Model Checking parallèle et réparti

Christophe PAJAULT

Laboratoire CEDRIC

Conservatoire National des Arts et Métiers, Paris

Séminaire MeFoSyLoMa – 02 décembre 2005

Introduction

Motivations

- Combattre le problème de l'explosion combinatoire.
- Vérifier des modèles de taille importante.
- Idée : dans un environnement à mémoire répartie, on utilise les ressources mémoires (et CPU) de chacun des nœuds du réseau pour stocker (et explorer) le graphe des marquages accessibles.

Présentation

- Etat de l'art sur la vérification parallèle et répartie.
- Premiers résultats.

Principe

Exploration de l'espace d'état (séquentiel)

2 possibilités de parcours :

- Parcours en profondeur (DFS)
- Parcours en largeur (BFS)

Idée

- L'exploration de l'espace d'état s'effectue en parallèle sur chacun des nœuds du réseau.
- Chaque nœud est responsable (ie. stocke) d'un sous-ensemble du graphe des marquages.
  - ➠ notion de partition de l'espace d'état.

I. Exploration de l'espace d'état.

Exploration

Algorithme

Lorsqu'un nœud explore un nouvel état, il vérifie s'il est responsable de cet état :

- si oui, l'exploration continue normalement,
- si non, l'état doit être envoyé au nœud dont il dépend, l'exploration continue en considérant que l'état a été traité.

➠ Chaque nœud (ou sous-ensemble de nœuds) doit connaître la répartition des états sur le réseau.

Fonction de partition

Objectifs

1. Répartir les états de façon uniforme sur chacun des nœuds du réseau.
2. Limiter le nombre d'échanges de messages.

Méthode

- Définition d'une fonction de partition statique connue de chacun des nœuds.

Partitionnement Uniforme

But

- Partitionner l'espace d'état en partitions de tailles égales.

Principe

- Simple : utiliser une fonction de hachage classique de la forme :

$$f : S \rightarrow \{0/1,\ldots,N\text{-}1\}$$

avec N = nombre de nœuds du réseau.

Évaluations

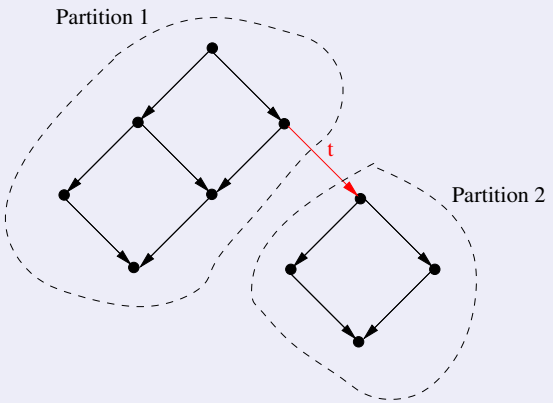
Partitionnement uniforme : évaluation sur un exemple

N	n1	n2	n3	n4	n5	messages
2	100%					0%
3	47%	53%				16%
4	32%	37%	31%			36%
5	13%	32%	34%	21%		24%
6	20%	24%	21%	17%	18%	40%

messages : pourcentage de transitions générant un échange de message.
nX : pourcentage d'états stockés par le nœud X.

Partitionnement Uniforme : synthèse

- En général on aboutit à une répartition uniforme . . .
- . . . mais on observe une forte dépendance entre les partitions.
  - grand nombre de communications.
  - notion de cross-transitions.



Une *cross-transition* engendre un message pour envoyer l'état exploré suite à son franchissement.
Le nombre de *cross-transitions* dépend directement du partitionnement.

Partitionnement "Structurel"

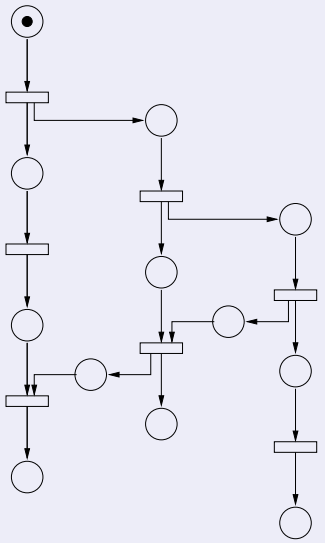
Objectif

- Limiter le nombre de communications ➠ limiter le nombre de *cross-transitions*.

Principe

- Se baser sur la structure du modèle pour définir la fonction de partition.

Pour un réseau de Petri, on considère le marquage d'un sous-ensemble de places lors du calcul de la partition.



Partitionnement "Structurel"

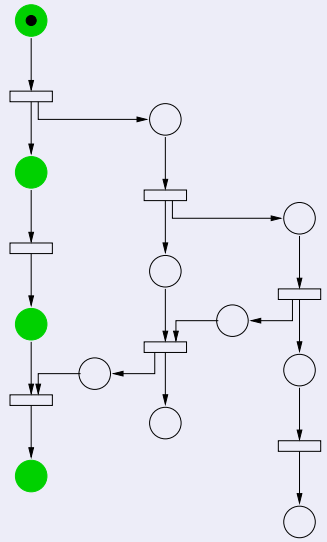
Objectif

- Limiter le nombre de communications ➠ limiter le nombre de *cross-transitions*.

Principe

- Se baser sur la structure du modèle pour définir la fonction de partition.

Pour un réseau de Petri, on considère le marquage d'un sous-ensemble de places lors du calcul de la partition.



Partitionnement "Structurel"

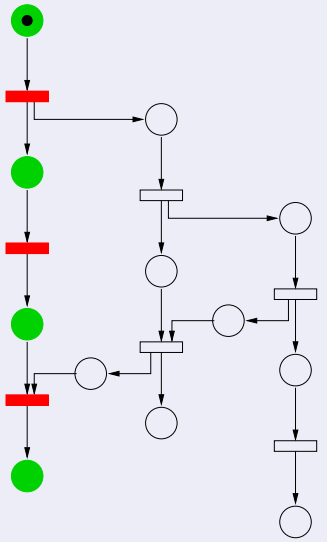
Objectif

- Limiter le nombre de communications ➠ limiter le nombre de *cross-transitions*.

Principe

- Se baser sur la structure du modèle pour définir la fonction de partition.

Pour un réseau de Petri, on considère le marquage d'un sous-ensemble de places lors du calcul de la partition.



Partitionnement "Structurel"

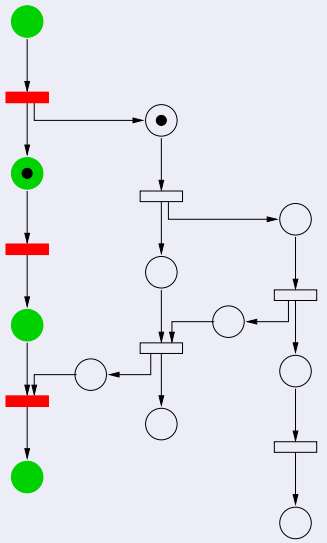
Objectif

- Limiter le nombre de communications ➠ limiter le nombre de *cross-transitions*.

Principe

- Se baser sur la structure du modèle pour définir la fonction de partition.

Pour un réseau de Petri, on considère le marquage d'un sous-ensemble de places lors du calcul de la partition.



Partitionnement "Structurel"

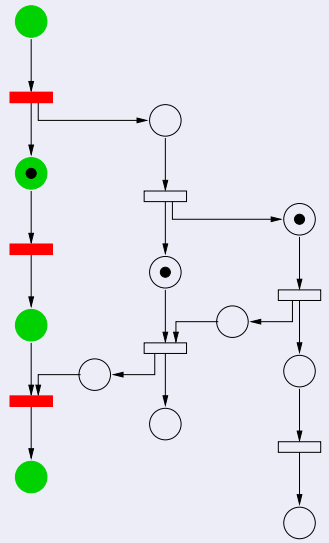
Objectif

- Limiter le nombre de communications ➠ limiter le nombre de *cross-transitions*.

Principe

- Se baser sur la structure du modèle pour définir la fonction de partition.

Pour un réseau de Petri, on considère le marquage d'un sous-ensemble de places lors du calcul de la partition.



Partitionnement "Structurel"

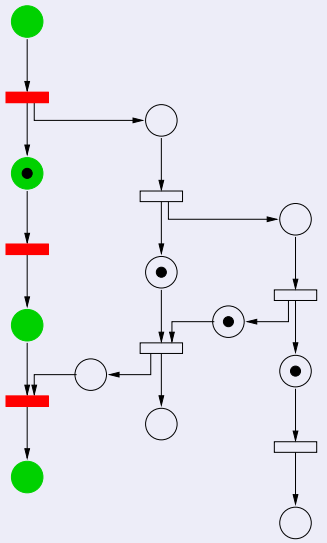
Objectif

- Limiter le nombre de communications ➠ limiter le nombre de *cross-transitions*.

Principe

- Se baser sur la structure du modèle pour définir la fonction de partition.

Pour un réseau de Petri, on considère le marquage d'un sous-ensemble de places lors du calcul de la partition.



Partitionnement "Structurel"

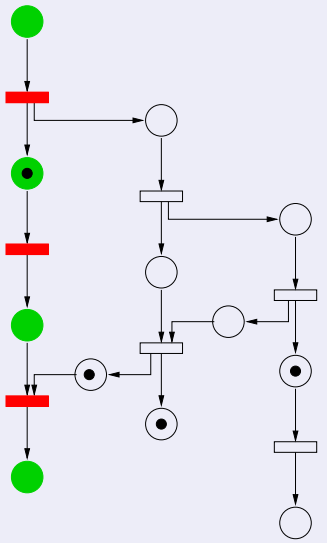
Objectif

- Limiter le nombre de communications ➠ limiter le nombre de *cross-transitions*.

Principe

- Se baser sur la structure du modèle pour définir la fonction de partition.

Pour un réseau de Petri, on considère le marquage d'un sous-ensemble de places lors du calcul de la partition.



Partitionnement "Structurel"

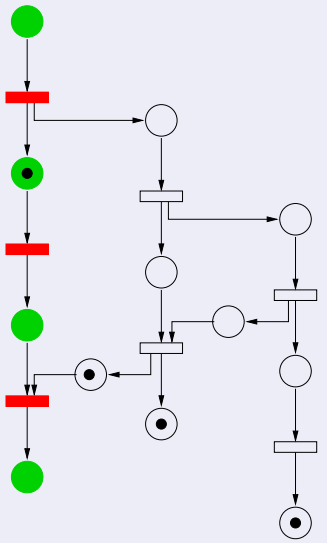
Objectif

- Limiter le nombre de communications ➠ limiter le nombre de *cross-transitions*.

Principe

- Se baser sur la structure du modèle pour définir la fonction de partition.

Pour un réseau de Petri, on considère le marquage d'un sous-ensemble de places lors du calcul de la partition.



Évaluations

Sous-ensemble A

N	n1	n2	n3	n4	messages
2	100%				0%
3	79%	21%			4%
4	59%	14%	27%		9%
5	54%	14%	22%	10%	9%

Sous-ensemble B

N	n1	n2	n3	n4	messages
2	100%				0%
3	47%	53%			10%
4	32%	37%	31%		19%
5	12%	19%	21%	48%	15%

Équilibrage de charge

Idée

Les fonctions de partition visant à limiter le nombre de *cross-transitions* sont efficaces sauf en terme d'uniformité de la répartition ➠ on veut donc réagir dynamiquement – ie, pendant l'exploration – et réorganiser le partitionnement à la manière des techniques d'équilibrage de charge.

Remarque

La répartition des états varie au cours du temps
➠ Un état peut donc être stocké sur un noeud à un instant t et sur un autre noeud à un instant $t+i$.

Classification

Définition : classe

- Le réassignement (la réorganisation) est décidée en fonction de la mémoire :
  - lorsque des problèmes de mémoire sont rencontrés sur un nœud, ou
  - lorsque les différences de charge entre les noeuds sont trop importantes.
- La granularité de réorganisation ne peut être l'état ➠ les états sont regroupés en classes : une classe ne peut appartenir qu'à une seule partition.
- Une table globale conservant le mapping *classe* $\leftrightarrow$ *noeud* est répliquée sur un sous-ensemble de nœuds.

Equilibrage de charge

- Grain de répartition plus fin.
  - ➠ possibilité d'effectuer un équilibrage de charge efficace.

- Déterminer les classes reste très complexe.

On utilise les mêmes mécanismes pour déterminer les classes que ceux utilisés pour déterminer les partitions.
Plusieurs politiques sont alors possibles pour déterminer les partitions et pour effectuer les réorganisations.

Évaluations pour l'équilibrage de charge (1)

Classes initialement réparties uniformément sur tous les sites.

RemoteAgent	**Memory** %	**Messages** %	**Time** (sec)
Global Hash Code	50/50	38	524.02
Local Hash Code	50/50	46	531.17
Local Hash Code (1)	47/53	11	216.56
Local Hash Code (2)	48/52	13	281.25
Program Counter (1)	52/48	17	340.97
Program Counter (2)	48/52	25	487.46
Program Counters (1)	52/48	17	311.22
Program Counters (1)	49/51	14	333.78

Évaluations pour l'équilibrage de charge (2)

Classes initialement placées sur un même noeud du réseau.

RemoteAgent	**Memory** %	**Messages** %	**Time** (sec)
Global Hash Code	47/53	16	251.74
Local Hash Code	47/53	13	150.09
Local Hash Code (1)	45/55	5	97.57
Local Hash Code (2)	53/47	7	125.09
Program Counter (1)	45/55	11	141.38
Program Counter (2)	44/56	11	134.14
Program Counters (1)	39/61	10	145.38
Program Counters (1)	48/52	14	182.00

Évaluations pour l'équilibrage de charge (3)

Classes initialement placées sur chaque noeud du réseau.

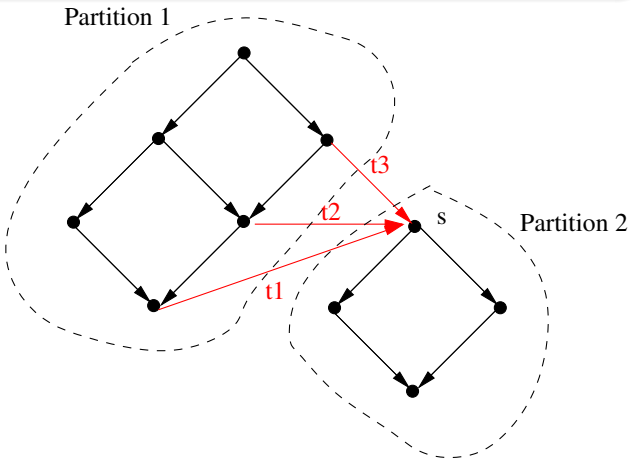
RemoteAgent	**Memory** %	**Messages** %	**Time** (sec)
Global Hash Code	50/50	9	251.17
Local Hash Code	48/52	7	120.92
Local Hash Code (1)	52/48	3	74.46
Local Hash Code (2)	48/52	2	80.65
Program Counter (1)	38/62	4	113.21
Program Counter (2)	49/51	4	122.66
Program Counters (1)	48/52	5	112.46
Program Counters (1)	52/48	3	113.84

Le *State-Caching*

Definition

- But : limiter les envois d'états redondants.
- Principe : les états envoyés sont stockés dans un cache. A chaque envoi d'état, on vérifie qu'il n'est pas déjà dans le cache.

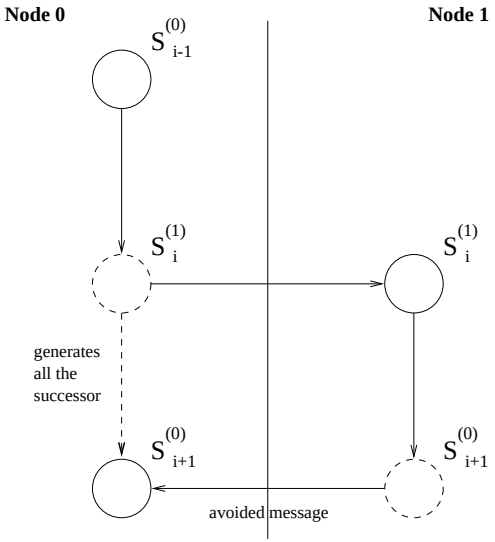
Soit un parcours dans lequel on traite $t1$ puis $t2$ et enfin $t3$. Lors du traitement de $t1$, on stocke s dans le cache, lors du traitement de $t2$ et $t3$, si s est toujours dans le cache, on ne le réexpédie pas.



Le *children look-ahead*

Definition

- But : limiter le nombre de messages échangés lors de l'exploration.
- Principe : lors de l'envoi d'un état, on vérifie que les successeurs directs n'appartiennent pas au noeud expéditeur.



II. Vérification de propriétés.

Propriétés d'accessibilité

Vérification

- simple : vérification à chaque état.

Rapport d'erreur

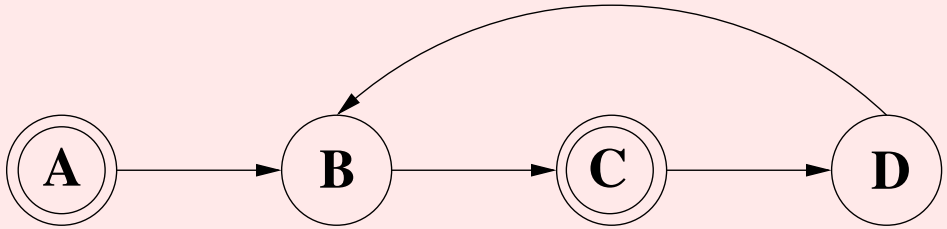
- Problème : il n'est pas possible de dépiler (comme en séquentiel).
- ➠ Lorsqu'un état est envoyé à un nœud, on envoie également la séquence d'actions aboutissant à ce nœud.

Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

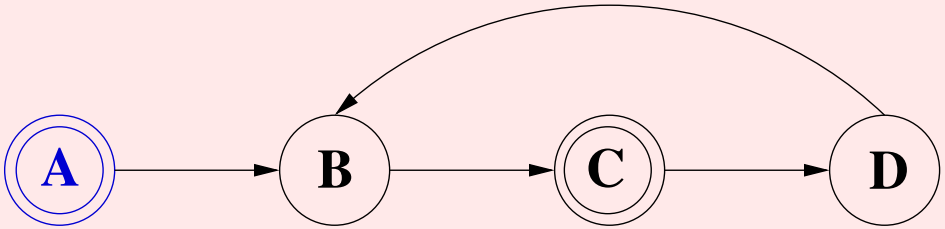


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

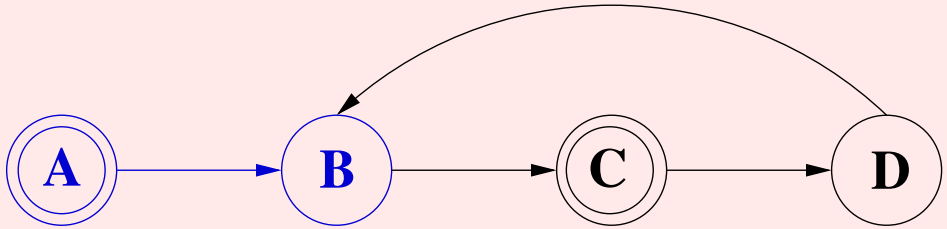


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

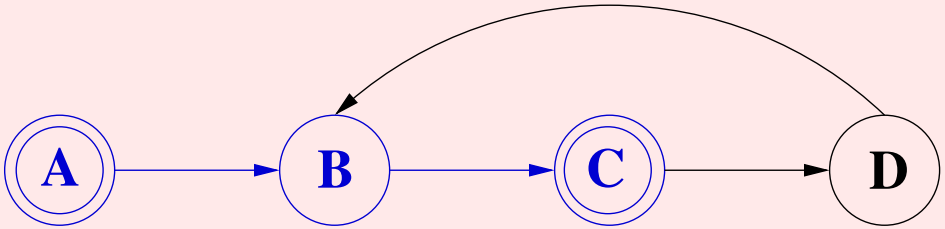


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

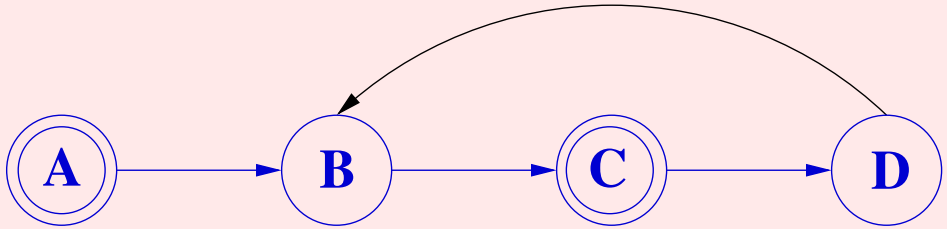


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

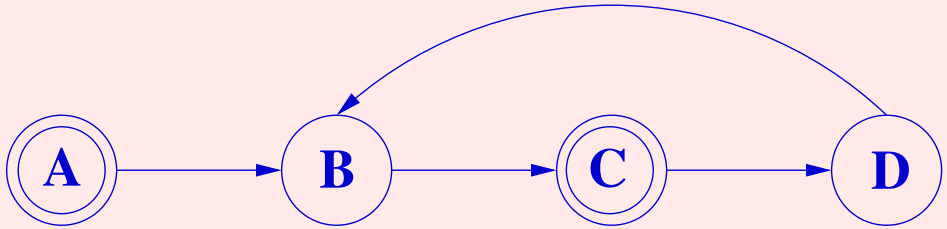


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

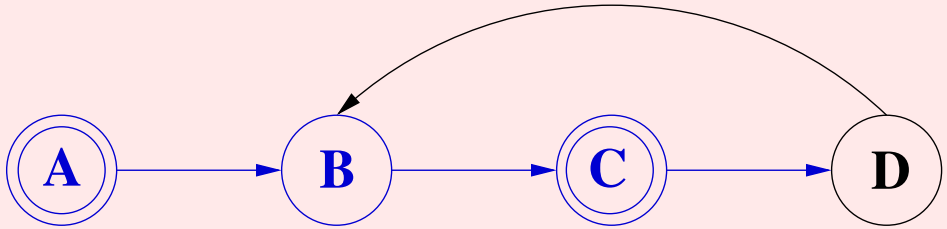


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

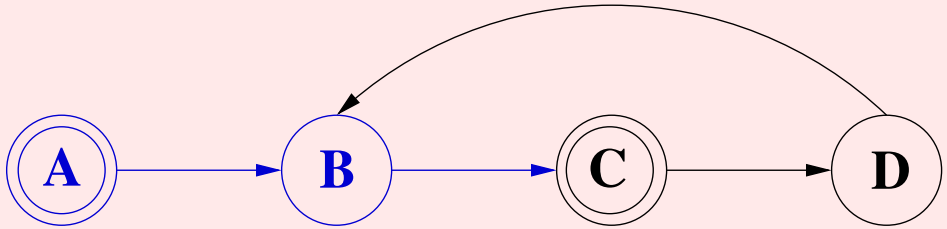


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

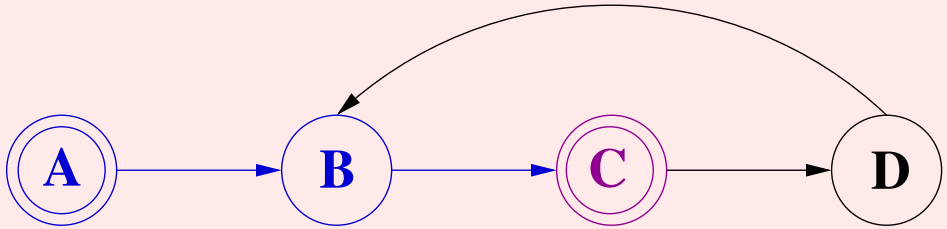


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

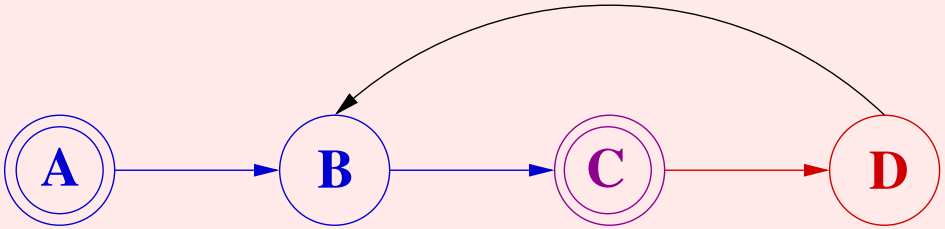


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

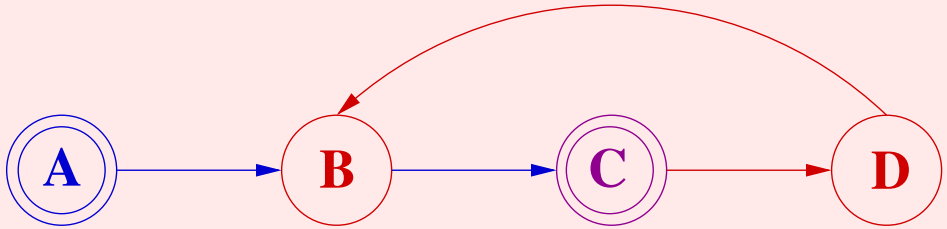


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

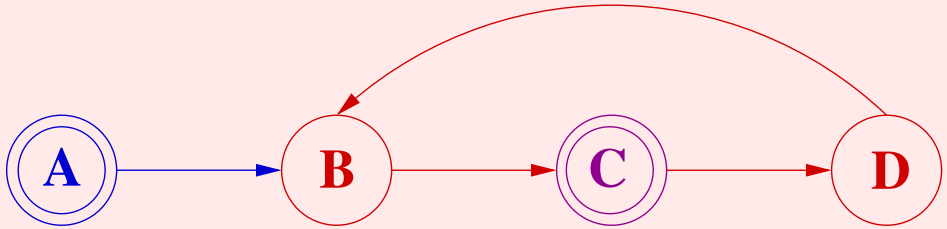


Propriétés LTL

Principe

La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !

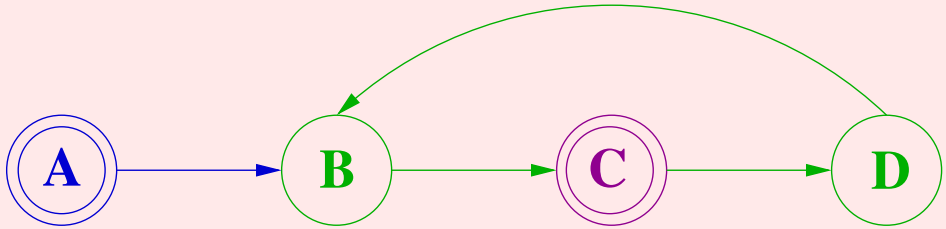


Propriétés LTL

Principe

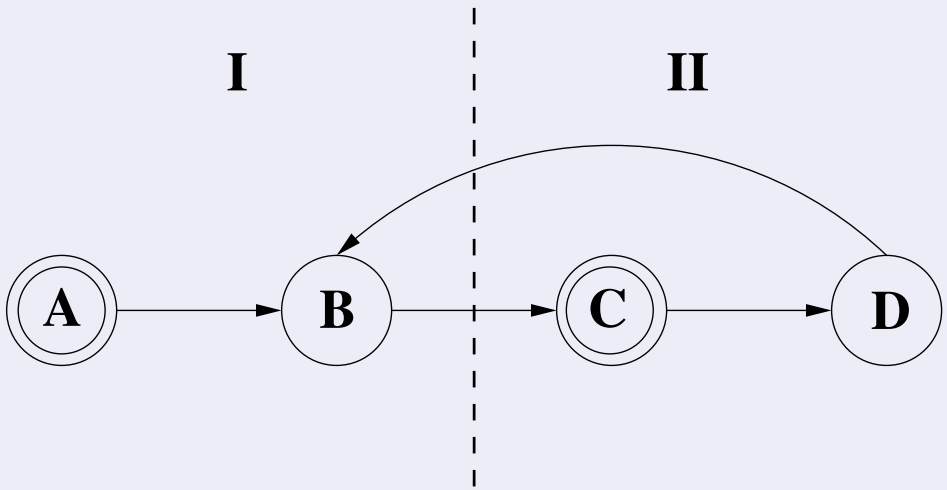
La vérification de propriétés LTL peut être réduite au un problème de détection de cycle autour d'un état acceptant.

L'ordre est important !



Comportement incorrect (1)

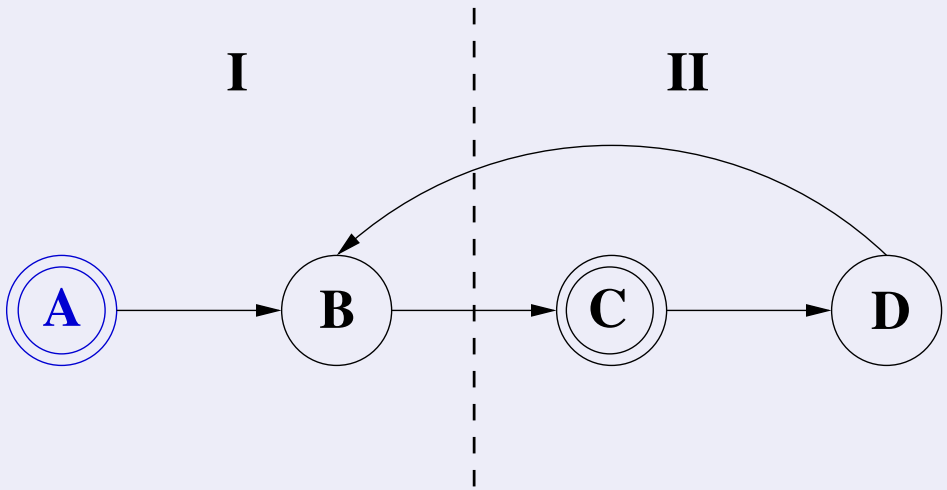
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

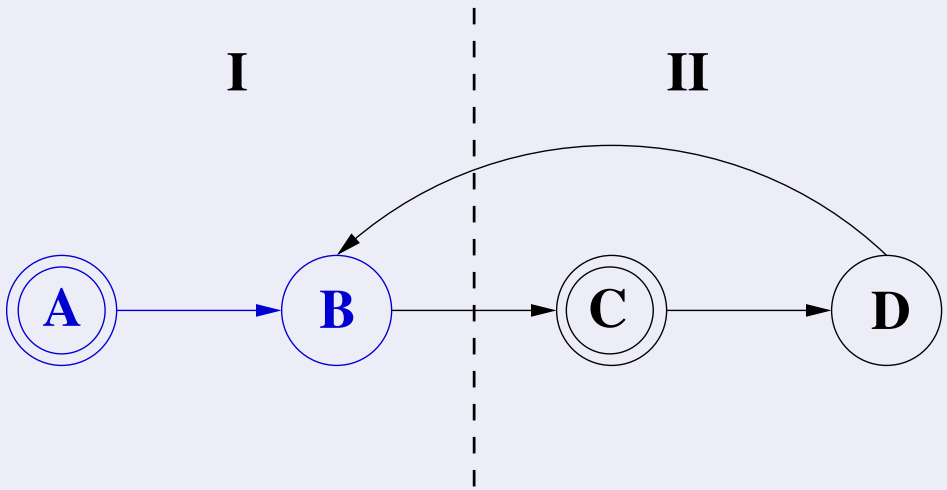
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

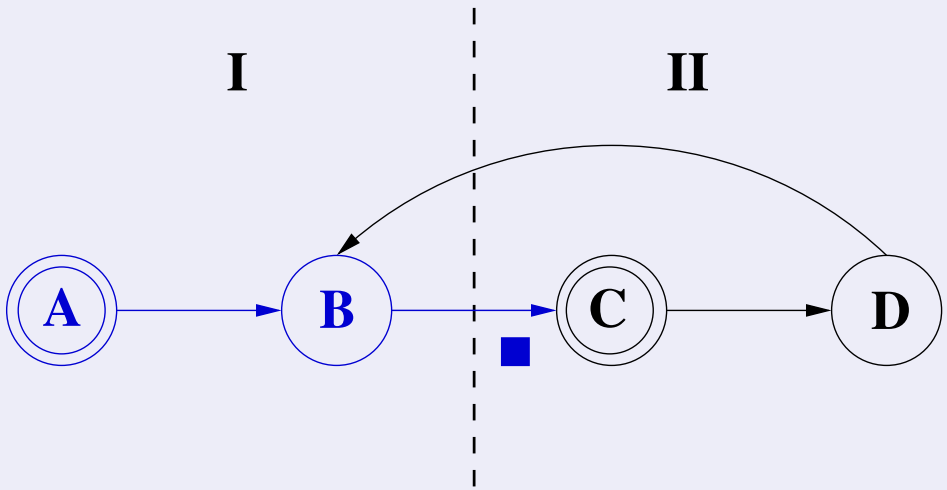
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

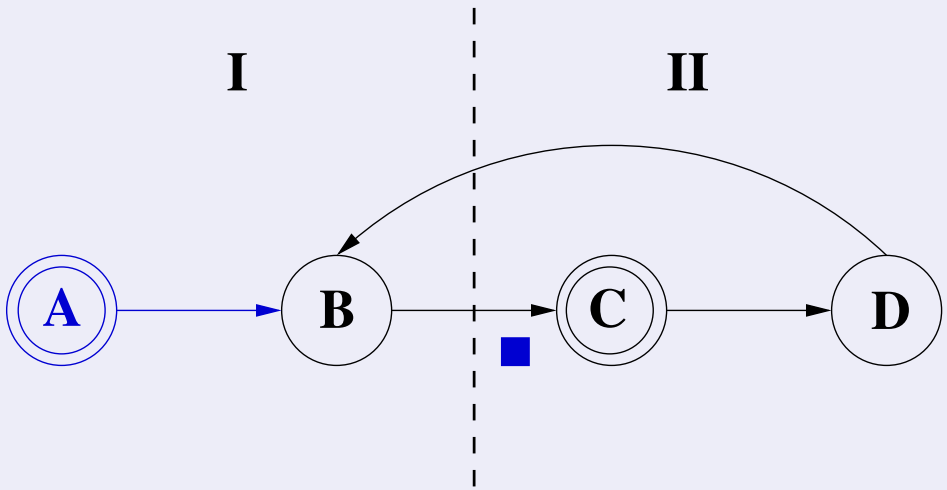
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

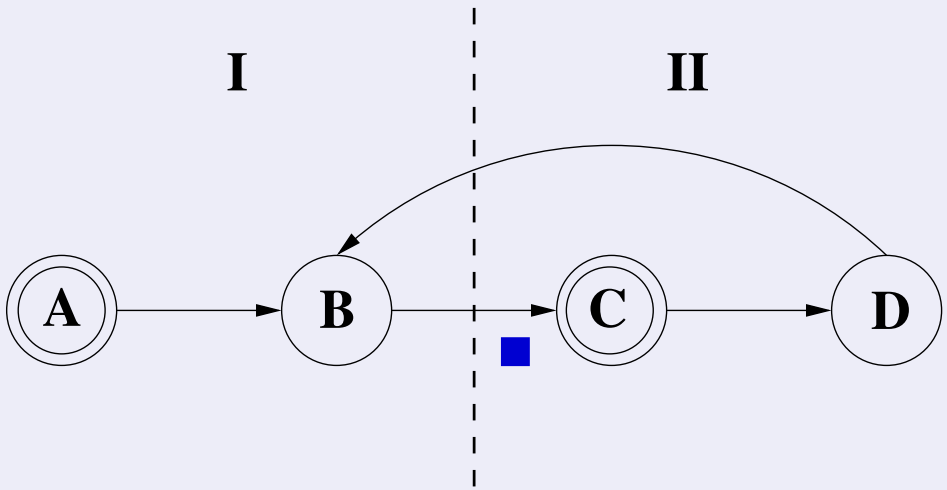
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

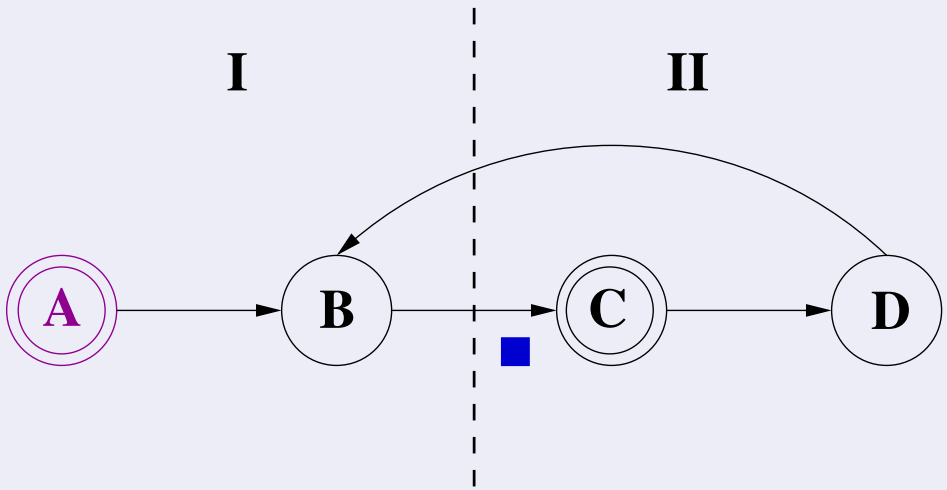
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

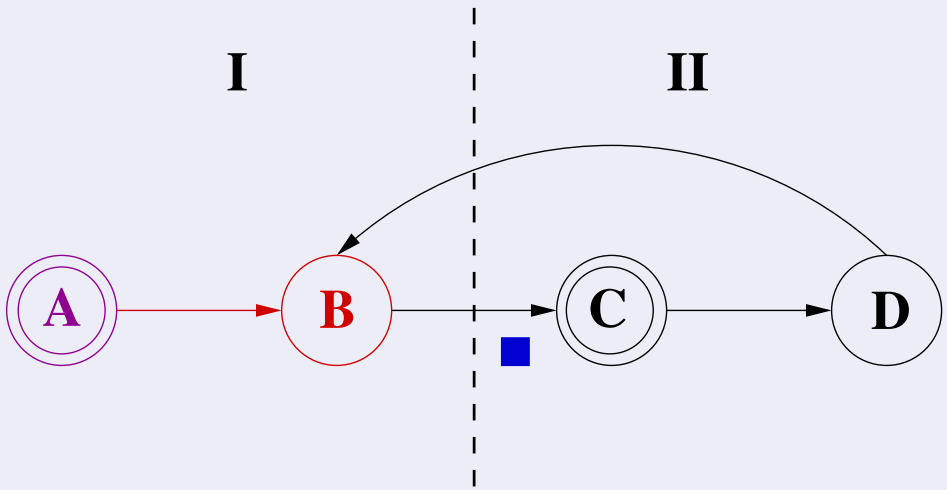
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

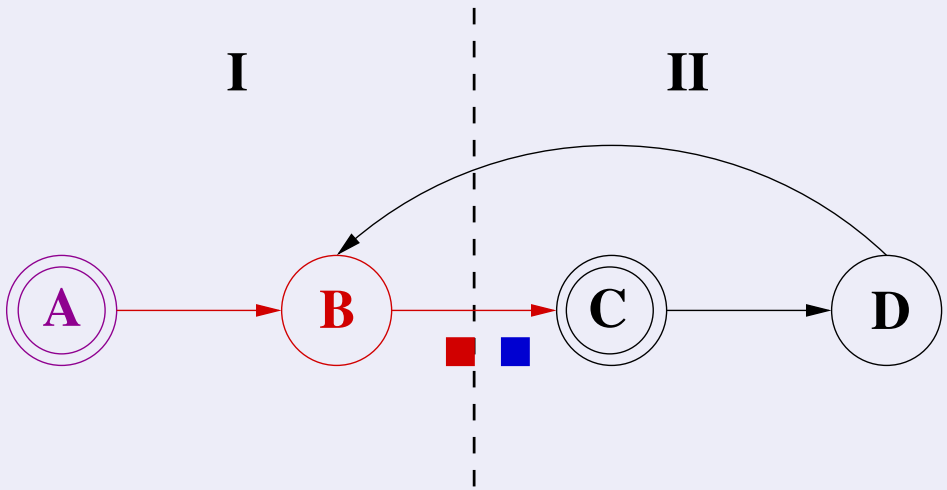
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

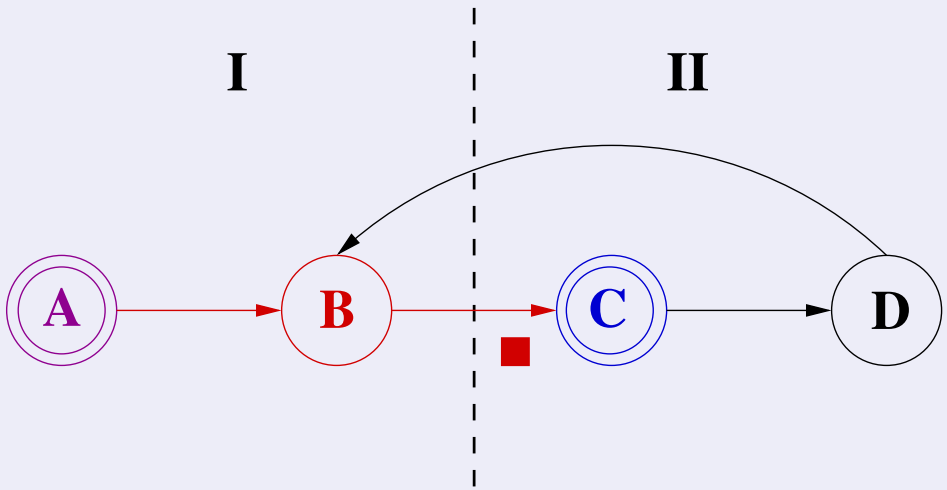
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

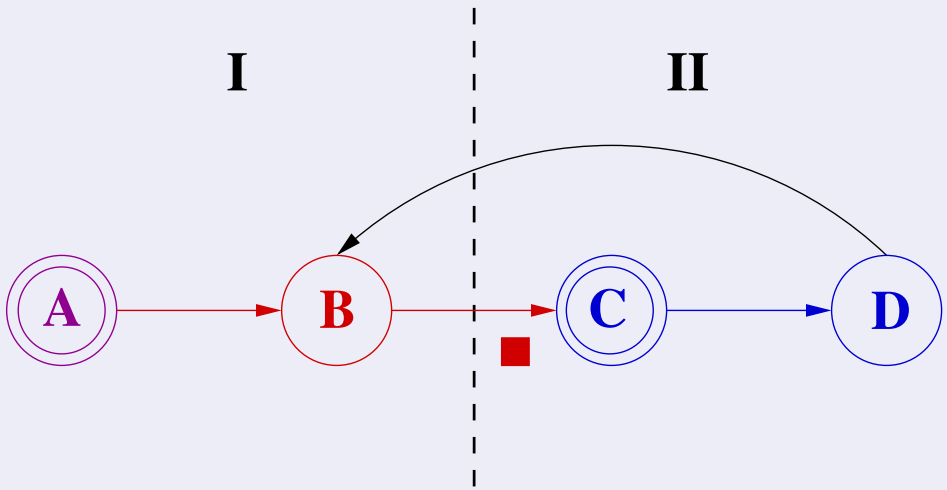
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

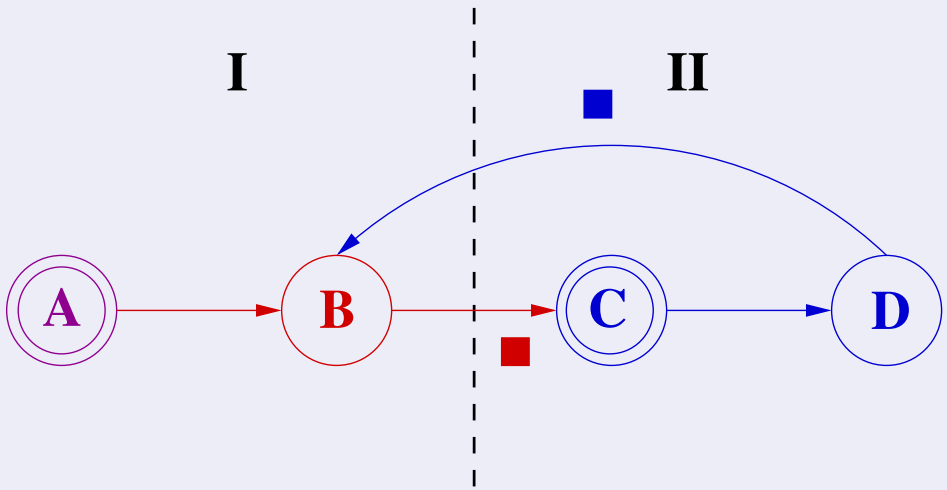
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

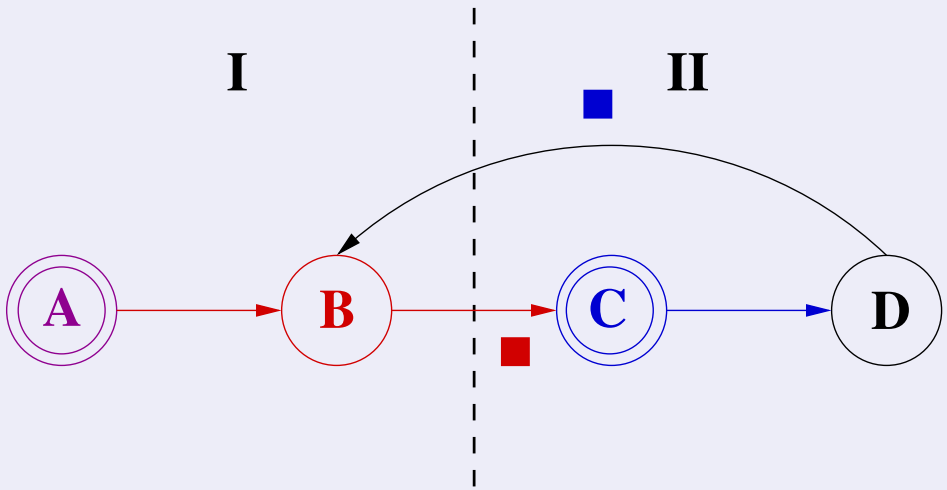
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

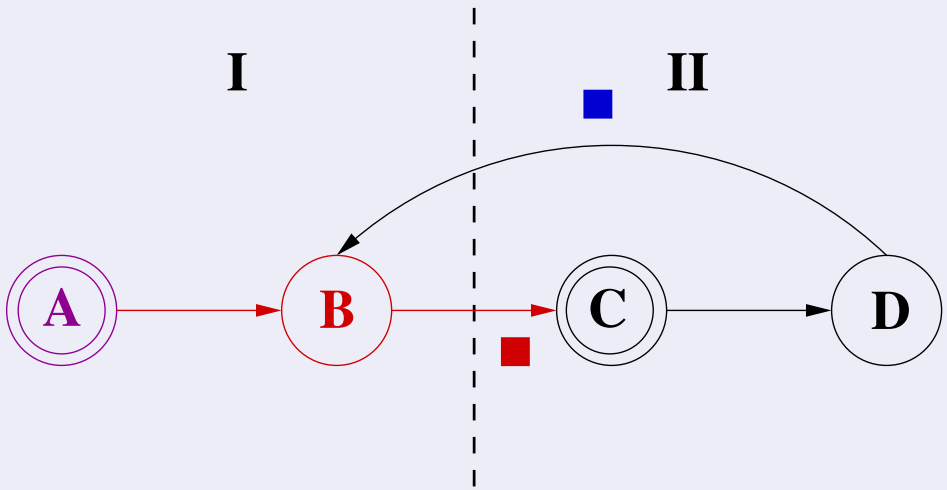
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

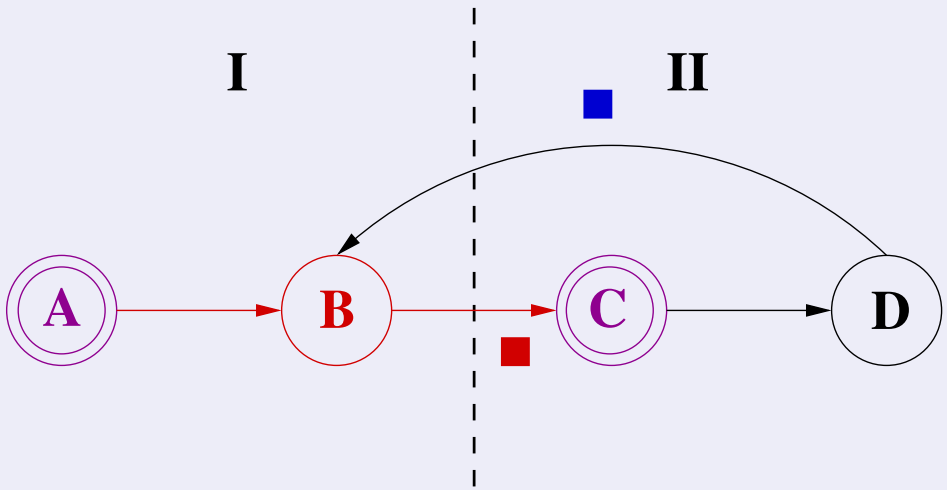
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

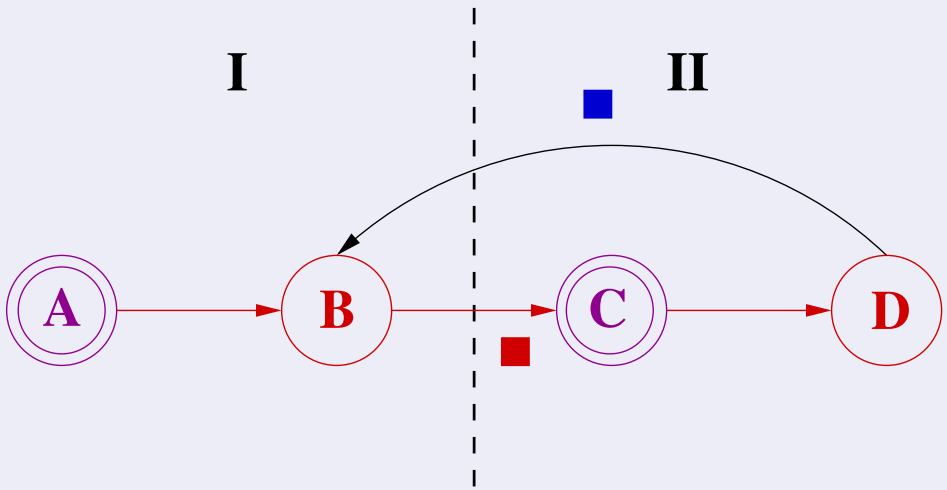
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

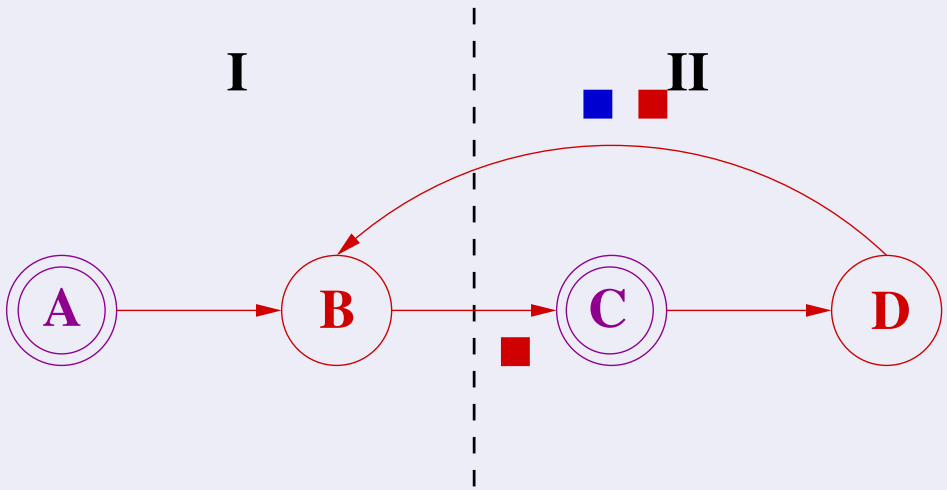
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

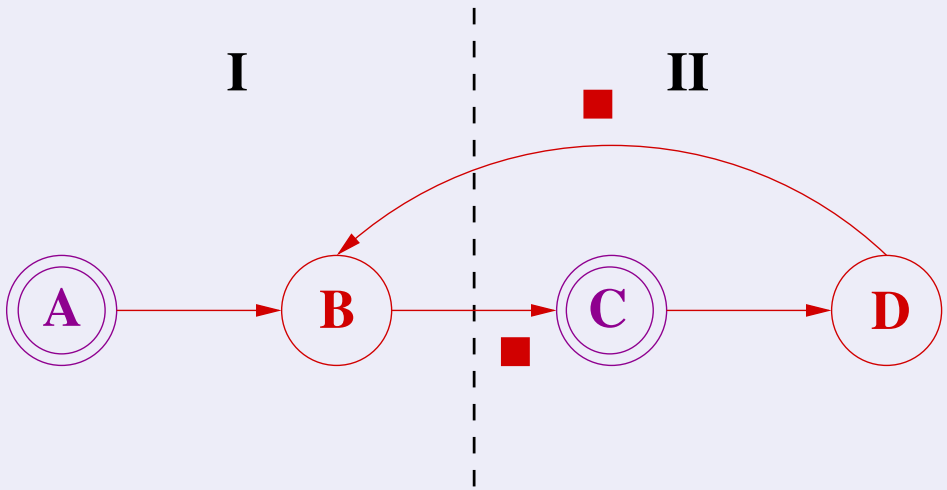
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

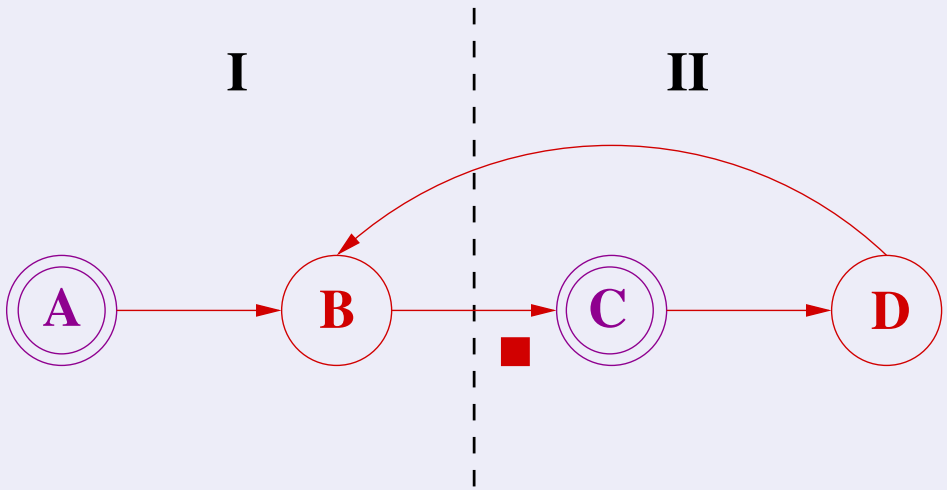
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

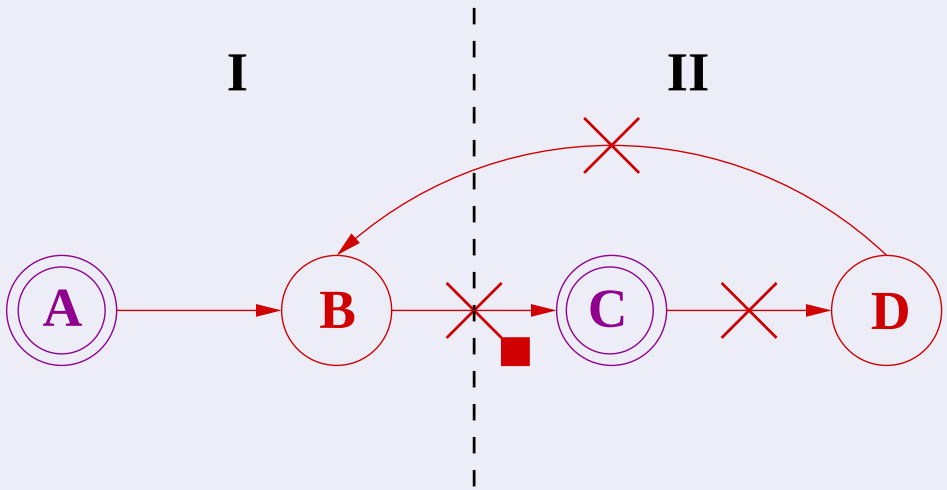
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (1)

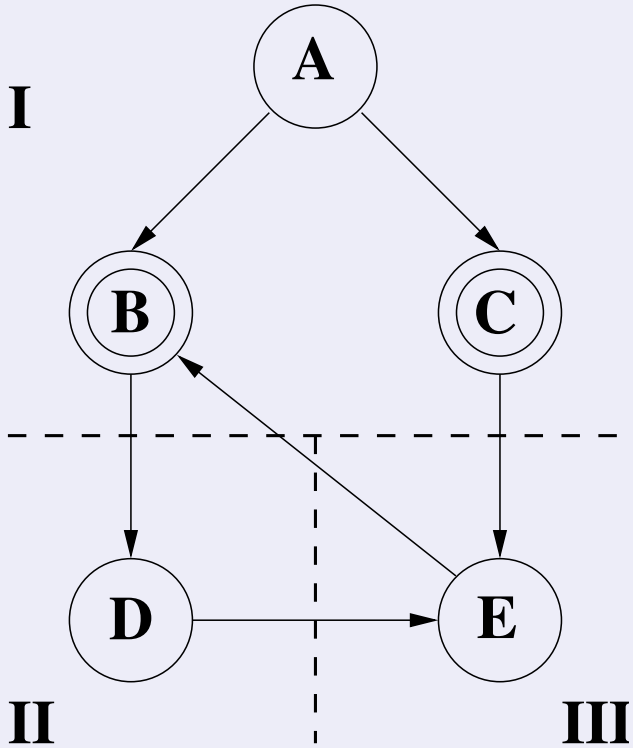
2 détections de cycle executées sans ordre



Si 2 procédures de détection de cycle sont lancées parallèlement sans débuter dans l'ordre on peut aboutir à des erreurs !

Comportement incorrect (2)

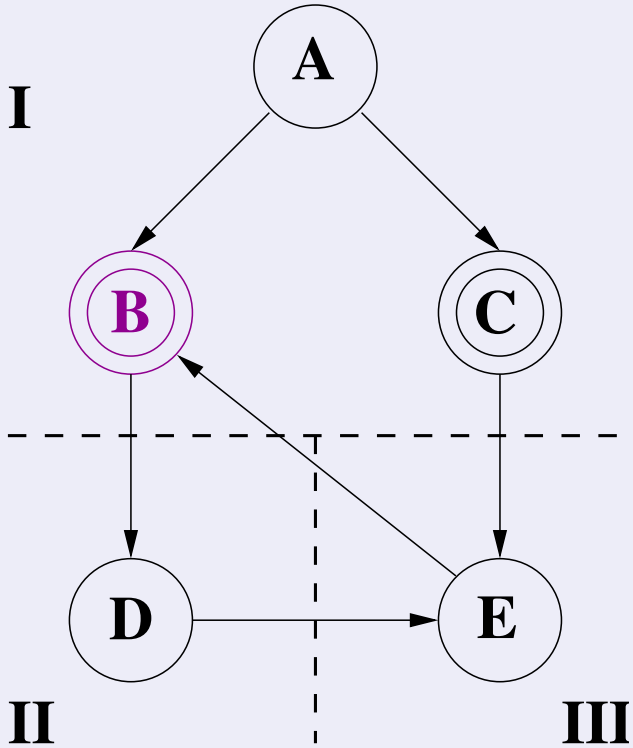
2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Comportement incorrect (2)

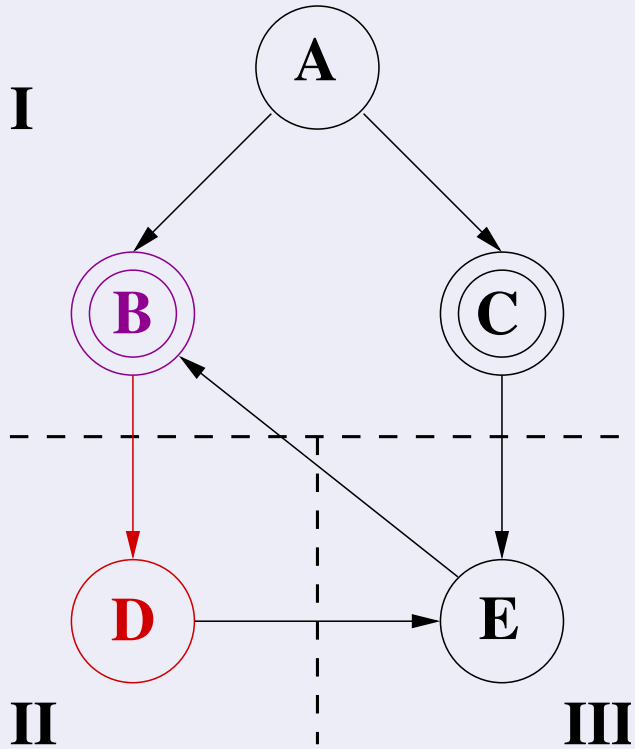
2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Comportement incorrect (2)

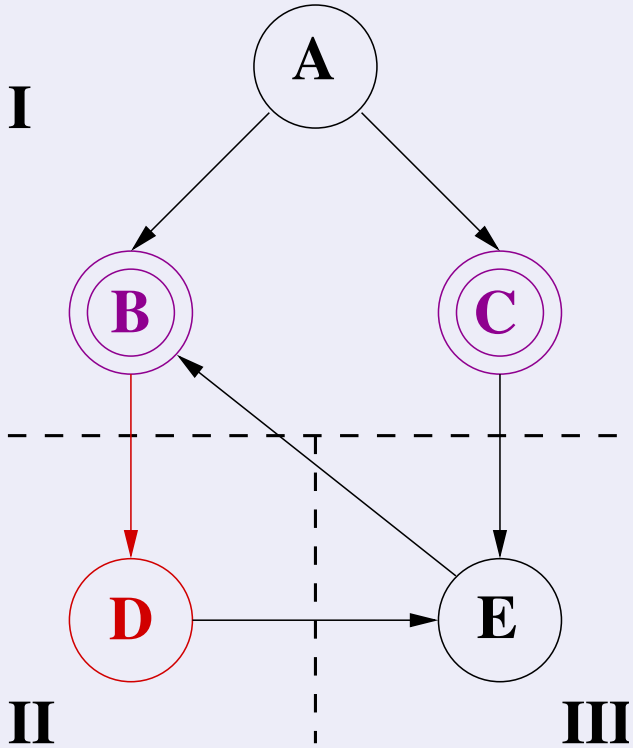
2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Comportement incorrect (2)

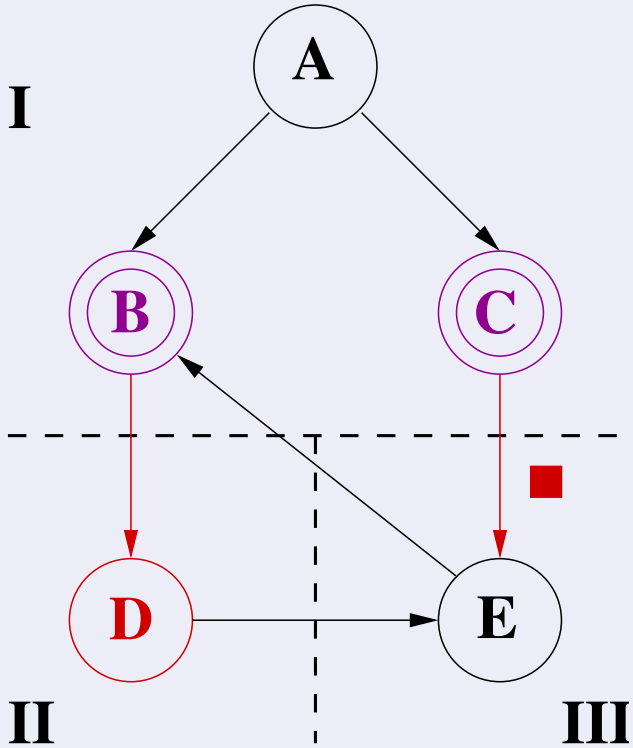
2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Comportement incorrect (2)

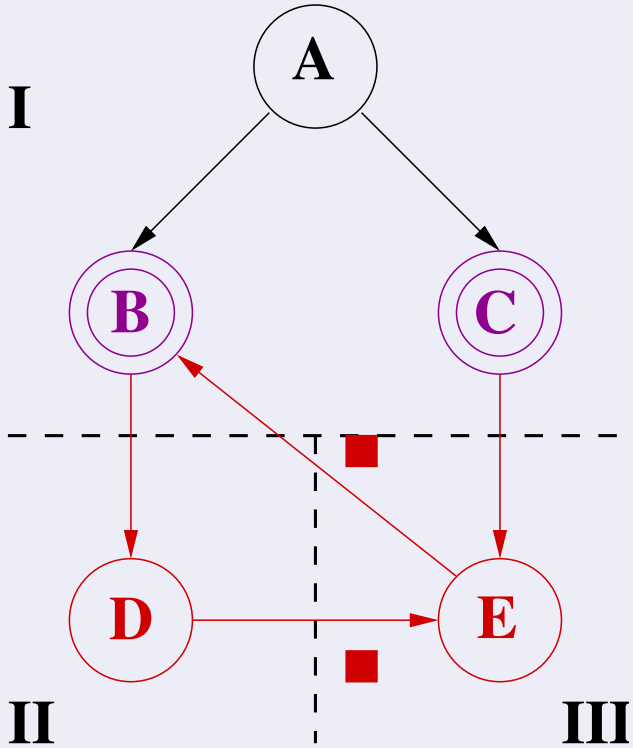
2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Comportement incorrect (2)

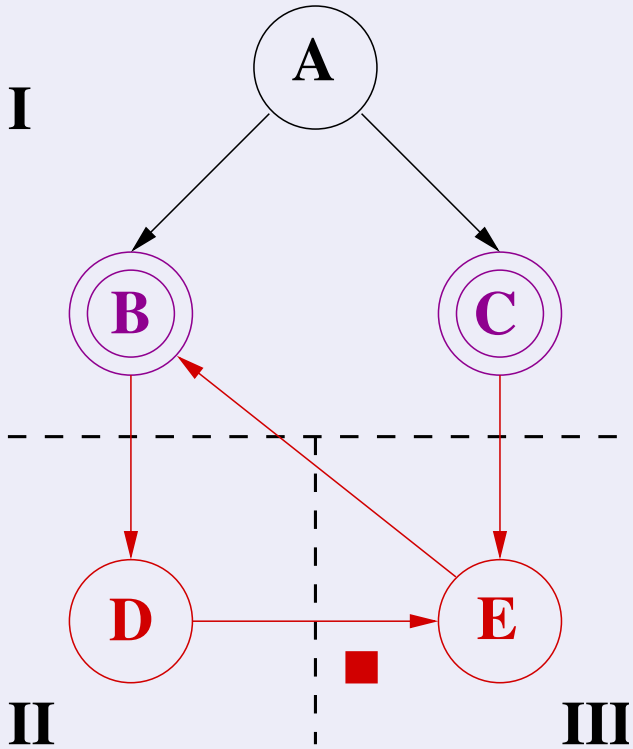
2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Comportement incorrect (2)

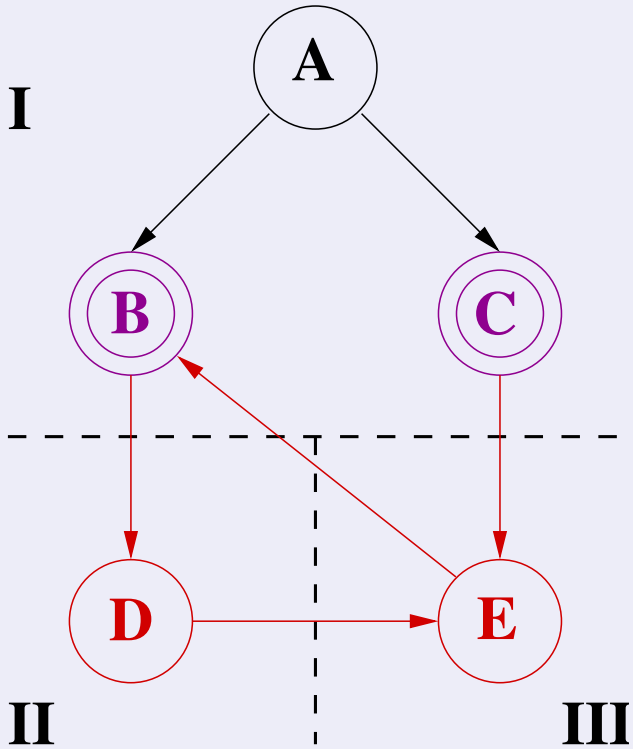
2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Comportement incorrect (2)

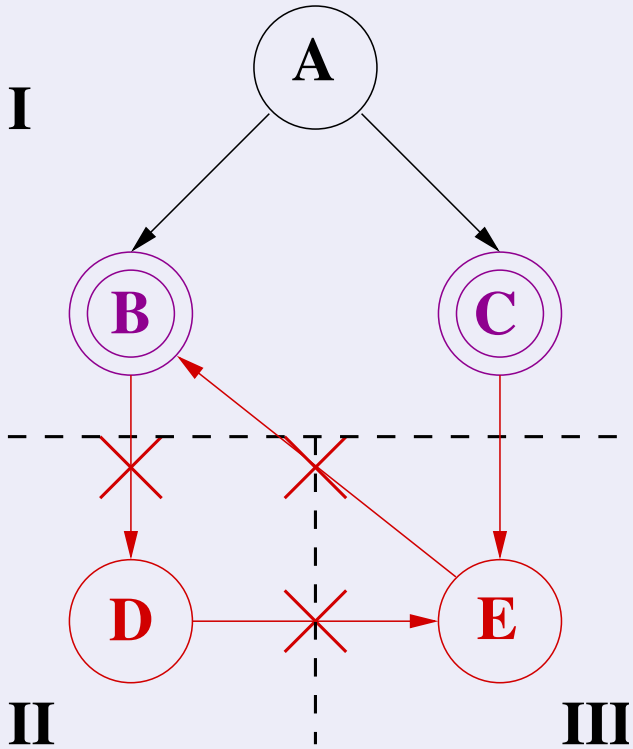
2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Comportement incorrect (2)

2 détections de cycle en parallèle



Lancer les procédures de détection de cycle dans l'ordre ne suffit pas !

Solutions

3 approches

1. Lancer les procédures de détection de cycle dans l'ordre et séquentiellement.
   - utilisation d'une structure de donnée pour conserver les relations d'ordre entre les états.
2. Permettre qu'un état soit visité par plusieurs procédures de détection de cycle (mais une seule fois par une même procédure de détection).
   - possibilité de lancer les procédures en parallèle.
3. Répartir les états de façon à ce que les cycles soient locaux.
   - l'ordre est maintenu et les détections peuvent s'effectuer en parallèle.

Conclusion

- Le partitionnement est un problème complexe :
  - trouver des sous-ensembles du graphe des marquages les plus indépendants possibles.
  - choix des places est complexe.
- Travail effectué :
  - étude de différentes approches pour le model-checking parallèle et réparti,
  - implémentation et évaluations de différentes techniques de partitionnement de l'espace d'état.
- Ce qui reste à faire :
  - equilibrage de charge, réductions d'ordre partiel, . . .
  - vérification de propriétés LTL

Quelques références I

O. Grumberg, T. Heyman and A. Schuster

A Work-Efficient Distributed Algorithm for Reachability Analysis

CAV, 2003

F. Lerda and R. Sisto

Distributed-Memory Model Checking with SPIN

Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, 1999

U. Stern and D.L. Dill

Parallelizing the Murphi Verifier

Proceedings of the 9th International Conference on Computer Aided Verification, 1997

H. Garavel, R. Mateescu and I. Smarandache

Parallel state space construction for model-checking

SPIN '01 : Proceedings of the 8th international SPIN workshop on Model checking of software

Quelques références II

R. Kumar and E.G. Mercer.

Load Balancing Parallel Explicit State Model Checking

Electronic Notes in Theoretical Computer Science, 2005

F. Lerda and W. Visser

Addressing dynamic issues of program model checking

SPIN '01 : Proceedings of the 8th international SPIN workshop on Model checking of software, 2001

J. Barnat

Distributed Memory LTL Model Checking

L.M. Kristensen and L. Petrucci

An Approach to Distributed State Space Exploration for Coloured Petri Nets

Proc. 25th Int. Conf. Application and Theory of Petri Nets (ICATPN'2004), 2004

. . .