Introduction à MEC
La vérification par model-checking

Pascal André

LINA - Université de Nantes
2, rue de la Houssinière- BP 92208 - 44322 Nantes Cedex 03
Pascal.Andre@irin.univ-nantes.fr

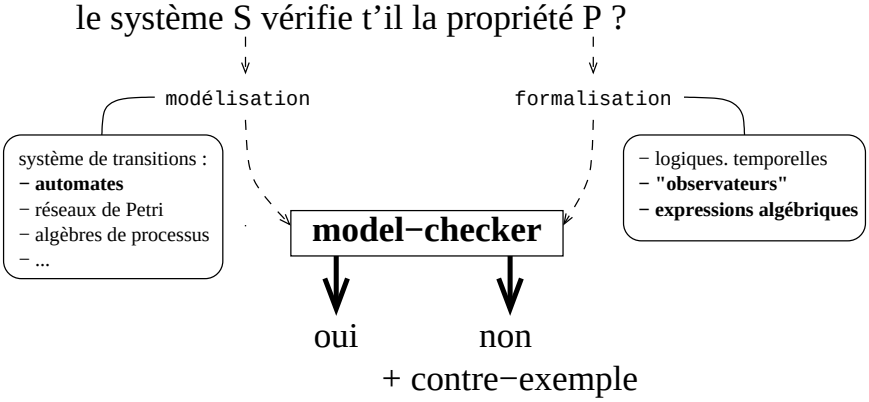
Année 2004-2005

1 Introduction

Ce document est une introduction au "*model-checking*" de systèmes concurrents avec l'outil MEC.

1.1 Vérification de modèles

Le "*model-checking*" désigne un ensemble de techniques **automatisables** permettant de **prouver formellement** qu'un **modèle vérifie une propriété**. On s'intéresse principalement aux **systèmes discrets** et aux propriétés **comportementales**. L'ouvrage collectif [SBB+99] est une excellente présentation des techniques et outils du model-checking pour les systèmes concurrents (modèles, propriétés, systèmes de vérification). Nous nous sommes aussi inspirés du cours de DESS de Sébastien Faucou.



Le model-checker permet d'affirmer formellement que **le modèle de S** vérifie **l'expression formelle de P**.

Il existe d'autres méthodes de vérification ([SBB+99] p. XIII) :
- tests : ils ne sont pas exhaustifs
- démonstration automatique (preuves en logique) : assez lourd, pas entièrement automatique
- *model-checking* : exhaustive, en grande partie automatique, mais domaine limité

et d'autres systèmes avec d'autres propriétés attendues : systèmes continus (temps), séquentiels-statiques (complétude, cohérence, non-redondance).

1.2 Modèles de systèmes concurrents

Les modèles qui nous intéressent sont représentés par des systèmes de transitions [Arn92] ou des automates ([AV01] chap. 5 et [AV02] chap. 4). Plus précisément, on s'intéresse aux automates composés par un vecteur de synchronisation ([AV01] p. 144). Il existe d'autres formalismes : variantes

des machines de Moore ou de Mealy, automates hiérarchiques, réseaux de Petri, Grafcet, algèbres de processus (CCS, CSP, Lotos), langages synchrones (Lustre, Esterel, Signal) ou asynchrones (Electre, SDL, Estelle), etc.

Le formalisme le plus simple est celui des **systèmes de transitions étiquetées** avec une composition synchrone par **produit synchronisé**. Les fondements théoriques de cette approche sont dûs à A. ARNOLD et M. NIVAT (on parle du modèle « ARNOLD-NIVAT »)

Un **système de transitions étiquetées** S sur un **alphabet d'actions** $\mathcal{A}$ est un quintuplet $\langle E, T, \alpha, \beta, \lambda \rangle$, où :

- E est un **ensemble fini d'états**
- T est un **ensemble fini de transitions**
- α et β sont les fonctions de $T \to E$, qui associent à chaque transition t son état **source** $\alpha(t)$ et son état **cible** $\beta(t)$
- λ est la fonction de $T \to \mathcal{A}$, qui **étiquette** chaque transition t par l'action ou événement $\lambda(t)$ dont l'occurrence provoque la transition

Prenons l'exemple d'une pile bornée à 3 emplacements ([AV01] p. 138).

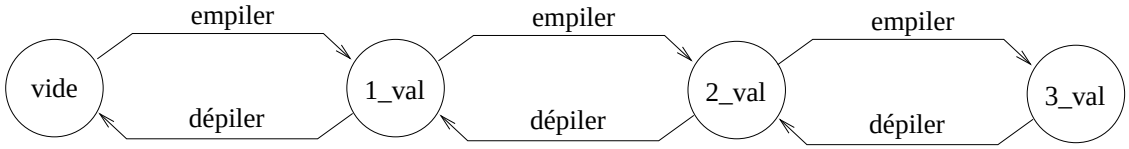


Figure 1 : *Automate de pile bornée à trois emplacements*

Le système de transition est $S = \langle E, T, \alpha, \beta, \lambda \rangle$ défini sur l'alphabet d'actions $\mathcal{A}$ où :

- $\mathcal{A} = \{\texttt{empiler}, \texttt{depiler}\}$
- $E = \{\texttt{vide}, \texttt{1\_val}, \texttt{2\_val}, \texttt{3\_val}\}$
- $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$
- $\alpha = \{(t_1 \mapsto \texttt{vide}), (t_2 \mapsto \texttt{1\_val}), (t_3 \mapsto \texttt{1\_val}), (t_4 \mapsto \texttt{2\_val}), (t_5 \mapsto \texttt{2\_val}), (t_6 \mapsto \texttt{3\_val})\}$
- $\beta = \{(t_1 \mapsto \texttt{1\_val}), (t_2 \mapsto \texttt{vide}), (t_3 \mapsto \texttt{2\_val}), (t_4 \mapsto \texttt{1\_val}), (t_5 \mapsto \texttt{3\_val}), (t_6 \mapsto \texttt{2\_val})\}$
- $\lambda = \{(t_1 \mapsto \texttt{empiler}), (t_2 \mapsto \texttt{depiler}), (t_3 \mapsto \texttt{empiler}), (t_4 \mapsto \texttt{depiler}), (t_5 \mapsto \texttt{empiler}), (t_6 \mapsto \texttt{depiler})\}$

Traitons maintenant le cas d'un système concurrent obtenu par synchronisation de systèmes de ansitions simples. Par exemple, supposons que les piles bornées soient composées en série comme le ontre la figure 2.

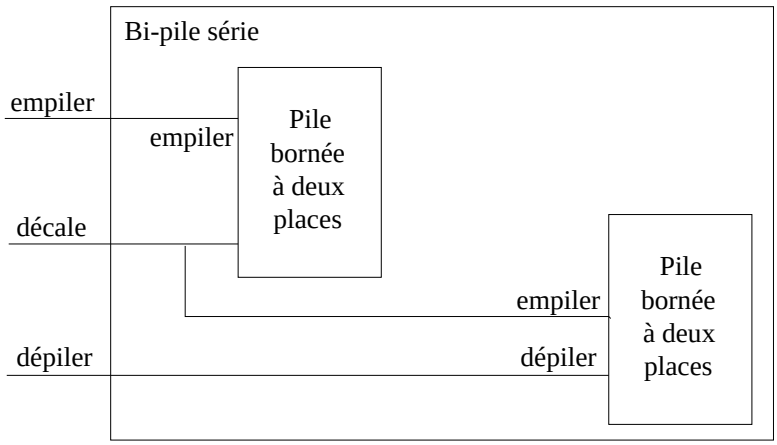


Figure 2 : *Piles en série*

Le **produit synchronisé**, défini par ARNOLD et NIVAT, est un produit libre contraint par un vecteur de synchronisation. L'automate (composé) global est obtenu par un produit des états et transitions des automates composants. Les actions autorisées, symbolisées ici par les actions globales de la bi-pile, sont définies par un vecteur de synchronisation, qui fixe les combinaisons acceptables.

```
empiler   :<  empiler  ε>
décaler   :<  dépiler  empiler>
dépiler   :<        ε  dépiler>
```

L'alphabet $\mathcal{A}$ de la pile bornée a été enrichi d'une nouvelle action ε qui correspond à une action vide. Des transitions ont été ajoutées, avec le label ε, qui ne changent pas d'état. Par exemple `vide` $\stackrel{\varepsilon}{\mapsto}$ `vide`

L'automate résultat est donné dans la figure 3. Ce petit exemple illustre le problème d'explosion combinatoire horizontale des automates. Notez que le passage de paramètres n'est pas précisé dans cette notation. Il peut se traduire dans les opérations associées aux transitions.

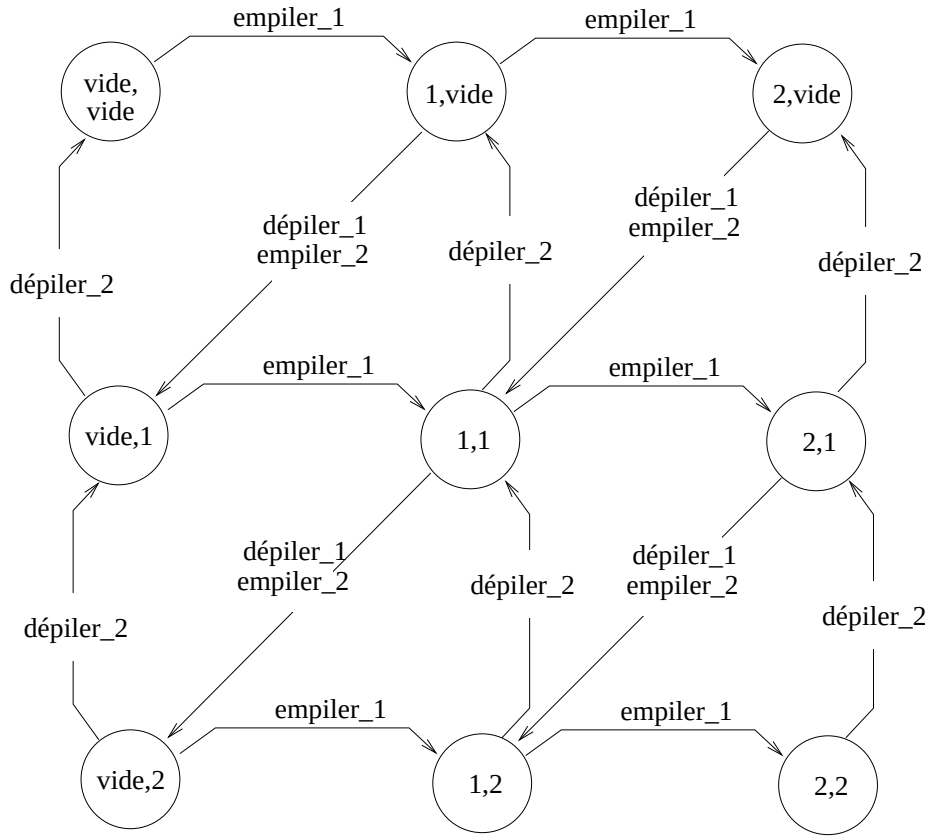


Figure 3 : *Automate synchronisé de la bi-pile*

1.3 Propriétés des systèmes concurrents

On cherche à répondre à la question : Le système S vérifie t'il la propriété P ?
- on dispose d'un modèle S' de S
- on souhaite exprimer P

Idée : on reformule la question sur S en une question sur S'.
Exemple :

« S n'est jamais bloqué »
devient
« S' ne comporte pas d'états puits »

Les propriétés générales attendues pour le comportement dynamique sont la sûreté (absence d'erreurs à l'exécution) et la vivacité (l'absence de blocage du système), la bonne répartition des ressources (absence de famine), l'absence de comportements infiniment bloquants (boucles), etc ([AV01] p. 134).
- les **propriétés de sûreté** (safety) : quelque chose de mauvais n'arrive jamais (deadlock, non respect de l'exclusion mutuelle, etc.)
- les **propriétés de vivacité** (liveness) : quelque chose de bon finit toujours par arriver (accès à une ressource partagée, etc.)

– les **propriétés d'équité** (fairness) : aucune entité n'est favorisée (système symétrique), ou pour le moins aucune entité n'est lésée

Les propriétés sont exprimées à partir de logiques, temporelles ou pas (ex : PTL, CTL, CTL*, etc.), d'algèbres (mu-calcul) ou d'algorithmes.

1.4 Outil MEC

MEC est un outil de description de systèmes de transitions et de vérification de propriétés. MEC a été développé au LABRI à l'université de Bordeaux. La présentation qui suit est basée sur l'ouvrage de référence de MEC [ACB94], sur des exemples fournis avec MEC 4 et des extraits de cours d'Alain Griffault [Gri98].

MEC permet de définir des systèmes concurrents basés sur la synchronisation d'automates (de systèmes de transitions) et de déterminer des propriétés par application d'algorithmes. MEC est un outil qui manipule :

– les systèmes de transition avec marques,
– les systèmes de synchronisation,
– le produit de synchronisation,
– les fonctions prédéfinies,
– les fonctions définies par l'utilisateur,
– des fonctions d'affichage et d'archivage.

Les propriétés sont exprimées par des expressions utilisant des opérateurs (arithmétiques ou logiques) et des fonctions sur les graphes (prédéfinies ou définies par l'utilisateur). Il s'agit essentiellement d'exprimer des propriétés structurelles sur le modèle. Les fonctions sont réalisées par des algorithmes sur les graphes.

2 MEC

2.1 Premier exemple avec MEC

Reprenons l'exemple de la pile série (figure 2) avec cette fois 3 emplacements dans la pile.

Le lancement de MEC se fait par la commande `mec`.

```
$./bin/mec
```

Une fenêtre (textuelle) standard s'affiche, qui contient une zone d'information et une zone de dialogue. On quitte MEC par la commande '`quit() ;`' ou par '`^C`' . Le chargement du fichier contenant la session est réalisé par la commande `load()`.

```
load("./Demos/Piles/pileSerie");
```

Le fichier source contenant la session à exécuter (définition du système, propriétés à calculer) est le suivant.

```
\* P. ANDRE
   Pile Serie
   15/01/04

   load(pileSerie);
*\

transition_system pileB < width = 0 >;
v |- e -> v,
     emp -> 1;
1 |- e -> 1,
     emp -> 2,
     dep -> v;
```

```
2 |- e -> 2,
     emp -> 3,
     dep -> 1;
3 |- e -> 3,
     dep -> 2;
< initial = { v } >.

synchronization_system PileSerie < width = 2 ;
    list = ( pileB, pileB ) >;

( dep . emp );
( emp . e );
( e . dep ).

function inevitable(Y:trans ; X:state) return Z:state;
begin
Z = X \/ (src(Y /\ rtgt(Z)) - src(Y /\ rtgt(*-Z)))
end.

sync(PileSerie, PileSerie);
dts(PileSerie);
dead := * - src(*);
deadlock:=inevitable(*,dead);
boucle := loop(*,*);
log(pileSerieLog);
version();
dts(PileSerie);
wts(*,*);   \* affiche le système global *\
stoplog();
```

Le fichier comprend la description d'un système de transitions simple (`pileB`) pour la pile bornée et d'un système synchronisé (`PileSerie`) pour la pile série. Noter qu'on peut synchroniser, à volonté, des systèmes synchronisés et des systèmes simples. Le système `PileSerie` est décrit par ses composants et son vecteur de synchronisation.

La fonction `inevitable(Y,X)` calcule les états à partir desquels il est inévitable d'aboutir dans `X` en employant uniquement des transitions de `Y`.

Les expressions et les propriétés calculées sont les suivantes :
- `sync` : calcule le produit synchronisé.
- `dts(sync)` : fixe l'automate courant
- `dead := * - src(*)` : détermine les états puits, qui n'ont pas de transitions sortantes : ce sont les états bloquants du système.
- `deadlock :=inevitable(*,dead)` : indique si le blocage est inévitable.
- `boucle := loop(*,*)` : calcule le nombre de boucles de l'automate.
- le reste de la session rempli le fichier résultat.

Le résultat obtenu est

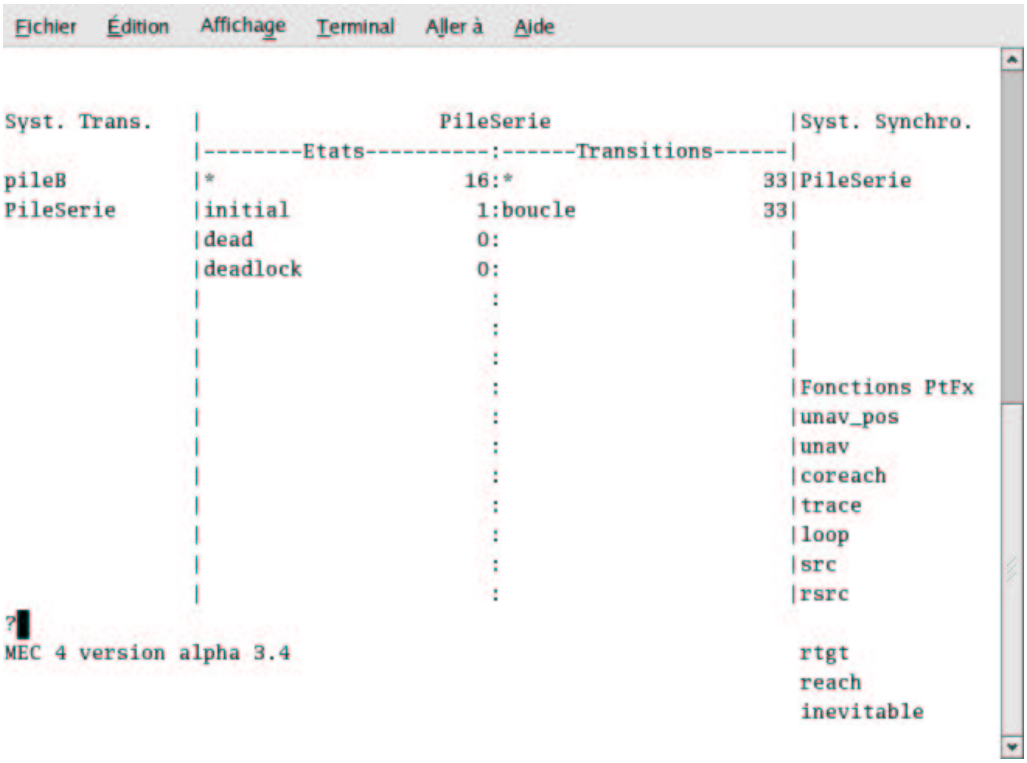


Figure 4 : *Résultats du cas pile série*

Le fichier résultat est le `pileSerieLog`.

```
Current St : PileSerie
//-- States :
// * : 16
//initial : 1
//dead : 0
//deadlock : 0
//-- Transitions :
// * : 33
//boucle : 33
//
wts(*,*);
//transition_system PileSerie
//< width = 2;  list = (pileB, pileB )>;
//
//
//e(v.v) |-
// (emp.e) -> e(1.v) < property = ( boucle  ) >;
//
//e(1.v) |-
// (emp.e) -> e(2.v) < property = ( boucle  ) >,
// (dep.emp) -> e(v.1) < property = ( boucle  ) >;
//
//e(2.v) |-
// (emp.e) -> e(3.v) < property = ( boucle  ) >,
// (dep.emp) -> e(1.1) < property = ( boucle  ) >;
//
//e(v.1) |-
// (e.dep) -> e(v.v) < property = ( boucle  ) >,
// (emp.e) -> e(1.1) < property = ( boucle  ) >;
//
//e(3.v) |-
// (dep.emp) -> e(2.1) < property = ( boucle  ) >;
//
//e(1.1) |-
// (e.dep) -> e(1.v) < property = ( boucle  ) >,
```

```
// (emp.e) -> e(2.1) < property = ( boucle  ) >,
// (dep.emp) -> e(v.2) < property = ( boucle  ) >;
//
//e(2.1) |-
// (e.dep) -> e(2.v) < property = ( boucle  ) >,
// (emp.e) -> e(3.1) < property = ( boucle  ) >,
// (dep.emp) -> e(1.2) < property = ( boucle  ) >;
//
//e(v.2) |-
// (e.dep) -> e(v.1) < property = ( boucle  ) >,
// (emp.e) -> e(1.2) < property = ( boucle  ) >;
//
//e(3.1) |-
// (e.dep) -> e(3.v) < property = ( boucle  ) >,
// (dep.emp) -> e(2.2) < property = ( boucle  ) >;
//
//e(1.2) |-
// (e.dep) -> e(1.1) < property = ( boucle  ) >,
// (emp.e) -> e(2.2) < property = ( boucle  ) >,
// (dep.emp) -> e(v.3) < property = ( boucle  ) >;
//
//e(2.2) |-
// (e.dep) -> e(2.1) < property = ( boucle  ) >,
// (emp.e) -> e(3.2) < property = ( boucle  ) >,
// (dep.emp) -> e(1.3) < property = ( boucle  ) >;
//
//e(v.3) |-
// (e.dep) -> e(v.2) < property = ( boucle  ) >,
// (emp.e) -> e(1.3) < property = ( boucle  ) >;
//
//e(3.2) |-
// (e.dep) -> e(3.1) < property = ( boucle  ) >,
// (dep.emp) -> e(2.3) < property = ( boucle  ) >;
//
//e(1.3) |-
// (e.dep) -> e(1.2) < property = ( boucle  ) >,
// (emp.e) -> e(2.3) < property = ( boucle  ) >;
//
//e(2.3) |-
// (e.dep) -> e(2.2) < property = ( boucle  ) >,
// (emp.e) -> e(3.3) < property = ( boucle  ) >;
//
//e(3.3) |-
// (e.dep) -> e(3.2) < property = ( boucle  ) >;
//<
//initial = { e(v.v) },
//dead = {  },
//deadlock = {  }
//>.
//Current St : PileSerie
//-- States :
// * : 16
//initial : 1
//dead : 0
//deadlock : 0
//-- Transitions :
// * : 33
//boucle : 33
//
stoplog();
```

D'autres exemples sont donnés en annexe.

2.2 Expression de propriétés avec MEC

Ce qui suit est extrait du cours d'Alain Griffault [Gri98].

Une façon simple et naturelle d'exprimer les propriétés d'un système de transition est de fournir comme résultat du calcul, les états ou les transitions qui confirment ou infirment cette propriété. A cette fin, le modèle des systèmes de transition est étendu au modèle des systèmes de transitions marqués. Une marque d'état (resp de transition) est un sous-ensemble des états (resp des transitions).

2.2.1 Les fonctions de base d'une logique

Les marques élémentaires

- `{}` : l'ensemble vide est une marque soit d'état, soit de transition.
- `*` : l'ensemble total est une marque soit d'état, soit de transition.
- `initial` : les états initiaux d'un système de transition.
- `!label op "chaine-de-caracteres"` : marque de transition.
  avec op un opérateur parmi : `{=, #, <, <=, >, >=}`.
- `!state op "chaine-de-caracteres"` : marque d'état.
- `!label[i] op "chaine-de-caracteres"` : marque de transition.
- `!state[i] op "chaine-de-caracteres"` : marque d'état.
- `marque-transition[i]` : marque de transition.
- `marque-etat[i]` : marque d'état.

Les opérateurs ensemblistes sur les marques

- **union** : `A \/ B`, avec `A` et `B` des marques soit d'état, soit de transition.
- **intersection** : `A /\ B`, avec `A` et `B` des marques soit d'état, soit de transition.
- **différence** : `A - B`, avec `A` et `B` des marques soit d'état, soit de transition.

Les fonctions relatives à un ensemble de transitions

- `src` : les états source d'une marque de transition.
- `tgt` : les états but d'une marque de transition.
- `rsrc` : fonction inverse de `src`.
  Attention `src(rsrc(S)) = S`, mais `rsrc(src(T)) ⊆ T`.
- `rtgt` : fonction inverse de `tgt`.
  Attention `tgt(rtgt(S)) = S`, mais `rtgt(tgt(T)) ⊆ T`.

Quelques fonctions pratiques

- `succ` : Cette expresssion calcule les successeurs d'un ensemble d'états par un ensemble de transitions.
  `succ(S,T) = tgt(rsrc(S) /\ T)`.
- `pred` : Cette expresssion calcule les prédécesseurs d'un ensemble d'états par un ensemble de transitions.
  `pred(S,T) = src(rtgt(S) /\ T)`.
- `reach` : Cette expresssion calcule les successeurs d'un ensemble d'états par un ensemble de chemins.
  `reach(S,T) = succ*(S,T)`.
- `coreach` : Cette expresssion calcule les prédécesseurs d'un ensemble d'états par un ensemble de chemins.
  `coreach(S,T) = pred*(S,T)`.
- `proj-state[i]` : Projection d'une marque d'état sur le $i^{ème}$ composant.
- `proj-trans[i]` : Projection d'une marque de transition sur le $i^{ème}$ composant.

2.2.2 Quelques propriétés simples usuelles

Exprimer les propriétés suivantes en terme de marque d'état ou de transition.

Question 2.1 *Le système risque t-il de se bloquer ?*

Réponse 2.1
Le système risque de se bloquer s'il existe un état accessible duquel ne part aucune transition.
L'expression `* - src(*)` calcule tous les états d'un système de transition (`*`) qui ne sont pas des états sources d'une transition `(- src(*))`.

Question 2.2 *Tous les états du $i^{ème}$ composant sont-ils utilisés ?*

Réponse 2.2
Pour comprendre l'intérêt d'un tel calcul, on peut faire l'analogie avec un développeur qui écrit des fonctions qui ne sont jamais appelées par le programme principal.
L'expression `(*`$_i$ `- proj-state(i ; *))` calcule l'ensemble des états du i^{ème} composant qui sont inaccessibles dans le système global.

Question 2.3 *Toutes les transitions du $i^{ème}$ composant sont-elles utilisées ?*

Réponse 2.3
L'expression `(*`$_i$ `- proj-trans(i ; *))` calcule l'ensemble des transitions du i^{ème} composant qui sont inaccessibles dans le système global.

Question 2.4 *Une marque de transition `Y` , lorqu'elle est possible dans le $i^{ème}$ composant, est-elle toujours possible dans le système global ?*

Réponse 2.4
L'expression `(src(Y )[i] - src(Y [i]))` calcule l'ensemble des états du graphe global dans lesquels l'action `Y` est impossible du fait du contexte, alors qu'elle est localement possible sur le i^{ème} composant.

2.2.3 Une séance type d'utilisation

Il est préférable d'enregistrer dans un premier temps dans des fichiers les modélisations des systèmes de transition et du système de synchronisation. Puis au prompt de l'outil mec :

1. `load("NomDuFichierContenantSystemesTransition") ;`
2. `load("NomDuFichierContenantSystemeSynchronisation") ;`
3. `sync(NomSystemeSynchronisation, NomSystemeTransitionResultat) ;`
4. `dts(NomSystemeTransitionResultat) ;`
5. `dead := * - src(*) ;`
6. `wts(dead,*) ;`
7. D'autres calculs de propriétés.
8. `wts(PetiteMarqueEtat, *) ;`
9. `wts(src(PetiteMarqueTransition), PetiteMarqueTransition) ;`
10. L'enregistrement de certains résultats par :
    (a) `ats(NomFichierArchive, MarqueEtat, MarqueTransition, NomSystemeTransitionResultat) ;`
    (b) l'enregistrement d'une session :
        i. `log(NomFichierArchive) ;`
        ii. Liste de commandes
        iii. `stoplog() ;`
11. `quit() ;`

Il est évidememnt possible de mettre la suite des calculs dans un fichier, puis d'effectuer un `load` de ce fichier.

2.3 Principales commandes de MEC

L'ensemble des commandes est défini dans le manuel de référence ([ACB94] chapitre 14).

2.3.1 Affichages

- `wts(E, T [,ST])` : affiche le sous-graphe du système `ST` (ou par défaut du système courant) des états ayant la propriété `E` et restreint aux transitions de `T`.
  Exemple : `wts(initial,*)`, `wts(dead,*)`, `wts(*,*)` complet
- `wss(STS)` : affiche le système synchronisé `STS`.
  Exemple : `wss(PileSerie)`

2.3.2 Calculs

Un calcul est l'affectation du résultat d'une expression à une variable. Pour éviter toute ambiguïté, le résultat doit être correctement typé. La caractère '`*`' est polymorphe, il désigne, selon son utilisation, un ensemble d'états, de transitions...

- `loop(T1, T2` : boucles et équité ([ACB94], p. 32) :
  Ensemble de transitions `t` telles que `t` $\in$ `loop(R, R')` $\Leftrightarrow$ `t` $\in$ `p` où `p` est un chemin non vide :
  1. la source de `p` est égale à sa cible (c'est une boucle)
  2. chaque transition de `p` est dans `R'`
  3. certaines transitions de `p` sont dans `R`

  Exemple : `loop(*,*)`
- `trace(E,T)` : définit un ensemble de transitions formant un chemin de l'état initial (facultatif) à un état de `E` avec les transitions de `T`.
  Exemple : `trace(initial,*,dead)` - philosophes
- `sync(STS,res)` : synchronise le système `STS`, le résultat est le système de transition (simple) `res`.
  Exemple : `sync(NomSystemeSynchronisation, NomSystemeTransitionResultat)`

2.3.3 Session

- `dts(ST)` : `ST` devient le système de transition courant.
- `new()` : réaffiche l'écran.
- `verbose()` : affiche les messages internes.
- `timer()` : affiche les consommations de ressources de la dernière commande.

2.3.4 Archives

- `load(fic)` : charge le fichier `fic`.
- `ar_ts(fic, S, T [, ST])` : charge la définition du système courant (ou `ST` si défini) restreint aux états de `E` et aux transitions de `T`.
- `ar_ss(fic, ST)` : charge la définition du système `ST`.
- `ar_all(fic)` : charge la définition des systèmes dans le fichier `fic`.

2.3.5 Divers

- `stop()` : quitte mec.
- `help([sujet])` : aide en ligne.
- `csh()` : accès au shell.

2.3.6 Fonctions usuelles définies par l'utilisateur

Monotonie

Les fonctions récursives sont calculées par un algorithme de résolution au point fixe. Un syst'eme d'équations au point fixe, lorsqu'il respecte certaines règles de monotonie, possède une solution. Cette règle syntaxique est implémentée dans MEC.

Les systèmes suivants qui définissent les états `pair`, `impair` et `nonimpair` vérifient la règle de monotonie :

```
function pair(S : state) return Z : state;
  var pair : state;
      impair : state
begin
  pair = S \/ tgt(rsrc(impair);
  impair = tgt(rsrc(pair));
  Z = pair
end.

function impair(S : state) return Z : state;
  var pair : state;
      impair : state
begin
  pair = S \/ tgt(rsrc(impair);
  impair = tgt(rsrc(pair));
  Z = impair
end.

function nonimpair(S : state) return Z : state;
  var pair : state;
      nonimpair : -state
begin
  pair = S \/ tgt(rsrc(* - nonimpair);
  nonimpair = * - tgt(rsrc(pair));
  Z = nonimpair
end.
```

Par contre, les exemples suivants ne respectent pas la règle.

```
function incorrecte1(S : state) return Z : state;
  var pair : state;
      nonimpair : state
begin
  pair = S \/ tgt(rsrc(* - nonimpair);
  nonimpair = * - tgt(rsrc(pair));
  Z = nonimpair
end.

function incorrecte2(S : state) return Z : state;
  var pair : state ou -state;
      nonimpair : state ou -state
begin
  pair = S \/ tgt(rsrc(* - nonimpair);
  nonimpair = tgt(rsrc(pair));
```

```
  Z = nonimpair
end.
```

Des exemples de fonctions

On définit à l'aide de système d'équations au point fixe les fonctions suivantes.

La fonction `inevitable` qui calcule les états `Z` à partir desquels il est inévitable d'aboutir dans `X` si l'on n'utilise que des transitions appartenant à `Y`.

```
function inevitable(Y:trans ; X:state) return Z:state;
begin
  Z = X \/ (src(Y /\ rtgt(Z)) - src(Y /\ rtgt(*-Z)))
end.
```

Autre version ([ACB94], p. 31)

```
function unavoidable(Zt:trans ; Zs:state) return X:state;
var Y :_ trans
begin
X = Zs \/ (* - src(Y));
Y = Zt /\ rtgt(* - X)
end.
```

La fonction `perte` qui calcule les transitions appartenant à un chemin allant d'une émission à une autre sans réception (exemple du protocole du bit alterné en annexe).

```
function perte(emis : trans; recu : trans) return chemin : trans;
  var t1 : trans;
      t2 : trans
begin
  t1 = ( emis - recu ) \/ (rsrc(tgt(t1)) - recu);
  t2 = rtgt(src(t2 \/ emis)) - recu;
  chemin = t1 /\ t2
end.
```

La fonction `duplique` qui calcule les transitions appartenant à un chemin allant d'une émission à une autre avec duplication de réception (exemple du protocole du bit alterné en annexe).

```
function duplique(emis : trans; recu : trans) return chemin : trans;
  var t1 : trans;
      t2 : trans
begin
  t1 = ( recu - emis ) \/ (rtgt(src(t1)) - emis);
  t2 = rsrc(tgt(t2 \/ recu)) - emis;
  chemin = t1 /\ t2
end.
```

La fonction `last_lost2` qui calcule les états gagnants dans un jeu à deux joueurs et dans lequel les états puits sont gagnants pour celui qui doit jouer (exemple du jeu de Nim en annexe).

```
function last_lost2(X:state) return Z:state;
  var T:state
begin
  Z=X\/src(rtgt(T));
  T=src(rtgt(Z)) - src(rtgt(*-Z))
end.
```

La fonction `last_wint2` (`last_wint3`) qui calcule les états gagnants dans un jeu à deux (trois) joueurs et dans lequel les états puits sont perdants pour celui qui doit jouer (exemple du jeu de Nim en annexe).

```
function last_win2(X:state) return Z:state;
  var T:state
begin
  Z=src(rtgt(T));
  T=X\/ (src(rtgt(Z)) - src(rtgt(*-Z)))
end.

function last_win3(X:state) return Z:state;
  var T:state;
      U:state
begin
  Z=src(rtgt(T));
  T=X\/ (src(rtgt(U)) - src(rtgt(*-U)));
  U=X\/ (src(rtgt(Z)) - src(rtgt(*-Z)))
end.
```

La fonction `last_lost3` qui calcule les états gagnants dans un jeu à trois joueurs et dans lequel les états puits sont gagnants pour celui qui doit jouer (exemple du jeu de Nim en annexe).

```
function last_lost3(X:state) return Z:state;
  var T:state;
      U:state
begin
  Z=X\/src(rtgt(T));
  T=src(rtgt(U)) - src(rtgt(*-U));
  U=src(rtgt(Z)) - src(rtgt(*-Z))
end.
```

2.4 Méthodologie d'utilisation de MEC

Les conseils qui suivent sont extraits de cours d'Alain Griffault [Gri98].

Définition 2.5
*La **modélisation** consiste à définir un modèle abstrait.*
*La **validation** consiste à montrer que le modèle abstrait est un bon candidat pour représenter le monde réel.*
*La **vérification** consiste à prouver que le modèle abstrait, et donc certainement le monde réel, vérifie les spécifications.*

Processus de travail

1. Concevoir un premier modèle avec beaucoup d'abstractions.
2. Valider le modèle.
3. Itérer 1 et 2 en augmentant la liste des hypothèses, jusqu'à ce que la validation soit satisfaisante.
4. Vérifier le modèle
5. Itérer 1, 2, 3 et 4 en augmentant la liste des hypothèses, jusqu'à ce que la vérification soit satisfaisante
6. Itérer 1, 2, 3, 4 et 5 en :
   - raffinant le modèle.
   - augmentant la liste des propriétés vérifiées.

   jusqu'à ce que l'une des conditions suivantes soit vraie :
   - Le temps imparti pour la spécification est terminé.
   - Le modèle est une implémentation.
   - Le système de transition global est trop important pour la machine utilisée.

Dans cette démarche, l'introduction d'une nouvelle hypothèse et/ou contrainte dans la description informelle doit renseigner obligatoirement les points suivants :

1. Une description en langage informel de la nouvelle contrainte en la justifiant si possible.
2. Une description formelle des différentes expressions possibles de la contrainte.
3. Une indication s'il s'agit d'une contrainte matérielle ou logicielle, et qui devra la traiter, après en avoir éventuellement référé au donneur d'ordres.
4. Le choix utilisé pour la suite de l'étude.

L'intérêt d'une telle démarche est : d'une part éviter la vérification d'un modèle non validé auparavant ; d'autre part permettre une relecture aisée de la description formelle. En pratique, cela évite de considérer des modèles non implémentables, ou non réalisables comme corrects.

La validation du modèle

Des validations syntaxiques et/ou sémantiques

Ces calculs doivent être effectués en priorité car les erreurs détectées indiquent soit des erreurs d'édition lors de la saisie du modèle, soit des erreurs de conception graves.

- Tous les états décrits sont-ils accessibles ?
- Toutes les transitions décrites sont-elles utilisées ?
- Tous les vecteurs de synchronisations décrits sont-il utilisés ?

L'idée intuitive de cette étape est la suivante : De même qu'un développeur de logiciel n'écrit pas dans un programme une fonction qui ne serait jamais appelée, le modélisateur ne décrit certainement pas des transitions, ou des interactions qui ne servent jamais dans le produit de synchronisation. Cette vérification purement syntaxique est en quelque sorte un garde fou.

Les propriétés simples et complexes usuelles

Les calculs suivants étant d'une part très généraux, d'autre part peu coûteux en temps, doivent être effectués pour toute modélisation. L'interprétation des résultats dépend à la fois de l'application et de la modélisation.

1. le blocage.
2. le blocage retardé, (inévitable).
3. la possibilité de ne pas revenir dans l'état initial.
4. l'impossibilité de revenir dans l'état initial.
5. le respect des événements incontrôlables. L'environnement d'un système réactif est généralement composé d'événements incontrôlables. Par définition, l'occurrence d'un tel événement est imprévisible, il faut donc veiller à ce qu'à tout instant le système soit capable de le recevoir.

La vérification du modèle

Dans cette partie, les propriétés vérifiées correspondent aux spécifications du cahier des charges. Elles sont très liées à la nature du problème, et c'est le donneur d'ordre qui définit l'ensemble des propriétés que doit satisfaire le modèle pour qu'il soit déclaré correct. Par exemple, un concepteur de protocoles souhaite que le protocole resiste aux pertes, aux duplications ou encore aux déséquencements des messages ; un concepteur d'algorithmes distribués désire une solution autostabilisante pour un problème d'élection. Le régulateur de vitesse est un système de type contrôle-commande, les propriétés généralement souhaitées pour de tels systèmes sont :

- une commande peut-elle être sans effet ?
- une commande peut-elle engendrer des effets indésirés ?
- la réponse à une commande peut-elle être retardée indéfiniment ?

3 Pratique de MEC

Voici une liste non exhaustive de cas pouvant servir de support aux travaux pratiques (E : Enoncé fourni - C : Code fourni).

Etudes de cas [Gri89]

- Circuits logiques - EC
- Philosophes - C
- Nim - C
- Hanoi - EC
- Bit alterné - EC
- Arnold-Nivat - EC
- autres (distributeur, ascenseur) [Gri89]

Etudes de cas Faucou

- producteur-consommateur- EC
- buffer à deux places - EC

Etudes de cas André

- piles, bipile - E [AV01]
- exo pile 11.2, 11.5 - E [AV02]
- kanban, ascenseur, distributeur, station - E [AV02]

Références

[ACB94] André Arnold, Paul Crubillé, and Didier Bégay. *Construction and Analysis of Transition Systems with MEC*. AMAST Series in Computing : Vol. 3. World Scientific, 1994. ISBN 981-02-1922-9.

[Arn92] André Arnold. *Systèmes de transitions finis et sémantique des processus communiquants*. Collection Etudes et recherches en informatique. Masson, 1992. ISBN 2-225-82746-X.

[AV01] Pascal André and Alain Vailly. *Conception de systèmes d'information, Panorama des méthodes et des techniques*, volume 1 of *Collection Technosup*. Editions Ellipses, 2001. ISBN 2-7298-0479-X.

[AV02] Pascal André and Alain Vailly. *Exercices corrigés de conception logicielle, modélisation des Systèmes d'Information par la pratique*, volume 3 of *Collection Technosup*. Editions Ellipses, 2002. ISBN 2-7298-1289-X.

[Gri89] Alain Griffault. *Contribution à l'étude des systèmes communicants et des algorithmes d'exclusion mutuelle*. PhD thesis, Université de Bordeaux I, Janvier 1989.

[Gri98] Alain Griffault. *Spécifications et vérifications formelles*. PhD thesis, Université de Bordeaux I, Mars 1998. cours de DESS (référence Web inaccessible).

[SBB+99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert Informatique. Vuibert, 1999. ISBN 2-7117-8646-3.

A First Example

```
load("./Demos/First_Example/input");
quit();
```

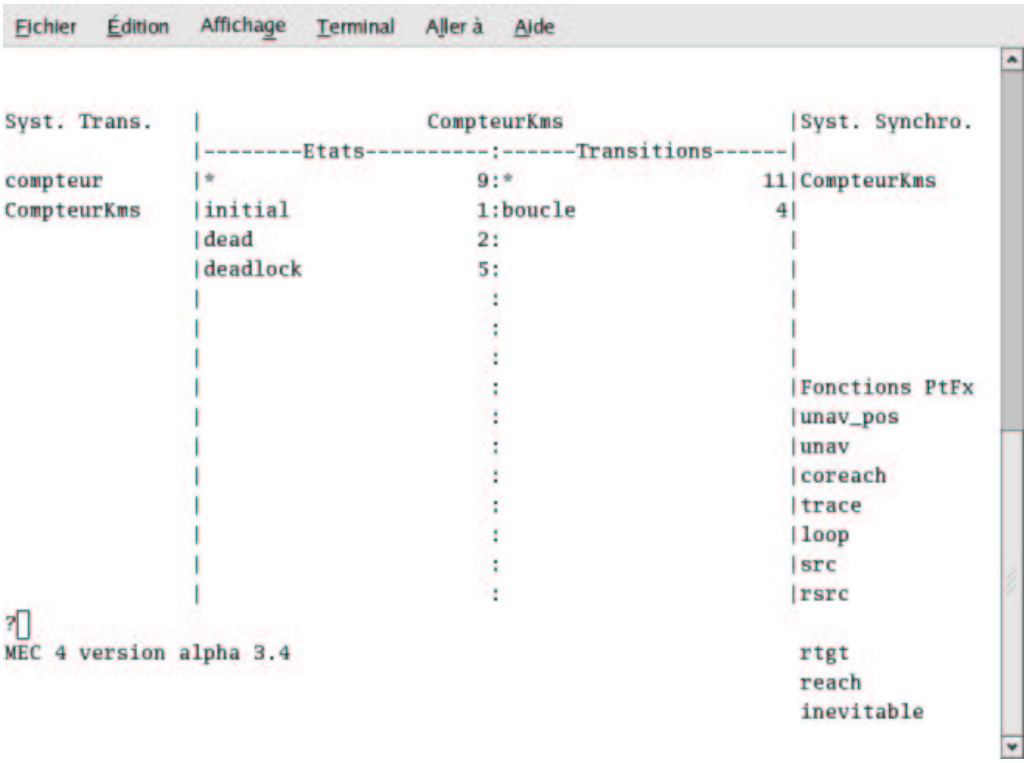


Figure 5 : *Résultats du cas First Example*

```
MEC 4 version alpha 3.4Current St : CompteurKms

//-- States :

// * : 9

//initial : 1

//dead : 2

//deadlock : 5

//-- Transitions :

// * : 11

//boucle : 4

//
```

B Hanoi

```
load("./Demos/Hanoi/input");
quit();
```

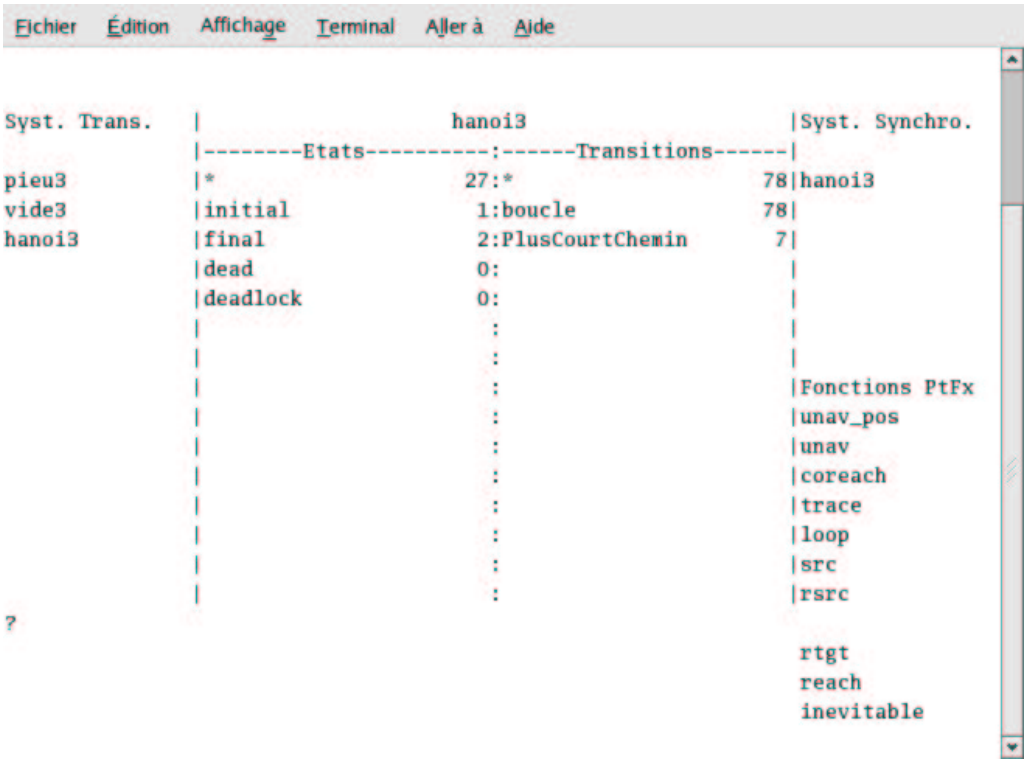


Figure 6 : *Résultats du cas Hanoi*

```
MEC 4 version alpha 3.4Current St : hanoi3
//-- States :
// * : 27
//initial : 1
//final : 2
//dead : 0
//deadlock : 0
//-- Transitions :
// * : 78
//boucle : 78
//PlusCourtChemin : 7
//transition_system hanoi3
//< width = 3;  list = (pieu3, vide3, vide3 )>;
//
//
//e(111.000.000) |-
// (d1.e.i1) -> e(110.000.001) < property = ( boucle, PlusCourtChemin  ) >, ;
//
//e(110.000.001) |-
// (d2.i2.e) -> e(100.010.001) < property = ( boucle, PlusCourtChemin  ) >;
//
//e(100.010.001) |-
// (e.i1.d1) -> e(100.011.000) < property = ( boucle, PlusCourtChemin  ) >, ;
//
//e(100.011.000) |-
// (d3.e.i3) -> e(000.011.100) < property = ( boucle, PlusCourtChemin  ) >;
//
//e(000.011.100) |-
// (i1.d1.e) -> e(001.010.100) < property = ( boucle, PlusCourtChemin  ) >, ;
//
//e(001.010.100) |-
// (e.d2.i2) -> e(001.000.110) < property = ( boucle, PlusCourtChemin  ) >, ;
//
//e(001.000.110) |-
// (d1.e.i1) -> e(000.000.111) < property = ( boucle, PlusCourtChemin  ) >, ;
//<
```

```
//initial = { e(111.000.000) },
//final = {  },
//dead = {  },
//deadlock = {  }
//>.
//
```

C Nim

```
load("./Demos/Jeu_de_Nim/input");
quit();
```

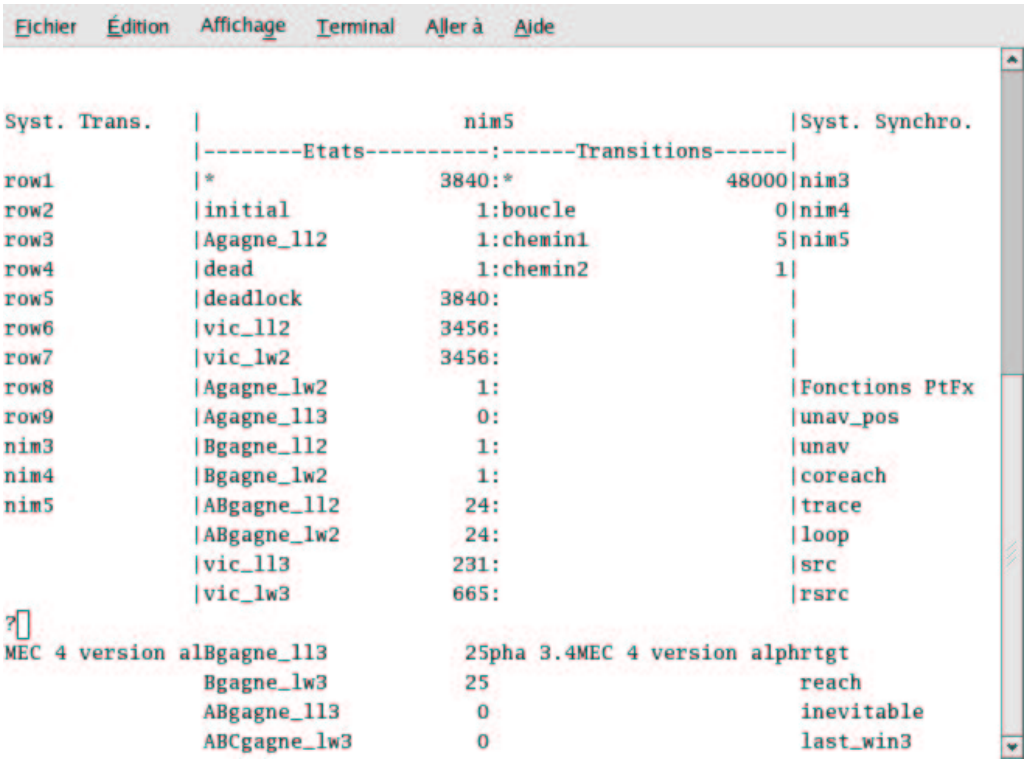


Figure 7 : *Résultats du cas Nim*

```
MEC 4 version alpha 3.4Current St : nim3
//-- States :
// * : 48
//initial : 1
//Agagne_l12 : 1
//dead : 1
//deadlock : 48
//vic_l12 : 40
//vic_lw2 : 40
//Agagne_lw2 : 1
//Agagne_l13 : 0
//Bgagne_l12 : 1
//Bgagne_lw2 : 1
//ABgagne_l12 : 8
//ABgagne_lw2 : 8
//vic_l13 : 20
//vic_lw3 : 35
//Agagne_lw3 : 0
//Bgagne_l13 : 5
//Bgagne_lw3 : 3
//ABgagne_l13 : 4
//Cgagne_l13 : 14
//Cgagne_lw3 : 6
```

```
//ABgagne_lw3 : 6
//ABCgagne_ll3 : 15
//ABCgagne_lw3 : 23
//-- Transitions :
// * : 216
//boucle : 0
//chemin1 : 3
//chemin2 : 1
//
MEC 4 version alpha 3.4Current St : nim4
//-- States :
// * : 384
//initial : 1
//Agagne_ll2 : 0
//dead : 1
//deadlock : 384
//vic_ll2 : 336
//vic_lw2 : 336
//Agagne_lw2 : 0
//Agagne_ll3 : 0
//Bgagne_ll2 : 0
//Bgagne_lw2 : 0
//ABgagne_ll2 : 16
//ABgagne_lw2 : 16
//vic_ll3 : 77
//vic_lw3 : 170
//Agagne_lw3 : 0
//Bgagne_ll3 : 16
//Bgagne_lw3 : 16
//ABgagne_ll3 : 0
//Cgagne_ll3 : 89
//Cgagne_lw3 : 74
//ABgagne_lw3 : 0
//ABCgagne_ll3 : 9
//ABCgagne_lw3 : 24
//-- Transitions :
// * : 3072
//boucle : 0
//chemin1 : 4
//chemin2 : 1
//
MEC 4 version alpha 3.4Current St : nim5
//-- States :
// * : 3840
//initial : 1
//Agagne_ll2 : 1
//dead : 1
//deadlock : 3840
//vic_ll2 : 3456
//vic_lw2 : 3456
//Agagne_lw2 : 1
//Agagne_ll3 : 0
//Bgagne_ll2 : 1
//Bgagne_lw2 : 1
//ABgagne_ll2 : 24
//ABgagne_lw2 : 24
//vic_ll3 : 231
//vic_lw3 : 665
//Agagne_lw3 : 0
//Bgagne_ll3 : 25
//Bgagne_lw3 : 25
//ABgagne_ll3 : 0
//Cgagne_ll3 : 250
//Cgagne_lw3 : 250
```

```
//ABgagne_lw3 : 0
//ABCgagne_ll3 : 0
//ABCgagne_lw3 : 0
//-- Transitions :
// * : 48000
//boucle : 0
//chemin1 : 5
//chemin2 : 1
//
```

D Philosophes

```
load("./Demos/Philosophes/input");
quit();
```

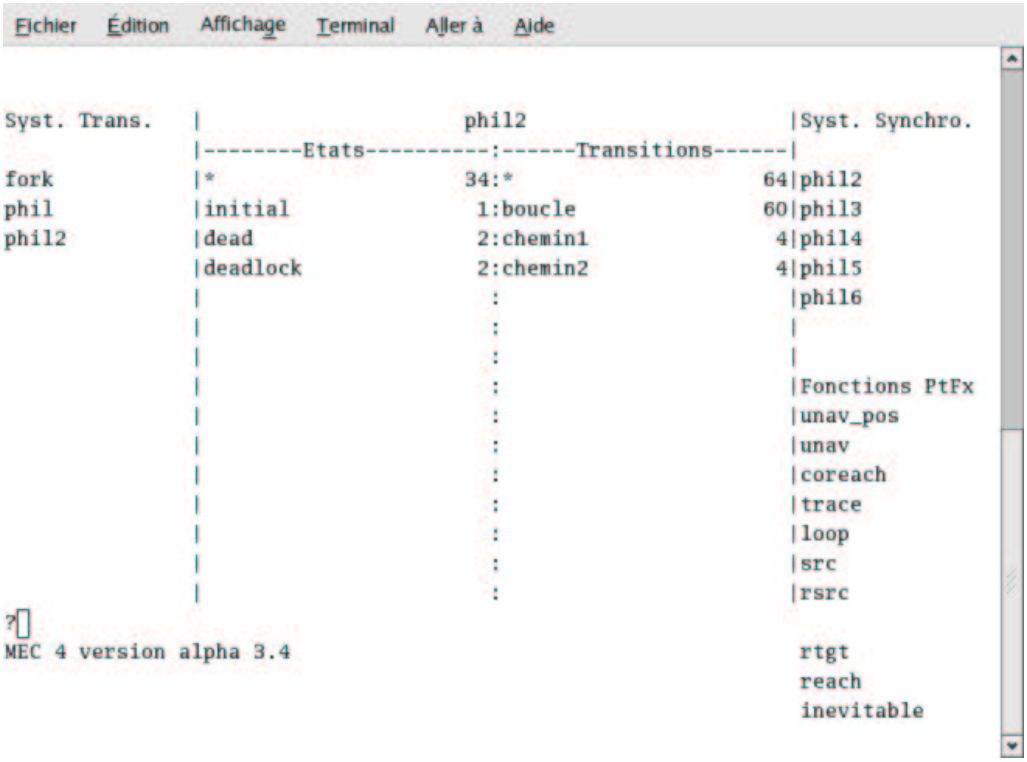


Figure 8 : *Résultats du cas Philosophes*

```
MEC 4 version alpha 3.4Current St : phil2
//-- States :
// * : 34
//initial : 1
//dead : 2
//deadlock : 2
//-- Transitions :
// * : 64
//boucle : 60
//chemin1 : 4
//chemin2 : 4
//
```

E Circuits logiques

```
load("./Demos/Circuits_logiques/input");
quit();
```

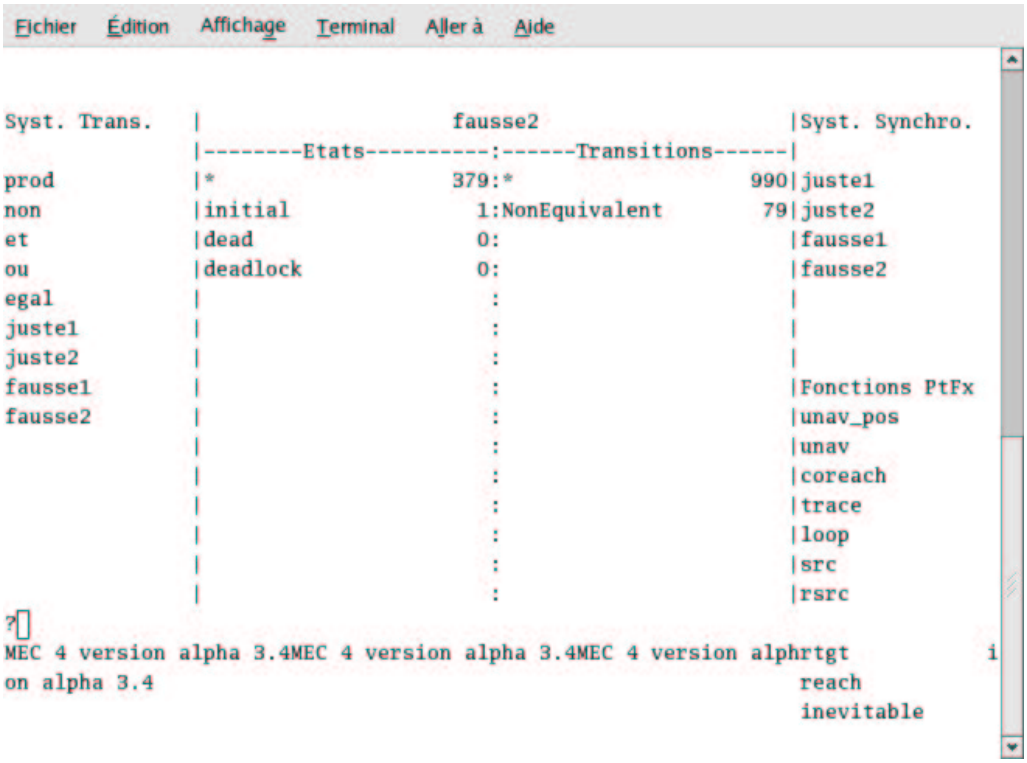


Figure 9 : *Résultats du cas Circuits logiques*

```
MEC 4 version alpha 3.4Current St : juste1
//-- States :
// * : 240
//initial : 1
//dead : 0
//deadlock : 0
//-- Transitions :
// * : 621
//NonEquivalent : 0
//
MEC 4 version alpha 3.4Current St : juste1
//-- States :
// * : 240
//initial : 1
//dead : 0
//deadlock : 0
//-- Transitions :
// * : 621
//NonEquivalent : 0
//
MEC 4 version alpha 3.4Current St : fausse1
//-- States :
// * : 379
//initial : 1
//dead : 0
//deadlock : 0
//-- Transitions :
// * : 990
//NonEquivalent : 79
//
MEC 4 version alpha 3.4Current St : fausse2
//-- States :
// * : 379
//initial : 1
//dead : 0
//deadlock : 0
```

```
//-- Transitions :
// * : 990
//NonEquivalent : 79
//
```

F Alternate bit protocol

```
load("./Demos/Alternate_bit_protocol/input");
quit();
```

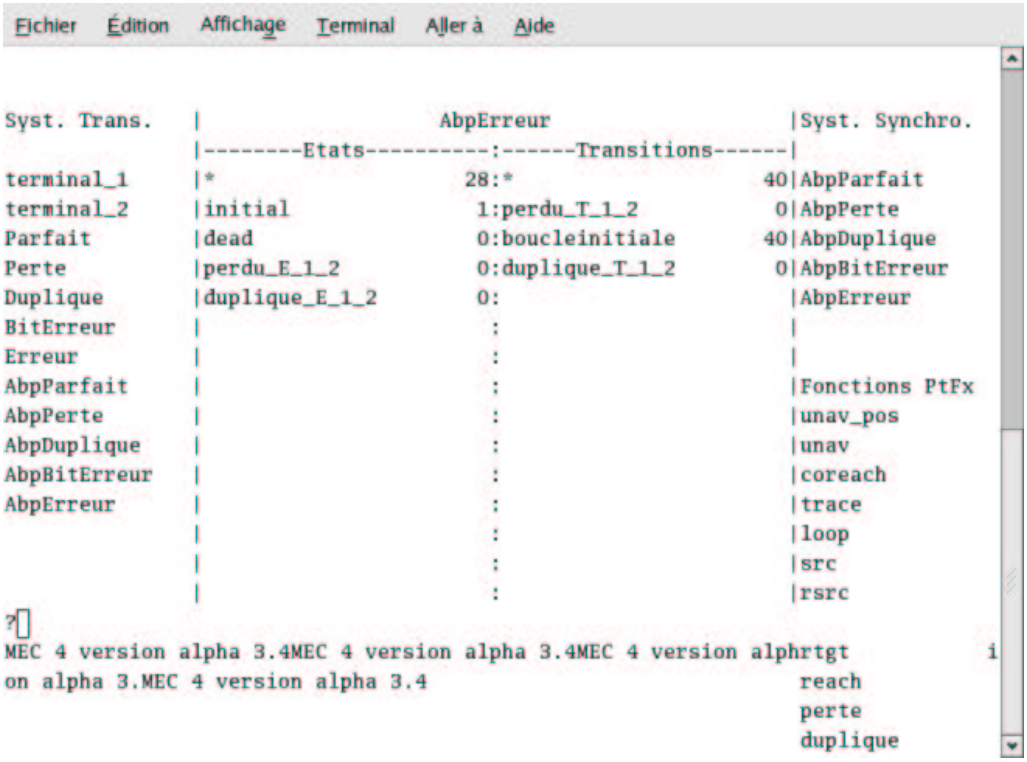


Figure 10 : *Résultats du cas Alternate bit protocol*

```
MEC 4 version alpha 3.4Current St : AbpBitErreur
//-- States :
// * : 32
//initial : 1
//dead : 0
//perdu_E_1_2 : 4
//duplique_E_1_2 : 4
//-- Transitions :
// * : 48
//perdu_T_1_2 : 28
//boucleinitiale : 48
//duplique_T_1_2 : 28
//
MEC 4 version alpha 3.4Current St : AbpDuplique
//-- States :
// * : 64
//initial : 1
//dead : 8
//perdu_E_1_2 : 0
//duplique_E_1_2 : 0
//-- Transitions :
// * : 96
//perdu_T_1_2 : 0
//boucleinitiale : 8
//duplique_T_1_2 : 0
```

```
//
MEC 4 version alpha 3.4Current St : AbpErreur
//-- States :
// * : 28
//initial : 1
//dead : 0
//perdu_E_1_2 : 0
//duplique_E_1_2 : 0
//-- Transitions :
// * : 40
//perdu_T_1_2 : 0
//boucleinitiale : 40
//duplique_T_1_2 : 0
//
MEC 4 version alpha 3.4Current St : AbpParfait
//-- States :
// * : 8
//initial : 1
//dead : 0
//perdu_E_1_2 : 0
//duplique_E_1_2 : 0
//-- Transitions :
// * : 8
//perdu_T_1_2 : 0
//boucleinitiale : 8
//duplique_T_1_2 : 0
//
MEC 4 version alpha 3.4Current St : AbpPerte
//-- States :
// * : 12
//initial : 1
//dead : 4
//perdu_E_1_2 : 0
//duplique_E_1_2 : 0
//-- Transitions :
// * : 12
//perdu_T_1_2 : 0
//boucleinitiale : 8
//duplique_T_1_2 : 0
//
```