# Interblocage

INF3173 – Principes des systèmes d'exploitation
Hiver 2024

Francis Giraldeau
giraldeau.francis@uqam.ca

Université du Québec à Montréal

# Plan

- Conditions nécessaires
- Graph de ressources
- Détection
- Évitement
- Prévention
- Méthodes de test
- Méthodes formelles
- Stratégies utilisées sous Linux

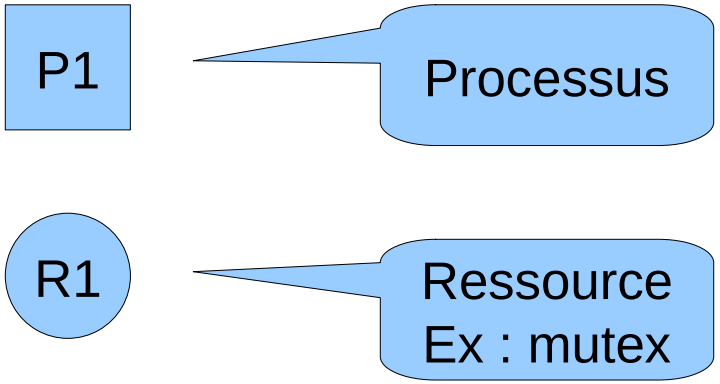
# Conditions nécessaires

1) Condition d'exclusion mutuelle
   - Une ressource est libre ou détenue par un processus.
2) Acquisition partielle suivie d'attente un verrou et un autre verrou
   – Un processus détenant des ressources peut en demander d'autres et bloquer parce qu'elles sont déjà occupées.
3) Les ressources détenues ne peuvent être volées
   – Seul le processus qui les détient peut les libérer.
4) Il doit y avoir une attente circulaire
   – Foo attend Bar qui attend Foo

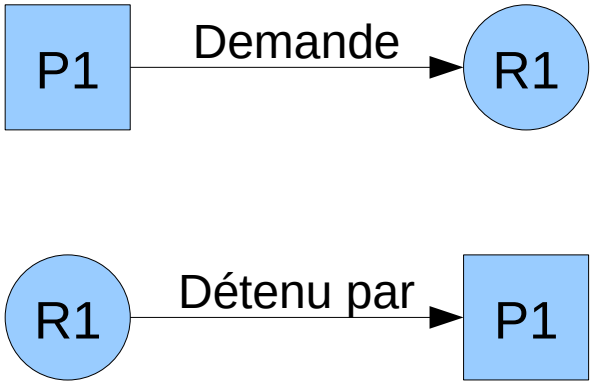
# Graph de ressource

Type : Graph dirigé

Noeuds




Relations

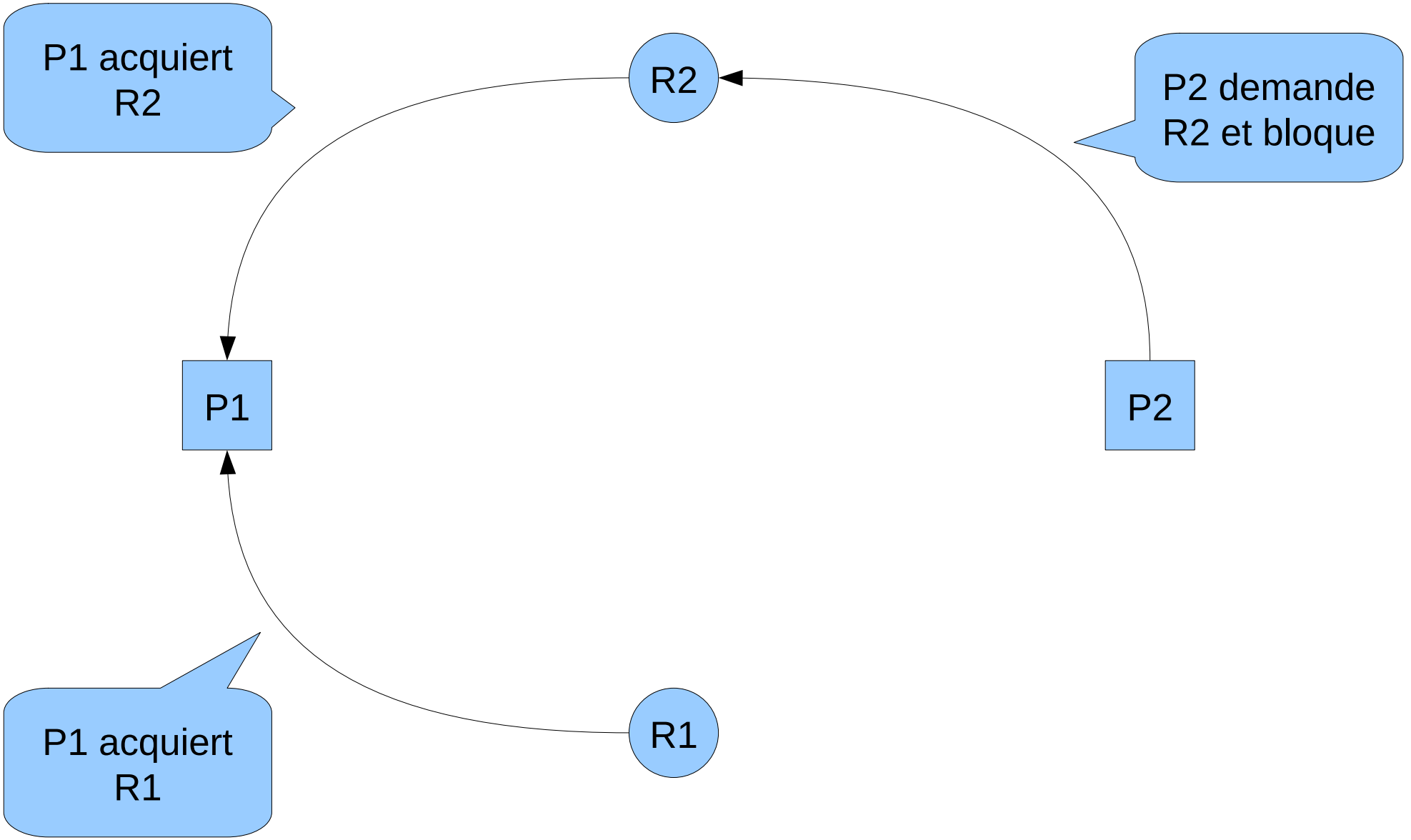


# Programme d'exemple

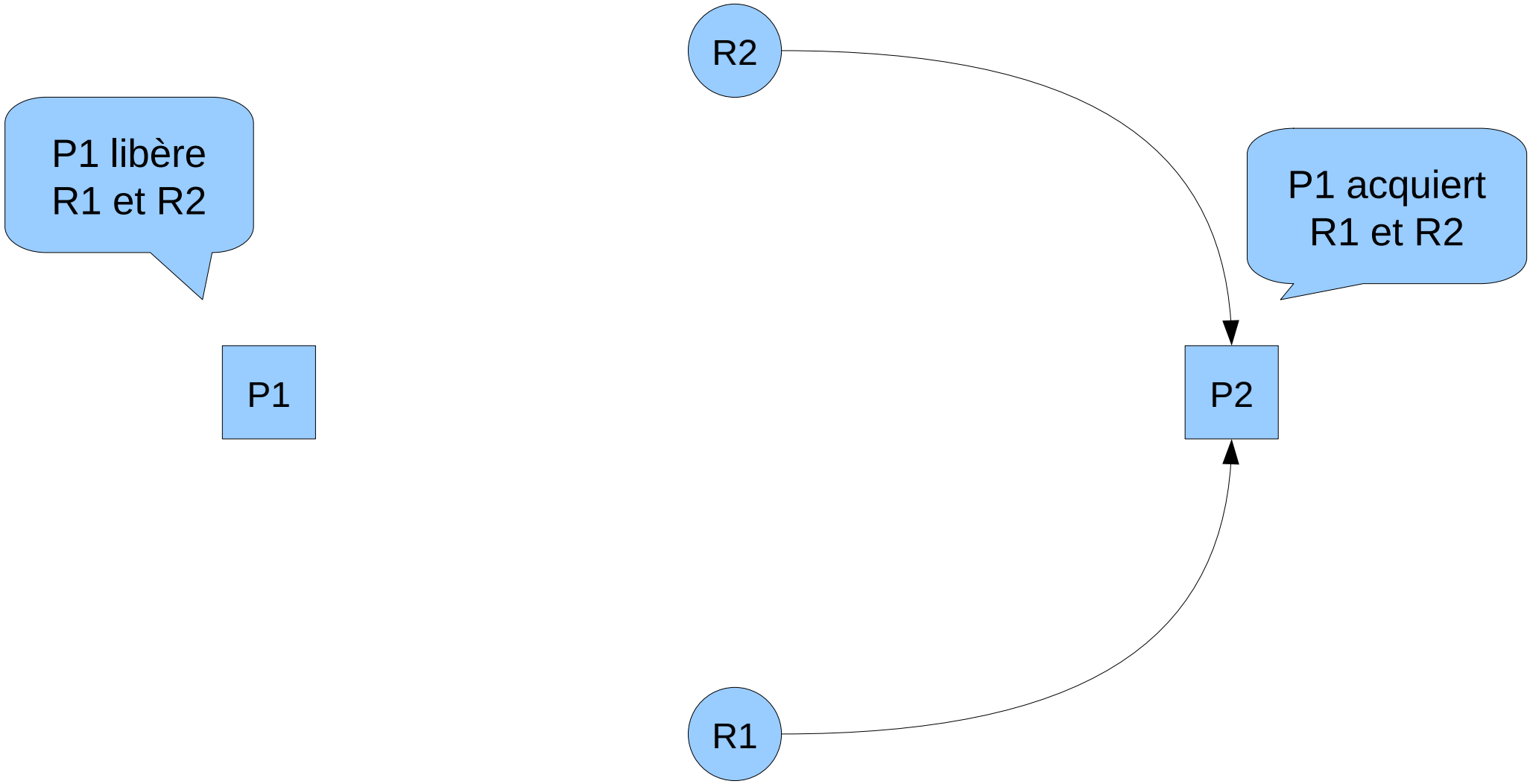
```
mutex_t r1, r2;

void thread_p1(void *arg) {
  mutex_lock(&r1)
  mutex_lock(&r2)
  /* do something */
  mutex_unlock(&r2)
  mutex_unlock(&r1)
}
```

```
void thread_p2(void *arg) {
  mutex_lock(&r2)
  mutex_lock(&r1)
  /* do something */
  mutex_unlock(&r1)
  mutex_unlock(&r2)
}
```

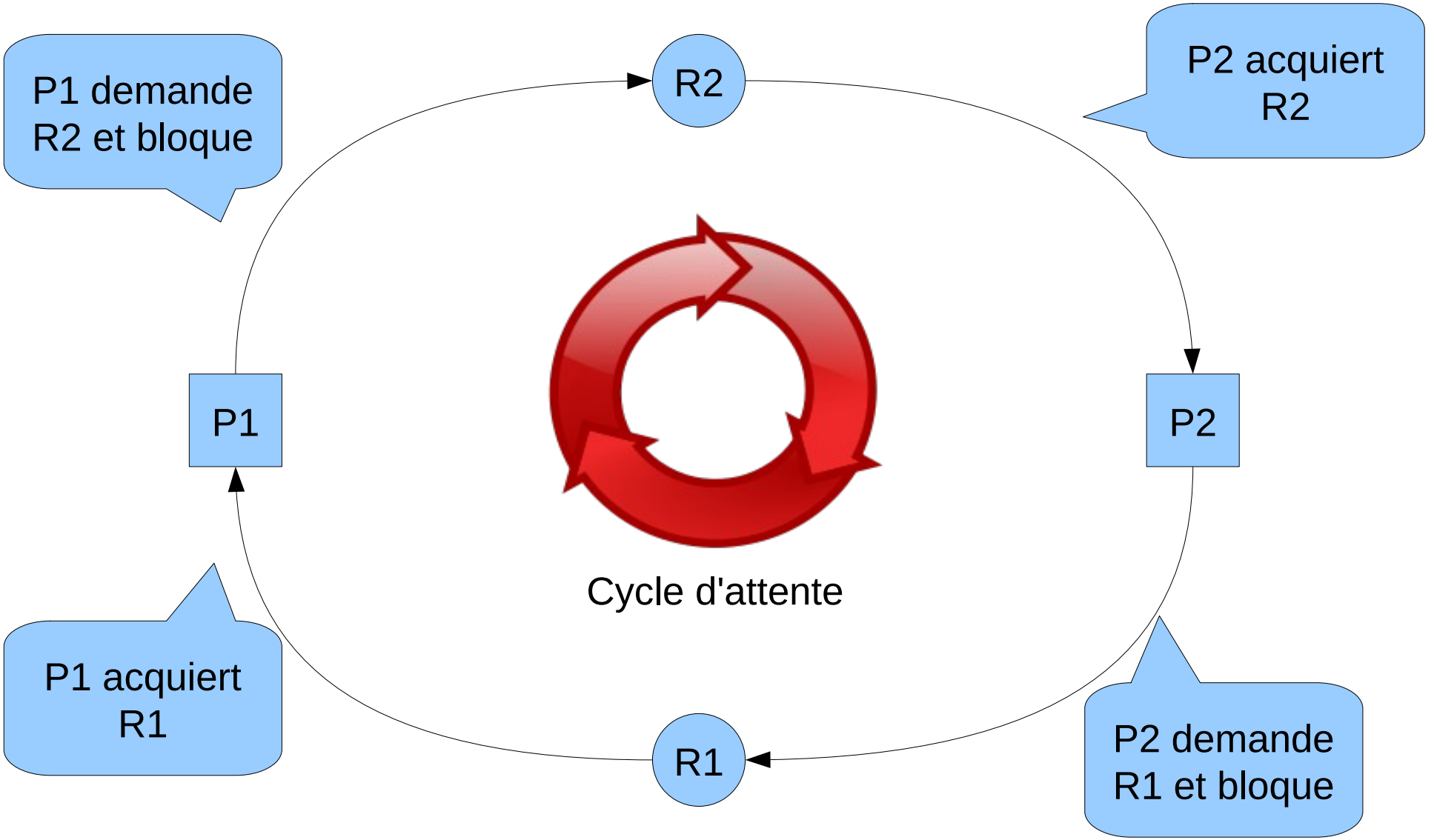
# Exécution sans interblocage (1)



# Exécution sans interblocage (2)



# Exécution menant à un interblocage



interblocage actif

# Exemple : ressources limitées

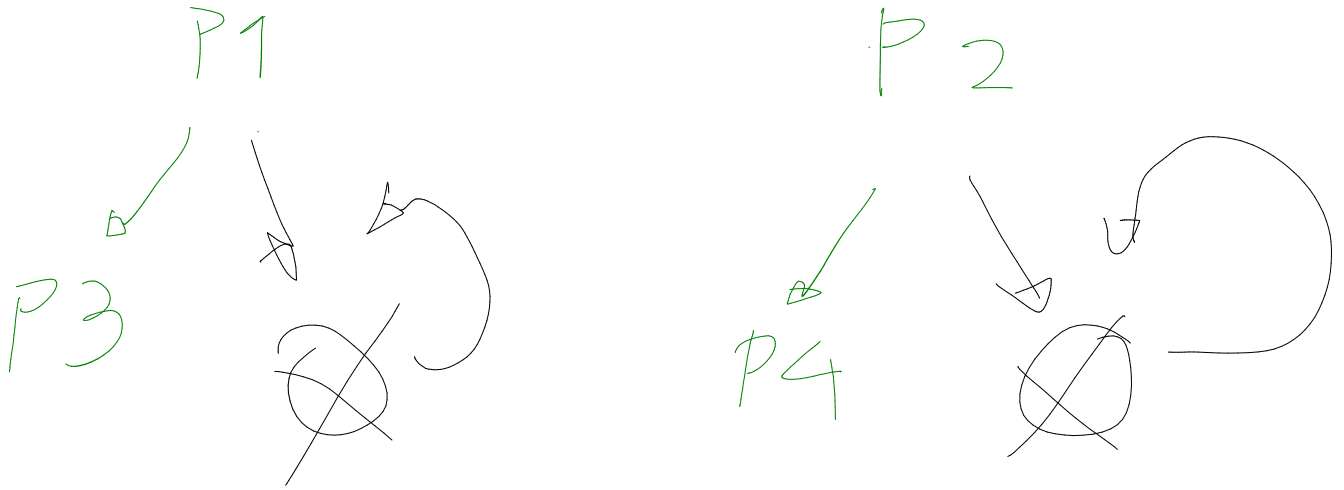
- Soit un système avec 2 processus.
- Le système permet au maximum 4 processus.
- Chaque processus doit créer 2 enfants pour se terminer correctement. Si un fork() échoue, alors le processus parent réessaye.
- Déterminer une s'équence d'événements (création, terminaison) qui produit un succès, et un autre qui produit une boucle infinie.
- Algorithme du banquier $$$

cas1


2 → 4 → 1 → 3 → 0

---

cas2



# Stratégies

- Ignorer le problème
  - … et il surviendra pendant une démo
  - Inacceptable pour des systèmes critiques : centrale nucléaire, navette spatiale, instrument médical, etc.
- Détecter les interblocages et réagir
  - Solution de dernier recours
- Rendre les interblocages impossible
  - Solution idéale

# Détection (1)

- Créer le graphe de ressources.
- Détecter un cycle dans le graphe.
  - Parcourir le graphe et construire une liste des noeuds visités. Si en parcourant le graphe, on rencontre un noeud déjà dans la liste, alors il existe un cycle.
- Le cycle indique quelles sont les ressources impliquées dans l'interblocage.

# Détection (2)

- Utilisation des interruptions non masquables périodiques
  - Survient même après avoir désactivé les interruptions
- Dans le gestionnaire, on vérifie qu'il s'est produit un changement d'état (comme un événement d'ordonnancement)
- Si rien ne s'est passé pendant plusieurs secondes, alors il est probable que le système soit en interblocage
- Action : redémarrage forcé
- La panne n'est pas évitée, mais réduit sa durée

# Briser un interblocage

- Retour en arrière (rollback)
  - Par exemple avec pthread_mutex_trylock()
  - Libérer tous les verrous acquis et réessayer depuis le début
- Terminer un processus dans le cycle
  - Construire le graph de ressources, parcourir le graph et détecter un cycle
  - N'est pas utilisé en pratique à ma connaissance

# Prévention des interblocages

- Empêcher l'attente indéfinie
  - Réserver une ressource à la fois, si une ressource n'est pas disponible, alors libérer les ressources et réessayer plus tard
  - Sujet à la famine et à l'interblocage actif « livelock »
  - Un processus peut se réessayer en boucle sans jamais réussir
  - Solution : se mettre dans une file d’attente
- Évite de créer un cycle
  - Toujours demander les ressources dans le même ordre empêche un cycle de se former

# Vérification interblocages (1)

- Test: utiliser un test de charge en concurrence, laisser rouler longtemps, espérer que si une possibilité d'interblocage existe, elle se manifeste.
- Si un interblocage survient, alors il existe une possibilité d'interblocage
- Si aucun interblocage ne survient, alors il se peut qu'il y ait une possibilité d'interblocage, mais que le test n'ait pas exécuté une séquence menant à une interblocage.
- Un test ne permet pas de prouver l'inexistence d'un interblocage, mais permet d'augmenter la confiance qu'il n'y en ait pas.

# Exemple de test de charge

```
/* test list implementation to look for deadlock */
void test_thread(void *arg) {
    int i;
    list_t *list = (list_t *) arg;
    for (i=0; i<1000000; i++) {
        list_append(list, i);
        list_remove(list, i);
    }
}

int main(int argc, char **argv) {
    int i;
    /* demarrer 1000 fils simultanes */
    list_t *list = list_new();
    for (i=0; i<1000; i++) {
        pthread_create(t[i], NULL, test_thread, list);
    }
    /* ... */
}
```

Test de charge

Exécution simultanée de plusieurs fils

# Vérification interblocages (2)

- Vérification formelle: explorer toutes les possibilités d'exécution et déterminer si un chemin conduit à un interblocage.
- Si un chemin mène à l’interblocage, alors il est certain qu’un interblocage peut survenir (même s’il est peu probable).
- Si aucun chemin ne mène à un interblocage, alors on est certain que le logiciel est valide.
- La vérification formelle est une preuve.

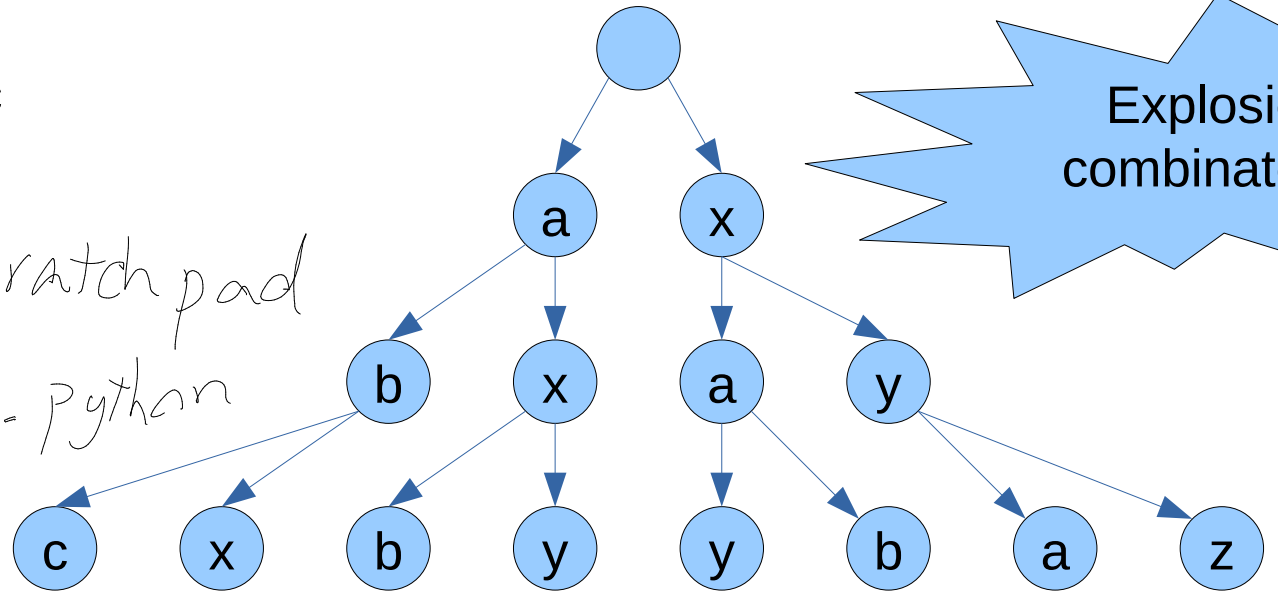
# Vérification formelle

- Modéliser le programme sous forme d'automate
- Explorer toutes les possibilités d'exécution
- Vérifier si une exécution mène à un interblocage

Exemple : deux processus, 3 instructions

P1 : a,b,c
P2 : x,y,z

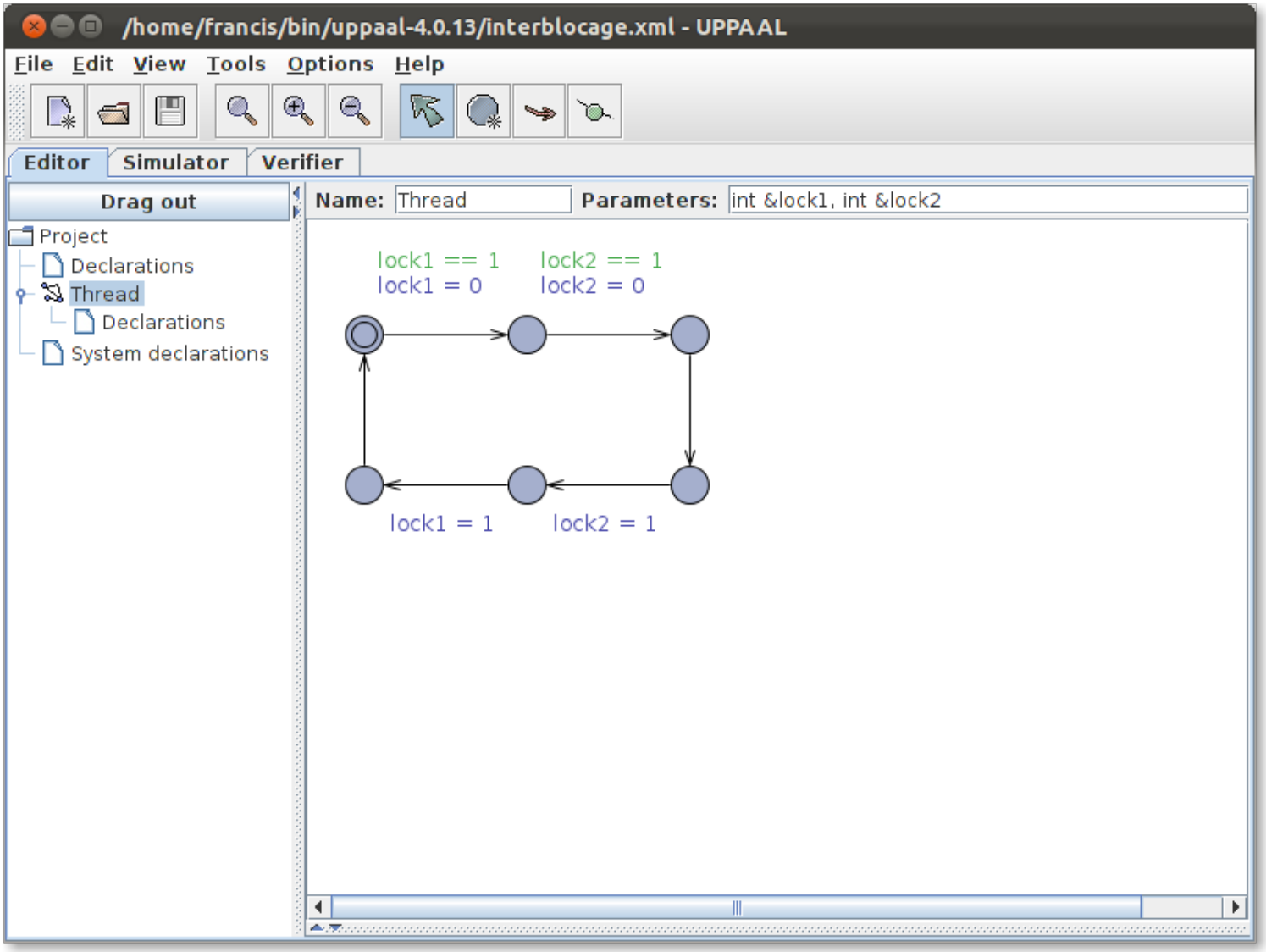
ex: scratch pad
Combi_python



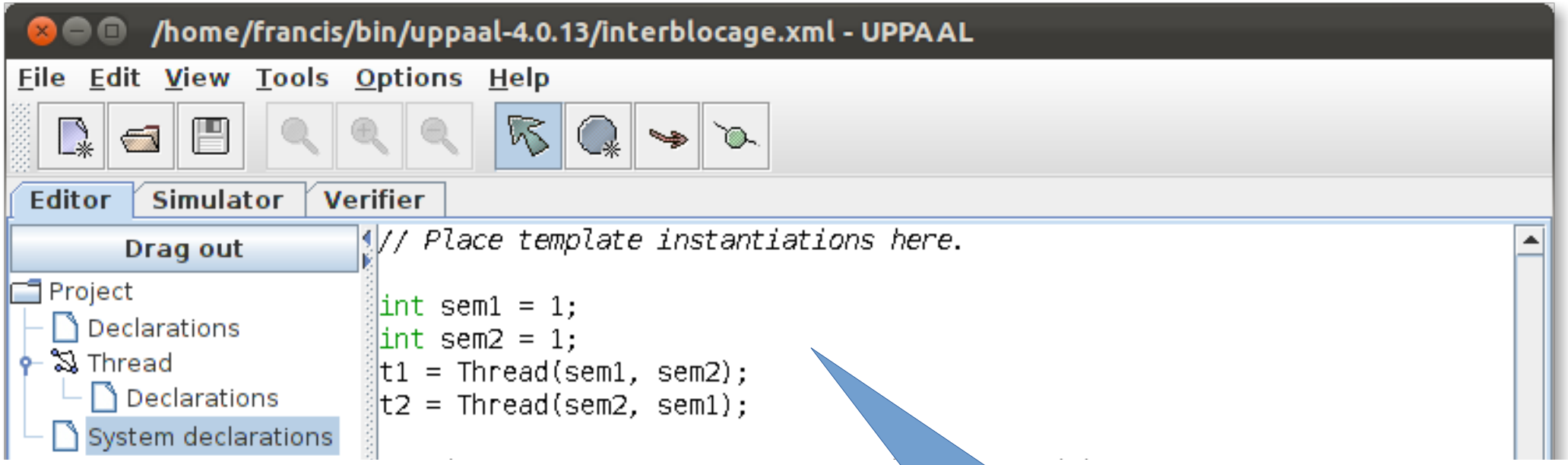

# Vérification formelle 101

- L'explosion combinatoire rend difficile la vérification et applicable que pour de petits logiciels ou quelques routines (ex: implémentation d'une structure de donnée)
- La vérification formelle est faite avec des algorithmes sophistiqués pour atténuer l'explosion combinatoire autant que possible.
- Exemple de logiciel de vérification formelle:
  – UPPAAL, Spin, Promela, Java Pathfinder, etc.

# Vérification formelle: exemple (1) : modèle

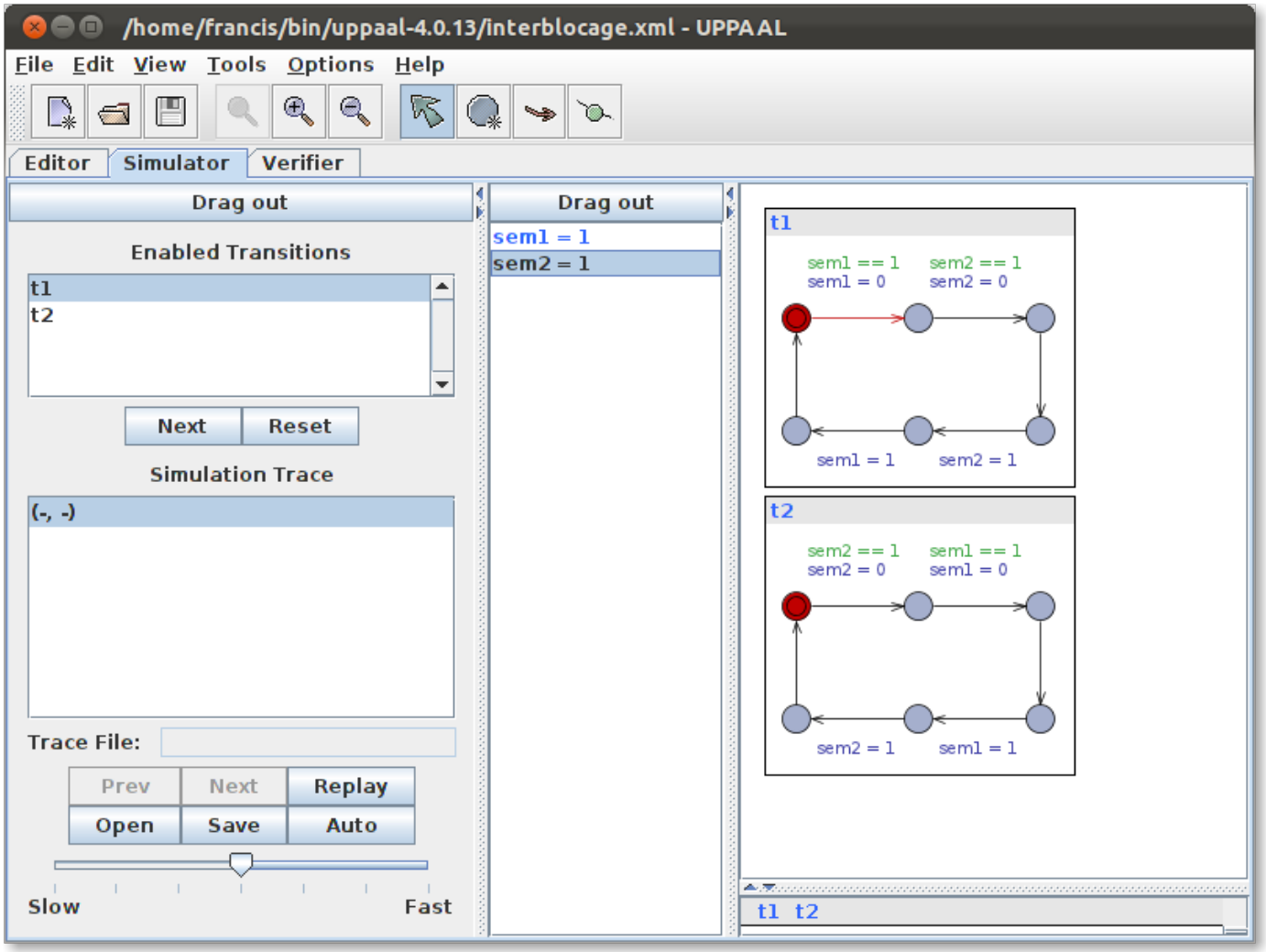


# Vérification formelle: exemple (2) : système complet

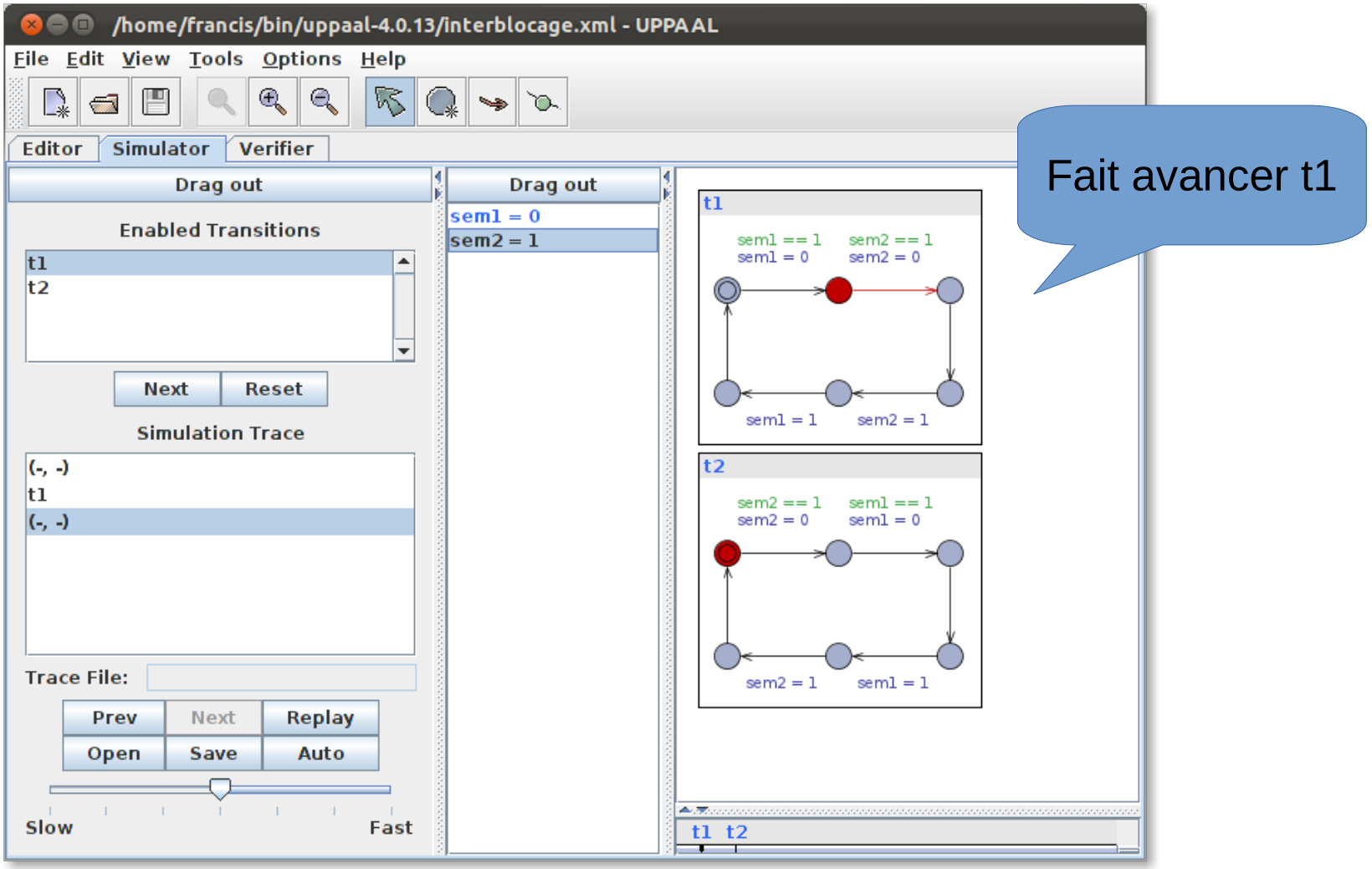



Le système comprend deux Thread, dont l'ordre de réservation des sémaphores est inversé.

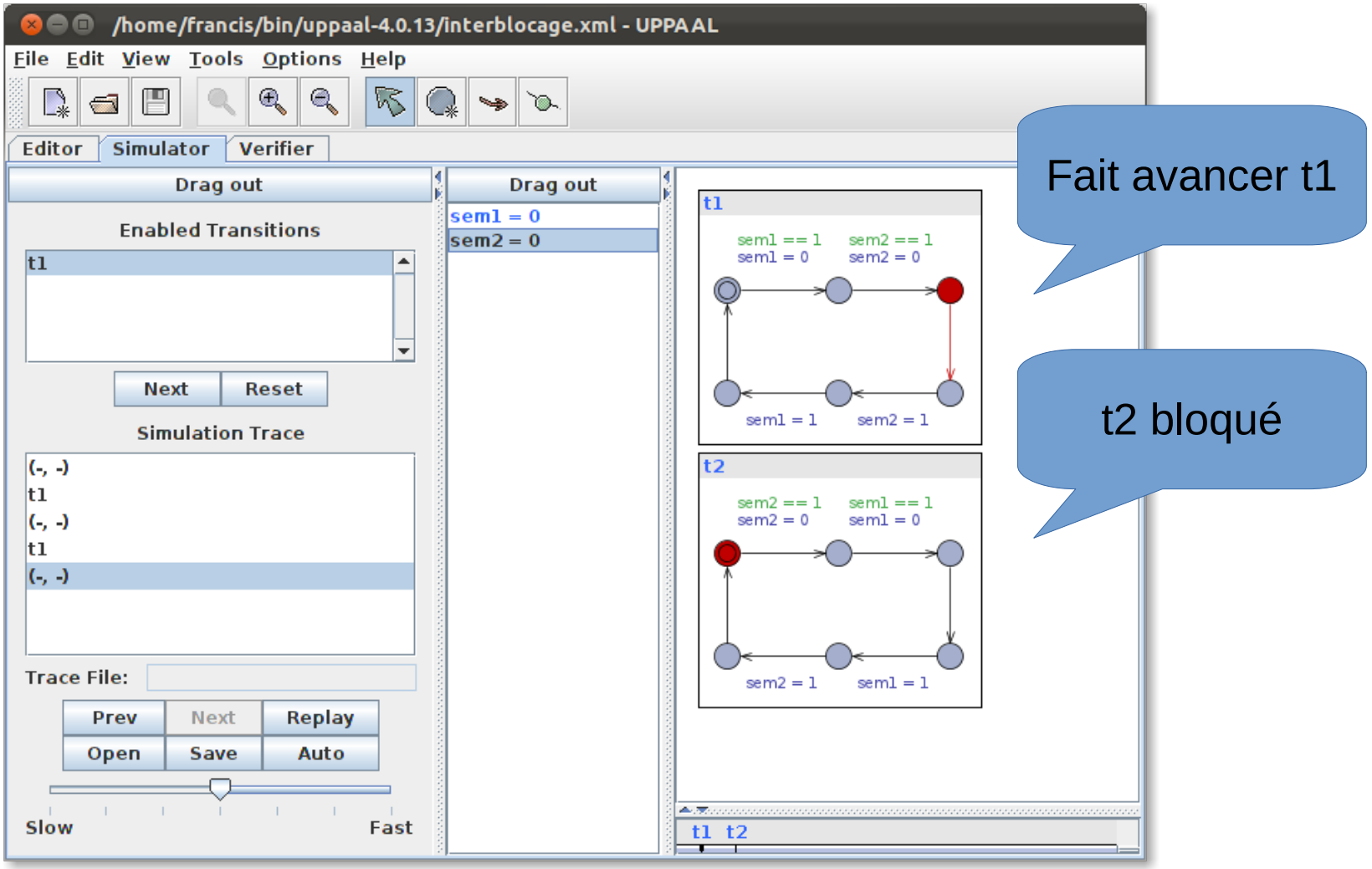
# Vérification formelle: exemple (3) simulation, état initial



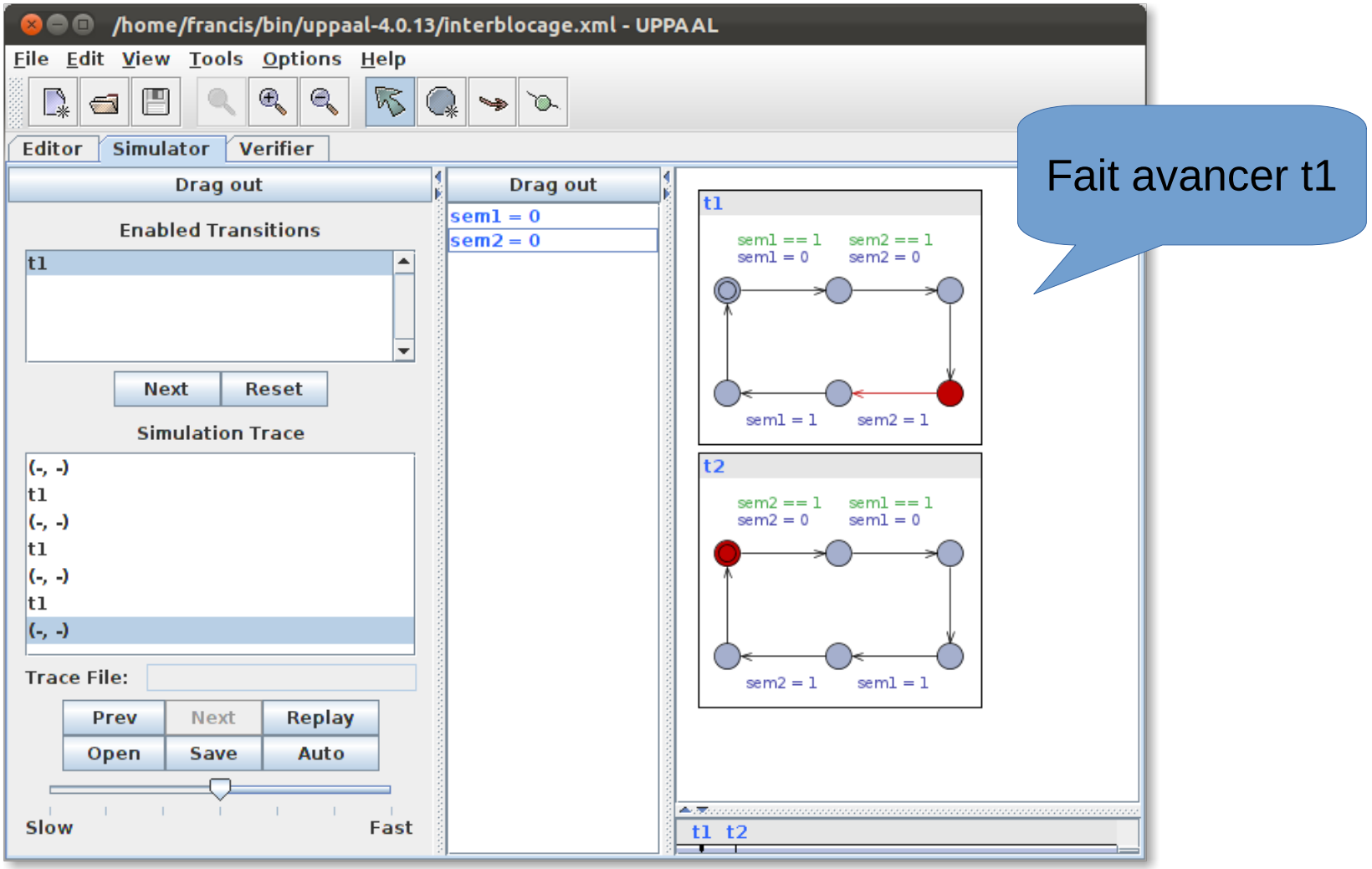
# Vérification formelle: exemple (4) simulation



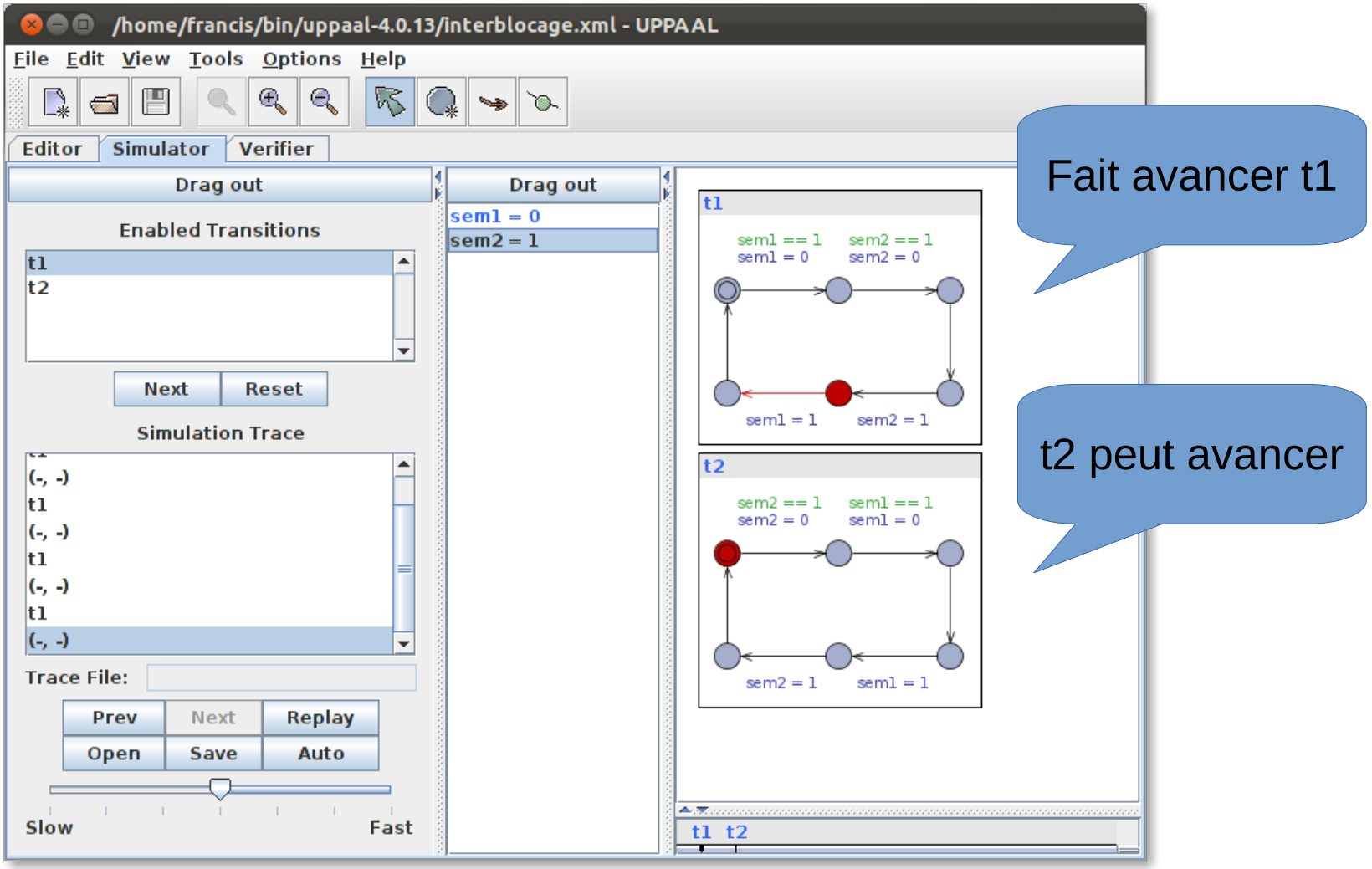
# Vérification formelle: exemple (5) simulation



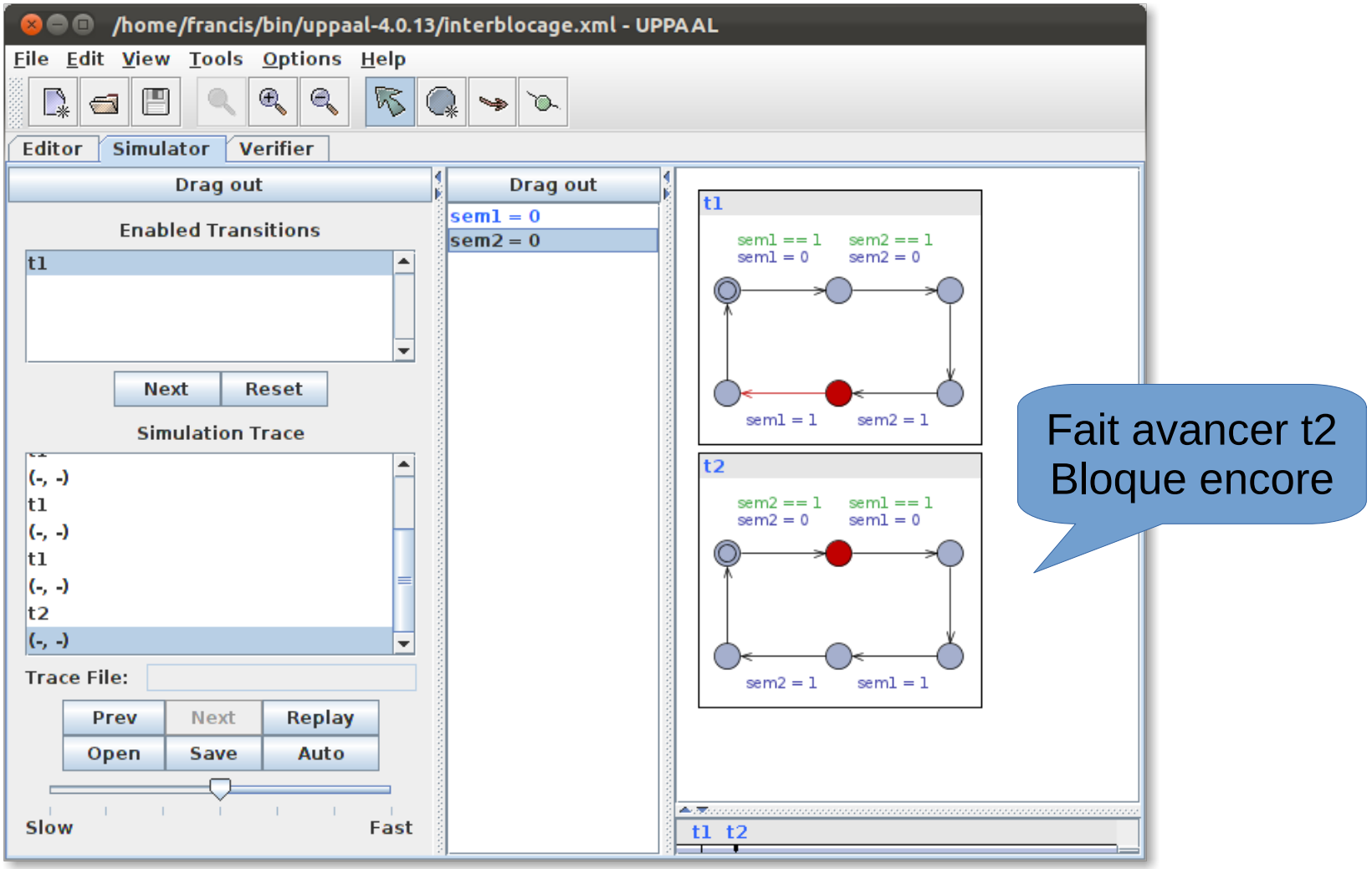
# Vérification formelle: exemple (6) simulation



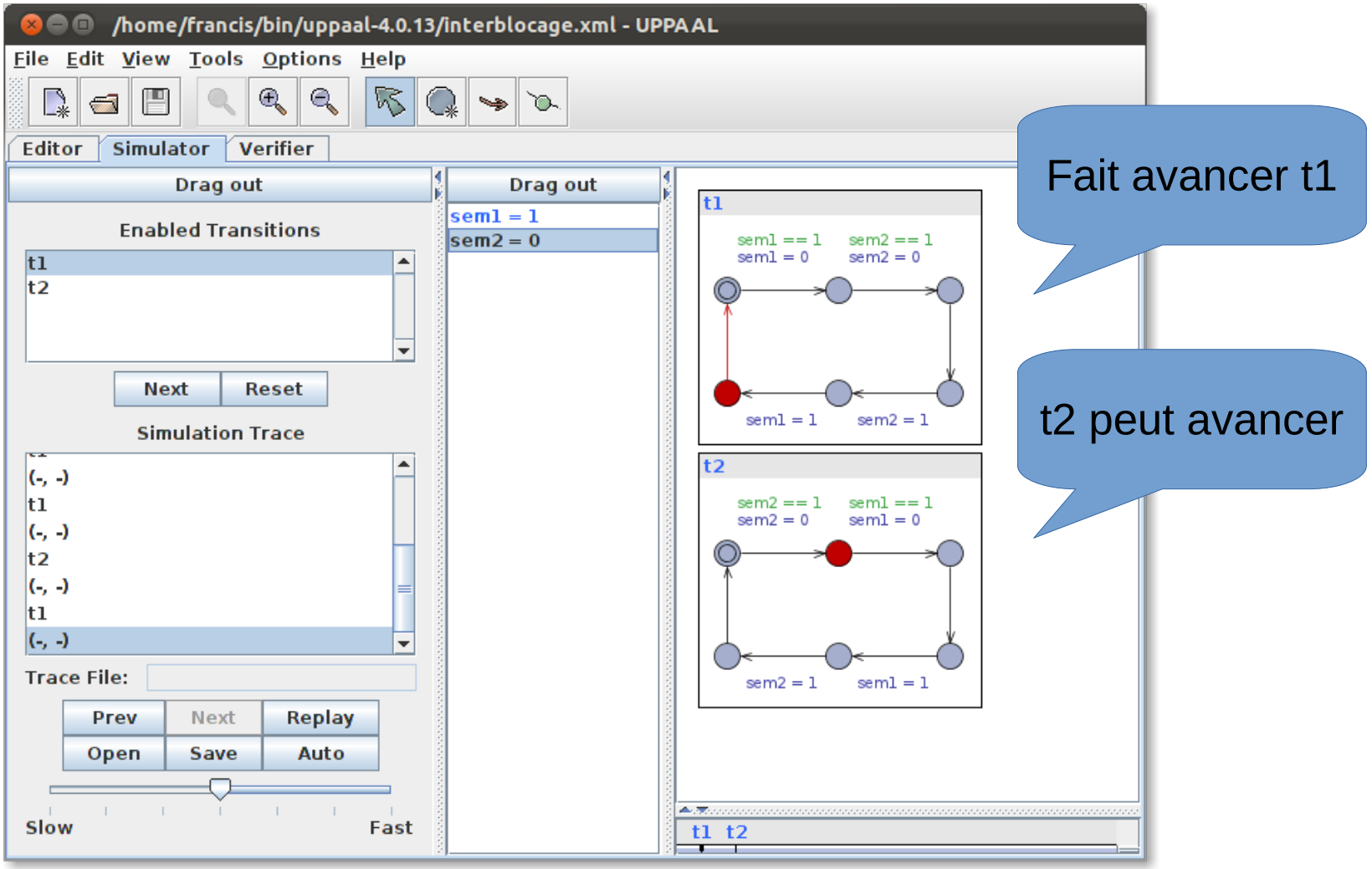
# Vérification formelle: exemple (7) simulation



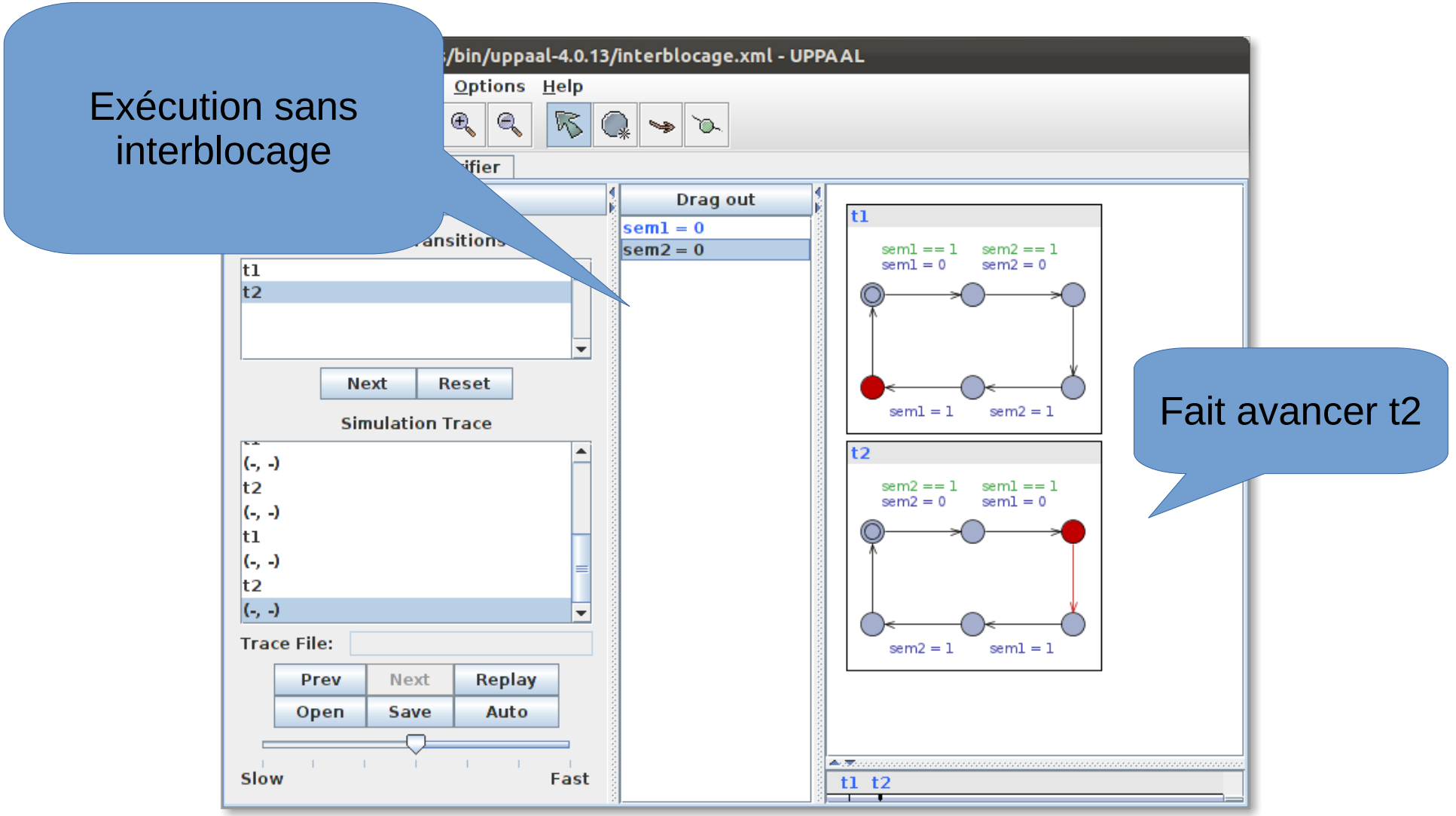
# Vérification formelle: exemple (8) simulation



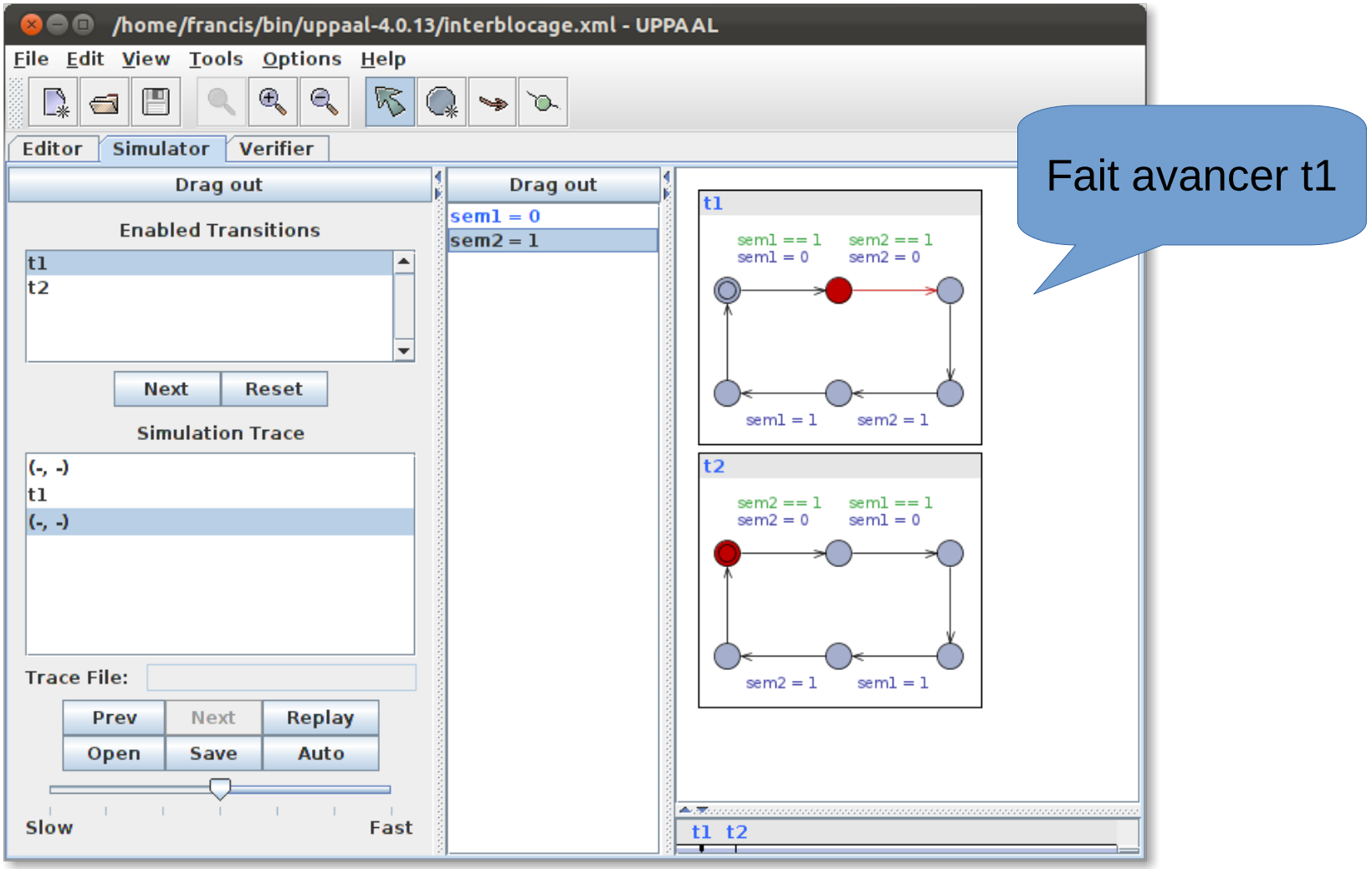
# Vérification formelle: exemple (9) simulation



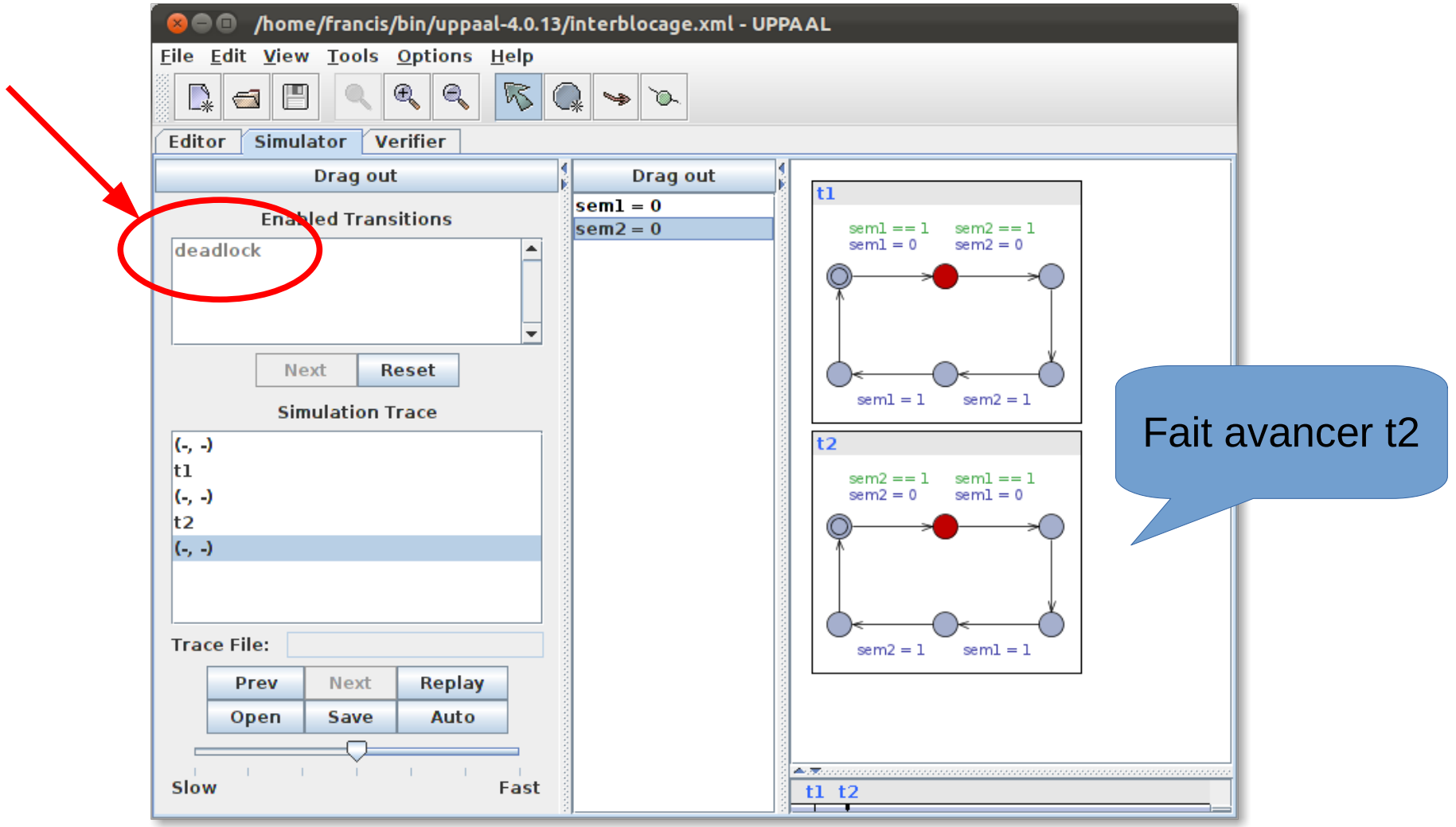
# Vérification formelle: exemple (10) simulation



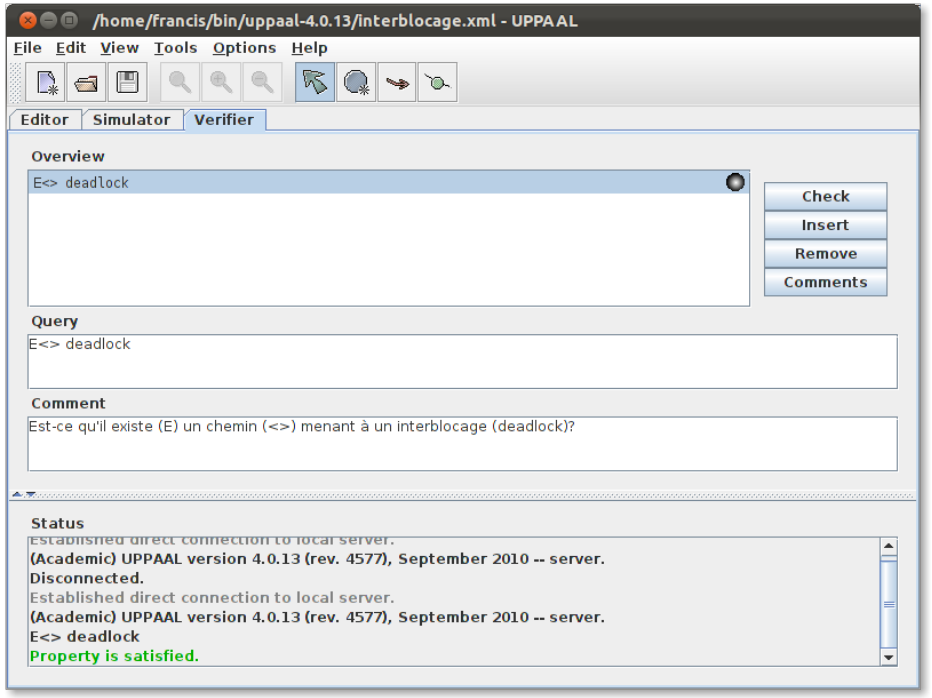
# Vérification formelle: exemple (11) simulation: recommence



# Vérification formelle: exemple (12) simulation interblocage!



# Vérification formelle: exemple (13) requête de vérification




- E<> deadlock
- Est-ce qu'il existe au moins un chemin menant à un interblocage?
- **Property is satisfied**
- Donc, il peut y avoir un interblocage!

# Changer l'ordre de réservation des ressources

- Réservation des ressources en ordre inverse
- t1 = Thread(sem1, sem2)
- t2 = Thread(sem2, sem1)
- E<> deadlock
- **Property is satisfied**
- **Un interblocage est possible!**

- Réservation des ressources dans le même ordre
- t1 = Thread(sem1, sem2)
- t2 = Thread(sem1, sem2)
- E<> deadlock
- **Property is not satisfied**
- **Aucun interblocage possible!**

# Vérification en fonctionnement

- Exécution du code prouve qu'il est correct
- Enregistrement de l'ordre et du contexte des verrous
- **ThreadSanitizer** : instrumentation à la compilation
  - Nécessite le code source, surcoût modéré
  - gcc -fsanitize=thread
- **Valgrind** : exécution dynamique
  - Fonctionne avec les programme précompilés
  - Surcoût élevé
  - Valgrind --tool=helgrind

# Interblocage sous Linux (1)

- Un interblocage dans le noyau est un bogue grave qui doit être réglé
- Options CONFIG_LOCKUP_DETECTOR
  - Detect hard and soft lockup
- Softlockups: si exécute en mode noyau plus de 60 secondes sans ordonnancement
  - Utilise un htimer
- Hardlockup: si exécute en mode noyau plus de 60 secondes sans aucune interruption
  - Utilise Non Maskable Interrupt
- Permet de forcer automatiquement un redémarrage si cette condition survient (et récupérer de cette erreur normalement fatale)

# Interblocage sous Linux (2)

- Option CONFIG_PROVE_LOCKING
- Vérification en fonctionnement
  - Pas une analyse statique : utilise l'exécution du noyau
  - Pas un test : permets de prouver que le code exécuté ne permet aucun interblocage
- Enregistre, lors de l'exécution, l'ordre de réservation des verrous et leur contexte (normal ou interruption)
- Le code doit être exécuté pour être vérifié
- Détecte si un verrou peut être réservé deux fois par récursion
- Détecte si l'ordre de réservation des verrous peut mener à un interblocage
- Activé lors du développement seulement

# Interblocage sous Linux (3)

- Option CONFIG_DEBUG_SPINLOCK_SLEEP
- Avertir si une routine du noyau se met en veille avec un verrou.
- Si une routine est mise veille avec un verrou, il se peut qu'une autre soit réveillée et en ait besoin, mais ne pourra pas s'exécuter.

# Interblocage sous Linux (4)

- CONFIG_RCU_TORTURE_TEST
- Exécute un test de charge sur les structures de données Read-Copy-Update pour les mettre à l'épreuve.
- Il s'agit d'un test, pas d'une preuve, mais augmente la confiance dans l'implémentation.
- Si aucun bogue ne se produit pendant le test, alors le logiciel est probablement correct.

# Thread Local Storage

- Variable distincte par fil d’exécution
- Évite l’accès concurrent
- Allocation en fonction du nombre de fil d’exécution
  - Essentiellement un tableau accédé avec un entier représentant l’ID du fil d’exécution
  - Alignement nécessaire pour éviter le faux-partage
  - Faux-partage : ralentissement dû à la réplication de la ligne de cache entre processeurs

# Variable par processeur

- Il y a toujours au plus 1 tâche par processeur
- Si la préemption et les interruptions sont désactivés, alors aucune condition critique n’est possible
- Fréquent en mode noyau (Linux)
- Exemple : liste des tâches pour un processeur (run queue de l’ordonnanceur)

```
DECLARE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
```

# Instructions atomiques

- Pas de verrou, pas d'interblocage possible!
- API C stdatomic.h
- Exemple : remplacer le verrou read/write dans le code météo
- Sur Intel i7-4600U 38x plus rapide que pthread_rwlock

```
for (int i = 0; i < repeat; i++) {
  struct meteo* next = &array[i];

  next->val1 = i;
  next->val2 = i;
  next->val3 = i;
  next->val4 = i;

  // Barrière de compilation ET mémoire
  __sync_synchronize();
  ACCESS_ONCE(data) = next;

  nanosleep(&ts, NULL);
}
```

Écriture

Lecture

```
while (ACCESS_ONCE(running)) {

  __sync_synchronize();
  struct meteo* tmp = ACCESS_ONCE(d
  // utilisation de la météo
```

# Read-Copy-Update

- Basé sur les instructions atomique et barrières mémoires
- Cycle de vie :
  - Publication d’une structure (i.e. météo)
  - Copie de la structure pour modification
  - Publier la nouvelle version
  - Quand on est certain que l’ancienne version n’est plus utilisée, on la libère
- Utilisé dans le noyau Linux pour éviter les verrous
  - Exemple : manipulation d’une liste partagée

# Multiple Version Concurency Control (MVCC)

- Permet l'accès en lecture et l'écriture simultanément à une donnée
- En cas de modification d'une donnée en utilisation, une copie de l'ancienne valeur est conservée
- La valeur à jour est visible lors du prochain accès
- Application : serveurs de base de données (i.e. PostgreSQL)

# Conclusion

- Conditions d'interblocage
  - Règle simple : réserver les ressources toujours dans le même ordre!
- Différence entre test et vérification
  - Test : facile à faire, mais ne prouve pas l'exactitude
  - Vérification formelle : difficile à faire, mais prouve l'exactitude
- Linux a plusieurs lignes de défense pour trouver les bogues reliés aux interblocages
- Techniques sans verrous sont complexes, mais sont généralement performantes et sans interblocages