UNIVERSITE PIERRE ET MARIE CURIE
UFR Informatique
Master 1 SESI

Rapport de réalisation du projet SESI.
Thème :

Modélisation et Analyse du protocole de cohérence de caches de la machine multiprocesseur TSAR : absence de deadlocks

Proposé et dirigé par :

- **M^e^ Emmanuelle ENCRENAZ**

Réalisé par :

- **M^r^ Akli MANSOUR**

2011-2012

SOMMAIRE

1. Objectif du projet

L'objectif du projet « Modélisation et Analyse du protocole de cohérence de caches de la machine multiprocesseur TSAR : absence de deadlocks » peut de résumé comme suit :

- Simplifier et optimiser les différents modèles de la machine multiprocesseurs TSAR déjà décrit par M.Najem, en l'occurrence le contrôleur mémoire et le cache L1.
- Simplifier le protocole de cohérence du cache.
- Simplifier la plateforme modélisant la machine TSAR pour vérifier l'absence de blocage (deadlocks) avec l'outil de vérification SPIN.
- Par la suite complexifier le protocole en intégrant le mécanisme LL/SC permettant de synchroniser des processus sur l'accès à des ressources partagées.

2. Environnement de travail

Dans cette partie je présente quelques définitions et une présentation de l'outil SPIN.

2.1. La machine TSAR

TSAR est l'acronyme de « Tera-Scale Architecture » qui est un projet européen coordonné par la société BULL qui consiste à concevoir une machine multiprocesseur scalable et cohérente jusqu'à 4096 processeurs sous forme de clusters de 4 processeurs interconnectés par un réseau de communication distribué (DSPIN), avec une mémoire répartie sur l'ensemble des clusters.

Chaque processeur admet deux caches L1 de type Write-through, (Instruction_cache & Data_cache). La mémoire étant de taille maximale de 1 Téraoctet (40 bits d'adressage virtuel), est distribuée physiquement entre les clusters mais un processeur peut accéder à tout l'espace d'adressage, ce qui rend le temps d'accès à une donnée dépendant de la distance entre le processeur qui fait la demande et la zone mémoire stockant la donnée.

2.2. Protocole de cohérence de TSAR

Un protocole de cohérence mémoire est l'ensemble des règles permettant d'organiser l'accès aux blocs mémoires partagés.

Pour la machine TSAR les réalisateurs du projet TSAR ont intégré un protocole de cohérence spécial, le DHCCP « Distributed Hybrid Cache Coherence Protocol ». Celui-ci est une combinaison de deux protocoles. Ces caractéristiques sont les suivantes :

- 2 niveaux de caches :
  - Cache L1 : Write through.
  - Contrôleur mémoire : Write back.
- 2 modes de diffusion du MC vers le cache L1:
  - Broadcast avec invalidation (B_inv)
  - Multicast avec mis à jour/avec invalidation de la ligne dans le cache (M_up/M_inv)
- Changement de mode diffusion dépend du nombre de copies valides N dans les caches (Cache_Threshold= 2) :
  - Si N > TH : B_inv.
  - Si N ≤ TH : M_up.

Remarque :

On à fait l'hypothèse que les deux lignes X et Y n'entrent pas en collision dans le contrôleur mémoire. De ce fait, il n'y a pas de M_inv (multicast_invalidate).

NB : Une analyse de ce protocole a été déjà faite par M.Najem, mais il n'y a pas de preuve sur l'absence de situation de deadlocks sur tout le système[voir rapport de réalisation de M.Najem].

2.3. SPIN model checker

SPIN est un outil développé par Bell-Labs qui permet :

- La simulation et la vérification de systèmes ou des modèles de systèmes décrit avec le langage Promela.
- Etudier la dynamique du système, représentée sous forme de graphe d'états accessibles tout en faisant un choix sur la manière de le parcourir (en profondeur ou en largeur).
- Vérifier des propriétés sur le système à étudier exprimées sous forme de formules logiques.

Le système est modélisé comme un ensemble de processus concurrents asynchrones, s'échangeant des messages sur des canaux bornés.

- **« ! m»,** signifie l'émission d'un message m sur un canal de communication.
- **« ?m »,** signifie la réception d'un message m sur un canal de communication**.**

Les émissions et réceptions sont bloquantes.

L'outil spin propose deux méthodes pour l'analyse de deadlocks :

1- **La vérification** : consiste à vérifier sur l'ensemble de tous les états du graphe des états accessibles l'absence de deadlocks suivant le mode de parcours choisi (en profondeur ou en largeur)

2- **La simulation** : étant moins générale, elle permet l'analyse de deadlocks sur un seul chemin du graphe d'états. Ce chemin peut être choisi aléatoirement par SPIN ou bien spécifié par l'utilisateur.

Exemple :

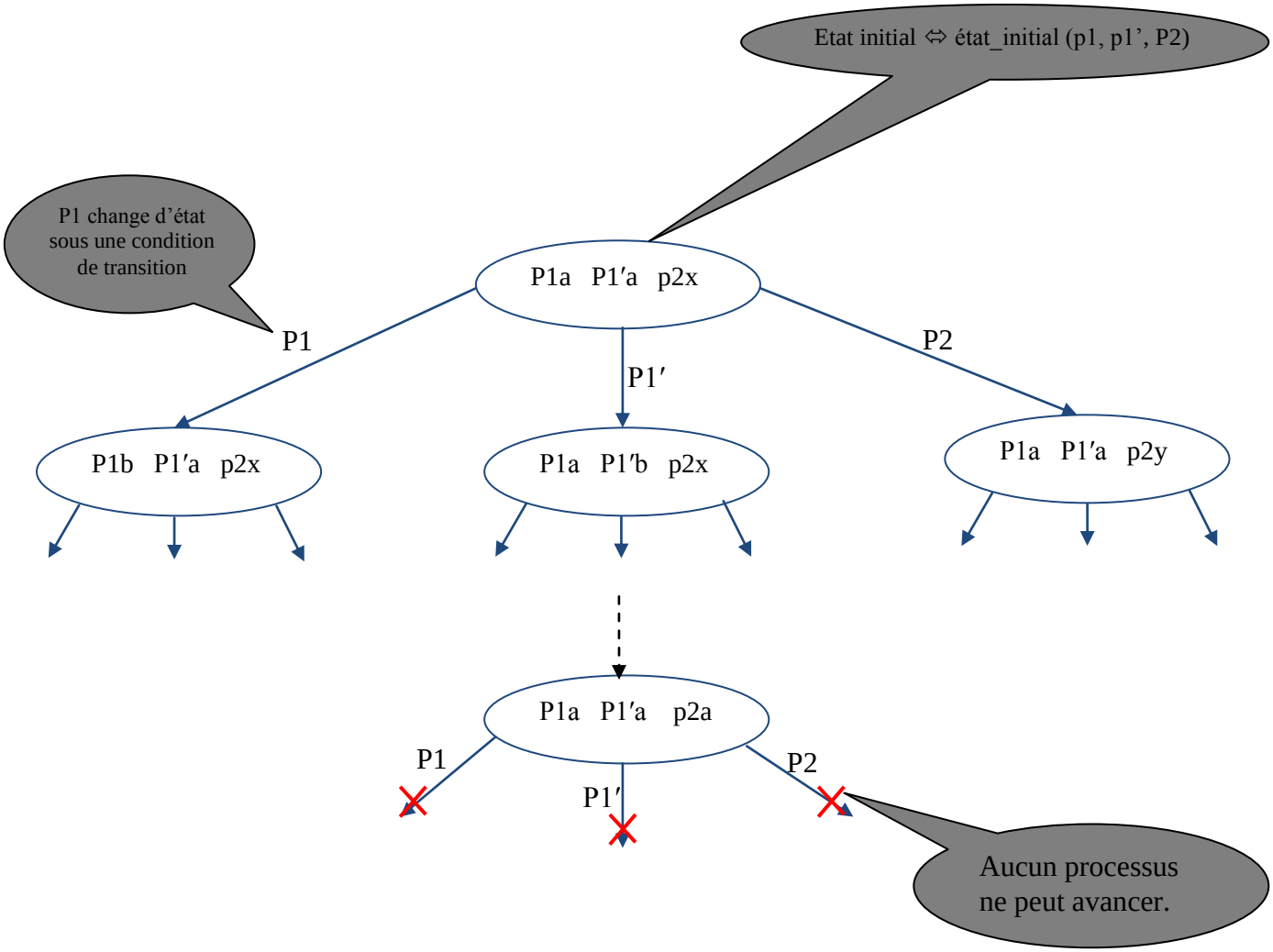
Dans cet exemple nous allons déclarer 2 types de processus (p1 et p2). Chaque type peut avoir un comportement bien spécifique avec des gardes des actions personnalisées.

Nous allons par la suite exécuter une ou plusieurs instances des ces processus avec des paramètres différents dans le processus d'initialisation du système « init {} », à l'aide de l'instruction « run ».

```
Proctype p1 ( ) {          Proctype p2 ( ) {          init{
   a ;                        x ;                        run p1 ( ) ;
   b ;                        y ;                        run p1 ( ) ;
   c ;                        z ;                        run p2 ( ) ;
 }                          }                          }
```

processus asynchrones

La dynamique du système peut être représentée par le graphe des états accessibles à partir de l'état initial. Ce graphe est le produit asynchrone des automates représentant les processus p1, p1′ et p3. (Page suivante)



Dans ce graphe, un état représente la position du compteur ordinal de chaque processus ainsi que la valeur courante de chaque variable (locale et globale). Chaque transition entre deux états représente l'exécution d'une instruction d'un des processus, les 2 autres ne bougeant pas. Une exécution du système est modélisée par une séquence de transitions dans le graphe des états accessibles.

Un état de deadlock est un état à partir duquel aucune transition n'est franchissable : aucun processus du système ne peut progresser.

3. Fournitures

3.1. Plateforme du projet

Cette figure représente la plateforme à trois processeurs du système à vérifier.
Chaque processeur possède son propre cache L1 (data cache & instruction cache)

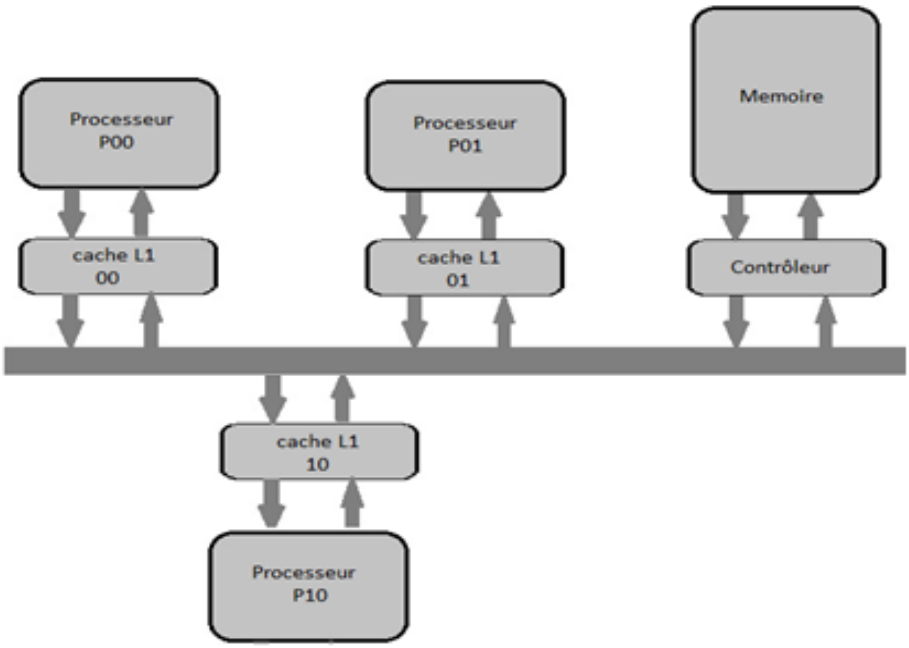


Figure 1 : Plateforme à 3 processeurs.

Le fait que le TH=2 (seuil pour le changement de mode de diffusion) impose qu'il faut au minimum 3 cache L1 pour pouvoir vérifier les deux modes de diffusion (B_inv, M_up).

3.2. Les canaux de communication [M.N]

Pour communiquer les processus utilisent des canaux de communication *fifo* qui sont regroupé dans la figure qui suit :

Les composants de la plate-forme communiquent entre eux par des requêtes dans ces canaux de communication qui sont partagés entre 3 groume selon l'émetteur et le récepteur :

1- **Proc-L1 :** c'est la catégorie des canaux de communication entre le processeur et le cache L1.

[M.N] : voir rapport de projet de M.Najem pour plus de détails.

2- **L1-MC :** c'est l'ensemble des canaux de communication entre le cache L1 et le contrôleur mémoire.

3- **MC-MEM :** représente l'ensemble des canaux de communication entre le contrôleur mémoire et la mémoire.

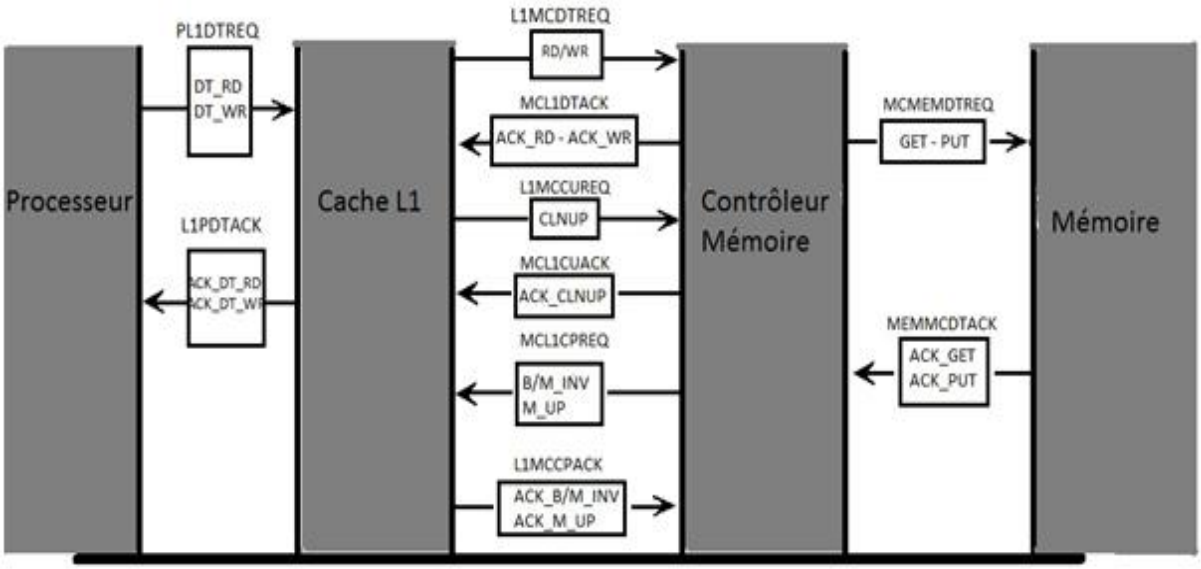


Figure 2 : Les différents canaux de communication

Hypothèses de modélisation :

- **L 1 :** une ligne du cache L1 stocke soit la ligne d'adresse X, soit la ligne d'adresse Y.
- **MC :** X et Y ne sont pas sur la même ligne, donc pas d'éviction dans le contrôleur mêmoire vu qu'elles ne rentrent pas en collision.

4. Réalisation :

4.1. Enrichissement des automates L1 et MC

Dans cette partie, à partir des graphes d'états fourni par mon encadrant j'ai vérifié l'adéquation entre la représentation graphique et le code Promela pour l'automate du cache L1 comme pour l'automate du contrôleur mémoire tout en s'assurant que toutes les transitions figurent sur les automates d'une manière correcte. Donc j'ai pu fournir deux automates graphiques complets et bien détaillés, l'un pour le cache L1 et l'autre pour le contrôleur mémoire dont la description des différents états et transitions est décrite en annexe 1 et 2.
[Consulter rapport de M.Najem pour une définition de chaque état.]

4.2. Optimisation du code

Dans cette partie du projet j'ai optimisé et simplifié le code du modèle représentant le protocole de cohérence des caches de la machine TSAR de manière à éviter toute manipulation de donnée, c'est-à-dire veiller à ne pas utiliser les valeurs réelles des lignes mémoire X et Y dans les messages.

Ceci nous à amené à modifier la structure des messages véhiculés dans le système en supprimant le champ dédié à la transmission de la valeur des cases mémoire.

J'ai aussi supprimé tous les observateurs qui servaient à vérifier certaines propriétés. Cela pour minimiser l'espace d'états accessibles de notre système afin d'augmenter nos chances d'éviter l'explosion combinatoire du nombre d'états.

Donc à la fin, j'ai pu fournir un modèle de la machine TSAR à trois processeurs et sans manipulation ni transmission de valeurs. [Dossier : plateforme_3_proc_simple]

4.3. Test du système (spin)

Après la phase de spécification et d'optimisation vient la phase de tests où j'ai passé les différentes plateformes réalisées à l'outil SPIN pour vérifier l'absence de deadlock sur tout ou partie de l'espace d'états du modèle.

4.3.1 Plateforme à 3 proc :

Dans cette partie j'ai passé la plateforme à 3 processeurs simplifiée sur l'outil SPIN avec un TH=2 (Threshold). A cause du phénomène de l'explosion combinatoire, la

vérification n'a pas pu aller jusqu'à parcourir tout l'ensemble des états accessibles du système, néanmoins on a pu vérifier l'absence de deadlock jusqu'à une certaine profondeur en parcourant le graphe des états atteignables en largeur d'abord.

Les résultats obtenus sont récapitulés dans le tableau suivant :

Stratégie de parcours	Profondeur atteinte (états)	Nombre d'états parcourus	Nombre de transitions franchies.
En profondeur d'abord	**28 089 348**	**$1{,}7.10^{8}$**	**$3{,}75.10^{8}$**
En largeur d'abord	**264**	**$1{,}95.10^{9}$**	**$1{,}23.10^{10}$**

Remarque

La vérification a été réalisée sur le serveur de calcul « rythm » du réseau de recherche de l'UPMC, qui est un 12 cœurs hyperthreading avec 72Go de RAM que nous avons utilisé en entier vu que c'est un espace d'adressage logiquement partagé.

4.3.2 Plateforme à 2 proc :

Vus les résultats obtenus lors de la vérification faite sur la plateforme à 3 processeurs, et pour tenter toujours d'éviter l'explosion combinatoire du nombre d'états on a décidé avec mon encadrant de réduire la plateforme à une plateforme de 2 processeurs.

Cette figure montre l'architecture de la plateforme à deux processeurs réalisée :

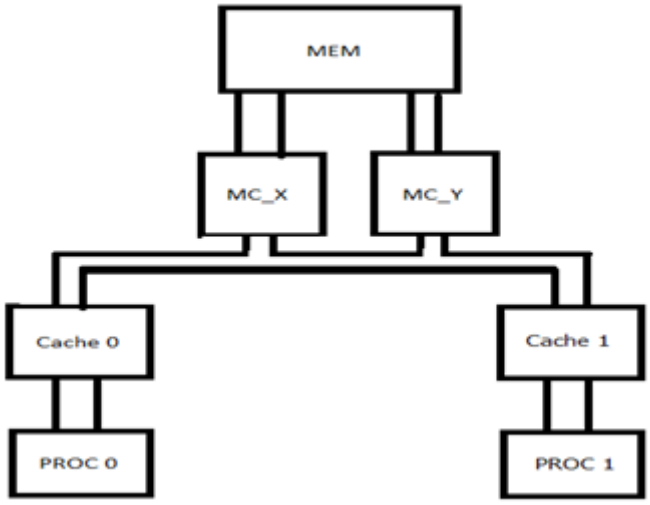


Figure 3 : Architecture de la plateforme à deux processeurs

Pour rappel, l'implémentation du protocole de cohérence de la mémoire de la machine TSAR nécessite au minimum la présence de 3 caches L1 (i.e. 3 processeurs) afin de pouvoir vérifier les deux modes de diffusion : Multicast et Broadcast ; si le nombre de copies d'une ligne est inférieur au seuil TH, une stratégie Multicast/Update est choisie (ce qui implique que les copies sont localisées). Si le nombre de copies dépasse le seuil TH, la stratégie Broadcast_Invalidate est adoptée (la localisation précise de chaque copie n'est plus nécessaire).

Donc pour remédier au fait que nous n'avons pas pu faire une vérification complète du système avec une plateforme à 3 processeurs, on a décidé de mener deux vérifications distinctes sur la plateforme à 2 processeurs en jouant sur la valeur du seuil « TH » :

- **Première vérification, TH = 2 :**

La vérification de la plateforme à 2 processeurs avec un TH=2 a abouti jusqu'à la fin, c'est-à-dire qu'il n'y a pas de deadlock sur tout l'ensemble des états accessibles du système.

Par contre, cette vérification ne nous permet de tirer conclusion que sur une partie du protocole DHCCP. Vu qu'on ne dispose que de deux caches L1, le nombre de copies d'un même bloc mémoire présent dans les caches ne peut être supérieur à 2. De ce fait, on ne peut rien dire sur le cas du Broadcast_Invalidate mais on peut conclure que le cas du Multicast_Update a été vérifié avec succès.

- **Deuxième vérification, TH = 1 :**

Contrairement à la première vérification, cette deuxième vérification nous ne permettra pas de voir le cas du Multicast/Update apparaître car dès que le nombre de copies valides présentes dans les caches passe à 2, on passe en mode Broadcast_Invalidate.

Compte tenu que la vérification a été faite sur tout l'ensemble des états accessibles de la plateforme on peut dire qu'il n'y a pas de deadlocks sur tout le système et le cas du Broadcast_Invalidate a été vérifié avec succès.

Conclusion sur la plateforme à 2 processeurs :

Les résultats obtenus lors des deux vérifications sont résumés dans le tableau suivant :

Valeur de TH	Temps (s)	Nombre d'états parcourus	Nombre de transitions franchies.
TH=1	**385**	**80 125 336**	**1,53 10^{+8}**
TH=2	**33,9**	**8 305 772**	**15 513 924**

Compte tenu des résultats obtenus avec ces deux vérifications sur la plateforme à deux processeurs, on peut dire que les deux modes de diffusion du protocole de cohérence de la machine TSAR ont été vérifiés avec succès séparément mais on ne peut rien dire sur la combinaison de ces deux modes.

4.4 Intégration du LL/SC

4.4.1 Spécification du mécanisme LL/SC

TSAR et le LL/SC:

L'architecture TSAR implémente les opérations de lecture et écriture (read then write) atomiques pour pouvoir supporter les différents mécanismes de synchronisation (verrou binaire, barrière, bascule set/reset…).

Avec ce mécanisme de LL/SC un programme logiciel pourra avoir la possibilité de lire une donnée à une adresse X et écrire une donnée à la même adresse X avec la garantie qu'aucun autre accès en écriture à cette adresse n'a été fait entre la lecture et l'écriture, tout en sachant que n'importe quelle adresse de l'espace d'adressage peut faire l'objet d'un accès atomique.

Principe générale :

Il existe plusieurs implémentations du mécanisme de synchronisation LL/SC :

- **Avec table de réservation :**

C'est l'implémentation naïve du mécanisme LL / SC : c'est une implémentation où c'est le contrôleur mémoire qui fait le gros du travail en gérant une "table de réservation".

On implante dans le contrôleur mémoire une table associative (toutes les cases ont la même sémantique) qui permet de conserver des paires (adresse, proc_id).

Problème :

Risque de débordement de la table de réservation notamment pour une architecture comme TSAR avec 4096 processeurs, le risque est bien présent.

- **Implémentation « optimisée » :**

L'idée de cette implémentation est la suivante :

La prise de réservation est mémorisée dans le L1 du processeur qui a exécuté le LL, et une seule réservation peut être enregistrée. On conserve l'adresse de réservation X et la donnée lue lors d'un LL. Donc, une instruction LL entraine une transaction "read" normale sur le réseau (un mot ou une ligne complète). Donc le contrôleur mémoire ne voit pas les LL.

L'instruction SC est interprétée par le cache L1 comme une instruction CAS (Compare And Swap). Une transaction CAS contient 3 informations : l'adresse X, l'ancienne valeur lue avec le LL, et la nouvelle valeur à écrire.

Le contrôleur mémoire compare la donnée lue par le cache L1 avec celle présente dans la mémoire. Si les deux valeurs sont égales le SC passe avec succès et la donnée est écrite dans la mémoire sinon ça renvoie un échec.

Dans cette implémentation un accès en écriture ordinaire (Write) à l'adresse de réservation engendre une invalidation du LL.

Exemple d'utilisation du LL / SC : (la prise de verrous)

R4 : contient l'adresse du verrou.

```
loop  :     ll    r1, 0(r4)        # r1 <= M[r4] : lire la valeur du verrou
            bne   r1, r0, loop     # si r4!=0 => verrou occupé, donc retester
            ori   r2, r0, 1        # r2 <= 1 : si non occupé => prendre le verrou
            sc    r2, 0(r4)        # M[r4] <= 1 tenter d'écrire pour prendre le verrou.
            beq   r2, R0, loop     # r2=0 si verrou non pris
```

Dans cet exemple on fait un LL sur l'adresse du verrou. Si le verrou est pris par un autre processus (i.e. r1!=0) alors on fait un autre LL jusqu'à ce que la valeur lue du verrou soit égale à '0'. Si c'est le cas on écrit dans un registre la valeur à écrire à l'adresse du verrou qui doit être différente de '0' pour prendre le verrou. Par la suite en tente d'écrire cette valeur à l'adresse du verrou avec un SC pour s'assurer que le verrou n'a pas été pris entre la lecture

et l'écriture à l'adresse de ce dernier. Si c'est un succès le SC renvoie dans 'r2' une valeur différente de '0' sinon il renvoie un '0'.

Choix d'implémentation pour notre projet :

Pour implémenter le mécanisme du LL/SC on a opté pour l'implémentation optimisée vu que LL/SC est implémenté de cette manière dans la machine TSAR et du fait aussi qu'elle soit une solution scalable. Mais on est passé par deux étapes de réflexion qui sont les suivantes :

Etape 1 :

Dans cette étape j'ai réalisé avec l'accord de mon encadrant une étude pour l'implémentation optimisé du mécanisme LL/SC mais sans manipulation de données dans les messages échangés dans les canaux de communication pour diminuer le phénomène de l'explosion combinatoire. Cette approche a été rapidement abandonnée car l'absence de valeur manipulée nécessitait d'introduire d'autres mécanismes complexes pour pallier ce manque (un bit de validité et des types de messages et des canaux supplémentaires).

Etape 2 :

C'est l'étape où on a choisi d'implémenter le mécanisme du LL/SC tel qu'il est le cas dans la machine TSAR et du coup on a été ramené à la manipulation de donnée dans les messages échangés dans les différents canaux de communication. La spécification de cette nouvelle implémentation est la suivante :

- **Spécification du LL :**

Un LL se traduit comme une transaction « read » classique avec quelques détails en plus au niveau de chaque composant de l'architecture.

Le processeur fait une requête de type LL sur une adresse X [LL(X)] vers le cache L1 et attend une réponse de type ACK_LL comme acquittement du LL indiquant que le LL est traité.

Le cache L1 envoie une requête de lecture ordinaire au contrôleur mémoire « RD » qui à son tour acquitte en fait un read ordinaire sur l'adresse X et renvoie la donnée au cache L1 dans acquittement « ACK_RD ».

A la réception d'un « ACK_RD » on enregistre la donnée lue dans le cache L1 comme pour un simple READ, également on enregistre l'adresse de réservation dans un registre du cache L1 et ainsi que la valeur lue et on acquitte le processeur avec un ACK_LL.

Remarque :

- Si « LL » après « LL », le dernier LL annule le précédent en écrasant l'ancienne valeur lue de l'adresse de réservation.
- Si « read » après « LL », aucun effet.
- Si « Write » après « LL », ce qui signifie que la valeur du bloc mémoire sur lequel on a fait une réservation a été modifiée, donc le LL est invalidé. De ce fait, il faut refaire un LL pour relire la nouvelle valeur et faire une réservation dans le cache L1.

Spécification du SC :

Le processeur fait une requête de type SC sur une adresse donnée X qui contient la valeur à écrire en cas de succès, et attend une réponse de type ACK_SC qui selon la valeur de retour à la signification suivante :

- Si val_ret = 1 => le SC est passé avec succès, symbolisé avec « ACK ».
- Si val_ret = 0 => le SC à fait échec et mets à zéro le registre source de l'instruction assembleur SC, symbolisé avec « NACK ».

Le cache L1 fait une transaction de type « W_SC »vers le contrôleur mémoire, où il transmet à la fois l'adresse destination, la valeur à écrire et la valeur lue avec le LL. Pour cela, j'ai créé un nouveau type de message qui permet d'envoyer les trois champs au même temps. Ses champs sont définis comme suit :

- **Mtype type :** Type du message (LL, SC, DT_RD, WR, M_UP, ACK…).
- **Bit addr :** L'adresse de la case mémoire.
- **Byte val :** La valeur de la case mémoire pour un READ.
  La valeur à écrire pour un WR ou un SC
- **Byte val_ll :** La valeur lue avec un LL pour un SC.

Ne rentre pas en considération pour RD/WR.

- **Byte cache_id :** Identifiant du cache.

Ce type de message n'intervient que pour la transmission des 3 requêtes suivantes du cache L1 vers le contrôleur mémoire : RD, WR, et W_SC

Par la suite, le cache L1 attend un acquittement « ACK_W_SC » positif ou négatif selon la valeur de retour.

A la réception du W_SC, le contrôleur mémoire procède comme suit :

- Fait un accès mémoire « GET » à l'adresse du W_SC pour récupérer la valeur présente en mémoire.
- Compare la valeur lue avec le LL avec celle présente en mémoire :
  - Si les deux valeurs sont égales => succès :
    - Acquitte le W_SC au L1 avec ACK_W_SC avec une valeur de retour = 1;
    - Effectue l'écriture de la donnée à l'adresse concernée.
    - Envoie une ou des requêtes d'invalidate ou d'update à tous les caches qui contiennent une copie valide du bloc mémoire concernée qui à leur tour invalide leurs LL s'ils en ont fait.
  - Si les deux valeurs sont différentes => échec : envoie un ACK_W_SC avec une valeur de retour = 0 et l'écriture n'est pas faite. Ce qui signifie qu'il y a eu soit une écriture ou un autre SC avant.

Réalisation de l'intégration du LL/SC:

- **Au niveau des canaux de communication :**

➢ **Proc-L1:**

Pour implémenter le LL/SC j'ai créé les types de messages suivants :

- LL (res SC) : message de demande d'un LL (res SC). Ces deux messages sont transmis dans le même canal que les DT_RD/DT_WR.

- ACK_LL (res ACK_SC) : message d'acquittement d'un LL (res SC). Ils proviennent du cache L1 vers le processeur. Ces deux message sont transmis dans le même canal que les ACK_DT_RD/ ACK_DT_WR.

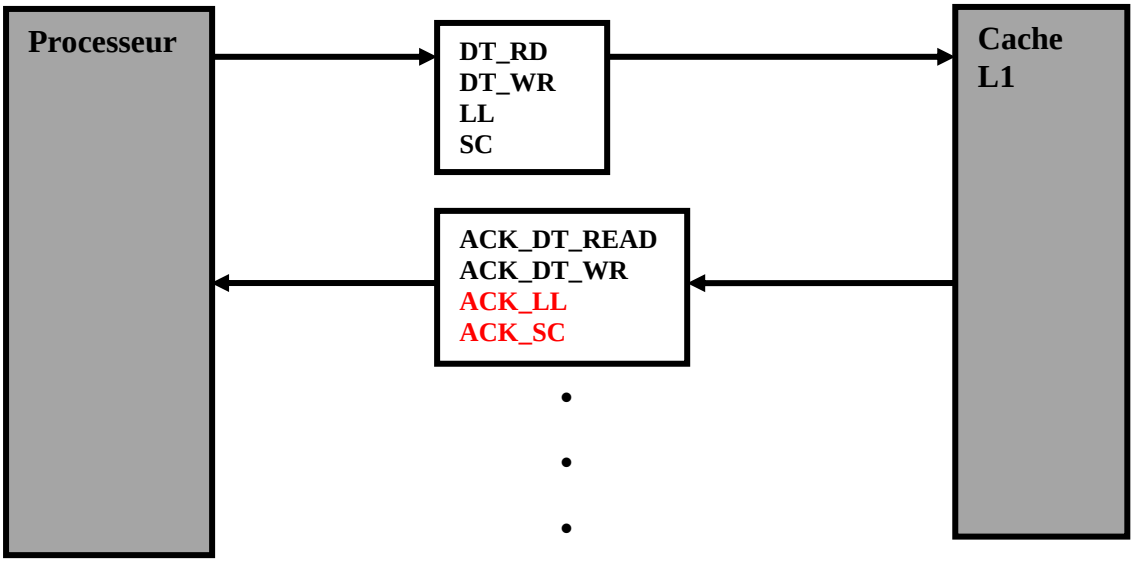


Figure 4: Quelques canaux de communication entre processeur et cache L1.

- **L1-Ctrl_Mem**

Pour implémenter le LL/SC j'ai créé les types de message suivants :

- W_ SC : message de demande d'un SC au contrôleur mémoire. Ce message est transmis dans le même canal que les RD/WR et qui ont touts les 3 le même type de message.

- ACK_W_SC : message d'acquittement d'un W_SC). Il provient du contrôleur mémoire vers le cache L1. Ce message est transmis dans le même canal que les ACK_RD/ ACK_WR.

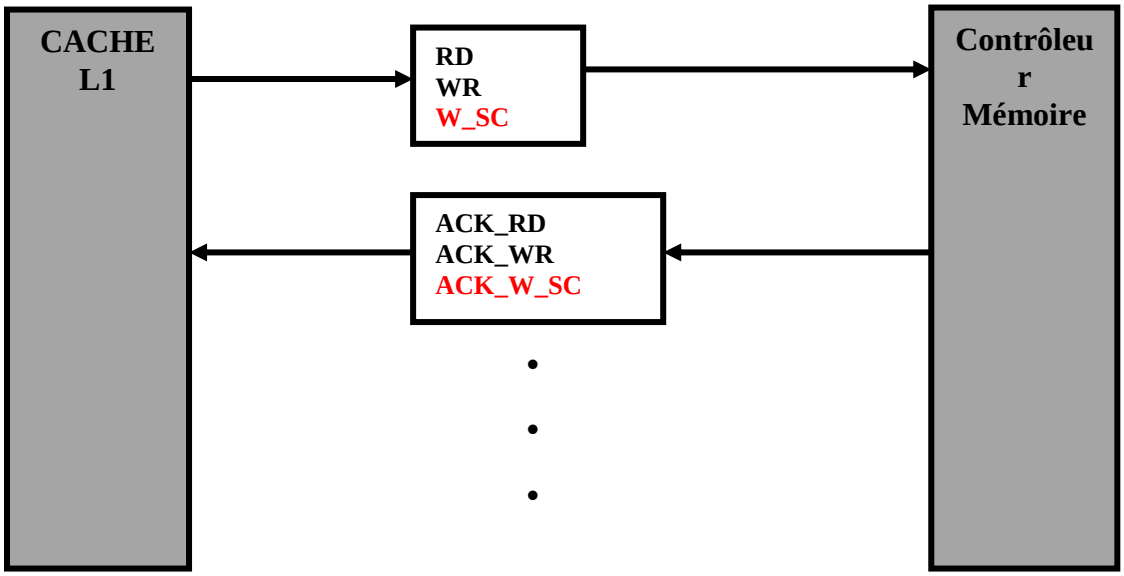


Figure 5: Quelques canaux de communication entre cache L1 et contrôleur mémoire.

Remarque :

Les canaux non représentés restent inchangés. (Voir rapport de spécification)

- **Au niveau des automates :**

Modèle du processeur :

Le graphe suivant représente l'automate du processeur enrichi avec les états et les transitions correspondantes au mécanisme LL/SC. Il génère des requêtes de lecture/écriture/LL/SC sur X et Y aléatoirement, et attend l'acquittement de ces requêtes. Le modèle « proc » est défini dans le fichier « proc.h » et son automate est représenté par le schéma ci-dessous.

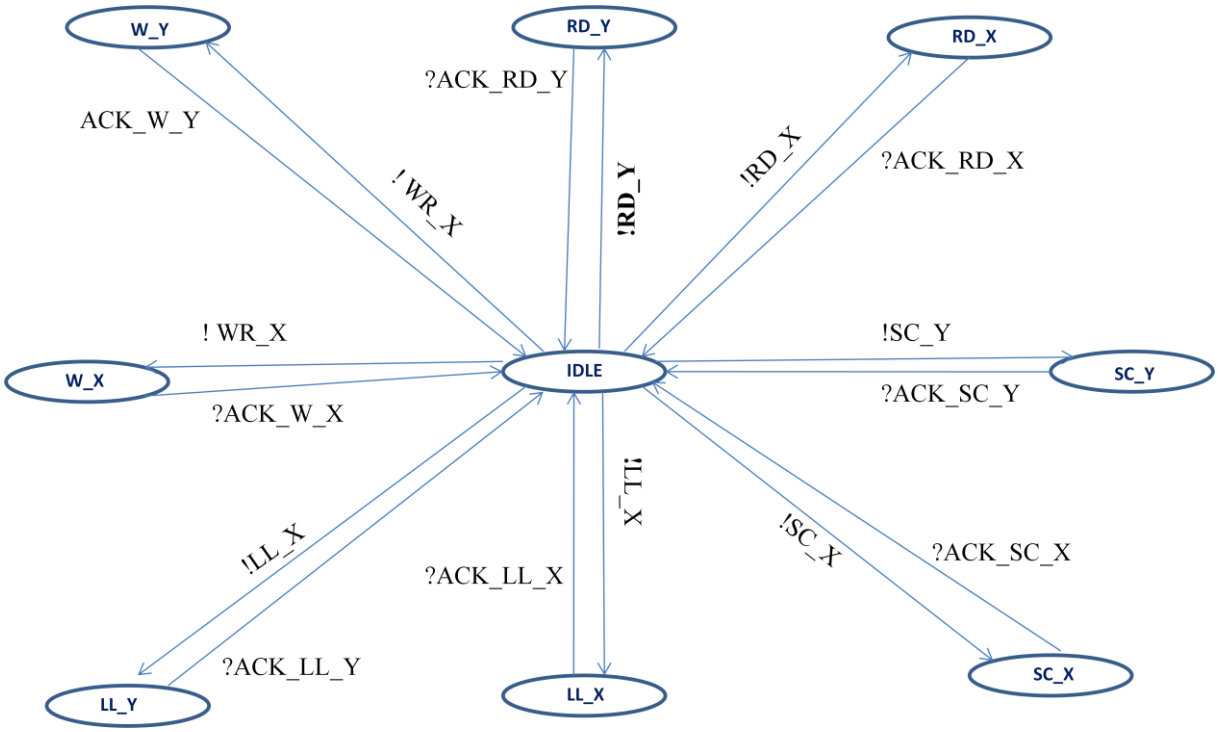


Figure 6 : Automate à états du processeur avec LL/SC.

On remarque bien qu'un processeur peut faire un SC en premier, c'est-à-dire non précédé d'un LL. Dans ce cas, le SC passe quand même jusqu'au contrôleur du cache L1 qui va renvoyer un échec vu que le qu'il n'y a de réservation qui à été faite avant.

Modèle du cache L1

J'ai réalisé un automate qui représente le comportement du Cache L1 dans la machine TSAR intégrant le mécanisme de LL/SC. Le modèle du cache L1 correspondant en PROMELA est dans le fichier « CacheL1.h » du dossier « PLAT_LL_SC ».

a- Prototype:

Proctype CacheL1 (chan PL1DTREQ, L1PDTACK, MCL1CPREQ, L1MCCPACK, L1MCCUREQ, MCL1CUACK; byte c_id);

Les canaux PL1DTREQ, L1PDTACK, MCL1CPREQ, L1MCCPACK, L1MCCUREQ et MCL1CUACK comme entrées/sorties de l'automate.

C_id : identifiant du cache.

b- Etats :

Pour modéliser cet automate, on définit d'abord les labels qui correspondent aux états, ici je présente les états correspondant au requêtes sur l'adresse X pour les avoir ceux correspondants à Y, il suffit de remplacer X par Y :

- **Empty** : Cache ne contient pas de données.
- **ATT_WR1_X** : Attente de l'acquittement de l'écriture de X. Etat associé à Empty.
- **ATT_LL1_X** : Attente de l'acquittement de LL de X. Etat associé à Empty.
- **ATT_LL2_X** : Attente de l'acquittement de LL de X.
- **CL_R_LL_X** : X est invalidé avant la réception de l'acquittement du LL.
- **V_LL_X** : Cache contient la valeur de l'adresse X et une réservation valide sur X.
- **ATT_SC_X** : Attente de l'acquittement de SC de X.
- **MISS_X** : Cache n'admet pas la donnée à l'adresse X.
- **ATT_D_X** : Attente de l'acquittement de la lecture de X.
- **CL_R_X** : X est invalidé avant la réception de l'acquittement.
- **V_X** : Cache contient la valeur de l'adresse X.
- **ATT_WR2_X** : Attente de l'acquittement de l'écriture de X. Etat associé à V_X (Cache contient X).
- **ATT_WR3_X**: Attente de l'acquittement de l'écriture de X. Etat associé à V_Y (Cache contient Y).

c- Variables internes :

Cette machine à état admet des variables locales qui servent à tester les requêtes, et une case mémoire qui contient la copie d'une adresse mappée en mémoire.

- Byte v_cache : Case mémoire interne qui contient la valeur d'une adresse (X ou bien Y).

- Bit v_cache_valide : bit de validité de la case mémoire interne.

- Bit v_addr : variable qui contient l'adresse associée à la donnée dans le cache.

- Bit Vcl : bit de validité du CLNUP. Ce bit interdit la soumission d'un deuxième CLNUP sur une adresse avant la réception de l'acquittement du premier.

- Msg m: structure d'un message, qui sert aux requêtes de cohérence (INV, UP et CL_UP).

- Msg1 m1 : structure d'un message, qui sert à la transmission des RD/WR/LL/SC dans le canal dédié « L1MCDTREQ ». C'est le type de message qui nous permet d'envoyer à la fois l'adresse de réservation, la valeur à écrire et la valeur lue pour un SC du cache L1 vers le contrôleur mémoire.

Ces variables internes peuvent être modifiées à chaque transition possible.

d- Automate à états :

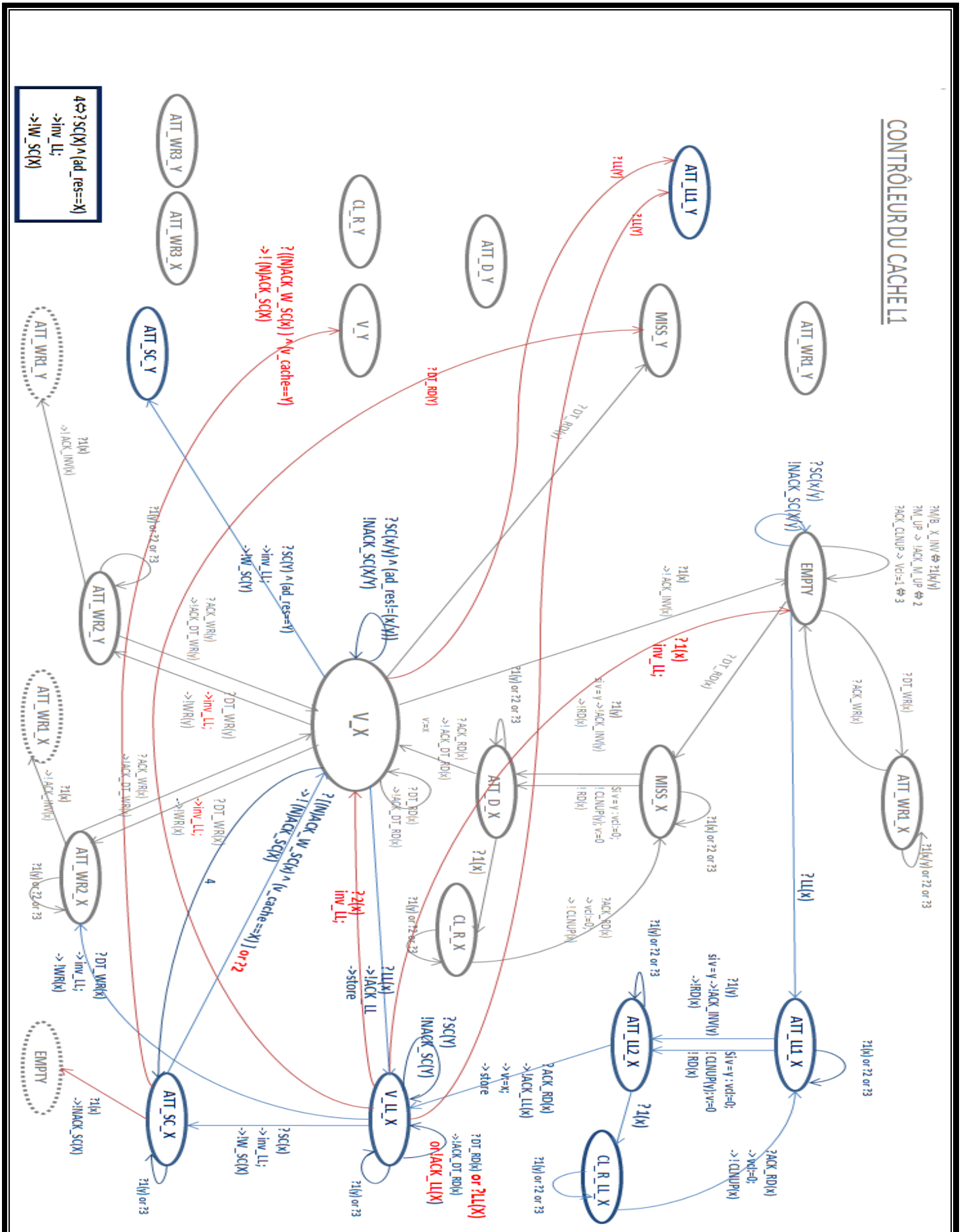


Figure 7 : Automate du cache L1 avec LL/SC intégré. [Document : Cache_L1_LL_SC.pptx]

Modèle du Contrôleur mémoire

Pour le contrôleur mémoire, je réalise 2 modèles qui fonctionnent en parallèle dans le système, un pour l'adresse X et l'autre pour Y intégrant le mécanisme de lecture/écriture atomique.

a- Prototype :

Proctype MC_X (chan L1MCDTREQ, MCL1DTACK, L1MCCUREQ, MCL1CUACK, MCL1CPREQ, L1MCCPACK, MCMEMDTREQ, MEMMCDTACK) ;

Proctype MC_Y (chan L1MCDTREQ, MCL1DTACK, L1MCCUREQ, MCL1CUACK, MCL1CPREQ, L1MCCPACK, MCMEMDTREQ, MEMMCDTACK) ;

Les canaux L1MCDTREQ, MCL1DTACK, L1MCCUREQ, MCL1CUACK, MCL1CPREQ, L1MCCPACK, MCMEMDTREQ et MEMMCDTACK sont les entrées/ sorties de l'automate de chaque modèle.

b- Etat :

Je décris les différents états de l'automate du modèle pour l'adresse X. Et pour le modèle de Y, il suffit de remplacer « X » par « Y ».

- **Empty_X** : Le contrôleur mémoire n'admet pas la valeur de l'adresse mémoire X.
- **ATT_WR_X** : Etat d'attente de lecture d'un bloc mémoire. Le contrôleur mémoire reçoit une écriture sur l'adresse X, et dans cet état une demande du bloc mémoire qui contient X est effectuée.
- **ATT_RD_X** : Etat d'attente de lecture d'un bloc mémoire. Le contrôleur mémoire reçoit une lecture sur l'adresse X, et dans cet état une demande du bloc mémoire qui contient X est effectuée.
- **ATT_RD_WR_X** : Etat d'attente de lecture d'un bloc mémoire. Le contrôleur reçoit une écriture après une lecture qui n'a pas encore été servie par la mémoire.
- **ATT_SC1_X :** Etat de demande de la valeur présente en mémoire lors d'un SC. Le contrôleur reçoit une requête de SC sur X et fait un GET de la valeur présente à l'adresse X.
- **ATT_SC2_X :** Etat d'attente l'acquittement du GET de la valeur présente en mémoire. Le contrôleur reçoit ACK_GET(X). il compare la valeur lue avec la valeur récupérée en mémoire et acquitte le cache L1.

NB : Pour des raisons de visibilité de l'automate du contrôleur mémoire, on décidé avec l'accord de mon encadrant de ne présenter qu'une seule fois le traitement d'un SC et pour un seul état stable en l'occurrence **V_M_X.**

Pour le reste des états stables (**V_M_D_X, V_B_X** et **V_B_D_X)** le traitement et similaire que celui de l'état représenté. Donc il suffit de dupliquer les deux états **ATT_SC1_X** et **ATT_SC2_X** avec les transactions correspondantes pour chacun de ces états.

- **V_M_D_X** : Le contrôleur mémoire admet une copie de la valeur de l'adresse mémoire X qui n'est pas propre (dirty : une écriture été effectuée sur cette adresse et le contrôleur mémoire n'a pas encore écrit en mémoire « Write-back»). Le contrôleur mémoire est en état multicast pour le protocole de cohérence.
- **V_M_X** : Le contrôleur mémoire admet une copie de la valeur de l'adresse mémoire X qui est propre (aucune écriture n'a été effectuée sur cette adresse). Le contrôleur mémoire est en état multicast pour le protocole de cohérence.
- **ATT_UP_X** : Etat d'attente des acquittements des caches après un multicast update. C'est l'état transitoire entre V_M_X et V_M_D_X, c.à.d. une écriture à été reçue par le contrôleur mémoire après un nombre limité de lecture qui garde multicast comme protocole de cohérence.
- **V_B_X** : Le contrôleur mémoire admet une copie propre de la valeur de l'adresse mémoire X. Le contrôleur mémoire est en état broadcast pour le protocole de cohérence.
- **V_B_D_X** : Le contrôleur mémoire admet une copie dirty de la valeur de l'adresse mémoire X. Le contrôleur mémoire est en état broadcast pour le protocole de cohérence.
- **ATT_B_INV_X** : Etat d'attente des acquittements des caches après un broadcast d'invalidation.
- **ATT_PUT_X** : Etat d'attente de l'acquittement de l'écriture du bloc en mémoire.

c- Variable interne :

- **Byte src** : Variable temporelle pour sauvegarder l'identifiant du cache source avant de l'ajouter au tableau de copie.
- **Byte v_x** : valeur de la case mémoire d'adresse X.
- **Byte c_id_tmp** : Variable temporelle pour sauvegarder l'identifiant du cache.
- **Byte cpt** : compteur interne.
- **Byte cpt_rep** : Compteur de réponse en cas de M_UP et B_INV.
- **Byte c_id[CACHE_TH]** : tableau qui sauvegarde les identifiants des caches qui ont une copie de X en cas de multicast.
- **Bit v_c_id[CACHE_TH]** : bit de validité de l'identifiant sauvegardé dans c_id.
- **Byte n_cache** : compteur de copie valide dans les caches.

d- Automate à états :

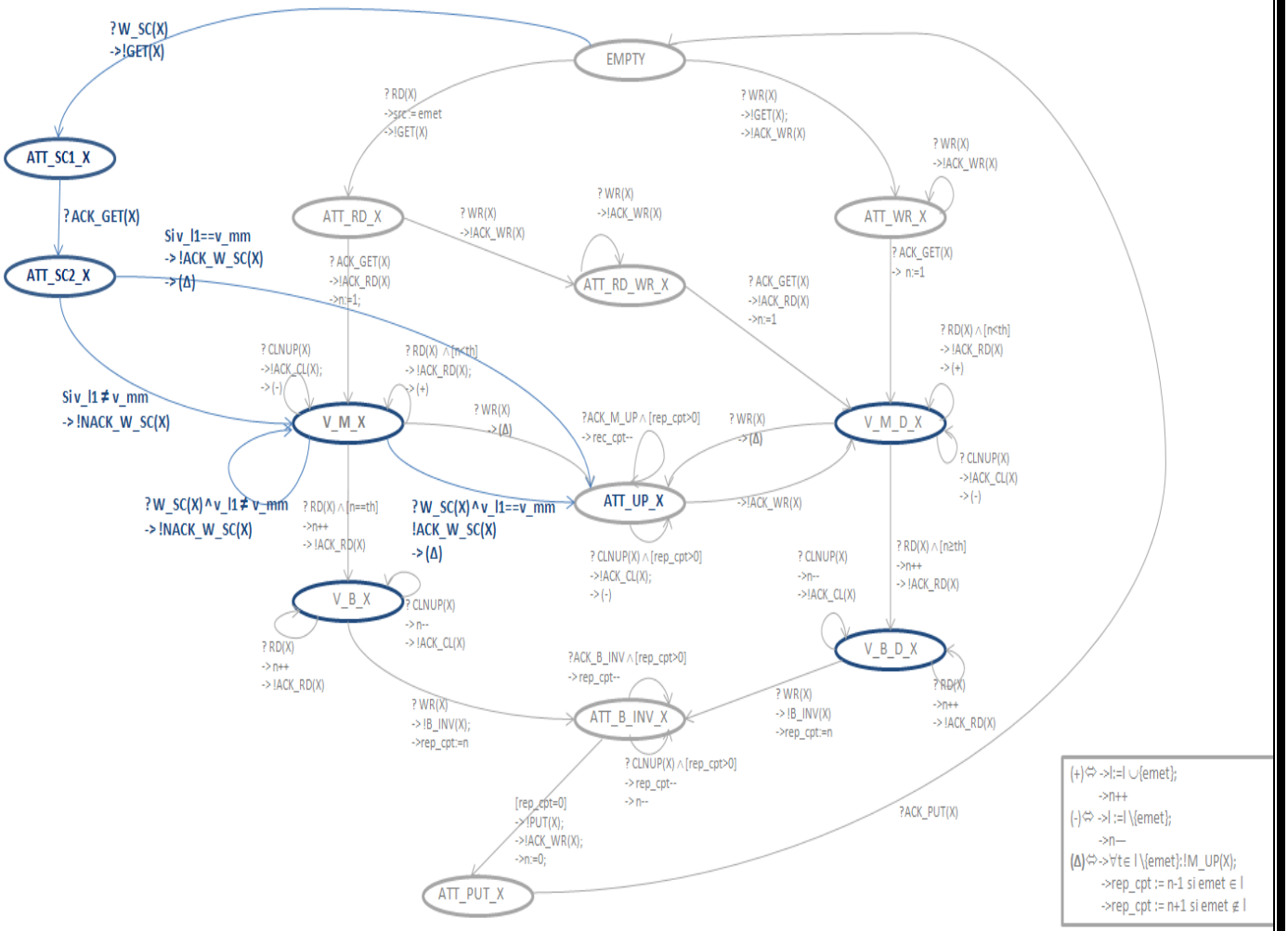


Figure 8 : automate du contrôleur mémoire avec LL/SC intégré. [Document : controleur_memoire_LL_SC.pptx]

5. test du cache L1 avec LL/SC :

Pour valider le modèle créé, je teste le comportement de l'automate décrit avec une série de tests qui implémente plusieurs scénarii possibles. Ces test consistent à tester le Cache L1 dans ses différentes états (vide, miss, hit, LL, SC…) dans une plateforme monoprocesseur. Pour cela j'ai modélisé un processeur qui demande les requêtes précisées dans les tableaux qui viennent, et j'ai remplacé le contrôleur mémoire par un modèle qui répond par des acquittements sur les requêtes du Cache L1 comme le montre la figure 9.

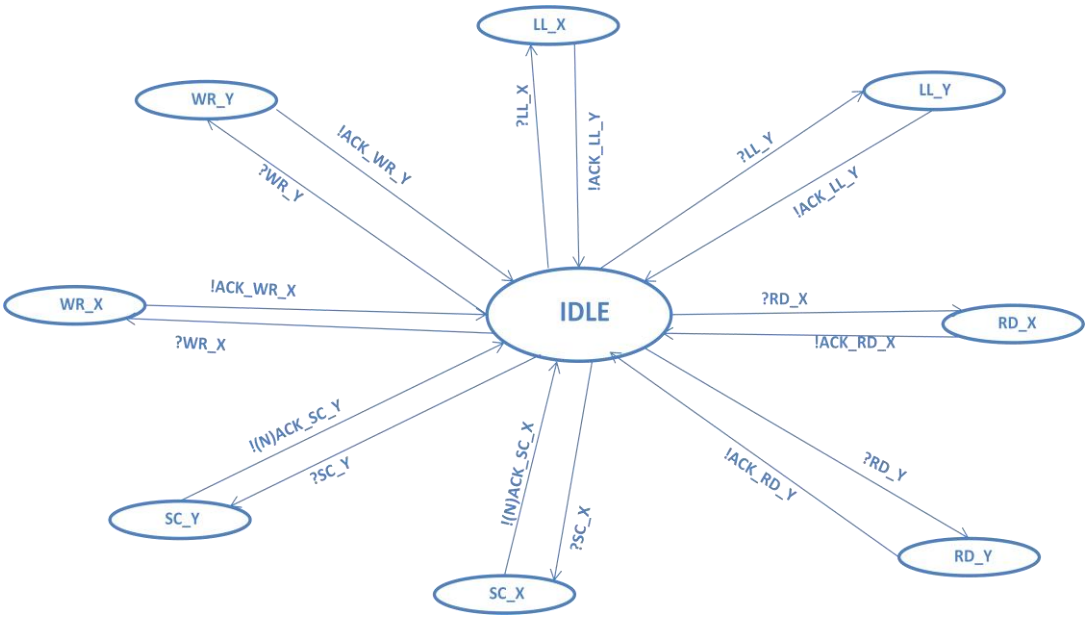


Figure 9: Automate du contrôleur mémoire test.

Chaque tableau qui suit représente une suite d'instructions particulières envoyées par le processeur pour vérifier une certaine propriété du mécanisme LL/SC dans le cache L1. Et à chaque instruction, je vérifie les transactions qui sont franchies, la succession des états dans chaque composant (processeur, cacheL1, contrôleur mémoire) ainsi que l'état des variables globales et locales pour vérifier la cohérence des résultats obtenus.

Test1 :

T	PL1DTREQ	Succession des états cache L1	LL_addr	V	L1PDTACK
0	Read(X)	Empty->MISS(X) -> ATT_D(X) ->V(X)	EMPTY	X	ACK_RD
1	LL(X)	V_LL(X)	X	X	ACK_LL
2	SC(X)	V_LL(X) ->ATT_SC(X) ->V(X)	EMPTY	X	ACK_SC
3	Read(Y)	V_LL(X) ->MISS(Y) -> ATT_D(Y) ->V(Y)	EMPTY	Y	ACK_RD
4	Read(X)	V(X) ->MISS(X) -> ATT_D(X) ->V(X)	EMPTY	X	ACK_RD

Ce test à été mis en œuvre pour pouvoir vérifier qu'un SC fait succès après un LL sur la même adresse et le cache est non vide.

Remarque :

Un test1_bis à été aussi fait pour vérifier la même propriété mais avec le cache vide (i.e. à l'état Empty)

Pour voir les résultats obtenus, vous simulez les fichiers « test_LL_SC1.pml » et « test_LL_SC1_bis.pml » avec l'outil spin.

Test2 :

T	PL1DTREQ	Succession des états cache L1	LL_addr	V	L1PDTACK
0	Read(X)	Empty->MISS(X) -> ATT_D(X) ->V(X)	EMPTY	X	ACK_RD
1	LL(X)	V(X)->V_LL(X)	X	X	ACK_LL
2	RD(Y)	V_LL(X) -> MISS(Y) -> ATT_D(Y) ->V(Y)	X	XY	ACK_RD
3	SC(X)	V_(Y) -> ATT_SC(X) ->V(Y)	EMPTY	Y	ACK_SC
4	Read(X)	V(X) ->MISS(X) -> ATT_D(X) ->V(X)	EMPTY	X	ACK_RD

Ce test à été mis en œuvre pour pouvoir vérifier qu'un Read après un LL n'influence pas sur le résultat du SC. Et grâce à ce test, j'ai pu découvrir qu'il ya une incohérence dans les résultats. Le problème c'est que quand on fait un SC sur X et que le cache était dans V_Y, on distingue deux cas :

- Si v_cache = X : Dans ce cas là on envoie un ACK_SC et en fait un goto V_X après la mise à jours du cache.
- Si v_cache = Y : Dans ce cas, on revient à l'état V_Y.

Ce qui m'a amené à apporter des modifications dans l'automate du cache L1 et son implémentation PROMELA.

Pour voir les résultats obtenus, on simule le ficher « test_LL_SC2.pml » avec l'outil spin.

Test3 :

T	PL1DTREQ	Succession des états cache L1	LL_addr	V	L1PDTACK
0	Read(X)	Empty->MISS(X) -> ATT_D(X) ->V(X)	EMPTY	X	ACK_RD
1	LL(X)	V(X)->V_LL(X)	X	X	ACK_LL
2	WR(X)	V_LL(X) ->ATT_WR2(X) ->V(X)	EMPTY	X	ACK_WR
3	SC(X)	V(X) ->V(X)	EMPTY	X	ACK_SC
4	Read(X)	V(X) ->V(X)	EMPTY	X	ACK_RD

Ce test à été fait pour pouvoir vérifier qu'un Write après un SC invalide le LL. Pour voir les résultats obtenus, on simule le ficher « test_LL_SC3.pml » avec l'outil spin.

Test4 :

T	PL1DTREQ	Succession des états cache L1	LL_addr	V	L1PDTACK
0	Read(X)	Empty->MISS(X) -> ATT_D(X) ->V(X)	EMPTY	X	ACK_RD
1	LL(X)	V(X) -> V_LL(X)	X	X	ACK_LL
2	SC(Y)	V_LL(X) -> V_LL(X) (échec)	EMPTY	X	ACK_SC
3	SC(X)	V_LL(X) -> ATT_SC(X) ->V(X) (succès)	EMPTY	Y	ACK_RD
4	SC(X)	V(X) ->V(X) (échec)	EMPTY	Y	ACK_WR

Ce test à été fait pour pouvoir vérifier qu'un SC qui vient après un SC sur la même adresse, fait échec. Pour voir les résultats obtenus, on simule le ficher « test_LL_SC4.pml » avec l'outil spin.

6. Conclusion :

La réalisation de ce travail m'a donné l'occasion d'acquérir de nouvelles connaissances et d'en approfondir d'autres sur la modélisation des systèmes basée sur l'implémentation de composants définis par des automates états. Il m'a permis aussi de me familiariser avec le model checker SPIN ainsi que le langage d'implémentation PROMELA. J'ai pu aussi découvrir et travailler durant ce projet sur de grosses machines, comme le serveur de calcul « rythm » du réseau de recherche de l'UPMC. Un travail supplémentaire sur ce projet est nécessaire pour implémenter en PROMELA de l'automate du contrôleur mémoire avec LL/SC et terminer les tests et les vérifications en utilisant des outils plus adaptés et qui utilisent des techniques de réduction de l'espace d'états d'un système come DVIN et SDD.

Après mon travail sur ce projet, je fournis :

- En première partie :

  - Les Automates graphiques pour le cache L1 ainsi que le contrôleur mémoire.

  - Les modèles simplifiés en langage PROMELA du cache L1, des Contrôleur mémoire pour X et pour Y et des composants minimaux (processeur et mémoire).

  - Les résultats des vérifications avec l'outil SPIN pour l'analyse de deadlocks.

- En deuxième partie :

  - Les automates graphiques avec mécanisme du LL/SC intégré du cache L1, du contrôleur mémoire et du processeur.

  - Les modèles du cache L1 ainsi que les mini-modèles (processeur et mémoire) en langage PROMELA.

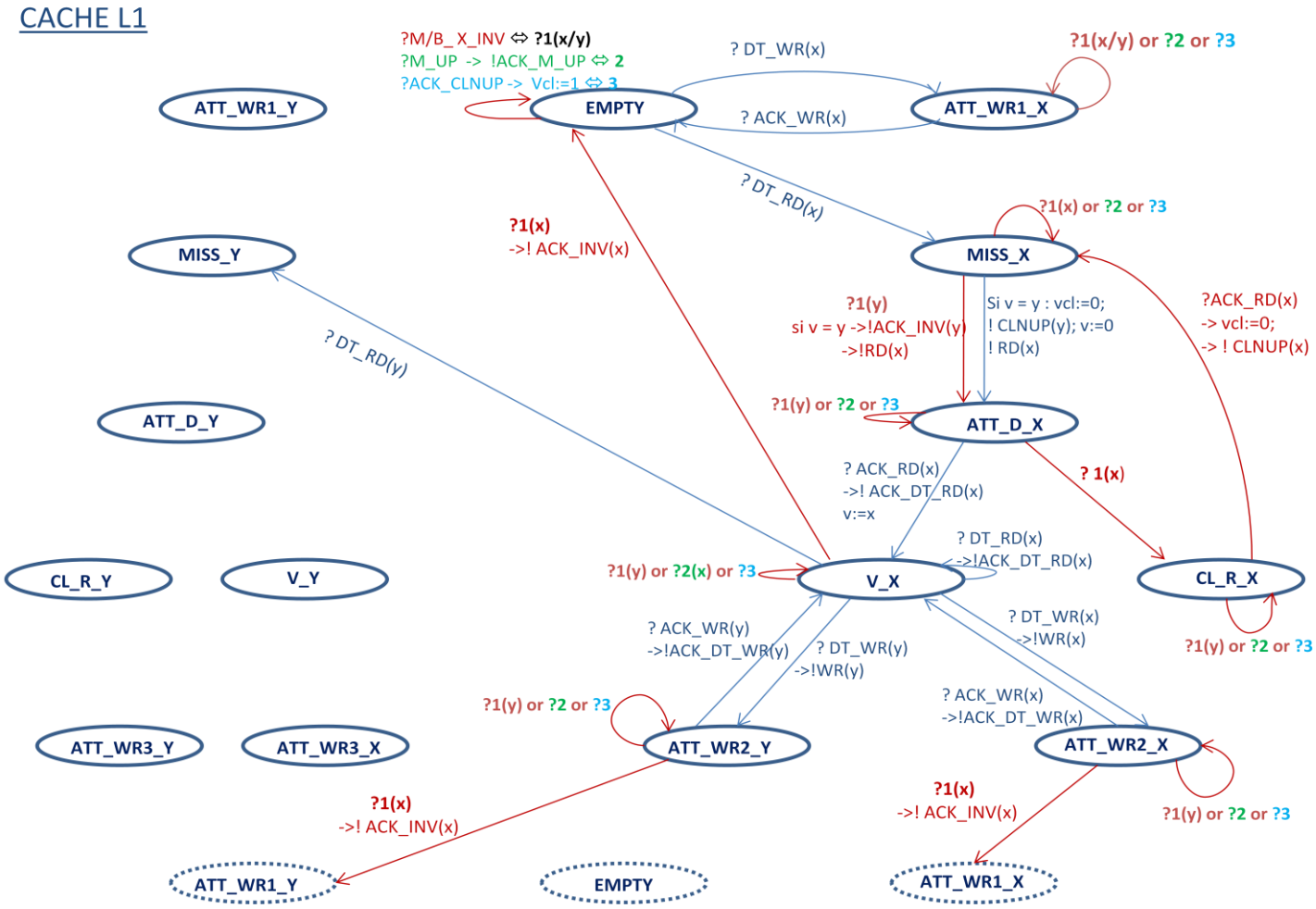
  - Les tests effectués sur le modèle du cache L1 avec LL/SC.

Ainsi, j'espère que mon travail sera une bonne base pour une éventuelle suite pour ce projet.

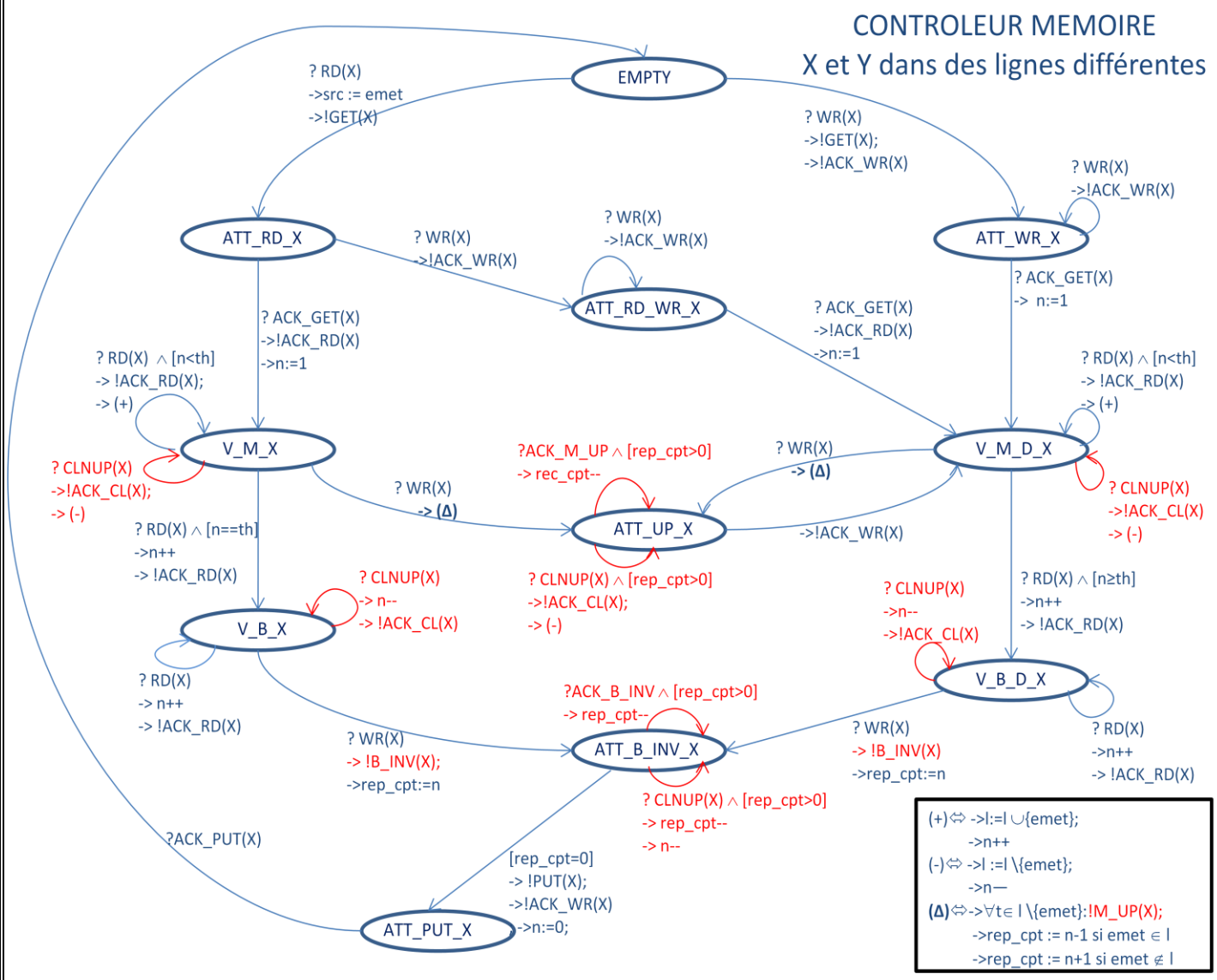
Annexe :

Les fichiers qui contiennent les modèles et les tests réalisés avec les vérifications se trouvent dans le répertoire « PSESI_TSAR_2012 » fourni avec le rapport. Vous trouvez dans ce répertoire les sous répertoire ci-dessous :

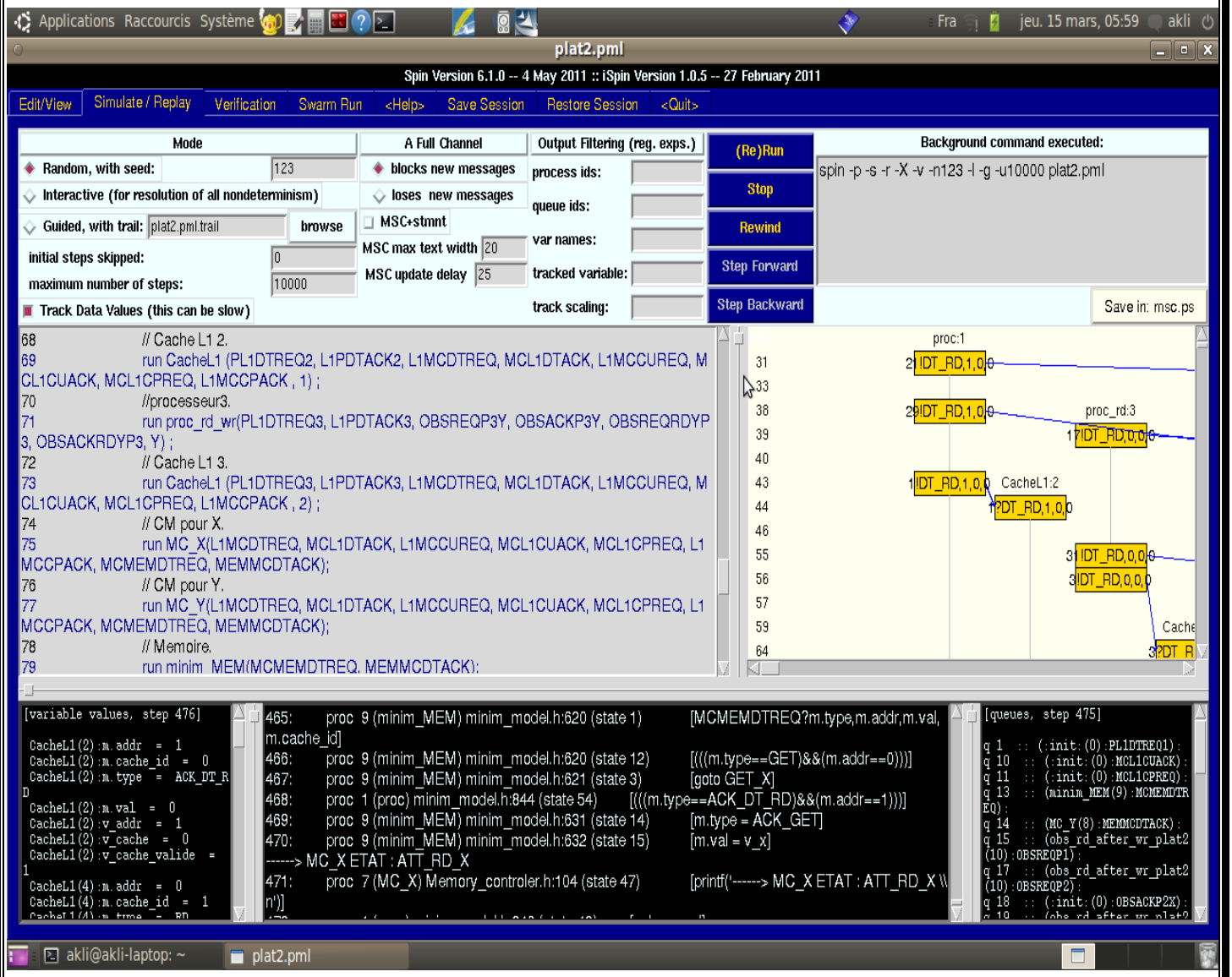
- Automates : Ce répertoire contient tous les automates réalisés sous forme de document (.pptx).
- Plateformes : Ce répertoire contient toutes les plateformes réalisées durant ce projet. Elles sont rangées dans des dossiers dont les noms sont mnémoniques. Vous obtenez les résultats des tests et des vérifications en simulant les fichiers (.pml).
- Test_LL_SC qui contient les répertoires suivants :
  - Test_LL_SC1 : Ce répertoire contient les tests réalisés sur le modèle du cache L1 avec le mécanisme du LL/SC intégré pour la suite d'instructions décrite dans ce rapport dans 'test1'. Ce modèle se trouve dans le fichier « CacheL1.h » qui est inclus dans le fichier «test_plat_LL_SC1.pml ». Vous obtenez les résultats des tests en simulant ce fichier.
  - Test_LL_SC1_bis : Ce répertoire contient les tests réalisés sur le modèle du cache L1 avec le mécanisme du LL/SC intégré pour la suite d'instructions décrite dans ce rapport dans 'test1'. Ce modèle se trouve dans le fichier « CacheL1.h » qui est inclus dans le fichier «test_plat_LL_SC1_bis.pml ». Vous obtenez les résultats des tests en simulant ce fichier.
  - Test_LL_SC2 : Ce répertoire contient les tests réalisés sur le modèle du cache L1 avec le mécanisme du LL/SC intégré pour la suite d'instructions décrite dans ce rapport dans 'test1'. Ce modèle se trouve dans le fichier « CacheL1.h » qui est inclus dans le fichier «test_plat_LL_SC2.pml ». Vous obtenez les résultats des tests en simulant ce fichier.
  - Test_LL_SC3 : Dans ce répertoire vous avez le test de la 3ème suite d'instructions réalisés sur le modèle du cache L1 avec le mécanisme du LL/SC intégré. Vous obtenez les résultats des tests en simulant le fichier « test_plat_LL_SC3.pml ».
  - Test_LL_SC4 : Dans ce répertoire vous avez le test de la 4ème suite d'instructions réalisés sur le modèle du cache L1 avec le mécanisme du LL/SC intégré. Vous obtenez les résultats des tests en simulant le fichier « test_plat_LL_SC4.pml ».



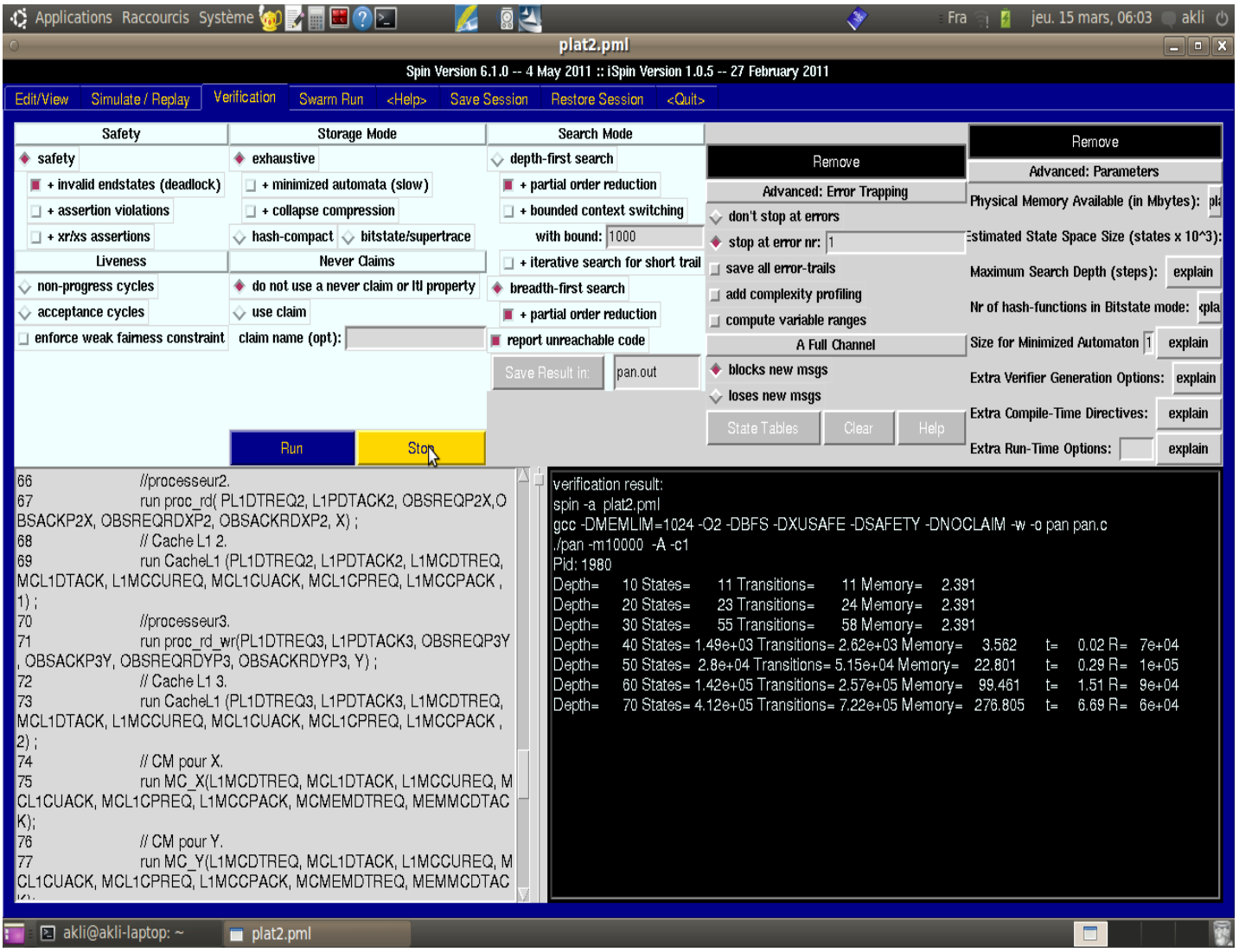
Annexe 1 : Automate à états du Cache L1 sans LL/SC



Annexe 2 : Automate à états du contrôleur mémoire (MC) sans LL/SC.



Annexe 3 : Interface graphique de spin en mode simulation.



Annexe 4 : Interface graphique de spin en mode vérification.