Modélisation en Lotos d'une mémoire virtuelle répartie à cohérence causale

V. Gal T. Cornilleau E. Gressier-Soudan

Laboratoire Cedric-CNAM
292, Rue Saint-Martin, F75141 Paris Cedex 03
E-mail: {viviane, vercor, gressier}@cnam.fr

1 Introduction

Le concept de mémoire partagée répartie a été développé au sein de projets de recherche comme Ivy [LH89], Munin [CBZp91], Mirage [FP89], et industriels comme Treadmarks [KCZ92], Chorus [OAA92]... De nombreux modèles de cohérence sont apparus depuis celui proposé par Censier et Feautrier [CF78]. En effet, aujourd'hui, l'ensemble des cohérences mémoire peut être divisé en deux classes [Mos93, RM93] : les cohérences uniformes et les cohérences hybrides.

Parmi ces modèles, nous nous intéressons aux modèles uniformes. Nous avons retenu la cohérence causale car elle permet un meilleur parallélisme tout en respectant les relations de causalité des accès à la mémoire. Nous avons travaillé à partir de la définition donnée dans [ABHN91] et implantée dans [JA93]. Nous souhaitons mettre en oeuvre une mémoire virtuelle répartie à cohérence causale au sein du système Chorus. Nous avons donc voulu maîtriser les concepts liés à cette cohérence. A cette fin, nous avons procédé à une spécification en Lotos [Gal95].

Nous présentons ci-après les concepts nécessaires à la mise en oeuvre d'une mémoire répartie, l'algorithme sélectionné pour gérer une mémoire à cohérence causale, la modélisation de celui-ci en Lotos et les résultats de ces travaux.

2 Concepts de base

2.1 Définition de la cohérence causale

La définition de la cohérence causale transpose le concept de causalité potentielle des messages [Lam78] aux accès en lecture et en écriture sur des variables. Cependant, la causalité mémoire diffère de la causalité des messages sur deux points. D'une part, certaines écritures produisent un résultat qui peut n'être jamais lu alors qu'un message émis est toujours reçu, si on considère qu'il n'y a pas de panne. D'autre part, un processus peut écrire successivement deux données et un autre les lire dans un ordre différent, ce qui n'est pas possible vis-à-vis de la causalité définie pour la communication par messages.

Le lien causal entre deux accès est caractérisé par la relation de causalité correspondant à la fermeture transitive des deux règles suivantes, en supposant que $\mathcal{E}_x$ est une écriture et $\mathcal{L}_x$ une lecture sur la même variable x [AHJ91, RMN92] :

1. Si $\mathcal{E}_x$ et $\mathcal{L}_x$ sont deux opérations successives effectuées par le même processeur alors $\mathcal{E}_x$ précède causalement $\mathcal{L}_x$. Cette règle représente l'ordre programme.

2. Si $\mathcal{L}_x$ lit le résultat de $\mathcal{E}_x$, sans qu'aucune autre opération d'écriture ne se soit produite sur la variable x entre $\mathcal{E}_x$ et $\mathcal{L}_x$, alors $\mathcal{E}_x$ précède $\mathcal{L}_x$.

Toutes les opérations d'écritures sont identifiées de façon unique et donc toute opération de lecture peut être rattachée à l'opération d'écriture dont elle dépend. Une mémoire est causalement cohérente si tous les accès des processeurs respectent l'ordre causal des opérations défini plus haut. On remarque que la règle 2 transpose l'émission d'un message en une opération d'écriture et sa réception en une opération de lecture.

2.2 Solutions actuelles

Dans la plupart des articles traitant de la mémoire causale, les auteurs cherchent à obtenir une plus grande concurrence d'accès et de meilleures performances en diminuant le nombre de messages sur le réseau.

ont retenu notre attention, sont :

- ceux de l'équipe du Gatech [AHJ91, JA93]. La technique utilisée pour capturer les relations causales repose sur les horloges vectorielles. Chaque processeur a une horloge locale qui évolue grâce aux communications inter-processeurs et aux invalidations. En reprenant l'approche de la cohérence relâchée [GLL+90], les auteurs ont étendu leur mémoire causale en utilisant des opérations de synchronisation. Ils ont de plus rajouté une hypothèse importante à savoir que les programmes sont sans concurrence d'accès aux données [JA94].

- celui de Raynal [RMN92]. L'article propose un schéma général de protocole capturant des relations causales entre objets pour implanter une mémoire causale à l'aide d'une technique similaire aux horloges vectorielles. Les auteurs mettent en avant la propriété de vivacité. Le protocole n'a pas été implanté. La suite de leurs travaux s'oriente vers la définition d'une cohérence séquentielle construite à partir d'une cohérence causale [RS95].

- celui de Boyer [Boy92]. L'article présente l'implantation d'une mémoire causale au-dessus de Mach avec l'environnement Guide. Etant donné le contexte particulier dans lequel évolue la gestion de la mémoire, les relations de causalité sont matérialisées au travers de liens de dépendances entre objets.

Une étude plus approfondie de ces algorithmes se trouve dans [CGO95].

3 L'algorithme modélisé

3.1 Choix

Comme nous souhaitons mettre en oeuvre une mémoire virtuelle répartie à cohérence causale au dessus d'un réseau de machines, les algorithmes qui nous semblent les plus adaptés sont ceux présentés dans [JA93] et [JA94]. Les relations de causalité dans les articles de l'équipe du Gatech sont mises en évidence grâce aux horloges vectorielles définies par Fidge et Mattern [Fid91, Mat88]. De ce fait, la mémoire répartie causale est construite sur des fondements connus. Bien que ces algorithmes ont été implantés, il nous est apparu indispensable d'en valider le fonctionnement avant toute implantation par une phase de spécification formelle et de validation. Nous avons modélisé à l'aide de la technique de spécification formelle LOTOS [ISO89] notre algorithme construit à partir de ceux de [JA93] et [JA94]. Il respecte la cohérence causale, limite les invalidations et gère certaines données utiles pour diminuer les transactions sur le réseau. Notre travail ne se place pas dans le cas de programmes sans concurrence d'accès aux données.

L'algorithme reprend la terminologie définie dans [LH89]. Chaque site i possède localement la copie des pages partagées dans un cache Ci. Pour chaque page x, le site i connaît :

- son droit d'accès sur la page (lecture ou écriture),
- l'identité du site gestionnaire connaissant toujours le site possédant la copie la plus à jour,
- l'estampille vectorielle ST associée à la page, indiquant la date de l'écriture correspondant à cette version.

De plus, chaque site i gère localement une horloge vectorielle HVi, chaque coordonnée de ce vecteur correspondant à un site. Cette horloge permet de dater les écritures. Elle permet aussi de comparer l'historique causal du site avec celui des autres sites via les estampilles des pages ramenées lors de la résolution des défauts. Elle sert à mettre à jour le cache Ci par invalidation.

Pour le traitement d'un défaut de page, trois types de sites sont mis en jeu :

- le site fautif exécutant la partie "défaut de page",
- le site gestionnaire, responsable de la page incriminée, exécutant la partie "serveur de défaut de page",
- le site propriétaire courant de la page exécutant principalement le traitement lié aux redirections.

Pour chaque type de défaut, nous donnons ci-dessous le traitement effectué par chacun de ces sites.

Traitement lié aux écritures

Défaut de page : accès en écriture à la page x sur le processeur Pi

```
 ei(x)v::
début
si(x.propriétaire = i) alors
  x.accès := écriture
  HVi := incrémenter[1](HVi)
  x.ST := HVi
sinon
  envoyer[ECRIRE, x] à x.gestionnaire
  recevoir[E_REPONSE, x, ST']
  x.accès := écriture
  x.propriétaire := i
  HVi := mettre_à_jour[2](HVi, ST')
  HVi := incrémenter(HVi)
  x.ST := HVi
  invalider(ST')
finsi
fin
```

1. Cette opération incrémente la ième coordonnée de l'horloge.
2. La nouvelle horloge est définie en choisissant pour chaque coordonnée la plus grande des coordonnées de même rang des deux horloges passées en paramètre.

processeur Pj, pour le processeur Pi, sur la page x

```
 [ECRIRE, x] ::
début
recevoir[ECRIRE, x] de Pj
si(x.propriétaire = i) alors
  x.accès := lecture
  envoyer[E_REPONSE, x, x.ST] à Pj
sinon
  envoyer[REDIRIGER, x, j, ECRIRE]
                à x.propriétaire
finsi
x.propriétaire = j
fin
```

Lors d'un défaut de page en écriture, si le site demandeur est propriétaire, plutôt que d'envoyer la requête au gestionnaire qui nous la retournerait, un test de propriété est fait sur le site. Cette action n'apparaît pas clairement dans [JA93]. Elle est stipulée dans [JA94] mais n'est pas décrite pas dans le corps de l'algorithme. Nous avons choisi de le faire figurer dans la requête d'un défaut de page. Comme l'opération demandée est une écriture, même si l'action reste locale, l'accès à la page est mis à jour, l'horloge est incrémentée de un. Ainsi la page se voit attribuer une nouvelle estampille.

Si le propriétaire de la page demandée n'est pas le site demandeur alors la requête est adressée au gestionnaire de la page. Le gestionnaire vérifie s'il est propriétaire. Le gestionnaire est par défaut propriétaire de toutes les pages non présentes dans les caches. Si c'est le cas, il met à jour son type d'accès et l'identificateur du nouveau propriétaire puis envoie la page au demandeur. Sinon, il vérifie le nom du propriétaire, redirige la requête vers le propriétaire et met à jour le champ propriétaire en y plaçant la nouvelle identification du propriétaire.

Lorsque le demandeur reçoit la page du propriétaire, il modifie le type d'accès et l'information indiquant qu'il est propriétaire. Il compare les estampilles, incrémente l'horloge et intègre la nouvelle valeur de l'estampille. On constate toujours que l'horloge d'un site évolue à chaque écriture. A la suite de cela, une série d'invalidations est lancée, si les conditions sont respectées. Les invalidations permettent de supprimer toutes les pages potentiellement obsolètes vis-à-vis des liens de causalité. Nous rappelons qu'à la différence des algorithmes de Kai Li, l'invalidation des pages est locale et n'entraîne aucun message.

Traitement lié aux lectures

Hormis les changements de droits différents et le fait que l'horloge ne soit pas incrémentée, le traitement d'un défaut en lecture est similaire à celui d'un défaut en écriture.

Défaut de page: accès en lecture à la page x sur le processeur Pi

```
 li(x)v ::
début
sinon
  envoyer[LIRE, x] à x.gestionnaire
  recevoir[L_REPONSE, x, ST']
  x.accès := lecture
  HVi := mettre_à_jour(HVi, ST')
  x.ST := ST'
  invalider(ST')
finsi
fin
```

On remarque que le processeur vérifie s'il est propriétaire de la page. En effet, si c'est le cas, cela signifie qu'il détient la dernière version valide de la page. Il suffira donc au processeur de changer le mode d'accès à la page dans sa table d'informations[3].

Serveur de défauts de page: requête de lecture d'un processeur Pj pour le processeur Pi sur la page x

```
 [LIRE, x] ::
début
recevoir[LIRE, x] de Pj
si(x.propriétaire = i) alors
  x.accès := lecture
  envoyer[L_REPONSE, x, x.ST] à Pj
sinon
  envoyer[REDIRIGER, x, j, LIRE]
                à x.propriétaire
finsi
fin
```

Traitement lié aux redirections

Cette partie concerne le traitement d'une requête d'accès en lecture ou en écriture sur la page x, initiée par le processeur Pj, et redirigée par le gestionnaire vers le processeur Pi, propriétaire de la page x.

```
 [REDIRIGER, x, j, mode] ::
début
recevoir[REDIRIGER, x, j, mode]
                  de x.gestionnaire
si(mode = ECRIRE) alors
  x.propriétaire := j
  x.accès := lecture
finsi
envoyer[REPONSE, x, x.ST] à Pj
fin
```

Traitement lié aux invalidations

```
 invalider(ST') ::
Qq soit y appartenant à Ci tel que
       y.ST < ST' et y.accès = lecture
alors
         y.accès := nul
```

3. Il faut noter qu'un tel cas ne peut arriver dans un système paginé comme Chorus.

obsolètes localement dès lors que nous effectuons une écriture ou une lecture. Il faut bien sûr respecter les conditions d'invalidations mentionnées ci-dessus. Nous avons également opté pour l'incrémentation de l'horloge locale dès lors que le processeur Pi demandeur d'une écriture est propriétaire de la page x et que le mode d'accès à cette page est en lecture. Le processeur Pi peut se trouver en mode lecture sur la page x si :

- localement, il en a fait la demande,
- un autre processeur lui a demandé la page x en lecture.

Nous pouvons constater que le fait de faire évoluer l'horloge dans ces conditions engendre l'obtention d'un nouveau numéro de version et ainsi reflète mieux l'évolution de la page.

L'algorithme proposé montre bien l'évolution des estampilles des pages et des horloges vectorielles de chaque site. En effet, à chaque défaut, un processeur demandeur reçoit la connaissance du temps logique du processeur propriétaire de la page accédée. La cohérence causale est bien respectée car chaque page se retrouve estampillée par la valeur de l'horloge du processeur qui effectue un défaut en écriture, chaque écriture sur la page propose bien une nouvelle estampille à chaque fois. De plus, chaque lecture est toujours reliée à l'écriture dont elle dépend. Enfin, les invalidations locales éliminent les pages obsolètes comme dans [JA93].

4 Langage et Outils

4.1 Lotos

Le choix du langage de spécification s'est porté sur Lotos. C'est un langage adapté à notre problème. En effet, Lotos initialement développé pour décrire des protocoles ISO de façon formelle permet la modélisation d'algorithmes répartis ou de systèmes distribués. Il offre la possibilité de définir aussi bien les structures de données que le comportement des processus. L'exécution symbolique et la simulation d'une spécification en langage Lotos permettent de valider le fonctionnement de l'environnement représenté.

Le langage Lotos présente deux points forts :

- il permet de donner une bonne idée du comportement du système surtout au niveau le plus externe.
- il possède des opérateurs et des types abstraits assez complets pour exprimer les fonctions du système.

Lotos offre une gestion de données souple et puissante même si elle est complexe. Il est composé de deux parties l'une pour décrire les données d'environnement, et l'autre pour le comportement associé :

- Les données sont décrites à l'aide du langage ActOne [EM85], langage de spécification de types de données assez simple et très rigoureuse. Par contre, si la spécification comporte une gestion de données évoluée, la spécification peut rapidement devenir difficile à lire.
- Le fonctionnement du système est exprimé au moyen d'une extension de CCS [Mil80]. Celui-ci est utilisé afin de décrire le comportement unitaire de processus et leurs échanges. Il a été sélectionné pour sa puissance d'expression, l'approche algébrique qui lui est sous-jacente et ses propriétés de vérification.

Lotos exploite des opérateurs explicites afin de combiner les processus entre eux. Il propose différents styles de spécifications complémentaires. Le style choisi dépend du comportement à représenter et donc des opérateurs utilisés dans la spécification. Pour les besoins du sujet, nous avons sélectionné :

- le style orienté contraintes puisque nous utilisons les opérateurs $|||$, $||$, $\gg$,
- le style monolithique puisque nous avons défini des processus comme des suites de choix,
- le style orienté données puisque le réseau est modélisé à partir de files.

4.2 Outils de vérification

Nous avons obtenu divers outils soit auprès d'Alcatel CIT à Lannion, soit auprès de Verimag à Grenoble.

SEDOS

SEDOS est une boîte à outils développée dans le cadre du projet ESPRIT LOTOSPHERE afin de fournir un support pour le développement de spécifications écrites en Lotos. On y trouve cinq outils importants permettant : l'analyse syntaxique d'une spécification en Lotos, la construction d'arbres de syntaxe abstraite, les analyses statique et sémantique, la simulation.

Les tests sont élaborés par l'utilisateur et peuvent être complexes. Ceci implique une simulation partielle. Un autre point non négligeable est que lorsqu'une modification est apportée dans la spécification, cette spécification doit être de nouveau complètement analysée. Ce sont, entre autres, deux raisons qui nous ont incités à utiliser une autre boîte à outils : Caesar/Aldebaran.

CAESAR/ALDEBARAN

Caesar/Aldebaran est une boîte à outils développée par le laboratoire de génie informatique de l'IMAG et l'INRIA. Elle est composée de trois outils permettant : la traduction des types abstraits de données en types concrets implantés en langage C, et la traduction d'une spécification en un réseau de Petri. La simulation exhaustive de tous les chemins possibles au travers du système engendre la production d'un graphe d'accessibilité (qui est l'équivalent d'une machine à états finis), la réduction et la comparaison des machines à états finis.

se reporter à [FM90], [GS90], [GM93] et [Gar94].

5 Modélisation

5.1 Principe de modélisation

Bien que le cheminement conduisant à l'élaboration de spécifications souffre encore de l'absence de méthode précise, nous avons opté pour une démarche progressive. Cette démarche se présente comme suit :

- détermination et élaboration des types abstraits de données nécessaires au fonctionnement de notre algorithme,
- décomposition du sujet en plusieurs niveaux d'abstraction,
- imbrication des données pas à pas dans la spécification.

Deux niveaux de spécification en Lotos sont étudiés : un niveau externe décrivant le système sans les détails de l'architecture du système et un niveau interne distinguant les différentes entités locales à un site pour la gestion de la mémoire. Leur spécification est alors plus précise et comporte plusieurs niveaux d'étude. Une définition de jeux de tests et une activité de vérification font également parties de cette phase. La simulation de la mémoire virtuelle des sites est réalisée à partir d'un fichier de tests contenant les différentes demandes d'accès aux pages.

5.2 Premier niveau : vue générale du système réparti global (SRG)

Ce système est décrit à l'aide de quatre boîtes (Cf. Figure 1). Trois boîtes symbolisent trois sites reliés par une quatrième boîte matérialisant le réseau. La description du point de vue externe du SRG nécessite quelques primitives. Ces primitives correspondent aux demandes suivantes :

- SRG_lect(page), demande d'une page en lecture,
- SRG_ecr(page), demande d'une page en écriture,
- SRG_page(page), réception de la page demandée ou demande d'un autre site.

Ces primitives Lotos correspondent aux différentes requêtes arrivant sur les portes Sn.

Le SRG est spécifié à l'aide de trois instanciations d'un processus Lotos "site". Chaque site agit de façon indépendante. Ces trois instanciations sont combinées à l'aide de l'opérateur parallèle |||. Elles sont synchronisées avec le processus "reseau" grâce à l'opérateur ||. Un processus "site" va simuler le comportement de ce qui se produit : d'une part, au niveau de la réception d'un défaut de page via la mémoire virtuelle locale et d'autre part, au niveau

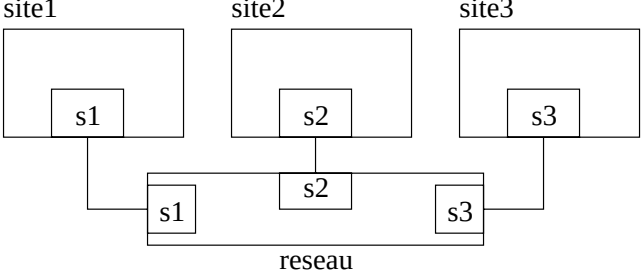

Fig. 1 - *Structuration du système réparti global*

du mappeur qu'il supporte. Trois portes de synchronisation *s1*, *s2*, et *s3* sont introduites : une pour chaque site. Les événements correspondants aux différentes demandes sont observés au travers de ces portes. Des paramètres sont associés au processus "reseau". Ces paramètres sont trois files de communication de type fifo. Ces files permettent l'ordonnancement asynchrone des messages arrivés sur chaque site.

Pour la simulation de trois sites, cela conduit en Lotos à la description suivante :

```
 specification SRG[s1, s2, s3] :  noexit
(* Déclaration des types abstraits de
données entrant dans la spécification du SRG *)
 Behaviour
(
site[s1]
|||
site[s2]
|||
site[s3]
)
||
reseau[s1, s2, s3](vide, vide, vide)
 where
   process site[sn] :  noexit :=
     def_page[sn]
     []
     recept_page[sn]
   where
     _include(def_page)
     _include(recept_page)
   endproc (* site *)
  _include(reseau)
endspec (* SRG *)
```

Remarques :

1. _include ne fait pas partie du langage Lotos mais est autorisé et permet l'inclusion de fichiers qui contiennent respectivement la spécification des traitements pour les défauts de page, la réception de pages et le réseau.
2. *exit* ou *noexit* au niveau d'un processus permet de signifier respectivement un comportement avec terminaison ou un comportement récursif.

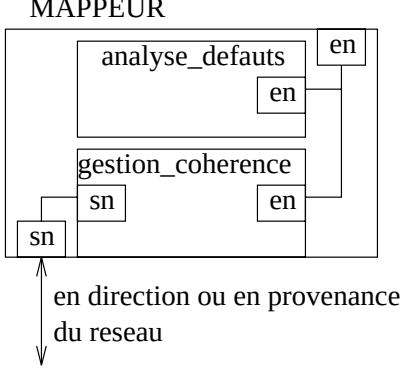

FIG. 2 - *Structure d'un site*

Les paramètres de valeur "vide" dans le processus "reseau" définissent les files de communication initialisées à vide au départ. La partie relative au site correspond à la vision extérieure qu'aurait un utilisateur. A ce niveau de la spécification, l'utilisateur n'a pas connaissance du comportement ou traitement interne du site. Or, sur les sites existent des traitements locaux comme la gestion de la cohérence causale à l'aide des horloges et des échanges de pages. Nous allons décrire ci-après ces mécanismes. Descendons d'un niveau et examinons ce qui se passe au niveau d'une boîte "site" appelé MAPPEUR qui contient la gestion de cohérence.

5.3 Deuxième niveau : le MAPPEUR

Le site traduisait, dans le niveau précédent, à la fois les défauts de page générés par la mémoire virtuelle et la résolution de ces défauts de page. Au deuxième niveau, l'entité MAPPEUR est introduite afin de modéliser le fonctionnement de cette résolution.

Le processus MAPPEUR adopte le même comportement que le processus "site". Les primitives décrivant les demandes de lecture ou d'écriture et les réceptions, définies pour le SRG restent valides pour la description du MAPPEUR puisque les mêmes requêtes vont lui être transmises.

Le comportement du MAPPEUR est lié à l'occurrence d'événements Lotos qui correspondent aux primitives qui lui parviennent. Ces primitives déclenchent des traitements puis l'élaboration de messages à émettre vers d'autres sites.

Le MAPPEUR est spécifié à l'aide des processus "analyse_defauts" et "gestion_coherence" (Cf. Figure 2). Ce dernier contient la description de la gestion de la cohérence causale. C'est en particulier au niveau de ce processus que sont réalisées toutes les opérations relatives aux évolutions de l'horloge et à la connaissance des pages.

Le processus MAPPEUR se synchronise avec le processus "gestion_coherence" sur la porte *sn* pour toutes les requêtes venant de sites distants. Les processus "analyse_defauts" et "gestion_coherence" se synchronisent sur la porte *en* pour la résolution des défauts locaux.

sait qu'une seule porte de synchronisation *sn*. Pour une meilleure compréhension et une meilleure lisibilité, il nous a semblé important de mettre en évidence les portes reliées à la mémoire virtuelle par rapport à celles liées au réseau. C'est la raison pour laquelle il apparaît maintenant *en* et *sn*.

De plus amples détails se trouvent dans [Gal95].

5.4 La vérification et les tests

La vérification de notre spécification a été réalisée par simulation à partir des outils Hippo et Caesar/Aldebaran. Un des intérêts de formaliser un système est de pouvoir vérifier le modèle élaboré. La possibilité d'analyse dynamique est un avantage majeur d'une description formelle, comparativement à la description informelle. A la différence des analyses statiques, l'analyse dynamique s'appuie, dans le cas présent, sur une exécution de la spécification.

Hippo et Caesar/Aldebaran nous ont permis dans un premier temps :

- de détecter les erreurs de syntaxes et de sémantique,

et, dans un deuxième temps :

- d'évaluer les données pas à pas,
- de déceler les interblocages et les dysfonctionnements,
- de vérifier la vivacité du modèle.

La phase de tests est importante par rapport au modèle car elle permet :

- de détecter les interblocages dans les communications, les dysfonctionnements au niveau de événements,
- une observation du comportement global de l'ensemble du système,
- de contrôler si les résultats correspondent bien à ceux attendus.

Un scénario est composé d'une succession d'événements. Un test peut être effectué de deux façons différentes : en mode interactif ou en mode fichier. Le mode interactif se fait au clavier et pas à pas. Le mode fichier se traduit par la combinaison d'un scénario, défini par un processus, et de la spécification munis de l'opérateur de parallélisme.

Spécification d'un test en mode fichier :

```
((
site[s1]
|||
site[s2]
|||
site[s3]
)
||
reseau[s1, s2, s3](vide, vide, vide) )
||
test[s1, s2, s3]
```

notre modèle ont permis :

- de contrôler l'évolution d'une page sur tous les sites,
- de vérifier que la contrainte "un écrivain et n lecteurs" était respectée,
- de voir que les écritures respectaient un ordre total,
- de contrôler les liens de causalité.

Les tests sont construits manuellement car il n'existe pas, pour le moment, d'outil générant des scénarios de tests de façon automatique. La difficulté est de trouver une série de scénarios couvrant la spécification.

6 Conclusion et perspectives

La modélisation nous a permis de détecter et de corriger des erreurs présentes dans la version originale de l'algorithme. La formalisation par une technique de spécification a mis en évidence la viabilité d'un projet sur une mémoire répartie à cohérence causale, et la validité du modèle.

La vérification de la spécification a montré que le modèle est vivant, sans interblocage ni famine, qu'il caractérise la causalité des accès et qu'il exhibe les invalidations.

Grâce à notre modélisation, nous avons pu observer des changements de comportement en faisant évoluer les horloges vectorielles de plusieurs façons. Les cohérences causales ainsi obtenues correspondent à différentes variantes d'une même sémantique. Celles-ci portent sur l'importance relative des lectures par rapport aux écritures et vice-versa, l'importance relative aux accès locaux par rapport aux accès distants, la prise en compte d'éventuelles interactions entre plusieurs accès sur des données différentes, et la prise en compte plus ou moins complète de l'histoire des différents accès. Suivant les choix, il est alors possible de générer un grand nombre d'algorithmes permettant d'implanter ces différentes cohérences causales.

Notre étude nous a conduit à une meilleure compréhension des problèmes de cohérence et à une première formalisation à partir de laquelle il sera possible de comparer différentes définitions de cohérence. En parallèle du précédent travail, nous avons défini une architecture logicielle [CGO95] permettant de mettre en oeuvre et d'utiliser différentes cohérences au-dessus du système Chorus/ClassiX [Cho94a]. Cette implantation est en cours d'intégration. A partir de cette implantation, il sera très facile de tester de nouveaux modèles de cohérences. La prochaine phase de notre travail examinera comment les travaux précédent peuvent s'intégrer dans le système COOL [Cho94b] qui offre un support de programmation par objets répartis.

Références

[ABHN91] M. Ahamad, J.E. Burns, P.W. Hutto, and G. Neiger. Causal memory. In *11th ICDCS*, pages 274–281, 1991.

menting and programming causal distributed shared memory. In *5th Int. Workshop on Dist. Alg.*, pages 9–30. Springer-Verlag, 1991.

[Boy92] F. Boyer. A causal distributed shared memory based on external pagers. Technical Report 34, Bull-IMAG, 1992.

[CBZp91] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implememtation and performance of munin. In *13th SOSP*, pages 152–164. ACM, October 1991.

[CF78] M. Censier and P. Feautrier. A new solution to the coherence problems in multicache systems. *IEEE Transactions on Computers*, December 1978.

[CGO95] T. Cornilleau, E. Gressier, and M-I. Ortega. A multiconsistency memory protocol test environment on chorus. In *ERSADS*, pages 281–286, L'Alpe d'Huez, April 1995. IMAG.

[Cho94a] Chorus Systèmes. *Chorus ClassiX i386at r1 Product Description*, 1994. CS/TR-94.44.3.

[Cho94b] Chorus Systèmes. *Chorus COOL v2r1 Product Description*, 1994. CS/TR-94.21.

[EM85] H. Ehrig and B. Mahr. Fundamental of algebraic specifications : Equations and initial semantics. In *EATCS Monographs on Theoretical Computers Sciences*, 1985.

[Fid91] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24:28–33, August 1991.

[FM90] J.C. Fernandez and L. Mounier. *Aldebaran : Users's manual.* IMAG-LGI, June 1990.

[FP89] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *12th SOSP*, pages 211–223. ACM, December 1989.

[Gal95] V. Gal. *Spécification à l'aide du langage LOTOS d'un algorithme de gestion d'une mémoire à cohérence causale.* Mémoire d'ingénieur, Cedric-CNAM, March 1995.

[Gar94] H. Garavel. *The Open/caesar reference manual.* Verimag, 1994.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Computer Architecture News*, 18(2):15–26, June 1990.

[GM93] H. Garavel and L. Mounier. A verification toolbox for lotos. In *13th Symp. on protocol specification, testing and verification*, Liège, May 1993. IFIP.

verification of Lotos specifications. Verimag, 1990.

[ISO89] ISO. *Lotos: A formal description technique based on the temporal ordering of observational behaviour*, 1989. ISO 8807.

[JA93] R. John and M. Ahamad. Causal memory: Implementation, programming support and experiences. Technical Report GIT-CC-10, Gatech, 1993.

[JA94] R. John and M. Ahamad. Evaluation of causal distributed shared memory for data-race-free programs. Technical Report GIT-CC-34, Gatech, 1994.

[KCZ92] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *SIGARCH*, pages 13–21. ACM, May 1992.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In North-Holland, editor, *Int. Conf. on Parallel and Dist. Computing*, pages 215–226, 1988.

[Mil80] R. Milner. A calculus of communicating systems. In Springer Verlag, editor, *LNCS 92*, 1980.

[Mos93] D. Mosberger. Memory consistency models. *SIGOPS Operating Systems Review*, 27(1), January 1993.

[OAA92] M-I. Ortega, F. Armand, and V. Abrossimov. A distributed consistency server for the chorus system. In *SEDMS III*, pages 129–148. USENIX, March 1992.

[RM93] M. Raynal and M. Mizuno. How to find his way in the jungle of consistency criteria for distributed objects memories. Technical Report 730, IRISA, May 1993.

[RMN92] M. Raynal, M. Mizuno, and M. Neilsen. Causality oriented shared memory for distributed systems. Technical Report 656, IRISA, 1992.

[RS95] M. Raynal and C. Schiper. From causal consistency to sequential consistency in shared memory systems. IRISA. Communication personnelle, 1995.