Relation toujours vraie entre entrées et/ou sorties,

- la sortie x est toujours positive,
- si l'entrée e est vraie, alors la sortie x est vraie
- sur **`speed`** : jamais **`early`** et **`late`** en même temps

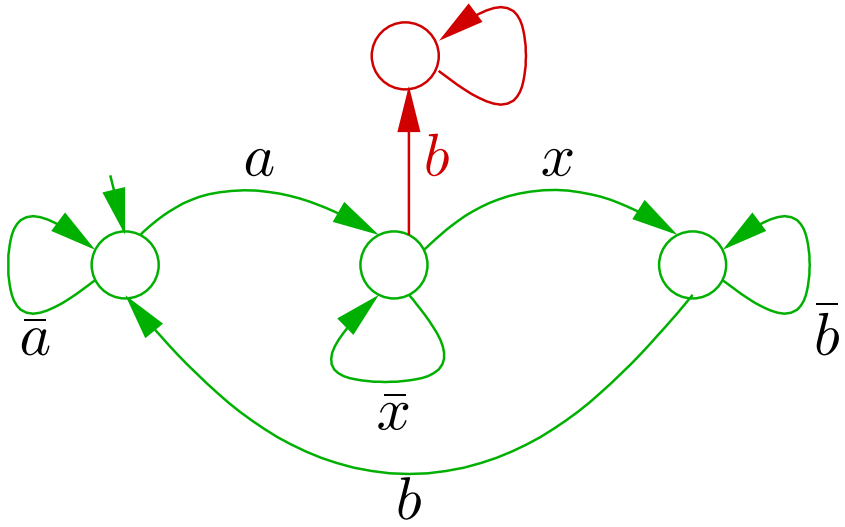
Un simple langage $\ll$ logique $\gg$ suffit, par exemple avec la syntaxe Lustre :

```
prop1 = (x > 0);
prop2 = (e => x);
prop3 = not (early and late);
```

Relations temporelles

Plus sophistiquées : dépendent de ce qui s'est passé *avant* ; nécessitent de la mémoire.

- toujours au moins une fois x entre a et b
- C'est un *langage*, exprimable avec un *automate* :



- ou avec un programme Lustre :

```
waitx = switch(false, a, x or b);
error = switch(false, waitx and b, false);
prop = not error;
```

Relations temporelles (suite)

Autre exemple, tiré de l'exemple **`speed`** :

- on ne peut pas passer directement d'en retard à en avance
- exo : proposer un automate ?
- exo : proposer un programme Lustre ?

Expression des propriétés par des programmes : notion d'*observateurs*

Observateurs

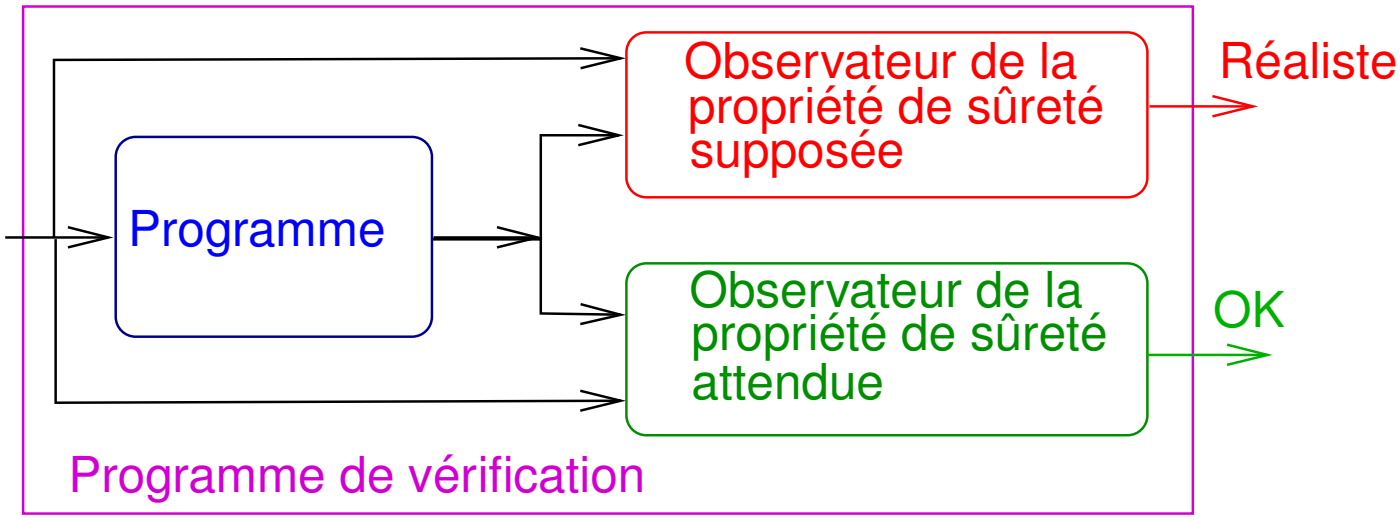
- Toute propriété de sûreté peut être exprimée par un programme :
  - ↪ qui *observe* (reçoit) les entrées/sorties du système à valider,
  - ↪ répond ≪ ok ≫ aussi longtemps que le comportement est correct
- De manière équivalente, on peut aussi définir des observateurs *négatifs*, qui émettent une erreur (≪ ko ≫) dès que le comportement observé est incorrect
  - ↪ ok ou ko : c'est juste une question de convention

Hypothèses et propriétés

- En général, un système réactif n'est pas censé fonctionner correctement si les entrées font *n'importe quoi*
- Il est nécessaire de prendre en compte des hypothèses, par exemple :
  - ↪ bornes de variation des entrées numériques,
  - ↪ exclusivité des entrées, e.g. on ne peut pas avoir x et y en même temps,
  - ↪ voire relations temporelles complexes (scénario)
- Hypothèse ET propriété sont des *sûretés*

Programme de preuve

- Programmer un observateur de la propriété de sûreté supposée (réaliste).
- Programmer un observateur de la propriété attendue (ok).
- Forme générale :



- Propriété temporelle complète : *(toujours réaliste) implique (toujours ok)*

Dans ce cours :

On utilisera le même langage (Lustre) pour écrire les systèmes à vérififier et les observateurs.

4. Modèle des systèmes réactifs et abstraction finie

Modèle équationnel

Définition

Soit un système réactif, caractérisé par des entrées E et des sorties S, et un ensemble de mémoires internes M, alors, si ce système est déterministe, il peut être modélisé par un n-uplet (E, S, M, M_0, f, g) où :

- M_0 est la valeur initiale de la mémoire,
- $S_t = f(E_t, M_t)$ est la *fonction* de sortie,
- $M_{t+1} = g(E_t, M_t)$ est la *fonction* de transition.

Remarque :

- quand on part d'un programme Lustre, l'extraction du modèle (E, S, M, M_0, f, g) est triviale,
- beaucoup moins quand on part d'un système multi-thread, asynchrone :
  - ↪ déterminisme difficile/impossible à établir
  - ↪ modèle *relationnel* et non pas fonctionnel

Une exécution

Pour une séquence d'entrées $E_0, E_1, \cdots$, on obtient :

- $1^{\text{ère}}$ réaction : $S_0 = f(E_0, M_0), M_1 = g(E_0, M_0)$
- $2^{\text{ème}}$ réaction : $S_1 = f(E_1, M_1), M_2 = g(E_1, M_1)$
- etc

Autre notation : chaîne de *transitions*

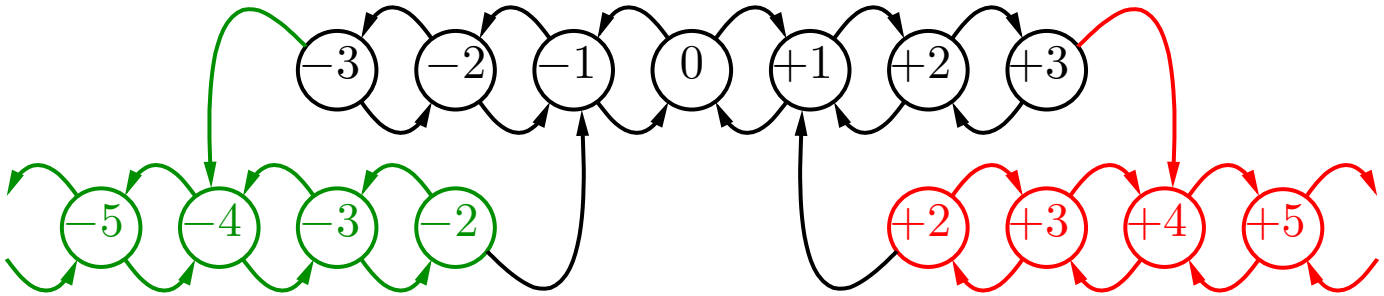
$$M_0 \overset{E_0/S_0}{\longrightarrow} M_1 \overset{E_1/S_1}{\longrightarrow} M_2 \overset{E_2/S_2}{\longrightarrow} M_3 \overset{E_3/S_3}{\longrightarrow} \cdots$$

Toutes les exécutions

- Ensemble de toutes les chaînes de transitions pour toutes les séquences d'entrées possibles.
- Forment un *arbre*, avec M_0 pour racine commune, et les M_i comme nœuds ...
- ou plutôt un *graphe*, car on peut retrouver la même M_i au cours de différentes exécutions.

Graphe des exécutions

- Exemple du compteur de balises :



- Vérification : raisonner sur le modèle équationnel (automate implicite) et/ou le graphe des exécutions (automate explicite)

Cas d'un programme de vérification

Uniquement deux sorties :

- $\Phi(M_t, E_t) \rightarrow$ **B** (fonction $\ll$ propriété $\gg$)
- $H(M_t, E_t) \rightarrow$ **B** (fonction $\ll$ hypothèse $\gg$)

Il faut démontre/établir que $\forall E_0, \cdots, E_n$

- soit la séquence $M_0, M_1, \cdots, M_n$ avec $M_{t+1} = g(M_t, E_t)$,
- alors $(\forall t \;\; H(M_t, E_t)) \Rightarrow (\forall t \;\; \Phi(M_t, E_t))$

Limites théoriques

Décidabilité

Le problème fixé est *indécidable en général*, et ceci à deux niveaux :

- intemporel : déterminer si une transition est possible est indécidable.
  (i.e. résoudre $H(M_t, E_t) \Rightarrow \Phi(M_t, E_t)$ pour (M_t, E_t) donné).
  - ↪ Classique : indécidabilité de la théorie des nombres
  - ↪ Cas particuliers décidables :
    - variables purement logiques (algèbre booléenne)
    - relations numériques linéaires (algèbre linéaire réelle, arithmétique de Presburger).
- temporel : déterminer si une configuration M_t est accessible est indécidable.
  - ↪ En particulier : même si on n'a que des relations linéaires, le problème est indécidable en général
  - ↪ il existe quelques cas particuliers décidables, avec des variables numériques (compteurs avec +1 et remise à 0)

↪ Cas trivialement décidable : système (dont le graphe des exécutions est) fini.

Systèmes finis

- manipulent un ensemble *fini* de variables à valeurs *finie*
- n.b. on peut toujours se ramener à des systèmes *strictement booléens* (on verra ça + en détails)

Dans ce cours :

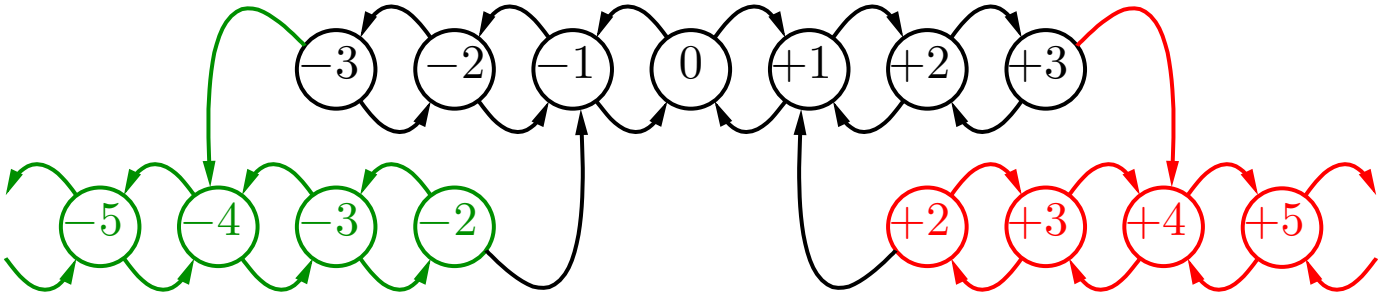
On se concentre sur l'étude/la vérification des systèmes finis, essentiellement des systèmes purement logiques.

Remarque sur la finitude

- Tout système informatique à l'exécution est fini !
- Mais en pratique, pas de différence entre ≪ gros ≫ et ≪ infini ≫
- e.g. un (petit) processeur avec quelques milliers de registres logiques est un système fini, mais avec $2^{\text{qq milliers}}$ d'états potentiels !
  (remarque : le nombre d'atomes dans l'univers est estimé à $\simeq 2^{266}$)
- On est, en pratique, limité à l'étude des ≪ petits ≫ systèmes finis

Exemple : compteur de balises

- Rappel, le graphe est ≪ virtuellement ≫ infini :



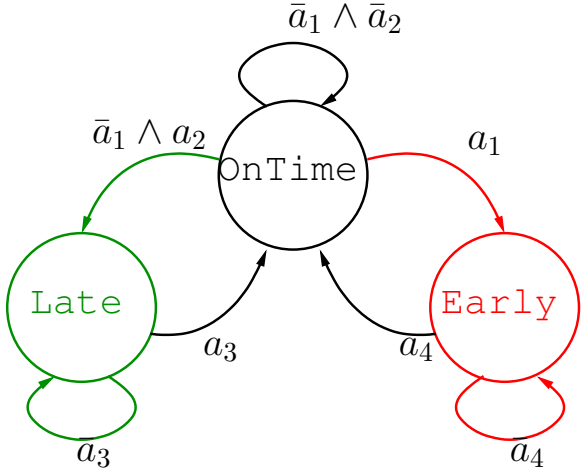
- Abstraction de la variable entière **`cpt`**, les conditions numériques sont remplacées par des variables logiques *libres et indépendantes* :

⋆ a_1 pour `cpt > 3`

⋆ a_2 pour `cpt < -3`

⋆ a_3 pour `cpt >= -1`

⋆ a_4 pour `cpt <= 1`



Abstraction et propriété

- Le graphe fini est *une simplification* du vrai graphe
- Il y a *perte d'information*
- Qu'est-ce qu'on peut espérer vérifier malgré cette perte ?

Propriété	système concrêt	abstraction
Jamais **`Late`** et **`Early`**	vrai	vrai
Jamais passer de **`Late`** et **`Early`**	vrai	vrai
Jamais un seul instant **`Late`**	vrai	faux
Possible un seul instant **`Late`**	faux	vrai

Conclusion :

- propriété perdue : pas grave (normal, preuve partielle)
- propriété introduite : grave (à proscrire !)

Attention à ce qu'on peut espérer vérifier !

Abstraction conservative

Définition :

Une abstraction est conservative *pour une classe de propriétés ssi, pour toute propriété P de la classe :*

l'abstraction satisfait P implique le système satisfait P

(En général, la réciproque est fausse)

Sur-approximation

- les comportements de l'abstraction sont un sur-ensemble de ceux du système
- abstraction finie $=$ un cas parmi d'autre de sur-approximation
  (cf. interprétation abstraite)
- Toute sur-approximation est conservative pour les propriétés de sûreté
  (si un comportement n'existe pas dans le sur-ensemble, il n'existe pas dans le sous-ensemble).
- c'est une des raisons pour lesquelles on se limite aux sûretés.

5. Exploration des modèles finis

Automate booléen (implicite)

- on suppose qu'une (éventuelle) abstraction a été faite
- on a :
  - ↪ un ensemble de mémoires (variables d'état) M à valeur dans $\mathbf{B}^{|M|}$
  - ↪ un ensemble d'entrées (variables libres) V à valeur dans $\mathbf{B}^{|V|}$
  - ↪ des états initiaux caractérisés par Init $: \mathbf{B}^{|M|} \rightarrow \mathbf{B}$

    (en général, on en a un seul)
  - ↪ des fonctions de transitions $g_1, \cdots g_{|M|}$, avec chaque $g_k : \mathbf{B}^{|M|} \times \mathbf{B}^{|V|} \rightarrow \mathbf{B}$
  - ↪ Une hypothèse $H : \mathbf{B}^{|M|} \times \mathbf{B}^{|V|} \rightarrow \mathbf{B}$
  - ↪ Une propriété $\phi : \mathbf{B}^{|M|} \times \mathbf{B}^{|V|} \rightarrow \mathbf{B}$

Automate explicite

Développement du modèle équationnel sous forme de graphe.

- Ensemble des états = valuations des M : $Q = \mathbf{B}^{|M|}$
- Ensemble des valuations des V : $\Sigma = \mathbf{B}^{|V|}$
- Etat initial : $\{q | \text{Init}(q)\}$
- Relation de transition : $T \subseteq Q \times \Sigma \times Q$ défini par

  $(q, v, q') \in T \quad \Leftrightarrow \quad \forall k \; q'_k = g_k(q, v)$

  On note $q \xrightarrow{v} q'$ (q' est le successeur de q par v)

  ↪ n.b. Déterminisme : c'est en fait une fonction $T \subseteq Q \times \Sigma \to Q$
- Ensemble des transitions réalistes $\{q \xrightarrow{v} q' | H(q, v)\}$

- Ensemble des transitions correctes $\{q \xrightarrow{v} q' | \Phi(q, v)\}$

Exploration du modèle

États remarquables

- État initiaux : Init
- États d'erreurs : Err $= \{q / \exists v \quad (q,v) \in H \wedge (q,v) \notin \Phi\}$

Fonctions pre et post

$$\begin{aligned} \text{post} : Q &\rightarrow 2^Q \\ q &\mapsto \{q' / \exists v \;\; H(q,v) \;\wedge\; q \xrightarrow{v} q'\} \\ \text{pre} : Q &\rightarrow 2^Q \\ q &\mapsto \{q' / \exists v \;\; H(q',v) \;\wedge\; q' \xrightarrow{v} q\} \end{aligned}$$

Généralisation aux ensembles d'états :

- Post : $2^Q \rightarrow 2^Q$ avec $\text{Post}(X) = \bigcup_{q \in X} \text{post}(q)$
- Pre : $2^Q \rightarrow 2^Q$ avec $\text{Pre}(X) = \bigcup_{q \in X} \text{pre}(q)$

Définition de Acc (états *accessibles*)

le plus petit ensemble qui vérifie Acc $=$ Init $\cup$ Post$($Acc$)$

Notation $\ll$ point fixe $\gg$: Acc $= \mu X.(X =$ Init $\cup$ Post$(X))$

Définition de Bad (mauvais états)

le plus petit ensemble qui vérifie Bad $=$ Err $\cup$ Pre$($Bad$)$

Notation $\ll$ point fixe $\gg$: Bad $= \mu X.(X =$ Err $\cup$ Pre$(X))$

Calcul de points fixes

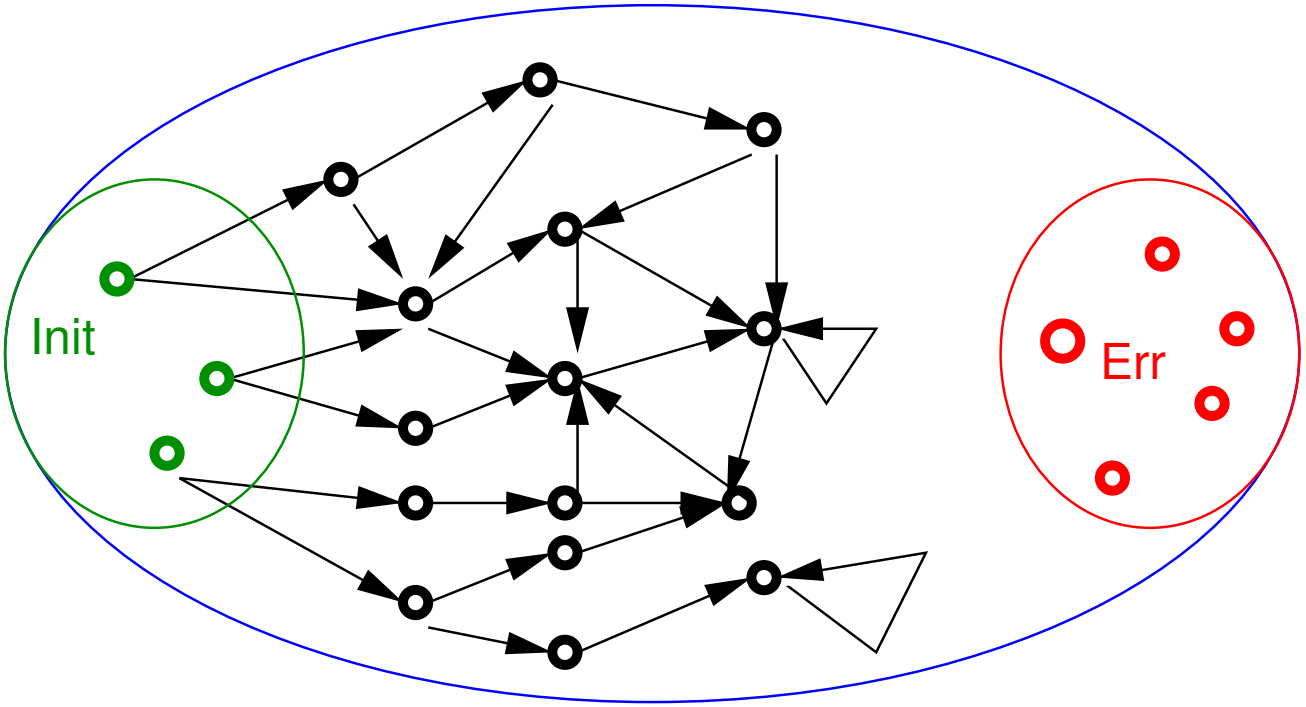
- Treillis finis, fonctions monotones : les solutions existent et sont trivialement calculables.

L'exploration à pour but (indifféremment) :

- de vérifier que Acc $\cap$ Err $= \emptyset$: méthode en avant
- de vérifier que Bad $\cap$ Init $= \emptyset$: méthode en arrière

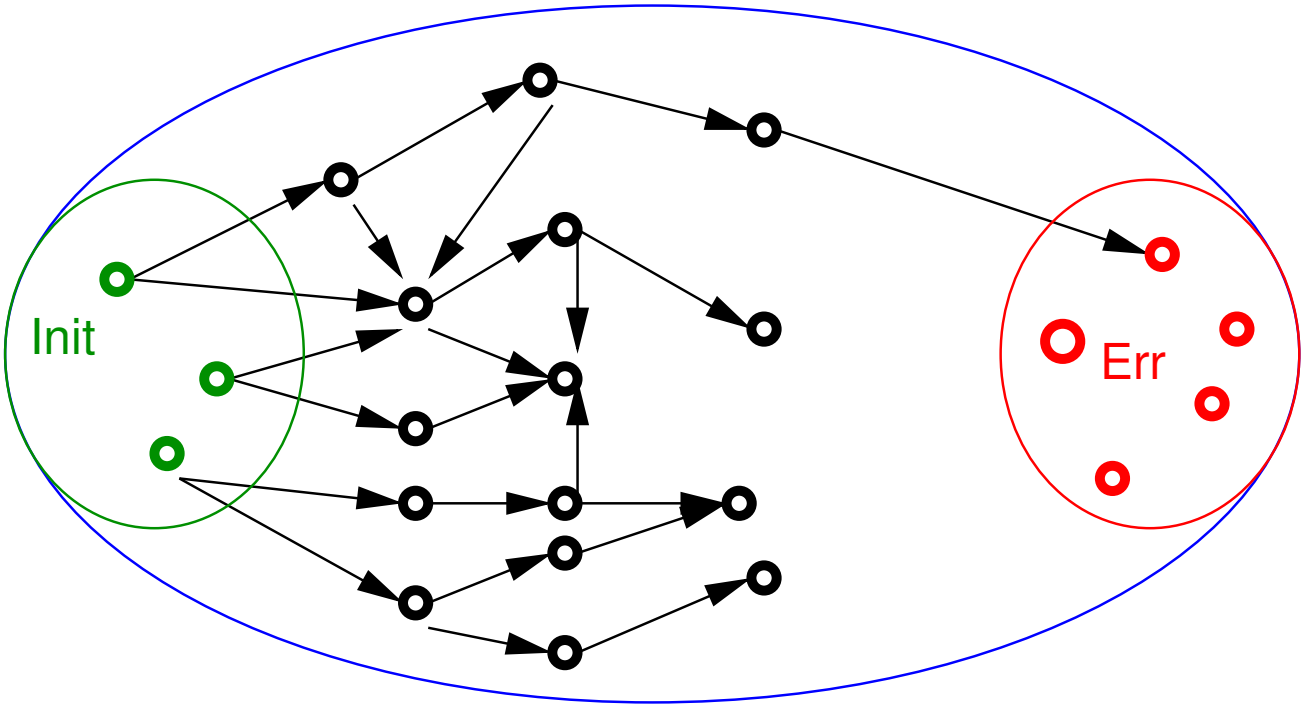
Remarque : le cas fonctionnel crée une dissymétrie entre les deux méthodes

Énumératif en avant : le cas où ça réussit ...



La preuve réussit

Énumératif en avant : le cas où ça échoue...



La preuve échoue !

Algorithme général ≫en avant ≫

```
CurAcc := Init
Traités := vide
tant qu'il existe q dans CurAcc - Traités faire {
      (* q ∈ CurAcc \ Traités *)
      pour tout q' dans post(q) faire {
            si q' dans Err SORTIR(échoue)
            mettre q' dans CurAcc
      }
      mettre q dans Traités
}
(* on a CurAcc = Traités = Acc *)
SORTIR(réussi)
```

Remarques

- Algo énumératif en arrière possible :
  - ↪ utiliser pre au lieu de post,
  - ↪ inverser les rôles de Init et Err

  ... mais pas vraiment utilisé en pratique
- Élément d'implémentation :
  - ↪ parcours en profondeur, largeur, etc.
  - ↪ codage minimal des états (e.g. chaînes de bits)
  - ↪ hash-code pour garantir l'unicité

Exemple/TD : exercice ≪ un peu d'arithmétique ≫(cf.9.2)

Limitation des méthodes énumératives

Généralités

- La complexité est intrinsèquement liée au nombre d'états
- Le nombre d'états est intrinsèquement exponentiel...

(Intuition) il existe des systèmes avec :

- beaucoup d'états
- mais qui sont malgré cela ≪ simples ≫

Méthodes symboliques ?

- Raisonner *globalement* sur des ensembles d'états
- La complexité n'est pas forcément lié à la taille de l'ensemble, par exemple :
  - ↪ 3 mémoires x, y, z, ensemble des états où x est vrai ou y est vrai :
    - 6 états ...
    - ... mais exprimable par une petite formule $x \vee y$
  - ↪ le même avec 1, 2, 3, etc mémoires en plus :
    - 12, 24, 48 états ...
    - ... mais toujours la même formule $x \vee y$
  - ↪ cas extrème, n mémoires, l'ensemble de tous les états :
    - 2^n états ...
    - ... formule très simple $1 =$ la formule toujours vraie !

Coder/manipuler des formules logiques

- Besoins : manipulation $\ll$ efficace $\gg$des formules
  - $\hookrightarrow$ opérations logiques $=$ opérations ensemblistes ($\cup$ = $\vee$, $\cap$ = $\wedge$, etc)
  - $\hookrightarrow$ décider si un ensemble est vide (i.e. formule toujours fausse)
- Binary Decision Diagrams (BDD) :
  - $\hookrightarrow$ Représentation *canonique* des formules :
    - $*$ En particulier : f est vide ssi $bdd(f) = 0$ ...
    - $*$ donc la décision a un coût $\ll$ nul $\gg$
  - $\hookrightarrow$ ... mais, la taille (et le temps de construction) de $bdd(f)$ est exponentiel dans le pire des cas par rapport à la taille de f

Dans ce cours :

Ouvront une parenthèse pour voir un peu plus en détail ce qu'est un BDD

6. Graphes binaires de décision (BDD)

Décomposition de Shannon

Décomposition en si-alors-sinon

- Soit une fonction d'arité n, $f \in \mathbb{B}^n \rightarrow \mathbb{B}$,
- il existe deux fonctions uniques d'arité $n-1$,

  $\hookrightarrow f_x = f(1, y, ..., z) \in \mathbb{B}^{n-1} \rightarrow \mathbb{B}$

  $\hookrightarrow f_{\bar{x}} = f(0, y, ..., z) \in \mathbb{B}^{n-1} \rightarrow \mathbb{B}$
- telles que : $f = x.f_x + \bar{x}.f_{\bar{x}}$

Exemple

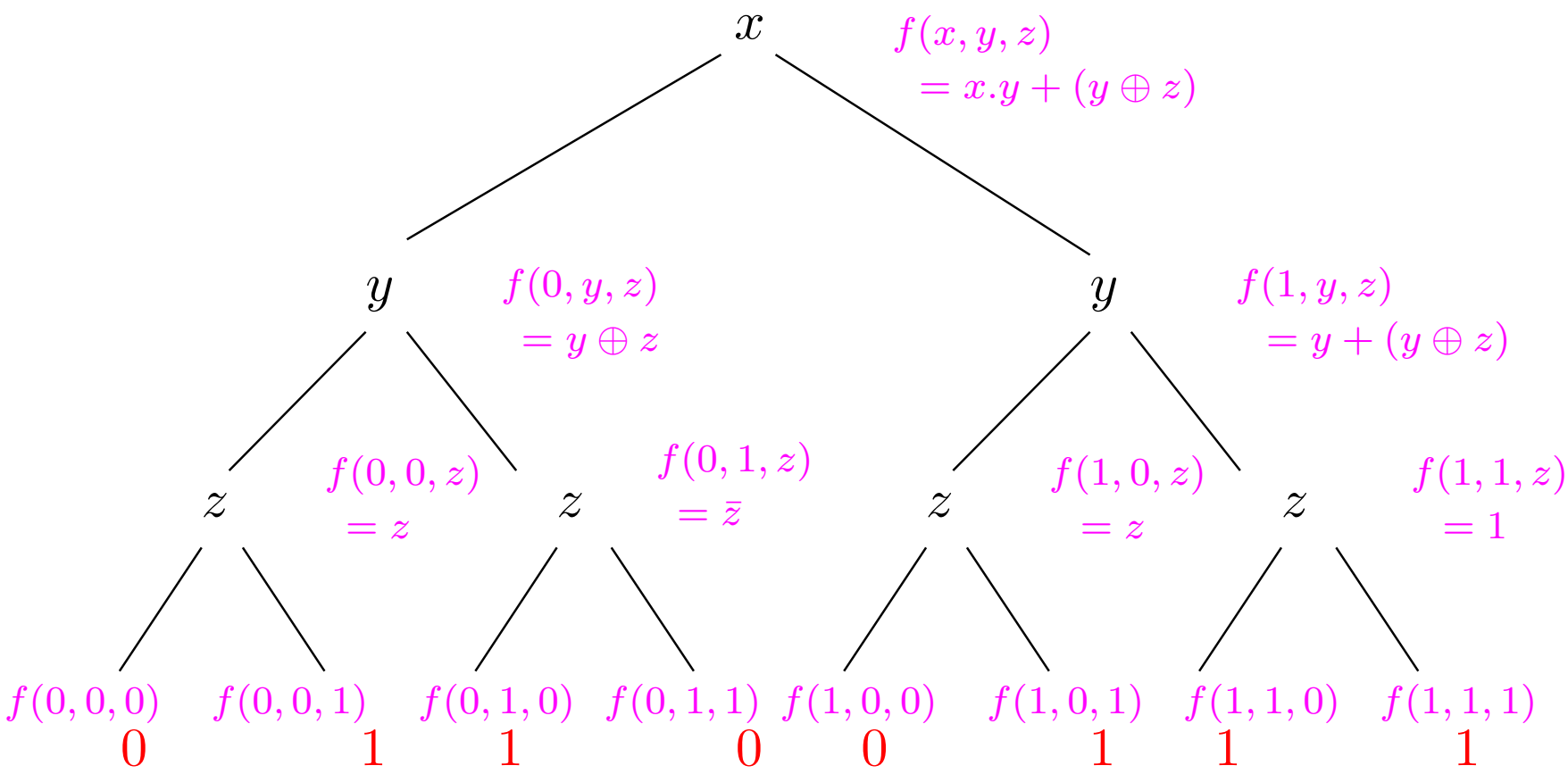
- $f(x, y, z) = x.y + (y \oplus z)$
- $f_x = y + (y \oplus z)$
- $f_{\bar{y}} = z$
- $f_z = x.y + \bar{y} = x + \bar{y}$

Décomposition récursive

- Si on itère la décomposition, on arrive sur une fonction constante :
  - $\hookrightarrow$ toujours vraie, notée 1,
  - $\hookrightarrow$ toujours fausse, notée 0.
- Exemple, f pour $\bar{x}, y, z$:

$$f_{\bar{x}} = f(0, y, z) = y \oplus z$$
$$f_{\bar{x}y} = f(0, 1, z) = \neg z$$
$$f_{\bar{x}yz} = f(0, 1, 1) = 0$$

Représentation graphique (arbre de Shannon)



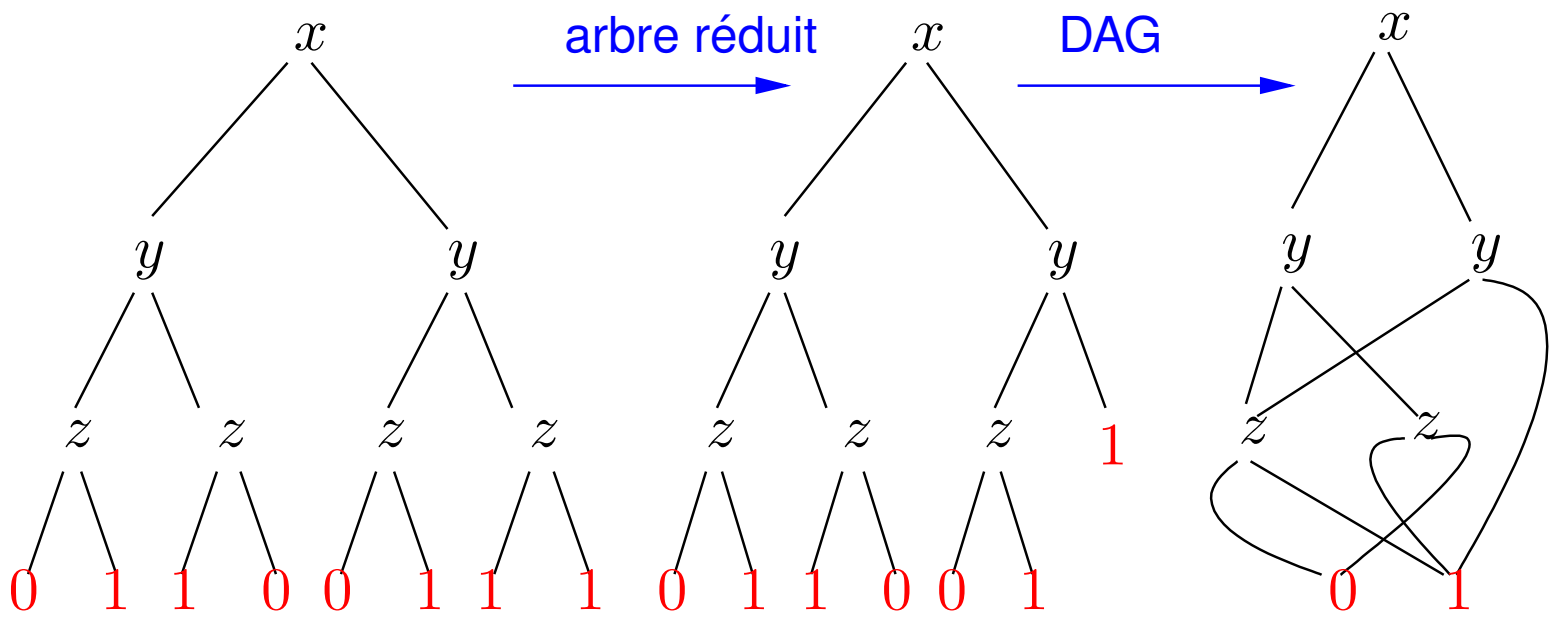
n.b. Pour un ordre donné des variables l'arbre est *unique*

- La dernière rangée est la table de vérité de la fonction.

BDD (binary decision diagram)

Qu'est-ce c'est ?

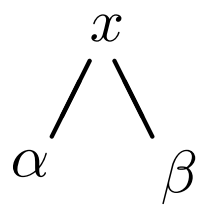
Représentation concise de l'arbre de Shannon :



- Suppression des nœuds inutiles : si x alors f sinon f (arbre réduit de Shannon)
- Partage des nœuds identiques : technique classique, D(irect)A(cyclic)G(graph)
- Pour un ordre donné, on a toujours *unicité*

Définition inductive

Un BDD sur l'ensemble de variables ordonné $(V, <)$ c'est :

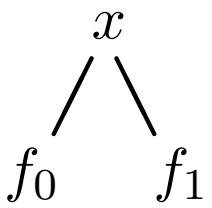
- la feuille 1 (toujours vraie)
- la feuille 0 (toujours fausse)
- un nœud binaire  avec :
  - $\hookrightarrow$ $x \in V$, α et β sont des BDDs
  - $\hookrightarrow$ $x < y$ pour tout $y \in Var(\alpha) \cup Var(\beta)$

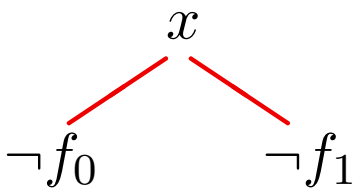
Opérations logiques sur les BDDs

Négation

Simple descente récursive :

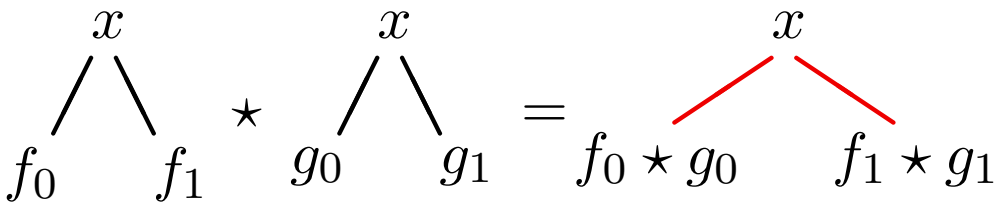
- $\neg 1 = 0$
- $\neg 0 = 1$

- $\neg$ 

$=$ 

Opérateurs binaires

Tous les opérateurs classiques (et, ou, non, xor etc) distribuent sur Shannon :



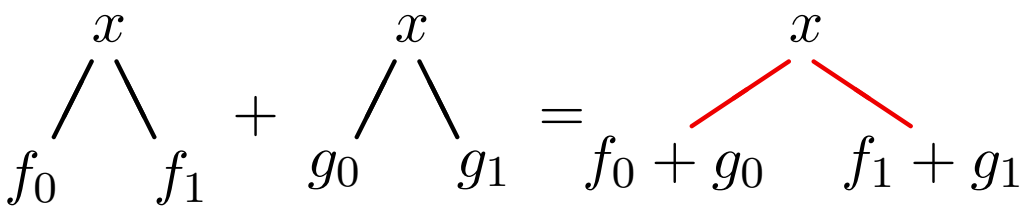
≪ OU ≫ logique (union)

- Règles terminales :

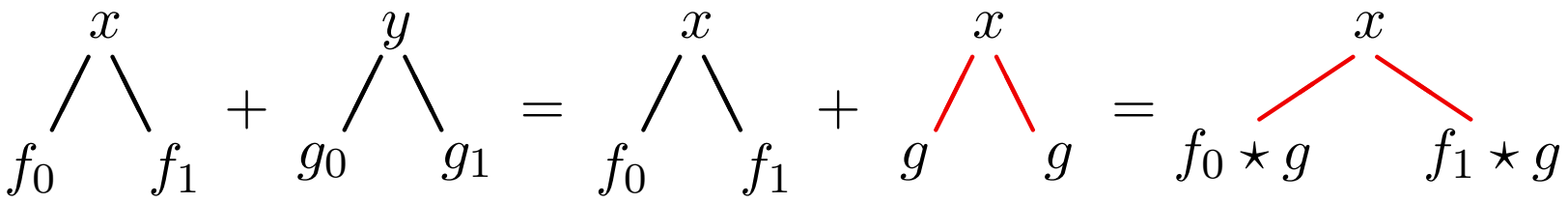
  $\hookrightarrow (1 + f) = (f + 1) = 1$

  $\hookrightarrow (0 + f) = (f + 0) = f$

- Descente récursive : si même variable racine x,



- Équilibrage : si $x \prec y$,



- Similaire pour les autres (seules les règles terminales changent)

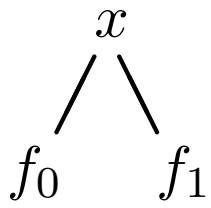
Quantification

- En booléen, la quantification (par exemple existentielle) est un opération « simple » :

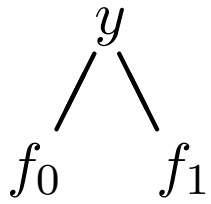
  $\exists x \;\; f(x,y,z) = f(0,y,z) + f(1,y,z)$

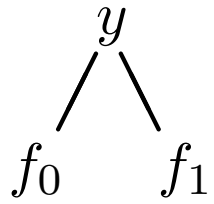
- D'où la définition :

$\hookrightarrow \exists x\ 1 = 1 \quad \exists x\ 0 = 0$

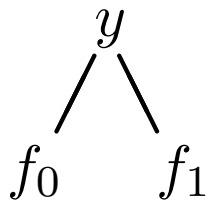
$\hookrightarrow \exists x$ 

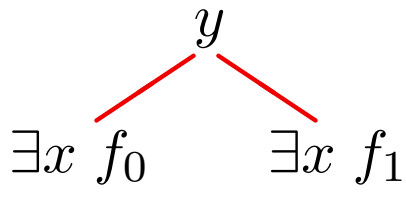
$= f_0 + f_1$

$\hookrightarrow \exists x$ 

$=$ 

si $x \prec y$

$\hookrightarrow \exists x$ 

$=$ 

si $y \prec x$

Éléments de complexité

En bref ...

- Coût de $\neg\alpha$: linéaire en nombre de nœuds
- Coût de $\alpha \star \beta$: de l'ordre de taille(α).taille(β)
- Passage formule (algébrique) vers bdd : exponentiel (en temps et mémoire) dans le pire des cas

Remarque : importance de l'ordre

$(x_1 \oplus x_2).(x_3 \oplus x_4).\cdots.(x_{2n-1} \oplus x_{2n})$

Taille en $O(n)$ pour $x_1 \prec x_2 \prec x_3 \prec \cdots \prec x_{2n}$

Taille en $O(2^n)$ pour $x_1 \prec x_3 \prec \cdots \prec x_{2n-1} \prec x_2 \prec x_4 \prec \cdots \prec x_{2n}$

Utilisation des BDDs

Implémentation(s)

- on a vu juste les principes de bases
- nombreuses variantes, améliorations

Librairie(s) de BDDs

- On est *utilisateur* de BDDs, pas *développeur* !
- Nombreuses librairies disponibles :
  - ↪ structure interne transparente,
  - ↪ ordre des variables caché,
  - ↪ API abstraite :
    - ∗ *true-bdd*, *false-bdd*, *idy-bdd(x)*
    - ∗ *and-bdd(f,g)*, *or-bdd(f,g)* etc
    - ∗ *exist-bdd(v, f)*, *forall-bdd(v, f)*

7. Exploration symbolique

Généralités

- Manipuler *globalement* des ensembles d'états, de transitions
- Les ensembles sont codés par des formules logiques
- Concrètement, les formules sont codées par des BDDs
- n.b. La couche BDD est abstraite : on raisonne à haut niveau (union, intersection etc) sans se soucier de la nature exacte des BDDs

Exploration symbolique en avant

Principes de l'algorithme

- s'applique à un système $(M, V, \text{Init}, G, \Phi, H)$ donné

  (on pose $Q = 2^M$ l'ensemble des états potentiels)

- manipule des ensembles :

  $\hookrightarrow$ d'états (formules sur M)

  $\hookrightarrow$ de transitions (formules sur $M \times V$)

- utilise des opérations ensemblistes (i.e. logiques)

- et le calcul d'image : Post : $2^Q \to 2^Q$

  $\text{Post}(X) = \{q' / \exists q \in X, v \in 2^V \;\; H(q, v) \wedge q \xrightarrow{v} q'\}$

  (On verra + tard comment l'implémenter)

Algorithme

On gère un BDD A = états accessibles en n transitions

- Initialement : $A := \text{Init}$
- Répéter :

  $\hookrightarrow$ Si $A \wedge \text{Err} \neq 0$ alors STOP (la preuve échoue)

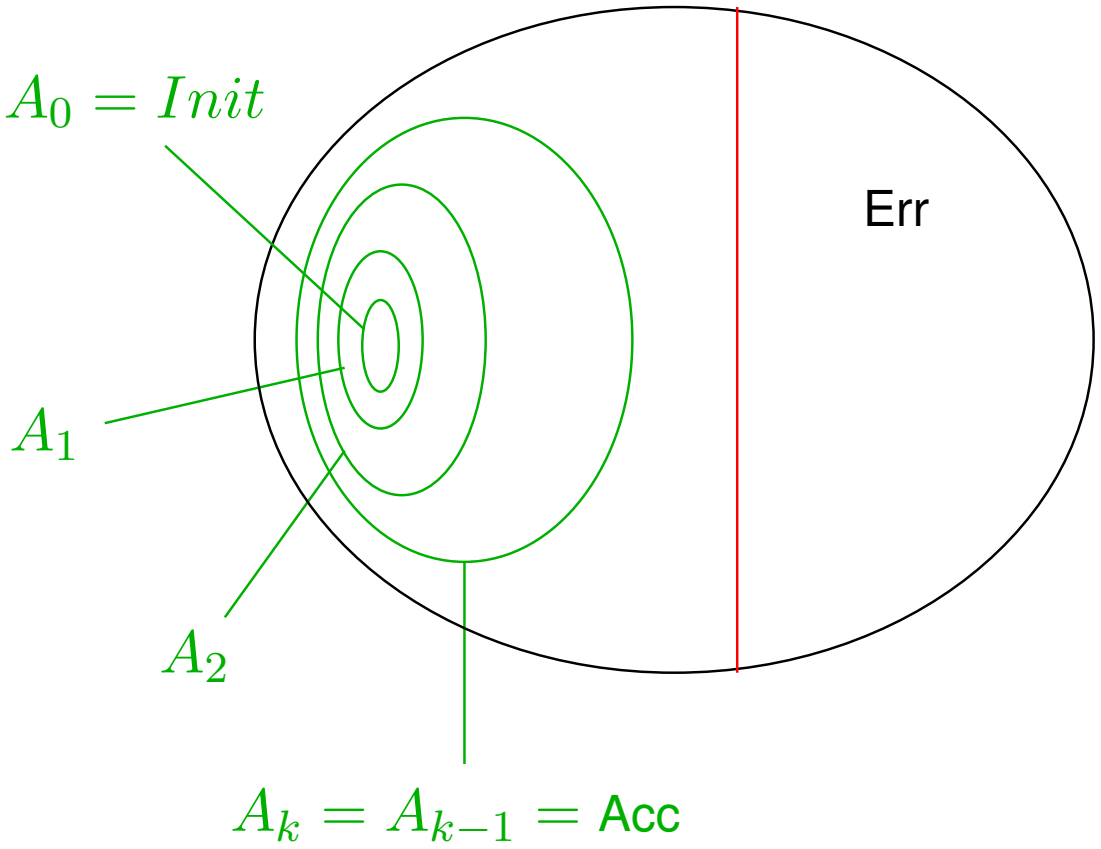
  $\hookrightarrow$ Sinon, soit $A' := A \vee \text{Post}(A)$:

  Si $A' = A$ alors STOP (la preuve réussit)

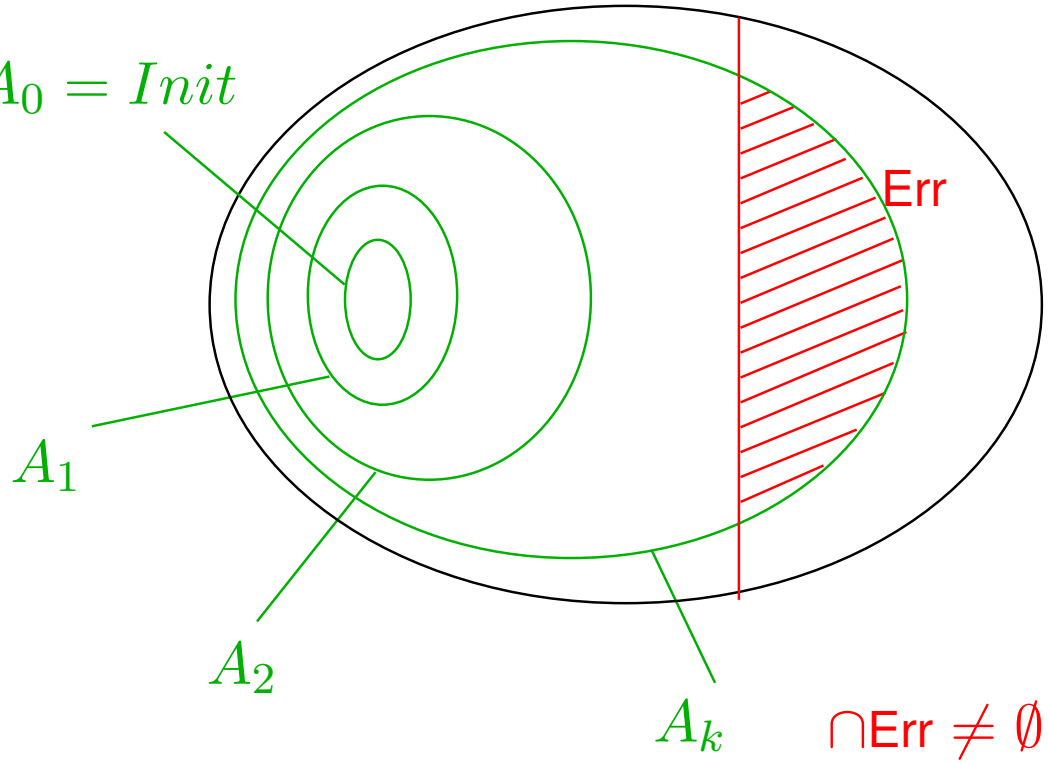
  Sinon $A := A'$ et on continue

N.B. quand la preuve réussit, on a bien $A = A' = \text{Acc}$

La preuve réussit



La preuve échoue



Note sur le calcul de Post(X)

- Peut sembler complexe ...
- ... mais on peut l'implémenter avec de simples opération logiques
- Idée : on construit une (grosse) formule ou apparaissent :

  $\hookrightarrow$ les variables d'état de départ $m_1, m_2, \cdots m_n$ (ou $\vec{m}$)

  $\hookrightarrow$ les variables libres $v_1, v_2, \cdots v_m$ (ou $\vec{v}$)

  $\hookrightarrow$ les variables d'état d'arrivée $m'_1, m'_2, \cdots m'_n$ (ou $\vec{m'}$)

$$\exists \vec{m}, \vec{v}(\ X(\vec{m}) \ \wedge H(\vec{m}, \vec{v}) \ \ \wedge \bigwedge_{i=1}^{n} m'_i = g_i(\vec{m}, \vec{v}) \quad)$$

  $\rightarrow \vec{m}$ est un état source

  $\rightarrow (\vec{m}, \vec{v})$ satisfont les hypothèses

  $\rightarrow$ chaque m'_i est l'image du g_i correspondant

  $\rightarrow$ élimination des m_i et des v_j

- On obtient la formule décrivant les successeurs : $N(\vec{m'})$
- n.b. un peu $\ll$ naïf $\gg$, les vrais algos sont plus sophistiqués ...

Exploration symbolique en arrière

Principes

- Dual du $\ll$ en avant $\gg$, il suffit de tout retourner :
  - $\hookrightarrow$ échanger les rôles de Err et de Init,
  - $\hookrightarrow$ Pre au lieu de Post

Algo

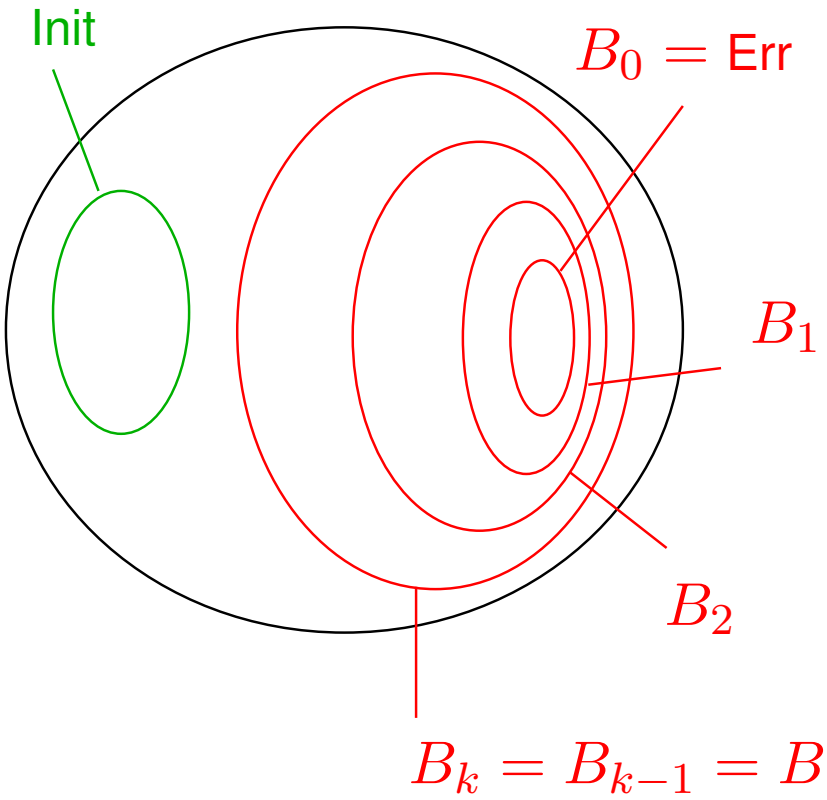
On gère un BDD B = états conduisant à erreur en n transitions

- Initialement : $B := \text{Err}$
- Répéter :
  - $\hookrightarrow$ Si $B \wedge \text{Init} \neq 0$ alors STOP (la preuve échoue)
  - $\hookrightarrow$ Sinon, soit $B' := B \vee \text{Pre}(B)$:

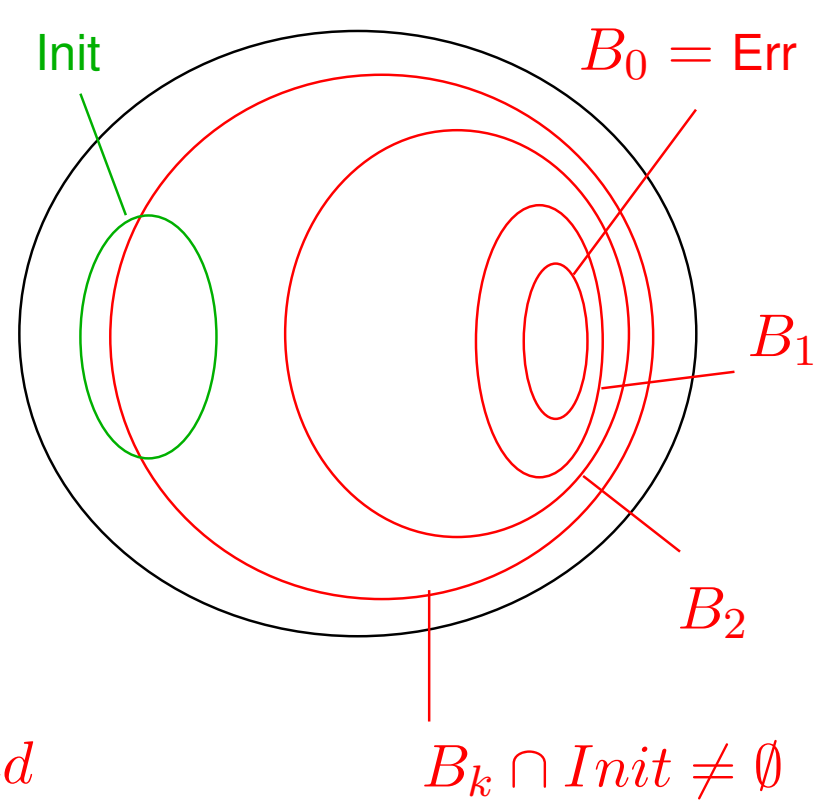
    Si $B' = B$ alors STOP (la preuve réussit)

    Sinon $B := B'$ et on continue
- N.B. quand la preuve réussit, on a bien $B = B' = \text{Bad}$

La preuve réussit



La preuve échoue



8. Conclusion

Exploration

- Les méthodes qu'on a vues sont issues du domaine du *Model-checking*.
- Dans la littérature, on parle d'analyse d'accessibilité (Reachability Analysis).
- En général, symbolique avant globalement meilleur que symbolique arrière ...
- ... et symbolique (bien) meilleur que énumératif.
- Avec des exceptions !
- Dans tous les cas, limité à des systèmes relativement $\ll$ simples $\gg$
  - $\hookrightarrow$ Attention : simplicité n'est pas forcément liée à la taille (cf. méthodes symboliques)
  - $\hookrightarrow$ Systèmes réputés $\ll$ complexes $\gg$: arithmétique binaire (utilise intensivement les xor)

Méthodes alternatives pour les systèmes finis

Utilisation d'un Sat-Solver (au lieu des BDDs)

- Un Sat-Solver répond à la question $\ll f$ est-elle satisfiable ? $\gg$
  - $\hookrightarrow$ en temps exponentiel (pire des cas),
  - $\hookrightarrow$ mais en mémoire polynomiale.

Preuve inductive

- Algo. classique :
  - $\hookrightarrow$ prouver $P(0)$
  - $\hookrightarrow$ prouver $P(t) \Rightarrow P(t+1)$
- MAIS toutes les propriétés ne sont pas inductives en 1 coup !
  - $\hookrightarrow$ en deux coups : $P(0)$ et $P(1)$ et $P(t) \wedge P(t+1) \Rightarrow P(t+2)$
  - $\hookrightarrow$ 3 coups, etc... Pire des cas : inductive en le *diamètre* du graphe !

Bounded Model-Checking

- Méthode très utilisée en pratique
- Vérifie $P(t)$ pour tout t inférieur à un n donné
- Établit *l'absence de bugs* en n coups
- Pas vraiment de la preuve : plutôt du $\ll$ super-debug $\gg$

Interprétation abstraite

- Model-checking (de systèmes finis) :
  - ↪ Approximation du système par un système fini,
  - ↪ calcul exact de points fixes sur ce système fini.
- Interprétation abstraite :
  - ↪ Approximation du système par un système abstrait,
  - ↪ calcul approximé de points fixes sur ce système abstrait.

9. Exemples/exos

Prise en main de Lustre

- Voir les slides ≪ tutoriel/prise en main ≫
- Écrire et simuler avec `luciole` les programmes vus en cours **`edge`**, **`switch`** etc.
- Définir des propriétés pour ces programmes, utiliser `xlesar` pour les vérifier.

  ↪ n.b. ce sont de (très) petits programmes, les propriétés sont simples ...

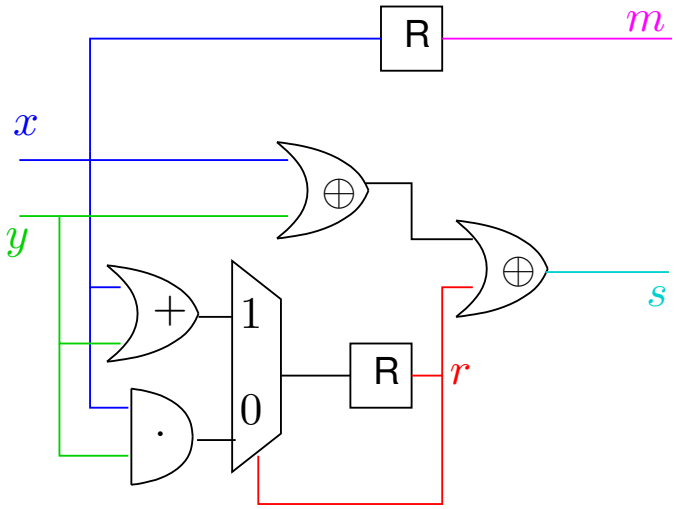
Un peu d'arithmétique

Additionneur série :

- entrées x, y
- somme s, retenue r

Doubleur :

- m code $2x$



temps →

r	0	0	1	0	
x	0	1	0		(2)
y	1	1	0		(3)
s	1	0	1		(5)
m	0	0	1		(4)

Propriété : si $x = y$ alors $s = m$

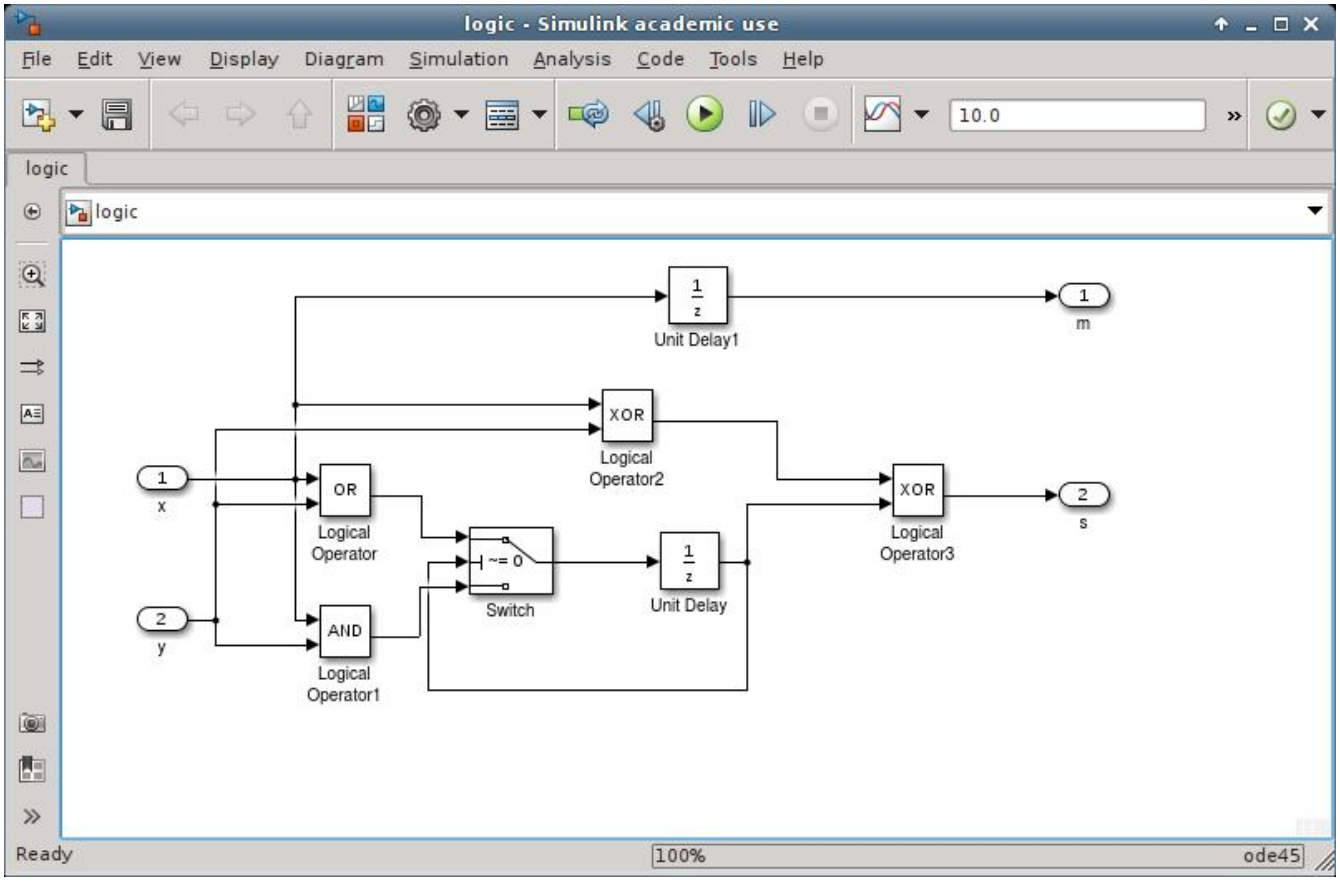
- Écrire en Lustre.
- Donner le modèle équationnel.
- Vérifier la propriété en explorant (à la main) l'automate.
- Vérifier la propriété avec Xlesar.

Remarque : on peut aussi programmer en Simulink, cf. slide suivant

Retour au cours, page 51

Remarque : On peut programmer Lustre en Simulink

- N'utiliser QUE du simulink fonctionnel, temps discret (i.e. `pre` = $1/z$)
- Sauvegarder en `mdl`
- Utiliser `mdl2lus` pour convertir (cf. cours ≪ lego ≫)



Berger, loup, chèvre et salade

Cet exercice est basé sur l'énigme bien connue ≪ du loup, de la chèvre mouton et du chou ≫, (remplacé ici par une salade !) dont voici l'énoncé : une personne, qu'on appelle le **B**erger, doit faire traverser une rivière à un **L**oup, un **C**hèvre et une **S**alade

Il dispose d'une barque très petite où il ne peut embarquer qu'une chose à la fois.

Bien entendu, il ne peut en aucun cas laisser sans surveillance : le **L**oup avec la **C**hèvre, la **C**hèvre avec la **S**alade.

Le but de cet exercice est de montrer que ce problème peut être modélisé par un système à états fini, qu'on peut explorer pour trouver une éventuelle solution.

Proposer un codage du problème sous la forme d'un système réactif.

Observateurs génériques

Certain types de propriétés reviennent très souvent quand on veut exprimer des hypothèses et/ou des propriétés attendues, en particulier pour tout ce qui concerne le comportement dynamiques des ≪ signaux logiques ≫.

Exemples :

- A et B alternent dans le temps,
- jamais X entre chaque A et le B qui le suit,
- continuellement X entre chaque A et le B qui le suit,
- au moins une fois X entre chaque A et le B qui le suit.

Attention :

- les 2 dernières propriétés font référence à une notion d'intervalle de temps A, B
- $\Rightarrow$ bien préciser les cas limite (inclus/exclus)
- Pour la suite : c'est l'intervalle de type ≪ inclus, exclu ≫qu'on utilise (i.e. exactement celui qui est implémenté par le `switch`)

Quelques indications pour ceux qui sèchent

- `once_between(x,a,b)` vrai ssi on a toujours au moins un x entre un a (inclus) et le b (exclu) qui suit
- utiliser un `switch` pour ≪ calculer ≫cet intervalle :

```
in_a_b = switch(false, a, b)
```

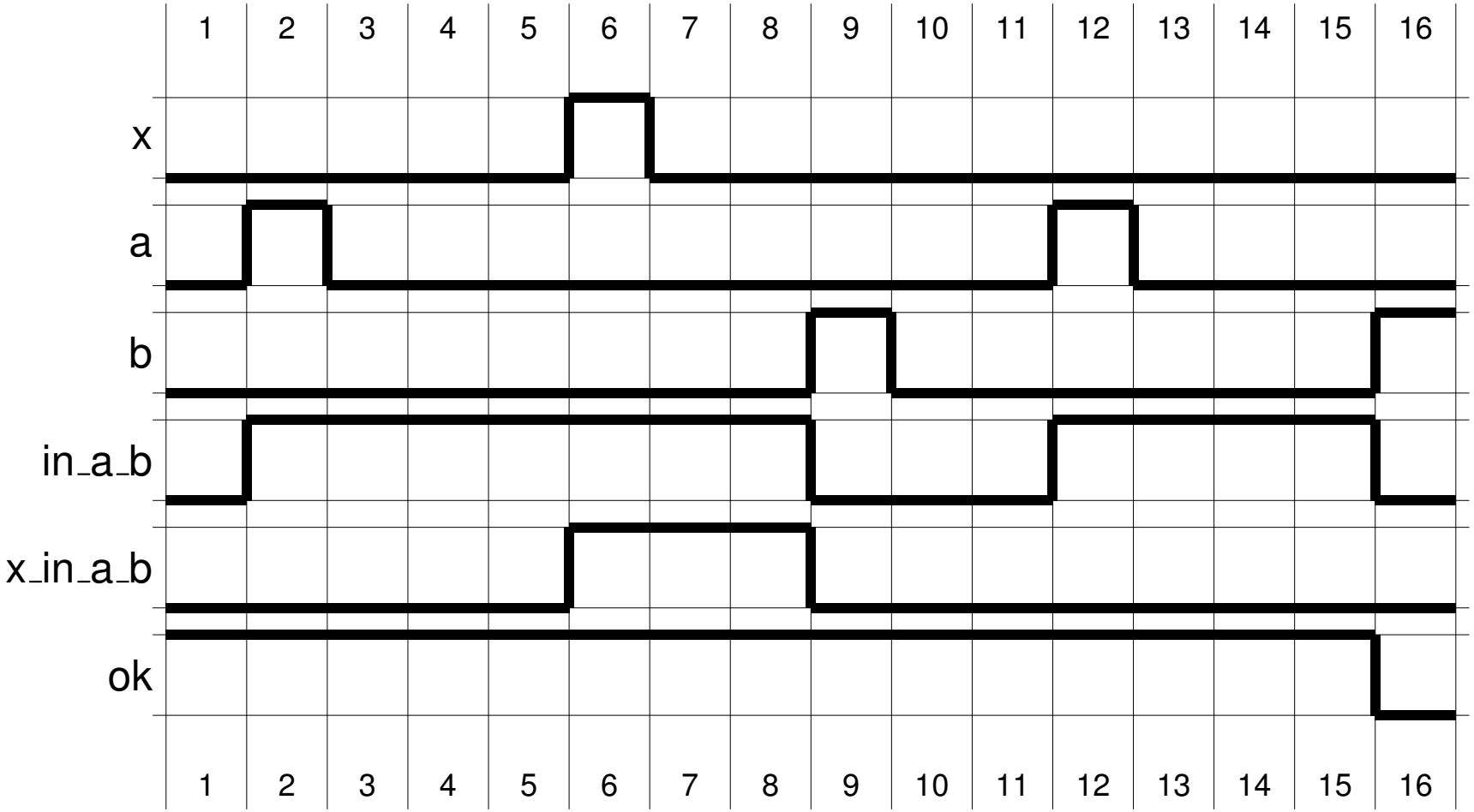
- utiliser un `switch` pour repérer et mémoriser un x qui apparait dans cet intervalle :

```
x_in_a_b = switch(false, x and in_a_b, b)
```

- la propriété est (toujours) vraie ssi chaque fin d'intervalle `in_a_b` est AUSSI la fin d'un intervalle `x_in_a_b` :

```
ok = xedge(not in_a_b) => xedge (not x_in_a_b)
```

Un chronogramme ...



Contrôleur de porte de tramway

But de la manip :

- on part d'un programme déjà écrit
- on veut valider formellement sa sécurité :
  - ↪ implique de (bien) comprendre les specs,
  - ↪ de les formaliser en propriétés de sûreté,
  - ↪ de formaliser d'éventuelles hypothèses nécessaires.

Le contrôleur

↪ inspiré des premières versions du tramway Grenoblois :

* passerelle rétractable sous chaque porte (accès PMR)

↪ Ici : une version simplifiée

↪ Entrées :

* Requêtes usager : `ramp_req` et `door_req`
* État de la rame :
  — `in_station` vrai quand la voiture est à quai
  — `end_station` signal de fin, pour indiquer qu'il faut fermer rampes et portes

↪ Sorties :

* État de la porte : `ramp_on` et `door_on`
* Autorisation de départ : `door_and_ramp_ok`

Les propriétés attendues (informelles)

- ne jamais rouler avec la porte ouverte et/ou la passerelle sortie
- protéger les chevilles des usagers : les mouvements de passerelle se font porte fermée

La manip

- simuler/tester (luciole) le programme pour se familiariser avec son fonctionnement
- formaliser les propriétés, essayer de les prouver avec xlesar,
- réfléchir à d'éventuelles hypothèses si nécessaire

Notes techniques

- télécharger les fichiers `streetcar.lus` (contient `door_control`) et `utils.lus` (contient `switch`, `edge` etc)
- utiliser `xlesar`
- utiliser le menu `import` de xlesar pour charger `utils.lus`, et pouvoir utiliser des nœuds dans les propriétés/hypothèses