



Vérification de systèmes hybrides objets - règles

Julie Vachon

DIRO, Université de Montréal (Canada)

vachon@iro.umontreal.ca

Avec la collaboration de

- Houari A. Sahraoui (Université de Montréal)
- Mustapha Essalih (CRIM, Montréal)
- Hafedh Mili (UQAM, Montréal)

Contexte

- Les compagnies développent une expertise croissante toujours plus complexe à décrire.

Exemples:

- Système de production d'aluminium.
- Système de planification de ressources hydro-électriques.

- Les systèmes informatiques doivent être modifiés constamment pour en donner une image fidèle.

- Format des données (stable)
- Traitements (description changeante)

- Conséquence -

Maintenance coûteuse:

- Système existant fermé
- Écrit en Fortran
- Basé sur des modèles mathématiques de simulation complexes.

Coûts et délais de maintenance difficilement supportables dans le contexte économique actuel.



- Solution acceptable -

Permettre aux utilisateurs (non informaticiens) d'apporter eux-mêmes les modifications.



Contexte

- Les systèmes hybrides objets - règles offrent une solution intéressante au problème de l'évolution.
 - Les données stables sont représentées par des objets.
 - La logique de planification est représentée par des règles.
- L'évolution consiste à modifier uniquement les règles, dans un style déclaratif.

Systemes objets – règles (éditeur)

Les modifications peuvent être faites grâce à un éditeur de règles simplifié

Nouvelle règle 1

Nom : risqueCTaLSJ Priorité : 4 Groupe:

Si
LSJ.deversementMoyenPrevuDansNjours(pc, 7) > 100 ;
Alors
LSJ.risqueDeversementDansCourtTerme = 100 ;

Nouvelle règle 1

Nom : risqueCTaLSJ Priorité : 4 Groupe:

Si
LSJ
debitTotalDeversement(pc)
debitTotalEvacue(pc)
debitTotalTurbine(pc)
perteChargeCanalCCetCS(pc)
perteChargeCanalLSJetIM(pc)
productionMaximaleDisponible(pc)
productionOptimale(pc)
productionReelle(pc)

Systeme objets – regles (code)

La regle generatee est (JRules de ILOG)

```
rule risqueCTaLSJ {  
    packet = risqueCTaLSJ;  
    priority = 4 ;  
    when{  
        ?sim: Simulation() ;  
        ?site: Site(?site.getCode().equals(new  
String("LSJ"))) ;  
        ?site.deversementMoyenPrevuDansNjours(?sim.g  
etPasCourant(), 7).compareTo(100.f) > 0) ; }  
    then {  
        modify ?site{  
            ?site.setRisqueDeversementCourtTerme(?sim  
.getPasCourant(), 100.f);  
            System.out.println("risqueCTaLSJ1"); } } }  
}
```

→
conditions

→
actions

Systemes objets- regles:

- JRules
- OPS-2000
- CLIPS
- ...

Systemes objets - regles

R : condition **->** **action**

R1	LHS ₁	→	RHS ₁
R2	LHS ₂	→	RHS ₂
R3	LHS ₃	→	RHS ₃
...			

Mémoire de travail

o1, o2, o5, o6, o11

Moteur d'inférence
(algorithme de
résolution de conflits)

Match set

Activation

Agenda

(ens. de conflits)

(R1, {o2, o6})

(R1, {o2, o5})

(R3, {o1, o2})

Activation: (R1, {o2, o5})
Action: RHS1(o2, o5)

Problème...

- Et si l'utilisateur se trompait:

Si

```
LSJ.deversementMoyenPrevuDansNjours (p  
c, 7) > 100
```

Alors

```
LSJ.risqueDeversementDansCourtTerme (pc)  
= -100
```

Violation d'un invariant...

Le problème

Comment s'assurer que la base de règles demeure correcte après les modifications des utilisateurs ?

- Les invariants du système sont-ils maintenus ?
- Le système contient-il des règles redondantes ?
- Le système de règles est-il complet ?
- Peut-on dériver des jugements inconsistants ?
- Le système peut-il se bloquer ?

SOLUTION: intégrer un module de vérification automatique dans le système

- Des méthodes de vérification existent pour les systèmes de règles classiques.
- MAIS ne s'appliquent pas aux systèmes objets – règles.

Pourquoi ?

Le problème

Particularités des systèmes objets - règles

- Non-monotonie
 - Système classique:
 - État d'un système = ensemble de valeurs de vérité associées aux faits.
 - Lorsqu'un fait est inféré, il demeure vrai jusqu'à la fin de session de l'inférence.
 - Système objets - règles:
 - État d'un système : ensemble des états des objets.
 - Au cours d'une session, l'état d'un objet peut être faux, devenir vrai et redevenir faux, etc.
- Héritage et sous-typage (polymorphisme).
 $\{(Personne \text{ age} \geq 18) p1\} \rightarrow (\text{modify } p1 \text{ peutBoire} = \text{true})$
 $\{(Homme \text{ age} \geq 18) p2\} \rightarrow (\text{modify } p2 \text{ peutBoire} = \text{true})$
- Absence d'axiomatisation des méthodes.
 $\{(Personne \text{ dateNaissance} \leq 1986) p\} \rightarrow (\text{modify } p \text{ peutBoire} = \text{true})$
 $\{(Personne \text{ lireAge}() \geq 18) p\} \rightarrow (\text{modify } p \text{ peutBoire} = \text{true})$

Cadrage du problème et hypothèses

Expérience antérieure:

- Module de vérification dynamique, exploration exhaustive de tous les chemins d'exécution.
- Solution peu efficace, manque de formalisation.

Nouvelle approche: Model-checking

- *La plupart des propriétés intéressantes d'un système sont algorithmiquement prouvables si le système peut être modélisé par un automate à états finis.*
- **Difficulté:** ... et si le système comporte une infinité d'états ??

Hypothèse

Nous considérons des systèmes finis i.e. composés d'un nombre fini d'objets eux-mêmes comportant un nombre fini d'états (on peut supposé le domaine des attributs borné).

Cette hypothèse est justifiée par le domaine des applications considérées soient des systèmes de gestion de ressources finies.

Approche de vérification

1. Modélisation du système objets - règles à l'aide de réseau de Petri colorés.

[Outil: Design CPN]



2. Calcul du graphe d'états du RdPC obtenu.

[Outil: Design CPN]



3. Formulation des propriétés à vérifier en logique temporelle.

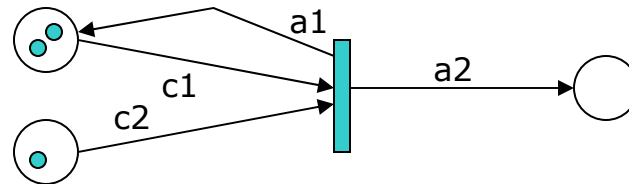
[Formalisme: CTL]



4. Model - checking: Vérification automatique des propriétés formulées sur le graphe d'états.

[Outil: model-checker]

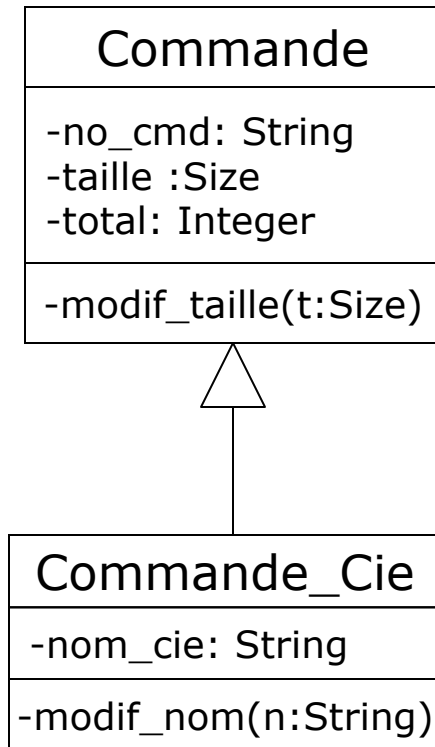
Modélisation en réseaux de Petri colorés



Systeme objets- règles	Réseaux de Petri colorés (CPN Design)
Objets	Structures de données ML.
Règles	Transitions
○ Conditions	○ Pré-conditions (arcs entrants)
○ Actions	○ Post-conditions (arcs sortants)
État initial	Marquage initial

Modélisation des objets

Classe



```
color Commande = record
```

```
ident : Object *
```

```
cmd_id : string
```

```
* taille : size
```

```
* total : int;
```

Identité

Attributs

Méthode

```
fun modif_taille(nt,  
    {cmd_id=i, taille=t, total=tt})  
    = {cmd_id=i, taille=nt, total=tt}
```

Sous-classe
(héritage par
extension)

```
color Commande_cie = record  
    commande:Commande * nom_cie:string;
```

```
fun modif_nom(n,  
    {commande=c, nom_cie=x})  
    = {commande=c, nom_cie=n}
```

Modélisation des objets

○ Sous-typage et substitutabilité

- Standard ML : polymorphisme « paramétrique ».
Ex. `fun idem a = a`
- Java : polymorphisme « structurel ».

Pbm: Comment permettre aux objets d'une sous-classe d'être soumis aux mêmes règles que les objets de la super-classes ?

Sln: *Regrouper toutes les couleurs d'une même hiérarchie sous une couleur commune à l'aide des unions disjointes.*

```
color all_commandes = union  
T_commande : Commande +  
T_commande_cie : Commande_cie;
```

Pour savoir si `c` est bien une commande d'entreprise:

```
Of_T_commande_cie' all_commandes(c)
```

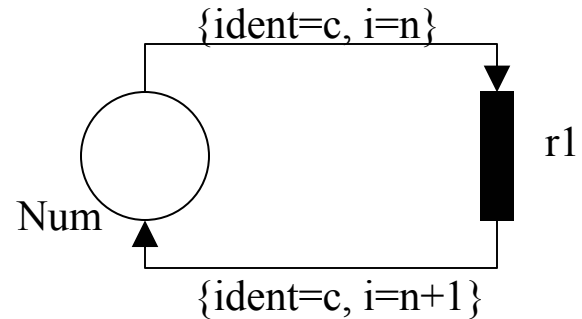
Modélisation des règles

Selon le type d'action....

Rule r1

Réfraction:

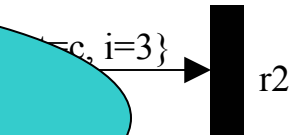
Une règle ne peut tirer les mêmes données sans qu'elles aient été modifiées.



Rule r2

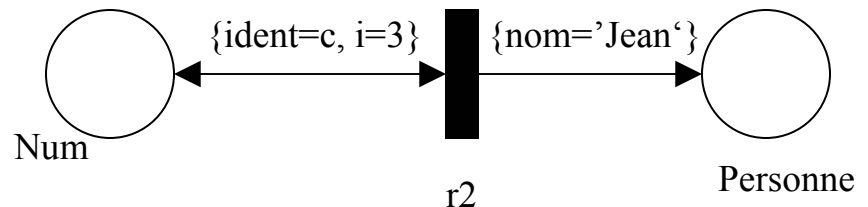
When { ?c, i=3 }
Then { retract ?c }

Comment faire pour empêcher cette règle d'être tirée indéfiniment ?



Rule r3

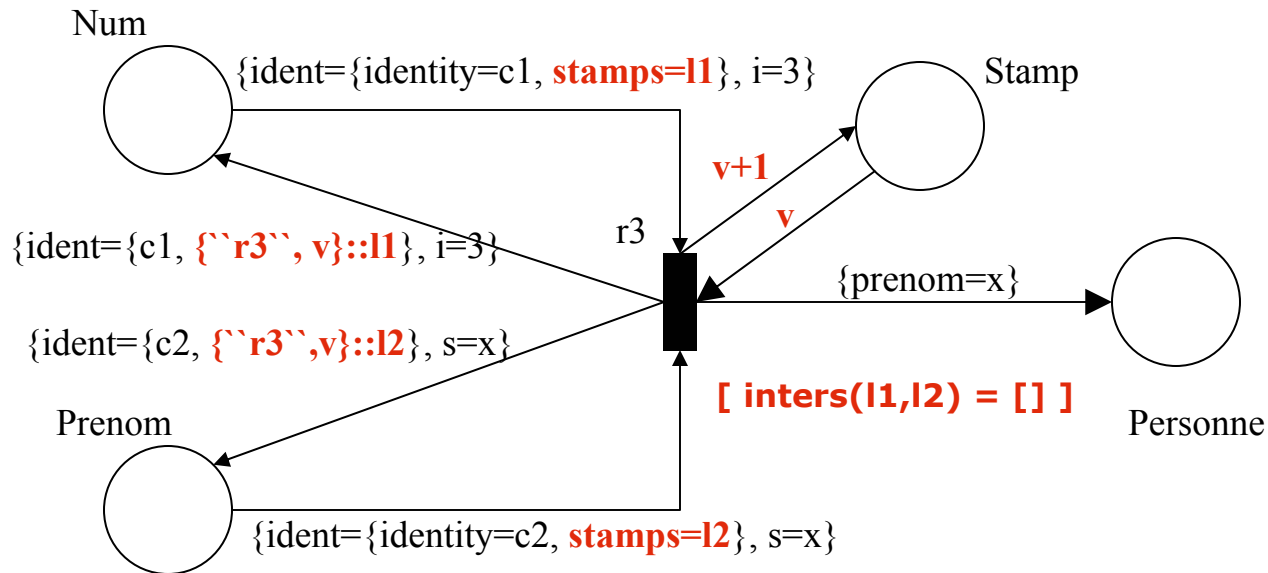
When { Num(i==3) }
Then { assert Personne(« Jean ») }



Modélisation des règles

Règle avec action « assert »

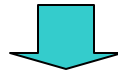
```
Rule r3
When { ?c1 : Num(i=3);
      ?c2 : Prenom(s= ?x) }
Then { assert Personne(prenom==?x) }
```



Approche de vérification

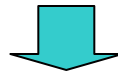
1. Modélisation du système objets - règles à l'aide de réseau de Petri colorés.

[Outil: Design CPN]



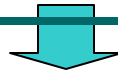
2. Calcul du graphe d'états du RdPC obtenu.

[Outil: Design CPN]



3. Formulation des propriétés à vérifier en logique temporelle.

[Formalisme: CTL]



4. Model - checking: Vérification automatique des propriétés formulées sur le graphe d'états.

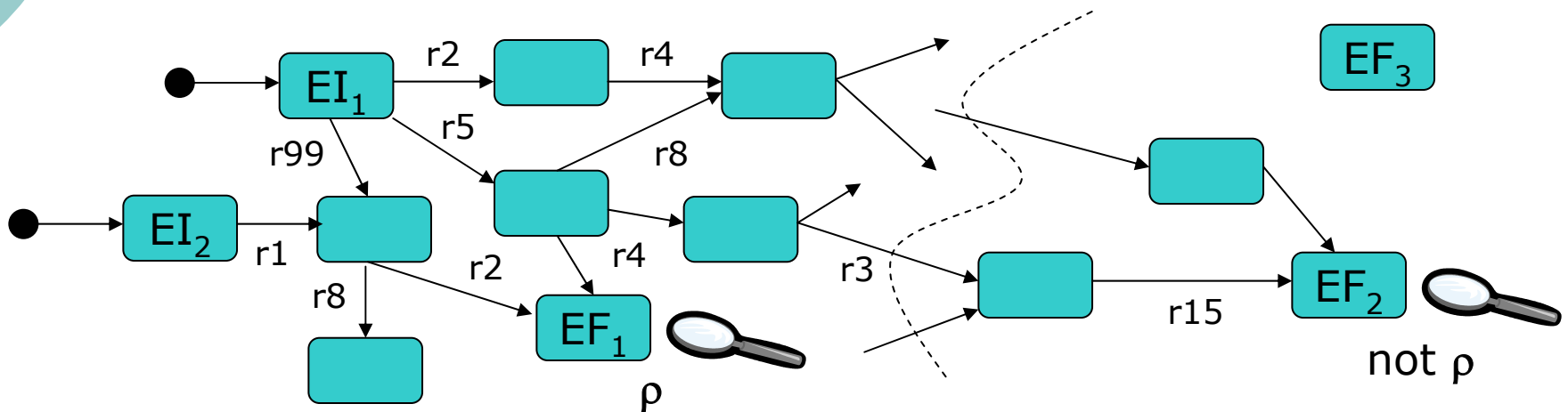
[Outil: model-checker]

Calcul du graphe d'états

Systeme

<EnsClasses, EnsRègles, EnsÉtatsInit, EnsÉtatFin>

Graphe des états



- Le graphe permet d'identifier toutes les exécutions possibles du système.
- Peut révéler des anomalies...



Les anomalies

- Défauts d'atteignabilité
 - Redondance.
 - Incomplétude.
- Défauts de vivacité
 - États de blocage (qui ne sont pas de états finaux).
- Défauts de sûreté
 - Inconsistance externe (contradictions des jugements finaux).
 - Inconsistance interne (invariants non vérifiés).

Formulation des propriétés à vérifier

La logique temporelle

spécifiquement conçue pour les énoncés et les raisonnements qui impliquent une notion d'ordonnancement dans le temps.

$$\pi ::= X\phi \mid F\phi \mid G\phi \mid \phi U \phi$$

$$\phi ::= pa \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid E\pi \mid A\pi$$

- $X\phi$ « le prochain état de l'exécution courante satisfait ϕ ».
- $F\phi$ « il existe un futur état de l'exécution courante qui satisfait ϕ ».
- $G\phi$ « tous les futurs états de l'exécution courante satisfont ϕ ».
- $E\phi$ « il existe une exécution, depuis l'état courant, qui satisfait ϕ ».
- $A\phi$ « toutes les exécutions, depuis l'état courant, qui satisfont ϕ ».

Formulation des propriétés (CTL)

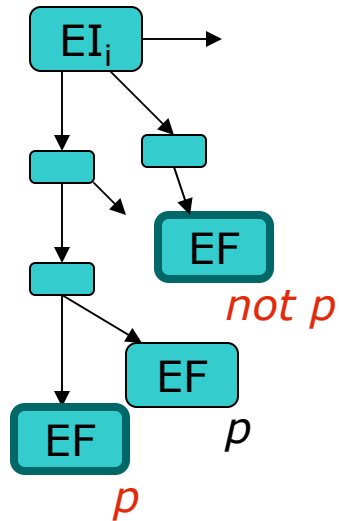
On définit un automate A_i pour représenter la portion du graphe d'états décrivant les états accessibles depuis le i^{e} état initial.

Inconsistance

Soit S un système dont le comportement est décrit par une famille d'automates $(A_i)_{1 \leq i \leq r}$ et un ensemble de propositions finales à observer. Le système S est inconsistant si et seulement si

$$\exists i (1 \leq i \leq r) \exists p \in Obs$$

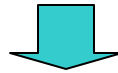
$$A_i \models \text{EF}(final_i \wedge \neg X \text{ true} \wedge p) \wedge \text{EF}(final_i \wedge \neg X \text{ true} \wedge \neg p)$$



Approche de vérification

1. Modélisation du système objets - règles à l'aide de réseau de Petri colorés.

[Outil: Design CPN]



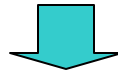
2. Calcul du graphe de marquages (graphe d'états) du RdPC obtenu.

[Outil: Design CPN]



3. Formulation des propriétés à vérifier en logique temporelle.

[Formalisme: CTL]



4. Model - checking: Vérification automatique des propriétés formulées sur le graphe d'états.

[Outil: model-checker]

Model - checking

Outils de model-checking

- Entrée: un automate fini et une formule f de logique temporelle (CTL)
- Sortie : réponse à la question la formule f est-elle vérifiée ? Si non, une exécution violant la propriété f est exhibée.
- Outils existants: SMV, SPIN, model-checker de CPN design, etc.
- Limite: Explosion combinatoire du nombre d'états à explorer!

Conclusions

- Nous avons
 - modèle formel du système de règles à vérifier et son graphe d'états.
 - Une formulation en CTL des propriétés à vérifier.
 - Des model-checkers à disposition.
- Un prototype est en développement.
- Le grand problème envisagé : l'explosion combinatoire du nombre d'états.
 - Optimisations et des abstractions permettant de réduire le nombre d'états.
 - Exécution symbolique.
 - Techniques de vérification incrémentale