Model checking et preuve de modèle

Simon Robillard

Mars 2021

Ce module

- module 2 : Model Checking et preuve de modèle
- 5×2h
- évaluation par un mini-projet
- ces slides seront mises en ligne
- contact : `simon.robillard@imt-atlantique.fr`

Section 1

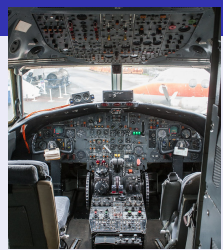
Pourquoi vérifier ?

Approche traditionnelle pour garantir la qualité d'un système logiciel : les tests

- conceptuellement faciles à mettre en œuvre
- utiles à différentes étapes de la vie du logiciel
  - développement (tests unitaires, tests d'intégration, *test-driven development*)
  - livraison du logiciel (tests de validation)
  - mises à jour (tests de non-régression)

Exemple : logiciels critiques en avionique

- normes ED-12C et DO-178C (*Software considerations in airborne systems and equipment certification*)
- classifie les systèmes selon la gravité des problèmes qu'un bug pourrait entraîner
- spécifie différents niveaux de tests suivant les catégories



Peut-on tout tester ?

"Tester des programmes peut être un moyen très efficace de révéler des bugs, mais est irrémédiablement inadapté pour en démontrer l'absence."

– Edsger W. Dijkstra

Issu de *The Humble Programmer*, discours prononcé lors de la réception du prix Turing en 1972

Tests et exhaustivité

Tester tous les inputs de la fonction `abs(n: int)` ?

Tests et exhaustivité

Tester tous les inputs de la fonction `abs(n: int)` ?

- si `int` est un entier 64-bit

$$2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 616 \text{ possibilités*}$$

*environ 100 fois le nombre de secondes écoulées depuis le Big Bang

- avec des entiers multiprécision (e.g., Python), nombre infini de possibilités

Tests et exhaustivité

Tester tous les inputs de la fonction `abs(n: int)` ?

- si `int` est un entier 64-bit

$$2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 616 \text{ possibilités*}$$

*environ 100 fois le nombre de secondes écoulées depuis le Big Bang

- avec des entiers multiprécision (e.g., Python), nombre infini de possibilités

Couverture de code

- une autre façon de définir l'exhaustivité
- différents niveaux possibles : exécuter toutes les fonctions/instructions/branches/conditions du programme
- exemple : critère MC/DC pour les logiciels critiques dans la norme DO-178
- impossible de déterminer la couverture sans avoir écrit le code

Tests et indéterminisme

Définition

Un système est *indéterministe* si une exécution depuis un état donné peut mener à plusieurs état différents

Sources d'indéterminisme

- hasard
  - exemples : générateur de nombre aléatoire, quicksort avec choix du pivot aléatoire
  - hasard $\subset$ indéterminisme, mais l'inverse n'est pas vrai !
- évènements hors du modèle
  - exemples : entrées utilisateur, date et heure
  - la notion de déterminisme dépend donc du modèle
- langage de programmation ou algorithme indéterministe
  - exemple : "l'algorithme choisit un élément x dans l'ensemble S"
- ordre d'exécution de différents processus

Tests et indéterminisme

✗ l'indéterminisme augmente très largement le nombre d'exécutions possibles

✗ certaines sources d'indéterminisme ne sont pas contrôlables

- bugs liés à l'ordre d'exécution des processus très difficiles à reproduire
- l'environnement de test peut même affecter cet ordre et cacher des bugs

Exemple : situation de compétition (*race condition*)

`x` initialisé à 0

processus 1 exécute	processus 2 exécute
`x += 1`	`if (x == 0) {x = 4}`

valeurs finales possible de `x` : 1, 5, mais aussi 4 !

Propriétés non-testables en temps fini

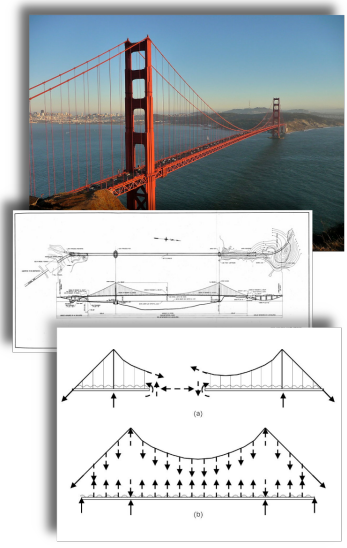
- Terminaison
- Propriétés de sûreté (*safety*) = le système n'entre jamais dans un état non-sûr
  - exemple : "il n'y a jamais plus d'un processus qui a accès à la section critique"
- Propriétés de vivacité (*liveness*) = le système finira par atteindre un état désiré
  - exemple : "tout processus qui demande accès à la section critique l'obtiendra éventuellement"

Tester avec un timeout ?

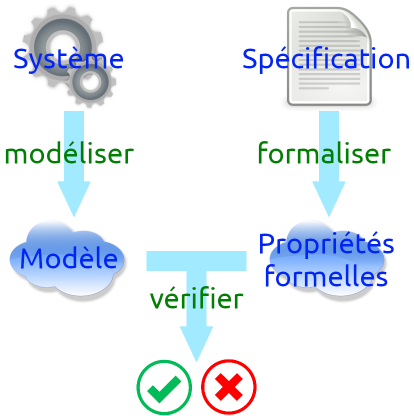
- pour la terminaison et la vivacité, cela rend la propriété testée plus forte que l'originale
  - utile pour des systèmes "temps réel" où le délai de réponse fait partie de la spécification
- pour la sûreté, on ne peut tester qu'une version faible de la propriété

Les méthodes formelles

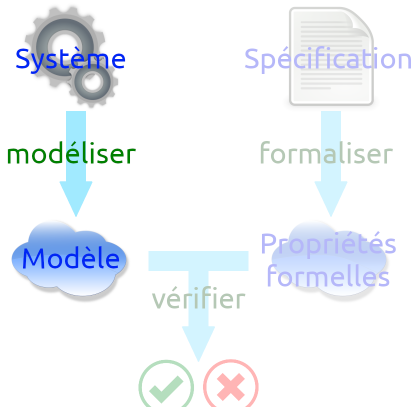
- pour remédier à ces problèmes
  - représentations abstraites
  - techniques de raisonnement mathématiques
- c'est l'approche standard pour les ingénieurs dans tous les autres domaines !
- en informatique, on appelle *méthodes formelles* l'ensemble de ces représentations et techniques



Abstraction

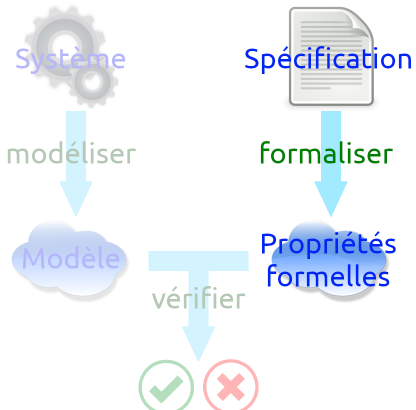


Abstraction



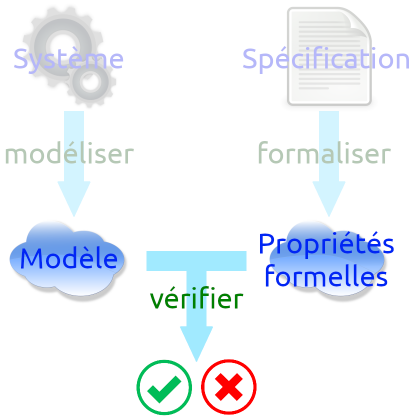
- 1er module : réseaux de Petri
- adapté pour des systèmes concurrents sur des variables discrètes
- d'autres modèles existent pour d'autres types de systèmes

Abstraction



- dans ce module : formaliser des propriétés temporelles en logique CTL
- d'autres logiques existent pour d'autres types de propriétés

Abstraction



- la vérification permet d'obtenir une preuve que le système satisfait des propriétés
- lorsque le système ne vérifie pas la propriétés, certaines techniques de vérification fournissent un contre-exemple
- dans ce module : utilisation d'un outil de vérification sans aborder les détails algorithmiques

Spécification en langage naturel

Les documents de spécification (cahier des charge, RFC, normes ISO...) sont écrits en langage naturel (français, anglais...)

- ✗ le sens peut être ambigu
- ✗ ne peut pas être interprété par une machine

Ils restent un moyen de communication nécessaire, à savoir lire et écrire

- établir une nomenclature
- utiliser un vocabulaire précis, des tournures idiomatiques

Une virgule ou son absence peut changer le sens d'une phrase !

Différents types de langages de spécification

- logique propositionnelle, logique du 1er ordre, logique d'ordre supérieur...
- logiques modales
  - inclut des modalités pour qualifier les propositions (nécéssité, temporalité, croyance...)
  - les **logiques temporelles** en font partie

la logique est-elle

- suffisamment expressive pour nos besoins de spécification ?
- décidable = existe-t-il un algorithme qui prend en entrée un modèle $\mathcal{M}$ et une propriété P, et détermine si $\mathcal{M}$ satisfait P
  - si oui, qu'elle est la complexité de l'algorithme ?
  - certaines logiques sont semi-décidables : il existe une procédure qui termine toujours si $\mathcal{M}$ satisfait P, mais pas toujours dans le cas inverse (ou vice versa)

Prochaine séance

Découverte de la logique temporelle CTL

- syntaxe et sémantique
- comment traduire une spécification en langage naturel vers CTL ?
- quelles sont les propriétés de la logique CTL (expressivité, décidabilité...) ?

Section 2

Spécification en logique temporelle

Une logique temporelle se décompose en général en deux parties

- language pour décrire les propriété du système à un instant donné
  - "la place P contient n jetons"
  - "aucune transition ne peut être franchie"
  - ...
- ajout de *modalités temporelles* pour exprimer des propriétés temporelles
  - "il est possible que..."
  - "il est certain que..."
  - "... toujours..."
  - "... à un moment..."
  - ...

Propriété d'un système dans un état donné

- dans Romeo : Generalized Mutual Exclusion Constraints (GMEC)
- formules interprétées sur un état du système = un marquage du RdP
  - inégalités (`<`, `<=`, `>`, `>=`, `==`, `!=`) sur les marquages des places

$$M(P1) + 3 \times M(P2) \geq 3$$

  - la formule `markingBounded(k)` est vraie si toutes les places ont k jetons ou moins
  - la formule `deadlock` est vraie si aucune transition n'est franchissable
  - les formules peuvent être combinées avec les opérateurs booléens habituels (`and`, `or`, `not`, `=>`)

La logique CTL

Computation **T**ree **L**ogic (logique du temps arborescent)

- logique (modale) temporelle
- interprétée sur un temps avec plusieurs futurs possibles (un arbre)
- pour les réseaux de Petri, l'arborescence temporelle correspond à l'arbre des marquages

Les opérateurs de la logique CTL

Les modalités temporelles de la logique CTL sont :

- si φ et ψ sont des formules (simples ou CTL) alors
- AFφ, AGφ, AXφ, A[φ U ψ], EFφ, EGφ, EXφ, E[φ U ψ] sont des formules CTL

Comprendre la signification des opérateurs

Quantificateur

- A = *along All paths* : toutes les branches de l'arbre doivent satisfaire la formule...
- E = *there Exists a path* : il doit exister une branche (au moins) qui satisfait la formule...

Qualificateur

- F = *Finally* : doit être satisfaite par au moins un état dans la branche
- G = *Globally* : doit être satisfaite par tous les états de la branche
- X = *neXt* : dans l'état suivant de la branche
- U = *Until* = la première formule doit être satisfaite par tous les états de la branche jusqu'à ce que la deuxième formule devienne vraie à un moment

Formule	Signification
AFφ	dans toutes les branches, il y a un état qui satisfait φ
AGφ	tous les états satisfont φ
AXφ	le deuxième état satisfait φ
A[φ U ψ]	il y a un état qui satisfait ψ et tous les états qui le précèdent satisfont φ
EFφ	il existe une branche où il y a un état qui satisfait φ
EGφ	tous les états satisfont φ
EXφ	le deuxième état satisfait φ
E[φ U ψ]	il y a un état qui satisfait ψ et tous les états qui le précèdent satisfont φ

Quelques exemples simples

- les processus $P1$ et $P2$ n'ont jamais accès en même temps

$$\mathsf{AG}\neg(\mathit{accessP1} \wedge \mathit{accessP2})$$

- un message peut être reçu

$$\mathsf{EF}\ \mathit{receive}$$

- la porte reste fermée tant que l'utilisateur ne s'est pas identifié

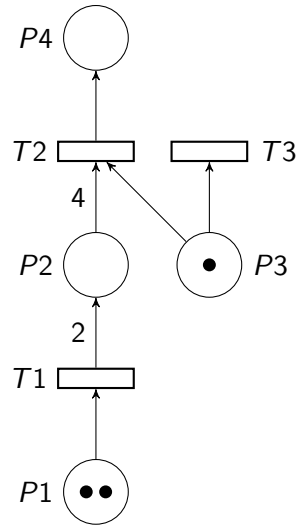
$$\mathsf{A}[\neg \mathit{open}\ \mathsf{U}\ \mathit{identified}] \vee \mathsf{AG}(\neg \mathit{open})$$

Ici les sous-formules *accessP*1, *accessP*2, *receive*, *open* et *identified* sont de simples variables propositionelles. Dans Romeo on utiliserait des GMECs qui décrivent l'état correspondant.

Exercice

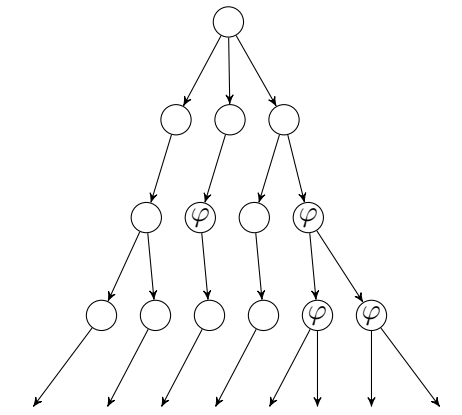
Vérifiez si ce réseau de Petri satisfait les formules

- $\mathrm{AG}(2 \times M(P1) + M(P2) = 4)$
- $\mathrm{EF}(M(P4) > 0)$



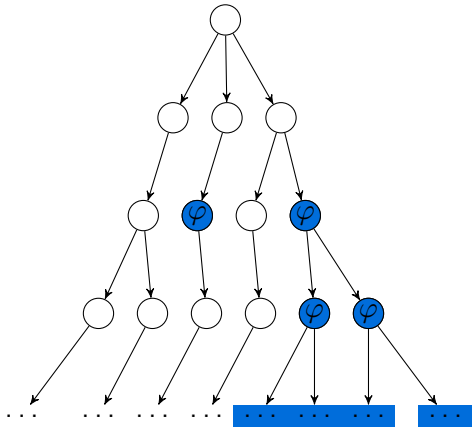
Interpréter des formules récursivement

- un arbre satisfait une formule si elle est vraie à sa racine
- pour les formules avec plusieurs opérateurs imbriqués, commencer par les formules à l'intérieur et remonter
- exemple : EF(AGφ)



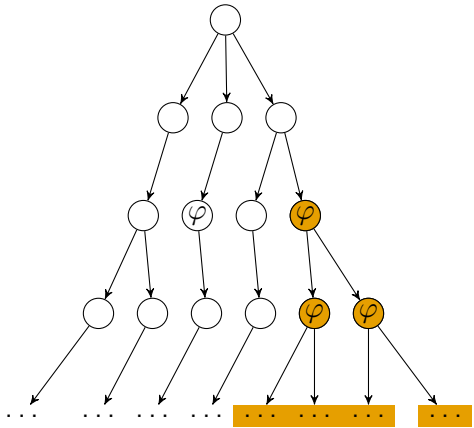
Interpréter des formules récursivement

- un arbre satisfait une formule si elle est vraie à sa racine
- pour les formules avec plusieurs opérateurs imbriqués, commencer par les formules à l'intérieur et remonter
- exemple : EF(AGφ)
  - nœuds satisfaisants φ



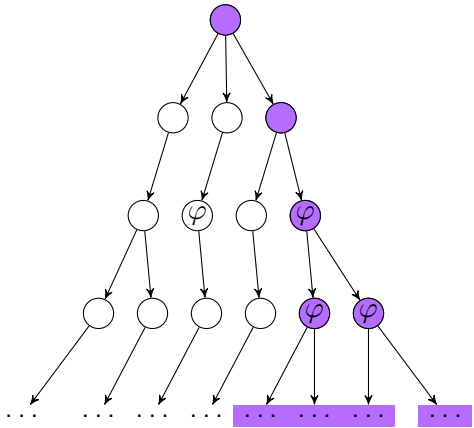
Interpréter des formules récursivement

- un arbre satisfait une formule si elle est vraie à sa racine
- pour les formules avec plusieurs opérateurs imbriqués, commencer par les formules à l'intérieur et remonter
- exemple : EF(AGφ)
  - nœuds satisfaisants φ
  - nœuds satisfaisants AGφ



Interpréter des formules récursivement

- un arbre satisfait une formule si elle est vraie à sa racine
- pour les formules avec plusieurs opérateurs imbriqués, commencer par les formules à l'intérieur et remonter
- exemple : EF(AGφ)
  - nœuds satisfaisants φ
  - nœuds satisfaisants AGφ
  - nœuds satisfaisants EF(AGφ)



Quelques exemples plus complexes

- le système peut rester allumer indéfinimment

$$\mathrm{EF}(\mathrm{EG}\ \mathit{on})$$

- toute requête sera immédiatemment suivie d'une réponse

$$\mathrm{AG}(\mathit{request} \implies \mathrm{AX}\ \mathit{reply})$$

- le processus finira toujours par obtenir l'accès

$$\mathrm{AG}(\mathrm{AF}\ \mathit{access})$$

CTL dans Romeo

Version allégée de CTL

- les opérateurs EX et AX sont absents
- un seul opérateur temporel par formule (pas de formules temporelles imbriquées)
- un opérateur (φ) --> (ψ) équivalent à AG($\varphi \implies$ AFψ)

La logique de Romeo est donc moins expressive que CTL, mais Romeo permet aussi d'exprimer et de vérifier des propriétés "temps-réel" (pas dans ce cours).

Peut-on tout vérifier ?

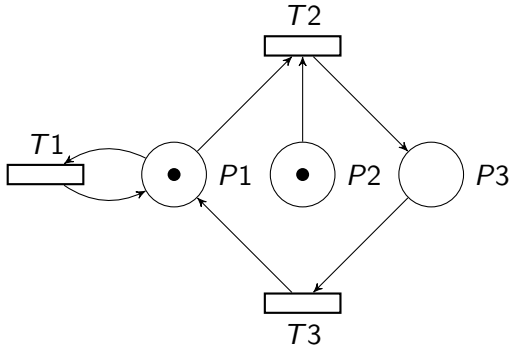
1. Peut-on tout exprimer ?
2. Peut-on tout décider ?

Expressivité de CTL

- une formule CTL = un ensemble d'arbres (formellement, un *langage* d'arbres)
- est-ce que à n'importe quel ensemble d'arbre correspond une formule CTL ?
  - problème : difficile de définir ce que veut dire "n'importe quel ensemble"
- existe-t-il d'autres façons de définir des langages d'arbres ?
  - automates d'arbres
  - grammaires formelles
  - un programme (ou une machine de Turing) qui prend la description d'un arbre en entrée et renvoit oui/non
  - ...
- on peut définir avec ces outils des langages que CTL ne peut pas définir

Les limites pratiques de l'expressivité de CTL

- soit la propriété "dans chaque branche, la formule φ restera toujours satisfaite à partir d'un moment"
- utile pour définir l'état final du système
- les formules suivantes expriment-elles correctement cette propriété ?
  1. $\mathsf{AG}\varphi$
  2. $\mathsf{AF}(\mathsf{EG}\varphi)$
  3. $\mathsf{AF}(\mathsf{AG}\varphi)$



Soit φ la formule $\equiv M(P1) > 0$

1. φ finit-elle toujours par être toujours vraie ?
2. le réseau satisfait-il AF(AGφ) ?

Décidabilité de CTL

✓ CTL est décidable sur les automates d'états finis (et donc sur les réseaux de Petri bornés)

- intuition : si RdP est borné, on peut représenter l'arbre des marquages par un graphe fini
- un chemin infini dans un graphe fini a forcément un cycle, on peut donc vérifier les propriétés de ces chemins

Décidabilité de CTL

- ✓ CTL est décidable sur les automates d'états finis (et donc sur les réseaux de Petri bornés)
  - intuition : si RdP est borné, on peut représenter l'arbre des marquages par un graphe fini
  - un chemin infini dans un graphe fini a forcément un cycle, on peut donc vérifier les propriétés de ces chemins
- ✗ CTL n'est pas décidable sur les réseaux de Petri non bornés
  - problème : si le nombre de marquage accessible est infini, le graphe des marquages l'est aussi
  - indécidable ne veut pas dire impossible dans tous les cas (en particulier, on peut touver un contre-exemple de longueur finie s'il existe)
  - sinon, on peut vérifier que la propriété tient jusqu'à une certaine profondeur dans l'arbre de marquages (*bounded model checking*)

$$\mathsf{AG}_{\#\leq 10}\varphi$$

Section 3

Aller plus loin dans la modélisation et la vérification

Système vs environnement

La plupart du temps, le modèle (réseau de Petri) représente à la fois

- le système que l'on souhaite concevoir et vérifier
- son environnement
  - évènements externes non-déterministes (exemple : bouton de feu piéton)
  - autre système avec lequel on interagit mais qu'on ne peut contrôler

Importance de bien représenter l'environnement

- si la modélisation de l'environnement est trop permissive : le model-checker peut trouver des contre-exemples fallacieux (faux positifs)
  - exemple : le model-checker retourne un contre-exemple où l'ascenceur passe directement de l'étage 1 à 3.
- si la modélisation est trop restrictive, le model-checker ne trouvera pas certains bugs même si le système est incorrect (faux négatifs)
  - la propriété $\mathrm{AG}(\mathit{request} \implies \mathrm{AF}\mathit{reply})$ est trivialement vraie si le modèle ne permet pas à *request* d'être vrai

Vérifier que le modèle de l'environnement est correct

- modélisation : si possible, séparer la modélisation de l'environnement et du système
  - distinguer ce qui peut survenir hors du contrôle du système (signaux d'entrée) et ce que le système doit faire (signaux de sortie)
- vérification : vérifier certaines propriétés de l'environnement, notamment la vivacité
  1. L'environnement ne s'arrête jamais
  2. La formule φ peut toujours devenir vraie
  3. La formule φ peut devenir vraie à n'importe quel moment

Vérifier que le modèle de l'environnement est correct

- modélisation : si possible, séparer la modélisation de l'environnement et du système
  - distinguer ce qui peut survenir hors du contrôle du système (signaux d'entrée) et ce que le système doit faire (signaux de sortie)
- vérification : vérifier certaines propriétés de l'environnement, notamment la vivacité
  1. L'environnement ne s'arrête jamais

     $$\mathrm{AG}(\neg \mathit{deadlock})$$

  2. La formule φ peut toujours devenir vraie

  3. La formule φ peut devenir vraie à n'importe quel moment

Vérifier que le modèle de l'environnement est correct

- modélisation : si possible, séparer la modélisation de l'environnement et du système
  - distinguer ce qui peut survenir hors du contrôle du système (signaux d'entrée) et ce que le système doit faire (signaux de sortie)
- vérification : vérifier certaines propriétés de l'environnement, notamment la vivacité
  1. L'environnement ne s'arrête jamais

     $$\mathrm{AG}(\neg \mathit{deadlock})$$

  2. La formule φ peut toujours devenir vraie

     $$\mathrm{AG}(\mathrm{EF}\varphi)$$

  3. La formule φ peut devenir vraie à n'importe quel moment

Vérifier que le modèle de l'environnement est correct

- modélisation : si possible, séparer la modélisation de l'environnement et du système
  - distinguer ce qui peut survenir hors du contrôle du système (signaux d'entrée) et ce que le système doit faire (signaux de sortie)
- vérification : vérifier certaines propriétés de l'environnement, notamment la vivacité
  1. L'environnement ne s'arrête jamais

$$\mathrm{AG}(\neg \mathit{deadlock})$$

  2. La formule φ peut toujours devenir vraie

$$\mathrm{AG}(\mathrm{EF}\varphi)$$

  3. La formule φ peut devenir vraie à n'importe quel moment

$$\mathrm{AG}(\mathrm{EX}\varphi)$$

Raisonner sur les transitions

- CTL se concentre sur les états du système (marquages)
- parfois il est utile de raisonner sur les actions (transitions)
- on peut utiliser CTL pour exprimer "la transition T n'est jamais franchissable"

- impossible d'exprimer "la transition T_2 n'est jamais franchie avant T_1" ?

Raisonner sur les transitions

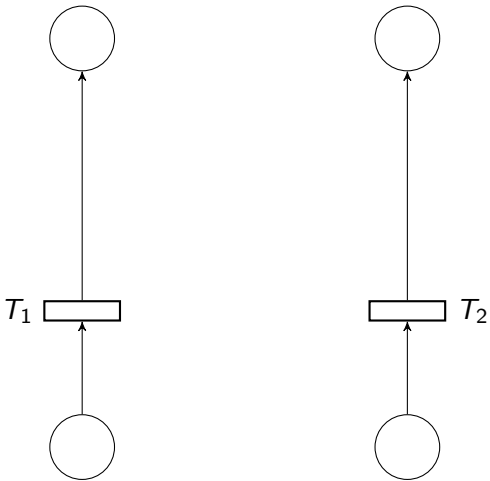
- CTL se concentre sur les états du système (marquages)
- parfois il est utile de raisonner sur les actions (transitions)
- on peut utiliser CTL pour exprimer "la transition T n'est jamais franchissable"

$$\mathsf{AG}\neg\varphi_T$$

où φ_T est une contrainte qui décrit les états où T est franchissable (dépend des places avec un arc vers T)

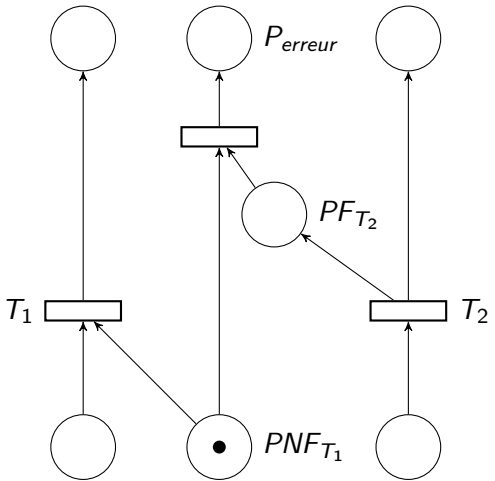
- impossible d'exprimer "la transition T_2 n'est jamais franchie avant T_1" ?

Vérifier la séquentialité avec un observateur



- comment vérifier "la transition T_2 n'est jamais franchie avant T_1" ?
- ajout d'un sous-réseau "observateur"

Vérifier la séquentialité avec un observateur



- comment vérifier "la transition T_2 n'est jamais franchie avant T_1" ?
- ajout d'un sous-réseau "observateur"
- on utilise le model-checker pour vérifier $\mathrm{AG}(M(P_{erreur}) = 0)$
- à utiliser avec modération
  - mélange modélisation et vérification
  - risque de modifier le comportement du modèle par inadvertance

Extensions des réseaux de Petri : temps réel

Time Petri Nets

- une extension des réseaux de Petri qui ajoute du temps réel
- pour chaque transition, un intervalle de temps est indiqué
- chaque transition est équipée d'une "horloge" qui se met en route (depuis 0) lorsque la transition est franchissable
- la transition ne peut être réelement franchie que quand l'horloge est dans l'intervalle de la transition

note : il existe d'autres formalismes "temps-réel" pour les réseaux de Petri

Extensions des réseaux de Petri : temps réel

Relation entre réseaux avec et sans temps réel

- RdP standard équivalent à un Time Petri Net où tous les intervalles sont $[0, \infty[$
- les conditions d'horloge ne font qu'empêcher certaines exécutions = élaguer certaines branches de l'arbre des marquages
  - si un RdP satisfait une propriété de sûreté, rajouter des conditions d'horloge préservera la sûreté
  - pour les propriétés de vivacité, ce n'est pas forcément le cas

Extensions des réseaux de Petri : arcs

Arc lecteur (*reader arc*)

- arc entrant : entre un place P et une transition T
- la condition de franchissement est la même que les arcs standards
- les jetons ne sont pas consommés lors du franchissement

Arc inhibiteur (*inhibitor arcs*)

- arc entrant : entre un place P et une transition T
- T ne peut pas être franchie si P contient des jetons
- T peut avoir d'autres arcs entrants qui imposent des condition de franchissement, comme d'habitude

Arc réinitialisateur (*reset arc*)

- arc sortant : entre une transition T' et une place P'
- lorsque T' est franchie, le nombre de jetons dans P' passe à 0

Extensions des réseaux de Petri : variables

Dans Romeo

- possibilité de définir un ensemble de variables et de définir leurs valeurs initiales
- à chaque transition, possibilité d'ajouter
  - condition sur les variables, la transition n'est franchissable que si la condition est vrai
  - commandes d'affectations pour modifier la valeur des variables lors du franchissement

Autres logiques temporelles (1)

- Timed CTL
  - utilisée dans Romeo
  - chaque opérateu temporel prend aussi en paramètre un intervalle de temps sur lequel il s'applique
- Linear Temporal Logic (LTL)
  - opérateurs X, F, G et U
  - ni A ni E, les formules ne parlent pas de branches mais sont interprétées comme s'appliquant à n'importe quelle branche
  - permet de définir certaines propriétés impossibles à exprimer avec CTL, comme "φ finira par être toujours vraie"
- CTL*
  - opérateurs X, F, G et U
  - quantificateurs A et E qui peuvent apparaître indépendamment des opérateurs
  - plus expressif que CTL ou LTL

Autres logiques temporelles (2)

- μ-calculus
  - permet de définir ses propres opérateurs de manière récursive
  - plus expressif que CTL*
- logiques d'intervalles temporels
  - plusieurs logiques pour raisonner sur des intervalles plutôt que des instants
  - le plus souvent, pas de branches

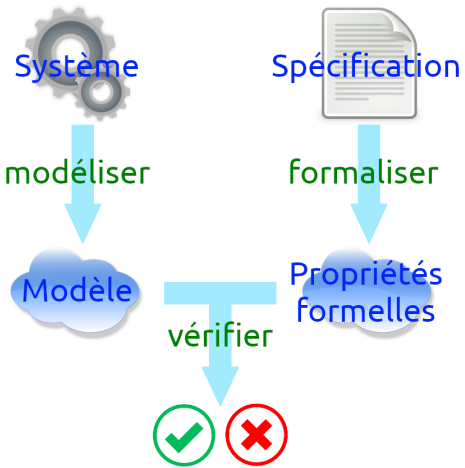
Expressivité vs décidabilité

- un formalisme plus expressif permet de modéliser des systèmes impossibles sinon
- mais rend la vérification plus difficile
  - ✓ vérifier si un RdP borné satisfait une formule CTL est décidable
  - ✗ indécidable en général sur les RdP non-bornés
  - ✓ l'accessibilité d'un marquage est décidable sur les RdP standards
  - ✗ indécidable en général sur les RdP avec arcs inhibiteurs
  - ✗ déterminer si un Time Petri Net est borné est indécidable

Choisir le bon outil pour le but poursuivi

- un formalisme plus expressif n'est pas meilleur si on n'arrive pas à prouver les propriétés qui nous intéressent !
- le même conseil s'applique pour le choix du langage formel de spécification

Les limites de l'abstraction



L'outil de vérification n'offre des garanties que sur la partie formalisée (abstraite)

- Comment s'assurer que les propriétés formelles correspondent bien à l'intention de la spécification ?
  - écrire une spécification formelle correcte (et complète) demande de l'expérience
  - il faut souvent débuguer la spécification elle-même
- Comment s'assurer que le système correspond bien au modèle ?

Implémentation vérifiée

Approche top-down

- partir du modèle et s'assurer que l'implémentation corresponde
  - extraction de code = compiler un modèle dans un langage de programmation
  - raffinement = modèles de plus en plus précis en prouvant que les propriétés sont préservées
- suit le processus de conception
- exemple : méthode B, outil utilisé notamment pour vérifier le logiciel embarqué de la ligne Meteor à Paris

Approche bottom-up

- extraction de modèle = écrire du code et en sortir automatiquement un modèle
- permet de faire de la vérification sans connaître le modèle formel
- difficile car les langages de programmation ont une sémantique beaucoup plus complexe que les RdP ou autres formalismes