UNIVERSITE D'EVRY VAL D'ESSONNE

Département d'Informatique

Rapport de stage de fin d'études

Master 2 MOPS

Vérification compositionnelle du Protocole de Cohérence de Cache de la Machine Multiprocesseur TSAR (Tera-Scale Architecture)

Proposé par :

Laboratoire LIP 6

Encadrants:

Mme Emmanuelle ENCRENAZ

M. Quentin MEUNIER

M. Yann THIERRY-MIEG

Réalisé par :

Melle Zahia GHARBI

Période :

25/03/2013 – 25/09/2013

REMERCIEMENTS

Je remercie en premier lieu mes encadrants Mme Emmanuelle ENCRENAZ, M. Quentin MEUNIER et M. Yann THIERRY-MIEG, pour leur disponibilité, leur patience et leurs précieux conseils qui m'ont été très utiles.

Je remercie également tous les enseignants chercheurs du Laboratoire LIP6 pour leur accueil et leur générosité.

Mes vifs remerciements vont également aux membres de jury pour avoir accepté d'évaluer mon travail.

Ma profonde reconnaissance à tous ceux qui ont contribué, de près ou de loin, au bon déroulement de mon stage.

Table des matières

Table des figures

Liste des tableaux

I. Introduction

Dans le cadre de ma formation de Master 2 « Modèles, Optimisation, Programmation et Services (MOPS) », à l'université d'Evry Val d'Essonne, j'ai effectué un stage de fin d'études d'une durée de 6 mois, du 25 Mars au 25 Septembre 2013, au sein du Laboratoire d'Informatique de l'Université de Paris 6 (LIP6).

Ma mission principale, durant ce stage, consiste en la modélisation et la vérification du protocole de cohérence de cache implanté dans une machine multiprocesseur, développée au LIP6. Il s'agit de modéliser le comportement des composants élémentaires (processeur, cache L1, contrôleur mémoire, mémoire et réseaux de communication) vis-à-vis des accès mémoire, puis prouver formellement avec des outils de vérification que le protocole de communication de cette architecture garantit bien les propriétés de cohérence (si un bloc mémoire a été modifié alors il faut mettre à jour toutes ses copies qui se trouvent dans les caches) et qu'il est sans blocage (le système pourra toujours traiter une nouvelle requête de lecture ou d'écriture). L'outil de vérification que j'ai utilisé est DIVINE (logiciel que je ne connaissais pas avant ce stage).

Dans ce rapport, je vous présenterai dans un premier temps le laboratoire d'accueil, puis dans un second temps, le contexte, l'objectif et le déroulement des phases de mon stage.

II. Laboratoire d'accueil et contexte du stage

II.1. Laboratoire d'accueil

Le Laboratoire d'Informatique de l'Université de Paris 6 (LIP6) a été créé en Janvier 1997 par la fusion des anciens laboratoires d'informatique de Paris 6, dont le LAFORIA, le MASI et une partie du LITP. Le laboratoire LIP6 est rattaché à l'Université Pierre et Marie Curie (Paris 6) et au Centre National de la Recherche Scientifique (CNRS). Il regroupe la quasi-totalité de la recherche en informatique de l'Université Pierre et Marie Curie (UPMC) et avec près de 400 chercheurs permanents et doctorants, il constitue l'un des plus importants laboratoires français d'informatique.

Le Laboratoire LIP6 est situé à Jussieu et est structuré en 5 départements, dont le département SOC qui regroupe les recherches portant sur le développement des systèmes intégrés sur puce et embarqué. L'équipe ALSOC fait partie de ce département et est située à la Maison de la Pédagogie où j'effectue mon stage pratique.

Mme Emmanuelle ENCRENAZ est la responsable de l'équipe ALSOC (Architecture et Logiciel pour Systèmes Embarqués sur Puce). Cette équipe concentre l'ensemble de ses recherches sur le développement de méthodes et outils pour la conception et l'utilisation d'architecture de processeurs many-cores intégrés sur puce. On retrouve parmi les membres de cette équipe M. Quentin MEUNIER qui y travaille en tant que permanent.

M. Yann THIERRY-MIEG est membre de l'équipe MoVe (Modélisation et Vérification) qui fait partie du département RSR (Réseaux et Systèmes Répartis). Ce département regroupe les recherches portant sur l'analyse et la conception de solutions pour construire et gérer les réseaux, les systèmes et les applications du futur. L'équipe MoVe est située au Site Jussieu tour 25-26, elle concentre ses recherches sur la modélisation et l'analyse de systèmes répartis complexes et dynamiques en particulier sur les techniques optimisées de vérification formelle par Model Checking.

II.2. Contexte du stage

L'équipe ALSOC travaille depuis des années sur une machine multiprocesseurs à mémoire partagée appelée TSAR, plus particulièrement sur son protocole de cohérence de caches nommé : DHCCP « Distributed Hybrid Cache Coherence Protocol ». Plusieurs caches peuvent contenir simultanément la même copie d'un bloc mémoire. Si un processeur réalise une écriture sur ce bloc, il faut alors assurer la cohérence de toutes les autres copies soit en les invalidant, soit en les mettant à jour. Vu la complexité de ce protocole, des défauts de conception peuvent être introduits, d'où la nécessité de prouver formellement que ce protocole garantit bien les propriétés de cohérence et qu'il ne contient pas de blocage.

La vérification formelle de tels protocoles est généralement effectuée par la technique de model-checking, qui consiste à analyser la structure du graphe des états accessibles du système afin de déterminer si ce dernier satisfait des propriétés logico-temporelles caractérisant son bon fonctionnement. Cette technique présente l'avantage d'être automatique une fois les éléments du protocole modélisés, mais se heurte au problème d'explosion combinatoire de la taille du graphe des états accessibles. En pratique, les protocoles sont analysés à un haut niveau d'abstraction, ce qui masque des erreurs liées à l'implémentation.

Des projets de recherche ont été faits autour de la vérification du protocole de cohérence de la machine TSAR mais qui ont donné des résultats incomplets. En effet, l'année dernière M. Akli MANSOUR a pu vérifier l'absence de blocage sur une plateforme qui intègre 2 processeurs dans un scénario particulier, mais la vérification d'une plateforme plus complète

n’a pas pu aller jusqu’à son terme et ce à cause du phénomène de l’explosion combinatoire du nombre d’états. Il est à noter que pour activer tous les mécanismes du protocole de communication de la machine multiprocesseur TSAR et donc pour assurer la cohérence de caches et l’absence de blocage, la présence d’au moins 3 processeurs est nécessaire dans la plateforme.

Mon stage de fin d’études concerne la vérification des propriétés de cohérence et d’absence de blocage en utilisant l’outil de vérification DIVINE (Model-Checking) sur une architecture qui intègre un faible nombre de composants (3 processeurs et donc 3 caches L1, un contrôleur mémoire à 2 adresses appelées X et Y, et une mémoire à 2 adresses aussi) en me basant sur les modèles déjà décris par Mr Akli MANSOUR sous PROMELA.

II.3. Présentation du projet TSAR

TSAR (Tera-Scale Architecture) est une machine multiprocesseur définie dans le cadre d'un projet Européen coordonné par la société BULL. Cette machine intègre jusqu'à 4096 processeurs répartis sur 1024 clusters, soit 4 processeurs par cluster, interconnectés par un réseau de communication distribué (DSPIN) avec une mémoire d'une taille maximale de 1 Téraoctet (40 bits d'adressage physique) répartie sur l'ensemble des clusters, soit 1 Gigaoctet par cluster.

L’architecture définie est à mémoire physiquement distribuée et logiquement partagée : chaque processeur peut accéder aux données stockées sur le cache L2 (contrôleur mémoire) de son cluster mais également au cache L2 de n’importe quel autre cluster, ce qui implique que le temps d'accès aux données et instructions est variable. Les données sont répliquées dans les caches locaux (L1) des processeurs qui en ont fait la demande. La cohérence des différentes copies est assurée en matériel au travers d’un protocole original, DHCCP, développé au LIP6 et détaillé plus loin. Des prototypes virtuels ont été développés et permettent de simuler des architectures intégrant jusqu’à 512 processeurs et exécutant une pile logicielle. Un prototype matériel à 16 processeurs a été réalisé sur FPGA.

Après plusieurs mois d’exploitation de la plateforme virtuelle, un deadlock est apparu lors des simulations. L’analyse de la simulation produisant le deadlock et sa correction étant complexe, la vérification du protocole devient cruciale.

En annexe 1, j’ai représenté la machine TSAR avec 2 clusters (4 processeurs pour chacun) reliés par un réseau d'interconnexion où :

- le processeur 0 du cluster 1 envoie une demande de lecture du bloc mémoire d'adresse 5, ce dernier ne se trouvant pas dans la Mémoire du cluster 1 la requête est envoyée via le réseau global au Directory du cluster 0. Le Directory vérifie si le bloc demandé est chargé dans la ligne du contrôleur mémoire. Comme ce n'est pas le cas, il envoie un message à la Mémoire locale qui lui répond en chargeant le bloc demandé dans la ligne du contrôleur mémoire. Un message d'acquittement (contenant la donnée) est envoyé du contrôleur mémoire au cache L1 du processeur 0 (cluster 1).
- Le processeur 1 du cluster 0 envoie une demande de lecture du bloc mémoire d'adresse 1 qui est géré par le directory du cluster 0. La requête est envoyée au Directory du même cluster. Le Directory vérifie et trouve que le bloc demandé n'est pas chargé dans la ligne du contrôleur mémoire du même cluster, il envoie un message à la mémoire qui répond en chargeant le bloc demandé dans la ligne du contrôleur mémoire (les deux blocs d'adresses 1 et 5 se trouvent sur deux lignes différentes). Un message d'acquittement est envoyé au processeur 1 du cluster 0 via son cache L1.

III. Ma mission

III.1. Sujet de stage

Afin d'assurer le bon fonctionnement du protocole de cohérence de caches de la machine multiprocesseur TSAR, mon travail consiste en :

- La compréhension de l'architecture TSAR et du protocole DHCCP.
- L'identification des niveaux hiérarchiques du protocole. Modélisation des entités constitutives du protocole (Processeur, Cache L1, Contrôleur Mémoire, Mémoire et les différents réseaux de communication).
- La construction d'une plateforme permettant l'activation de tous les mécanismes du protocole.
- L'identification des propriétés à vérifier (sûreté et vivacité).
- La vérification des propriétés et analyse des résultats.
- Mise en évidence du deadlock, modélisation de sa correction et vérification de la correction.

III.2. Déroulement des phases du stage

III.2.1. Présentation du protocole de cohérence de la machine multiprocesseur TSAR

Les réalisateurs du projet TSAR ont défini un protocole à répertoire [1] pour assurer la cohérence des données dans les caches. On a un répertoire par cluster chargé de maintenir la cohérence des données implantées dans la plage d'adresse dédiée au cluster. A chaque entrée du répertoire, on associe un bloc de la mémoire et chaque répertoire conserve pour chaque bloc mémoire les informations suivantes :

- La validité de la ligne,
- Modification de la ligne par rapport à la mémoire,
- Le nombre de copies du bloc mémoire présentes dans les caches L1,
- Les identifiants de tous les caches qui ont des copies du bloc (le nombre maximum d'identifiants stockés est borné par un seuil).

Pour localiser une copie d'un bloc mémoire on a besoin d'un bit par processeur. Comme on a 4096 processeurs on doit avoir au total 4096 bits pour localiser toutes les copies d'un bloc mémoire dans la machine multiprocesseur TSAR, ce qui est très coûteux en espace mémoire c'est pourquoi les réalisateurs du projet TSAR ont défini le protocole DHCCP « Distributed Hybrid Cache Coherence Protocol » qui permet la localisation d'un nombre limité de copies. Ce nombre est paramétrable et défini par la constante TH, qui représente le nombre maximum de copies d'un même bloc mémoire qu'on peut localiser dans cette machine.

Il existe deux modes de maintien de la cohérence [2] selon le nombre de copies actuelles par rapport au seuil TH :

1. Si le nombre de copies est inférieur ou égal à TH : La cohérence est maintenue par *mise à jour*. L'entrée du répertoire associée au bloc mémoire modifié contient les identifiants de tous les caches qui en ont une copie valide. Un message de mise à jour (Multicast Update) est envoyé à tous ces caches pour mettre à jour leurs lignes.

2. Si le nombre de copies est supérieur à TH : La cohérence est maintenue par *invalidation*. L'entrée du répertoire associée au bloc mémoire modifié ne garde pas les identifiants de tous les caches qui en ont une copie valide. Un message d'invalidation (Broadcast invalidate) est

envoyé à tous les caches du système et à réception de ce message, seuls les caches qui ont une copie de ce bloc l'invalident. Seul le Contrôleur Mémoire contient la donnée à jour.

En général, la transmission des écritures d'un niveau de la hiérarchie vers le niveau supérieur peut suivre la stratégie *write-through* ou la stratégie *write-back* [1]. Dans la stratégie write-through, la nouvelle valeur de la donnée est immédiatement transmise au niveau de la hiérarchie supérieure, et éventuellement copiée dans la copie locale si cette dernière existe. Dans la stratégie write-back, la nouvelle valeur de la donnée est écrite uniquement dans le cache, et ne sera répercutée en mémoire que lorsque la ligne du cache sera invalidée ou évincée, ce qui implique que le contrôleur mémoire gérant cette ligne doit connaitre en plus de la localisation, l'état de la ligne (MESI : Modified, Exclusive, Shared et Invalid).

La stratégie write-through garantit la cohérence des données car les caches ont toujours les mêmes données que la mémoire. Elle est plus simple à mettre en œuvre et est privilégiée quand il y a peu d'écritures et beaucoup de lectures concurrentes, et ce pour réduire le nombre de messages de mise à jour envoyés dans le réseau pour chaque écriture (le nombre de messages envoyés est proportionnel au nombre de caches). L'avantage de la stratégie write-back, est le fait qu'à la fin d'une série d'écriture dans la même ligne une seule se fait en mémoire, ce qui réduit les accès mémoire et ainsi éviter d'encombrer le réseau. Mais cette technique est plus compliquée à mettre en œuvre car il faut garder au niveau de la mémoire plus d'information par rapport à la stratégie write-through. C'est pourquoi ces deux stratégies sont utilisées dans le protocole DHCCP comme suit :

- **Le write-through du Cache L1 vers le Contrôleur Mémoire :** La nouvelle valeur est écrite à la fois dans la ligne du Cache L1 (s'il en contient une copie) et dans le Contrôleur Mémoire, qui la répercute sur les autres copies, soit par un Update ou par un Invalidate selon le nombre de copies actuelles dans les caches.
- **Le write-back du Contrôleur Mémoire vers la Mémoire :** La nouvelle valeur est écrite uniquement dans la ligne du Contrôleur Mémoire et ne sera recopiée en Mémoire que si la ligne du Contrôleur Mémoire qui la contient est remplacée ou en cas d'un Broadcast Invalidate.

III.2.2. Modèles et résultats préexistants

Au début du stage, les modèles préexistants du protocole m'ont été fournis. Ces modèles sont décrits en PROMELA et ont permis une vérification partielle du protocole [2] [3]. La première étape de mon stage a consisté en l'étude de ces modèles, leur simulation et la vérification d'absence de blocage avec SPIN [6].

III.2.2.1. Modèles PROMELA

Un programme PROMELA modélise un système concurrent composé d'entités asynchrones communiquant par variables partagées ou par envoie / réception de messages. Les entités concurrentes sont décrites sous la forme de processus (dont le comportement est séquentiel) et le langage offre des primitives de description d'envoi ou réception de messages au travers de canaux FIFO bornés. Ce programme est ensuite utilisé à des fins de vérification par le model checker SPIN [6].

Les données sont décrites dans les modèles PROMELA décrits par M. NAJEM [2], mais la vérification n'as pas pu aller jusqu'à parcourir tout l'ensemble des états accessibles du système à cause du phénomène de l'explosion combinatoire du nombre d'états, M. MANSOUR a optimisé et simplifié ces modèles de manière à éviter toute manipulation des données et donc réduire l'espace d'états [3]. Pour réaliser mon projet je me suis basée sur ces modèles simplifiés.

a) Eléments de base

1) Les automates :

Le comportement de chaque élément constitutif de la machine multiprocesseur TSAR a été modélisé sous forme d'un automate d'états finis comme suit [3]:

- **Modèle du processeur :** envoie aléatoirement des demandes de lecture et d'écriture d'un bloc mémoire à une adresse donnée, puis se met en attente de leurs acquittements. Après réception de l'acquittement, l'automate passe à l'état initial.
- **Modèle du cache L1 :** contient une ligne et donc ne peut contenir qu'un bloc mémoire (ie : une ligne du cache ne peut contenir qu'un seul bloc à un instant donné, mais au fil du temps il peut contenir des blocs correspondants à différentes plages d'adresses). L'automate du cache L1 est joint en Annexe 3.
- **Modèle du contrôleur mémoire :** contient deux lignes et peut contenir un bloc mémoire par ligne. Dans ce cas un automate a été modélisé pour chaque ligne (Annexe 4), ces automates fonctionnent en parallèle dans le système.

- **Modèle de la mémoire :** contient deux blocs mémoire d'adresses X et Y. Les états de cet automate correspondent aux demandes de lecture et d'écriture sur ces blocs.

NB : Les blocs mémoire d'adresse X et Y sont stockés dans des lignes différentes dans le contrôleur mémoire mais dans la même ligne dans le cache L1.

2) Les canaux de communication :

Dans la machine multiprocesseur TSAR, les composants tels que le processeur, le cache L1, le contrôleur mémoire et la mémoire communiquent entre eux par l'envoi de messages dans des canaux de communication FIFO [2].

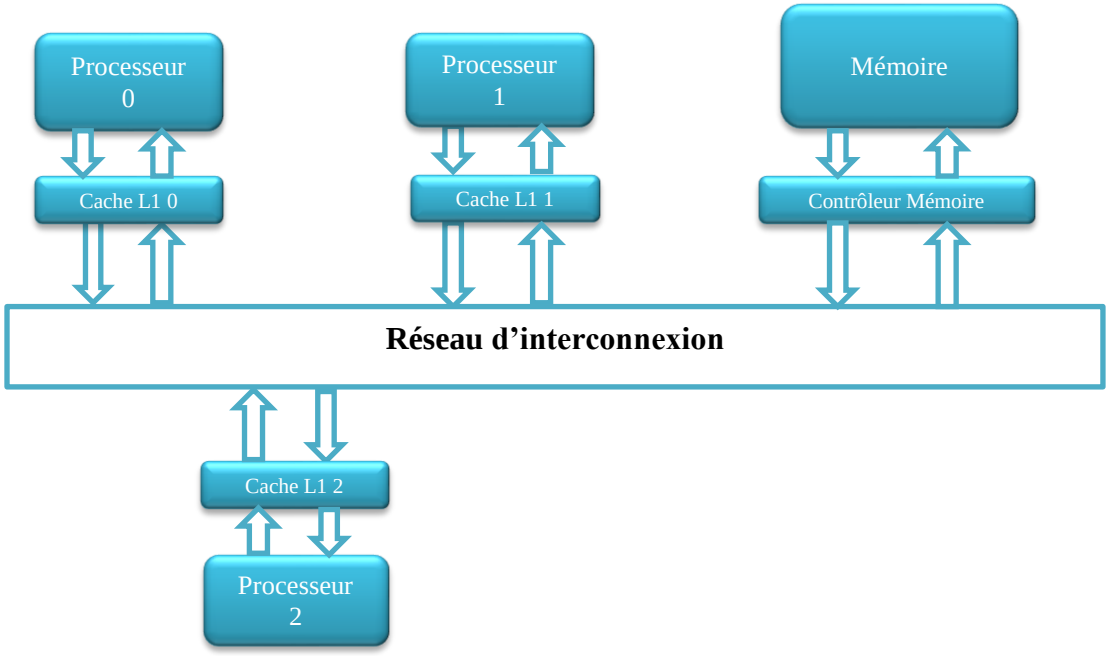


Figure 1 : Plateforme à trois Processeurs

3) Les messages :

Les messages envoyés dans les canaux de communication véhiculent les informations suivantes :

- Le type du message envoyé (lecture, écriture,…),
- L'adresse du bloc mémoire,
- L'identifiant du cache L1.

En PROMELA, un « ! » devant un message signifie son émission dans un canal de communication. De même, un « ? » signifie sa réception sur un canal de communication.

b) Plateformes :

Après avoir modélisé chaque élément constitutif de la machine multiprocesseur TSAR, on construit des plateformes intégrant différents nombres de composants par instanciation des modèles des composants élémentaires. Ainsi, pour construire une plateforme à 2 processeurs et 2 adresses on doit intégrer l'automate du processeur 1 et de son cache L1, l'automate du processeur 2 et de son cache L1, l'automate du contrôleur mémoire pour l'adresse X ainsi que celui pour l'adresse Y, et enfin l'automate de la mémoire. La composition de ces automates est asynchrone, la lecture et l'écriture des messages se font à travers des canaux bornés (un seul message dans le canal) qui sont déclarés comme variables globales du programme.

III.2.2.2. Compréhension des modèles et premières vérifications

Dans un premier temps, les modèles fournis ont été analysés en simulation et l'absence de blocage a été vérifiée avec SPIN. Ci-dessous j'explique les différentes étapes du protocole DHCCP sur deux plateformes :

1) Plateforme à un processeur :

Cette plateforme intègre un processeur, un cache L1, un contrôleur mémoire à 2 adresses et une mémoire. Le processeur exécute séquentiellement les opérations suivantes : lecture adresse X, écriture adresse Y, lecture adresse Y, lecture adresse X et écriture adresse X.

L'exécution de l'opération de lecture charge le bloc mémoire d'adresse X dans le cache L1. L'écriture ne change pas le contenu du cache L1 mais elle met à jour le bloc mémoire d'adresse Y au niveau du contrôleur mémoire. La lecture de l'adresse Y, évince le bloc d'adresse X et charge le bloc d'adresse Y dans la ligne du cache L1. La lecture de l'adresse X, évince le bloc d'adresse Y et recharge le bloc d'adresse X dans la ligne du cache L1. L'écriture d'adresse X, modifie la donnée dans le cache L1et répercute cette nouvelle valeur vers le contrôleur mémoire d'adresse X.

2) Plateforme à deux processeurs :

Cette plateforme intègre deux processeurs et donc deux caches L1, un contrôleur mémoire à 2 adresses et une mémoire. Les processeurs exécutent séquentiellement les opérations suivantes : processeur 1 (lecture adresse X), processeur 2 (lecture adresse X), processeur 1 (lecture adresse Y), processeur 2 (écriture adresse Y).

L'exécution de l'opération de lecture charge le bloc mémoire d'adresse X dans le cache L1 du processeur 1. La deuxième demande de lecture charge le bloc mémoire d'adresse X

dans le cache L1 du processeur 2 et le directory maintien le nombre et la liste des copies d'adresses X. La lecture de l'adresse Y, évince le bloc d'adresse X et charge le bloc d'adresse Y dans la ligne du cache L1 (processeur 1), le compteur de copies passe à 1. L'écriture ne change pas le contenu du cache L1 (processeur 2) mais elle met à jour le bloc mémoire d'adresse Y au niveau du contrôleur mémoire qui répercute cette nouvelle valeur sur la copie du cache L1 (processeur 1).

3) Absence de deadlock :

L'absence de deadlock a été vérifiée avec SPIN sur une plateforme à un processeur, mais la vérification a été partielle sur les plateformes à deux et à trois processeurs. En effet, la vérification prend plusieurs heures et se termine à une certaine profondeur à cause de l'espace d'états accessible du système qui devient très grand.

III.2.2.3. Problématique

La vérification complète d'un système qui intègre trois processeurs avec un seuil TH = 2 n'a pas pu s'achever à cause du nombre trop élevé d'états à explorer. L'absence de blocage a pu être vérifiée sur une portion de l'espace d'états, en parcourant celui-ci en largeur d'abord. Le graphe des états accessibles a pu être construit exhaustivement jusqu'à une profondeur de 300 transitions, il comprend 1,95 $.10^9$ états et 1,23 $.10^{10}$ transitions [3]. D'après ces résultats on voit bien que l'espace d'états parcouru par l'outil de vérification SPIN est très important, c'est pourquoi mes encadrants ont choisi d'utiliser l'outil de vérification DIVINE pour deux raisons principales :

- La modélisation sous forme d'automate est plus abstraite en DIVINE qu'en PROMELA (multiples affectations en PROMELA regroupées sur une seule transition en DIVINE).
- Pouvoir aller plus loin dans la vérification avec des outils de model checking symbolique (SDD) qui acceptent les modèles DIVINE.

III.2.3. Divine Model-Checker

III.2.3.1. Définition

DIVINE [5] est développé depuis 2010 par le Laboratoire Paradise de la Faculté d'Informatique de l'Université Masaryk, de Brno, qui est le second établissement de l'enseignement supérieur Tchèque.

Divine est un Model Checker, utilisé pour la vérification de systèmes décrits sous différents formats d'entrée (LLVM, UPPAAL, DVE, CESMI, MURPHI et COIN). La version 2.5.2 n'offrant pas toutes les fonctionnalités nécessaires (notamment la vérification sous contrainte d'équité) nous avons utilisé la version 2.97 qui est plus récente.

Divine fonctionne sous Linux et MS Windows, il est doté d'une interface graphique avec un simulateur et un analyseur de contre-exemple. En annexe 2 se trouve un aperçu de l'interface de Divine en mode Simulation.

Divine offre la possibilité de vérifier des propriétés du système qui sont exprimées sous forme de formules LTL. Pour lancer la construction du graphe des états accessibles du système il suffit de cliquer sur le bouton Reachability, s'il n'y a aucun état de blocage dans le modèle, les résultats de la vérification sont affichés : le nombre d'états parcourus et de transitions franchies. Si un état de blocage existe dans le modèle, un diagramme est affiché. Ce diagramme représente tous les états parcourus depuis l'état initial jusqu'à l'état invalidant la propriété. J'explique dans le chapitre « Production du contre exemple » comment procéder pour corriger ces erreurs à partir du diagramme affiché.

III.2.3.2. Modélisation d'un système et syntaxe en Divine

Sous Divine, j'ai modélisé le système de la figure 1 comme un ensemble de processus qui s'exécutent simultanément de manière asynchrone (une transition franchie à la fois dans le système) en s'échangeant des messages dans des canaux de communication. Chaque processus modélise le comportement d'un élément constitutif du système sous forme d'un automate d'états finis. Ci-joint en Annexe 3 l'automate du Cache L1 et en Annexe 4 celui du Contrôleur Mémoire.

Un fichier Divine contient des variables globales, des canaux et des processus. Chaque processus peut avoir ses propres variables locales, ses états et ses transitions. Chaque transition peut avoir une garde et peut être synchronisée par la commande **sync**, qui permet l'envoi ou la réception d'un message dans un canal de communication. Une transition est franchissable si sa garde est évaluée à vrai et que l'opération de synchronisation est possible. Quand une transition est franchie, le processus passe de son état actuel vers un autre état, les variables sont mises à jour selon les effets de la transition. Chaque processus est modélisé comme suit :

Process Nom {

Déclaration des variables locales : **byte** Nom de la variable;

Déclaration des états de l'automate : **state** Etat1, Etat2… ;

Définition de l'état initial de l'automate : **init** Etat1;

Définition des transitions :

trans

Etat_i -> Etat_j {

Guard (conditions de franchissement de la transition) ;

Effect (Affectation1, Affectation2,…) ;

};

}

Il existe deux type de canaux sous Divine, les canaux sans buffer et les canaux à buffer. Les canaux sans buffer sont utilisés pour la communication synchrone, le processus qui envoie et celui qui reçoit le message franchissent en même temps leurs transitions. Les canaux à buffer sont utilisés pour la communication asynchrone. Si le canal est vide un message peut être envoyé et si le canal est plein un message peut être consommé, ces deux opérations s'exécutent séparément.

La déclaration des variables globales se fait au tout début du modèle, et à la fin du modèle, après avoir défini tous les processus du système ajouter la ligne « **system sync;** » pour une exécution synchrone du système, ou bien ajouter « **System async;** » pour une exécution asynchrone.

Limitations de Divine par rapport à PROMELA :

- Les types des objets manipulés sont plus restreints en Divine qu'en PROMELA, notamment il n'y a pas de possibilité de définir des types énumérés (voir tableau 2).
- PROMELA offre la possibilité de consulter le contenu du message qui se trouve dans le canal sans le consommer, ce qui n'est pas possible en Divine.

III.2.3.3. Algorithme de vérification de propriétés LTL

Divine offre la possibilité de vérifier des propriétés du système qui sont exprimées sous forme de formules LTL. Comme Divine ne reconnait pas directement ces formules, on doit les fournir dans un fichier séparé avec l'extension « .ltl » (Annexe 5). La propriété LTL est transformée en un automate de Büchi reconnaissant les séquences contre-exemple de la propriété. Cet automate est intégré par produit synchrone à la description du système à vérifier.

Le vérificateur parcourt tous les états accessibles du système (éventuellement synchronisé avec l'automate de Büchi) en commençant par l'état initial et vérifie la validité de la formule en recherchant des cycles d'état accepteur dans le graphe des états accessibles. Si le système vérifie la propriété alors la vérification se termine avec succès et les résultats de cette vérification s'affichent. Si le vérificateur arrive dans un état qui ne vérifie pas la propriété LTL alors la vérification s'arrête et un diagramme s'affiche. Ce diagramme, qui est un contre-exemple de la propriété LTL, représente tous les états parcourus depuis l'état initial jusqu'à l'état qui ne vérifie pas la propriété.

Exemple :

Soit un système composé d'un modèle de processeur et d'un modèle (abstrait) de hiérarchie mémoire, communiquant par envoi de messages au travers de deux canaux C1 et C2, et progressant de façon asynchrone.

Soit l'automate du processeur représenté dans la figure 2 où :

- Le processeur est initialement à l'état Rdy, il envoie une demande de lecture sur C1 puis passe à l'état Wrd.
- Le processeur étant à l'état Wrd, reçoit un acquittement à sa demande de lecture sur C2 puis passe à l'état Rdy.

Soit en figure 3 l'abstraction de la hiérarchie mémoire où :

- A la réception du message M sur C1, la hiérarchie mémoire passe de l'état initial X à l'état Y (dans Y on abstrait tout le trajet de la transaction rapatriant la donnée dans le cache L1).
- La hiérarchie mémoire peut répondre à ce message en envoyant l'acquittement ACK sur C2 et passe à l'état X (dans ce cas la hiérarchie mémoire a reçu la donnée du niveau supérieur et la transmet au processeur).
- La hiérarchie mémoire peut ne pas répondre au message M et donc passe à l'état Z (dans ce cas la hiérarchie mémoire n'a pas reçu la donnée du niveau supérieur).

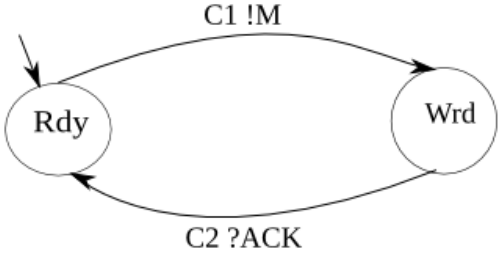


Figure 2 : Automate du Processeur

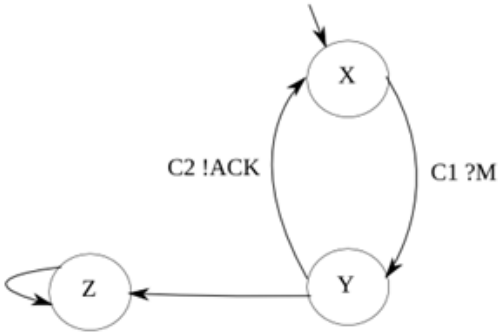


Figure 3 : Automate de la hiérarchie mémoire

Comme notre système s'exécute de manière asynchrone, le produit de ces deux automates est représenté dans la figure 4.

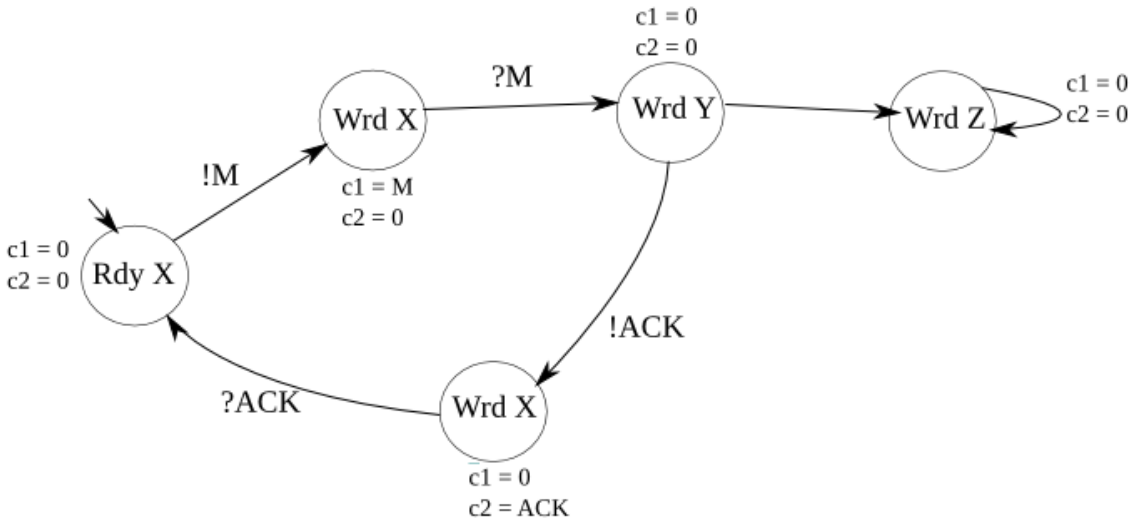


Figure 4 : Produit asynchrone de l'automate du processeur avec celui de la hiérarchie mémoire

On cherche à vérifier la propriété suivante : Pour chaque état de chaque séquence, si le processeur est en attente de lecture alors il passera à l'état Rdy (ce qui implique que chaque requête, émise par le processeur, sera finalement acquittée).

L'expression LTL de la propriété est **: G**(Processeur. Wrd -> **F**(Processeur.Rdy))

L'outil DIVINE construit automatiquement un automate de Büchi reconnaissant les séquences infinies invalidant la propriété LTL. Il est donné sur figure 5.

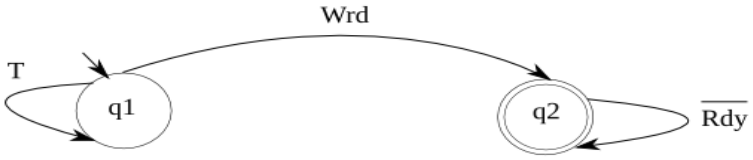


Figure 5 : Automate de Büchi

L'état accepteur de l'automate est **q2** donc toutes les séquences se terminant par $(\mathbf{q2})^{\omega}$ sont des contres exemples de la propriété. Elles correspondent à des séquences le long desquelles un nombre fini de requêtes du processeur a été acquitté mais une requête reste indéfiniment sans acquittement (le processeur reste en Wrd).

Le produit synchrone [4] de l'automate de Büchi avec celui du système à vérifier (figure 4) nous donne l'automate suivant :

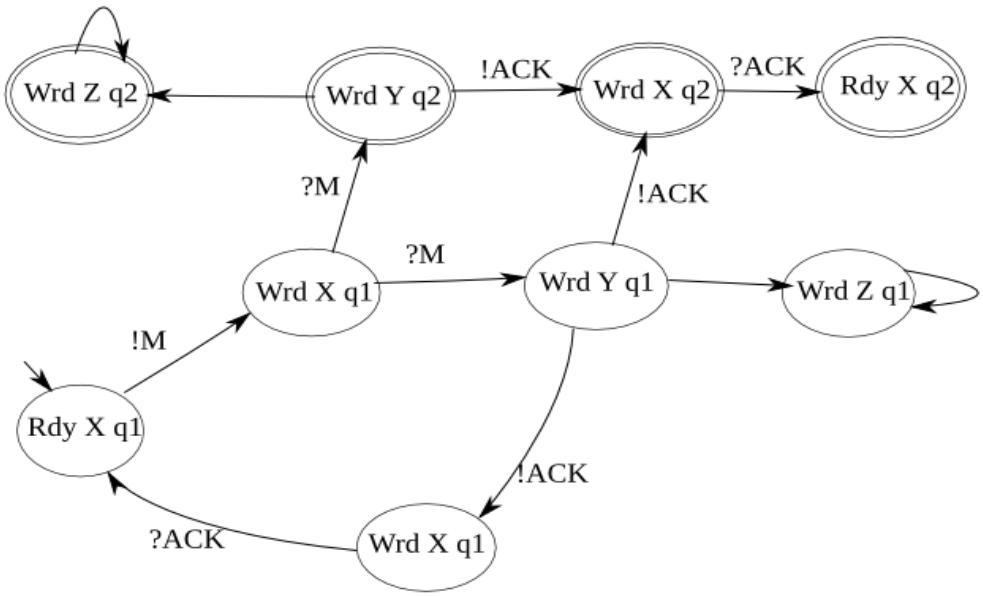


Figure 6 : Produit synchrone de l'automate de Büchi avec celui du système à vérifier

Toutes les séquences infinies qui bouclent sur l'état « Wrd Z q2 » sont acceptées par l'automate de Büchi. En effet, ces séquences ne vérifient pas la propriété LTL et donc représentent un contre exemple de la propriété. Le processeur ne passe plus à l'état Rdy (ne reçoit pas d'acquittement à sa demande de lecture).

Toutes les séquences finies qui se terminent par l'état « Rdy X q2 » sont des séquences non acceptées par l'automate de Büchi car ce sont des séquences qui vérifient la propriété (le processeur reçoit un acquittement à sa demande puis passe à l'état Rdy et ne fait plus de demande).

III.2.3.4. Production du contre exemple

Le contre exemple est affiché dans deux cas :

- Dans la vérification d'accessibilité s'il existe un deadlock.
- Dans la vérification LTL si le système ne vérifie pas la propriété.

Pour trouver l'erreur et la corriger, il faut parcourir toute la trace d'exécution depuis l'état initial tout en vérifiant les conditions de franchissement des transitions ainsi que l'affectation des variables dans les états. Dans le cas d'une vérification LTL, en plus des

vérifications précédentes, il faut vérifier les points de synchronisations de l'automate de Büchi avec celui du système.

III.2.4. Modélisation et vérification du protocole de communication

III.2.4.1. Niveau d'abstraction

A. Choix de modélisation d'un canal

Les composants de la plateforme communiquent entre eux en envoyant des messages dans différents canaux de communication qu'on a pu définir et utiliser dans SPIN. Sous Divine, on n'a pas utilisé de canaux pour la communication car ils ne fonctionnent pas de la même façon que ceux utilisés dans SPIN.

Dans SPIN, la vérification du contenu du message qui se trouve dans le canal peut se faire avant sa consommation. Par contre dans Divine, les messages doivent être consommés avant de lire leurs contenus, ce qui pose un problème pour les canaux multi-lecteurs avec des messages à destination d'un seul lecteur ciblé. Si un composant de l'architecture consomme un message et après vérification de son contenu il trouve qu'il ne lui était pas destiné et si entre temps un autre composant écrit dans le canal alors l'ordre FIFO des messages qui circulent dans les canaux est brisé. Cette fonctionnalité étant fondamentale pour notre modèle, nous avons reconstruit des canaux multi-lecteur ou multi-écrivain avec test du destinataire avant consommation à partir de variables globales. Chaque canal est composé de trois à quatre champs, selon les couples source et destination, qui sont utilisés et partagés par les différents composants de l'architecture.

B. Nomenclature et formats des canaux et des messages

1) Nomenclature et formats des canaux

Le tableau ci-dessous récapitule les canaux de l'architecture.

Nom du canal	Source	Destination	Type du message	Full / Empty	Adresse du bloc	Id du cache L1
PL1DTREQ Utilisé pour envoyer des messages de lecture et d'écriture	Processeur	Cache L1	DT_RD DT_WR	1 / 0	0 / 1	/
L1PDTACK Utilisé pour répondre aux messages de lecture et d'écriture.	Cache L1	Processeur	ACK_DT_RD ACK_DT_WR	1 / 0	0 / 1	/
L1MCDTREQ Utilisé pour envoyer des messages de lecture et d'écriture	Cache L1	Contrôleur Mémoire	RD WR	1 / 0	0 / 1	Id
MCL1DTACK Utilisé pour répondre aux messages de lecture et d'écriture	Contrôleur Mémoire	Cache L1	ACK_RD ACK_WR	1 / 0	0 / 1	Id
L1MCCUREQ Utilisé pour envoyer les demandes de Clean Up pour vider la ligne de cache	Cache L1	Contrôleur Mémoire	CLNUP	1 / 0	0 / 1	Id
MCL1CUACK Utilisé pour répondre aux demandes de Clean Up	Contrôleur Mémoire	Cache L1	ACK_CLNUP	1 / 0	0 / 1	Id
MCL1CPREQ Utilisé pour envoyer les messages d'invalidation et d'update.	Contrôleur Mémoire	Cache L1	M_UP B_INV M_INV	1 / 0	0 / 1	Id
L1MCCPACK Utilisé pour répondre aux messages de cohérence	Cache L1	Contrôleur Mémoire	ACK_M_UP ACK_B_INV ACK_M_INV	1 / 0	0 / 1	Id
MCMEMDTREQ Utilisé pour envoyer les demandes de lecture et d'écriture	Contrôleur Mémoire	Mémoire	GET PUT	1 / 0	0 / 1	/
MEMMCDTACK Utilisé pour répondre aux messages de lecture et d'écriture	Mémoire	Contrôleur Mémoire	ACK_GET ACK_PUT	1 / 0	0 / 1	/

Tableau 1 : Nomenclature et formats des canaux

2) Nomenclature et formats des messages

Les messages qu'on a utilisé dans Divine sont de type Byte. Le tableau ci-dessous montre le code dans DIVINE affecté à chaque type de message défini dans SPIN **:**

Description du message	Sous SPIN	Sous DIVINE
Demande de lecture	DT_RD	0
Demande d'écriture	DT_WR	1
Lecture validée	ACK_DT_RD	2
Ecriture validée	ACK_DT_WR	3
Demande de lecture	RD	4
Demande d'écriture	WR	5
Lecture validée	ACK_RD	6
Ecriture validée	ACK_WR	7
Demande d'évincement de la ligne (CLEAN UP)	CLNUP	8
Ligne évincée	ACK_CLNUP	9
Message d'invalidation (Broadcast Invalidate)	B_INV	10
Message d'invalidation (Multicast Invalidate)	M_INV	11
Message de mise à jour (Multicast Update)	M_UP	12
Acquittement du Broadcast Invalidate	ACK_B_INV	13
Acquittement du Multicast Invalidate	ACK_M_INV	14
Acquittement du Multicast Update	ACK_M_UP	15
Demande de lecture	GET	16
Ecriture	PUT	17
Lecture validée	ACK_GET	18
Écriture validée	ACK_PUT	19

Tableau 2 : Nomenclature et formats des messages sous SPIN et DIVINE

III.2.4.2. Plateformes modélisées et résultats de vérification

J'ai effectué l'ensemble des opérations de vérification sur le serveur « Berlioz » du réseau de recherche de l'UPMC, qui est un 64 bits à 4 cœurs hyperthreadés, et avec 4 Go de RAM. Processeur Intel Xeon 3GHz.

III.2.4.2.1. Plateforme à un processeur et une adresse mémoire

J'ai commencé mon projet par la modélisation d'une plateforme qui intègre un processeur, un cache L1, un contrôleur mémoire à une adresse et une mémoire. Les requêtes envoyées ne concernent qu'une adresse mémoire, d'où la modélisation suivante:

- **Le Processeur :** envoie des demandes de lecture et d'écriture du bloc mémoire d'adresse X = 0.
- **Le Cache L1 :** Contient une seule ligne et donc un bloc mémoire.
- **Le Contrôleur Mémoire :** Gère une seule adresse mémoire (X = 0).
- **La Mémoire :** Contient un unique bloc d'adresse X = 0.
- Les canaux de communication utilisés dans cette plateforme sont représentés dans la figure 7.

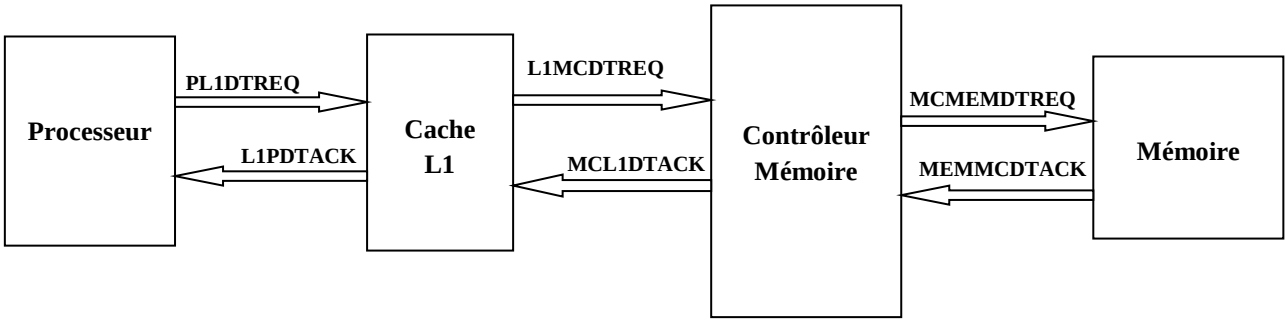


Figure 7: Plateforme à un processeur et une adresse mémoire

Divine permet d'afficher les métriques d'un système, on obtient pour la plateforme ci-dessus les résultats suivants:

```
================================================
56 states
75 transitions
0 accepting
0 deadlocks
================================================
```

D'après ces résultats, on voit que le système comporte 56 états et 75 transitions, et ne possède pas de deadlocks.

Ci-dessous je donne les résultats de la vérification de chaque propriété LTL. Le temps de vérification de chaque propriété passée sur cette plateforme était de quelques secondes.

1. Propriété : Infiniment souvent le processeur sera en attente de lecture ou en attente d'écriture.

P1: Property G(F(Processeur.wait_rd || Processeur.wait_wr))

```
initialise... |S| = 14

---------- iteration 1 ----------
reachability... |S| = 14
elimination & reset... |S| = 0
```

2. Propriété : Pour chaque état de chaque séquence, si le processeur est en attente de lecture ou d'écriture alors plus tard il passera à l'état Ready (donc il aura reçu un acquittement pour sa demande).

P2: Property G((Processeur.wait_rd || Processeur.wait_wr) -> F(Processeur.ready))

```
initialise... |S| = 47

---------- iteration 1 ----------
reachability... |S| = 47
elimination & reset... |S| = 0
```

3. Propriété : Pour chaque état de chaque séquence, si le Cache L1 est à l'état Miss alors à un moment dans le futur le Cache L1 et le Contrôleur Mémoire passent respectivement à l'état valid_data et valid_multicast (Le cache L1 a bien reçu la donnée attendue et le contrôleur mémoire a bien enregistré la présence d'une copie)

P3 : property G(Cache L1.miss -> F(Cache L1.valid_data && Mem_cache.valid_multicast))

```
initialise... |S| = 12

---------- iteration 1 ----------
reachability... |S| = 12
elimination & reset... |S| = 0
```

III.2.4.2.2. Plateforme à un processeur et deux adresses mémoire

Les requêtes envoyées dans cette plateforme concernent deux adresses mémoire, d'où la modélisation suivante:

- **Le Processeur :** envoie des demandes de lecture et d'écriture du bloc mémoire d'adresse X = 0 ou Y = 1.
- **Le Cache L1 :** Contient une seule ligne d'adresse X = 0 ou Y = 1.
- **Un Contrôleur Mémoire :** Gère deux adresses mémoire (X = 0 et Y = 1).
- **La Mémoire :** Contient deux lignes d'adresse X = 0 et Y = 1.
- Les canaux de communication utilisés dans cette plateforme sont représentés dans la figure 8.

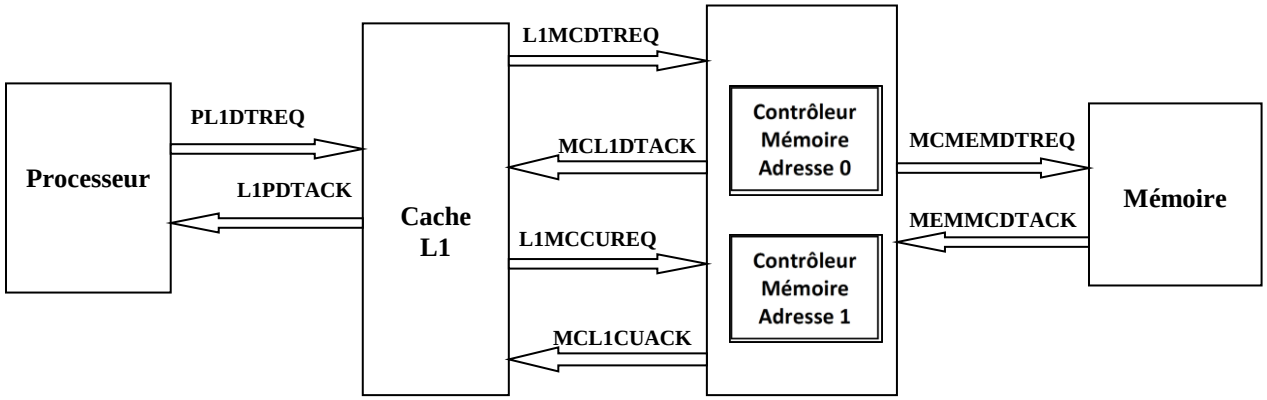


Figure 8 : Plateforme à un processeur et deux adresses mémoire

Les métriques obtenues pour le système modélisé de cette plateforme montrent qu'il comporte 1090 états et 1984 transitions, et ne possède pas de deadlock.

```
============================================
1090 states
1984 transitions
0 accepting
0 deadlocks
============================================
```

Ci-dessous je donne les résultats de la vérification de chaque propriété LTL. Le temps de vérification de chaque propriété passée sur cette plateforme était de quelques secondes.

1. Propriété **P2** :

```
initialise... |S| = 1005

---------- iteration 1 ----------
reachability... |S| = 1005
elimination & reset... |S| = 0
```

2. Propriété : Pour chaque état de chaque séquence, si le processeur est en attente de lecture du bloc mémoire d'adresse 1 et que la ligne de son cache est valide et contient une copie du bloc mémoire d'adresse 0, alors à un moment dans le futur le cache L1 envoie un CLNUP au contrôleur mémoire d'adresse 0, plus tard la ligne du cache L1 sera valide et contiendra une copie du bloc mémoire d'adresse 1.

P4: property G((Processeur.wait_rd && DT_RD_1 && Ligne_valid && ADDR_CACHE_0) -> (F(CLNUP_0) && F(Ligne_valid && ADDR_CACHE_1)))

```
initialise... |S| = 200

---------- iteration 1 ----------
reachability... |S| = 200
elimination & reset... |S| = 0
```

3. Propriété : Pour chaque état de chaque séquence,si le processeur est en attente de lecture du bloc mémoire d'adresse 1 alors à un moment dans le futur le contrôleur mémoire de l'adresse 1 passe à l'état valid_multicast et le compteur de copies est égal à 1.

P5: property G((Processeur.wait_rd && DT_ADDR_1) -> F(Mem_cache.valid_multicast_1 && NB_COPIE_ADDR1))

```
initialise... |S| = 146

---------- iteration 1 ----------
reachability... |S| = 146
elimination & reset... |S| = 0
```

III.2.4.2.3. Plateforme à deux processeurs et une adresse mémoire

La plateforme ci-dessous intègre deux processeurs et donc deux caches L1, un contrôleur mémoire à une adresse (X = 0) et une mémoire. Les messages envoyés ne concernent qu'une adresse mémoire, d'où la modélisation suivante:

- **Le Processeur0 et le Processeur1 :** envoient des demandes de lecture et d'écriture du bloc mémoire d'adresse X = 0.
- **Le Cache L1_0 et le Cache L1_1 :** Contiennent une seule ligne.
- **Le Contrôleur Mémoire :** Gère une seule adresse mémoire (X = 0).
- **La Mémoire :** Contient une ligne d'adresse X = 0.
- Dans cette plateforme, tous les canaux de communication sont utilisés et sont représentés dans la figure 9.

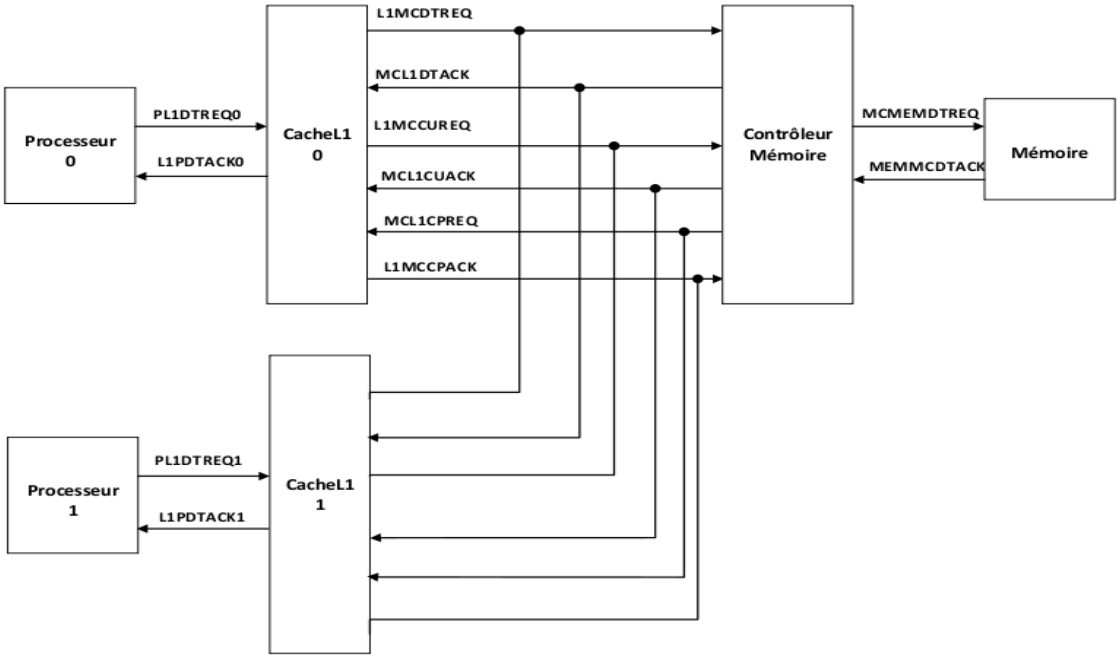


Figure 9 : Plateforme à deux processeurs et une adresse mémoire

1) Seuil TH = 2 :

D'après les résultats ci-dessous obtenus par la vérification du système de la plateforme précédente, on voit que le système comporte 566 états et 838 transitions, et ne possède pas deadlocks.

```
============================================
566 states
838 transitions
0 accepting
0 deadlocks
============================================
```

Ci-dessous je donne les résultats de la vérification LTL ainsi que la vérification avec Fairness de chaque propriété LTL. La vérification avec Fairness assure que tous les processus progressent. Le temps de vérification de chaque propriété passée sur cette plateforme était de quelques secondes.

1. Propriété P2 étendue à deux processeurs :

Property G(((Processeur0.wait_rd || Processeur0.wait_wr) -> F(Processeur0.ready)) || ((Processeur1.wait_rd || Processeur1.wait_wr) -> F(Processeur1.ready)))

```
initialise... |S| = 2403

---------- iteration 1 ----------
reachability... |S| = 2403
elimination & reset... |S| = 0
```

2. Vérification avec Fairness de la propriété : Pour tout état de chaque séquence équitable, si les lignes des deux caches (cache L1_0 et cache L1_1) sont valides, contiennent une copie du bloc mémoire d'adresse 0, et que le processeur0 envoie une demande d'écriture sur ce même bloc alors à un moment dans le futur le contrôleur mémoire envoie un Multicast Update pour le cache L1_1 pour mettre à jour sa ligne.

P6: Property G((v_cache_valide0 && v_cache_valide1 && wr_addr0_0) -> F (up1_0))

==========

831 states
1536 transitions
0 accepting
0 deadlocks

==========

3. Vérification avec Fairness de la propriété : Pour tout état de chaque séquence équitable, si les lignes des deux caches (cache L1_0 et cache L1_1) sont valides, contiennent une copie du bloc mémoire d'adresse 0, et que le processeur1 envoie une demande d'écriture sur ce même bloc alors à un moment dans le futur le contrôleur mémoire envoie un Multicast Update pour le cache L1_0 pour mettre à jour sa ligne.

P7: Property G((v_cache_valide0 && v_cache_valide1 && wr_addr1_0) -> F (up0_0))

==========

831 states
1532 transitions
0 accepting
0 deadlocks

==========

4. Vérification LTL de la propriété : Pour chaque état de chaque séquence, le contrôleur mémoire n'envoie jamais des messages d'invalidation (Broadcast_invalidate et Multicast_invalidate).

P8: Property G(!(B_INV) && !(M_INV))

initialise... |S| = 0

---------- iteration 1 ----------
reachability... |S| = 0
elimination & reset... |S| = 0

2) Seuil TH = 1 :

D'après les résultats ci-dessous obtenus par la vérification du système précédent avec un seuil TH = 1, on voit que le système comporte 1493 états et 2632 transitions, et ne possède pas de deadlocks.

```
==========================================
1493 states
2632 transitions
0 accepting
0 deadlocks
==========================================
```

Ci-dessous je donne les résultats de la vérification avec Fairness de chaque propriété LTL. Le temps de vérification de chaque propriété était de quelques secondes.

1. Vérification avec Fairness de la propriété : Pour tout état de chaque séquence équitable, si le nombre de copies dans les caches est égal à 2 alors le contrôleur mémoire est dans l'état Valid_broadcast

P9: Property G((nb_copie = 2) -> Mem_cache0.valid_broadcast)

```
==========================================
1391 states
2476 transitions
0 accepting
0 deadlocks
==========================================
```

2. Vérification avec Fairness de la propriété : Pour tout état de chaque séquence équitable, si le Contrôleur Mémoire de l'adresse 0 est à l'état valid_broadcast avec un nombre de copies supérieur à 0 et qu'il reçoit une demande d'écriture, alors à un moment dans le futur il va envoyer un Broadcast_invalidate pour le cache L1_0 et un autre pour le cache L1_1.

P10: Property G((cm_vb_0 && wr_0 && (nb_copie > 0)) -> F(b_inv_0) && F(b_inv_1))

```
==========================================
1391 states
2453 transitions
0 accepting
0 deadlocks
==========================================
```

III.2.4.2.4. Plateforme à deux processeurs et deux adresses mémoire

La plateforme ci-dessous intègre deux processeurs et donc deux cache L1, un contrôleur mémoire à deux adresses (X = 0, Y = 1) et une mémoire. Les requêtes envoyées concernent deux adresses mémoire, d'où la modélisation suivante :

- **Le Processeur0 et le Processeur1 :** envoient des demandes de lecture et d'écriture du bloc mémoire d'adresse X = 0 et Y = 1.
- **Le Cache L1_0 et le Cache L1_1 :** Contiennent une seule ligne.
- **Le Contrôleur Mémoire :** Gère deux adresses mémoire (X = 0 et Y = 1).
- **La Mémoire :** Contient deux lignes d'adresse X = 0 et Y = 1.
- Tous les canaux de communication sont utilisés dans cette plateforme et sont représentés dans la figure 10.

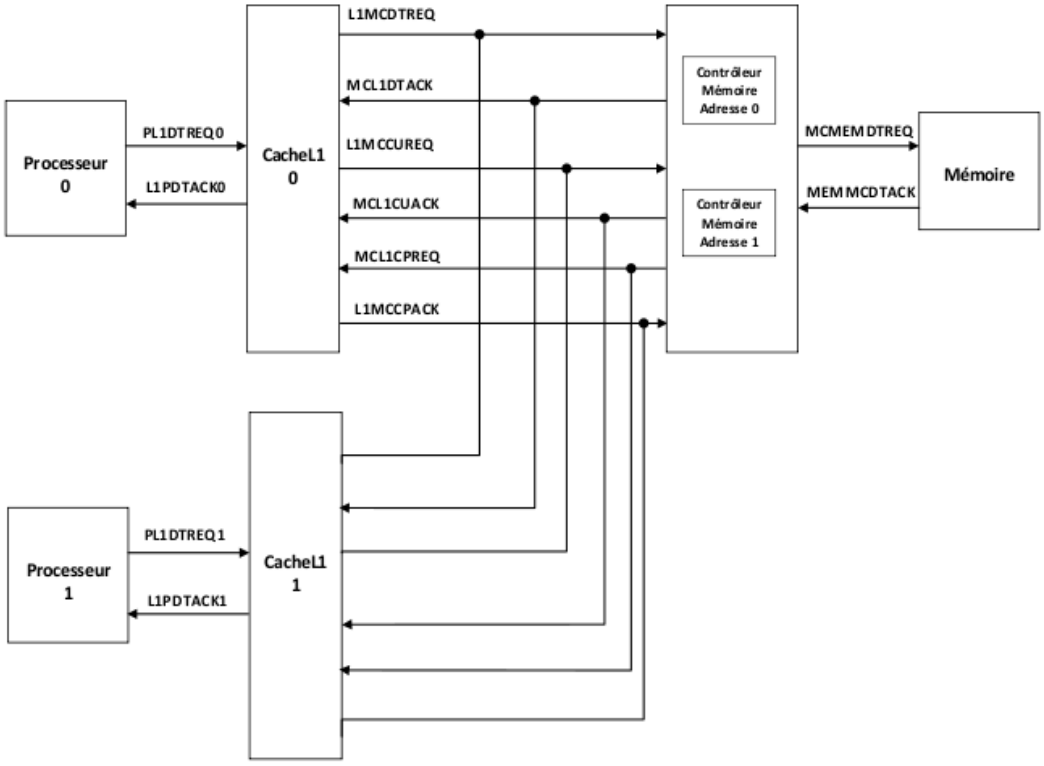


Figure 10 : Plateforme à deux Processeurs et deux adresses mémoire

1) Seuil TH = 2 :

D'après les résultats ci-dessous obtenus par la vérification du système de la plateforme précédente, on voit que le système comporte 78160 états et 191232 transitions, et ne possède pas de deadlocks.

==========

78160 states
191232 transitions
0 accepting
0 deadlocks

==========

Ci-dessous je donne les résultats de la vérification LTL ainsi que la vérification avec Fairness. Le temps de vérification de chaque propriété passée sur cette plateforme était de quelques secondes.

1. Vérification avec fairness de la propriété : Pour tout état de chaque séquence équitable, si le processeur0 est en attente de lecture du bloc mémoire d'adresse 1 alors à un moment dans le futur la ligne de son cache L1 passe à l'état Valide et contient une copie de ce bloc.

P11 : Property G((Processeur0.wait_rd && RD_DT0_1) -> F(V_CACHE_VALIDE_0 && ADDR_CACHE_0))

==========

31125 states
55838 transitions
0 accepting
0 deadlocks

==========

2. Vérification avec fairness de la propriété : Pour tout état de chaque séquence équitable, si le processeur0 est en attente d'écriture et que la ligne de son cache L1 est à l'état Valide et contient une copie du bloc mémoire d'adresse 0, alors le cache L1 reste dans cet état et contient cette même copie jusqu'à ce que le processeur0 envoie une demande de lecture du bloc mémoire d'adresse 1.

P12 : Property G((Processeur0.wait_wr && V_CACHE_VALIDE_0 && ADDR_CACHE_0) -> ((V_CACHE_VALIDE_0 && ADDR_CACHE_0) U RD_DT0_1))

==========

31125 states
55855 transitions
0 accepting
0 deadlocks

==========

3. Vérification avec fairness de la propriété : Pour tout état de chaque séquence équitable, si le processeur0 envoie une demande de lecture du bloc mémoire d'adresse 0 et que le contrôleur mémoire de l'adresse 0 est à l'état Empty, alors à un moment dans le futur le

contrôleur mémoire passe à l'état valide_multicast avec un nombre de copies égal à 1.

P13 : property G((RD_DT0_0 && cm_ety_0) -> F(cm_vm_0 && (nb_copie_addr0 = 1))

==========

31125 states
55846 transitions
0 accepting
0 deadlocks

==========

4. Vérification avec fairness de la propriété : Pour tout état de chaque séquence équitable, si le processeur0 envoie une demande d'écriture du bloc mémoire d'adresse 0 et que le contrôleur mémoire de l'adresse 0 est à l'état Empty alors à un moment dans le futur le contrôleur mémoire passe à l'état valide_multicast avec un nombre de copies égal à 0.

P14 : property G((WR_DT0_0 && cm_ety_0) -> F(cm_vm_0 && nb_copie_addr0 = 0))

==========

31125 states
55967 transitions
0 accepting
0 deadlocks

==========

5. Vérification avec Fairness de la propriété : Pour tout état de chaque séquence équitable, si les lignes des deux caches (cache L1_0 et cache L1_1) sont valides et contiennent une copie du bloc mémoire d'adresse 1, si le processeur1 envoie une demande d'écriture sur ce même bloc, alors à un moment dans le futur le contrôleur mémoire envoie un multicast_update pour le cache L1_0 pour mettre à jour sa ligne.

P15 : Property G((ligne_valid0 && ligne_cache0_1 && ligne_valid1 && ligne_cache1_1 && wr_addr1_1) -> F (up0_1))

==========

31125 states
55958 transitions
0 accepting
0 deadlocks

==========

6. Vérification avec Fairness de la propriété : Pour tout état de chaque séquence équitable, si les lignes des deux caches (cache L1_0 et cache L1_1) sont valides et contiennent une copie du bloc mémoire d'adresse 1, si le processeur0 envoie une demande d'écriture sur ce même bloc alors à un moment dans le futur le contrôleur mémoire envoie un multicast_update pour

le cache L1_1 pour mettre à jour sa ligne.

P16 : Property G((ligne_valid0 && ligne_cache0_1 && ligne_valid1 && ligne_cache1_1 && wr_addr0_1) -> F (up1_1))

```
==========================================
31125 states
55985 transitions
0 accepting
0 deadlocks
==========================================
```

7. Propriété **P8 :**

```
initialise... |S| = 0
---------- iteration 1 ----------
reachability... |S| = 0
elimination & reset... |S| = 0
```

2) Le seuil TH = 1 :

La vérification du système de la plateforme précédente avec un seuil TH = 1 donne les résultats ci-dessous, on voit que le système comporte 802771 états et 2602704 transitions, et possède 236 deadlocks.

```
==========================================
802771 states
2602704 transitions
0 accepting
236 deadlocks
==========================================
```

La vérification de cette plateforme s'arrête à un certain niveau, aucun processus ne progresse. Le diagramme des états parcourus depuis l'état initial s'affiche. J'ai commencé par parcourir ce diagramme depuis l'état initial tout en vérifiant les transitions franchies et les variables affectées dans chaque état pour trouver ce qui cause ce blocage.

J'ai trouvé que ce deadlock est dû au fait que le Message B_INV est consommé tardivement et avant la consommation du message ACK_RD présent dans le canal. J'explique ce blocage à travers le schéma de la figure 11 où :

- Le Contrôleur Mémoire envoie le message B_INV au Cache L1_1 qui ne le consomme pas tout de suite,

- Le Cache L1_1 envoie une demande de lecture (RD) au Contrôleur Mémoire qui l'acquitte en envoyant la réponse (ACK_RD), le compteur de copies (N_copies) passe à 1,
- Le Cache L1_1 décide de consommer en premier le message (B_INV), ce qui invalide sa ligne,
- Le Cache L1_1 consomme le message (ACK_RD) puis redemande le bloc mémoire en envoyant le message (RD),
- Le Contrôleur Mémoire répond à la demande de lecture en envoyant le message (ACK_RD) puis passe en mode Broadcast pour le protocole de communication, son compteur de copies passe à 2 (ie : au niveau du contrôleur mémoire le même cache L1 possède deux copies de ce même bloc),
- Le Cache L1_1 consomme le message (ACK_RD),
- Le Cache L1_0 envoie une demande de lecture,
- Le Contrôleur Mémoire répond à la demande de lecture en envoyant le message (ACK_RD), le compteur de copies passe à 3 ce qui est incohérent vu que la plateforme est composée de 2 caches L1.
- Le Cache L1_1 envoie une demande d'écriture, le Contrôleur Mémoire envoie deux messages d'invalidation (B_INV_0 et B_INV_1) pour les deux caches puis se met à attendre 3 réponses (car le nombre de copies est égal à 3).
- Le Contrôleur Mémoire ne reçoit que deux réponses sur trois attendues ce qui crée son blocage et ainsi le blocage complet du système.

NB : Ce deadlock existe réellement dans la machine TSAR, et n'a été trouvé qu'une année après. Sous Divine, nous avons trouvé ce deadlock au bout de 3 mois. Les réalisateurs du projet TSAR ont pris 6 mois pour le résoudre.

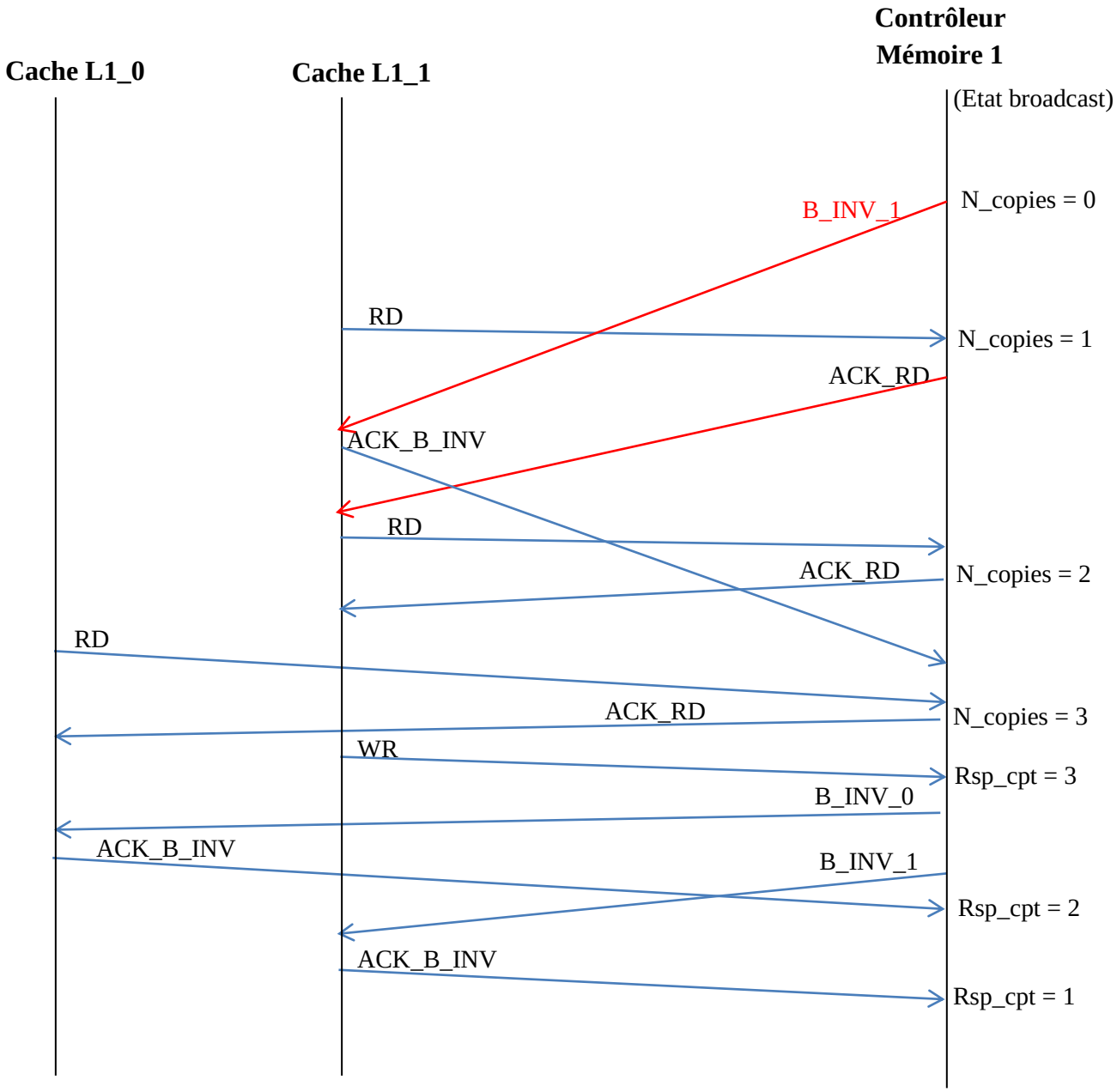


Figure 11 : Deadlock de la plateforme à deux processeurs et deux adresses mémoire avec TH = 1

III.2.4.3. Solution au Deadlock trouvé

Les réalisateurs du projet TSAR ont opté pour la solution suivante :

- Les canaux de communication partagés entre le Processeur et le Cache L1 et ceux partagés entre le Contrôleur Mémoire et la Mémoire restent inchangés.
- Les canaux de communication partagés entre le Cache L1 et le Contrôleur Mémoire sont les suivants :
  - ✓ Les canaux L1MCDTREQ, MCL1DTACK et MCL1CPREQ restent inchangés

- ✓ Les canaux L1MCCUREQ et MCL1CUACK ne sont plus utilisés.
- ✓ Canal L1MCCPACK : Le cache L1 envoie au Contrôleur Mémoire un message du type CLNUP comme réponse aux B_INV et aux M_INV reçus. Il envoie aussi un message du type ACK_M_UP en réponse au M_UP reçu.
- ✓ Canal MCL1CLACK : Le Contrôleur Mémoire envoie au cache L1 un message du type CLACK (CLNUP_ACK) en réponse au CLNUP reçu.

La figure 12 représente la nouvelle architecture (deux processeurs, deux cache L1, un contrôleur mémoire à deux adresses et une mémoire) avec la solution au deadlock.

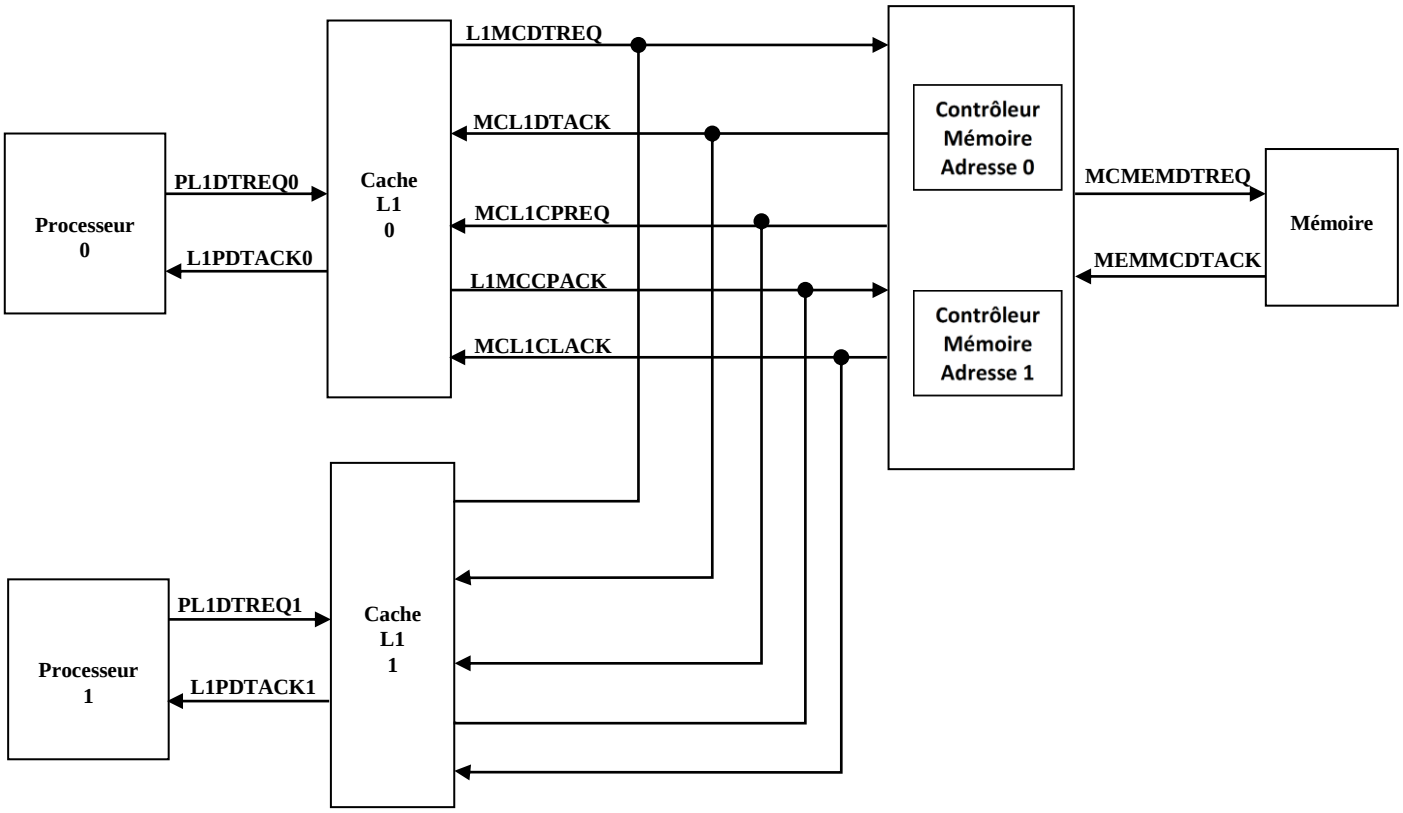


Figure 12 : Plateforme à deux processeurs avec la solution au deadlock

Les propriétés de P2 bis et P6 à P16 ont été vérifiées avec succès avec ces nouveaux modèles de composants éliminant le deadlock. Ci-dessous je donne les résultats de la vérification du système de la plateforme précédente avec un seuil TH = 1. On voit qu'on n'a plus de deadlocks, le système comporte 519152 états et 1730128 transitions.

```
==========================================
519152 states
1730128 transitions
0 accepting
0 deadlocks
==========================================
```

Dans ce qui suit, je donne les résultats de la vérification LTL ainsi que la vérification avec Fairness de la plateforme précédente.

1. Propriété **P2** étendue à deux processeurs. Le temps de vérification de cette propriété était de quelques secondes.

Property G(((Processeur0.wait_rd || Processeur0.wait_wr) -> F(Processeur0.ready)) ||
((Processeur1.wait_rd || Processeur1.wait_wr) -> F(Processeur1.ready)))

```
initialise... |S| = 506878
---------- iteration 1 ----------
reachability... |S| = 506878
elimination & reset... |S| = 0
```

2. Propriété **P10**. Le temps de vérification de cette propriété était de 52 mn.

```
==============================================
390342 states
1093270 transitions
0 accepting
0 deadlocks
==============================================
```

3. La vérification avec Fairness de la propriété : Pour tout état de chaque séquence équitable, si les lignes des deux caches (cache L1_0 et cache L1_1) sont valides et contiennent une copie du bloc mémoire d'adresse 1, alors le contrôleur mémoire de l'adresse 1 est à l'état valid-broadcast et son compteur de copies est égal à 2. Le temps de vérification de cette propriété était de 51 mn.

P17: property G((ligne_valid0 && ligne_cache0_1 && ligne_valid1 &&
ligne_cache1_1) -> (cm_vb_1 && (nb_copie_1 = 2)))

```
==============================================
390342 states
1093579 transitions
0 accepting
0 deadlocks
==============================================
```

IV. Travaux en cours

Actuellement, nous travaillons sur une plateforme qui intègre trois processeurs, trois caches L1, un contrôleur mémoire à deux adresses et une mémoire (Voir schéma ci-dessous).

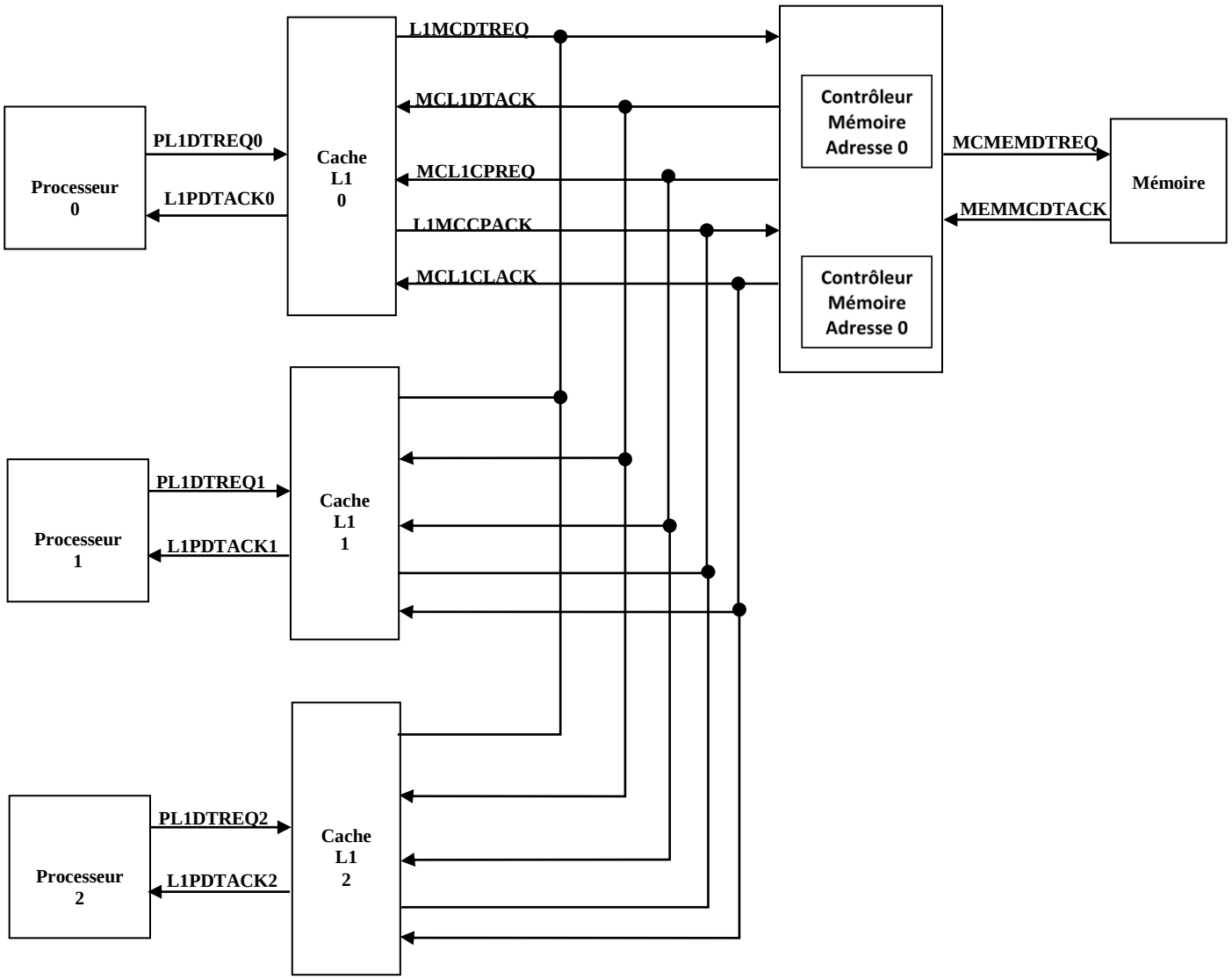


Figure 13 : Plateforme à trois processeurs

Le temps de vérification du système de cette plateforme avec un seuil TH = 3 a été très long (un jour), ci-dessous les résultats trouvés :

==========

16083038 states
52217606 transitions
0 accepting
0 deadlocks

==========

V. Conclusion et perspectives

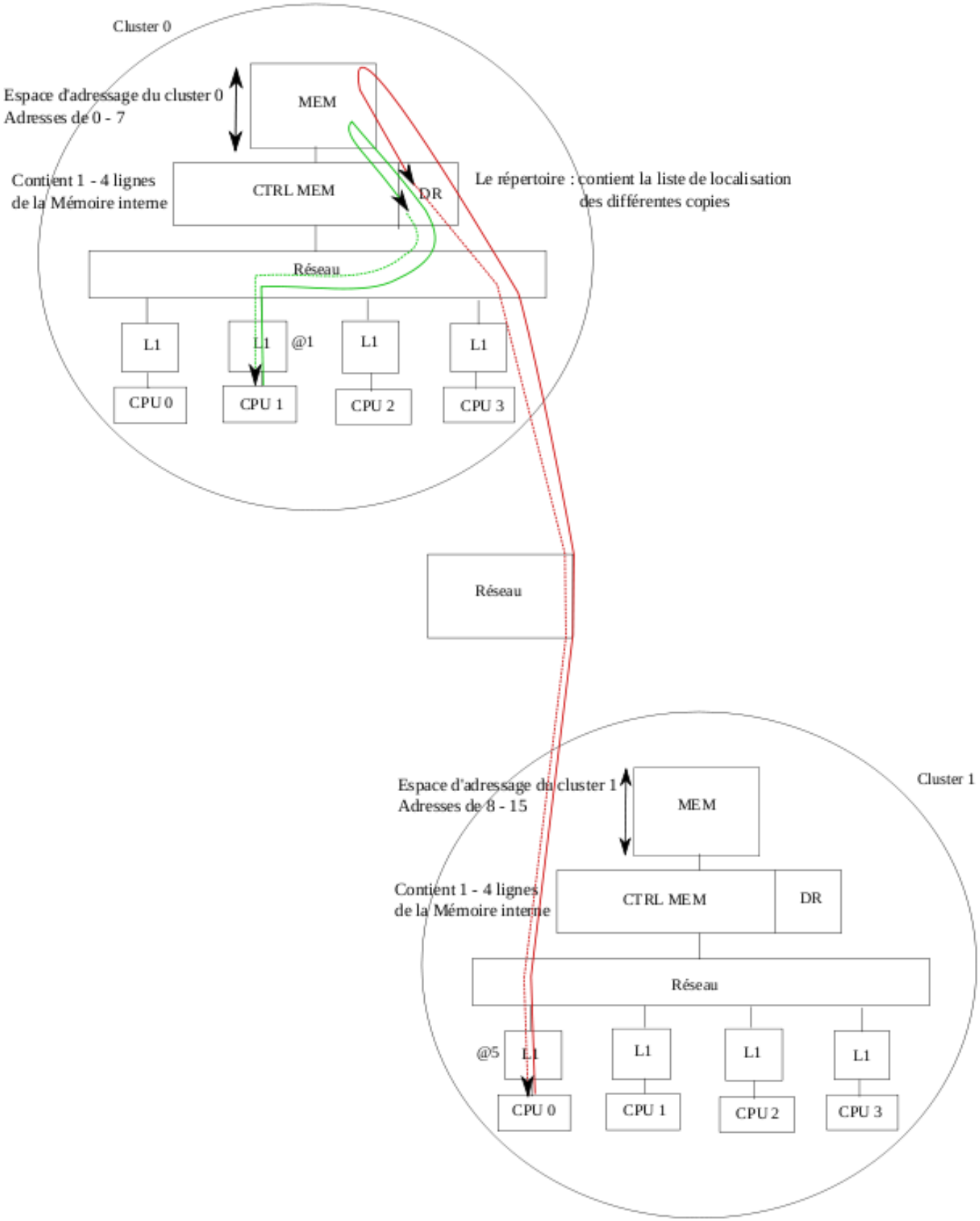
Ce stage en Laboratoire a été pour moi une expérience intéressante et enrichissante. Il m'a permis de mettre en pratique mes connaissances théoriques acquises durant le premier semestre (Propriétés LTL, automates,…) et d'en acquérir de nouvelles notamment dans les architectures multiprocesseur et ce en travaillant sur la machine TSAR qui intègre jusqu'à 4096 processeurs répartis sur 1024 clusters. De plus, j'ai pu découvrir le protocole de communication DHCCP développé au LIP6. Ce stage m'a permis aussi de découvrir comment modéliser un système en automates et effectuer des vérifications avec les outils SPIN et DIVINE.

Un travail supplémentaire sur ce projet est nécessaire pour :

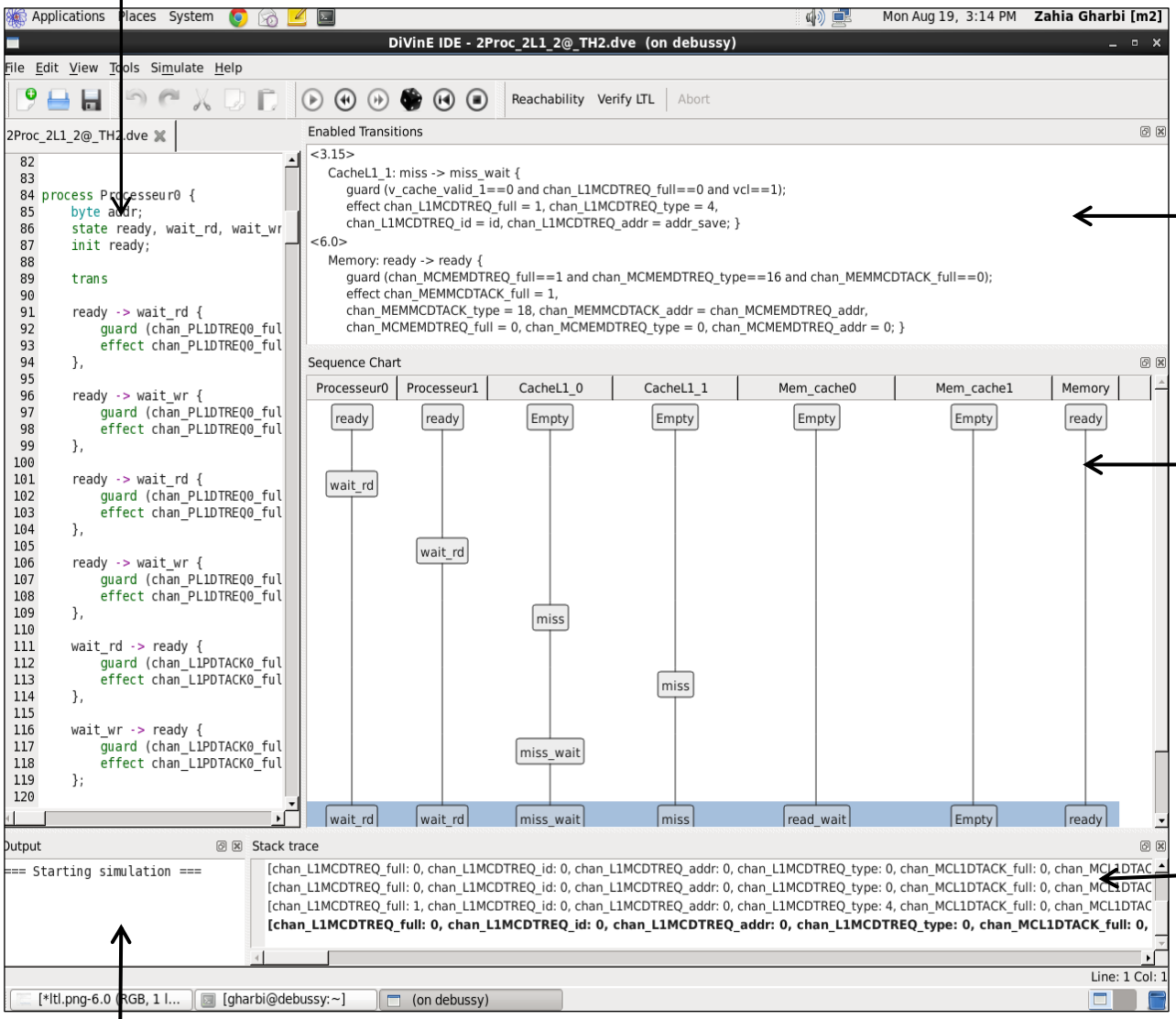
- ☞ Aller plus loin dans la vérification en utilisant les SDD qui utilisent des techniques de réduction de l'espace d'états du système,
- ☞ Ajouter le traitement des données aux modèles décrits sous DIVINE,
- ☞ Modéliser une mémoire à trois adresses pour avoir des évictions au niveau du contrôleur mémoire.

Enfin, j'ai eu la satisfaction d'avoir participé au projet TSAR ce qui me sera très utile pour la suite de ma vie professionnelle.

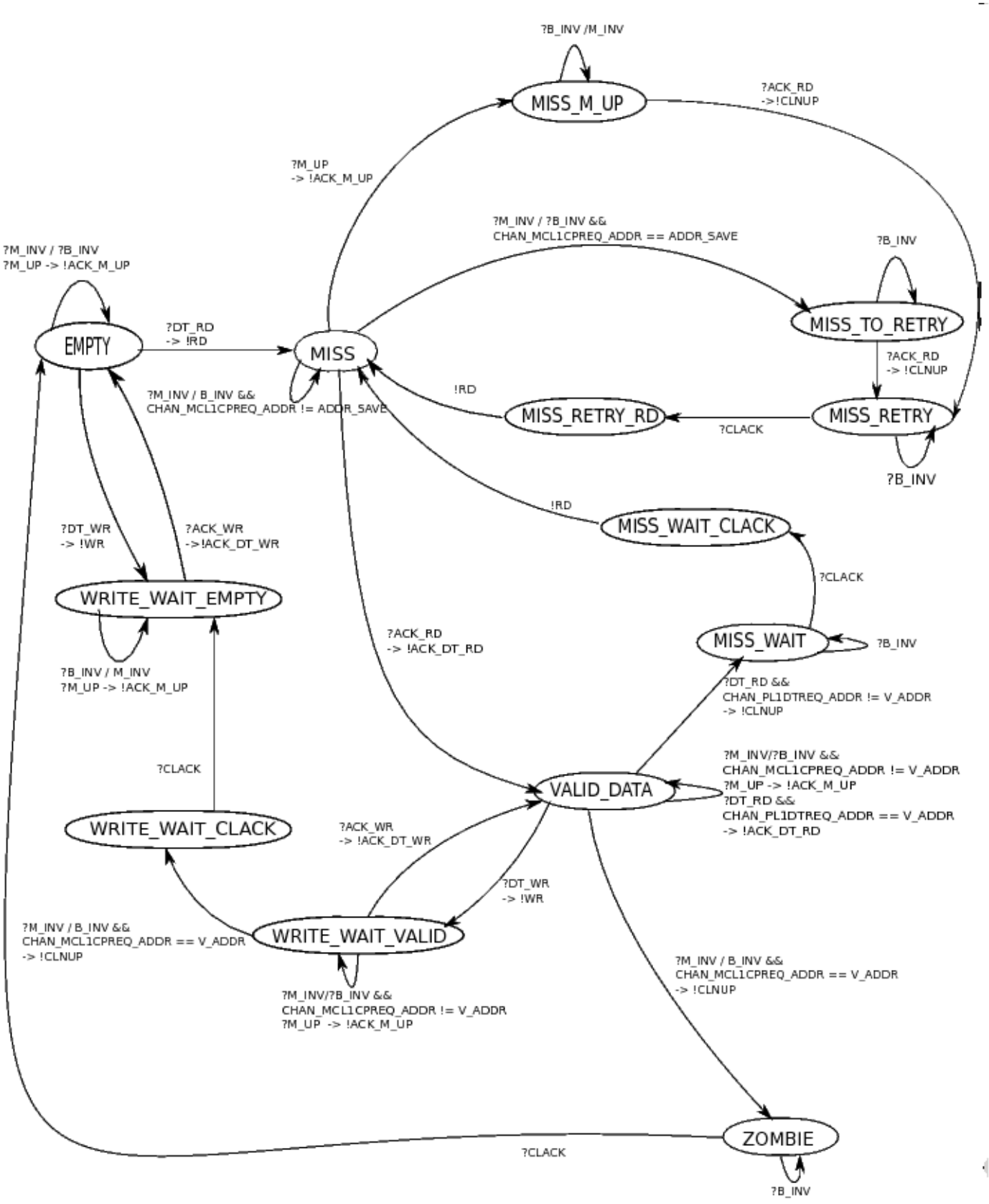
VI. Annexes



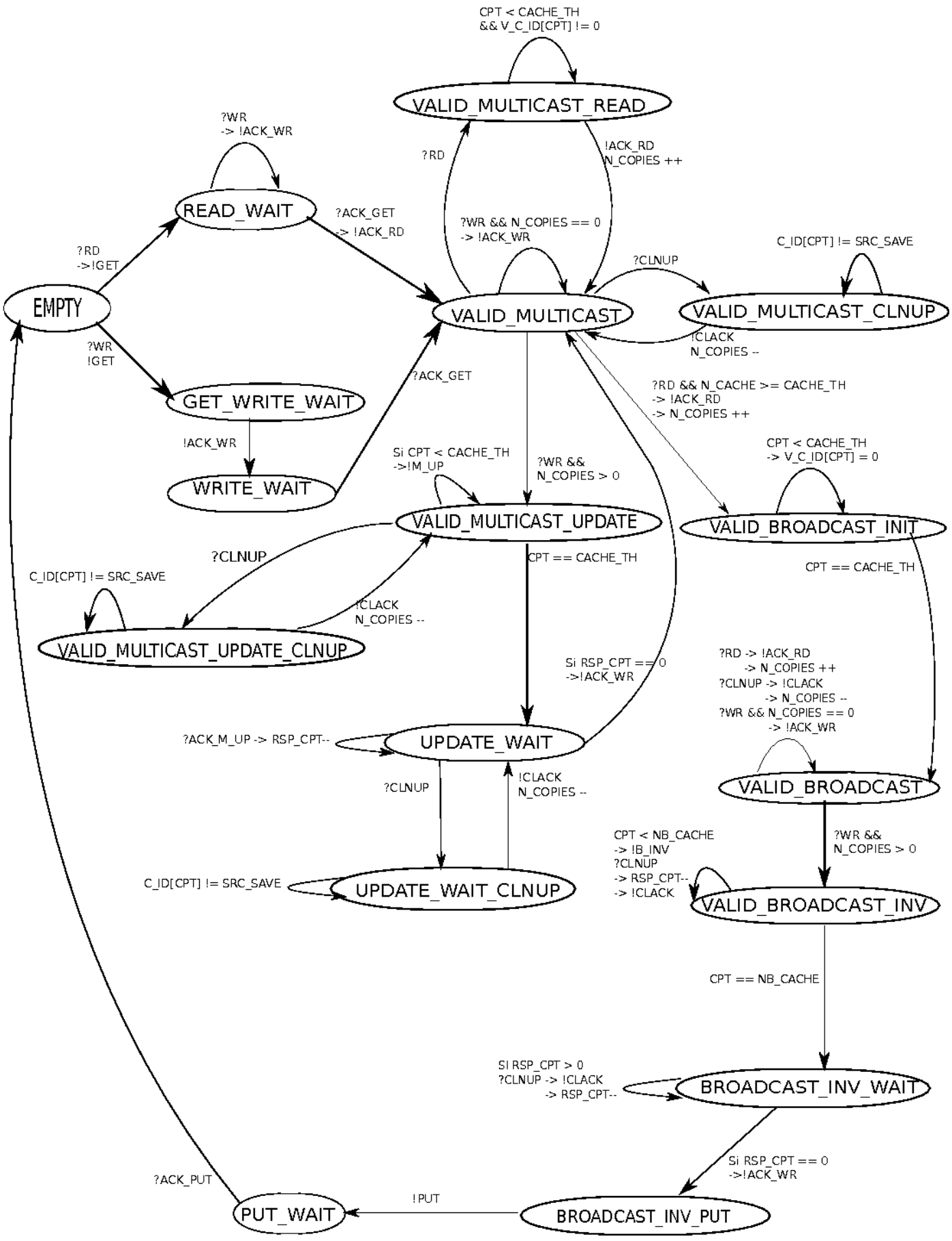
Annexe 1 : Machine Multiprocesseur TSAR à 2 clusters



Annexe 2 : Interface graphique de DIVINE en mode simulation



Annexe 3 : Automate du Cache L1



Annexe 4 : Automate du Contrôleur Mémoire

```
#define p0_w_rd (Processeur0.wait_rd)
#define p0_w_wr (Processeur0.wait_wr)
#define p0_rdy (Processeur0.ready)
#define p1_w_rd (Processeur1.wait_rd)
#define p1_w_wr (Processeur1.wait_wr)
#define p1_rdy (Processeur1.ready)
#define rd_dt0_1 (chan_PL1DTREQ0_full == 1 && chan_PL1DTREQ0_type == 0 &&
        chan_PL1DTREQ0_addr == 1)
#define ligne_valid0 (v_cache_valid_0 == 1)
#define ligne_cache0_1 (addr_cache_0 == 1)
#define ligne_valid1 (v_cache_valid_1 == 1)
#define ligne_cache1_1 (addr_cache_0 == 1)
#define wr_addr0_1 (chan_PL1DTREQ0_full == 1 && chan_PL1DTREQ0_type == 1 &&
        chan_PL1DTREQ0_addr == 1)
#define wr_addr1_1 (chan_PL1DTREQ1_full == 1 && chan_PL1DTREQ1_type == 1 &&
        chan_PL1DTREQ1_addr == 1)
#define up0_1 (chan_MCL1CPREQ_full == 1 && chan_MCL1CPREQ_type == 12 &&
        chan_MCL1CPREQ_addr == 1 && chan_MCL1CPREQ_id == 0 )
#define up1_1 (chan_MCL1CPREQ_full == 1 && chan_MCL1CPREQ_type == 12 &&
        chan_MCL1CPREQ_addr == 1 && chan_MCL1CPREQ_id == 1 )
#define b_inv (chan_MCL1CPREQ_full == 1 && chan_MCL1CPREQ_type == 10)
#define m_inv (chan_MCL1CPREQ_full == 1 && chan_MCL1CPREQ_type == 11)

#property G(F(p0_w_rd || p0_w_wr || p1_w_rd || p1_w_wr))
#property G(((p0_w_rd || p0_w_wr) -> F(p0_rdy)) || ((p1_w_rd || p1_w_wr) -> F(p1_rdy)))
#property G((p0_w_rd && rd_dt0_1) -> F(ligne_valid0 && ligne_cache0_1))
#property G((ligne_valid0 && ligne_cache0_1 && ligne_valid1 && ligne_cache1_1 &&
        wr_addr1_1) -> F (up0_1))
#property G((ligne_valid0 && ligne_cache0_1 && ligne_valid1 && ligne_cache1_1 &&
        wr_addr0_1) -> F (up1_1))
#property G(!(b_inv) && !(m_inv))
```

Annexe 5 : Exemple d'un fichier LTL

VII. Références Bibliographiques

[1] : Architecture des Ordinateurs : Une approche quantitative (3ème édition).
John L. Hennessy, David A. Patterson. (MORGAN KAUFMANN PUBLISHERS)

[2] : Rapport de projet M1- Master Informatique/SESI, UPMC : Modélisation et Analyse du protocole de cohérence de caches de la machine multiprocesseurs TSAR (2011). M. Najem.

[3] : Rapport de projet M1- Master Informatique/SESI, UPMC : Modélisation et Analyse du protocole de cohérence de caches de la machine multiprocesseurs TSAR : Absence de deadlocks (2012). A. Mansour.

[4] : Systems and Software Verification: Model-Checking Techniques and Tools.
B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci & P. Schnoebelen (2010).

[5] : Model checker DIVINE : http://divine.fi.muni.cz/

[6] : Model checker SPIN : http://spinroot.com/spin/whatispin.html