INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Formalization and Verification of Coherence Protocols with the Gamma Framework

David Mentré, Daniel Le Métayer, Thierry Priol

N°3953

June 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3953--FR+ENG

Formalization and Verification of Coherence Protocols with the Gamma Framework

David Mentré, Daniel Le Métayer, Thierry Priol

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n ° 3953 — June 2000 — 18 pages

Abstract: This paper presents an approach to formalize coherence protocols for shared virtual memories as multiset rewriting systems. The global state of the protocol is represented as a multiset and rewriting rules are used to describe state changes. Invariants are expressed as properties on the cardinality of subsets which characterize specific relations. We present an automatic algorithm to check that a property is an invariant of a protocol. Both the formalization and the verification steps are illustrated on the Li and Hudak single-writer/multiple-readers coherence protocol.

Key-words: Gamma, Verification, Coherence protocol, Multiset rewriting

(Résumé : tsvp)

This paper should appear in PDSE'2000, June 2000, Limerick, Ireland

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Formalisation et vérification de protocoles de cohérence avec le formalisme Gamma

Résumé : Ce papier présente une approche pour la formalisation de protocoles de cohérence pour les mémoires virtuelles partagées sous forme de systèmes de réécriture. L'état global d'un protocole est représenté par un multi-ensemble et les règles de réécriture sont utilisées pour décrire les changements d'état. Les invariants sont exprimés sous forme de propriété sur la cardinalité de sous-ensembles qui caractérisent des relations spécifiques. Nous présentons un algorithme de vérification automatique pour vérifier qu'une propriété est un invariant du protocole. Les deux étapes de formalisation et de vérification sont illustrées sur le protocole de cohérence de Li et Hudak, à écrivain unique et lecteurs multiples.

Mots-clé : Gamma, Vérification, Protocole de cohérence, Réécriture de multi-ensemble

1 Motivation and approach

Distributed systems, made of autonomous machines linked by a network, are widely used, ranging from high-performance cluster computing to fault-tolerant systems. However designing and programming such systems is a difficult task, involving management of explicit parallelism and duplicated information. Coherency is one of the most difficult problems arising in distributed systems. It appears for example in distributed file systems, virtual shared memory or web caches. As information is spread all over the system, one needs protocols to access and update this information coherently, while meeting specific constraints such as fault-tolerance, performances, latency, security, etc. Building such protocols is a complex matter. To master this complex process, it is crucial to be able to separate design and implementation issues. The design framework should include facilities to formally check properties of the protocol. In a second stage, it should be possible to specialize the abstract version of the protocol and to get an efficient implementation on the target platform.

Coherence protocols are usually presented using pseudo-code or natural language (in the area of shared virtual memory, [4, 5, 12, 21] are representative examples). Checking such protocols is not an easy task. First, the specification and a model of the protocol should be clearly stated; then the model should be checked against its specification. A variety of verification techniques are available [10, 18, 19, 22] but they either face the state space explosion problem (in the case of state enumeration and symbolic model checking techniques) or need human intervention and thus are difficult to use (in the case of theorem proving).

To address those issues, we advocate in this paper a domain specific language to describe coherence protocols. This approach relies on multiset rewriting to formalize protocol entities and control. Moreover, we propose an algorithm to check invariant properties over such a formal description. The algorithm is based on a symbolic representation and avoids the exploration of states that are not relevant for a given property. The result of the algorithm is a condition that has to be satisfied by the initial state of the system for the property to be an invariant. The verification is automatic and the only actions required from the user are the specification of the putative invariant and the property specifying the initial state of the system.

This paper is focused on verifying shared virtual memories coherence protocols which we present in section 2, in conjunction with an example protocol used throughout this paper. In section 3, we describe our formalism to express protocols and properties and to illustrate it with the example. In section 4, we present the verification algorithm and sketch its application on the example. Related work and further work are discussed in section 5 and section 6 respectively.

2 Shared Virtual Memories

Shared Virtual Memories (SVM) have been proposed by Li and Hudak [14] as a technique to facilitate the design of distributed applications. This concept offers the illusion of a global address space over distributed address spaces. In such a system, the shared address space is

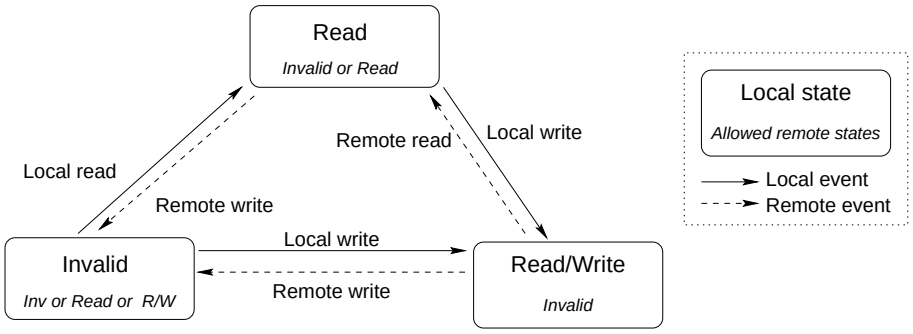

Figure 1: Page state graph of Li and Hudak coherence protocol

divided in elementary units called *cache lines* (in hardware distributed shared memory) or *pages* (in software SVM). Those units are replicated in a local part of the distributed system nodes memory called a *cache*. This distribution allows simultaneous accesses to data and thus several copies of the same logical piece of information can be concurrently modified. If a node of the distributed system modifies its local copy of a piece of data, other copies of the same piece of information should be updated accordingly. This should be made in a coherent manner, such as at any time, any view of the system is perceived as coherent (according to a certain memory model, i.e. a contract between the programmer and the SVM). This coherent modification of distributed data is achieved by a cache coherence protocol. As different applications require to manage data in different ways, several protocols have been proposed, like the lazy release consistency [12] or the scope consistency [11].

To implement a protocol, most software SVM use the paging mechanism of common processors to detect access to data. On each node, the address space elementary unit is a *page* (typically 4 to 8 Kbytes). Read and write access to a page can be individually set. If a SVM system must detect, for example, processor load instructions on a specific page, it only needs to disable read access to this page through the processor paging logic. Next time the processor will issue a load instruction for a data in this page, a read page fault will be triggered and caught by the SVM system. Each page is associated with one page-frame that owns data in physical memory.

In this paper, we use as an example the Li and Hudak single-writer/multiple-readers coherence protocol [14]. This protocol is both concise and realistic to illustrate the approach. The Li and Hudak protocol handles each memory page of the shared address space independently. A page is either written by a unique node or read by one or more nodes. Each reading node performs a local copy of the page. When a page in read state is written, all copies of the page must be deleted (through an *invalidation* phase). Figure 1 shows the state graph of a page. State change is triggered either by a local or a remote access. For each page, a manager stores information about the status of each page state. Access requests are sent to it and it triggers the necessary invalidations.

We now describe our formalism and illustrate it with the above Li and Hudak protocol.

3 Protocol formalization

Our protocol specification language is an elaboration of the Gamma formalism [2, 3]. Gamma is based on multiset rewriting (also called the "chemical reaction paradigm") which has already been used in various contexts to reason about distributed computations [7, 8, 13].

The Gamma formalism is based on the chemical reaction metaphor. The unique data structure in Gamma [2] is the multiset (a set than can contain several occurrences of the same element) which can be seen as a "chemical solution". A simple program is a set of rules *Reaction condition* $\rightarrow$ *Action*. Execution proceeds, without an explicit order, by replacing elements in the multiset satisfying the reaction condition by the products of the action ("chemical reaction"). The result is obtained when a stable state is reached, that is to say when no more reactions can take place. Locality and lack of ordering allows a description of algorithms without unnecessary sequentiality. An example of a Gamma program is sum: $x, y \rightarrow x + y$. This program substitutes two elements x and y by their sum (action) as long as two elements are available in the multiset (reaction condition). Applied to multiset $\{1, 2, 2, 3\}$ it could produce the following execution sequence:

$$\{1,2,2,3\} \xrightarrow[\substack{1,3\rightarrow 4\\ 2,2\rightarrow 4}]{} \{4,4\} \xrightarrow[4,4\rightarrow 8]{} \{8\}$$

Using this formalism, we now present how protocols and then properties are described.

3.1 Expressing protocols as multiset rewriting systems

The multiset represents a global view of the state of the system. In this multiset, relations $\mathbf{R}\ x_1\ \ldots\ x_n$ describe protocol entities or events (for example, for an SVM protocol, access rights to pages, page-frames, read or write page faults, etc.). The logic of the protocol is described with a Gamma program made of rules (like $\mathbf{R_1}\ x\ ,\ \mathbf{R_2}\ y \rightarrow \mathbf{R_3}\ x\ y$). If the left hand side of a rule is available in the multiset ($\mathbf{R_1}\ x\ ,\ \mathbf{R_2}\ y$), then it consumed and the right hand side of the rule ($\mathbf{R_3}\ x\ y$) is added to the multiset. A rewriting step is supposed to be atomic. A protocol *Prt* is described by the following grammar:

$$\begin{array}{rcl} Prt & ::= & Rl \mid Rl, Prt \\ Rl & ::= & T_1\ \neg T_2 \rightarrow T_3 \\ T_i & ::= & \mathbf{R}\ x_1\ \ldots\ x_n \mid T_i, T_i \mid \emptyset \end{array}$$

A protocol is made of a set of rules *Rl*, each rule having three sets of terms T_1, T_2 and T_3. A set of terms T_i is made of zero or more relations $\mathbf{R}\ x_1\ \ldots\ x_n$. In a rule, T_1 is the set of relations that must be present in the multiset to be able to apply this rule. T_2 is the set of relations that must *not* be present in the multiset to be able to apply this rule. When the rule is applied, all relations of T_1 are removed from the multiset and relations in T_3 are added to it. A formal semantic of this syntax in given in [15].

R_1 : **ReadDetect** $p\ n_1$, **RMode** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$
→ **RMode** $p\ n_1$, **RMode** $p\ n_2$, **PageFrame** $pf_2\ p\ n_1$, **PageFrame** $pf_2\ p\ n_2$

R_2 : **ReadDetect** $p\ n_1$, **RWMode** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$
→ **RMode** $p\ n_1$, **RMode** $p\ n_2$, **PageFrame** $pf_2\ p\ n_1$, **PageFrame** $pf_2\ p\ n_2$

R_3 : **WriteDetect** $p\ n_1$, **RWMode** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$
→ **RWMode** $p\ n_1$, **PageFrame** $pf_2\ p\ n_1$

R_4 : **WriteDetect** $p\ n_1$, ¬**RMode** $p\ n_1$, **RMode** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$, **Ok** p
→ **InvalidationPhase** $pf_2\ p\ n_1$

R_5 : **InvalidationPhase** $pf_1\ p\ n_1$, **RMode** $p\ n_2$, **PageFrame** $pf_1\ p\ n_2$
→ **InvalidationPhase** $pf_1\ p\ n_1$

R_6 : **InvalidationPhase** $pf_1\ p\ n_1$, ¬**RMode** $p\ n_2$
→ **RWMode** $p\ n_1$, **PageFrame** $pf_1\ p\ n_1$, **Ok** p

R_7 : **WriteDetect** $p\ n_1$, **RMode** $p\ n_1$, **PageFrame** $pf_1\ n_1$, **Ok** p
→ **InvalidationPhase** $pf_1\ p\ n_1$

Figure 2: Structured Gamma version of single-writer/multiple-readers Li and Hudak protocol

Figure 2 presents the complete Gamma specification of the Li and Hudak protocol. The rules defining the protocol are numbered R_1 to R_7. The relations contained in the multiset characterize physical entities (page table, page content) and logical entities (protocol phase for example). **RWMode** $p\ n$ states that page p on node n is in read/write mode (page table rights). **RMode** $p\ n$ states a read right. **ReadDetect** $p\ n$ (respectively **WriteDetect** $p\ n$) states that a read (respectively write) exception has occurred for page p on node n. **PageFrame** $pf\ p\ n$ states that a page frame pf (a physical page) corresponding to virtual page p exists on node n. **InvalidationPhase** $pf\ p\ n$ states that page p is in invalidation phase in order for page frame pf to be accessible in read/write mode on node n. **Ok** p states that page p is not involved in an invalidation phase.

Each rule expresses a state change with the associated precondition. For example, the occurrence of **RWMode** $p\ n$ in the left hand side but not in the right hand side of a rule (as in R_2) implies that the read/write right is cancelled for page p on node n. Rules R_1 and R_2 correspond to the case where node n_1 requests read access for a page and this page is currently used in read (R_1) or write (R_2) mode by another node n_2. Rules R_3, R_4 and R_7 state the case where node n_1 writes on a page and this page is (a) either accessed in write (R_3) or read (R_4) mode by another node n_2 or (b) accessed by itself (node n_1) in read mode (R_7). Finally, rules R_5 and R_6 state respectively the invalidation loop of page copies and the grant of read/write access when all invalidations are made. One should notice that

$$
\begin{array}{lll}
P_1 & : & \overline{\mathbf{RWMode}}\ p\ * \leq 1 \\
P_2 & : & \overline{\mathbf{RMode}}\ p\ * > 0 \Rightarrow \overline{\mathbf{RWMode}}\ p\ * \leq 0 \\
P_3 & : & \overline{\mathbf{RWMode}}\ p\ * > 0 \Rightarrow \overline{\mathbf{RMode}}\ p\ * \leq 0 \\
P_4 & : & \overline{\mathbf{WriteDetect}}\ p\ n > 0 \Rightarrow \overline{\mathbf{ReadDetect}}\ p\ n \leq 0 \\
P_5 & : & \overline{\mathbf{ReadDetect}}\ p\ n > 0 \Rightarrow \overline{\mathbf{WriteDetect}}\ p\ n \leq 0 \\
P_6 & : & \overline{\mathbf{InvalidationPhase}}\ *\ p\ n > 0 \Rightarrow \overline{\mathbf{RWMode}}\ p\ n \leq 0 \\
P_7 & : & \overline{\mathbf{RMode}}\ p\ n > 0 \Rightarrow (\overline{\mathbf{PageFrame}}\ pf\ p\ n > 0 \wedge \overline{\mathbf{PageFrame}}\ pf\ p\ n \leq 1) \\
P_8 & : & \overline{\mathbf{RWMode}}\ p\ n > 0 \Rightarrow (\overline{\mathbf{PageFrame}}\ pf\ p\ n > 0 \wedge \overline{\mathbf{PageFrame}}\ *\ p\ * \leq 1)
\end{array}
$$

Figure 3: Properties for the Li and Hudak protocol

rules R_4 and R_6 contain a negation. This extension to the pure Gamma formalism make the protocol description easier and does not have a significant impact on verification.

3.2 Properties

An invariant is made of properties P which are defined according to the following grammar:

$$
\begin{array}{lll}
P & ::= & \overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K \mid \overline{\mathbf{R}}\ y_1\ \ldots\ y_n > K \\
 & & \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \mid \neg P \mid \mathsf{True} \mid \mathsf{False} \\
K & ::= & 0 \mid 1
\end{array}
$$

Notation $\overline{\mathbf{R}}\ y_1\ \ldots\ y_n$ expresses the cardinality of a relation named R in the multiset according to the pattern $y_1\ \ldots\ y_n$ (where a y_i can be a variable name or the match-all symbol $*$). For example, in the multiset $\{\mathbf{R}\ a\ b,\ \mathbf{R}\ a\ c\}$ we have $\overline{\mathbf{R}}\ i\ * = 2$, $\overline{\mathbf{R}}\ i\ b = 1$, $\overline{\mathbf{R}}\ *\ * = 2$ and $\overline{\mathbf{R}}\ i\ i = 0$. Properties are implicitly quantified (universally) on their free variables. One should also notice that the implication $\Rightarrow$ can be easily added to the syntax of this language of properties using the usual equivalence $A \Rightarrow B \iff \neg A \vee B$. The formal semantics of the language of properties is presented in [15].

Figure 3 presents a collection of putative invariants for the Li and Hudak protocol. The properties are numbered P_1 to P_8. As mentioned above, an overlined relation like $\overline{\mathbf{RWMode}}\ p\ *$ expresses the number of occurrences of tuples satisfying the relation with matching arguments ($*$ matching any value). Property $\overline{\mathbf{RWMode}}\ p\ * \leq 1$ states that for a page p, there is at most one node n such that $\mathbf{RWMode}\ p\ n$ exists. Within the protocol, it is equivalent to say that at any time, at most one node (node n) can write on page p. Property P_1 states that at most one node can have read/write access on a page. P_2 states that if a node has a read access on a page, no other node can have write access to it. P_3 states the dual case. P_4 and P_5 check that read and write page faults cannot be simultaneously detected on a page. P_6 states that in invalidation phase, the node having triggered invalidation should not have read/write access to this page. P_7 states that if there is a read access to the page then this node should have one and only one physical copy of

the page. Finally P_8 checks that if a node has a write access to a page, it should have a physical copy of the page and no more than one copy of this page should exist in the system.

This formalization of the Li and Hudak protocol does not mention how those pieces of information are stored in a distributed system (i.e. how relations **RWMode** or **InvalidationPhase** are implemented in the system). This declarative formalization is sufficient to verify the *essence* of a protocol. Such a protocol can then be implemented in various manners, depending on technology and other constraints. For example, all the algorithms described in [14] can be seen as refinements of the abstract version presented here.

4 Automatic verification of properties

The framework presented in the previous section allows us to formalize both the protocol and the properties that this protocol is expected to satisfy. What is needed now is a way to check that the property is indeed an invariant of the protocol. This verification should be automatic and reasonably efficient to be applicable on real size protocols. We present now our verification algorithm along with needed auxiliary functions and show its result for the Li and Hudak protocol.

4.1 Verification algorithm

The proposed algorithm is based on the simple observation that a rule in a Gamma program removes and introduces a fixed, finite number of relations. From a property to be satisfied *after* the application of a rule, it is possible to derive a property that must be satisfied *before* this rule is applied, that is to say a *precondition*. For example, consider property P_2:

$$\overline{\mathbf{RMode}}\ p\ * \leq 0 \vee \overline{\mathbf{RWMode}}\ p\ * \leq 0$$

and rule R_2:

$$\begin{array}{l} \mathbf{ReadDetect}\ p\ n_1\ ,\ \mathbf{RWMode}\ p\ n_2\ , \\ \mathbf{PageFrame}\ pf_2\ p\ n_2 \\ \qquad\qquad \rightarrow \\ \mathbf{RMode}\ p\ n_1\ ,\ \mathbf{RMode}\ p\ n_2\ , \\ \mathbf{PageFrame}\ pf_2\ p\ n_1\ ,\ \mathbf{PageFrame}\ pf_2\ p\ n_2 \end{array}$$

The overlapping between P_2 and R_2 is manifested by the projection (from rule R_2 variables to property P_2 variables) μ:

$$\mu(p) = p \quad \mu(pf_2) = pf \quad \mu(n_1) = * \quad \mu(n_2) = *$$

In rule R_2, one relation **ReadDetect** and one relation **RWMode** are suppressed in the multiset. At the same time, two relations **RMode** and one relation **PageFrame** are also

added to it. Considering cardinality variations, it gives:

$$\begin{array}{lll} \Delta\overline{\mathbf{ReadDetect}}\ p* & = & -1 \\ \Delta\overline{\mathbf{RMode}}\ p* & = & +2 \\ \Delta\overline{\mathbf{RWMode}}\ p* & = & -1 \\ \Delta\overline{\mathbf{PageFrame}}\ pf\ p* & = & +1 \end{array}$$

Therefore, if P_2 must be verified once R_2 is applied, one need to ensure before that:

$$\overline{\mathbf{RMode}}\ p* \leq \underbrace{-2}_{-\Delta\overline{\mathbf{RMode}}\ p*}$$

$$\bigvee \quad \overline{\mathbf{RWMode}}\ p* \leq \underbrace{1}_{-\Delta\overline{\mathbf{RWMode}}\ p*}$$

The property $\overline{\mathbf{RMode}}\ p* \leq -2$ is equivalent to false since cardinalities cannot be negative. So the above property can be simplified into $\overline{\mathbf{RWMode}}\ p* \leq 1$. Moreover this condition must be satisfied when R_2 is applicable, that is to say when $\overline{\mathbf{ReadDetect}}\ p* > 0 \wedge \overline{\mathbf{RWMode}}\ p* > 0 \wedge \overline{\mathbf{PageFrame}}\ pf\ p* > 0$. Thus we have obtained the *weakest precondition* of P_2 for rule R_2 which is:

$$\begin{array}{l} (\overline{\mathbf{ReadDetect}}\ p* > 0 \wedge \overline{\mathbf{RWMode}}\ p* > 0 \\ \wedge \overline{\mathbf{PageFrame}}\ pf\ p* > 0) \Rightarrow \overline{\mathbf{RWMode}}\ p* \leq 1 \end{array}$$

This new property should in turn be an invariant of the protocol. So, the verification algorithm is defined as a process which iteratively builds a strengthened invariant. The process must converge because the language of properties is finite. Its result is either "false" or a collection of properties (the strengthened invariant) which must be satisfied by the initial state of the system. The verification that this condition is satisfied can also be mechanized using the technique presented in this section but this step is not considered further in this paper.

The verification algorithm is shown in Figure 4. The algorithm starts from the TODO set of properties. Each property Q of this set is considered in turn up to convergence (which occurs when the new properties are implied by the already generated properties). The set of projections μ_j^i from variables of rule Rl_i (of the protocol Prt) to property Q is computed. Then the weakest precondition Q_j^i of property Q for rule Rl_i with projection μ_j^i is evaluated using function WP. Once this new property is normalized through $\mathcal{N}$, it is added to the TODO set if this property is not implied by properties in TODO $\cup$ DONE ($\mathcal{N}$ew function call). At each iteration step, the generated system in sets TODO and DONE is checked for contradiction by calling function $\mathcal{N}$otfalse.

This algorithm terminates because the set of properties that can be produced is finite. The sets of variables used to define protocols and properties are assumed to be disjoint (without loss of generality since renaming can be applied otherwise). We write PROP_P^{Prt}

```
Verify(P, Prt) =
  let TODO = {P}, DONE = ∅
  let Prt = {Rl_1, ..., Rl_n}
  let  For i ∈ [1,n], PROJ_i = set of projections from
       Rl_i to P. PROJ_i = {μ^i_1, ..., μ^i_{k_i}}
  while TODO ≠ ∅ and 𝒩otfalse(DONE ∪ TODO)
  do
    let Q ∈ TODO
    TODO := TODO − {Q}
    DONE := DONE ∪ {Q}
    for i ∈ [1,n], j ∈ [1,k_i] do
      Q^i_j = 𝒩 (WP (Q, Rl_i, μ^i_j))
      let R_1 ∧ ··· ∧ R_m = Q^i_j
      for k ∈ [1,m] do
        if 𝒩ew(R_k, DONE ∪ TODO) then
          TODO := TODO ∪ {R_k}
        end if
      end for
    end for
  end while
  let {P_1, ..., P_n} = DONE
  result = 𝒩_1(P_1 ∧ ··· ∧ P_n)
```

Figure 4: Verification algorithm

$$\mathrm{WP}\,(P, T_1\ \neg T_2 \rightarrow T_3, \mu) =$$

$$\underbrace{\bigwedge \overline{\mathbf{R_i}}\ y_1\ \ldots\ y_n > K}_{\text{part 1}} \qquad \left\{ \begin{array}{l} K = 0|1, \forall R_i \text{ such that} \\ \mathrm{Card}(\{\mathbf{R_i}\ x_1\ \ldots\ x_n \in T_1 \mid \\ \quad \mu(x_j) = y_j \vee y_j = *\}) > K \end{array} \right.$$

$$\underbrace{\bigwedge \overline{\mathbf{R'_i}}\ y_1\ \ldots\ y_n \leq 0}_{\text{part 2}} \qquad \left\{ \begin{array}{l} \forall R'_i \text{ such that } \mathbf{R'_i}\ x_1\ \ldots\ x_n \in T_2, \\ \quad \mu(x_j) = y_j,\ x_j \neq * \Rightarrow y_j \neq * \end{array} \right.$$

$$\underbrace{\Rightarrow \mathcal{S}\left(P\left[\begin{array}{c}\overline{\mathbf{R''_i}}\ y_1\ \ldots\ y_n + I_i \\ \hline \overline{\mathbf{R''_i}}\ y_1\ \ldots\ y_n\end{array}\right]\right)}_{\text{part 3}} \qquad \left\{ \begin{array}{l} \forall R''_i \text{ appearing in } P \text{ with} \\ I_i = \quad \mathrm{Card}(\{\mathbf{R''_i}\ x_1\ \ldots\ x_n \in T_3 \mid \\ \qquad\quad \mu(x_j) = y_j \vee y_j = *\}) \\ \qquad -\mathrm{Card}(\{\mathbf{R''_i}\ x_1\ \ldots\ x_n \in T_1 \mid \\ \qquad\quad \mu(x_j) = y_j \vee y_j = *\}) \end{array} \right.$$

Figure 5: Weakest precondition generation

the lattice of properties specialized for P and Prt. It is a restriction of PROP built from free variables of P (which is noted $FV(P)$) and from relations appearing in Prt and P. The ordering of the lattice is $\Rightarrow$ and $\vee$ is the smallest upper bound. It is finite modulo usual equivalence (which can be verified through a transformation of properties into normal conjunctive and simplifications using the standard equivalences between connectors).

4.2 Auxiliary functions

This algorithm uses several auxiliary functions that are defined below.

Weakest precondition computation A projection from a rule Rl towards a property P a function μ of signature $\mathrm{FV}(Rl) \cup \{*\} \rightarrow \mathrm{FV}(P) \cup \{*\}$ such that:

- $x, y \in \mathrm{VAR}$,
  $x \neq y \Rightarrow \quad \mu(x) \neq \mu(y) \quad \vee \quad \mu(x) = \mu(y) = *$
- $\mu(*) = *$

Figure 5 presents the definition of WP. The result of WP takes the form (Figure 5): part 1 $\wedge$ part 2 $\Rightarrow$ part 3. Part 1 and part 2 represent condition of application of the updated proposition P in part 3. Part 1 and part 2 express the fact that the computed precondition has to be satisfied when the rule is applicable, that is to say when T_1 relations are available in the multiset (cardinality strictly greater than 0) and T_2 relations are not available in the multiset (cardinality less than or equal to zero). Part 3 is derived from the initial proposition P taking into account the modifications to cardinalities (according to the

$$\mathcal{S}(A \vee B) = \mathcal{S}(A) \vee \mathcal{S}(B)$$

$\mathcal{S}(\overline{\mathbf{R}}\ y_1\ \ldots\ y_n + I > K) =$
if $K - I < 0$ **then**
  True
else
  if $K - I > 1$ **then**
    False
  else
    $\overline{\mathbf{R}}\ y_1\ \ldots\ y_n > K - I$
  end if
end if

$\mathcal{S}(\overline{\mathbf{R}}\ y_1\ \ldots\ y_n + I \leq K) =$
if $K - I < 0$ **then**
  False
else
  if $K - I > 1$ **then**
    $\overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq 1$
  else
    $\overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K - I$
  end if
end if

Figure 6: Simplification function

creation and removal of relations as indicated by the rule). This part is then simplified by $\mathcal{S}$ defined in Figure 6.

Function $\mathcal{S}$ is used to ensure that the new property belongs to the language presented in Section 3.2. To achieve this goal, it is necessary to approximate properties involving constants other than 0 or 1. This transformation is a source of approximation of the algorithm.

Normalization function Function $\mathcal{N}$ defined in Figure 7 is used to normalize each property before it is submitted to the $\mathcal{N}$ew function defined below. $\mathcal{N}$ is the composition of two functions $\mathcal{N}_2$ and $\mathcal{N}_1$ (shown in Figure 7 as rewriting systems). $\mathcal{N}_2$ removes negations by propagating them inside expressions. $\mathcal{N}_1$ puts the proposition into normal conjunctive form; in addition it deletes redundant propositions and occurrences of True and False. Note that the dual cases in the definition of $\mathcal{N}$ have been omitted for the sake of conciseness.

Detection of new preconditions Function $\mathcal{N}$ew checks that a given property P is not implied by a the set of properties E:

$$\begin{array}{rcl} \mathcal{N}\text{ew}(P, E) & = & \textbf{let } E = \{P_1, \ldots, P_n\} \\ & & (P_1 \not\Rightarrow P) \wedge \cdots \wedge (P_n \not\Rightarrow P) \end{array}$$

4.3 Application to the Li and Hudak protocol

We have implemented a prototype of this algorithm in Objective Caml 2[1], an ML-like language developed at INRIA. This program uses BDD's (CMU bddlib[2]) to implement the $\mathcal{N}$otfalse test and to check the produced properties against initial conditions.

[1] Further information and full development environment can be found at `http://caml.inria.fr/ocaml/`
[2] Available at: `http://www.cs.cmu.edu/~modelcheck/bdd.html`

$$
\begin{array}{rcll}
\mathcal{N}(P) & = & \mathcal{N}_1(\mathcal{N}_2(P)) & \\
\\
\neg(\neg P) & \xrightarrow[\mathcal{N}_2]{} & P & \\
\neg(A \vee B) & \xrightarrow[\mathcal{N}_2]{} & \neg A \wedge \neg B & \\
\neg(A \wedge B) & \xrightarrow[\mathcal{N}_2]{} & \neg A \vee \neg B & \\
\neg \mathsf{True} & \xrightarrow[\mathcal{N}_2]{} & \mathsf{False} & \\
\neg \mathsf{False} & \xrightarrow[\mathcal{N}_2]{} & \mathsf{True} & \\
\neg \overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K & \xrightarrow[\mathcal{N}_2]{} & \overline{\mathbf{R}}\ y_1\ \ldots\ y_n > K & \\
\neg \overline{\mathbf{R}}\ y_1\ \ldots\ y_n > K & \xrightarrow[\mathcal{N}_2]{} & \overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K & \\
\\
A \vee (B \wedge C) & \xrightarrow[\mathcal{N}_1]{} & (A \vee B) \wedge (A \vee C) & \\
A \wedge \mathsf{False} & \xrightarrow[\mathcal{N}_1]{} & \mathsf{False} & \\
A \wedge \mathsf{True} & \xrightarrow[\mathcal{N}_1]{} & A & \\
A \vee \mathsf{False} & \xrightarrow[\mathcal{N}_1]{} & A & \\
A \vee \mathsf{True} & \xrightarrow[\mathcal{N}_1]{} & \mathsf{True} & \\
\overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K \wedge \overline{\mathbf{R}}\ y'_1\ \ldots\ y'_n > K' & \xrightarrow[\mathcal{N}_1]{} & \mathsf{False} & \text{if } K' \geq K \wedge (y_i = y'_i \vee y_i = *) \\
\overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K \wedge \overline{\mathbf{R}}\ y'_1\ \ldots\ y'_n \leq K' & \xrightarrow[\mathcal{N}_1]{} & \overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K & \text{if } K \leq K' \wedge (y_i = y'_i \vee y_i = *) \\
\overline{\mathbf{R}}\ y_1\ \ldots\ y_n > K \wedge \overline{\mathbf{R}}\ y'_1\ \ldots\ y'_n > K' & \xrightarrow[\mathcal{N}_1]{} & \overline{\mathbf{R}}\ y_1\ \ldots\ y_n > K & \text{if } K \geq K' \wedge (y'_i = y_i \vee y'_i = *) \\
\overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K \vee \overline{\mathbf{R}}\ y'_1\ \ldots\ y'_n > K' & \xrightarrow[\mathcal{N}_1]{} & \mathsf{True} & \text{if } K' \leq K \wedge (y'_i = y_i \vee y'_i = *) \\
\overline{\mathbf{R}}\ y_1\ \ldots\ y_n \leq K \vee \overline{\mathbf{R}}\ y'_1\ \ldots\ y'_n \leq K' & \xrightarrow[\mathcal{N}_1]{} & \overline{\mathbf{R}}\ y'_1\ \ldots\ y'_n \leq K' & \text{if } K \leq K' \wedge (y_i = y'_i \vee y_i = *) \\
\overline{\mathbf{R}}\ y_1\ \ldots\ y_n > K \vee \overline{\mathbf{R}}\ y'_1\ \ldots\ y'_n > K' & \xrightarrow[\mathcal{N}_1]{} & \overline{\mathbf{R}}\ y'_1\ \ldots\ y'_n > K' & \text{if } K \geq K' \wedge (y'_i = y_i \vee y'_i = *)
\end{array}
$$

Figure 7: Normalization function

When applied to the Li and Hudak SVM protocol (as described in figures 2 and 3), our algorithm returns a strengthened invariant made of 62 properties. The initial conditions of the system are very simple: all pages are used in read mode on exactly one node. There are no invalidation, no read and write faults, no page in read-write access. Thus the property below characterizes the initial state:

$$\bigwedge \left(\begin{array}{l} \overline{\mathbf{RMode}}\ p\ n > 0 \wedge \overline{\mathbf{RMode}}\ p\ * \leq 1 \\ \overline{\mathbf{PageFrame}}\ *\ p\ * > 0 \\ \overline{\mathbf{PageFrame}}\ *\ p\ * \leq 1 \\ \overline{\mathbf{Ok}}\ p > 0 \wedge \overline{\mathbf{Ok}}\ p \leq 1 \\ \overline{\mathbf{RWMode}}\ *\ * \leq 0 \\ \overline{\mathbf{InvalidationPhase}}\ *\ *\ * \leq 0 \\ \overline{\mathbf{WriteDetect}}\ *\ * \leq 0 \\ \overline{\mathbf{ReadDetect}}\ *\ * \leq 0 \end{array} \right)$$

The generated properties are automatically checked against these initial conditions and we can conclude that the Li and Hudak protocol satisfied the expected properties (without any restriction on the numbers of nodes and pages).

5 Related work

In the field of multiprocessor caches, shared virtual memories and distributed file systems, numerous approaches have been used to formalize and verify coherence protocols. They can be divided into two categories: automatic and manual ones. The authors of survey [19] provide an overview of automatic verification techniques: state enumeration, symbolic model checking and symbolic state model. In state enumeration methods, protocols are described as sets of iterated guarded commands. State changes are expressed in an imperative style. The Murφ tool [6] has been used to check various coherence protocols. It is based on protocol descriptions which are close to the implementation and suffers from the state space explosion problem; therefore only small systems can be fully verified. Regarding symbolic model checking, a first issue is to translate the protocol state graph into boolean formulae represented as Binary Decision Diagrams (BDD). In [19], this translation is achieved through a formalization of protocol entities (processor, caches, ...) as finite state modules communicating asynchronously. This step is not easy and introduces irrelevant details in the protocol model. Another approach is to transform an imperative description of the protocol, as used for example in Murφ, directly into a BDD [10]. This approach is easier for the end-user. However the generated BDD's are huge and thus elaborated generation techniques are necessary to be able to apply the technique on real protocols [1, 9]. The second issue is to effectively use the generated BDD for property verification. Even if BDD are well known for reducing the average complexity, its worst-case space complexity is still exponential with respect to the number of variables and thus this approach also suffers from the state space explosion problem. Third, the symbolic state model exploits homogeneity in a coherence protocol to drastically reduce the state space, mainly by abstracting the number of nodes.

This techniques works well for the Li and Hudak protocol; however, as underlined by the authors [19], it does not seem to be easy to extend to other kinds of protocols like linked list protocols.

A variety of manual or semi-automatic techniques have also been proposed. In [22], model checking is used over an abstraction of individual runs of a protocol. This abstraction reduces considerably the state space and makes model checking cheaper. However designing such an abstraction needs considerable human involvement. In fact, this paper is based on a formalization [17] made for manual proof of the same properties. Another approach is to use a theorem prover to assist manual demonstration. The work presented in [18] is based on proving properties over a protocol abstraction with a theorem prover. This abstraction is linked to the implementation through an aggregation function which must possess certain characteristics. The advantage of theorem-proving techniques is that demonstrated properties are usually more general (not related to the number of nodes in the formalized system for example). However they need considerable involvement from the user. It is not clear whether those techniques could be applied in an software development process where specifications can evolve rapidly.

Our approach is closer to the automatic techniques listed above. The Gamma rewriting rules used in protocol description are akin to automata description through state changes used in state enumeration, symbolic model checking and symbolic state model techniques. But our framework also involves an abstraction step which makes it possible to prove properties of systems without any restrictions on their number of states or components. Furthermore we alleviate the state explosion problem by specializing automatically the lattice of properties for a given pair of an invariant and a protocol.

6 Conclusion

We have presented in this paper an new approach to formalize coherence protocols and to check invariance properties. A first version of our verification algorithm has been implemented and applied successfully to the Li and Hudak protocol. We are now focusing on improving the efficiency of this prototype. Regarding the formalism itself, we are also studying its application to other domains. Experiments have been made in the area of security, for example verifying a building access control protocol [20]. Finally, the remaining problem (the most crucial one from an engineering point of view) is to link a formalized protocol to its implementation. As stated in section 3.2, different implementations can be derived from the same protocol model. Our objective is to develop an environment to translate a formalized protocol in a running implementation, mainly using Aspect-Oriented Programming techniques [16].

Acknowledgment

Thanks are due to Christophe René and Michaël Périn for commenting an earlier version of this paper. We also thank Thomas Colcombet for commenting a draft of this paper and for providing us with a first Objective Caml interface to the CMU bddlib.

References

[1] A. J. Hu, G. York, and D.L. Dill. New Techniques for Efficient Verification with Implicity Conjoined BDDs. In *31st ACM/IEEE Design Automation Conference (DAC)*, San Diego, CA, June 1994. San Diego Convention Center. ch. 18.2.

[2] J.-P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, Jan. 1993.

[3] J.-P. Banâtre and D. Le Métayer. Gamma and the Chemical Reaction Model: Ten Years After. In J. Andreoli, C. Hankin, and D. Le Métayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 3–41. Imperial College Press, 1996.

[4] B. N. Bershad and M. J. Zekauskas. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, Sept. 1991.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–64. Association for Computing Machinery SIGOPS, Oct. 1991.

[6] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design, VLSI in Computers and Processors*, pages 522–525, Los Alamitos, Ca., USA, Oct. 1992. IEEE Computer Society Press.

[7] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 372–385, New York, NY, USA, 1996. ACM Press.

[8] C. Hankin, D. Le Métayer, and D. Sands. Refining Multiset Transformers. *Theorical Computer Science*, 192(2):233–258, Feb. 1998.

[9] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In A.-S. IEEE, editor, *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 266–271, Dallas, TX, June 1993. ACM Press.

[10] A. J. Hu, D. L. Dill, A. J. Drexler, and C. H. Yang. Higher-level specification and verification with bdds. In *Workshop on Computer-Aided Verification*, Montreal, Quebec, June 1992.

[11] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, June 1996.

[12] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, Jan. 1994.

[13] D. Le Métayer. Software architecture styles as graph grammars. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23, San Francisco, California, USA, Oct. 1996.

[14] K. Li and P. R. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings 1986 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, New York, NY, 1986. ACM.

[15] D. Mentré. *Une approche pour la spécification, la vérification et l'exécution de protocoles de cohérence de MVP*. PhD thesis, Irisa, 2000. To appear.

[16] D. Mentré, D. Le Métayer, and T. Priol. Towards designing SVM coherence protocols using high-level specifications and aspect-oriented translations. In *Proceedings of the 1999 Workshop on Software Distributed Shared Memory, held in conjunction with 1999 International Conference in Supercomputing*, pages 101–107, Rhodos, Greece, 1999.

[17] L. B. Mummert, J. M. Wing, and M. Satyanarayanan. Using belief to reason about cache coherence. In *13th ACM Conference on Principles of Distributed Computing*, Los Angeles, CA, USA, Aug. 1994.

[18] S. Park and D. L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, July/Aug. 1998.

[19] F. Pong and M. Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Computing Surveys*, 29(1):82–126, Mar. 1997.

[20] M. Périn. Cohérence de spécifications multi-vues. In *Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2000*, Grenoble, France, Jan. 2000.

[21] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, Massachusetts, Oct. 1996. ACM Press.

[22] J. M. Wing and M. Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28(2–3):273–299, Apr. 1997.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399