

INF6603

Vérification des systèmes temps réel

3 Crédits

$3 / 1,5 / 4,5$

<https://moodle.polymtl.ca/>

Hanifa Boucheneb

Local M-4107

(514) 340-4711 –4101

hanifa.boucheneb@polymtl.ca

INF6603

Vérification des systèmes temps réel

Cours théorique :

Vendredi 12h45 – 15h35 A-404

Début des cours : 31 août 2018

Laboratoire :

Lundi 12h45 – 15h35 (B1) L-4712

Début des laboratoires : 17 septembre 2017

Objectifs

- Présenter les **principales approches de vérification formelle** en insistant sur leurs caractéristiques, avantages et leurs limitations.
- Étudier deux approches de vérification : « **model-checking** » et **synthèse de contrôleurs**.
- Étudier différentes variantes de model-checking : **model-checking temporel**, **model-checking symbolique**, **model-checking borné**, **software model-checking**.
- Montrer comment utiliser le « **model checking** » et la **synthèse de contrôleurs** pour notamment analyser l'ordonnancement de systèmes temps réel et générer automatiquement **des cas de tests**.
- Appliquer ces approches de vérification en utilisant des outils de model-checking et de synthèse de contrôleur (objectifs du laboratoire).

Plan du cours

| | |
|-------------------|---|
| Séance 1 (31 / 8) | Plan de cours et généralités |
| Séance 2 | Modélisation de systèmes temps réel (systèmes de transitions, automates, automates temporisés, interface-UPPAAL, autres extensions des automates). |
| Séance 3 | Spécification de propriétés attendues. Logiques temporelles (CTL*,LTL,CTL). logiques temporelles temporisées (TLTL,MITL,TCTL). |
| Séance 4 | LTL Model-checking explicite de systèmes de transitions finis. |
| Séance 5 | Quiz 1 : Automates temporisés & Logiques temporelles. |
| Séance 6 | CTL Model-checking explicite de systèmes de transitions finis. |
| Séance 7 | Model-checking des automates temporisés par abstraction (graphes des régions et graphes des zones).UPPAAL (model-checker). |
| Séance 8 | Synthèse de contrôleur (automates de jeu, calcul de stratégies gagnantes, automates temporisés de jeu). UPPAAL –TIGA. Analyse de l'ordonnancabilité par model-checking/synthèse de contrôleur. |
| Séances 9 | Quiz 2 : LTL/CTL model-checking. Graphes de régions, Graphes de zones et synthèse de contrôleur. |
| Séance 10 | Software model-checking (programmes booléens). |
| Séance 11 | Model-checking symbolique basé sur les diagrammes binaires de décision (BDDs). Model-checking symbolique borné basé sur la satisfiabilité (SAT, SMT). |
| Séance 12 | Génération automatique de cas de test par model-checking. UPPAAL-COVER. |
| Séances 13 et 14 | Présentations orales |

Laboratoire

- Un travail dirigé et trois travaux pratiques sont au programme. Le but du laboratoire est de vous familiariser avec certains outils de vérification formelle (UPPAAL , UPPAAL TIGA, LLBMC).
 - Le premier travail pratique vise à modéliser un système, spécifier ses requis et utiliser un outil de model-checking pour vérifier si le modèle satisfait ses requis.
 - Le second travail pratique a pour but de montrer comment utiliser la synthèse de contrôleur pour synthétiser un modèle qui satisfait des requis.
 - Le troisième travail pratique montre comment vérifier un programme C par model-checking (software model-checking).

Documentation

Voir la section « Quelques références » du site du cours.

- E. M. Clarke, « The Birth of Model Checking » (2006).
- C Baier, J-P Katoen, « Principles of Model Checking » (2008).
- E. M. Clarke, B. H. Schlingloff, « Model Checking » (2000).
- F. CASSEZ, N. MARKEY « Contrôle des systèmes temporisés » (2008).

Évaluation

| <i>Nature</i> | <i>Pondération</i> | <i>Dates</i> | <i>Remise</i> |
|----------------------|--------------------|--------------------------|-----------------------|
| Travail dirigé | 0% | 17 septembre | Pas de remise |
| Travail pratique 1 | 25 % | 2 octobre – 22 octobre | 29 octobre avant 18h |
| Travail pratique 2 | 15 % | 5 novembre | 12 novembre avant 18h |
| Travail pratique 3 | 15 % | 19 novembre | 26 novembre avant 18h |
| Quiz 1 | 20 % | 28 septembre | |
| Quiz 2 | 20 % | 2 novembre | |
| Présentations orales | 15% | 30 novembre - 3 décembre | |

UPPAAL - Microsoft Internet Explorer

Fichier Edition Affichage Favoris Outils ?
Précédente → Rechercher Favoris Média |
Adresse http://www.uppaal.com/ OK
Liens
Google UPPAAL Search New! 4350 blocked ABC Check AutoLink AutoFill Options UPPAAL
RELATED SITES: TIMES | UPPAAL CORA | UPPAAL TRON

UPPAAL

Home

Home | About | Documentation | Download | Examples | Bugs

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).

The tool is developed in collaboration between the [Department of Information Technology](#) at Uppsala University, Sweden and the [Department of Computer Science](#) at Aalborg University in Denmark.

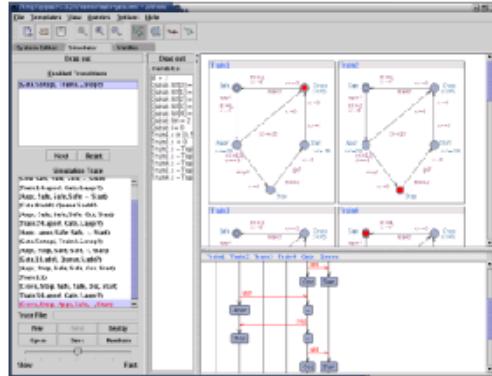


Figure 1: UPPAAL on screen.

Download

The current official release is UPPAAL 3.4.11 (Jun 23, 2005). A development snapshot of UPPAAL 3.5.9 (Aug 8, 2005) is also available. For more information about UPPAAL version 3.4, we refer to this [press release](#).


TIDDCAT A



License

The UPPAAL tool is **free** for non-profit applications but we have a [license agreement](#) that all users must fill in before downloading and using the tool. To find out more about UPPAAL, read this short [introduction](#). Further information may be found at this web site in the pages [About](#), [Documentation](#), [Download](#), and [Examples](#).

Mailing Lists

UPPAAL has an open [discussion group](#) at Yahoo! Groups intended for users of the tool. To join the group, email uppaal-subscribe@yahoo-groups.com, to post use uppaal@yahoo-groups.com. To email the development team directly, please use bug-uppaal@list.it.uu.se for bug reports and uppaal@list.it.uu.se otherwise.

UPPAAL - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

Les plus visités Imprimeur DGI

<http://www.uppaal.com/download/registration.php?id=5&subid=2>

UPPAAL

RELATED SITES: UPPAAL | UPPAAL CORA | UPPAAL TRON

UPPAAL TIGA

UPPAAL for Timed Games

Main Page | Download | Contact us

Welcome!

UPPAAL TIGA (Fig. 1) is an extension of UPPAAL [BDL04] and it implements the first efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and safety properties. Though timed games for long have been known to be decidable there has until now been a lack of efficient and truly on-the-fly algorithms for their analysis.

The algorithm we propose [CDFLL05] is a symbolic extension of the on-the-fly algorithm suggested by Liu & Smolka [LS98] for linear-time model-checking of finite-state systems. Being on-the-fly, the symbolic algorithm may terminate long before having explored the entire state-space. Also the individual steps of the algorithm are carried out efficiently by the use of so-called zones as the underlying data structure. Our tool implements various optimizations of the basic symbolic algorithm, as well as methods for obtaining time-optimal winning strategies (for reachability games).

References

- [BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. LNCS 3185. Springer-Verlag 2004, pp 200-236.
- [CDFLL05] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient On-the-fly Algorithms for the Analysis of Timed Games. LNCS 3653. Springer-Verlag 2005, pp 66-80.
- [LS98] Xinxin Liu and Scott A. Smolka. Simple Linear-Time Algorithm for Minimal Fixed Points. LNCS 1443. Springer-Verlag 1998, pp 53-66.

Figure 1: UPPAAL TIGA on screen.

Latest News

Version 0.12 released.
11 Dec 2007

Versions 0.12 is released today. This version fixes all the known issues concerning time optimal strategies. The simulator has been updated and the simulation is more robust w.r.t. precision. This version inherits a new feature from the development version of UPPAAL, namely the addition of stopwatches. You should not use this feature in a TIGA model. Another major new feature is the possibility to get C-code instead of a strategy (option -x) and choose to store strategies as federations or CDD (option -g).

Versions 0.10 and 0.11 released.
7 July 2007

Versions 0.10 and 0.11 are released today. Version 0.11 contains a new concrete simulator that allows the user to play strategies from the GUI. Both versions fix the following bugs: maximal constants in the formula are now taken into account, the command line simulator is new and works better, delay when no clock was used, better user feedback, end-of-game detection fixed, other bugs involving delays in the strategy, precision problems in the simulator, and leak in the DBM library. These new versions have also the following new features: options to control the type of strategy output, better control on the search ordering (forward and

The screenshot shows a Firefox browser window with the following details:

- Address Bar:** llbmc.org/index.html
- Toolbar:** Includes standard Firefox icons for back, forward, search, and zoom.
- Header:** Shows the title "LLBMC: The Low-Level Bou..." and a status bar indicating "Faire une capture d'écran so...".
- Page Content:** The main content area displays the LLBMC website. It features the KIT logo and navigation links like "Introduction", "News", "Downloads", etc. A yellow sticky note on the left says "New version (2013.1) of LLBMC available!". The main content includes:
 - Welcome to the Software Analysis Tool LLBMC!**
 - Overview:** Describes LLBMC as a static software analysis tool for C/C++ programs using Bounded Model Checking.
 - LLBMC can help to:**
 - reduce the time and effort needed for software testing,
 - improve the quality of software,
 - achieve high test coverage ratios,
 - obtain stable and secure software in reduced time.
 - LLBMC is fully automatic and requires minimal preparation efforts and user interaction. It supports all C constructs, including not so common features such as bitfields. LLBMC models memory accesses (heap, stack, global variables) with high precision and is thus able to find hard-to-detect memory access errors like heap or stack buffer overflows. LLBMC can also uncover errors due to uninitialized variables or other sources of non-deterministic behavior. Due to its precise analysis, LLBMC produces almost no false alarms (*false positives*).**
 - Technically, LLBMC incorporates several innovations:**
 - Instead of working on the C source code directly, it employs a compiler frontend (the LLVM compiler infrastructure) and starts verification on the RISC-assembler-like intermediate representation. **LLBMC's analysis results are thus much closer to what actually runs on the machine.**
 - LLBMC supports all C language features (currently with the exception of floating-point numbers), even esoteric ones like bitfields, and models their semantics with high precision on the bit-level.
 - By using an SMT solver (Boolector or STP) as a logical backend, it achieves **high performance**.
 - Rewrite-based simplifications can discharge many "easy" proof obligations before invocation of the core SMT solver. This again improves performance.
 - Built-in Checks:**
 - Integer overflow
 - Division by zero
 - Invalid bit shift
 - Illegal memory access (array index out of bound, illegal pointer access, etc.)
 - Invalid free
 - Double free
 - User-customizable checks (via `__llbmc_assume` / `__llbmc_assert`)
- Footer:** THIS SITE IS HOSTED BY KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT), GERMANY. BACK TO TOP.

Généralités

- **Qu'est ce que la vérification formelle ?**
- **Pourquoi vérifier formellement ?**
- **Comment vérifier formellement ?**
 - **Comment spécifier un comportement ?**
 - **Comment spécifier une propriété ?**
 - **La vérification effective**

Qu'est ce que la vérification formelle ?

Vérification et spécification

- Selon Sommerville (2004, Software engineering), la **vérification** est la réponse à la question :

“Are we building the product **right** ?”

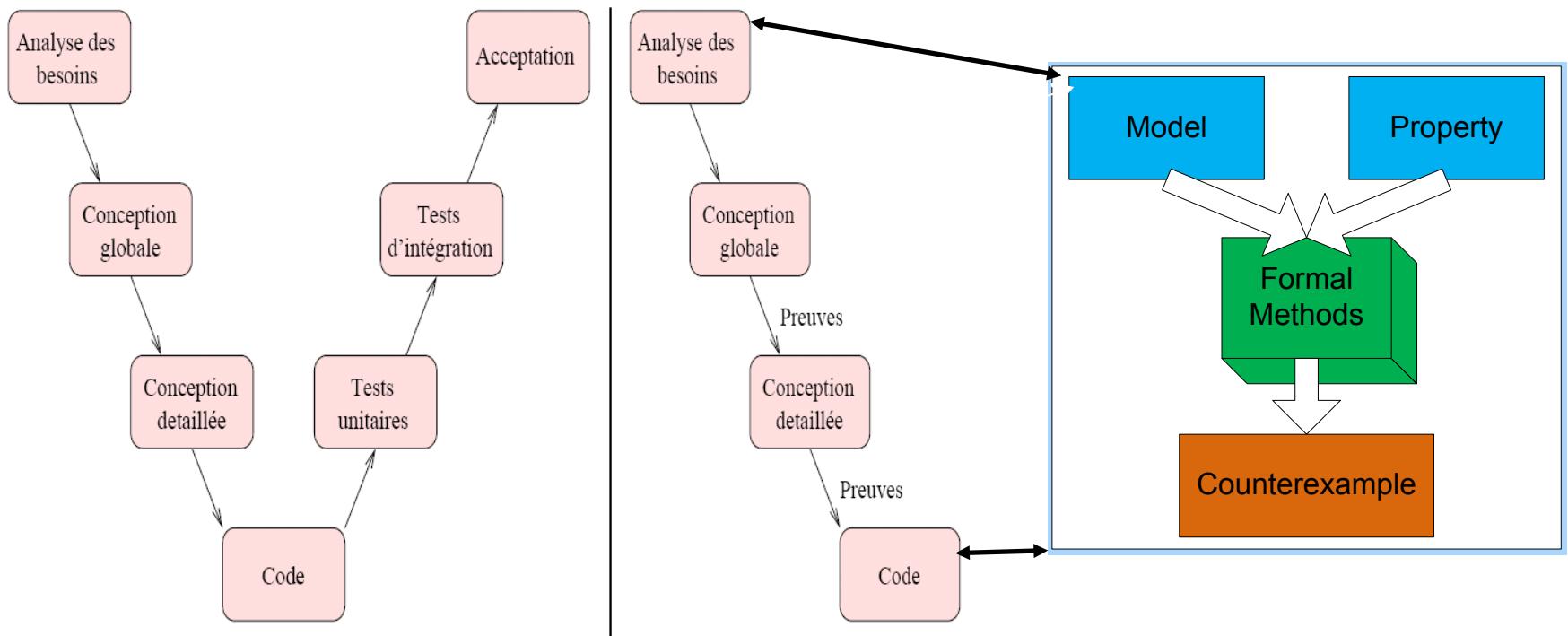
right → une notion de référence (ce qui doit être correct).
building the product → un processus de conception
Are we → une méthode pour s'assurer que la réalisation est correcte.
- **Les propriétés attendues** du produit servent de référence pour la vérification.
- Trois principales techniques : test, simulation et vérification formelle.
- Le test de logiciel et la simulation de matériel sont des méthodes de vérification qui valident le système pour un ensemble de valeurs, d'entrées ou d'évolutions.

Vérification formelle

- **Est-ce que la réalisation d'un système satisfait les propriétés quelles que soient les données en entrée ou l'évolution de son environnement ?**
- **Vérification formelle complète** : exploration de façon exhaustive de tous les états possibles du système (model-checking) et démonstrateurs de théorèmes.
- **Vérification formelle incomplète** : exploration bornée (des séquences d'états de longueur bornée).

Pourquoi vérifier formellement ?

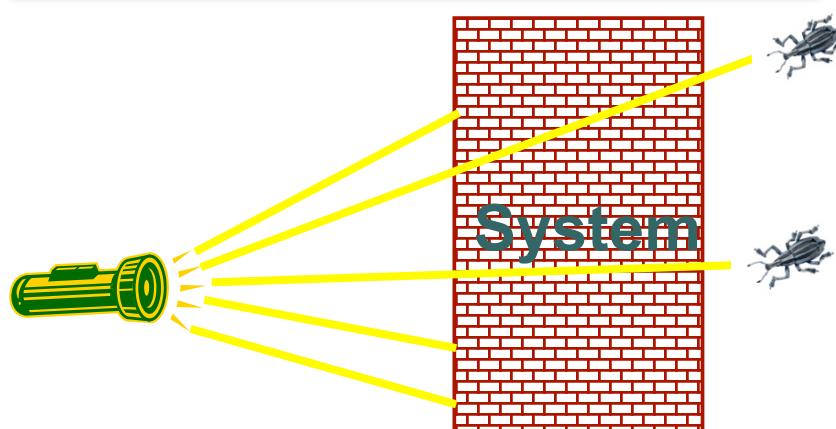
Renforcer le processus de développement



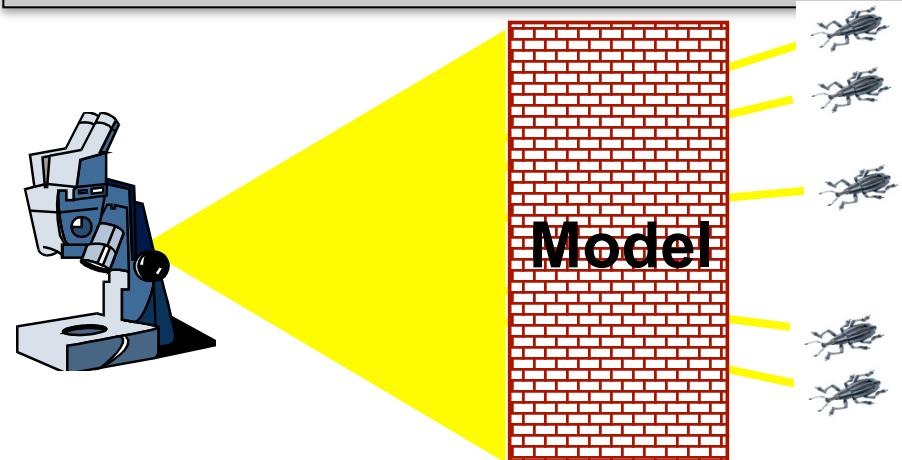
→ Développement dirigé par les modèles (*Model-Driven Development MDD*)

Renforcer le processus de développement

Test vérifie les évolutions jugées critiques



Vérification formelle vérifie toutes les évolutions



- peut détecter des erreurs (bugs),
- *ne peut pas prouver l'absence d'erreurs,*
- *peut s'appliquer à de gros systèmes.*

- peut détecter des erreurs,
- peut prouver l'absence d'erreurs,
- est très gourmande en mémoire (*problème d'explosion combinatoire*).

Quelques réalités

- Plusieurs accidents liés à des erreurs logiciels (<http://catless.ncl.ac.uk/risks>)
 - **AECL(Canada)/CGR MeV(Fr) (1985-1987)** : THERAC-25 → des doses de radiations jusqu'à 100 fois supérieures à la normale.
 - Plusieurs morts et blessés graves.
 - **AT&T (1990)** : pannes en cascade des relais téléphoniques d'AT&T

```
switch (...) {  
    case 1: ....  
        break;    ← instruction manquante  
    case 2: reboot();  
        break;  
}
```

 - 7 000 000 appels perdus et une réputation à refaire.



Quelques réalités

- **Arianespace (1996)** : Vol 501 de la fusée Ariane V → destruction du lanceur. → Pertes financières et de clients au profit de la concurrence
- **NASA (1997)** : Sojourner était incapable de transmettre ses résultats d'analyse vers la terre → une inversion de priorité (blocage d'une tâche périodique de haute priorité).
- Erreurs inacceptables pour les systèmes critiques :
 - Transport (avionique, ferroviaire, spatial,...)
 - Énergie (production nucléaire, extraction pétrolière,)
 - Médical (traitements automatisés, supervision,...)
- ...
- Les normes et les certifications imposent d'apporter des preuves de fiabilité (DO-178B/ED-12B, DO-178C/12C pour l'avionique et les applications militaires, [http://www.rtca.org/downloads>List%20of%20Available%20Docs%20-%20Jun%202012.pdf](http://www.rtca.org/downloads/List%20of%20Available%20Docs%20-%20Jun%202012.pdf)).

Quelques réalités

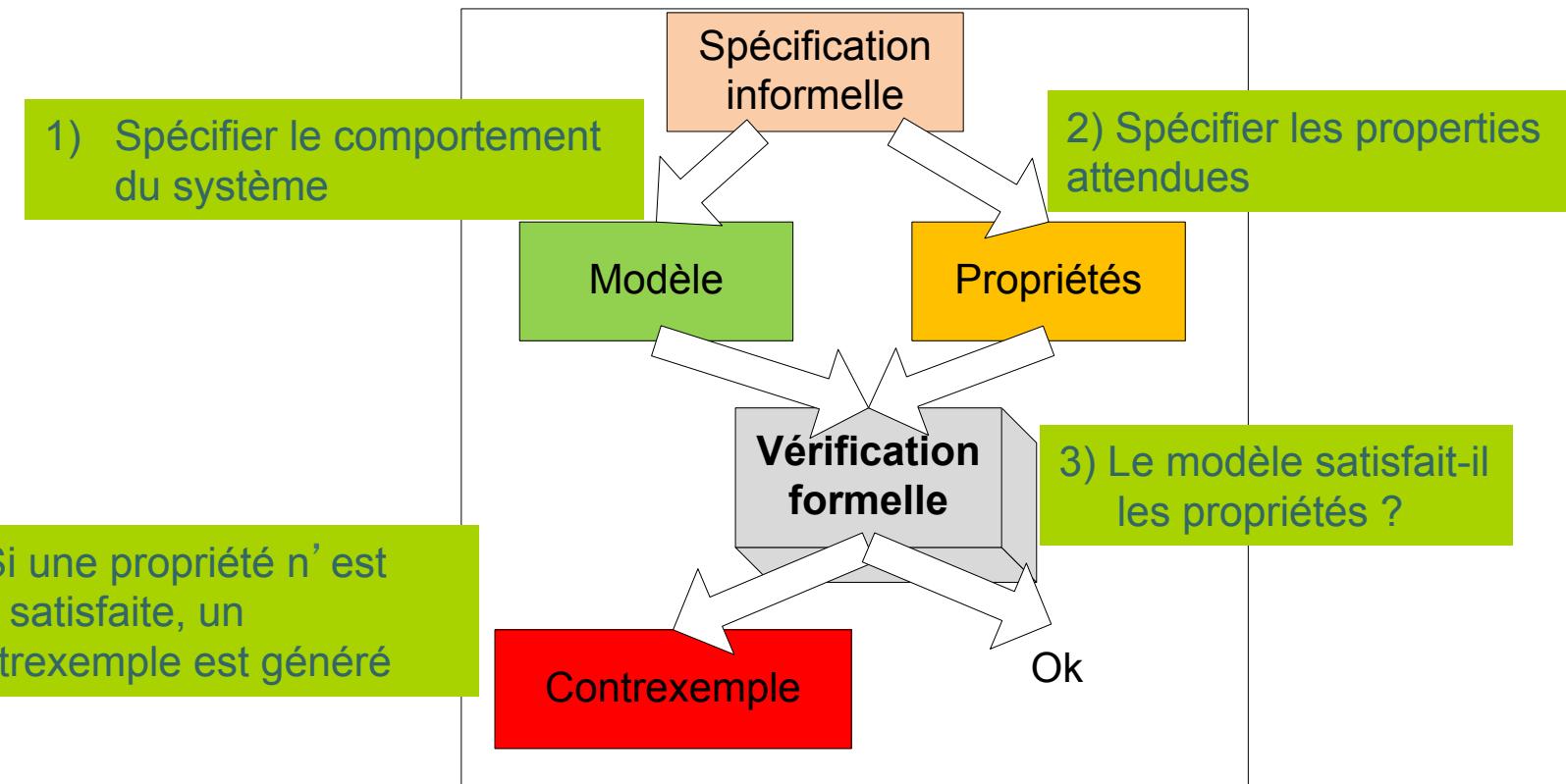
- DO-178B/ED-12B est un document produit par la RTCA (DO-178B, USA) et EUROCAE (ED-12B, Europe) qui fournit des lignes directrices précisant la façon de produire du code pour les applications aéronautiques (1ère version en 1992).
- Les normes **ED-12C** et **DO-178C (2012)** précisent notamment les contraintes de développement liées à l'obtention de la certification d'un logiciel d'avionique (exigent l'utilisation de techniques de vérification formelle).
- Ces normes définissent 5 niveaux de criticité pour un système :
 1. Niveau A : Un défaut peut provoquer un problème catastrophique (ex. crash de l'avion).
 2. Niveau B : Un défaut peut provoquer un problème majeur entraînant des dégâts sérieux.
 3. Niveau C : Un défaut peut provoquer un problème sérieux entraînant un dysfonctionnement des équipements vitaux de l'appareil.
 4. Niveau D : Un défaut peut perturber la sécurité du vol.
 5. Niveau E : Un défaut est sans effet sur la sécurité du vol.
- Objectifs de vérification à atteindre dépendent du niveau de criticité.

Comment vérifier formellement ?

Comment vérifier formellement ?

- Pour vérifier formellement, il faut spécifier (modéliser) le système que l'on veut vérifier ainsi que les propriétés qu'il doit satisfaire.
- La vérification comporte, en général, trois étapes :
 1. Modélisation du système : vise à décrire de façon claire et non ambiguë un système. Elle aboutit à un modèle ou un programme.
 2. Spécification des propriétés attendues du système.
 3. Preuve que le système modélisé possède bien les propriétés attendues (la vérification effective).

Comment vérifier formellement ?

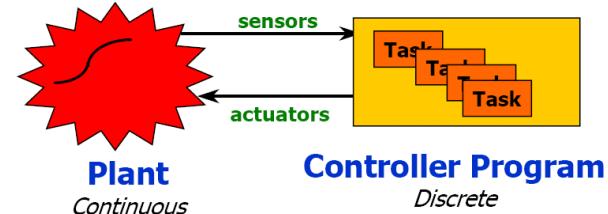


Comment spécifier un comportement ?

Modèles

- Le but de la modélisation est d' exprimer, au moyen d' un formalisme la manière dont le système se comporte et réagit face à son environnement → un modèle.
- Le formalisme utilisé doit :
 - être assez expressif,
 - reposer sur une sémantique rigoureuse permettant, à la fois, de décrire clairement le système et de construire tous les comportements possibles du système,
 - offrir des possibilités d' analyse systématique (le but final de la modélisation est, dans ce cas, la vérification formelle).
- Un modèle est une représentation simplifiée d'un système (ou d'un aspect du système).

Modèles



- Un système temps réel :
 - le fonctionnement est assujetti à l' évolution dynamique de l' environnement et
 - la réaction aux stimuli (temps de réponse) est soumise à des contraintes temporelles imposées par l' environnement
- Le modèle utilisé doit aussi permettre d' exprimer les contraintes temporelles.

Systèmes critiques : le non respect de certaines contraintes temporelles peut avoir des conséquences graves non acceptables.

Modèles

- Plusieurs modèles formels ont été développés puis adaptés aux systèmes temps réel (intégration de contraintes temporelles).
- Ils se distinguent par la manière et l'aptitude à exprimer :
 - la séquentialité, la concurrence,
 - le non-déterminisme, la synchronisation,
 - la communication, la compositionnalité,
 - les contraintes temporelles,
- Les classes de modèles proposés dans la littérature sont :
 - les langages de programmation et de spécification,
 - les modèles à base de transitions,
 - les modèles logiques,
 - les modèles algébriques,

Modèles ([Z. Mammeri](#))

- D'une manière générale, les modèles permettent de spécifier les :
 - **actions** (déplacement de l'ascenseur vers le haut/bas, fermeture de vanne, Ecriture dans une base de données, etc.),
 - **événements** qui déclenchent les actions (dépassement de température, réception d'un message, ouverture de porte d'ascenseur, etc.),
 - **contraintes temporelles** à respecter (trouver un vol pour une destination donnée en moins de 2 minutes, envoi de message toutes les 100 secondes, etc.),
 - **conditions d'activation** à respecter avant de déclencher des actions (si le nombre de pièces est égale à 10, fermer la boîte, s'il n'y a pas d'obstacle devant la cellule de l'ascenseur, fermer la porte de l'ascenseur, etc.),
 - **situations anormales** et leurs traitements (arrêt de l'ascenseur entre deux étages et appui de touche sans résultat, etc.),
 - **états significatifs** du système (ascenseur à l'arrêt, ascenseur en mouvement, porte ouverte et ascenseur en maintenance sont des exemples d'états significatifs d'un système Ascenseur).

Langages de programmation / spécification

Langages de programmation / spécification

- **On distingue deux classes de langages :**
 - Ceux qui complètent les langages classiques de programmation au moyen de primitives répondant aux besoins des systèmes à modéliser (la synchronisation, la communication, le parallélisme, les contraintes temporelles).
- Exemples :** les langages ADA, OCCAM, JAVA, C, C++, C#.
- Ceux qui se basent sur un modèle à transitions d' états, modèle algébrique, modèle de flot de données ou modèle logique.
- Exemples :** Les langages RT-LOTOS (types abstraits, algèbre de processus temporisés), PROMELA (automate étendu + C), ESTEREL (Statecharts) et LUSTRE (réseaux d' équations).

**Exemple 1 : Langage de programmation Java
(software model-checker Java PathFinder
de La NASA, <http://babelfish.arc.nasa.gov/trac/jpf>)**

Langage de programmation Java (Java PathFinder)

Code Java (Java PathFinder de La NASA) <http://babelfish.arc.nasa.gov/trac/jpf>

```
import java.util.Random;

public class Rand {
    public static void main (String[] args) {
        Random random = new Random(42);          // (1)
        int a = random.nextInt(2);                // (2)
        System.out.println("a=" + a);

        //... lots of code here

        int b = random.nextInt(3);                // (3)
        System.out.println("  b=" + b);

        int c = a/(b+a -2);                     // (4)
        System.out.println("    c=" + c);
    }
}
```

Une exécution possible du code

```
> java Rand
a=1
b=0
c=-1
>
```

Langage de programmation Java (Java PathFinder)

Code Java (Java PathFinder de La NASA)

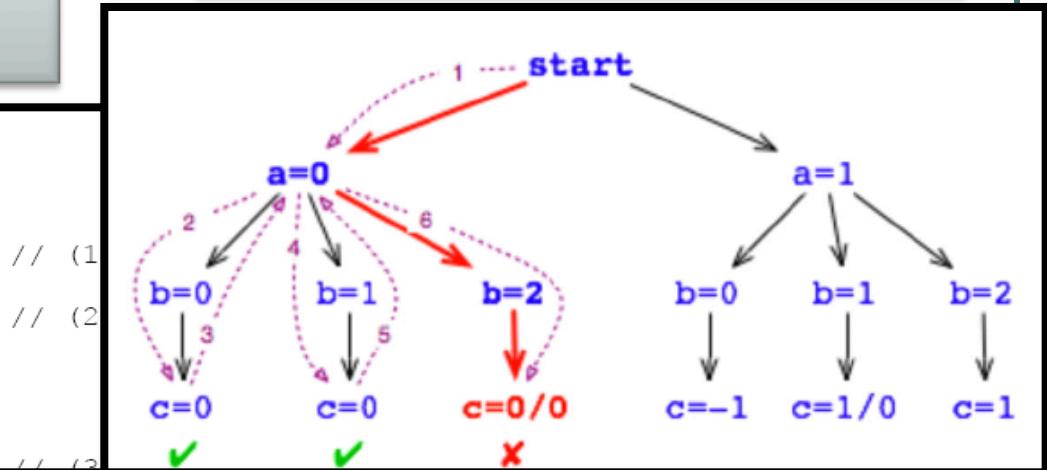
<http://babelfish.arc.nasa.gov/trac/jpf>

```
import java.util.Random;

public class Rand {
    public static void main (String[] args) {
        Random random = new Random(42);
        // (1)
        int a = random.nextInt(2);
        System.out.println("a=" + a);
        //... lots of code here
        // (2)
        int b = random.nextInt(3);
        System.out.println("  b=" + b);
        // (3)
        int c = a/(b+a -2);
        System.out.println("      c=" + c);
    }
}
```

```
> java Rand
a=1
b=0
c=-1
>
```

Software model-checker Java PathFinder



```
> bin/jpf +vm.enumerate_random=true Rand
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
=====
application: /Users/pcmehlitz/tmp/Rand.java
=====
search started: 5/23/07 11:49
=====
a=0
  b=0
    c=0
  b=1
    c=0
  b=2
===== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmetricException: division by zero
      at Rand.main(Rand.java:15)
...
>
```

Exemple 2 : Langage de programmation C(C++)
(LLBMC Low Level Bounded Model-Checker,
<http://llbmc.org/>)

Langage de programmation C (C++) (LLBMC)

Code C

(LLBMC : Detecting Memory Access Violations)

<http://llbmc.org/>

```
1 int a[3];
2 void foo(int m)
3 {
4     int i;
5     for (i = 0; i < m; i++)
6         a[i] = i; // memory access violation, if m>3
7 }
```

```
// générer en utilisant « clang » le « bytecode » (un code intermédiaire entre les instructions machines et le code source) dans ex1.bc  
clang -c -g -emit-llvm ex1.c -o ex1.bc
```

```
// Lancer le model-checker LLBMC (borne=4)
llbmc ex1.bc --max-loop-iterations=4
```

A bounded model-checker based on Satisfiability (SMT)

... Assertion failed: Memory write access

— — — — —

Error context:

2: void foo(int m)

3: {

4: int i;

5: for (*i* = 0; *i* < *m*; *i*++)

6: a[i] = i; //memory access violation, if m>3

3.1

7. }

— — — — —

#0 void @foo(i32 %m=6)

**Exemple 3 : Langage de spécification Promela
(Model-Checker SPIN,
<http://spinroot.com/spin/whatispin.html>)**

Langage de spécification Promela

```
#define true      1
#define false     0
#define Aturn      0
#define Bturn      1
bool x, y, t;
proctype A()
{
    x = true;
    t = Bturn;
    (y == false || t == Aturn);
    /* critical section */
    x = false;
}
proctype B()
{
    y = true;
    t = Aturn;
    (x == false || t == Bturn);
    /* critical section */
    y = false;
}
Init { run A(); run B() }
```

Code Promela (algorithme d'exclusion mutuelle de Peterson)

<http://spinroot.com/spin/whatispin.html>

- Promela (Protocol/Process Meta Language) est utilisé par le model-checker SPIN (Simple Promela Interpreter développé par les laboratoires Bell).
- Il permet la création dynamique de processus concurrents, la communication entre processus via des variables partagées et des canaux de messages (par rendez-vous ou asynchrone).
- La spécification est d'abord transformée en un ensemble d'automates qui sont ensuite vérifiés par **model-checking**.

Exemple 4 : Langage synchrone LUSTRE (boîte à outils SCADE,

<http://www.estrel-technologies.com/products/scade-suite/>)

Langage synchrone LUSTRE (SCADE)

- LUSTRE est un langage synchrone à flots de données, utilisé par la boîte à outils SCADE (Safety-Critical Application Development Environment).
- LUSTRE permet d'exprimer un système sous forme d'équations qui définissent l'évolution des valeurs de ses variables (après chaque cycle d'exécution).
- Une variable x de LUSTRE représente un flot c-à-d une séquence infinie $(x_0, x_1, \dots, x_n, \dots)$ de valeurs, où x_0 est la valeur initiale et x_i est la valeur de x au $i^{\text{ème}}$ cycle d'exécution.
- Une équation est un invariant temporel, par exemple $x = y + z$ définit le flot $(x_0, x_1, \dots) = (y_0 + z_0, y_1 + z_1, \dots)$.
- Expression Flot de données
pre x $(-, x_0, x_1, \dots)$.
 $x \rightarrow y$ (x_0, y_1, y_2, \dots) .
 $0 \rightarrow \text{pre } x + 1$ $(0, 1, 2, \dots)$

Langage synchrone LUSTRE (outil SCADE)

- Programme LUSTRE = réseau d' équations (nœuds) produisant et consommant des flots de données à chaque cycle d 'exécution.

```
node MinMax(X : int) returns (min, max : int);  
  
let  
  
min = X -> if (X < pre min) then X else pre min;  
  
max = X -> if (X > pre max) then X else pre max;  
  
tel
```

<http://www-verimag.imag.fr/~raymond/edu/eng/lustre-a.pdf>

Ce nœud donne le min et le max de la séquence X_0, X_1, \dots

Langage synchrone LUSTRE (SCADE)

- SCADE est un environnement de développement basé sur les modèles (MBD) pour les systèmes temps réel critiques. Il est commercialisé par la société Esterel Technologies et très utilisé dans le domaine de l'embarqué (avionique, nucléaire, automobile).



Langage synchrone LUSTRE (SCADE)

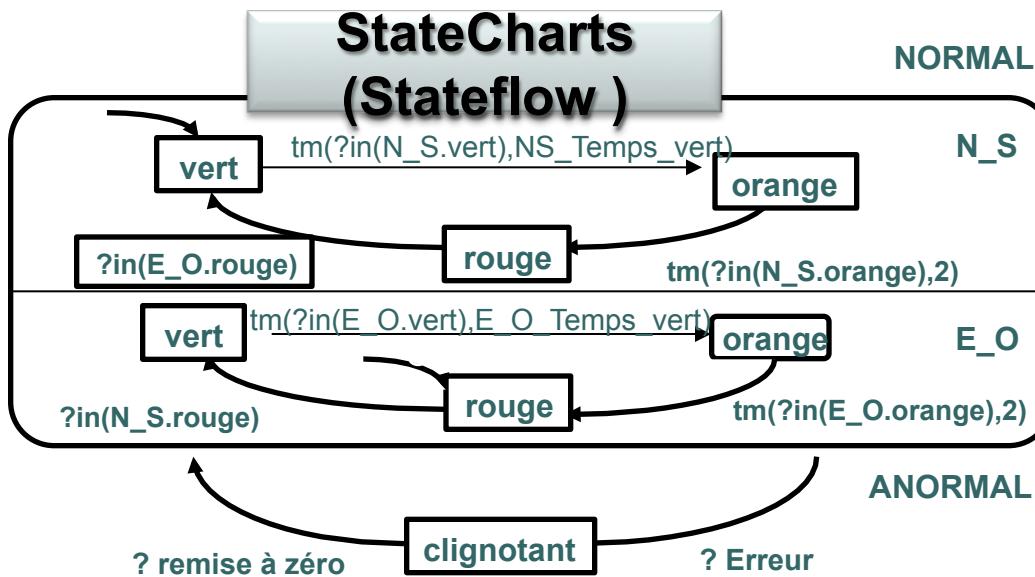
- SCADE permet la modélisation, la simulation, la génération de cas de test (selon différents critères de couverture), **la vérification formelle (Design-Verifier)**, la génération automatique de codes, etc.
- **La vérification formelle vient en complément des tests. Elle utilise l'outil « SCADE Suite Design-Verifier ».**
- **Design-Verifier** est basé sur l'outil **Prover Plug-In** (commercialisé par Prover Technology) qui est aussi intégré à l'environnement de développement de SIMULINK.
- Il permet **la vérification par model-checking, la preuve d'équivalence, et la génération de cas test.**

Modèles à base de transitions

Modèles à base de transitions

- Dans le cas de ces modèles, un système est décrit en spécifiant ses états, son état initial et les transitions possibles entre états.
- Ces modèles sont, en général, graphiques :
 - Les « Statecharts » (diagrammes d'états de [Harel](#)).
 - Les réseaux de Petri (colorés, temporisés, stochastiques (C.A. [Petri](#)) ;
 - Les **automates (avec variables discrètes/continues/aléatoires)**.
- Les automates offrent un bon compromis entre la puissance de modélisation et la complexité de vérification.

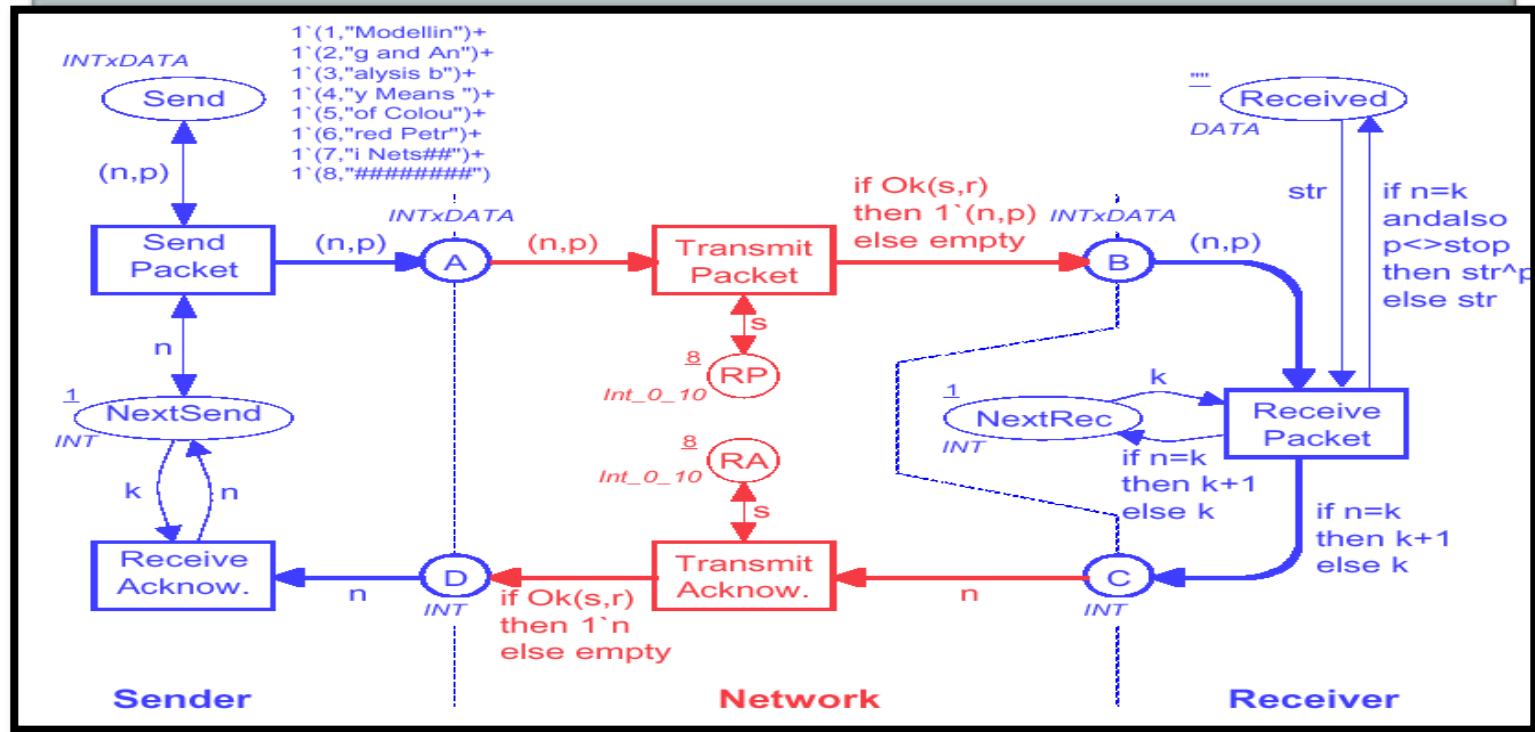
StateCharts



- StateCharts sont utilisés dans **SCADE** et **SIMULINK**, Ils sont proches des machines à états d'UML.
 - Ils permettent des compositions parallèles, séquentielles et hiérarchiques des états.
- Outil **Design-Verifier pour la vérification formelle.**

Les réseaux de Petri

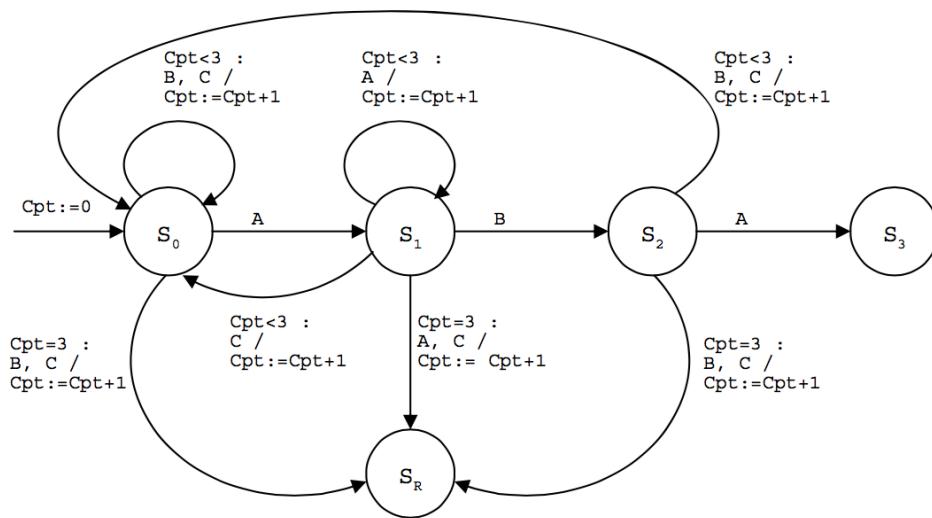
Réseaux de Petri colorés
(CPNTOOLS utilise des techniques de model-checking)



Automates avec variables discrètes ([z. Mammeri](#))

(Sommets, S0, Actions, Relation_de_transitions, Ens_Variables, Ens_Initialisations)

Digicode



- Porte déverrouillée dès que l'utilisateur a tapé la combinaison ABA (code mémorisé par le système pour déverrouiller la porte).
- Si la touche tapée ne correspond pas à la touche attendue par le système, le nombre d'erreurs (Cpt) et incrémenté de 1 et l'utilisateur doit recommencer le code depuis le début.
- Au bout de 4 erreurs, le système passe dans un état d'erreur (par exemple, il peut déclencher une alarme dans cet état).
- Par exemple, si l'utilisateur tape ABA, AAABA, BABA, BBABA, CAABA la porte est déverrouillée.

Modèles logiques

Modèles logiques

Table de décisions (RSML, SCR)

2.3 Flight Director (FD)

The Flight Director (FD) displays the pitch and roll guidance commands to the pilot and copilot on the Primary Flight Display. This component defines when the Flight Director guidance cues are turned on and off.

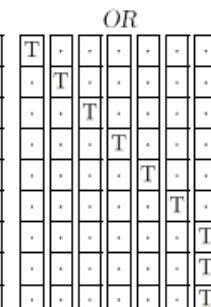
Definitions of Values to be Imported

[MACRO]

When_Turn_FD_On

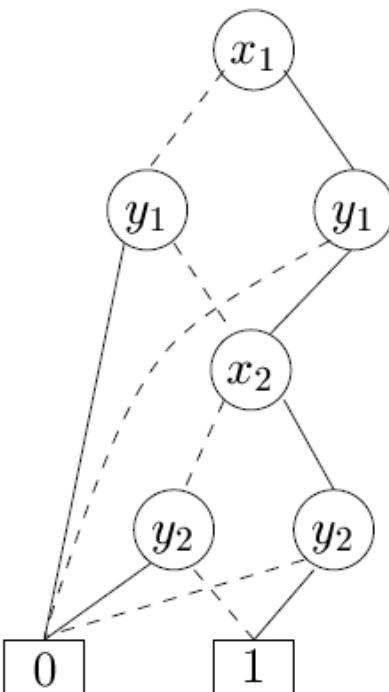
Condition:

| |
|--|
| When_FD_Switch_Pressed _{m-96()} |
| When(AP _{v-129} =Engaged) |
| When(Overspeed _{v-118}) |
| A When_GA_Switch_Pressed _{m-102()} |
| N When_Lateral_Mode_Manually_Selected _{m-23()} |
| D When_Vertical_Mode_Manually_Selected _{m-24()} |
| When_Pilot_Flying_Transfer _{m-26()} |
| Pilot_Flying _{v-26} =THIS_SIDE _{LEFT} |
| Were_Modes_On _{m-31()} |



Purpose: This event defines when the onside FD is to be turned on (i.e., displayed on the PFD).

Diagramme de décision



A BDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ with ordering $x_1 < y_1 < x_2 < y_2$.

Modèles algébriques

Modèles algébriques

- Le comportement d'un système est décrit au moyen d'une expression combinant des opérateurs algébriques et des actions. Cette expression est appelée « expression de comportement ».
- Les opérateurs classiques sont :
 - préfixage « ; » : $a;C$
 - parallèle « || » : $C1||C2$
 - choix (non déterminisme) « [] » ou « + » : $C1[]C2$
- L'algèbre des processus CCS (Calculus for Communicating Systems) et ATP (Algebra of Timed Processes) sont dans cette catégorie.
- Exemples d'expressions de comportement :
 - $C : (?ack[] (!m ; (perte[] ?ack ; next)) ; C$
 - $C : (?ack[] (!m [0,1]; (perte [5,5] [] ?ack[0,5] ; next[1,2])) ; C$

Comment spécifier une propriété ?

Comment spécifier une propriété ?

Assertions (assume-assert), invariants, observateurs, logiques modales, automates de Büchi ...

Logiques :

- Logique propositionnelle :

$\Phi \rightarrow$ proposition atomique | Φ and Φ | not Φ | (Φ)

proposition atomique : constante (true or false) ou variable booléenne.

| P | Q | $\neg P$ | $P \wedge Q$ | $P \vee Q$ | $P \Rightarrow Q$ | $P \Leftrightarrow Q$ |
|---|---|----------|--------------|------------|-------------------|-----------------------|
| T | T | F | T | T | T | T |
| T | F | F | F | T | F | F |
| F | T | T | F | T | T | F |
| F | F | T | F | F | T | T |

Exemple : not Error => not deadlock

Comment spécifier une propriété ?

- Logique des prédictats (premier ordre) étend la logique propositionnelle par des quantificateurs universels et des variables non forcément booléennes :
$$\Phi \rightarrow \text{predicat} \mid \Phi \text{ and } \Phi \mid \text{not } \Phi \mid (\Phi) \mid \forall x \in D, \Phi \mid \exists x \in D, \Phi$$

predicat est une fonction vers l'ensemble booléen.
D est un domaine de valeurs.

$$\forall x \in D, \text{ not deadlock}(x) \Rightarrow \text{ not Error }(x)$$
- En 1951, Arthur Prior a étendu la logique propositionnelle pour pouvoir spécifier l'évolution des propriétés d'un système.
- En 1977, Amir Pnueli a utilisé cette logique temporelle pour vérifier formellement des systèmes (prix Turing, 1996).
- La logique temporelle s'est beaucoup développée depuis 1977 : logique temporelle linéaire, logique temporelle arborescente (temporisée)....

Logique temporelle linéaire (LTL)

(Arthur Prior 1951, Amir Pnueli 1977)

- LTL permet d'exprimer comment les propriétés doivent se succéder le long de l'évolution d'un modèle.

Invariant : $G p$



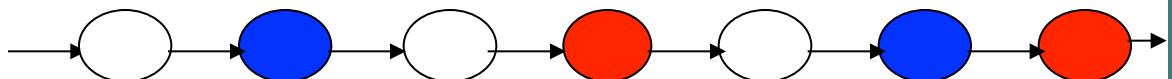
$G \text{ not deadlock}$

$G \text{ not error}$

$G (\text{not error} \Rightarrow \text{not deadlock})$

deadlock (resp. error) est une proposition vraie si l'état courant est un deadlock (resp. état erreur).

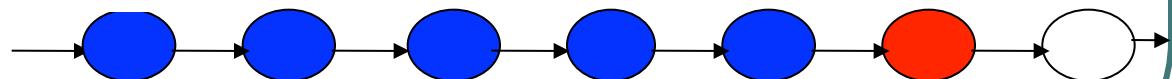
Vivacité : $G (p \Rightarrow F q)$



$G (\text{request} \Rightarrow \text{F response})$

$G (\text{error} \Rightarrow \text{F alarm})$

$(p \ U q)$



$(\text{not alarm}) \ U \text{ error}$

Logique temporelle linéaire (LTL)

- Cependant, cette logique ne permet pas de quantifier les chemins d'exécution.
- Elle opère sur les séquences d'états et ne permet pas d'exprimer, par exemple ce qui suit :
Il existe un chemin d'exécution tel que tous les états du chemin satisfont une propriété donnée p.
- C'est [Lamport](#) qui a soulevé ce problème en 1980, et l'année suivante, Clarke et [Emerson](#) ont défini une nouvelle logique : la logique temporelle arborescente (prix Turing 2007).

Logique temporelle arborescente (CTL)

([Lamport](#) 1980, [Emerson](#) 1981) :

- Cette logique permet de «quantifier» les chemins d'exécution
- **Exemples :**
 - Il existe un chemin d'exécution vérifiant la propriété suivante : la proposition `error` n'est pas vraie dans tous les états atteints le long du chemin d'exécution :
EG not error
 - Toute exécution, à partir d'un état correct, est vouée à l'échec :
AG (not error \Rightarrow AF error)
 - Le système est réinitialisable :
AG (true \Rightarrow EF init)

On suppose, `init` est vrai pour un état initial et faux pour tous les autres.

La vérification effective

La vérification effective

- La vérification est dite décidable pour un ensemble de modèles M et une classe de propriétés P , ssi il existe un programme qui :
 - prend en entrée un modèle quelconque de M et une propriété quelconque de P , et
 - détermine, au bout d'un temps fini, si la propriété est satisfaite ou non par le modèle.
- Pour pouvoir l'automatiser et garantir la terminaison, la vérification doit être décidable et aussi de complexité acceptable.
- On distingue deux grandes catégories de méthodes : les méthodes syntaxiques et les méthodes sémantiques.

Méthodes syntaxiques : Preuve de théorèmes

Méthodes syntaxiques

- Ce sont des preuves au sens mathématique du terme.
- Elles cherchent à déterminer si une propriété peut être obtenue à partir d' hypothèses et de règles de déduction propres à la logique utilisée.
- Elles sont difficiles à automatiser.
- Les démonstrateurs les plus populaires sont [NQTHM](#) (Boyer & Moore) et ses successeurs (par exemple [PVS](#)).

Méthodes syntaxiques

Considérons l'hypothèse (axiome) et les règles d'inférence suivantes :

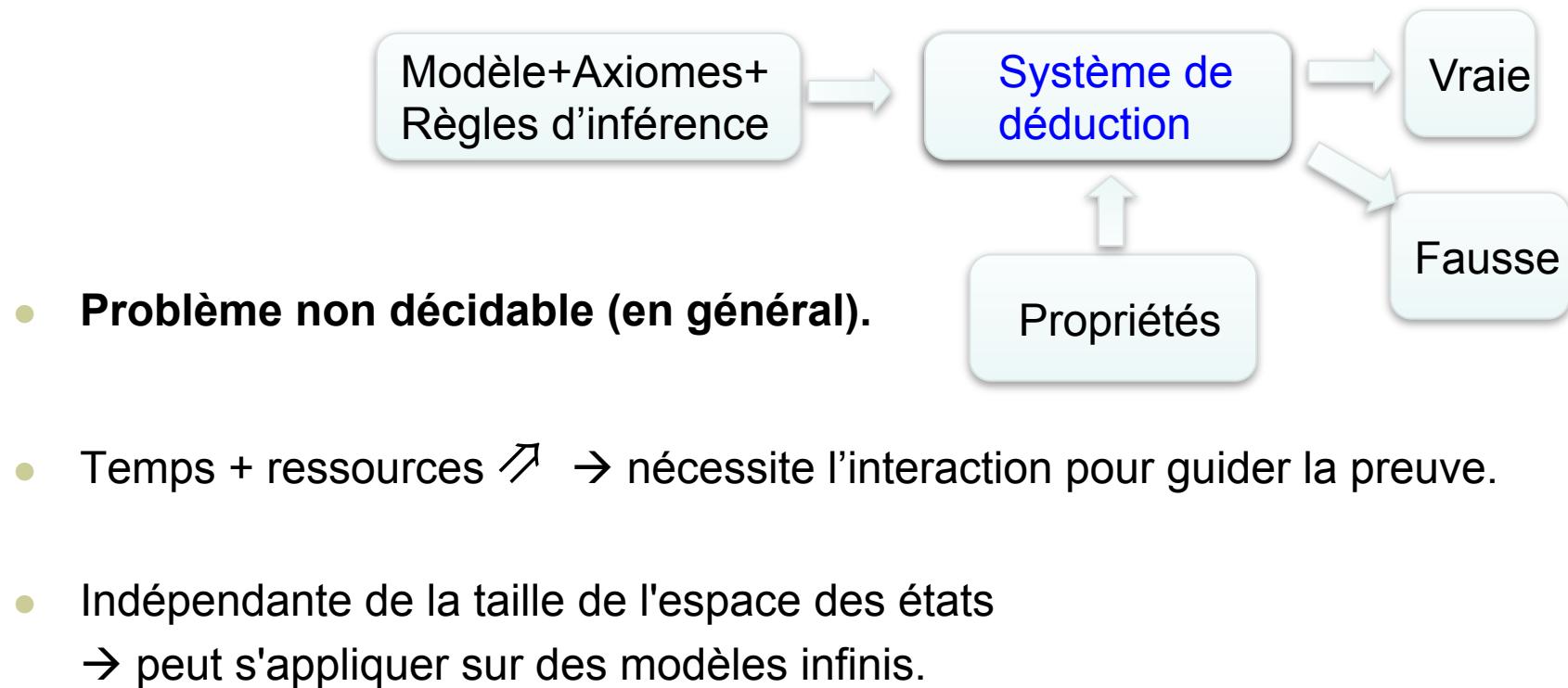
- A : “Il ne fait pas beau et il fait froid.”
- R1 : “Si on va nager alors il fait beau.”
- R2 : “Si on ne va pas nager alors on va faire du canoë.”
- R3 : “Si on va faire du canoë alors on rentrera tôt à la maison.”

Prouvons le théorème suivant: “**on rentrera tôt à la maison.**”

On réalise souvent des **preuves par réfutation.**

Méthodes syntaxiques

Preuve automatique de théorèmes

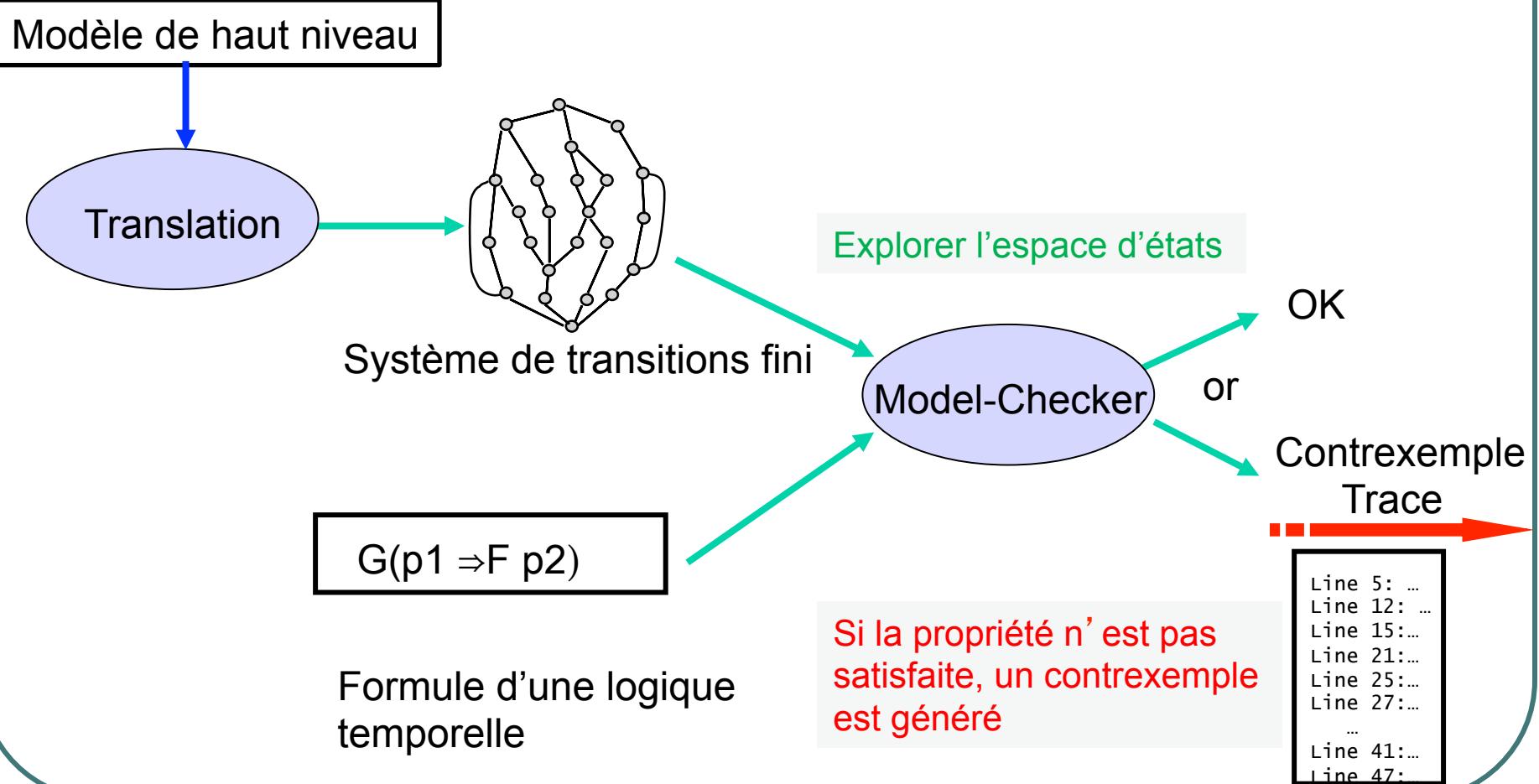


Méthodes sémantiques : Model-checking & Synthèse de contrôleur

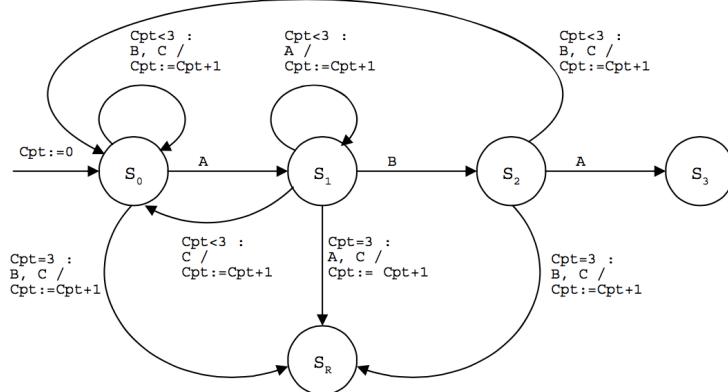
Méthodes sémantiques

- Elles se basent sur l' exécution du modèle (l' exploration de l' espace des états).
- L' approche la plus populaire est le «model-checking » qui s'appuie sur deux formalismes :
 - système de transitions et
 - logique temporelle.
- Les premiers travaux sur le model-checking sont attribués à :
 - Edmund Clarke, Allen Emerson et
 - Joseph Sifakisqui ont reçu le prix **Turing en 2007** pour leurs contributions dans ce domaine. (http://fr.wikipedia.org/wiki/Prix_Turing).

Model-checking

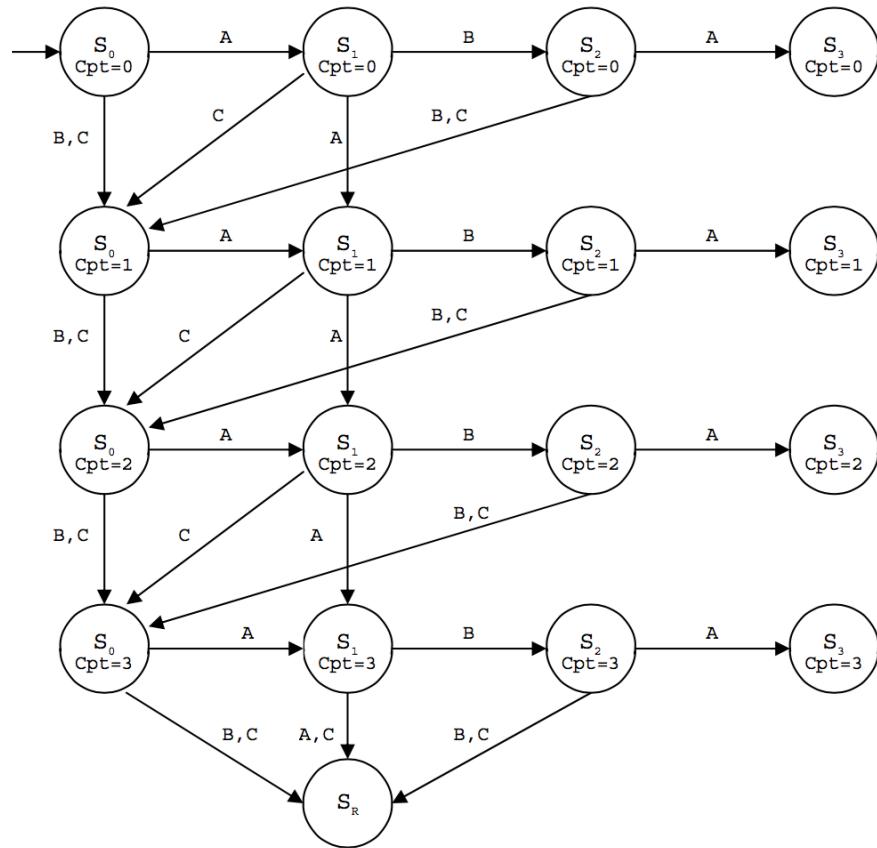


Model-checking



Automate d'un digicode

Propriété :
 $G (s_R \Rightarrow Cpt == 4)$



Son système de transitions

Model-checking

- Model-checking est totalement automatique et produit des contrexemples sous forme de traces qui sont utiles à la compréhension des situations d'erreur et à la correction.
- Il est, en général, basé sur la sémantique d'entrelacements de comportements parallèles « interleaving semantics ».
- Soit un système composé de n processus P_1, \dots, P_n concurrents. Si chaque processus P_i réalise une seule action, la sémantique d'entrelacements pourrait générer $n!$ évolutions et plus de 2^n états.
- Model-checking se heurte au problème d'**explosion combinatoire** de l'espace des états.

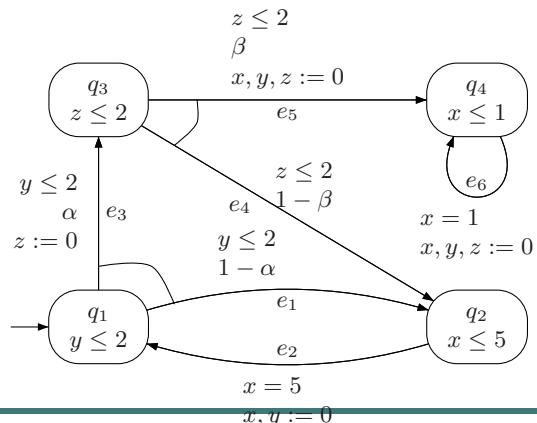
➔ Comment atténuer ce problème ?

Quelques model-checkers

- SPIN (1997) est un **model-checker de propriétés linéaires et de spécifications PROMELA**, développé par les laboratoires Bell.
- Uppaal (1997) est **un model-checker d'automates temporisés (étendus au langage C) et d'un sous-ensemble de propriétés de branchement**, développé par les universités Uppsala et Alborg.
- Hytech (1997) est **un outil de vérification par model-checking dédié aux automates hybrides** (variables discrètes et continues). Il est développé par l'université de Berkeley. Il est capable de calculer les paramètres pour lesquels le modèle satisfait des propriétés temporelles.

Quelques model-checkers

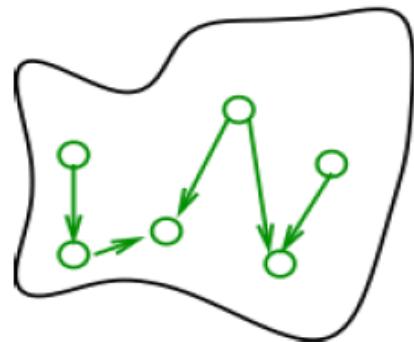
- NuSMV (1999) et NuXMV (2014) sont des **model-checkers** développés par CMU (Carnegie Mellon University) et (IRST (Istituto per la Ricerca Scientifica e Tecnologica). Les propriétés peuvent être linéaires ou de branchement. NuXMV étend NuSMV en permettant notamment la spécification et la vérification de systèmes infinis. Il intègre plusieurs techniques **de model-checking**.
- CBMC, LLBMC (2013) est un **model-checker pour les programmes C/C++**.
- PRISM (Probabilistic Model Checker) est un **model-checker pour les automates temporisés probabilistes**.



Synthèse de contrôleur

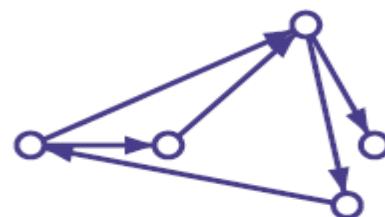
Synthèse de contrôleur

- Peut-on forcer le comportement du système afin de satisfaire sa spécification ?
- Si oui, comment ?



Un système
de transitions

||



Existe-t-il un
contrôleur ?

φ



Logique
temporelle

Outil UPPAAL-TIGA