Slide 1

Une introduction à la vérification de modèles

Séminaire du doctorat en informatique cognitive

UQAM

16 octobre 2003

Guy Tremblay

Dépt. d'informatique
UQAM

Slide 2

Aperçu

- Que sont les méthodes formelles de spécification et vérification?
- Qu'est-ce qu'un système réactif?
- De quelle façon le comportement d'un système réactif peut-il être décrit?
- De quelle façon les propriétés d'un système réactif peuvent-elles être spécifiées?
- Qu'est-ce que la vérification de modèles?
- Quelles sont les principales techniques permettant de mettre en oeuvre la vérification de modèles?
- Existe-t-il des pistes de recherche possibles en informatique cognitive liées à la vérification de modèles?

1 Que sont les méthodes formelles?

"[Formal methods are] mathematically based techniques used to describe the properties of computing systems. They [are used to] specify, develop, and verify systems in a systematic and rigorous manner [. . .]" [Wing90]

Principaux éléments d'une méthode formelle :

Slide 3

- Langage *formel* pour l'écriture de spécifications
- Règles pour vérifier la qualité des spécifications
- Stratégies et règles pour raffiner et *vérifier* les spécifications

Fondation sur laquelle repose *tout* le reste = Spécifications formelles

Qu'est-ce qu'un langage formel de *spécification*?

Langage formel $\Rightarrow$ syntaxe et sémantique bien définies :

- Syntaxe = EBNF, diagrammes syntaxiques, etc.
- Sémantique = algèbres, automates et systèmes de transitions, relations et prédicats, etc.

Slide 4

Langage de spécification $\Rightarrow$ décrit le *comportement* externe d'un composant logiciel . . .

- en décrivant ses principales propriétés
- d'une façon *abstraite*
  (sans détails inutiles de mise en oeuvre)
- sans dire *comment* ce comportement sera réalisé
  (de façon *non algorithmique*)

Slide 5

Donc, un langage de programmation *n'est pas* un langage de spécification parce que . . .

- il est algorithmique
- il n'est pas suffisamment abstrait (tableaux, pointeurs, etc.)

Une spécification fournit une description *non algorithmique*

$\Rightarrow$ Décrit le "quoi?" plutôt que le "comment?"

Un exemple (en Spec) :

```
FUNCTION racine_carree{ precision: real SUCH THAT precision > 0.0 }

  MESSAGE racine( x: real SUCH THAT x >= 0.0 )
    REPLY( r: real )
        WHERE r >= 0.0  &  presque_egaux( r^2, x )

  CONCEPT presque_egaux( r1 r2: real ) VALUE( b: boolean )
      WHERE b <=> abs(r1 - r2) <= precision
END
```

Slide 6

Quels sont les principaux avantages d'utiliser des spécifications formelles et des méthodes formelles?

> *"Having to better understand the* specificand *by compeling the analyst to be abstract yet precise about the properties of the system can be more rewarding than having the specification itself."* [Wing90]

- Les spécifications sont plus explicites, plus précises, moins ambiguës.
- *Importance de l'effort de formalisation* $\Rightarrow$ aide à identifier *tôt* dans le cycle de vie les erreurs, les ambiguïtés, les problèmes.

- Fournit une meilleure fondation pour le travail de mise en oeuvre subséquent.
- Permet l'utilisation d'outils (manipulation, analyse, simulation).
- Fournit une base pour le développement des tests.

- Fournit une base pour effectuer des vérifications formelles.

Slide 7

Pourquoi y-a-t-il plusieurs langages et méthodes formels?

Plusieurs *styles* différents de spécifications :

- Modélisation abstraite pour machines et objets
  (VDM, Z, Spec, etc.) ;
- Spécifications algébriques pour types abstraits de données
  (Larch, ACT ONE, etc.) ;
- Spécifications de comportement pour systèmes réactifs
  (CCS, CSP, LOTOS, ACP, etc.) ;
- Spécifications de propriétés de sûreté et vivacité
  (logiques modale et temporelle) ;
- etc.

Situation semblable à celle des langages de programmation :

- Domaines d'applications variés
- Styles et paradigmes variés
- Divers niveaux d'expressivité et de facilité d'analyse

Slide 8

2 Spécification de systèmes réactifs et concurrents

Un système est dit *réactif* . . .

- lorsqu'il maintient une interaction *constante* avec son environnement
- quand son comportement est dirigé par les événements ("*event-driven*")

Un système est dit *concurrent* . . .

- lorsque son comportement est déterminé par l'*interaction* d'un groupe de tâches (processus) qui échangent de l'information et coopèrent

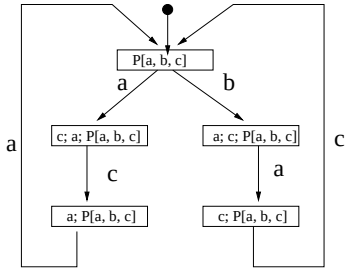
Slide 9

Modélisation du comportement de systèmes réactifs

- Le comportement d'un système réactif peut être décrit en spécifiant les actions qu'*il peut* (ou *ne peut pas*) exécuter
- La trace de l'exécution d'un système réactif est généralement *infinie*
  $\Rightarrow$ utilisation de systèmes de transitions (automates)

Un petit exemple : spécification Lotos et sa représentation graphique

```
process P[a, b, c]: noexit :=
  a; c; a; P[a, b, c]
  []
  b; a; c; P[a, b, c]
endproc
```



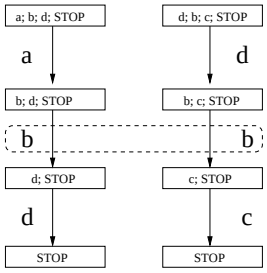
Slide 10

Modélisation de systèmes concurrents

Un comportement concurrent peut être exprimé par *l'entrelacement* des actions :

- Actions concurrentes = actions non ordonnées $\Rightarrow$ peuvent s'exécuter *dans n'importe quel ordre* = n'importe quel entrelacement est permis
- Actions synchronisées = actions exécutées de façon *synchrone* par deux (ou plusieurs) agents $\Rightarrow$ une seule action est visible

```
a; b; d; STOP
|[b]|
d; b; c; STOP
```



Ensemble des différentes traces (*visibles*) d'exécution =

$\{\mathtt{adbdc}, \mathtt{dabdc}, \mathtt{adbcd}, \mathtt{dabcd}\}$

Slide 11

Spécification des *propriétés* du comportement

- Automate = forme de *description opérationnelle*
  ≈ décrit comment générer les différentes séquences possibles d'actions

- Mais ... une telle description ne décrit pas façon explicite les *propriétés* qui doivent être satisfaites par le comportement

  - Propriétés de sécurité : *nothing bad will ever happen*.
  - Propriétés de vivacité : *something good will eventually happen*.

Différentes approches à la spécification des propriétés :

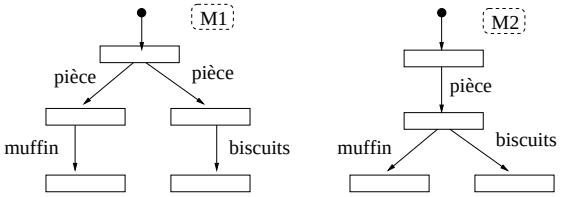
- Logique modale : propriétés locales de l'état courant
- Logique temporelle : propriétés des séquences d'exécutions (*runs*)
  - Logique de temps linéaire
  - Logique de temps bifurcant

Slide 12

Logique de temps linéaire

Propriété de temps linéaire = propriété le long d'un *unique* chemin d'exécution
⇒ Un état satisfait une propriété de temps linéaire si *tous les chemins complets* qui partent de cet état satisfont la propriété

Exemple :



Ces deux machines génèrent le même ensemble de chemins (complets) :

```
{ pièce;muffin, pièce;biscuits }
```

⇒ les deux vont satisfaire les mêmes propriétés de temps *linéaire*

... mais ont-elles réellement le même comportement?

Slide 13

Logique modale

Logique modale = exprime des propriétés (locales) de l'état courant

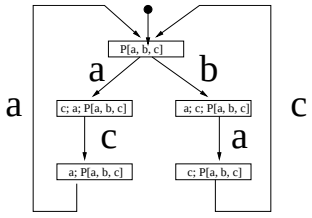
- Possibilité (may) : $\langle \mathtt{a} \rangle \phi$
  = il est possible d'exécuter une action a et ensuite d'atteindre un état qui satisfait ϕ
- Nécessité (must) : $[\mathtt{a}]\phi$
  = lorsque l'action a est effectuée, l'état résultant satisfait ϕ

Deux expressions idiomatiques typiques :

- $\langle \mathtt{a} \rangle \mathtt{tt}$ = il est possible d'exécuter l'action a
- $[\mathtt{a}]\mathtt{ff}$ = l'action a ne peut pas être exécutée

Slide 14

Exemples sur P =



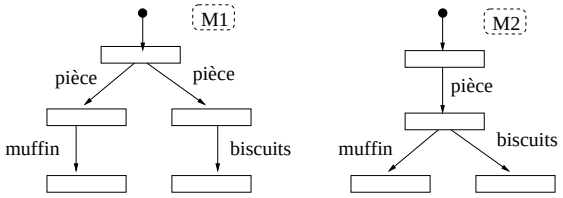
- $\mathtt{P} \models \langle \mathtt{a} \rangle \mathtt{tt}$
  (Vivacité) P peut faire a comme première action

- $\mathtt{P} \models [\mathtt{a}][\mathtt{b}]\mathtt{ff}$
  (Sûreté) Dans l'état initial, P ne peut pas faire un a suivi d'un b

- $\mathtt{P} \models [-\mathtt{b}]\langle \mathtt{c} \rangle \mathtt{tt} \wedge [-\mathtt{a}]\langle \mathtt{a}, \mathtt{d} \rangle \mathtt{tt}$
  (Vivacité) Si la $1^{ière}$ action n'est pas b, alors la $2^{ième}$ est un c et si la $1^{ière}$ n'est pas un a, alors la $2^{ième}$ est un a ou un d

Slide 15

Logique modale $\Rightarrow$ Les machines M1 et M2 *peuvent maintenant être distinguées* :



- $\texttt{M1} \not\models [\texttt{pièce}]\langle\texttt{muffin}\rangle\texttt{tt}$
- $\texttt{M2} \models [\texttt{pièce}]\langle\texttt{muffin}\rangle\texttt{tt}$

Slide 16

Logique temporelle (temps bifurcant)

Logique temporelle

= Exprime les propriétés de chemins (de suites d'actions = *runs*)

$\Rightarrow$ Décrit de façon *qualitative* l'occurrence d'événements dans le temps

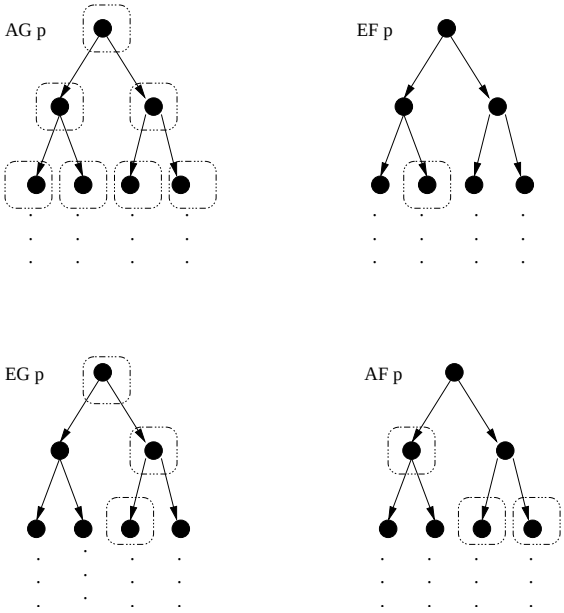
CTL = *Computation Tree Logic* :

- $s \models \texttt{AG}\phi$
  = ϕ est vrai dans tous les états pouvant être atteints à partir de l'état s
  = *Always*(ϕ)

- $s \models \texttt{EF}\phi$:
  = à partir de s, il existe un chemin où ϕ deviendra éventuellement vrai
  = *Eventually*(ϕ)

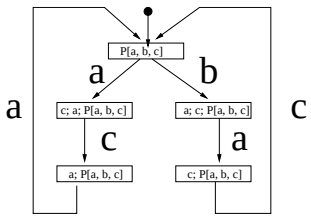
Slide 17

Quatre principaux *quantificateurs temporels* : AG, EF, EG, AF



Slide 18

Exemples sur P =



- $P \models AG([b][c]ff)$

  (Sûreté) Pour n'importe quelle séquence d'exécution, il n'est jamais possible de faire b suivi de c

- $P \models AG(EF\ \langle a \rangle tt)$

  (Vivacité faible) Le long de n'importe quel chemin partant de l'état initial de P, éventuellement une action a sera possible.

Slide 19

Mu-calcul

Mu-calcul modal = logique temporelle avec opérateurs explicites de points fixe

Syntaxe :

$$\phi ::= \texttt{tt} \mid \texttt{ff} \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [L]\phi \mid \langle L \rangle \phi \mid \mu X.\phi \mid \nu X.\phi$$

Définition de *Always* et *Eventually* à l'aide des opérateurs de points fixe :

$$\begin{aligned} \textit{Always}(\phi) &= \nu X.\phi \wedge [-]X \\ \textit{Eventually}(\phi) &= \mu X.\phi \vee \langle - \rangle X \end{aligned}$$

Slide 20

3 Vérification de modèles

Vérification de modèles

= "*A technique that relies on building a* finite *model of a system and checking that a desired property holds in that model.*" [ClarkeEtAl96]

= Une technique *automatique* pour *vérifier* les propriétés de systèmes avec un nombre *fini* d'états.

Approche générale pour la vérification de modèles :

1. On construit M = un modèle (du comportement du système)
2. On spécifie ϕ = une propriété attendue du système (exprimée en logique modale/temporelle)
3. On vérifie que M satisfait ϕ.
   Si ce n'est pas le cas, on génère un (ou plusieurs) *contre-exemple(s)*.

Slide 21

La mise en oeuvre de la vérification de modèles nécessite *l'exploration de l'espace des états*.

$\Rightarrow$ Contrainte importante sur M = M doit être *fini*

Avantages/désavantages de la vérification de modèles (+/-) :

+ Vérification complètement automatique
+ Peut générer des *contre-exemples* qui représentent parfois des erreurs *subtiles*

- Problème de l'explosion du nombre des états (*state explosion problem*)

Slide 22

Principales applications = vérification de matériel et de protocoles de communication :

- Protocole de cohérence de cache IEEE Futurebus+ [McMillan93]
  (plusieurs erreurs non détectées ont été découvertes)
- Protocole de télécommunication ISDN/ISUP [Holzmann92]
  (122 erreurs identifiées)
- Contrôleur de canal HDLC [DePalmaGla96]
  (découverte d'un bogue majeur)
- Système de contrôle actif structural en génie civil [ElseaidyEtAl96]
  (découverte d'un bogue majeur qui aurait pu aggraver l'effet des vibrations)
- etc.

4 Mise en oeuvre de la vérification de modèles

4.1 Vérification globale vs. locale

- Vérification globale : étant donné un modèle fini, M, et une formule, ϕ, on doit déterminer *l'ensemble des états de* M *qui satisfont* ϕ.
- Vérification locale : étant donné un modèle fini, M, une formule, ϕ, et un état s de M, on doit déterminer *si* s *satisfait* ϕ.

Slide 23

Caractéristiques de la vérification de modèles globale vs. locale :

- Solution au problème global
  $\Rightarrow$ solution au problème local
- Solution problème global
  $\Rightarrow$ exploration de la totalité de l'espace des états
- Solution au problème local
  $\Rightarrow$ exploration *au besoin* (*demand-driven*) de l'espace des états

4.2 Équations récursives et calcul de points fixes

La mise en oeuvre de la vérification de modèles demande de trouver la (les) solution(s) d'équations récursives.

Notons par $\langle - \rangle$ et $[-]$ l'utilisation des modalités avec des actions arbitraires.

Rappelons que :

Slide 24

- AG ϕ = *Always*(ϕ)
- EF ϕ = *Eventually*(ϕ)

Always et *Eventually* peuvent être définis *récursivement* :

$$\begin{aligned} \textit{Always}(\phi) &= \phi \wedge [-]\textit{Always}(\phi) \\ \textit{Eventually}(\phi) &= \phi \vee \langle - \rangle \textit{Eventually}(\phi) \end{aligned}$$

Slide 25

Définition (point fixe) :

x est un point fixe de f

ssi

$f(x) = x$

Fait : Une solution à une équation récursive est *toujours* un point fixe d'une *fonction τ liée à l'équation.*

Exemple : $x = 2 * x$

- Fonction associée : $\tau(x) = 2 * x$
- Solution : 0 est une (l'unique) solution puisque $\tau(0) = 0$

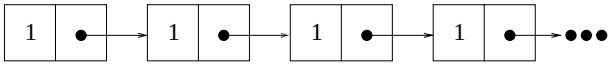
Exemple : $x = x$

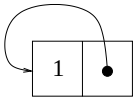
- Fonction associée : $\tau(x) = x$
- Solution : n'importe quel n est une solution puisque $\tau(n) = n$

Slide 26

Exemple : une définition récursive d'une liste d'entiers

- Équation : $l = 1 : l$
- Fonction associée : $\tau(l) = 1 : l$
- Solution : Soit $uns = [1, 1, 1, 1, \ldots]$ une liste infinie de 1s.
  Alors $\tau(uns) = uns$.





Fait : Le plus petit point fixe d'une fonction τ peut être obtenu comme la *limite d'une séquence d'approximations* (où $\perp$ est le plus petit élément) :

$$\bigsqcup_{n=0}^{\infty} \tau^n(\perp)$$

Exemple :

- Soit $\tau(l) = 1 : l$
- Donc :
  - $\tau^0(l) = \perp$
  - $\tau^{i+1}(l) = \tau(\tau^i(l)) = 1 : \tau^i(l)$

Slide 27

$$\begin{aligned}
\tau^0(\perp) &= \perp \\
\tau^1(\perp) &= 1 : \perp \\
\tau^2(\perp) &= 1 : 1 : \perp \\
&\dots \\
\tau^{i+1}(\perp) &= 1 : 1 : \dots : \perp \\
&\dots
\end{aligned}$$

4.3 Vérification globale de modèles pour le mu-calcul

= Déterminer l'ensemble des états qui satisfont une propriété ϕ

$\approx$ Calculer la sémantique dénotationelle d'une propriété
(ensemble des états du modèle qui satisfont la propriété)

Slide 28

$$\begin{aligned}
[\![\mathtt{tt}]\!]_{\mathcal{V}} &= \mathcal{P} \\
[\![\mathtt{ff}]\!]_{\mathcal{V}} &= \{\} \\
[\![X]\!]_{\mathcal{V}} &= \mathcal{V}(X) \\
[\![\phi_1 \wedge \phi_2]\!]_{\mathcal{V}} &= [\![\phi_1]\!]_{\mathcal{V}} \cap [\![\phi_2]\!]_{\mathcal{V}} \\
[\![\phi_1 \vee \phi_2]\!]_{\mathcal{V}} &= [\![\phi_1]\!]_{\mathcal{V}} \cup [\![\phi_2]\!]_{\mathcal{V}} \\
[\![[L]\phi]\!]_{\mathcal{V}} &= \{p \mid \forall\, a \in L, p' \in \mathcal{P} :: p \xrightarrow{a} p' \Rightarrow p' \in [\![\phi]\!]_{\mathcal{V}}\} \\
[\![\langle L\rangle\phi]\!]_{\mathcal{V}} &= \{p \mid \exists\, a \in L, p' \in \mathcal{P} :: p \xrightarrow{a} p' \wedge p' \in [\![\phi]\!]_{\mathcal{V}}\} \\
[\![\mu X.\phi]\!]_{\mathcal{V}} &= fix_{\mu}\, \tau_{\phi,\mathcal{V}} \\
&\quad \text{où } \tau_{\phi,\mathcal{V}}(x) = [\![\phi]\!]_{\mathcal{V}[x \mapsto X]}
\end{aligned}$$

$$fix_\mu\, \tau_{\phi,\mathcal{V}} \quad = \quad \bigcup_{n=0}^{\infty} \tau_{\phi,\mathcal{V}}^n(\{\})$$

Où $\quad \tau^0(x) = x$

$\tau^{i+1}(x) = \tau(\tau^i(x))$

Slide 29

Terminaison de la vérification (globale) de modèles : puisque le modèle est *fini* (*i.e.,* possède un nombre fini d'états), alors un point fixe sera atteint après un nombre fini d'itérations.

4.4 Vérification locale de modèles pour le mu-calcul

= Déterminer si un état s satisfait une propriété ϕ

$\approx$ Calculer la sémantique axiomatique associée à un processus (un état) p
(règles d'inférence)

= Ensemble de règles (inductives) qui spécifient si p satisfait une formule ϕ

Slide 30

$p \models \texttt{tt}$

$p \not\models \texttt{ff}$

$p \models \phi \wedge \psi$ ssi $p \models \phi$ et $p \models \psi$

$p \models \phi \vee \psi$ ssi $p \models \phi$ ou $p \models \psi$

$p \models [L]\phi$ ssi $\forall a \in L, p' \in \mathcal{P} :: p \xrightarrow{a} p' \Rightarrow p' \models \phi$

$p \models \langle L \rangle \phi$ ssi $\exists a \in L, p' \in \mathcal{P} :: p \xrightarrow{a} p' \wedge p' \models \phi$

$p \models \mu X.\phi$ ssi $\ldots$

5 Vérification de modèles en parallèle et le problème de l'explosion des états

La modélisation de la concurrence par l'entrelacement des actions $\Rightarrow$ le nombre total d'états peut croître de façon exponentielle avec le nombre de composants qui s'exécutent de façon concurrente.

Exemple :

Slide 31

- Spécification Lotos de 100 lignes avec 10 petits processus $\Rightarrow$
  - 56 000 états
  - 180 000 transitions

La vérification globale nécessite une exploration *exhaustive* de l'espace des états

$\Rightarrow$ préférable de conserver l'espace des états en mémoire pour éviter des explorations multiples du même état

$\Rightarrow$ beaucoup d'espace est requis pour conserver le graphe (LTS)

Solutions possibles au problème de l'explosion des états

- Vérification *symbolique* de modèles (avec BDD)
- Exploitation de diverses informations (par ex., symétrie) pour réduire le nombre d'états ou de transitions
  (en autant que les propriétés clés soient préservées)
- ...

Slide 32

- Utilisation d'une machine parallèle avec plusieurs *noeuds* (multi-ordinateurs) de façon à augmenter l'espace mémoire disponible $\Rightarrow$
  - Parcourir le graphe pour distribuer et évaluer la fonction de transition
  - Utiliser une fonction de *dispersion* pour distribuer les divers états sur les différents processeurs
  - Ne jamais traiter plusieurs fois une transition en s'assurant de garder une trace des états *visités*

  $\Rightarrow$ nombreuses communications entre processeurs, y compris pour détecter la terminaison

6 Quelques pistes de recherche possibles liées à l'informatiqu cognitive

- Application de la vérification de modèles à la spécification et vérification de *processus d'affaires* (par ex., commerce électronique).
- Définition d'un langage "*convivial*" pour la spécification de propriétés de processus d'affaires.

Slide 33

- Intégration des concepts de la logique floue à la vérification de modèles :
  - Définitions des concepts ;
  - Problèmes de mise en oeuvre ;
  - Applications possibles . . .
- Exploration des difficultés *cognitives* qui limitent l'*essor* des méthodes formelles de spécificatio et vérification (approche *non algorithmique*).
- Tutoriel intelligent pour l'apprentissage des méthodes formelles de spécification et vérification.

7 Conclusion

- La vérification de modèles est une approche intéressante, et puissante, à la vérification formelle *parce qu'elle est automatique*.
- Principale difficulté = traitement d'un immense espace d'états.

- Travaux en cours et futurs :

Slide 34

  - Voir comment une mise en oeuvre parallèle et distribuée pourrait aider au problème de l'explosion du nombre d'états.
  - Appliquer la vérification de modèles au π-calcul
    $\Rightarrow$ nécessite de traiter des processus mobiles
    (espace dynamique d'états et non-fini **:(**
  - Appliquer la vérification de modèles à des domaines *non informatiques*.