Université Pierre et Marie Curie

Mastère de sciences et technologies

MENTION INFORMATIQUE

2014 – 2015

Spécialité : SESI

SYSTÈMES ELECTRONIQUES ET SYSTÈMES INFORMATIQUES

Vérification de protocole de cohérence de cache hybride multicast/broadcast avec les techniques de model-checking

RAPPORT DE SOUTENANCE

10 septembre 2015

PRÉSENTÉ PAR

DI ZHAO

ENCADRANTS

QUENTIN MEUNIER

YANN THIERRY-MIEG

Laboratoire d'accueil : LIP6

Equipe MOVE

Table des matières

Table des figures

Chapitre 1

Contexte

1.1 Motivations

Ce sujet de stage est proposé par les équipes Alsoc et Move du LIP6. L'équipe Alsoc travaille sur une machine multiprocesseur à mémoire partagée appelée TSAR (Tera-Scale ARchitecture), plus particulièrement sur son protocole de cohérence de caches original nommé DHCCP (Distributed Hybrid Cache Coherence Protocol). On souhaite prouver formellement que ce protocole garantit bien des propriétés de cohérence et qu'il ne contient pas d'interblocage, à l'aide des outils développés au sein de l'équipe Move.

Le protocole de cohérence DHCCP est asynchrone, concurrent et distribué ce qui rend sa validation par des moyens classiques (tests, validation manuelle) difficile et non exhaustive. La vérification formelle fournit des garanties sur la correction du système souhaitable dans le cadre du projet TSAR. Le model-checking est une technique de vérification formelle automatique, mais qui se heurte à l'explosion combinatoire de l'espace d'états.

Un précédent projet LIP6 a permis de modéliser dans le langage `DVE` le protocole (stage M2 2013 de Z. GHARBI [4]), mais l'analyse des configurations cibles s'est avéré impraticable.

L'objectif de ce stage est donc de compléter l'étude existante et de permettre le passage à l'échelle de l'analyse. Pour cela, on s'appuiera sur le langage `GAL` , qui permet en particulier une description hiérarchique du système. La description hiérarchique et compositionnelle est exploitée par l'outil de vérification pour aider à faire face à la complexité d'analyse.

Le stage a donc pour objectifs de modéliser DHCCP de façon modulaire et hiérarchique en `GAL` , puis d'analyser ces spécifications, et enfin de généraliser l'approche pour permettre son application à d'autres modèles `DVE` .

1.1.1 L'architecture TSAR

TSAR (Tera-Scale Architecture) [1] est une architecture multiprocesseur définie dans le cadre d'un projet européen. Elle est destinée à supporter les applications standards des systèmes d'exploitation fonctionnant sur des PC, tels que Linux ou NetBSD.

C'est une architecture qui intègre jusqu'à 1024 processeurs répartis sur 256 clusters. Ces cluster sont interconnectés par un réseau de communication distribué (DSPIN).

L'architecture définie est à mémoire physiquement distribuée et logiquement partagée. Chaque processeur peut lire ou écrire des données stockées dans les caches L2 (aussi appelées MemCache) non seulement de son cluster mais aussi des autres clusters. Cela implique que le temps d'accès aux données et aux instructions est variable. Les données sont répliquées dans les caches L1 des processeurs qui en ont fait la demande. La cohérence des différentes copies est assurée en matériel au travers du protocole DHCCP, détaillé plus loin. Des prototypes virtuels ont été développés et permettent de simuler des architectures intégrant jusqu'à 1024 processeurs. Un prototype matériel à 16 processeurs a été réalisé sur FPGA.

La Figure 1.1 montre un exemple de l'architecture comprenant 2 clusters de 4 processeurs chacun, reliés par un réseau d'interconnexion. Le fonctionnement de cet exemple est le suivant :

- 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 cache L2 du cluster 0. Le contrôleur vérifie si le bloc demandé est chargé dans le cache. Comme ce n'est pas le cas, il envoie un message à la mémoire principale, qui lui répond en chargeant le bloc demandé dans le cache L2. 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 cache L2 du cluster 0. La requête est envoyée vers ce cache. Le contrôleur vérifie et trouve que le bloc demandé n'est pas chargé dans le cache, et envoie donc un message à la mémoire qui répond en chargeant le bloc demandé dans la ligne du cache L2 (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.

1.1.2 Protocole de cohérence de la machine multiprocesseur TSAR

Le protocole DHCCP utilise un répertoire pour assurer la cohérence des données entre les caches L1 et les caches L2. Il y a un cache L2 avec son répertoire dans chaque cluster. Les caches L2 sont associés avec des blocs mémoire, i.e. une certaine portion de l'espace d'adressage. Le répertoire conserve entre autres pour chaque bloc mémoire les informations suivantes :

- La validité de la ligne,
- Le fait que le ligne soit modifiée par rapport à la mémoire (dirty),
- Le nombre de copies du bloc mémoire présentes dans les caches L1,
- Les identifiants de tous les caches ayant 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 il y peut y avoir jusqu'à 1024 coeurs, cela implique d'avoir 1024 bits par bloc pour pouvoir localiser toutes les copies de ce bloc. Cela est très coûteux en matériel,

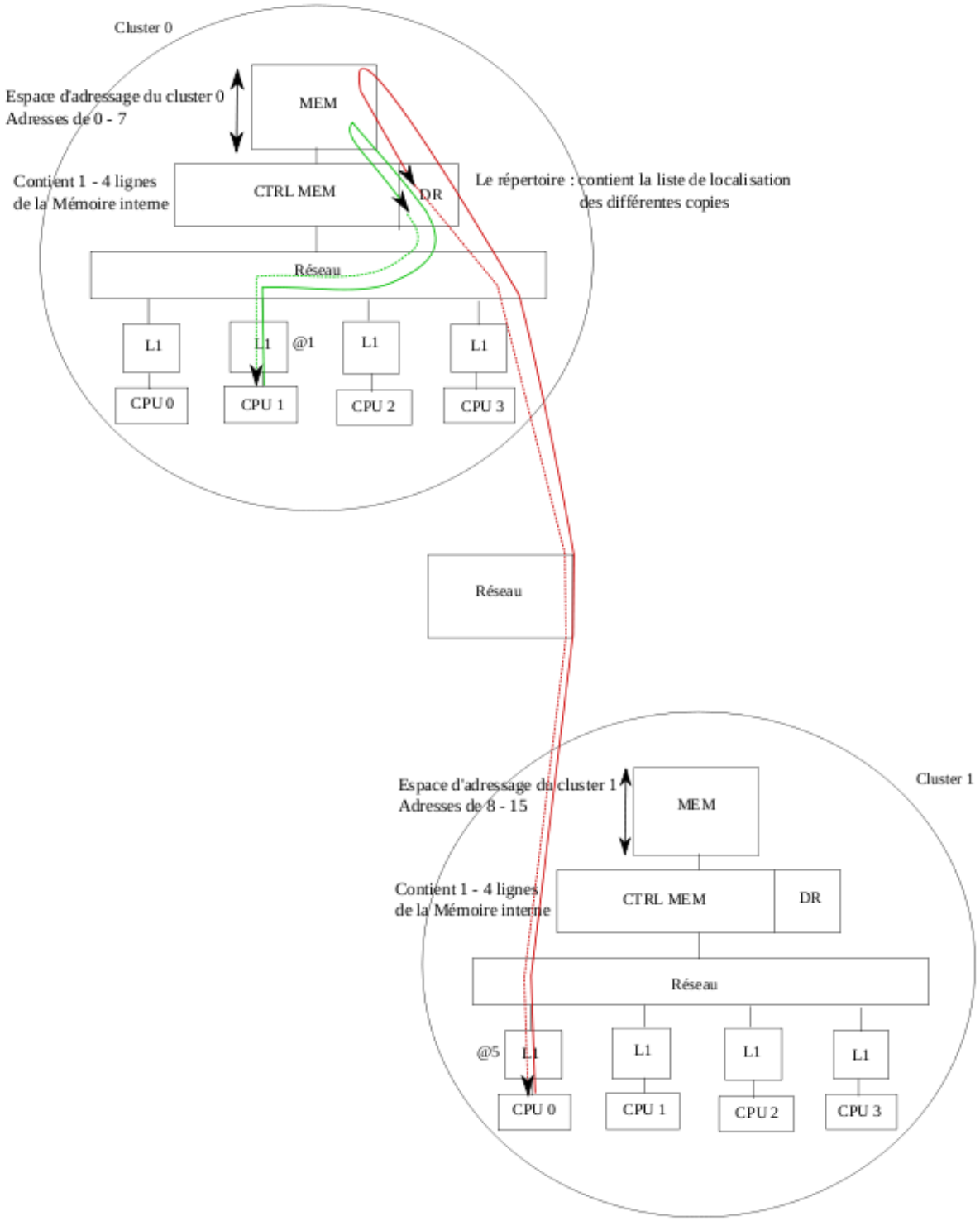

FIGURE 1.1 – Exemple de scénario du protocole DHCCP

c'est pourquoi les concepteurs du projet TSAR ont défini le protocole DHCCP : ce dernier permet la localisation d'un nombre limité de copies, paramétrable et défini par la suite par la constante TH. Cette constante représente le nombre maximum de copies qu'un même bloc mémoire peut localiser dans cette machine.

Il existe deux modes de maintien de la cohérence 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 L1 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 et répondent. Seul le cache L2 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 [3]. 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 le nombre d'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 cache L2 vers la Mémoire :** La nouvelle valeur est écrite uniquement dans la ligne du cache L2 et ne sera recopiée en Mémoire que si la ligne du cache L2 qui la contient est évincée.

1.1.3 Le model-checker Divine

divine est un *model-checker*, utilisé pour la vérification de systèmes décrits dans le langage DVE . Ce langage permet la description de processus communiquant via des canaux et/ou des variables globales.

divine est doté d'une interface graphique avec un simulateur et d'un analyseur de contre-exemple. **divine** offre la possibilité de vérifier des propriétés du système qui sont exprimées sous forme de formules LTL.

Un fichier DVE 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.

DVE supporte une composition asynchrone des différents processus.

1.1.4 Le model-checker gal

GAL est un langage pivot développé dans l'équipe Move et qui offre une syntaxe similaire à C pour décrire des systèmes concurrents. L'outil de *model-checking* ITS tools permettant l'analyse de propriétés CTL et LTL de modèles décrits en GAL basé sur une technologie à base de diagrammes de décision. Un éditeur riche intégré dans Eclipse permet de saisir les modèles et de lancer l'analyse. La description d'un système en GAL consiste en la définition de systèmes GAL élémentaires, et de leur composition possiblement hiérarchique.

Module GAL

Un système GAL contient des déclarations de variables et des transitions (éventuellement labelisées) qui ont une condition et une action, qui est une séquence d'affectations. Un état est défini comme une valuation des variables. Toute transition dont la condition est vraie dans l'état actuel peut être prise, donnant un ou plusieurs successeur(s), obtenu(s) par l'exécution de chaque affectation de la transition dans l'ordre. L'effet de chaque transition est atomique, à savoir que les états accessibles sont ceux obtenus après le calcul de tous les effets de la transition. Cette sémantique d'entrelacement, similaire à la sémantique des réseaux de Petri, est adaptée à la modélisation de systèmes concurrents.

Les caractéristiques du langage GAL sont les suivantes :

- Le variables sont toutes de type entier (32 bits). Elles peuvent également être des tableaux d'entiers de taille fixe.
- Le transitions sont constituées d'une condition (garde), d'un label (éventuellement vide), et d'une séquence d'affectations
- Les expressions supportées sont celles utilisant les opérateurs C sur les entiers
- La sémantique d'une transition est la suivante : si la condition est vraie, le corps de cette transition peut être exécuté (de manière atomique).

La syntaxe du langage est montrée sur l'exemple figure 1.2.

```
gal system {

    // Variable declarations
    int variable = 5 ;
    array [2] tab = (1, 2) ;


    // Transitions
    transition t1 [variable > 9] {
        self."a";
        tab [0] = tab [1] * tab [0] ;
        variable = variable * 5 ;
    }

    transition t2 [variable == 23] label "a" {
        tab [1] = 0 ;
    }
    transition t3 [variable < 23] label "a"  {
        tab [1] = 1 ;
    }
    transition t4 [true] {
        variable = variable * 2 ;
    }

}
```

FIGURE 1.2 – Exemple de programme en GAL

Lorsque l'on appelle un label "a" avec `self."a"`, le model checker va chercher toutes transitions du `GAL` portant ce label (t2 et t3 dans l'exemple), et vérifier la condition de ces transition. Si les conditions de plusieurs transitions sont vraies, une transition est choisie de façon non déterministe. Sinon, si une seule transition peut être prise, elle est choisie. Dans l'exemple, si la variable `variable` est égale à 23, seule la transition t2 est exécutable. Si la variable `variable` est inférieure à 23, seule t3 est exécutable. Si `variable` est supérieure à 23, aucune transition portant le label "a" n'est franchissable, et donc la transition t1 ne peut pas être exécutée.

Compositions

Un module `composite` contient une ou plusieurs instances d'autres modules, eux-même décrits en `GAL` ou comme des `composite`. L'état d'un composite est défini comme l'état de ses composants. Un fichier ".gal" peut contenir plusieurs définitions de modules, dont un est le point d'entrée principal (le `main`).

Un composite définit des synchronisations pour permettre la communication entre les sous-systèmes. Une synchronisation force les sous-systèmes à évoluer de façon

conjointe, en exploitant les labels des transitions.

La syntaxe du `composite` est montrée sur l'exemple figure 1.3.

```
gal system1 {
   int a = 1 ;
   transition system1_t1 [] label "t1"   {
      a++ ;
   }

}

gal system2 {
   int b = 1 ;
   transition system2_t2 []  label "t2"  {
      b-- ;
   }
}

composite c {
   system1 s1 ;
   system2 s2 ;

   synchronization syn0 [] {
       s1.t1 ;
       s2.t2 ;
   }

}

main c ;
```

FIGURE 1.3 – Exemple de composite

Dans ce modèle, on définit deux modules `GAL` `system1` et `system2`. Le module de nature composite "c" contient une instance de chacun de ces modules (nommés $s1$ et $s2$). Pour franchir la `sychronization` "syn0", $s1$ doit franchir une transition de label "t1" et $s2$ doit franchir une transition de label "t2".

Chapitre 2

Objectifs du Stage

2.1 Définition et analyse du problème

Des projets de recherche ont été faits autour de la vérification du protocole de cohérence DHCCP mais ont donné des résultats incomplets. La stagiaire Zahia Gharbi a fait un modèle de DHCCP en utilisant Divine, mais celui-ci s'est heurté à certaines limitations de ce langage.

La technique de model-checking de cet outil utilise une méthode d'énumération, qui atteint ses limites pour l'instance du problème que nous avons modélisé, à cause de l'explosion combinatoire du nombre d'états. La vérification de certaines configurations n'a en effet pas pu s'achever avec l'outil **divine** . Un des objectifs du stage est donc d'évaluer la capacité de passage à l'échelle de l'outil **gal** en utilisant une modélisation dédiée à cet outil, intégrant de la hiérarchie en particulier.

Comparé à `DVE` , `GAL` tire partie de la hiérarchie et de la symétrie présente dans le modèle. Pour modéliser le protocole en prenant en compte la hiérarchie – et espérer ainsi dépasser les limites de l'outil **divine** –, nous avons commencé par faire une réécriture des modèles sous forme de `GAL` (Guarded Action Language). La Figure 2.1 montre l'approche hiérarchique d'une modélisation `GAL` .

Sur cette figure, on voit qu'un automate ayant un sous-ensemble d'états et de transitions qui se répète peut se modéliser en ayant une seule représentation en mémoire de ce sous-ensemble (les transitions de la figure à droite pointent vers la même fonction). La difficulté associée à cette représentation consiste à trouver un bon ordre d'évaluation des variables, de manière à pouvoir factoriser au maximum des sous-ensembles d'un automate. La composition hiérarchique de gal et composite permet de guider l'outil vers un ordre d'évaluation particulier ; reste à trouver un ordre approprié qui permette un meilleur passage à l'échelle de l'analyse.

2.2 Résultats attendus

On espère obtenir les mêmes résultats de vérifiction sur les modèles `GAL` et `DVE` . On va comparer le temps que les solveurs **gal** et **divine** prennent pour vérifier un certain nombre de propriétés décrites dans la suite.

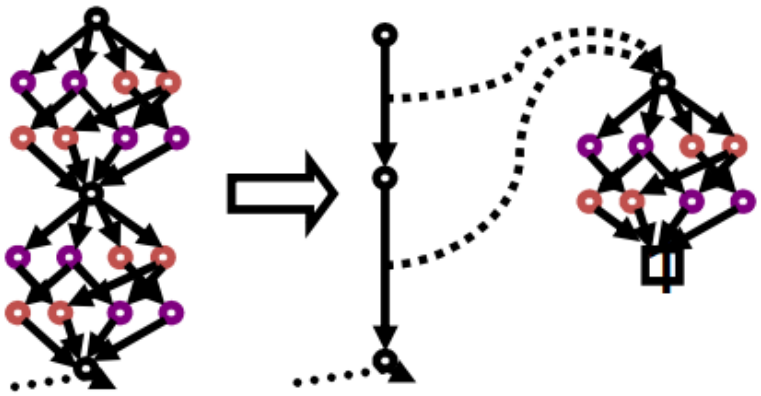

FIGURE 2.1 – Illustration du mécanisme de représentation hiérarchique de l'espace d'états dans GAL

On espère que l'écriture d'un modèle spécifique en GAL permette d'exploiter au mieux ce solveur et qu'il soit ainsi plus efficace. Si tel est le cas, on pourra donc vérifier des propriétés sur des configurations plus complexes du modèle que sur le modèle DVE .

2.3 Identification des tâches à accomplir et retro-planning

2.3.1 Tâche 1 : prise en main

- Lire les rapports et les documents décrivant le protocole TSAR, comprendre l'architecture que je vais modéliser.
- Apprendre à manipuler les outils et les langages (divine et gal) que je vais utiliser.

Objectif pleinement atteint

2.3.2 Tâche 2 : construction du modèle composant par composant

Le protocole DHCCP contient les composants suivants : les canaux, les processeurs, les caches L1, les caches L2 et la mémoire. Cette tâche nécessitera de réaliser :

- Un modèle de chaque composant, a priori sous la forme de modules GAL. Les modules seront configurables pour permettre leur réutilisation.
- Validation des modèles à travers des tests unitaires : écrire des testeurs pour valider que les modèles sont corrects vis-à-vis des comportements attendus. Pour ce faire, on tracera les séquences d'exécution du système en cherchant à couvrir tous les états et transitions de chaque module.

Prévisionnel 16/03 -> 30/04, durée 6 semaines

Recette Tâche 2 Livrer les composants élémentaires (modèles du processeur, des caches L1 et L2 et de la mémoire). Chaque composant sera accompagné de configurations de test (cf. section 3.2).

Objectif atteint au 15/05. La description du modèle obtenu est dans les sections 3.2 et 3.3. Ces composants ont été livrés dans une archive svn dans laquelle j'ai fait des commits réguliers au cours du stage. Chaque composant est décrit dans un fichier. Plusieurs testeurs ont été écrits pour les composants Cache L1 et Cache L2 afin d'en vérifier le bon fonctionnement, et sont intégrés dans ces fichiers.

2.3.3 Tâche 3 : assemblage du modèle complet

L'objectif à cette étape est d'assembler les composants définis à l'étape 2 pour construire des configurations de plus en plus réalistes.

On partira de configurations simples, avec un seul cache L1 et une seule adresse mémoire, puis on augmentera le nombre de caches et la taille de la mémoire. Le modèle compositionnel devra être facilement configurable pour permettre des expérimentations avec divers paramètres (dimensionnement et valeur du seuil `CACHE_TH`)

Enfin on pourra affiner le modèle selon les capacités de l'outil d'analyse.

Prévisionnel 01/05 -> 30/06 , durée 8 semaines

Recette Tâche 3 Livrer le modèle assemblé et configurable de l'architecture. Les configurations cibles iront jusqu'à 3 caches L1 et 2 adresses (et un seuil variable). (cf.)

Objectif atteint au 20/07. La description du modèle obtenu est dans le section 4.1. Comme il n'est pas possible des faire des "`import`" en `GAL` , le code de chaque composant a dû être recopié dans le(s) composant(s) de niveau supérieur. Malgré cette limite, le modèle hiérarchique a été bien testé et nous avons bonne confiance dans le fait qu'il ne contient plus de bugs. Ce modèle est paramétré selon les trois paramètres définis (`NB_CACHE`, `NB_MEM` et `CACHE_TH`), et les fichiers "flat" correspondant aux configurations testés ont été générés à la demande de Yann pour faciliter la réutilisation future dans le modèle checker.

2.3.4 Tâche 4 : ajout des propriétés

En parallèle de la tache précédente, on va définir et vérifier les propriétés du système.

Dans un premier temps, on analysera les propriétés de sûreté (dont l'absence de deadlocks), puis on validera des propriétés de vivacité plus complexes, exprimées en LTL (absence de famine, réactivité et équité...).

Prévisionnel 01/05 -> 15/06 , durée 6 semaines

Recette Tâche 4 Décrire dans un document les propriétés au format LTL, ainsi que les résultats d'analyse obtenus.

Objectif atteint au 4/08. La description des propriétés obtenu est dans la section 5.2.

2.3.5 Tâche 5 : généralisation

En fonction des résultats obtenus, proposer des annotations dans DVE et/ou des extensions pour GAL qui permettent d'exploiter au mieux les caractéristiques de l'outil, en particulier la description hiérarchique du système.

Prévisionnel 16/06 -> 31/07 , durée 6 semaines

Recette Tâche 5 Définir des annotations pour DVE et optimiser les transformations de DVE vers GAL .

Objectif atteint au 4/08. Cette tâche a été redéfinie vers des extensions du langage GAL permettant de plus facilement modéliser les communications entre composants (et tableau d'instances) (voir 6).

2.3.6 Tâche 6 : rapport

Prévisionnel 01/08 -> 31/08 , durée 4 semaines

Objectif atteint au 03/09. Cette tâche a été redéfinie vers des extensions du langage GAL permettant de plus facilement modéliser les communications entre composants (et tableau d'instances) (voir 6).

Chapitre 3

Modélisation de l'architecture TSAR

3.1 Architecture globale

L'architecture TSAR est modélisée comme montré sur la figure 3.1.

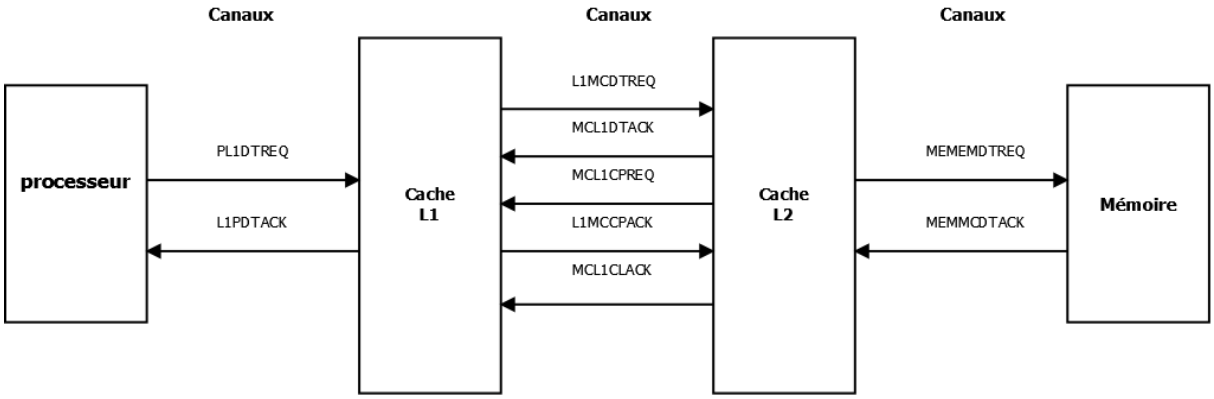

FIGURE 3.1 – Modélisation de l'architecture TSAR

Dans notre modélisation, la notion de cluster n'apparait pas car la latence ne nous intéresse pas. Ainsi, chaque processeur a une vision des caches L2 "à plat". Du point de vue d'un cache L2, tous les caches L1 sont similaires.

Le système présenté contient un processeur, un cache L1, un cache L2 et une mémoire. Tous ces éléments, ainsi que les canaux, seront modélisés comme des entités `GAL` . Par cette modélisation, on peut faire une composition hiérarchique de ces entités. Par exemple, les canaux entre un processeur et son cache L1 et les canaux entre un cache L2 et la mémoire contiennent 3 informations : si un message est présent ou non dans le canal, l'adresse du message, et son type. De cette manière, on n'a pas besoin de faire 4 modules pour les 4 canaux, mais on peut construire un seul module `GAL` s'appelant channelAddrType qui sera utilisé pour les 4.

Par exemple, la transition de EMPTY à MISS du cache L1, cache L1 a 4 tâches à faire :

– changer le statut de EMPTY à MISS ;

– recevoir la requête de lecture venant du processeur ;
– copier l'adresse reçue dans sa variable `save_addr` ;
– envoyer une requête au cache L2.

On peut écrire une synchronisation pour la réaliser.

3.2 Méthode de modélisation et de validation pour l'écriture du modèle GAL de DHCCP

Avec la transformation automatique de `DVE` à `GAL` , on peut obtenir un fichier `GAL` , mais ce n'est pas la modélisation optimale. En effet, ce fichier se contente de traduire la représentation `DVE` , et ne contient pas de hiérarchie. Il ne contient que des entités `GAL` , mais pas de synchronisation. Pour optimiser la représentation mémoire, il faut donc écrire un modèle à la main. Nous rappelons que l'objectif final du stage est de proposer une extension du langage `DVE` sous forme d'annotations pour permettre de faire une meilleure transformation de `DVE` vers `GAL` .

À cause du fait qu'il n'y a pas d'interface de simulation, on ne peut pas voir une exécution pas à pas de nos modèles. Ainsi, pour vérifier que la modélisation est correcte, il nous faut tester le modèle morceau par morceau. Pour cela, nous avons écrit 4 fichiers qui contiennent respectivement le processeur, le cache L1, le cache L2 et la mémoire. Pour tester ces modèles étape par étape, la solution est d'écrire des modules de test (ou *testeurs*) pour chaque processus.

Dans le cache L1, le mécanisme en cas de miss est le suivant :
– le processeur envoie une requête de lecture au cache L1 par le canal PL1DTREQ
– le cache L1 ne trouve pas la donnée
– l'état du cache L1 passe de EMPTY à MISS
– le cache L1 demande au cache L2 la donnée par le canal L1MCDTREQ
– le cache L2 reçoit la requête et renvoie la réponse par le canal MCL1DTACK
– Après avoir reçu cette réponse, l'état du cache L1 passe de MISS à VALID_DATA
– Enfin, il envoie la réponse au processeur par le canal L1PDTACK

Pour tester cette séquence, il nous faut écrire un *tester*, qui remplace le processeur et le cache L2, et effectue les communications afférentes à ces modules. Autrement dit, le tester va produire les messages qui devraient produire le processeur et le cache L2, et consommer les messages que ceux-ci devraient consommer, dans l'ordre de la séquence.

Si tout s'est bien passé, le testeur devrait atteindre son dernier état ; dans le cas contraire, il restera bloqué dans un de ses états intermédiaires. Cela peut facilement être vérifié à l'aide d'une propriété d'accessibilité.

3.3 Modélisation en GAL

L'architecture multiprocesseur que j'ai étudiée est complexe, mais certaines partie sont identiques, par exemple les caches L1 et processeur qui sont tous les mêmes. On souhaite exploiter cette symétrie pour rendre le modèle paramétrable, c'est-à-dire

changer le nombre de processeurs (dans cet exemple) en ne changeant qu'une valeur dans le code source du modèle.

Les paramètres que l'on a défini pour le modèle sont les suivants :
- Nombre de processeurs (= nombre de caches L1) : `NB_CACHES`
- Nombre de lignes de caches dans le L2 et la mémoire : `NB_MEM`
- Valeur du seuil de DHCCP : `CACHE_TH`

Dans un second temps, on cherchera à rendre ce modèle hiérarchique afin d'aider le solveur **gal** dans la représentation interne des états.

3.4 Composants GAL élémentaires

3.4.1 Canal channelIdAddrType

Ce `GAL` modélise un canal de communication comprenant trois informations :
- **id** : id du processeur faisant la requête où à destination de la requête
- **addr** : adresse de destination de la requête ou adresse cible d'une requête de cohérence
- **type** : type de la requête (directe ou de cohérence)

3.4.2 Canal channelAddrType

Ce `GAL` est canal de communication comprenant deux informations : **addr** et **type**. Ce canal est utilisé entre les caches L1 et les processeurs, ou le cache L2 et la mémoire, c'est pourquoi on n'a pas besoin d'information d'**id**.

Les `GAL` associés à ces canaux contiennent deux transitions : une pour écrire dans le canal et l'autre pour consommer un message présent (lecture du canal).

3.4.3 Processeur

Ce `GAL` comprend deux variables, **addr** et **state**. Il y a trois états *$PRO_IDLE*, *$PRO_WAIT_READ* et *$PRO_WAIT_WRITE*. La Figure 3.2 illustre l'automate du processeur.

Sur cette figure, on voit les transitions et les états d'un processeur. Lorsque le processeur émet une demande de lecture au cache L1, l'état change de *$PRO_IDLE* à *$PRO_WAIT_READ* . Lorsqu'il reçoit une réponse du cache L1, l'état passe de *$PRO_WAIT_READ* à *$PRO_IDLE*. De même, lorsque le processeur émet une demande d'écriture au cache L1, l'état change de *$PRO_IDLE* à *$PRO_WAIT_WRITE* ; et lorsqu'il reçoit une réponse, l'état retourne à *$PRO_IDLE*.

Le tableau 3.1 montre les états du processeur définis dans le `GAL` .

3.4.4 Cache L1

Il y a 4 variables dans le modèle `GAL` du Cache L1 :
- **state** contient l'état du cache L1 ;

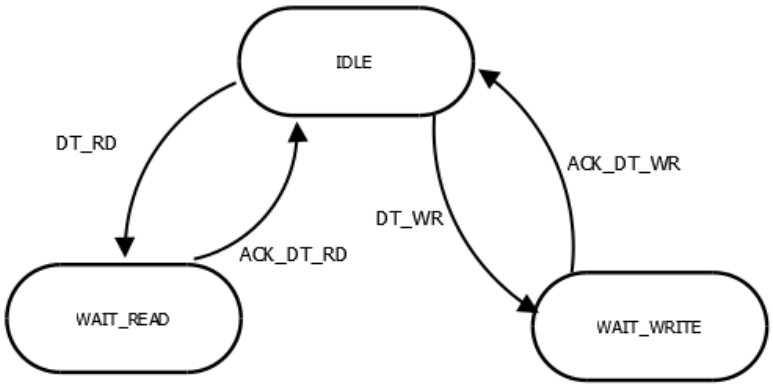

FIGURE 3.2 – Automate du processeur

Etat du processeur	Valeur correspondante
PROC_IDLE	0
PROC_WAIT_READ	1
PROC_WAIT_WRITE	2

TABLE 3.1 – Représentation du GAL processeur

- **v_addr** contient l'adresse de la ligne en cache lorsque celle-ci est valide (la validité est définie par l'état) ;
- **addr_save** permet de sauvegarder l'adresse d'une requête après l'envoi de cette requête ;
- **id** est l'identifiant de ce cache, il est différent pour chaque cache L1 et il est défini à l'initialisation.

La figure de l'annexe A.1 montre l'automate du cache L1.

Sur l'automate, on voit que la plupart des transitions comportent une émission et/ou une réception de message. Certains messages de différentes transitions utilisent le même canal. On peut utiliser cette caractéristique dans le modèle `GAL` : on classifie ces transitions en fonction des canaux que le(s) message(s) utilise(nt), et on leur donne un label comme `read_<canal>` si la transition consomme un message, `write_<canal>` si la transition produit un message, ou encore `read_<canal1>_write_<canal2>` si la transition consomme un message et en produit un autre. Le <canal> dans le label est le nom du canal dans lequel le message est lu ou écrit.

Par exemple, la transition de *$L1_EMPTY* à *$L1_MISS* du cache L1 s'écrit :

```
transition t_Empty_Miss  (id_t $id, addr_t $addr) [state == $L1_EMPTY && id == $id  ] label
"read_PL1DTREQ_write_L1MCDTREQ"($id, $addr, $DT_RD, $RD) {
    state = $L1_MISS ;
    addr_save = $addr;
}
```

Le cache L1 reçoit un message `DT_RD` du processeur et émet un message RD à destination du cache L2. Sur la table 5.1, on voit que le message `DT_RD` utilise le canal

PL1DTREQ et que le message RD utilise le canal L1PDTACK. Donc, on donne à cette transition le label read_PL1DTREQ_write_L1PDTACK.

Pour les transitions sans émission ou réception de message, on ne donne pas de label. Ces transitions sont donc franchissables tout le temps.

Le tableau 3.2 montre le codage des états du cache L1.

Etat du Cache L1	Valeur correspondante
INIT	0
L1_EMPTY	1
L1_MISS	2
L1_MISS_WAIT	3
L1_MISS_RETRY	4
L1_MISS_CLNUP	5
L1_VALID_DATA	6
L1_WRITE_WAIT_VALID	7
L1_WRITE_WAIT_EMPTY	8
L1_MISS_M_UP	9
L1_MISS_TO_RETRY	10
L1_ZOMBIE	11
L1_WRITE_WAIT_CLACK	12
L1_WRITE_WAIT_CLACK2	13
L1_MISS_RETRY_RD	14

TABLE 3.2 – Codage des états du cache L1

3.4.5 Cache L2

Le GAL du Cache L2 représente le comporement d'une ligne du cache L2, et comporte les variables suivantes :

- **state** est l'état de cache L2 ;
- **src_save** et **src_save_clnup** servent à sauvegarder des identifiants de cache L1 pour les requêtes nécessitant une requête de cohérence (src_save) et les requêtes de cleanup (src_save_clnup) ;
- **ligne_addr** est l'adresse de la ligne du cache L2 modélisée par le GAL . Avec un cache de 2 lignes, on doit avoir les automates CacheL2_0 (ligne_addr = 0) et CacheL2_1 (ligne_addr = 1) ;
- **n_copies** est le nombre de copies, i.e. le nombre de caches L1 ayant une copie de cette ligne ;
- **dirty** signifie que la ligne a été modifiée depuis qu'elle a été lue en mémoire ;
- **cpt**, **cpt_clnup** et **rsp_cpt** sont compteurs utilisés dans l'envoi et la réception de requêtes de type multicast ou broadcast.
- les tableaux **v_c_id[CACHE_TH]** et **c_id[CACHE_TH]** représentent la liste explicite des copies pour la ligne : si **v_c_id[i] == 1**, alors **c_id[i]** contient l'id du

cache ayant la copie ; sinon (**v_c_id[i]** == 0), l'entrée n'est pas utilisée. On remarque qu'une ligne du cache L2 peut avoir au maximum [CACHE_TH] copies explicites. Au-delà, la ligne passe en mode compteur.

La figure de l'annexe A.2 montre l'automate du cache L2.

Comme pour le cache L1, les transitions du GAL du cache L2 sont labellisées en fonction des lectures et écritures qu'elles font dans les canaux.

Le tableau 3.3 montre le codage des états du GAL du cache L2.

Etat du Cache L2	**Valeur correspondante**
MC_EMPTY	17
MC_READ_WAIT	18
MC_GET_WRITE_WAIT	19
MC_WRITE_WAIT	20
MC_VALID_MULTICAST	21
MC_VALID_MULTICAST_CLNUP	22
MC_VALID_BROADCAST	23
MC_VALID_MULTICAST_UPDATE	24
MC_VALID_BROADCAST_INV	25
MC_UPDATE_WAIT	26
MC_UPDATE_WAIT_CLNUP	27
MC_BROADCAST_INV_WAIT	28
MC_BROADCAST_INV_PUTD	29
MC_PUT_WAIT	30
MC_VALID_BROADCAST_INIT	31
MC_VALID_MULTICAST_READ	32
MC_VALID_MULTICAST_UPDATE_CLNUP	33

TABLE 3.3 – Codage des états pour le cache L2

3.4.6 Memoire

Le modèle de la mémoire est le plus simple. Lorsqu'il reçoit une demande de lecture (type GET), il renvoie une réponse de type ACK_GET. Lorsqu'il reçoit une demande d'écriture (type PUT), il renvoie une réponse de type ACK_PUT. Il n'y a donc que deux transitions avec le même label : "read_MCMEMDTREQ_write_MEMMCDTACK".

Chapitre 4

Assemblage et Plateformes

4.1 Compositions

4.1.1 Composite de niveau 1 entre le processeur et le cache L1

Le composite **`ProcessorCacheL1`** agrège un `GAL` processeur `Processor p`, un `GAL` du cache L1 `CacheL1 c` et 2 `GAL` modélisant les canaux de communications entre le cache L1 et le processeur `ChannelAddrType chan_PL1DTREQ` et `ChannelAddrType chan_L1PDTACK`.

Les transitions du processeur et du cache L1 sont encapsulées dans des synchronisations.

Dans le `GAL Processor`, il n'y a que deux labels : `write_PL1DTREQ` et `read_L1PDTACK`. Les canaux `chan_PL1DTREQ` et `chan_L1PDTACK` sont connectés avec le processeur. Lorsque `p` envoie un message par le canal `chan_PL1DTREQ`, le canal `chan_PL1DTREQ` doit écrire le message. Et lorsque le canal `chan_L1PDTACK` contient un message, `p` peut lire le message. Du coup, les deux synchronisations sont de la forme montrée sur la figure 4.1 (exemple pour l'écriture dans le canal).

```
synchronization s_write_PL1DTREQ (addr_t $addr,type_t $type) {
    p."write_PL1DTREQ" ($addr, $type) ;
    chan_PL1DTREQ."write" ($addr, $type) ;
}
```

FIGURE 4.1 – Synchronisation du composite Processeur – Cache L1 pour l'écriture dans le canal interne `chan_PL1DTREQ`

La première ligne de cette synchronisation peut déclencher toutes les transition du processeur ayant le label `write_PL1DTREQ`, soit une des 2 transitions de la figure 4.2.

Dans cette synchronisation, on doit utiliser des paramètres pour communiquer entre modules (un module est soit un `GAL` soit un composite).

On définit les paramètres après le nom de la transition (pour un `GAL`) ou synchronisation (pour un composite). Les types `adddr_t` et `type_t` sont les types de paramètres. Ces types sont définis au début de chaque fichier avec le mot clé `typedef`. Ces types

```
transition t_Idle_WaitRead (addr_t $addr) [state == $PROC_IDLE] label
"write_PL1DTREQ" ($addr, $DT_RD) {
    state = $PROC_WAIT_READ;
    addr = $addr;
}
transition t_Idle_WaitWrite (addr_t $addr) [state == $PROC_IDLE] label
"write_PL1DTREQ" ($addr, $DT_WR) {
    state = $PROC_WAIT_WRITE;
    addr = $addr;
}
```

FIGURE 4.2 – Transitions franchissables pour la première partie de la synchronisation

définissent le domaine de définition des variables, qui est un intervalle entier. La figure 4.3 montre les domaines de chaque type.

```
$NB_CACHES = 1 ;
$NBMEM = 1 ;
$CACHE_TH = 1 ;

typedef addr_t = 0 .. $NBMEM - 1 ;
typedef type_t = 0 .. 19 ;
typedef id_t = 0 .. $NB_CACHES - 1 ;
typedef isfull_t = 0..1;
typedef dirty_t = 0 .. 1 ;
typedef copie_t = 0 .. $NB_CACHES ;
typedef cpt_t = 0 .. $CACHE_TH ;
typedef valid_t = 0 .. 1 ;
```

FIGURE 4.3 – Domaines des types définis pout le modèle

`$addr` et `$type` sont les noms des paramètres.

Une représentation `GAL` paramétrée peut être "applatie" par le *front-end* (transformation dite *flatten*), et doit l'être avant d'être donnée au model-checker. Cette opération consiste à supprimer tous les paramètres en instanciant autant de transitions et synchronisations que de valeurs possibles pour les paramètres. Par exemple, dans la synchronisation de la figure 4.1, `$addr` peut être égale qu'à 0 (car `NB_MEM` = 1) et `$type` peut être égal à 0, 1, ..., 19. De fait, les synchronisations suivantes sont créées (le nom des synchronisation est changé pour contenir la valeur des paramètres instanciés) :

- `s_write_PL1DTREQ_0_0;`
- `s_write_PL1DTREQ_0_1;`
- ...
- `s_write_PL1DTREQ_0_19`.

Cependant, seuls les messages `DT_RD` (valeur correspondante : 0) `DT_WR` (valeur correspondante : 1) peuvent passer par le canal `PL1DTREQ`. Les types de message 2, 3, ..., 19 ne sont jamais utilisés dans cette synchronisation. La transformation *flatten* va les supprimer automatiquement. Finalement, seules les synchronisations représentées sur la figure 4.4 vont être créées.

```
synchronization s_write_PL1DTREQ_0_0 label "" {
  p."write_PL1DTREQ_0_0" ;
  chan_PL1DTREQ."write_0_0" ;
}
synchronization s_write_PL1DTREQ_0_1 label "" {
  p."write_PL1DTREQ_0_1" ;
  chan_PL1DTREQ."write_0_1" ;
}
```

FIGURE 4.4 – Synchronisations restantes après suppression des synchronisations non atteignables

Sur la figure 4.2, on peut remarquer que le paramètre `$DT_RD` du label n'est pas un paramètre de la transition. Il doit être défini au début du fichier en tant que constante. Dans ce cas, lors de la transformation *flatten*, il prend une unique valeur qui est celle définie. Il s'agit d'une facilité d'écriture (qui facilite aussi le travail de l'outil), mais on pourrait aussi écrire cette transition comme sur la figure 4.5.

```
transition t_Idle_WaitRead (addr_t $addr,type_t $type)
[state == $PROC_IDLE && $type == $DT_RD] label
"write_PL1DTREQ" ($addr, $type) {
    state = $PROC_WAIT_READ;
    addr = $addr;
}
```

FIGURE 4.5 – Autre écriture pour la transition "Idle_WaitRead"

La transition pourra être exécutée seulement dans le cas où `state == $PROC_IDLE` et `type == DT_RD`.

Les paramètres fixés `DT_RD` `DT_WR` des labels de ces transition garantissent les types des messages que les transitions vont écrire dans le canal.

La synchronisation `s_write_PL1DTREQ` n'a pas besoin de label car l'écriture se fait sur un canal interne, c'est-à-dire appartenant au composite ; l'interface de lecture et d'écriture exportée par le label ne doit donc pas être visible de l'exterieur. Du point de vue du composite, il s'agit d'une synchronisation ("transition") interne qui peut être franchie sans qu'il y ait d'échange de message avec les autres modules, qui ne sont donc pas modifiés.

Le `GAL` cache L1 comporte 10 labels différents : `read_MCL1CLACK`, `read_MCL1CPREQ`, `read_MCL1CPREQ_write_L1MCCPACK`, `read_MCL1DTACK`, `read_MCL1DTACK_write_L1PDTACK`, `read_PL1DTREQ_write_L1MCCPACK`, `read_PL1DTREQ_write_L1MCDTREQ`, `read_PL1DTREQ_write_L1PDTACK`, `write_L1MCCPACK` et `write_L1MCDTREQ`. Cela se traduit en autant de synchronisations au niveau du composite, avec des transitions des canaux associés aux labels. Les labels de ces synchronisations conservent la partie correspondant aux canaux qui n'appartiennent du composite et cachent la partie correspondant aux canaux internes au composite.

On sépare les transitions en trois types : Les transitions qui échangent un message avec le processeur, les transitions qui échangent un message avec le Cache L2 et les transitions qui éhangent un message avec le Cache L2 et un message avec le processeur.

Pour le premier type, il n'y a pas de label comme expliqué au-dessus. On synchronise la transition du cache L1 avec une transition labelisée "`read`" ou "`write`" du canal `chan_PL1DTREQ` ou `chan_L1PDTACK` (voir figure 4.6).

```
synchronization s_read_PL1DTREQ_write_L1PDTACK (addr_t $addr, type_t $type1, type_t $type2)
 {  chan_PL1DTREQ."read" ($addr, $type1) ; // DT_RD ou DT_WR
    c."read_PL1DTREQ_write_L1PDTACK"($addr, $type1, $type2);
    chan_L1PDTACK."write" ($addr, $type2) ; // ACK_DT_RD ou ACK_DT_WR
 }
```

FIGURE 4.6 – Exemple du premier type de synchronisation

Pour le deuxième type, il n'y a pas de canal correspondant dans le composite **`ProcessorCacheL1`**, on copie (exporte) donc le label sur la synchronisation, et ces synchronisations vont être appellées dans le composite de niveau supérieur (voir figure 4.7).

```
synchronization s_read_MCL1CPREQ_write_L1MCCPACK
(id_t $id, addr_t $addr, type_t $type1, type_t $type2) label
"c_read_MCL1CPREQ_write_L1MCCPACK"($id, $addr, $type1, $type2) {
    c."read_MCL1CPREQ_write_L1MCCPACK"($id,$addr, $type1, $type2);
}
```

FIGURE 4.7 – Exemple du deuxième type de synchronisation

Pour le troisième type, on appelle dans la synchronisation une transition ayant label `read` ou `write` du canal interne correspondant (`chan_PL1DTREQ` ou `chan_L1PDTACK`) et on exporte comme label pour la synchronisation la partie la correspondant au canal externe. ces synchronisations peuvent également être appelées dans le composite de niveau 2 (voir figure 4.8).

```
synchronization s_read_MCL1DTACK_write_L1PDTACK
(id_t $id, addr_t $addr, type_t $type1, type_t $type2) label
"c_read_MCL1DTACK"($id, $addr, $type1){
    c."read_MCL1DTACK_write_L1PDTACK"($id,$addr, $type1, $type2);
    chan_L1PDTACK."write" ($addr, $type2) ; // ACK_DT_RD ou ACK_DT_WR
}
```

FIGURE 4.8 – Exemple du troisième type de synchronisation

La Figure 4.9 montre la nomenclature des labels dans le composite **`ProcessorCacheL1`**.

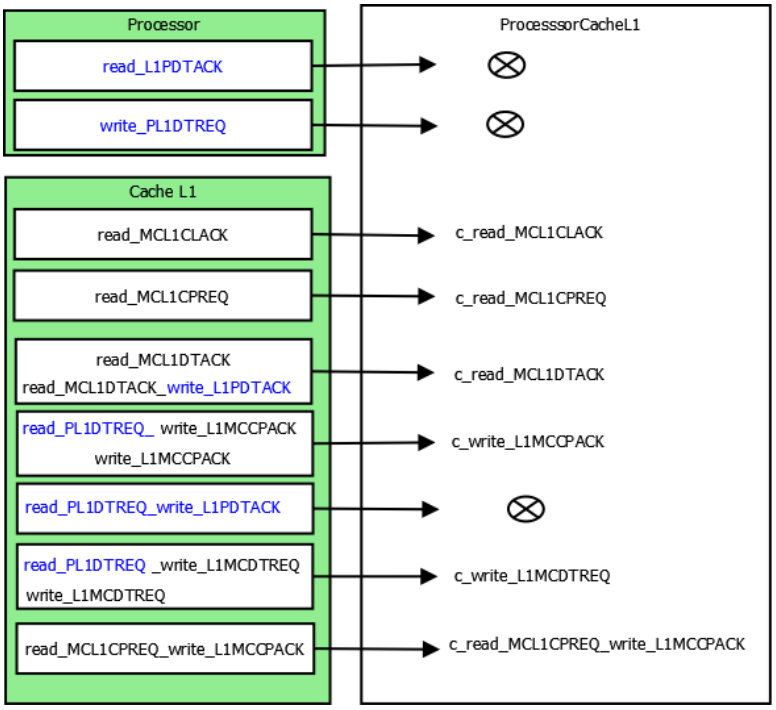

FIGURE 4.9 – Encapsulation des labels dans le composite ProcessorCacheL1

4.1.2 Composite de niveau 1 entre le cache L2 et la mémoire

Le composite **`CacheL2Memory`** définit un tableau de caches L2 `CacheL2 c[NB_MEM]`, le nombre de caches L2 est `$NBMEM` (parce que le cache L2 peut avoir plus d'une ligne d'adresse), un `GAL` modélisant la mémoire `Memory m` et 2 canaux de communication entre le cache L2 et la mémoire `ChannelAddrType chan_MCMEMDTREQ` et `ChannelAddrType chan_MEMMCDTACK`.

Ce composite comporte des synchronisations qui sont construites de la même manière que pour le composite `ProcessorCacheL1`.

La Figure 4.10 montre la nomenclature des labels dans le composite `CacheL2Memory`.

4.1.3 Composite de niveau 2 : Top

L'unique composite de niveau 2 modélise la plateforme TSAR complète. Il contient un tableau de composites `ProcessorCacheL1 pcl1[NB_CACHES]`, un composite `CacheL2Memory cl2m` et 5 canaux de communication de type `ChannelIdAddrType` entre les deux : `chan_L1MCDTREQ`, `chan_MCL1DTACK`, `chan_MCL1CPREQ`, `chan_L1MCCPACK`, `chan_MCL1CLACK`.

Les synchronisations du niveau maximum ne peuvent pas avoir de label (sinon ces synchronisations ne pourront jamais être appelées) ; elles appellent une transition comportant un label exporté par un des composites et une transition du canal correspondant (avec le label "`read`" ou "`write`" selon). Ainsi, lorsque `pcl1` envoie un message,

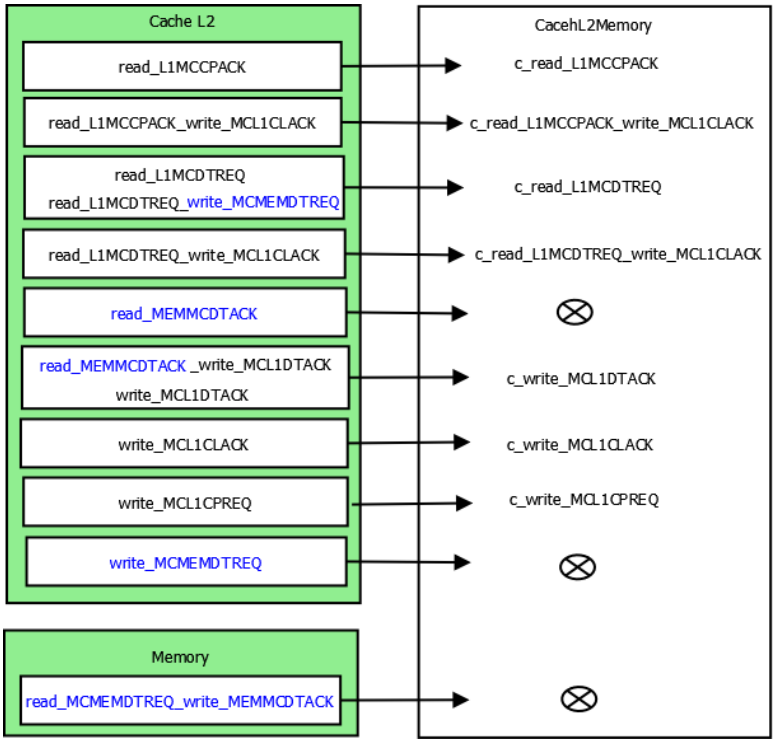

FIGURE 4.10 – Encapsulation de label dans le composite CacheL2Memory

le canal correspondant écrit le message, et plus tard lorsque `cl2m` veut lire ce message, ce dernier est consommé du canal.

La figure 4.11 montre un exemple de synchronisation dans le composite **`Top`**.

```
synchronization t0 (id_t $id, addr_t $addr, type_t $type) {
    cl2m."c_write_MCL1CLACK"($id, $addr, $type);
    chan_MCL1CLACK."write"($id, $addr, $type);
}
```

FIGURE 4.11 – Exemple de synchronisation dans le composite Top

La Figure 4.12 montre quant à elle un exemple de modèle complet hiérarchique à deux processeurs et une adresse mémoire.

Le tableau en Annexe A.2 récapitule les canaux de l'architecture.

4.2 Les différentes plateformes modélisées

4.2.1 Avec 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

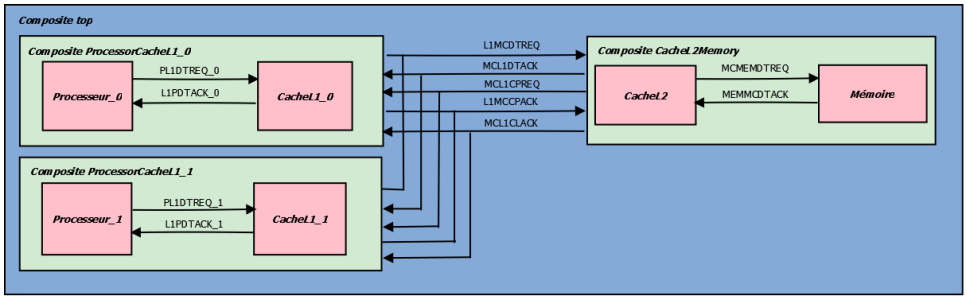

FIGURE 4.12 – Modèle complet hiérarchique de l'architecture TSAR, avec $NB_MEM = 1 et $NB_CACHE = 2

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 Cache L2 :** 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 4.13.

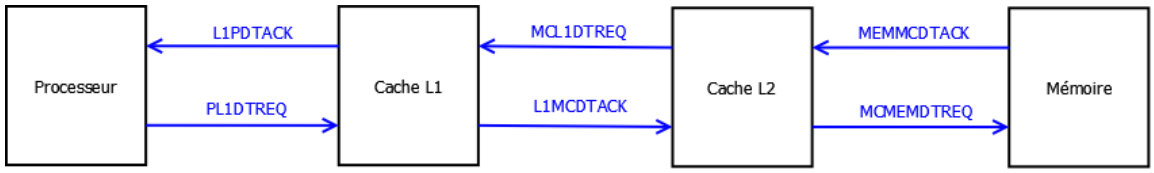

FIGURE 4.13 – Plateforme à un processeur et une adresse mémoire

4.2.2 Avec 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
- **Le Cache L2 :** 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 4.14.

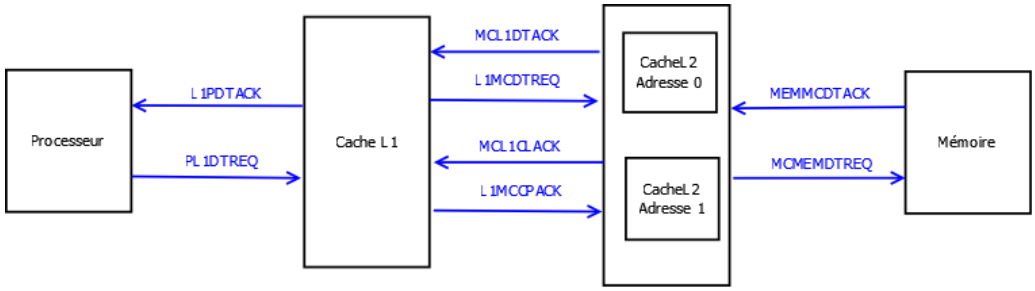

FIGURE 4.14 – Plateforme à un processeur et deux adresses mémoire

4.2.3 Avec 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 Processeur_0 et le Processeur_1 :** envoie des demandes de lecture et d'écriture du bloc mémoire d'adresse X = 0
- **Le Cache L1_0 et le Cache L1_1 :** Contient une seule ligne
- **Le Cache L2 :** 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 4.15.

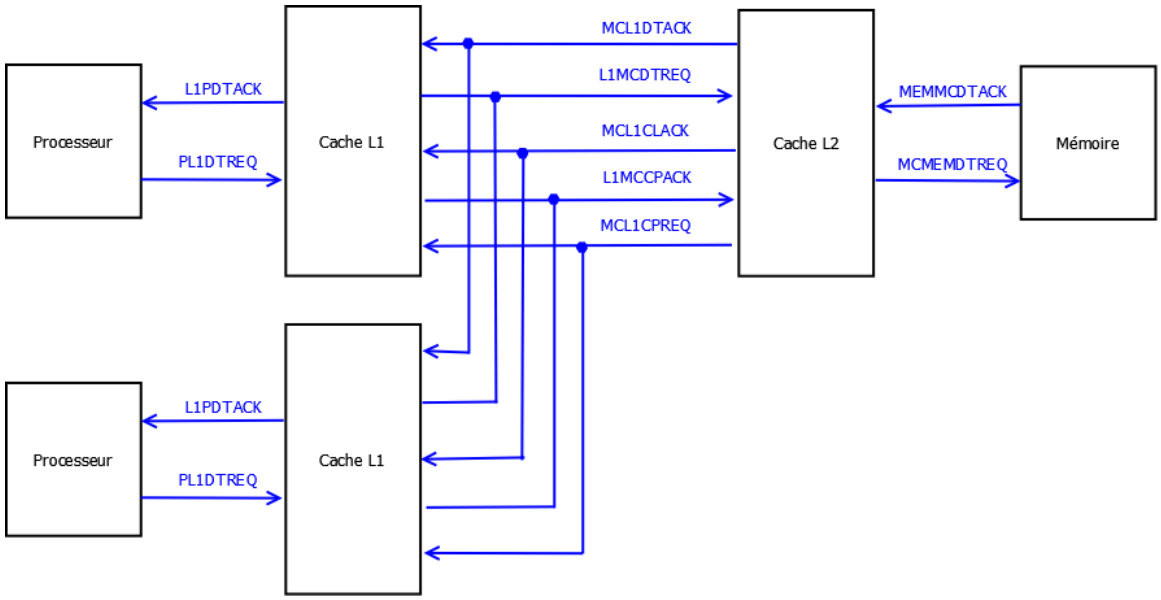

FIGURE 4.15 – Plateforme à deux processeurs et une adresse mémoire

4.2.4 Avec 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 Processeur_0 et le Processeur_1 :** envoie 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 :** Contient une seule ligne
- **Le Cache L2 :** 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 4.16.

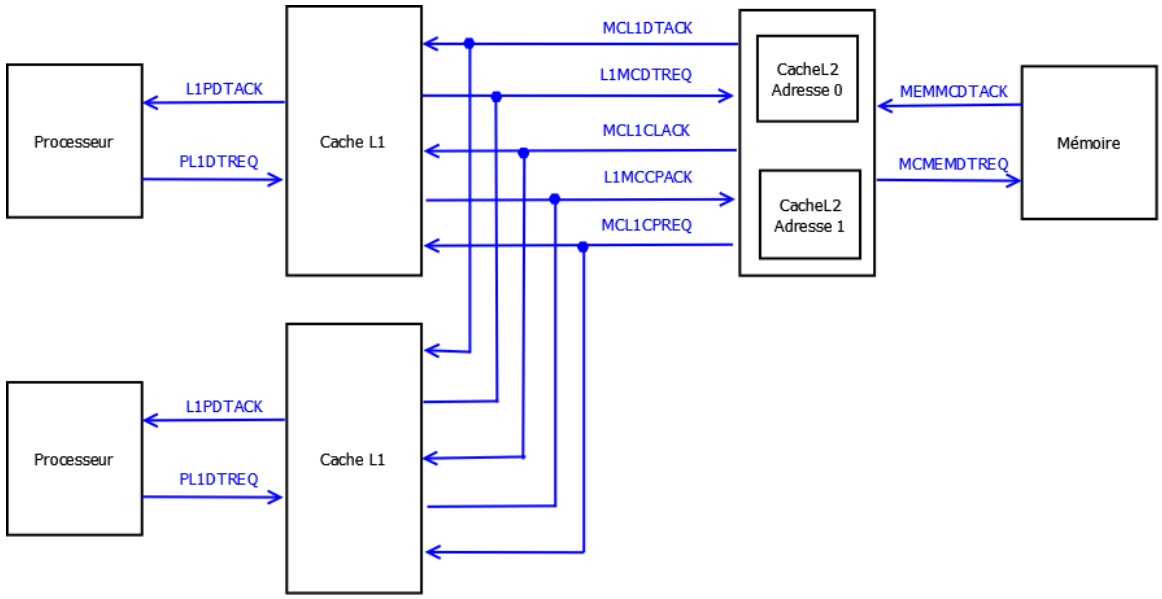

FIGURE 4.16 – Plateforme à deux processeurs et deux adresses mémoire

Chapitre 5

Validation et Analyse du modèle GAL

5.1 Validation des modèles

Pour vérifier les modèles GAL , j'ai défini des modules particuliers appelés *testeurs* pour chaque GAL qui n'est pas un canal (processeur, cache L1, cache L2 et mémoire). Un testeur réalise un scenario d'utilisation du composant à tester. L'écriture d'un testeur se base sur l'instanciation d'un GAL appelé testeur, mais nécessite de modifier les synchronisations du composite **top**, dans lesquelles vont se trouver toute la complexité. La difficulté est qu'en l'absence de simulateur pour GAL, les typos et autres erreurs de modélisation peuvent mener à des résultats qui ne sont pas fiables. Cette étape s'est appuyée sur un grand nombre de tests pour valider chacun des modules GAL séparément, puis pour valider leurs assemblages dans des composite.

Les figures A.3, A.4, A.5 et A.6 en annexe montrent 4 testeurs pour le Cache L2.

Un module GAL Tester du cache L2 est constitué de la manière suivante : il doit émuler le comportement des modules Cache L1 et Mémoire qui sont connectés avec ce Cache L2. Dans le GAL Tester, il n'y a qu'une variable d'état appelée state, incrémentée à chaque transition.

Je vais maintenant détailler une étape de l'instance tester_1. Chaque synchronisation dans le top commence par appeler une transition du GAL tester qui le fait changer d'état.

La figure 5.1 montre un exemple du début d'un tester du cache L2 comportant deux transitions, dans lequel le cache L2 reçoit un message RD et envoie un message GET.

Dans la première synchronisation, on force l'écriture d'un message RD dans le canal en provenance du cache L1 pour émuler son comportement. Dans la seconde synchronisation, il y a deux transitions à franchir (en plus de celle du tester) :

- une transition du cache L2 avec label read_L1MCDTREQ_write_MCMEMDTREQ. La transition du cache L2 qui va être franchie doit bien sûr porter ce label (pour "acquitter" la réception du message), mais aussi avoir sa garde à true, en particulier l'état testé dans la garde doit être l'état courant du cache
- une transition qui force la lecture du message GET dans le canal MCMEMDTREQ. On ne pourra franchir cette transition que si le cache L2 a bien écrit le message GET dans le canal, auquel cas on espère que son comportement est valide.

Au cas où le cache L2 a bien reçu le message RD et écrit le message GET dans le canal

```
//EMPTY
synchronization tester7_step_0 label "" {
    tester1."state_0" ;
    //c."empty";
    chan_L1MCDTREQ."write" (0, 0, $RD) ;
}
synchronization tester7_step_1 label "" {
    tester1."state_1" ;
    self."c_read_L1MCDTREQ_write_MCMEMDTREQ" ;
    chan_MCMEMDTREQ."read" (0, $GET) ;
}
//READ_WAIT
```

FIGURE 5.1 – Exemple de partie d'un testeur du cache L2

`chan_MCMEMDTREQ`, l'état du cache L2 a dû changer de `$MC_EMPTY` à `$MC_READ_WAIT`, et la variable `state` du `GAL testeur` doit être passée à 1.

À la fin du modèle, on écrit une propriété avec le mot clé `property` pour vérifier que le testeur a bien atteint les différents états du scenario (figure 5.2). On met une propriété par état pour le cas où le scenario n'arrive pas à son terme ; dans ce cas, on peut cibler plus précisément l'état du cache L2 qui pose problème.

```
property tester_1_success1 [reachable] : tester1 : state == 1 ;
property tester_1_success2 [reachable] : tester1 : state == 2 ;
property tester_1_success3 [reachable] : tester1 : state == 3 ;
property tester_1_success4 [reachable] : tester1 : state == 4 ;
property tester_1_success5 [reachable] : tester1 : state == 5 ;
property tester_1_success6 [reachable] : tester1 : state == 6 ;
property tester_1_success7 [reachable] : tester1 : state == 7 ;
property tester_1_success8 [reachable] : tester1 : state == 8 ;
property tester_1_success9 [reachable] : tester1 : state == 9 ;
```

FIGURE 5.2 – Propriétes d'atteinte des états du testeur

Si le modèle checker trouve que la propriété `state == 1` est correcte, on pourra lire : `Reachablity property tester_1_success1 is true.`

Cela montre que la partie modélisant la passage de l'état `$MC_EMPTY` vers l'état `$MC_READ_WAIT` est conforme à ce qui est attendu.

5.2 Propriétés et description en LTL

Cette section présente la modélisation des propriétés souhaitées du protocole DHCCP, en utilisant la logique temporelle linéaire (LTL). Cette logique et la logique CTL ont été utilisées au cours du stage pour analyser le modèle. Les propriétés présentées ici utilisent LTL pour permettre l'utilisation de l'équité dans **gal** . C'est également la seule logique que sait traiter le model checker **divine** , ce qui permet de comparer les résultats entre **divine** et **gal** .

Comme l'outil dans sa version actuelle n'intègre pas une gestion complète de la logique, il est nécessaire de traduire les identifiants de message en indices numériques. Les messages qu'on a utilisé en LTL sont type int. Le tableau 5.1 montre les types de message défini dans GAL et les valeurs correspondants à ces messages ; ces valeurs sont utilisées dans les formules au lieu des constantes symboliques.

TABLE 5.1 –

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

Propiété 1 (P1) :

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

G(F(("pcl1[0].p.state==1")+("pcl1[0].p.state==2")))

Propiété 2 (P2) :

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 "$PROC_IDLE"(donc il aura reçu un acquittement pour sa demande)

- étendue à un processeur :
  G((("pcl1[0].p.state==1")+("pcl1[0].p.state==2"))->F("pcl1[0].p.state==0"))
- étendue à deux processeur :
  G((((("pcl1[0].p.state==1")+("pcl1[0].p.state==2"))->F("pcl1[0].p.state==0"))
  +((("pcl1[1].p.state==1")U("pcl1[1].p.state==2"))->F("pcl1[1].p.state==0")))

Propriété 3 (P3) :

Pour chaque état de chaque séauence, si le Cache L1 est à l'état "$L1_MISS" alors à un moment dans le futur le Cache L1 et le Cache L2 passent respectivement à l'état "$L1_VALID_DATA" et "$MC_VALID_MULTICAST" (Le Cache L1 a bien reçu la donnée attendue et le Cache L2 a bien enregistré la présence d'une copie)

G(("pcl1[0].c.state==2")->F(("pcl1[0].p.state==6")*("cl2m.c[0].state==21")))

Propriété 4 (P4) :

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 mémorie d'adresse 0, alors à un moment dans le futur le Cache L1 envoie un $CLNUP au Cache L2 d'adresse 0, plus tard la ligne du cache L1 sera valide et contiendra une copie du bloc mémoire d'adresse 1

G((("pcl1[0].p.state==1")*("pcl1[0].c.state==2")*("pcl1[0].chan_PL1DTREQ.addr==1"))
->F(("chan_L1MCCPACK.type==8")*("chan_L1MCCPACK.addr==0")*("pcl1[0].c.v_addr==1")
****("pcl1[0].c.state==6"))***

Propriété 5 (P5) :

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 Cache L2 de l'adresse 1 passe a l'état "$MC_VALID_MULTICAST" et le compteur de copies est égale à 1

G((("pcl1[0].p.state==1")*("pcl1[0].chan_PL1DTREQ.addr==1")) -
>F(("cl2m.c[1].state==21")*("cl2m.c[1].n_copies==1")))

Propriété 6 (P6) : fairness

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 de d'adresse 0, et que le processeur 0 envoie une demande d'écriture sur ce même bloc alors à un

moment dans le futur le Cache L2 envoie un $M_UP pour le cache L1_1 pour mettre à jour sa ligne

G((("pcl1[0].c.state==6")*("pcl1[1].c.state==6")*("pcl1[0].p.state==2"))
->F(("chan_MCL1CPREQ.addr==0")*("chan_MCL1CPREQ.type==12")
****("chan_MCL1CPREQ.id==1")*("chan_MCL1CPREQ.addr==0")))***

Propriété 7 (P7) : fairness

Pour tous 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 de d'adresse 0, et que le processeur 1 envoie une demande d'écriture sur ce même bloc alors à un moment dans le futur le Cache L2 envoie un $M_UP pour le cache L1_0 pour mettre à jour sa ligne

G((("pcl1[0].c.state==6")*("pcl1[1].c.state==6")*("pcl1[1].p.state==2"))
->F(("chan_MCL1CPREQ.addr==0")*("chan_MCL1CPREQ.type==12")
****("chan_MCL1CPREQ.id==0")*("chan_MCL1CPREQ.addr==0")))***

Propriété 8 (P8) :

Pour tout état de chaque séquence, le Cache L2 n'envoie jamais des messages d'invalidation($B_INV et $M_INV)

G(("chan_MCL1CPREQ.type !=10")*("chan_MCL1CPREQ.type !=11"))

Propriété 9 (P9) : fairness

Pour tout état de chaque séquence équitable, si le nombre de copies dans les caches est égal à 2, alors le Cache L2 est dans l'état "$MC_VALID_BROADCAST"

G(("cl2m.c[0].n_copies==2")->("cl2m.c[0].state==23"))

Propriété 10 (P10) : fairness

Pour tout état de chaque séquence équitable,si le Cache L2 de l'adresse 0 est à l'état "$MC_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 $B_INV pour le cache L1_0 et un autre pour le cache L1_1.

G((("cl2m.c[0].state==23")*("chan_L1MCDTREQ.type==5")*("cl2m.c[0].n_copies>0"))
->((F(("chan_MCL1CPREQ.type==10")*("chan_MCL1CPREQ.id==0")))
****(F(("chan_MCL1CPREQ.type==10")*("chan_MCL1CPREQ.id==1")))))***

Propriété 11 (P11) : fairness

Pour tout état de chaque séquence équitable, si le processeur 0 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

***G((("pcl1[0].p.state==1")*("pcl1[0].c.state==2")*("pcl1[0].chan_PL1DTREQ.addr==1"))
->F(("pcl1[0].c.state==6")*("pcl1[0].c.v_addr==1")))***

Propriété 12 (P12) : fairness

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 $L1_ValidData et contient une copie du bloc mémoire d'adresse 0, alors le cache L1 reste dans cet état et contient une même copie jusqu'à ce que le processeur0 envoie une demande de lecture du bloc mémoire d'adresse 1.

***G((("pcl1[0].p.state==2")*("pcl1[0].c.state==6")*("pcl1[0].c.v_addr==0")) -
>(("pcl1[0].c.state==6")*("pcl1[0].c.v_addr==0))U("pcl1[0].c.state==2")))***

Propriété 13 (P13) : fairness

Pour tout état de chaque séquence équitable,si le processeur0 envoie une demande de lecture du bloc mémoire d'adresse 0 et que le cache L2 de l'adresse 0 est l'état $MC_EMPTY, alors à un moment dans le futur le cache L2 passe à l'état $MC_VALID_MULTICAST avec un nombre de copies égal à 1

***G((("pcl1[0].p.state==1")*("pcl1[0].chan_PL1DTREQ.addr==0)*("cl2m.c[0].state==17"))
->F(("cl2m.c[0].state==21")*("cl2m.c[0].n_copies==1")))***

Propriété 14 (P14) : fairness

Pour tout état de chaque séquence équitable, si le processeur0 envoie une demande d'écriture du bloc mémoire d'adresse 0 et que le Cache L2 de l'adresse 0 est à l'état $MC_EMPTY alors à un moment dans le futur le Cache L2 passe à l'état $MC_VALID_MULTICAST avec un nombre de copies égal à 0.

***G((("pcl1[0].p.state==2")*("pcl1[0].chan_PL1DTREQ.addr==0)*("cl2m.c[0].state==17"))
->F(("cl2m.c[0].state==21")*("cl2m.c[0].n_copies==0")))***

Propriété 15 (P15) : fairness

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 Cache L2 envoie un $M_UP pour le cache L1_0 pour mettre à jour sa ligne.

***G((("pcl1[0].c.state=6")*("pcl1[0].c.v_addr==1")*("pcl1[1].c.state=6")
("pcl1[1].c.v_addr==1")("pcl1[1].p.state==2")*("pcl1[1].chan_PL1DTREQ.addr==1"))
->F(("chan_MCL1CPREQ.addr==1")*("chan_MCL1CPREQ.type==12")
*("chan_MCL1CPREQ.id==0")))***

Propriété 16 (P16) : fairness

Pour tout état de chaque séquence équitable, si les lignes des deux caches(cacheL1_0 et cacheL1_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 Cache L2 envoie un $M_UP pour le cache L1_1 pour mettre à jour sa ligne

G((("pcl1[0].c.state=6")*("pcl1[0].c.v_addr==1")*("pcl1[1].c.state=6")
****("pcl1[1].c.v_addr==1")*("pcl1[0].p.state==2")*("pcl1[0].chan_PL1DTREQ.addr==1"))***
->F(("chan_MCL1CPREQ.addr==1")*("chan_MCL1CPREQ.type==12")
****("chan_MCL1CPREQ.id==1")))***

5.3 Évaluation de performance

Nous avons mesuré les temps de vérification de chaque propriété en `GAL` (avec le *model-checker* **gal**) et en `DVE` (avec le *model-checker* **divine**). Cette mesure s'est faite avec la commende `time` de linux. Le tableau 5.2 contient la comparaison pour la vérification des propriétés décrites section 5.2 et le tableau 5.3 contient les résultats pour la vérification de l'absence de *deadlock*.

D'après le tableau 5.2, on voit que le model-checker **gal** prend plus de temps que le model-checker **divine** . Sur le modèle avec les configurations `NB_MEM = 2`, `NB_CACHE = 3`, on a attendu 20 heures mais sans obtenir de résultat.

Toutes les vérifications des propriétés comprenant une hypotèse d'équité (*fairness*) échouent avec **gal** et **divine** (mais sont vérifiées dans le stage de Z. Gharbi), car **gal** comme l'outil **divine** utilisé ne possèdent pas la possibilité d'introduire une notion d'équité. Les temps de vérification peuvent néanmoins servir de base à la comparaison.

TABLE 5.2 – Comparaison des résultats de performances entre gal et divine pour les propriétés LTL

Configuration	Propriété	Divine	Gal
$NB_CACHE = 1 $NB_MEM = 1 $CACHE_TH = 1	P1	0m0.099s	0m0.146s
	P2	0m0.113s	0m0.060s
	P3	0m0.113s	0m0.061s
$NB_CACHE = 1 $NB_MEM = 2 $CACHE_TH = 1	P2	0m0.157s	0m0.174s
	P4	0m0.154s	0m0.183s
	P5	0m0.153s	0m0.219s
$NB_CACHE = 2 $NB_MEM = 1 $CACHE_TH = 2	P2	0m0.176s	0m1.365s
	P6	0m0.185s(NOT hold)	0m0.400s
	P7	0m177s(NOT hold)	0m0.447s
	P8	0m0.160s	0m0.311s
$NB_CACHE = 2 $NB_MEM = 1 $CACHE_TH = 1	P9	0m1.028s(NOT hold)	0m0.156s
	P10	0m0.203s(NOT hold)	0m0.248s
$NB_CACHE = 2 $NB_MEM = 2 $CACHE_TH = 2	P11	0m0.429s	1m40.869s
	P12	0m0.614s(NOT hold)	1m12.748s
	P13	0m0.675s(NOT hold)	4m32.792s
	P14	0m1.466s(NOT hold)	9m38.994s
	P15	0m3.264s(NOT hold)	1m35.421s
	P16	0m0.858s(NOT hold)	1m41.944s

TABLE 5.3 – Comparaison des résultats de performances entre gal et divine pour l'absence de deadlock

NB_CACHE	NB_MEM	CACHE_TH	Divine	GAL
1	1	1	0m0.098s	0m0.043s
1	2	1	0m0.134s	0m0.138s
2	1	2	0m0.159s	0m0.376s
2	1	1	0m0.353s	0m0.349s
2	2	2	0m17.637s	1m17.627s
3	1	3	0m0.541s	0m20.158s
3	1	1	0m0.609s	0m27.104s
3	1	2	0m0.743s	0m15.515s
3	2	3	0m44.446s	-
3	2	1	2m11.439s	-
3	2	2	3m38.362	-

Chapitre 6

Généralisation de l'approche

Un des objectifs du stage est une évaluation empirique de l'utilisation en situation du langage `GAL` et `DVE` et des outils d'analyse liés. On n'a pas généralidé vers `DVE` finalement, mais plutôt étendu `GAL` pour permettre de plus facilement modéliser le type de système qui nous intéresse. Un certain nombre d'extensions à la fois au langage et à l'outillage ont été nécessaires pour mener le travail à son terme. Ces réalisations ont été faites avec mon encadrant pour servir mes besoins en modélisation.

6.1 Labels avec paramètres

Les labels de `GAL` au début du stage ne permettaient pas de définir des paramètres. Cette limite empêchait une modélisation facile des échanges de données entre composants, à l'aide de synchronisations en particulier.

Par exemple, sans possibilité d'utiliser des paramètres dans les labels, la transition `t_Idle_WaitRead` montrée sur la figure 4.2 nécessite d'écrire des transitions pour chaque adresse, comme montré figure 6.1 (on suppose que `NB_MEM = 2`).

```
transition t_Idle_WaitRead_0 [ state == 0 ] label "write_PL1DTREQ_0_0" {
  state = 1    ;
  addr = 0     ;
}
transition t_Idle_WaitRead_1 [ state == 0 ] label "write_PL1DTREQ_1_0" {
  state = 1    ;
  addr = 1     ;
}
```

FIGURE 6.1 – Exemple de label si on ne peut pas utiliser de paramètres

On donc a étendu les mécanismes de définition de paramètres pour les transitions qui existaient déjà, pour intégrer des paramètres au sein des labels de transition. Cette approche permet une modélisation paramétrée dans le sens où la même modélisation reste valable quel que soit le nombre de messages.

6.2 Tableaux d'instances et taille paramétrable des tableaux

La deuxième extension importante qui a été réalisée concerne les tableaux. D'une part, les tableaux ne portaient au début du stage que sur les variables entières et ne pouvaient pas contenir des éléments de type instances du langage GAL ;d'autre part, la taille des tableaux n'était pas paramétrable par une constante symbolique, mais devait être un entier fixé en dur. Ces deux points étaient des limitations importantes :

- Les tableaux d'instances étaient nécessaires pour pouvoir rendre génériques les synchronisations portants sur plusieurs instances du même `GAL` (sans quoi, il faut écrire une synchronisation par `GAL`)
- Avoir une taille paramétrable était moins critique, mais ne pas en avoir obligeait à changer à la main la taille des tableaux dans dès que l'on voulait changer les paramètres `CACHE_TH` ou `NB_MEM`.

Ainsi, Yann a ajouté ces deux aspects suite aux discussions que l'on a eues. Par exemple, le composant `CacheL2Memory` contient un tableau de caches L2, comme illustré sur la première ligne de la figure 6.2.

```
CacheL2 [$NBMEM] c;

synchronization initialization label "init" {
    for ($addr : addr_t) {
        c[$addr]."init" ($addr);
    }
}
```

FIGURE 6.2 – Exemple de déclaration d'un tableau d'instances

6.3 Initialisation des tableaux d'instances

Un problème posé par les tableaux d'instances est qu'il sont initialisés statiquement, et tous élements (instances `GAL`) du tableau contiennent les mêmes valeurs de variables, en particulier la variable `id` censée les différencier. Pour résoudre ce problème, nous avons introduits dans les modules des caches L1, L2 et les composites de niveau 1 une transition labelisée `init` éventuellement paramétrée par l'identifiant du module. On force la prise de cette transition depuis l'état initial, comme illustré figure 6.2 pour le module `CacheL2Memory`.

Lors de ma première modélisation des caches L1 et L2, les communications étaient modélisées d'une manière différente, dans laquelle les modules `GAL` étaient triviaux, li-

mités à contenir les variables et à la structure de l'automate (transitions sans les gardes ni actions). Le composite au contraire manipulait ces instances par des séquences complexes d'invocations de labels associés à des actions simples de lectures et écritures de variables du GAL .

Cela reportait donc toute la complexité des GAL dans les composites de niveau supérieur, et n'était pas une bonne modélisation pour 2 raisons :

- D'un point de vue de l'interface du GAL , ce modèle exposait le comportement du module à son environnement, en particulier les transitions entre états, ainsi que les gardes et les actions associées. Les modèles GAL étaient donc peu réutilisables, par exemple dans le cadre des tests décrits section 3.2.
- Si l'on a plusieurs façons de lire ou d'écrire dans un canal, le nombre de synchronisations qu'il faut exprimer peut croitre de manière polynomiale.

À titre d'exemple, la figure 6.3 montre la première modélisation d'une synchronisation, qui contient plein d'invocations de labels, faisant chacun une action simple sur le composant (test d'une variable, mise à jour d'une variable).

```
synchronization MC_ValidMulticastUpdateClnup_ValidMulticastUpdate2( id_t $id,addr_t $addr,cpt_t $cpt)
label "read_MCL1CLACK" {
   mc0."ValidMulticastUpdateClnup_ValidMulticastUpdate" ;
   mc0."check_cpt_clnup_less_than_CACHE_TH";
   mc0."read_cpt_clnup"($cpt);
   mc0."check_v_c_id"($cpt,1);
   mc0."check_c_id_equal"($cpt,$id);
   mc0."read_src_save_clnup"($id);
   self."c_read_MCL1CLACK"($id,$addr,$CLACK);
   mc0."write_v_c_id"($cpt,0);
   mc0."write_c_id"($cpt,0);
   mc0."substraction_n_copies";
   mc0."write_src_save_clnup"(0);
}
```

FIGURE 6.3 – Exemple de synchronisation complexe avant restructuration du modèle

La figure 6.4 montre que ces actions ont été déportés vers la transition du cache L2 correspondante (qui était vide avant). La synchronisation qui utilise cette transition est similaire à celle représentée sur la figure 4.1.On peut observer que l'on n'a plus besoin de connaitre l'intérieur du cache L2 pour le synchroniser.

```
transition t_ValidMulticastUpdateClnup_ValidMulticastUpdate2 (id_t $id, addr_t $addr)
      [state == $MC_VALID_MULTICAST_UPDATE_CLNUP && cpt_clnup < $CACHE_TH &&
      v_c_id[cpt_clnup] == 1 && c_id[cpt_clnup] == src_save_clnup &&
      src_save_clnup == $id && n_copies > 0 && ligne_addr == $addr]
      label "write_MCL1CLACK" ($id, $addr, $CLACK) {
    state = $MC_VALID_MULTICAST_UPDATE ;
    v_c_id[cpt_clnup] = 0 ;
    c_id[cpt_clnup] = 0 ;
    n_copies = n_copies - 1;
    src_save_clnup = 0 ;
    cpt_clnup = 0 ;
}
```

FIGURE 6.4 – Transition du cache L2 correspondante après la restructuration

Chapitre 7

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, architectures multiprocesseur) et d'en acquérir de nouvelles notamment dans les techniques de model checking. Ce stage également m'a permis de découvrir comment modéliser un système en automates dans les langages `DVE` et `GAL` et comment effectuer des vérifications avec les outils **divine** et **gal** . Du côté de l'équipe MoVe, je pense que ce stage a constitué un cas d'utilisation réel du langage `GAL` , dont Yann a su tirer parti pour l'améliorer.

Malheureusement, nous n'avons pas pu obtenir les résultats que l'on espérait au niveau des performances, mais Yann va pouvoir analyser le modèle pour déterminer d'où viennent ces performances décevantes. Enfin, si le langage `GAL` a répondu aux besoins, il n'est pas encore complètement mature et profitera encore surement d'améliorations dans le futur. En particulier :

- L'ajout d'un mécanisme de simulation interactive serait utile pour faciliter le debug et avoir une meilleure idée de ce que font les modèles que l'on a écrits.
- La logique LTL n'est pas entièrement prise en compte et son parseur est très limité
- Un mécanisme d'import de modules d'un fichier dans un autre fichier serait très utile pour mieux organiser le code. L'absence de ce mécanisme m'a obligée à recopier le code des modules d'un fichier à un autre après chaque modification, et à avoir des modules top très conséquents en taille.

Annexe A

A.1 Automate des Cache L1 et L2

A.2 Nomenclature et formats des canaux

A.3 Testeurs du Cache L2 en détail

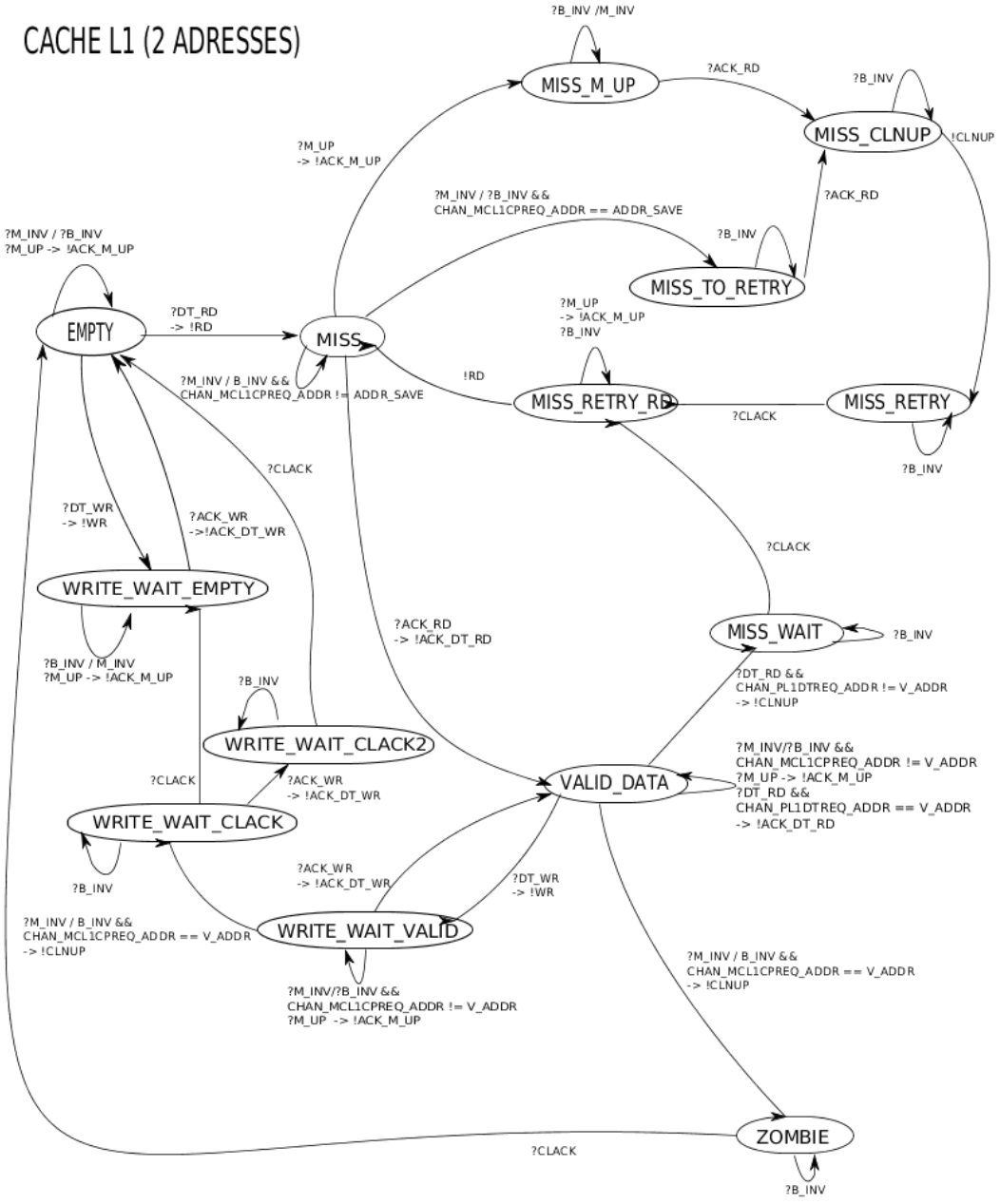

FIGURE A.1 – Automate de Cache L1

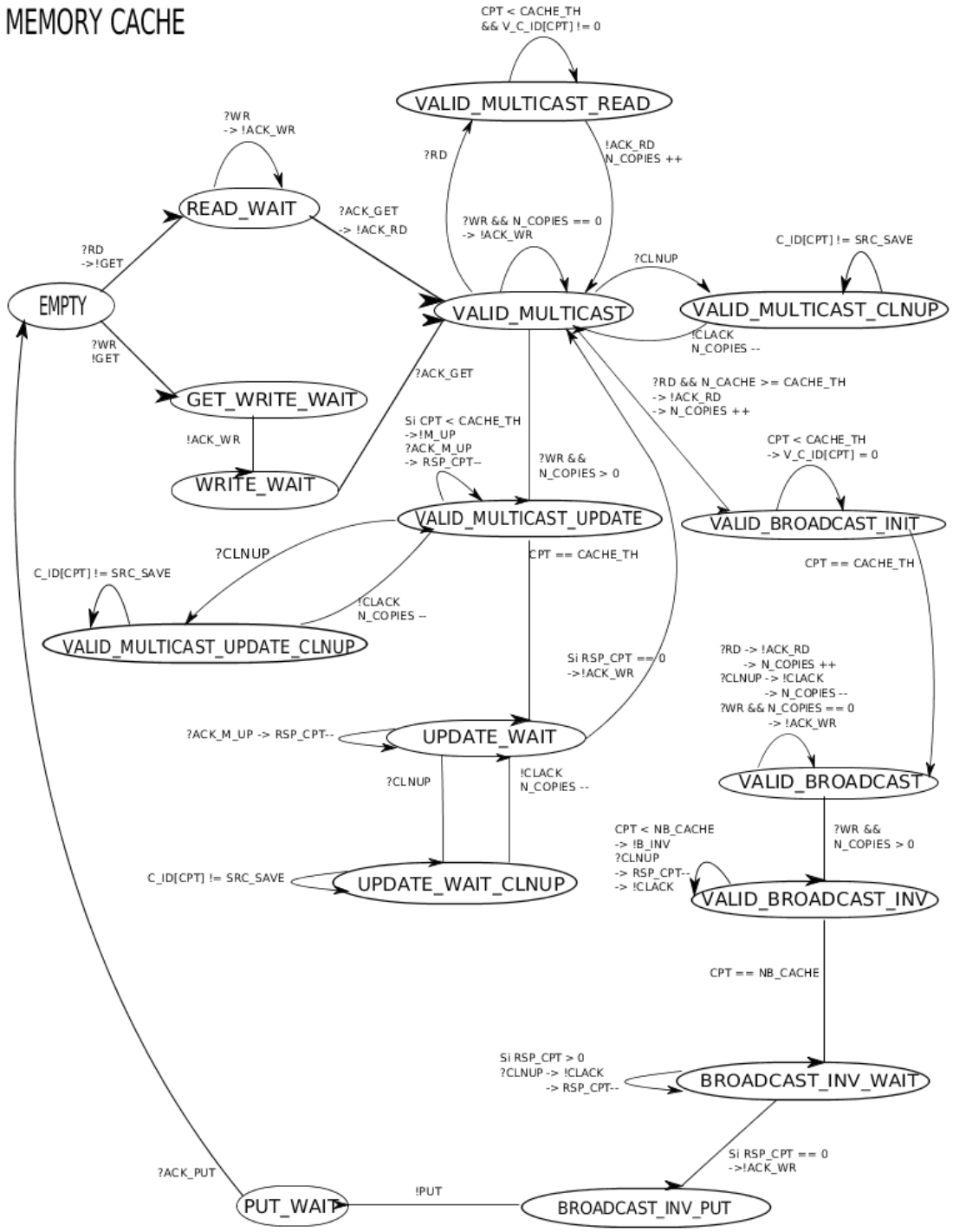

FIGURE A.2 – Automate de Cache L2

Nom de canal	Type de canaux	Source	Destination	Type de message	Utilisation
PL1DTREQ	chanAddrType	Processeur	Cache L1	. DT_RD DT_WR	Envoyer les messages de lecture et d'écriture
L1PDTACK	chanAddrType	Cache L1	Processeur	. ACK_DT_RD ACK_DT_WR	Répondre aux messages de lecture et d'écriture
L1MCDTREQ	chanIdAddrType	Cache L1	Cache L2	. RD WR	Envoyer les messages de lecture et d'écriture
MCL1DTACK	chanIdAddrType	Cache L2	Cache L1	. ACK_RD ACK_WR	Répondre aux messages de lecture et d'écriture
MCL1CLACK	chanIdAddrType	Cache L2	Cache L1	CLACK	Répondre aux demandes de Clean Up
MCL1CPREQ	chanIdAddrType	Cache L2	Cache L1	. M_UP B_INV M_INV	Envoyer le messages d'invalidation et de mise à jour
L1MCCPACK	chanIdAddrType	Cache L1	Cache L2	ACK_M_UP ACK_B_INV ACK_M_INV	Répondre aux messages de cohérence
MCMEMDTREQ	chanAddrType	Cache L2	Mémoire	. GET PUT	Envoyer les demandes de lecture et d'écriture
MEMMCDTACK	chanAddrType	Mémoire	Cache L2	. ACK_GET ACK_PUT	Répondre aux messages de lecture et d'écriture

TABLE A.1 – Nomenclature et formats des canaux

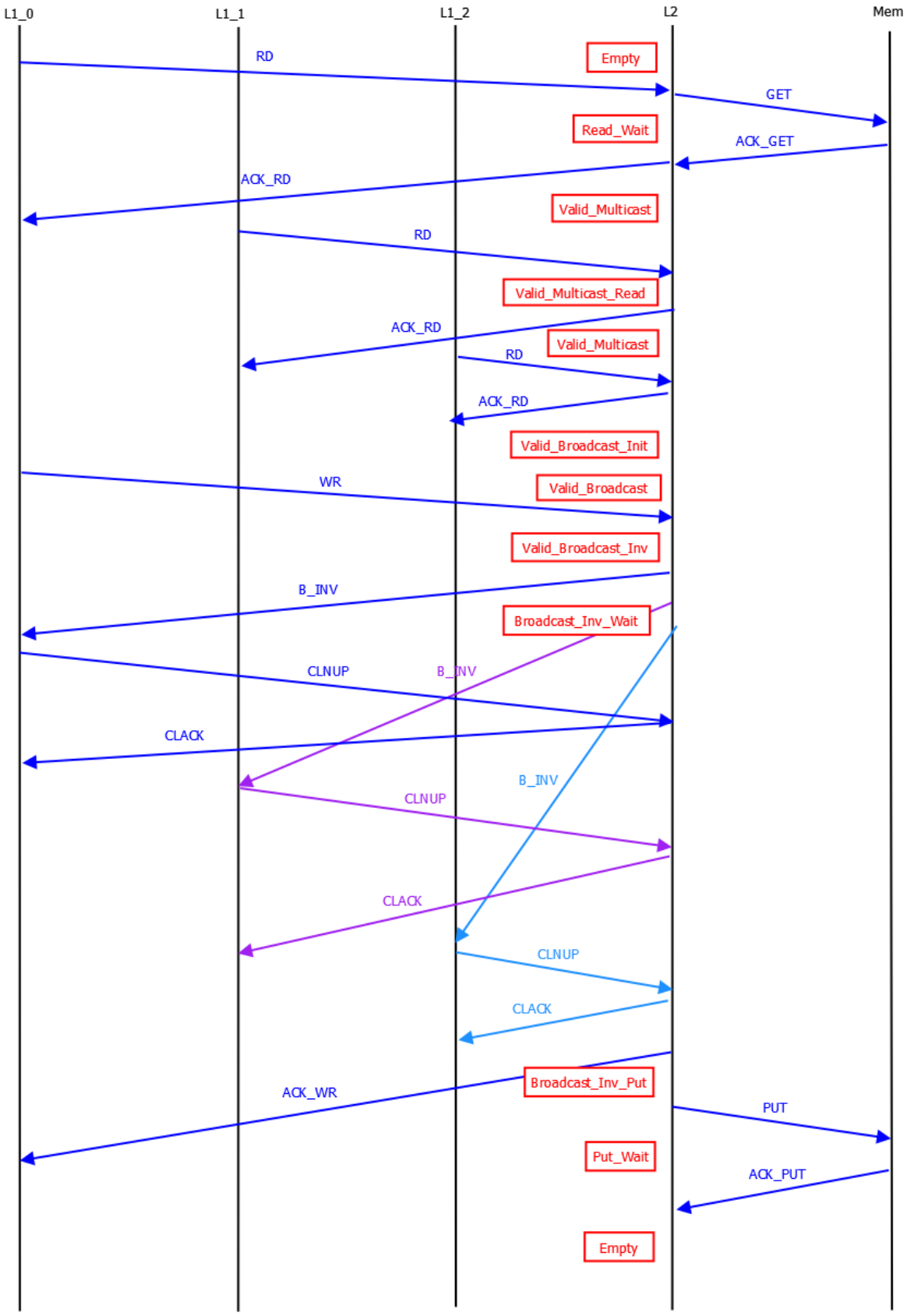

FIGURE A.3 – testeur_1 pour Cache L2

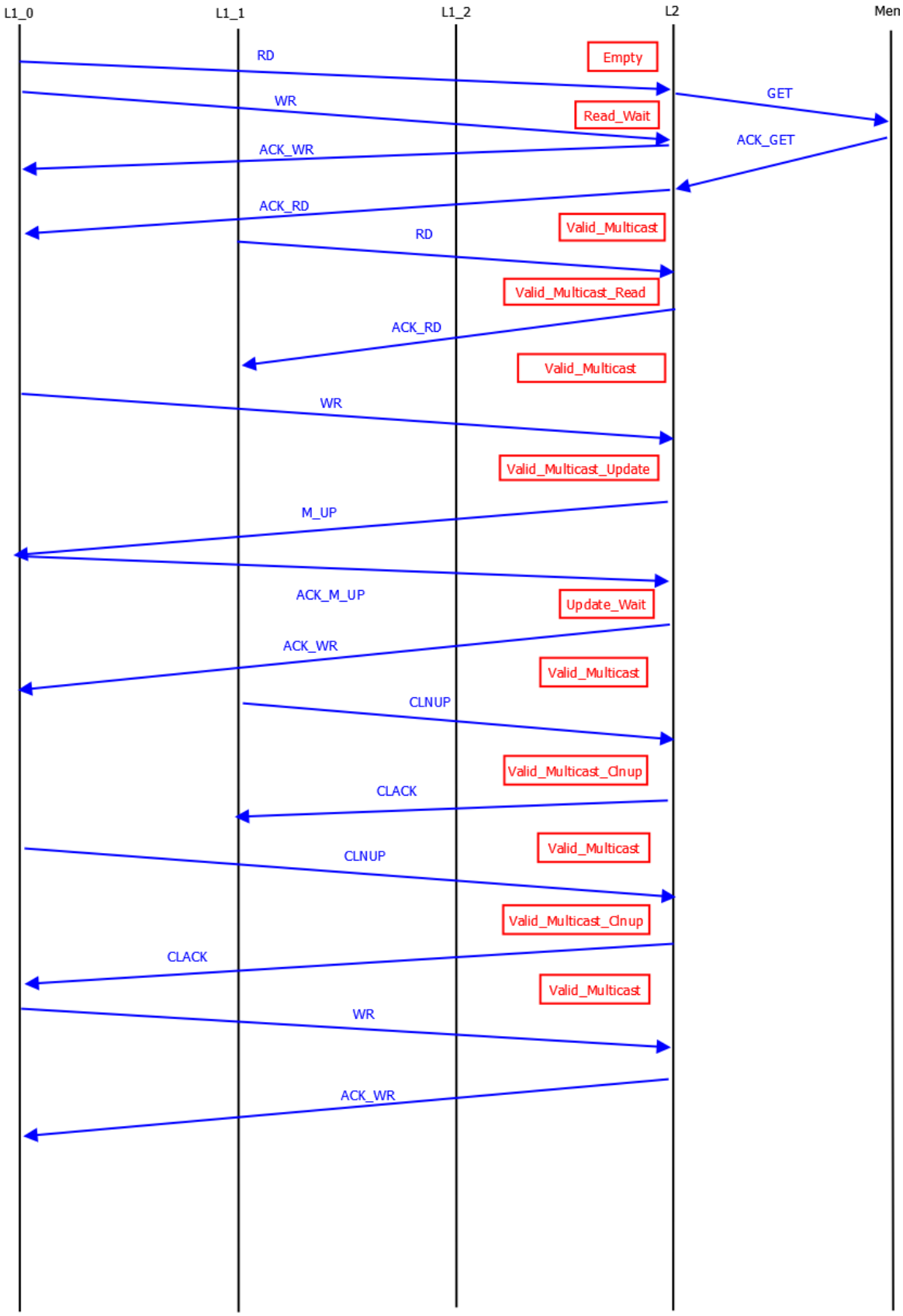

FIGURE A.4 – testeur_2 pour Cache L2

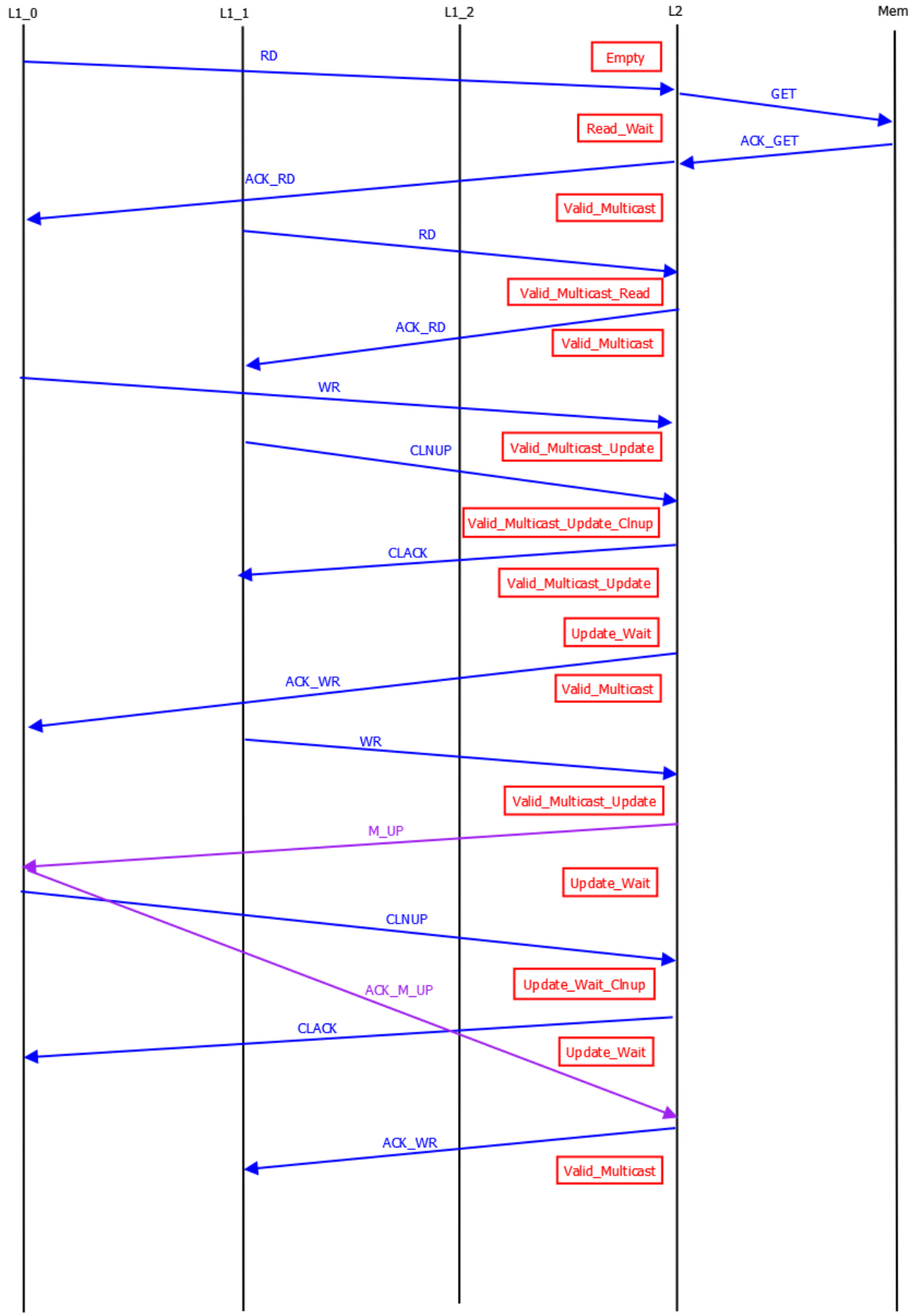

FIGURE A.5 – testeur_3 pour Cache L2

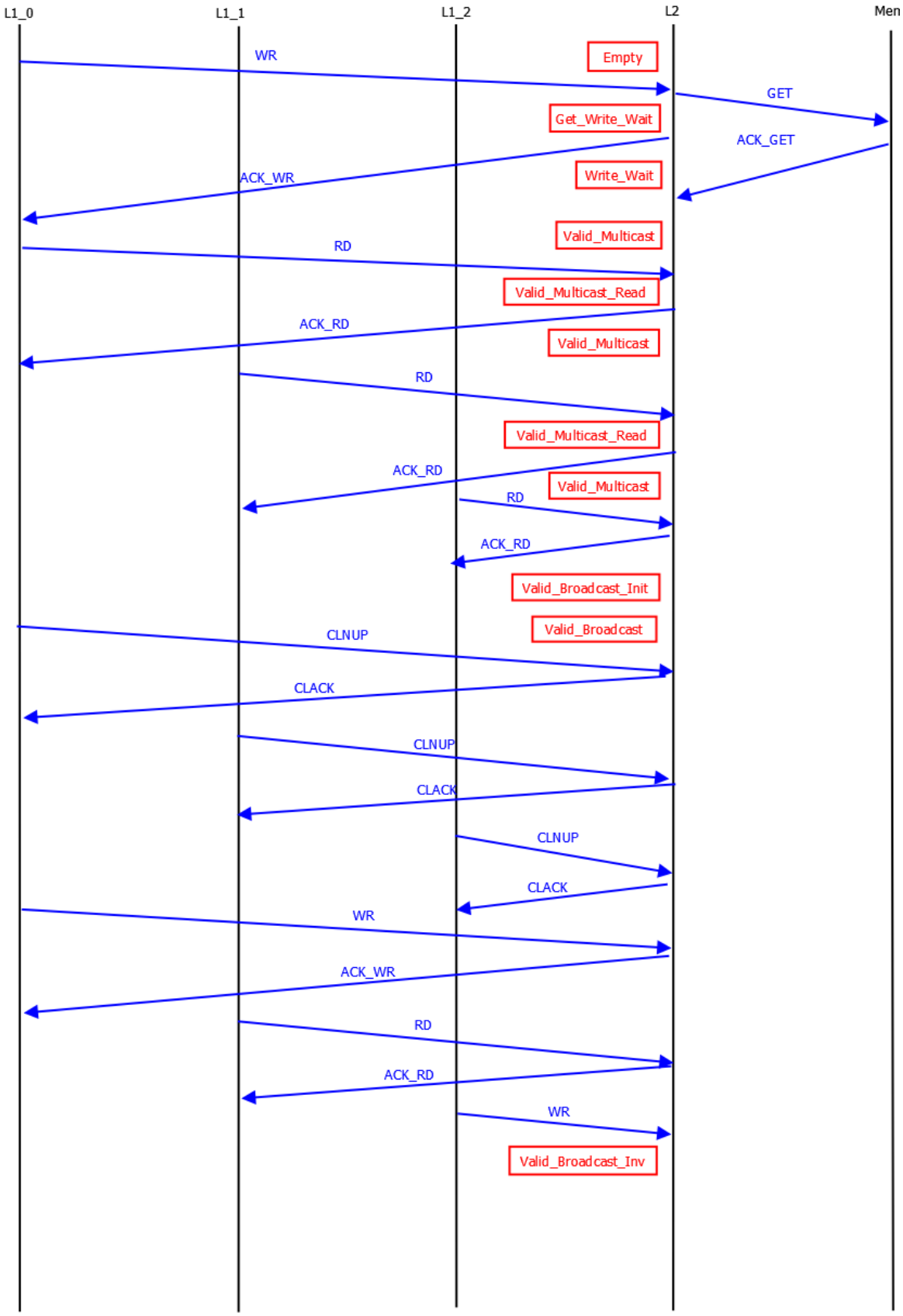

FIGURE A.6 – testeur_4 pour Cache L2

Bibliographie

[1] *TSAR : Tera-Scale Multiprocessor ARchitecture*. Available : https ://www-soc.lip6.fr/trac/tsar, 2009.

[2] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai, and Pavel Šimeček. Divine–a tool for distributed verification. In *Computer Aided Verification*, pages 278–281. Springer, 2006.

[3] Pierre Guironnet de Massas and Frédéric Pétrot. Comparison of memory write policies for noc based multicore cache coherent systems. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 997–1002. IEEE, 2008.

[4] Zahia Gharbi. *Verification compositionnelle du protocole de cohÃ©rence de cache de la machine multiprocesseur TSAR*. LIP6, 2013.

[5] Yann Thierry-Mieg. Symbolic model-checking using its-tools. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 231–237. Springer, 2015.