Combiner Méthodes Formelles et Simulation à évènements discrets

A. YACOUB, M. HAMRI, C. FRYDMAN

Laboratoire des Sciences de l'Information et des Systèmes (LSIS UMR 7296)
Aix-Marseille University
France

JDF'16
Journées DEVS francophones
11-15 avril 2016, Cargèse, France

Contents

Contents

Introduction

- La compréhension et la mise au point de systèmes fiables (software, hardware...) nécessite de plus en plus de compétences et de connaissances.
- La compréhension de systèmes (évènements climatiques, feux de circulation...) est aussi difficile en raison de la quasi-impossibilité de faire des tests sur le système réel.
- Deux domaines dans la littérature proposent des solutions : **La vérification formelle (FV)** et **la Modélisation et Simulation (M&S)**.

Vérification et Validation

Le processus de Vérification et Validation (V&V) est une phase importante dans le développement d'un système. (PMBOK Guide, 2011)

- *La Validation* consiste à s'assurer qu'un produit est conforme aux attentes utilisateurs (conformité au cahier des charges). "Avons-nous fait le bon produit ?"

- *La Vérification* consiste à s'assurer qu'un produit est conforme à un ensemble de contraintes de conception (spécification) imposées. "Faisons-nous le produit correctement ?"

Vérification et Validation

Dans le cadre logiciel,

- *La Validation* consiste donc à tester dynamiquement le logiciel en condition réelle, donc à exécuter le code. La validation permet alors de s'assurer de la conformité du logiciel aux spécifications.

- *La Vérification* consiste en un ensemble de tests statiques et dynamiques pour vérifier que le code logiciel est conforme à la conception.

Vérification et Validation

Dans le cadre de la Modélisation et Simulation (M&S),

- *La Validation* consiste à s'assurer que le modèle de simulation représente bien le système étudié avec un degré de précision suffisant pour les utilisations prévues pour le modèle.

- *La Vérification* consiste à s'assurer que le modèle de simulation (le modèle implémenté) est conforme au modèle conceptuel.

Vérification formelle

- Parmis les techniques de vérification, on trouve **la vérification formelle** = Action de prouver la fiabilité d'un système par rapport à un ensemble de spécifications et de propriétés en utilisant **les méthodes formelles**.
- Une méthode formelle est un ensemble d'une notation formelle et d'une sémantique formelle (Bowen, 1995).
- Deux grandes familles de méthodes formelles : le *theorem proving* et le *model checking*.
- Le model-checking consiste à vérifier que syntaxiquement et sémantiquement un modèle M (généralement un automate fini) donné satisfait une propriété ϕ. Principalement, le model-checking consiste donc à explorer (grâce à divers algorithmes) systématiquement la totalité de l'espace d'états [symboliques] d'un modèle.

Avantages du Model Checking

- Vérification exhaustive.
- Modèles basés sur la logique, syntaxiques, rigoureux et simples. On ne modélise que ce qui est intéressant pour la vérification des propriétés.
- Modèles non-déterministes présentés comme un avantage permettant de raisonner sans se soucier de la résolution du non-déterminisme dans le système.
- Algorithmes performant dans la grande majorité des cas.

Vérification formelle

- Prenons par exemple *PROMELA* - Process Meta Language (Holzmann, 1991).
- PROMELA : language de spécification + sémantique d'exécution pour représenter des processus concurrents.
- PROMELA permet d'écrire des **modèles de vérification** : une description du système + un ensemble de propriétés à vérifier sur le modèle (safety - "Rien de mauvais ne se produit", liveness - "Finalement, quelque chose de bien finit par se produire").
- Un modèle de vérification est un modèle qui se concentre uniquement sur les aspects conceptuels nécessaires pour la vérification des propriétés => Notion de *simplicité* => Abstraction de très haut-niveau.

Vérification formelle

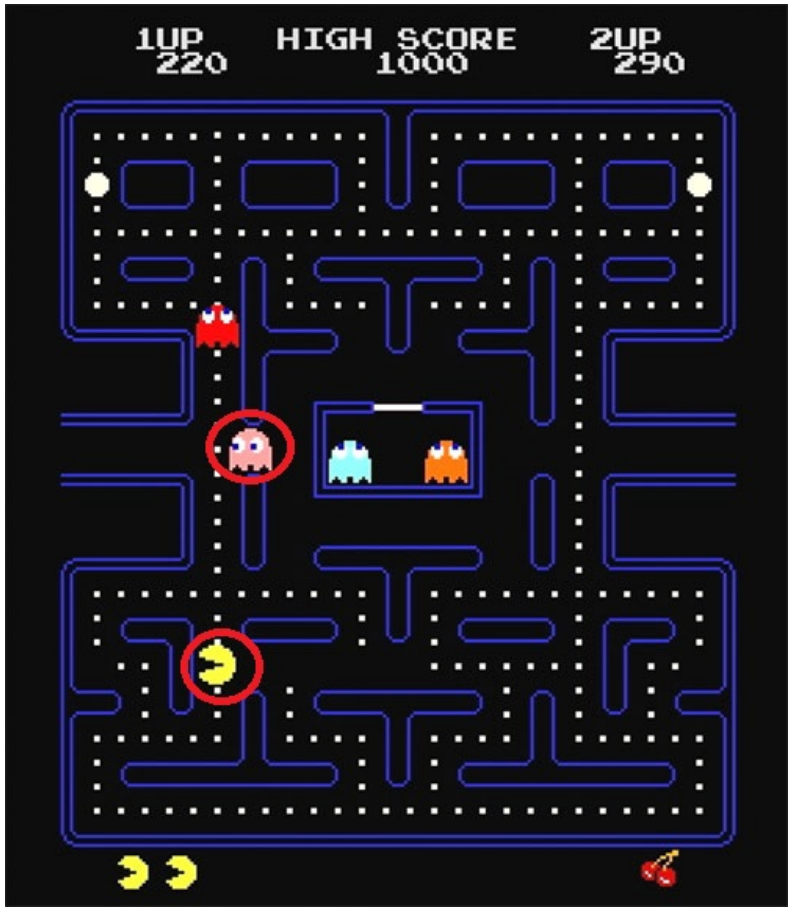
Problèmes :

- Un modèle de très haut-niveau oblige à faire abstraction de concepts importants.
- Toutes les classes de systèmes temporels ne sont pas vérifiables ou difficilement vérifiables (Alur, 1994).
- Problème d'explosion combinatoire récurrent (Clarke, 2008).
- $\Rightarrow$ Que se passe-t-il si les interactions entre composants dépendent de valeurs de données ou de leur coordination temporelle qu'on ne peut pas représenter ?

Prenons l'exemple du Pacman

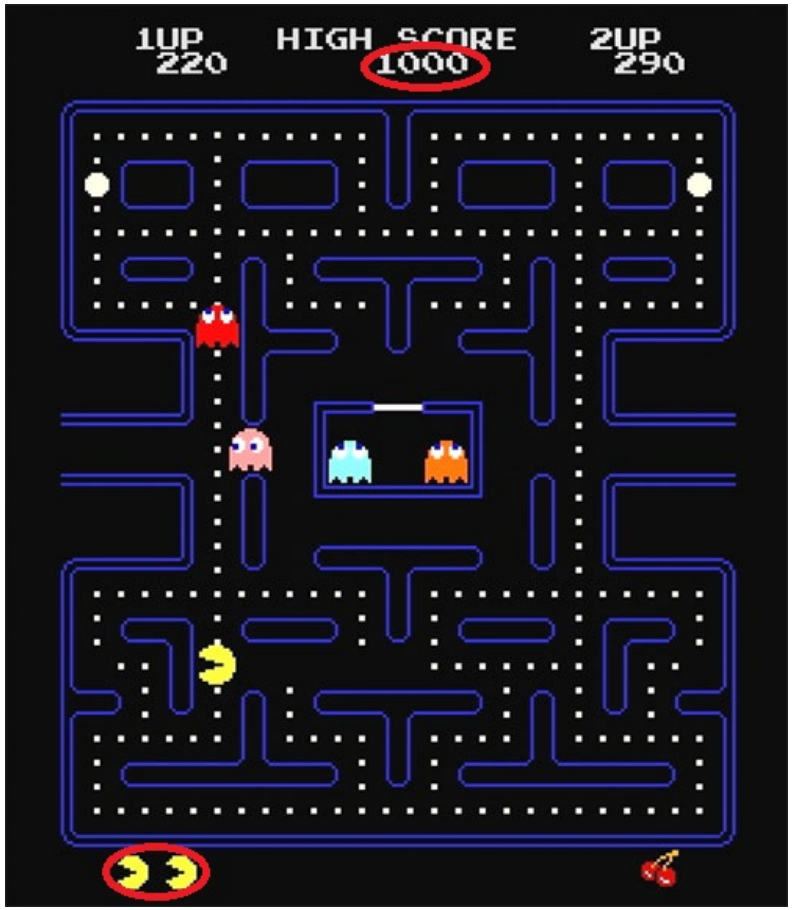
- Modélisons et vérifions des propriétés avec Pacman.

Prenons l'exemple du Pacman



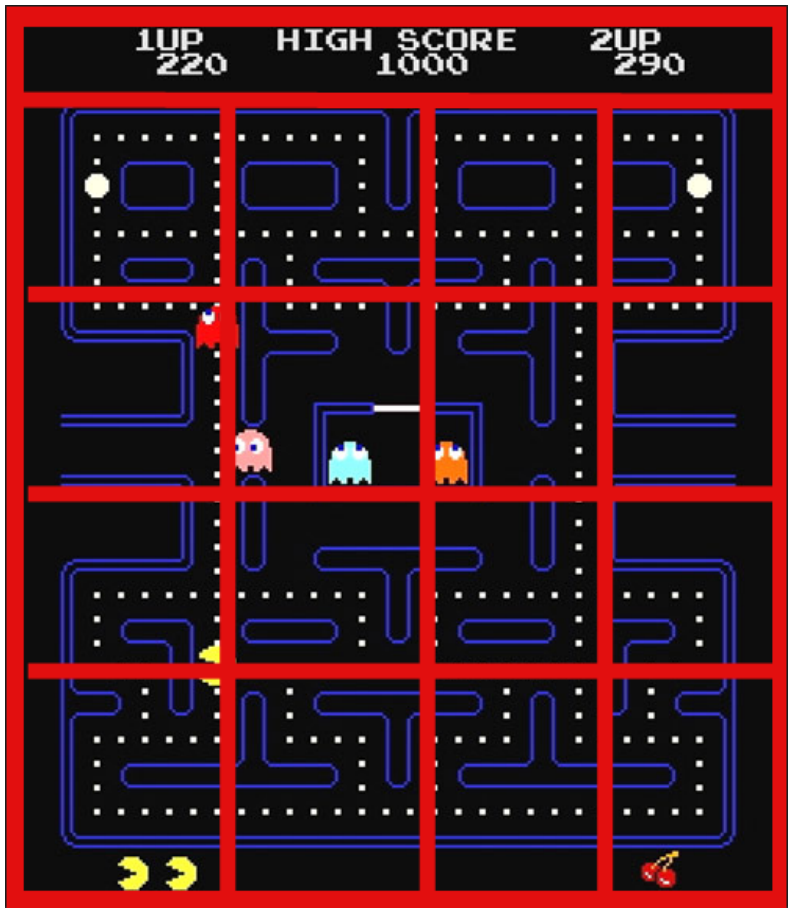
- Pacman, fantomes = FSM/processus asynchrones.

Prenons l'exemple du Pacman



- Score, vies = Entiers.

Prenons l'exemple du Pacman



- Position : nombre réel. Impossibilité d'utiliser les réels avec certaines méthodes formelles. Alors on utilise une **grille** et **des entiers**.

Prenons l'exemple du Pacman

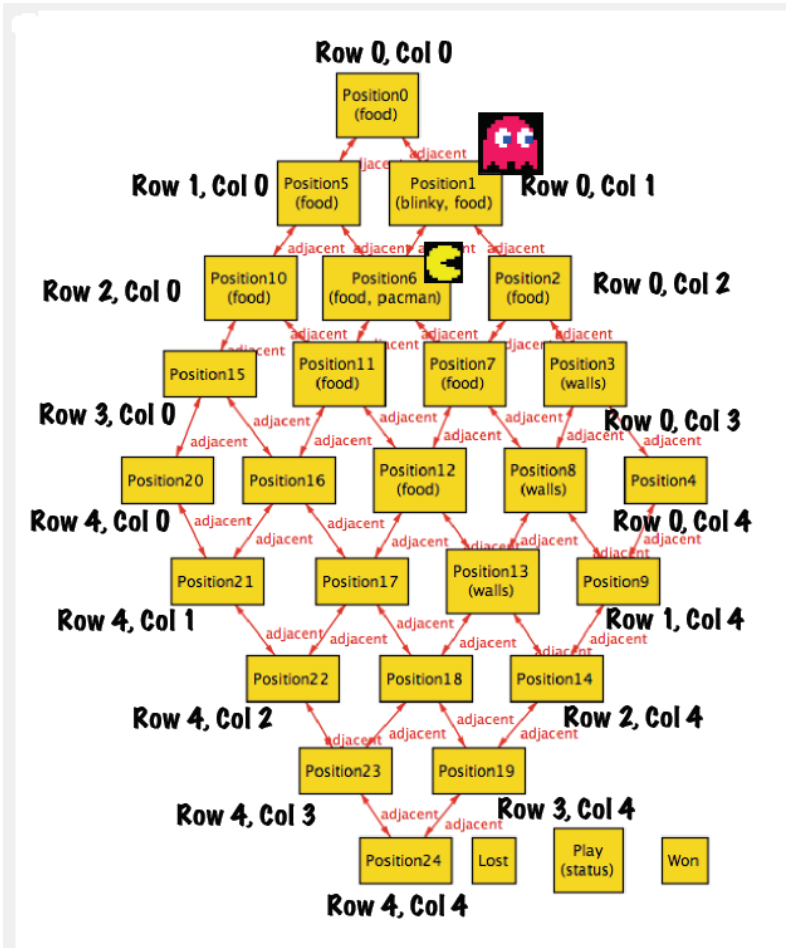
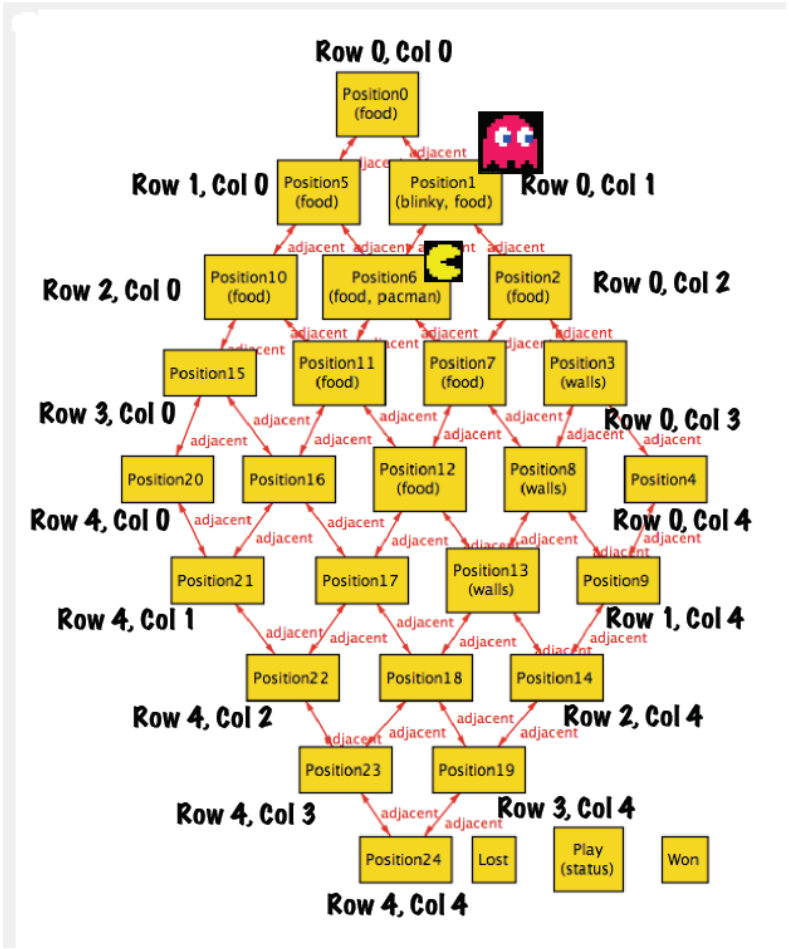


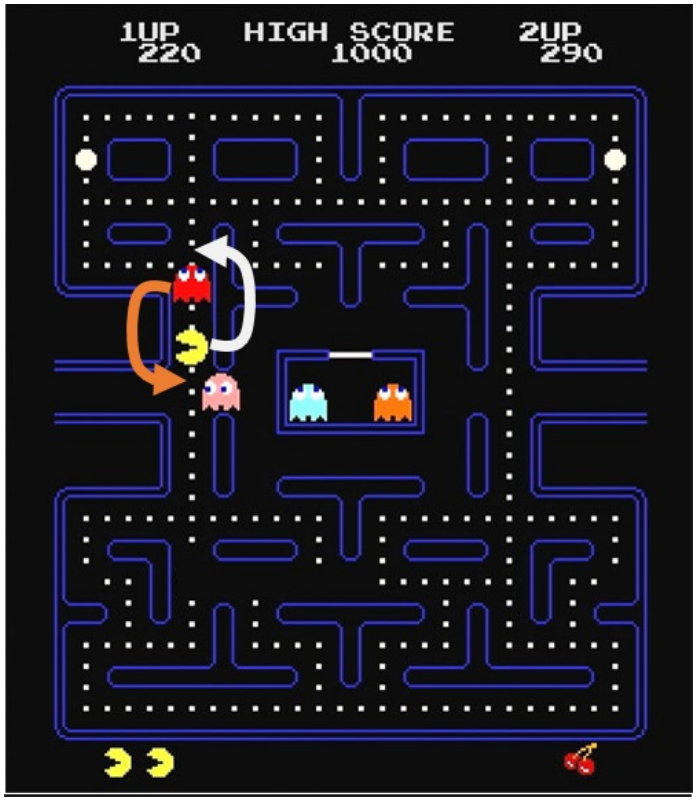
Figure : From "Conceptual Model Verification with Alloy", Ross Gore, SpringSim'15.

Prenons l'exemple du Pacman



- Vérifier les collisions = Pas d'état avec un pacman et un fantôme dans la même case. Pas de prise en compte de la vitesse.

Prenons l'exemple du Pacman



- Ok, donc ce modèle garantit qu'un Pacman et un fantôme ne peuvent pas se croiser sans collisioner. Mais il ne prend pas en compte la vitesse de déplacement qui dépend du temps écoulé entre deux états, c'est à dire le temps entre le calcul de deux images successives à l'écran. **Ce modèle n'est donc pas suffisant pour garantie cette propriété...**

Avec les Timed Automata

- Peut-on résoudre ce simple problème ?
- Solution 1 : Affiner la précision de la grille => explosion de l'espace d'états ;
- Solution 2 : Existe-t-il un autre formalisme vérifiable pour modéliser la vitesse dans l'espace des réels ?
- Les Timed Automata (Alur, 1994) permettent de prendre en compte la notion d'horloge si les contraintes sont linéaires. Mais ils ne permettent pas de modéliser l'interpolation. Et globalement, la vérification des systèmes temporels est difficile.

FV : En bref...

- L'objectif d'un modèle n'est pas de définir entièrement un système. Un modèle de vérification aide les designers à découvrir les erreurs et les mauvaises hypothèses avant implémentation. Mais :
  - Plus un modèle est précis, plus la probabilité que le système soit correct augmente.
  - Trop de détails dans un modèle de vérification conduisent à des erreurs de design, mais un trop grand niveau d'abstraction aussi...
- Le problème dans Pacman ? Le temps... Quelle solution ?

Une solution ?

En M&S, Zeigler (1976) définit un cadre et des techniques universelles de modélisation et simulation. Notamment,

- La séparation entre modélisation et simulation assure la réutilisabilité des modèles (modèles conceptuels vs modèles de simulation).
- Un formalisme (DEVS) et une sémantique opérationnelle claire (fournie par le simulateur abstrait) pour la représentation d'un ensemble large de systèmes.
- La notion de Cadre Experimental (EF) facilite la découverte d'hypothèses sur le système réel.

Une solution ?

- Des solutions de vérification à base de simulation existent malgré que la simulation n'est pas exhaustive.
- Des solutions combinant vérification formelle et simulation existent à base de remodélisation.
- Les MF sont utilisées pour vérifier les modèles de simulation... On peut donc combiner MF et DEVS.
- **Notre idée : Introduire la sémantique de DEVS au sein des méthodes formelles.**

Contents

Idée

- **Idée : Introduire dans le langage de spécification d'une méthode formelle, la sémantique du simulateur abstrait DEVS, en modifiant légèrement la sémantique opérationnelle de la MF si elle existe. Pourquoi ?**
  - On peut regénérer un modèle plus abstrait qui sera vérifiable en profitant de la puissance des algorithmes de vérification formelle, pour les invariants temporelles.
  - On peut regénérer un modèle de simulation pour la vérification à base de simulation pour les propriétés comportementales.
  - On peut alors combiner VF et Simulation sans remodéliser.
  - On peut faire un rapprochement entre simuler et exécuter. Important pour la validation.
- Un seul modèle conceptuel est nécessaire pour vérifier à base de model-checking et de simulation.

Construire Dev-PROMELA

- PROMELA (Holzmann, 1997), un langage de spécification qui permet la modélisation et la vérification de processus asynchrones.
- Un modèle PROMELA :
  - Un semble de processus asynchrones.
  - Un processus est un ensemble d'instructions et un semble de données de types "basiques" (entiers, booléens, bytes...).
  - La sémantique est donnée par un automate fini.
- On introduit des ajouts syntaxiques pour modéliser :
  - **Les évènements** : le modèle évolue selon des évènements qu'il émet ou qu'il reçoit ;
  - **Le temps écoulé entre chaque instruction** : chaque instruction est associée à un nombre réel qui indique le temps qui doit s'écouler entre la dernière instruction exécutée et celle-ci ;
  - **Types de transitions** : s'il s'agit d'une transition interne au sens des DEVS ou une transition externe.
  - **Réels** : Un type abstrait pour modéliser les nombres réels.

Construire DEv-PROMELA

Program 5 Fischer's Mutual Exclusion Protocol in Classic PROMELA.

```
1: int pid = 0;
2:
3: active proctype P ( int id ) {
4: do ::
5: /* non-critical section */
6: wait_L:
7:     if
8:     :: pid != 0 → goto wait_L;
9:     :: else → skip;
10:    fi;
11:    pid = id;
12:    if
13:    :: pid != id → goto wait_L;
14:    :: else → skip;
15:    fi;
16:    /* critical section */
17:    pid = 0;
18: od;
19: }
20:
```

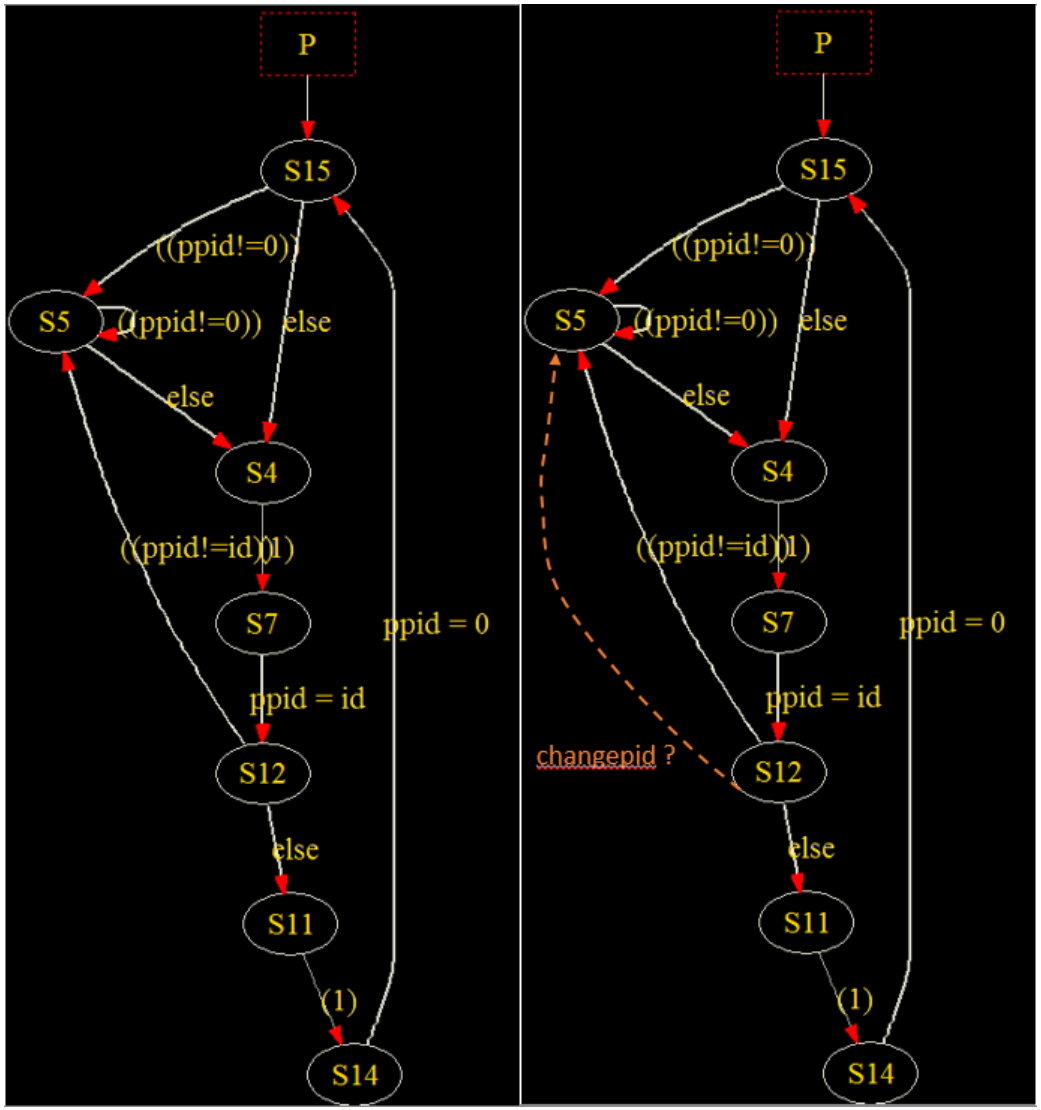
Program 6 Fischer's Mutual Exclusion Protocol in DE-PROMELA.

```
1: int pid = 0;
2: mtype = { changepid }
3:
4: [ priority = id ]
5: active proctype P ( int id ) {
6: real delay = 2.0 * id;
7: do ::
8: /* non-critical section */
9: wait_L:
10:    if
11:    :: [clt:0.1 → emit:silent] pid != 0 → goto wait_L;
12:    :: [clt:0.1 → emit:silent] else → skip;
13:    fi;
14:    [clt:0.1 → emit:changepid] pid = id;
15:    if
16:    :: [clt:delay → emit:silent | evt: changepid]
17:       pid != id → goto wait_L;
18:    :: [clt:delay → emit:silent] else → skip;
19:    fi;
20:    /* critical section */
21:    [clt:0.1 → emit:changepid] pid = 0;
22: od;
23: }
24:
```

Construire DEv-PROMELA

- La sémantique d'un modèle DEv-PROMELA = semantique donnée par le simulateur DEVS. Un DEv-PROMELA est simulable.
- Un DEv-PROMELA peut-il être vérifié ? Oui puisque le modèle PROMELA atemporel à la même sémantique que le modèle temporel pour les propriétés structurelles.

Construire DEv-PROMELA



Construire DEv-PROMELA

- Structurellement, un DEv-PROMELA sans notions d'évènements, de temps... = une bonne abstraction PROMELA.
- VF $\rightarrow$ Recherche de deadlock induit par la structure (absence de transition...) ou des propriétés indépendantes du temps écoulées.
- Simulation $\rightarrow$ Verification de propriétés dépendantes du temps écoulé et validation du modèle conceptuel.

Avantages

- On peut modéliser ce qui n'était pas modélisable avant (ou difficilement).
- On peut vérifier ce qu'on ne pouvait pas vérifier avant par simulation, tout en gardant la puissance de la vérification formelle (le modèle est vérifiable et simulable).
- On peut réduire en partie le problème d'explosion combinatoire (la temporalité implique l'ordonnancement => tous les chemins n'ont pas tous un sens dans le monde réel).
- On passe d'une abstraction très haut-niveau à une abstraction plus bas-niveau.
- Un seul et unique modèle, contrairement aux techniques par transformation explicite.

Avantages

- Un nouveau formalisme syntaxique avec la sémantique de DEVS.
- Plus facile de vérifier le modèle de simulation à partir de règles de réécriture depuis un DEv-PROMELA.
- Un DEv-PROMELA peut être implémenté dans n'importe quel simulateur DEVS.

Avantages

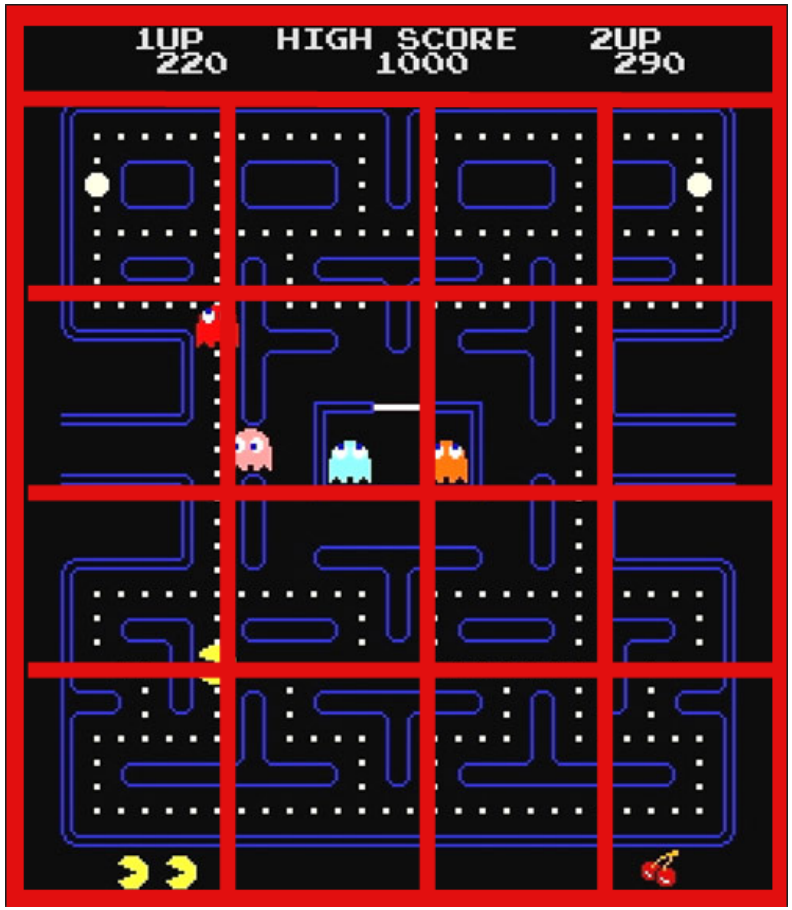
- Valider un logiciel = Exécuter l'implémentation.
- Simuler => Découvrir de nouvelles hypothèses sur le système.
- Simuler un modèle conceptuel de logiciel => "Pré-valider" les spécifications.
- Valider via un modèle à évènements discrets : gain de temps substantiel par rapport à une exécution temps réel.

Contents

Retour à Pacman ?

- Dans l'exemple du Pacman : l'interpolation est modélisable en DEv-PROMELA. On peut la vérifier par simulation.
- MC : "Existe-t-il un cas où un fantôme et un Pacman peuvent se croiser sans déclancher une collision ?" Model-checking affirme que non !
- Simulation : On fait varier la vitesse et on vérifie la même propriété. La simulation affirme que oui. Que s'est-il passée ?

L'abstraction sur des entiers...



Alors, on fait quoi ?

- Résultat : Ici, la simulation détecte une erreur que le model-checking n'est pas capable de détecter (le déplacement dépend du temps écoulé entre deux états). Le modèle ne garantit pas la propriété de non-croisement. ;
- Pour les autres propriétés qui ne dépendant pas du temps (le nombre de vie peut-il être négatif ? Le score est-il toujours positif ? Un pacman peut-il sortir du labyrinthe..), le model-checking est suffisant.

Contents

Conclusion

- Le DEv-PROMELA est une extension de PROMELA pour la modélisation, vérification et validation de modèles à évènements discrets qui combinent VF et simulation. Une seule modélisation pour la VF et pour la Simulation.
- VF = vérification de propriétés indépendantes du temps, simulation complète la VF pour les propriétés dépendantes du temps.
- DEv-PROMELA est un langage syntaxique, rendant plus aisé l'implémentation.
- Mais, DEv-PROMELA est limité au Classic DEVS.
- La simulation dépend toujours des scénarii joués. Identifier les bons scénarii reste un challenge.

Perspectives

- Etendre la classe des systèmes modélisables avec DEv-PROMELA (Parallel DEVS, GDEVS...)
- Etablir des règles pour identifier le cadre expérimental.
- Améliorer le MC en tenant en compte que des chemins définis réellement par le modèle DEv-PROMELA (et pas l'ensemble des chemins exprimés par le DEVS symbolique).

Merci !

Merci pour votre attention. Des questions ?