Verification of a distributed Cache Memory by using abstractions *

Susanne Graf

VERIMAG **, Rue Lavoisier, F-38330 Monbonnot, e-mail: graf@imag.fr

Abstract. The purpose of this paper is to verify a distributed cache memory system by using the following general verification method: verify the properties characterizing a complex system on some small finite abstraction of it, obtained as a composition of abstractions of each component of the system. For a large class of systems including infinite state systems, the abstractions of the components can be obtained by replacing all operators on concrete domains by abstract operators on some abstract domain. This holds also for the abstraction of the control part of the system as we consider a kind of guarded command programs where all the control is expressed in terms of operations on explicit control variables.

1 Introduction

The purpose of this paper is to show the practical applicability of the verification method proposed in [LGS+92, GL93a, GL93b, CGL91, Lon93] for infinite state systems. This verification method, based on the principle of abstract interpretation [CC77], proposes to verify a program defining some complex system, where the specification must be given in the form of a set of $\forall$CTL* [SG90] formulas, as follows: define an appropriate abstract program, obtained compositionally from the the given program, and verify the required properties on it. Our way of computing abstract programs is very similar to that proposed in [CGL91, Lon93], but our concept of compositionality is different from that proposed in [Lon93] or in [Pnu85]. We construct a global abstraction of the system by composing abstractions of its components, whereas the other method consists in deducing properties of the composed system from properties of its components. Both approaches are useful, but in the example we treat in this paper, the global properties cannot be deduced easily from properties of the components. An abstraction of each component is obtained applying the principle of abstract interpretation by means of a relation ϱ relating the domain of its variables and the domain of the set of some abstract variables.

In [Loi94] is described a tool allowing to verify finite state systems in a fully automatic way by using this method. Here, we show that the same method is also tractable in practice for infinite state systems where a complete automatization is not possible. In fact, if — depending on the formula one wants to verify — for each component P_i one can guess an appropriate abstraction relation ϱ_i verification becomes often a relatively simple task as

- the corresponding finite state abstract program is reasonably easy to obtain,
- the verification of the properties on the obtained abstract program can be done fully automatically.

In Section 2, we recall all the ingredients we need for our verification method:

- a simple program formalism similar to that used e. g., in [Pnu86],

* This work was partially supported by ESPRIT Basic Research Action "REACT"

** Verimag is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Universitée J. Fourier and Verilog SA associated with IMAG

- a method to compute abstract programs, consisting in defining for each operator on the concrete domains a corresponding *abstract operator*; this is the only step in the proposed method that cannot be fully automated.
- the temporal logic CTL* and its fragments, used for the description of properties,
- the preservation results allowing to deduce the validity of a property on the concrete program from its validity on the abstract program and
- the compositionality results allowing to compute an abstract program by composing abstractions of its components.

We illustrate all the definitions and results on a small buffer example. In Section 3, we verify — by applying in a systematic way our method — a distributed cache memory system defined in [ABM93]. In [DGJ+93] several (complex) correctness proofs are given for this system based on different methods. Using our method, the verification of this system is almost as simple as the verification of the tiny buffer, as we need almost the the same abstract operations.

2 A verification method using abstractions

2.1 A program description formalism

We adopt a simple program formalism which is not meant as a real programming language but which is sufficient to illustrate our method. A complex system is a parallel composition of basic programs P of the following form

Variables :	$x_1 : T_1, ..., x_n : T_n$
Transitions :	(ℓ_1) $action_1(x_1, ..., x_n,\ x'_1, ..., x'_n)$
	...
	(ℓ_p) $action_p(x_1, ..., x_n,\ x'_1, ..., x'_n)$
Initial States :	$init(x_1, ..., x_n)$

where x_i are variables of type T_i and $L_P = \{\ell_1, ..., \ell_n\}$ is the set of program labels. Each $action_i$ is an expression with variables in the set of program variables and a set of primed variables which is a copy of the set of state variables; as in [Pnu86, Lam91], $action_i$ represents a transition relation on the domain of the program variables by interpreting the valuations of $X_P = (x_1, ..., x_n)$ as the state *before*, and the valuations of $X'_P = (x'_1, ..., x'_n)$ as the state *after* the transition. We denote the set of valuations of X_P by $Val(X_P)$.

Semantics : Program P defines in an obvious manner a transition system $S_P = (Q_P, R_P)$ where

- $Q_P = Val(X_P)$ is the set of states,
- $R_P \subseteq Q_P \times Q_P$ is a transition relation defined by $R_P = \{(q, q') \mid \exists i \,.\, action_i(q, q')\}$.

The predicate *init* defines the set of initial states. It is used in the formulas specifying the program: properties are in general of the form $init \Rightarrow \phi$ where ϕ expresses the property one wants to verify.

We do not distinguish variables representing inputs as they need not be treated in a particular manner. However, we annotate in the programs the variables which are meant as inputs as this makes programs easier to read.

Labels are used to name "events" or "actions". If ℓ_i is a label and (v, v') a pair of valuations such that $action_i(v, v')$ is true, then the transition from state v to state v' is an event ℓ. If e is the valuation of the "input" variables extracted from v, then we call this event also $\ell_i(e)$. Events are used for the expression of properties.

Example 1 an infinite lossy buffer. The following program represents an unbounded buffer taking as input elements e of some data domain *elem*. The event *push*(e) enters e (if it has never been entered yet) into the buffer or arbitrarily "loses" it, and *pop*(e) takes e out of the buffer if it is its first element.

Variables :	$e : elem$ (Input) $E : set$ of $elem$ (already occurred events $push(e)$) $B : buffer$ of $elem$
Transitions :	(push(e)) $allowed(e, E, E') \wedge (add(B, e, B') \vee \mathbf{unch}(B))$ (pop(e)) $first(B, e) \wedge tail(B, e, B') \wedge \mathbf{unch}(E)$
Initial States :	$empty(B)$

E contains the elements e such that $push(e)$ has already occurred, and $\mathbf{allowed}(e, E, E')$ is necessarily *false* if $e \in E$. All other predicates have the intuitive meanings: $\mathbf{add}(B, e, B')$ holds if B' is obtained by adding element e at the end of the buffer B; $\mathbf{tail}(B, e, B')$ holds if B' is obtained by eliminating e from B if e is its first element ($\mathbf{first}(B, e)$ is *true*); $\mathbf{empty}(B)$ is *true* if B is the empty buffer. $\mathbf{unch}(X)$, where $X = x_1, ... x_n$ is a tuple of program variables, represents the transition relation which lets all variables in X unchanged, i. e., $\mathbf{unch}(X) = \bigwedge_i (x_i' = x_i)$.

We use predicates of the form $add(B, e, B')$ instead of $B' = ADD(B, e)$ where ADD is a function, as abstract operations are in general nondeterministic. This is also the way of representing operations which is proposed, e. g. in [CGL91, Lam91].

Composed programs : In [GL93b] we obtain our results for more general parallel composition operators, but here we need only asynchronous composition. If P_1 and P_2 are programs defined on a tuple of state variables X_1, respectively X_2, then $P_1 \| P_2$ is the parallel composition of P_1 and P_2 defining the transition system $S = (Val(X_1 \cup X_2), R)$ where

$$R \;=\; R_{P_1} \wedge \mathbf{unch}(X_2 - X_1) \;\; \vee \;\; R_{P_2} \wedge \mathbf{unch}(X_1 - X_2)$$

Each transition of $P_1 \| P_2$ is either a transition of P_1 which leaves all variables which are declared in P_2 but not in P_1 unchanged or the other way round.

2.2 Abstract programs

As proposed in [CGL91, LGS+92], given a program *Prog* and a predicate ϱ on the variables of *Prog* and a tuple of *abstract variables* $X^A = (x_1^A, ... x_m^A)$, representing a relation between the concrete and the abstract domain (a function in [CGL91]), then any program $Prog^A$ defined on X^A, such that for each action *action* of *Prog* there exists an action $action^A$ of $Prog^A$ with the same label, such that

$$\exists X \exists X' \,.\, \varrho(X, X^A) \wedge \varrho(X', X^{A'}) \wedge action(X, X') \;\Rightarrow\; action^A(X^A, X^{A'}) \quad (1)$$

and

$$\exists X \,.\, \varrho(X, X^A) \wedge init(X) \;\Rightarrow\; init^A(X^A)$$

is an *abstraction* or more precisely a *ϱ-abstraction* of *Prog*.

When verifying composed programs, it is interesting to compute an abstract program compositionally, i. e., by composing abstract component programs. From a more general result given in [GL93a], we obtain the following result which is sufficient for the verification of the distributed cache memory system in the following section.

Proposition 1. *Let P_1 and P_2 be programs and ϱ_i* total functions *from the domain of the variables of P_i into some abstract domains such that $\varrho_1 \cap \varrho_2$ is total and P_1^A, P_2^A are ϱ_i-abstractions of P_i, then $P_1^A \parallel P_2^A$ is a $(\varrho_1 \cap \varrho_2)$-abstraction of $P_1 \parallel P_2$.*

Computation of abstract programs in practice : The idea of abstract interpretation [CC77] is to replace every *function* on the concrete domain used in the program by a corresponding *abstract function* on the abstract domain, and then to analyze the so obtained simpler abstract program instead of the concrete one. Consider the program $Prog^A$ obtained by replacing every basic predicate $op(X, X')$ on the concrete variables by a predicate $op^A(X^A, X^{A'})$ on the abstract variables is a ϱ-abstraction of $Prog$ if, instead of (1), for every basic operation

$$\exists X \exists X' \,.\, \varrho(X, X^A) \wedge \varrho(X', X^{A'}) \wedge op(X, X') \quad \Rightarrow \quad op^A(X^A, X^{A'}) \qquad (2)$$

holds. If the expressions in $Prog$ are negation free (as in our buffer), then $Prog^A$ is in fact a ϱ-abstraction of $Prog$. The definition of abstract predicates op^A is the only part of our verification method which cannot be fully automatized. But as we will see, we only need a restricted number of such abstract operations in order to verify a whole class of programs. For example, in the domain of protocol verification, the used data structures are "messages" on which no operations are carried out, "memories" or "registers" in which data can be stored, integers which are mostly used as counters and "buffers" with the usual operations *add*, *tail*, *first*,.., as in our examples. In [CGL91] a similar method is proposed.

Example 2 An abstract lossy buffer. To illustrate the idea, consider again the buffer of Example 1. In order to show that the buffer has the property of "order preservation" (see Example 3), it is sufficient to show that the order of any pair of elements $(e_1, e_2) \in elem \times elem$ is preserved. All the information we need about the content of the buffer B is, if and in which order, it contains the elements e_1 and e_2. Furthermore, as each element is supposed to be put into the buffer at most once, we need not distinguish amongst the valuations of B containing e_i more than once. Similarly, for the input variable e we only need to distinguish if its value is e_1, e_2 or any other value. Concerning the value of E determinating which events $push(e)$ are still allowed, we only need to know if the event $push(e_1)$, respectively $push(e_2)$ is still possible or not. This leads us naturally to the abstract domain defined by the abstract variables,

$$\begin{aligned} &e_A : elem_A^2 = \{0, 1, 2\} \\ &E_A^1, E_A^2 : Bool \\ &B_A : buffer_A^2 = \{\epsilon, e_1, e_2, e_1 \bullet e_2, e_2 \bullet e_1, \perp\} \end{aligned}$$

and the following abstraction relation ϱ^2 defining the correspondence between the concrete and the abstract variables

$$\begin{aligned} \varrho^2(e, E, B, \;\; e_A, (E_A^1, E_A^2), B_A) \;=\; & \varrho^2_{elem}(e, e_A) \;\wedge\; \varrho^2_{set\_of\_elem}(E, (E_A^1, E_A^2)) \;\wedge \\ & \varrho^2_{buffer}(B, B_A) \end{aligned}$$

where for $e : elem$ and $e_A : elem_A^2$

$$\begin{aligned} \varrho^2_{elem}(e, e_A) = & ((e_A = 0) \equiv (e \notin \{e_1, e_2\})) \wedge \\ & ((e_A = 1) \equiv (e = e_1)) \wedge \\ & ((e_A = 2) \equiv (e = e_2)) \end{aligned}$$

for E : *set* of *elem* and $E_A^1, E_A^2 : Bool$, E_A^i expresses that e_i has not occurred yet:

$$\varrho^2_{set\_of\_elem}(E, E^1_A, E^2_A) = ((E^1_A = \exists E' \,.\, allowed(\mathrm{e}_1, E, E') \wedge \\ ((E^2_A = \exists E' \,.\, allowed(\mathrm{e}_2, E, E')$$

and for B : *buffer* of *elem* and B_A : $buffer^2_A$

$$\varrho^2_{buffer}(B, B_A) = \\ ((B_A = \epsilon) \equiv empty(B_{|\{\mathrm{e}_1,\mathrm{e}_2\}})) \wedge ((B_A = \mathrm{e}_1 \bullet \mathrm{e}_2) \equiv (B_{|\{\mathrm{e}_1,\mathrm{e}_2\}} = \mathrm{e}_1 \bullet \mathrm{e}_2)) \wedge \\ ((B_A = \mathrm{e}_1) \equiv (B_{|\{\mathrm{e}_1,\mathrm{e}_2\}} = \mathrm{e}_1)) \wedge ((B_A = \mathrm{e}_2 \bullet \mathrm{e}_1) \equiv (B_{|\{\mathrm{e}_1,\mathrm{e}_2\}} = \mathrm{e}_2 \bullet \mathrm{e}_1)) \wedge \\ ((B_A = \mathrm{e}_2) \equiv (B_{|\{\mathrm{e}_1,\mathrm{e}_2\}} = \mathrm{e}_2)) \wedge ((B_A = \bot) \text{ in all other cases }))$$

where $B_{|\{\mathrm{e}_1,\mathrm{e}_2\}}$ is the buffer B restricted to the elements e_1 and e_2. In order to construct an abstract program, we have to define abstract predicates for all the basic predicates used in the concrete buffer program, such as *allowed*, *add*, *tail*, **unch**, etc.

In the case that every abstract variable is related to a single concrete variable, the abstract predicate associated with **unch**(v) is obviously **unch**(v_A) for any abstract variable v_A related to v. The following abstract predicates satisfy the condition (2).

$$allowed^2_A(e_A, (E^1_A, E^2_A), (E^{1\prime}_A, E^{2\prime}_A)) = (E^{1\prime}_A \equiv E^1_A) \wedge (E^{2\prime}_A \equiv E^2_A) \;\wedge\; (e_A = 0) \;\vee \\ (E^1_A \wedge \neg E^{1\prime}_A) \wedge (E^2_A \equiv E^{2\prime}_A)) \;\wedge\; (e_A = 1) \;\vee \\ (E^1_A \equiv E^{1\prime}_A) \wedge (E^2_A \wedge \neg E^{2\prime}_A))) \;\wedge\; (e_A = 2)$$

$$add^2_A(B_A, e_A, B'_A) = (B_A = B'_A) \qquad \wedge (e_A = 0) \vee \\ (B_A \in \{\epsilon, \mathrm{e}_2\}) \wedge (B'_A = \mathrm{e}_1 \bullet B_A) \vee (B_A \notin \{\epsilon, \mathrm{e}_2\}) \wedge (B'_A = \bot) \wedge (e_A = 1) \vee \\ (B_A \in \{\epsilon, \mathrm{e}_1\}) \wedge (B'_A = \mathrm{e}_2 \bullet B_A) \vee (B_A \notin \{\epsilon, \mathrm{e}_1\}) \wedge (B'_A = \bot)) \wedge (e_A = 2)$$

$$tail^2_A(B_A, e_A, B'_A) = (B_A = B'_A) \qquad \wedge (e_A = 0) \vee \\ ((B_A \in \{\mathrm{e}_1, \mathrm{e}_1 \bullet \mathrm{e}_2\}) \Rightarrow (B_A = B'_A \bullet \mathrm{e}_1)) \wedge (e_A = 1) \vee \\ ((B_A \in \{\mathrm{e}_2, \mathrm{e}_2 \bullet \mathrm{e}_1\}) \Rightarrow (B_A = B'_A \bullet \mathrm{e}_2)) \wedge (e_A = 2)$$

$$empty^2_A(B_A) = (B_A = \epsilon)$$

$$first^2_A(B_A, e_A) = (e_A = 0) \vee \\ (B_A \in \{\mathrm{e}_1, \mathrm{e}_1 \bullet \mathrm{e}_2, \bot\}) \wedge (e_A = 1) \vee \\ (B_A \in \{\mathrm{e}_2, \bot\}) \wedge (e_A = 2)$$

tail is an example of a predicate defining a function on the concrete domain, but which is nondeterministic on the given abstract domain; $tail^2_A(\bot, 1, B'_A)$ necessarily evaluates to *true* for any value of B'_A (the value of the buffer in the next state).

Using these abstract predicates, the definition of a program representing a ϱ-abstraction of the buffer program becomes trivial. We just replace variables by corresponding abstract variables and every occurrence of a predicate by corresponding abstract one. The resulting abstract program looks almost as the concrete program but defines a very small finite transition system.

The useful abstractions are often obtained by using this kind of abstract domains. Here, we gave in detail the more complicated abstraction of a buffer particularizing two different data elements. But often, it is sufficient to particularize in the same way a single data element. The corresponding abstraction relations ϱ^1_{elem}, $\varrho^1_{set\_of\_elem}$, ϱ^1_{buffer} and abstract predicates $allowed^1_A$, add^1_A, $tail^1_A$,... can be defined by simplifying the above definitions in an obvious manner. For the verification of the cache memory we use also existential abstractions of buffers. The corresponding abstract predicates $add^{ex}(e_A)$, $tail^{ex}(e_A)$,... necessarily evaluate to *true* if e_A is an allowed value of the existentially abstracted buffer.

2.3 Temporal Logic

It remains to recall the definition of temporal logic. Here we restrict ourselves to subsets of CTL* [EH83] for the expression of properties. The preservation results in [LGS+92] are given for subsets of the more powerful branching time μ-calculus augmented by past time modalities.

Definition 2. CTL* is the set of state formulas given by the following definition.

1. Let $\mathcal{P}$ be a set of atomic (a) state respectively (b) path formulas.
2. If ϕ and ψ are (a) state respectively (b) path formulas then $\phi \wedge \psi$, $\phi \vee \psi$ and $\neg\phi$ are (a) state respectively (b) path formulas.
3. If ϕ is a path formula then $\mathbf{A}\phi$ and $\mathbf{E}\phi$ are state formulas.
4. If ϕ and ψ are (a) state or (b) path formulas then $\mathbf{X}\phi$, $\phi\mathcal{U}\psi$ and $\phi\mathcal{W}\psi$ are path formulas.

As usual, we also use the abbreviations $\mathbf{F}\phi$ denoting $true\mathcal{U}\phi$ and $G\phi$ denoting $\phi\mathcal{W}false$.

$\mathcal{U}$ is a strong and $\mathcal{W}$ a weak "until" operator, a sequence satisfies $\phi\mathcal{W}\psi$ if ϕ holds up to some point in which ψ holds, and $\phi\mathcal{U}\psi$ expresses the same property and moreover the obligation that such a point satisfying ψ exists. That means that $\mathcal{U}$ and $\mathcal{W}$ are duals by inversing the arguments: $\phi\mathcal{W}\psi = \neg(\neg\psi\mathcal{U}\neg\phi)$.

$\forall$CTL* [SG90] is the subset of CTL* obtained by allowing negations only on atomic formulas and restricting Rule 3 by allowing only the universal path quantifier **A**.

The *semantics* of CTL* is defined over Kripke structures of the form $M = (S, \mathcal{I})$ where $S = (Q, R)$ is a transition system and $\mathcal{I}$ is a interpretation function mapping elements of $\mathcal{P}$ into sets of states of S.

Definition 3. A *path* in a transition system S is an infinite sequence $\pi = q_0 q_1 \ldots$ such that for every $i \in \mathcal{N}$. $R(q_i, q_{i+1})$. We denote by π_n the nth state of path π and by π^n the sub-path of π starting in its nth state.

Definition 4. Let be $M = (S, \mathcal{I})$ a Kripke structure, $q \in Q$ and π a path of M. Then the satisfaction ($\models_M$) of CTL* formulas on M is defined inductively as follows.

1. Let be $p \in \mathcal{P}$. Then $q \models_M p$ iff $q \in \mathcal{I}(p)$ and $\pi \models_M p$ iff $\pi_0 \in \mathcal{I}(p)$.
2. Let ϕ and ψ be (a) state respectively (b) path formulas. Then,
   (a) $q \models_M \neg\phi$ iff $q \not\models_M \phi$, $q \models_M \phi \wedge \psi$ iff $q \models_M \phi$ and $q \models_M \psi$, $q \models_M \phi \vee \psi$ iff $q \models_M \phi$ or $q \models_M \psi$.
   (b) analogous by replacing q by π
3. Let ϕ be a path formula. Then,
   $q \models_M \mathbf{A}\phi$ iff for every path π starting in q, $\pi \models_M \phi$ and
   $q \models_M \mathbf{E}\phi$ iff there exists a path π starting in q such that $\pi \models_M \phi$.
4. Let ϕ and ψ be (a) state respectively (b) path formulas. Then,
   (a) $\pi \models_M \mathbf{X}\phi$ iff $\pi_1 \models_M \phi$,
   $\pi \models_M \phi\mathcal{U}\psi$ iff there exists $n \in \mathcal{N}$ such that $\pi_n \models_M \psi$ and $\forall k < n$. $\pi_k \models_M \phi$,
   $\pi \models_M \phi\mathcal{W}\psi$ iff for all $n \in \mathcal{N}$. $((\forall k < n \,.\, \pi_k \models_M \neg\psi)$ implies $\pi_n \models_M \phi)$.
   (b) the same definition obtained by replacing in (a) all states π_i by subsequences π^i.

We say that $M \models \phi$ iff $q \models_M \phi$ for all states of M.

From the more general results given in [LGS+92] we obtain the following proposition concerning preservation of properties of $\forall$CTL*.

Proposition 5 Preservation of $\forall$CTL*. *Let Prog be a program, ϱ a total relation from the domain of Prog into some abstract domain, and $Prog_A$ a ϱ-abstraction of Prog. Then, for any $\phi \in \forall$CTL*, $\mathcal{P}$ the set of atomic formulas occurring in ϕ and $\mathcal{I}$ an interpretation function mapping $\mathcal{P}$ into sets of states of S_{Prog}, we have*

$Im[\varrho^{-1}] \circ Im[\varrho] \circ \mathcal{I}\ (p) \subseteq \mathcal{I}\ (p)$ (*) for all $p \in \mathcal{P}$ occurring positive in ϕ
implies
$(S_{Prog_A}, Im[\varrho] \circ \mathcal{I}) \models \phi \quad \Rightarrow \quad (S_{Prog}, \mathcal{I}) \models \phi$

where $Im[\varrho]$ is the image function of ϱ. Condition (*) is called consistency *of ϱ with $\mathcal{I}(p)$.*

This proposition expresses that, if $\phi \in \forall$CTL* holds on a ϱ-abstraction of the program *Prog* by translating the interpretations of all atomic propositions occurring in the formula by $Im(\varrho)$ into predicates on the abstract domain, and if all these predicates are consistent with ϱ, then we can deduce that ϕ holds on *Prog*. Consistency is not needed for predicates that occur only negated in ϕ as $Im[\varrho^{-1}](\overline{Im[\varrho](\mathcal{I}(p))}) \subseteq \overline{\mathcal{I}(p)}$. We conclude that, if ϕ holds on $Prog_A$ using the abstract interpretation of $\neg p$ $(\overline{Im[\varrho](\mathcal{I}(p))})$, then a stronger property than ϕ using the concrete interpretation of $\neg p$ $(\overline{\mathcal{I}(p)})$ holds on *Prog*. In particular, for the verification of a formula of the form $init \Rightarrow \phi$, *init* need not to be consistent with ϱ.

Example 3. The property of *order preservation* can be expressed by the following set of formulas on the set of "observable" atomic predicates

$\mathcal{P} = \{init, enable(push(e)), after(push(e)), enable(pop(e)), after(pop(e)), ...\}$,

where $enable(\ell)$ is interpreted as the set of states in which event ℓ is possible, and $after(\ell)$ those in which ℓ has just occurred — $after(\ell)$ becomes expressible by adding an explicit boolean program variable $after\_\ell$ which is *true* exactly after any event ℓ.

$$\forall e', e \in elem \ : \ init \ \Rightarrow \ \mathsf{A}(\ [\neg after(push(e))\mathcal{W}after(push(e'))] \ \Rightarrow \\ [\neg enable(pop(e))\mathcal{W}after(pop(e'))] \ )$$

These formulas can be transformed into $\forall$CTL* formulas in which only the predicates $after(push(e))$ and $after(pop(e'))$ occur non negated. In order to verify that the concrete buffer program has the property of order preservation, it is sufficient to verify the formula obtained by instanciating e_1 for e and e_2 for e' on the abstract program defined in Example 2. In fact, as e_1 and e_2 represent an arbitrary pair of data values, this verification of a single representative of the set of formulas is sufficient. It is easy to obtain the consistency of predicates of the form $after(\ell)$ by not abstracting the variable $after\_\ell$. In the sequel we suppose, without mentioning it explicitly, that for every predicate $after(\ell)$ occurring in the considered formula such a variable is defined.

3 Verification of a distributed cache memory

3.1 Concrete and abstract specification of a sequential consistent memory

Now we use this verification method for the verification of a particular distributed cache memory which has been presented in [ABM93] and verified using different methods in [JPR93, DGJ$^+$93]. The cache memory is a system of the form $P_1 \parallel\!| P_2 ... \parallel\!| P_n$ where each process P_i is defined as in Figure 3.1. the predicates *add*, *tail*, *first* and *empty* are as in the Example 1 and *update* is defined by

$update(m, (a, d), m') \equiv (m'[a] = d) \wedge (\forall b : address\ .\ (b \neq a \ \Rightarrow \ m'[b] = m[b])\ )$.

Variables :	Input :	$a : \mathit{address},\ d : \mathit{datum}$
	local :	$AD_i : \mathit{set}$ of $\mathit{address} \times \mathit{datum}_i$, (data already written)
		$\mathbf{C}_i : \mathit{array}[\mathit{address}]$ of $\mathit{datum} \cup \{\epsilon\}$ (local cache memory)
		$Out_i : \mathit{buffer}$ of $(\mathit{address} \times \mathit{datum}_i)$
	shared :	$\mathbf{M} : \mathit{array}[\mathit{address}]$ of datum (global memory)
		$In_k : \mathit{buffer}$ of $(\mathit{address} \times (\mathit{datum} \cup \{\epsilon\}))$, $k : \mathit{index}$
Transitions :		
$(write_i(a,d))$		$allowed((a,d), AD_i, AD'_i) \wedge add(Out_i, (a,d), Out'_i) \wedge \mathbf{unch}(\mathbf{C}_i, \mathbf{M}, In_1, ... In_n)$
$(read_i(a,d))$		$(\mathbf{C}_i[a] = d) \wedge empty(Out_i) \wedge empty(In_{i\,\|address \times datum_i}) \wedge \mathbf{unch}(\mathbf{C}_i, Out_i, AD_i, \mathbf{M}, In_1, ... In_n)$
$(mw_i(a,d))$		$first(Out_i, (a,d)) \wedge tail(Out_i, (a,d), Out'_i) \wedge update(\mathbf{M}, (a,d), \mathbf{M}') \wedge \forall k : index\ .\ add(In_k, (a,d), In'_k) \wedge \mathbf{unch}(\mathbf{C}_i, AD_i)$
$(cu_i(a,d))$		$first(In_i, (a,d)) \wedge tail(In_i, (a,d), In'_i) \wedge update(\mathbf{C}_i, (a,d), \mathbf{C}'_i) \wedge \mathbf{unch}(Out_i, AD_i, \mathbf{M}, \{In_j, j \neq i\})$
Init :		$(\forall b : address\ .\ (\mathbf{C}_i[b] = \mathbf{M}[b] = \epsilon)) \wedge empty(Out_i) \wedge empty(In_i)$

Fig. 1. A distributed cache memory system

The event $write_i(a,d)$ does not have any immediate effect neither on the local nor on the central memory, but pushes the pair (a,d) into the local buffer Out_i ; from there it is by the internal event $memory\_write_i(a,d)$ written into the central memory and dispatched into all buffers In_i; the internal event $cashupdate_i(a,d)$ takes the first element (a,d) out of In_i and writes datum d into address a of the local cache memory $\mathbf{C}_i$. The event $read_i(a,d)$ is possible only if $\mathbf{C}_i[a] = d$ and no local *write* event is pending, i. e., if $empty(Out_i) \wedge empty(In_{i\,|address \times datum_i})$. The only difference between our system and the one used in [JPR93] concerns the fact that each pair (a,d) can be the parameter of at most one event *write*. The way we obtain this, is by defining the type *datum* by $datum = \bigcup_i datum_i$, such that each process "signs" the data it writes, and by using in each process a variable AD_i of type *set* of $address \times datum_i$ which stores the information if the event $write_i(a,d)$ has already occurred or not, as in the example of the buffer.

The abstract specification that the system must verify is *sequential consistency* [Lam79], which originally is given in the form of an abstract program. In order to apply our method, we give the abstract specification in terms of a set of properties. Under the assumption that every pair of the form (a,d) can occur at most once as the parameter of some *write* event, sequential consistency can be characterized by the following set of properties expressed in terms of observable events:

Safety properties characterizing a sequential consistent memory:

(S1) $\forall (a,d),(a',d') : address \times datum, j, i : index$ such that $(a,d) \neq (a',d')$
$$init \Rightarrow \mathsf{A}(\ [\neg after(write_j(a,d))\mathcal{W} after(write_j(a',d'))\ ] \Rightarrow [\neg enable(read_i(a,d))\mathcal{W}(enable(read_i(a',d'))\vee AG(\neg enable(read_i(a',d'))))\ ]\ )$$

(S2) $\forall (a,d) : address \times datum, i : index$

$$init \;\Rightarrow\; \mathbf{A}(\neg enable(read_i(a,d))\mathcal{W}\textstyle\bigvee_{j:index}\; after(write_j(a,d))\;)$$

(S3) $\forall(a,d) : address \times datum, i : index$
$$init \;\Rightarrow\; \mathbf{AG}(after(write_i(a,d)) \;\Rightarrow$$
$$\mathbf{A}((enable(read_i(a)) \;\Rightarrow\; enable(read_i(a,d)))\mathcal{W}AG(\neg enable(read_i(a,d)))))$$

(S4) $\forall(a,d),(a,d') : address \times datum, i_1, i_2 : index$ such that $d \not\equiv d'$
$$init \;\Rightarrow\; \mathbf{A}(\;[\neg after(read_{i_1}(a,d))\mathcal{W}after(read_{i_1}(a,d'))\,] \;\Rightarrow$$
$$[\neg enable(read_{i_2}(a,d))\mathcal{W}AG(\neg enable(read_{i_2}(a,d')))\,]\;)$$

(S1) expresses that in every execution sequence the subsequence of $read_i$ events respects the order of $write_j$ events: whenever (a',d') is written before (a,d) by P_j, then $read_i(a,d)$ is not enabled before either $read_i(a',d')$ has already been enabled or is never enabled again (the second clause is necessary because $read_i(a',d')$ may never be enabled in some computation sequences). (S2) expresses that every $read_i(a,d)$ event is preceded (caused) by some $write_j(a,d)$ event. This is slightly stronger than sequential consistency which may allow $write_j(a,d)$ to occur after $read_i(a,d)$. (S3) expresses the fact that $read_i$ and $write_i$ events od the same process P_i must be consistent with a central memory, i. e., after a $write_i(a,d)$, P_i can read nothing different from d in address a until $read_i(a,d)$ is never enabled again. (S4) expresses analogously to (S1) the fact that *read* events concerning the same memory cell must be consistent in all processes. All these formulas can be translated into $\forall$CTL* formulas.

3.2 Verification of the cache memory

We verify each parameterized set of formulas on a different abstract program. Our aim is not to find the smallest abstract program that can be used for the verification of each formula, but we want to apply, whenever possible, the already predefined abstractions in order to show that the application of the method is simple and can be done systematically. The cache memory uses the data types and operations of the buffer of Example 1; it uses also a data type "memory"= *array*[*address*] of *datum*. As for buffers, we use three different types of abstractions of a variable X of type *memory* depending on the formula to be verified: we may

- completely forget about it (we do this always for the central memory **M**)
- keep information about a single pair (a,d) by taking an abstract boolean variable X_A and an abstraction relation $\varrho^1_{memory}(X, X_A) = X_A \equiv (X[\mathbf{a}] = \mathbf{d})$.
- keep information about two pairs $(\mathbf{a}_1, \mathbf{d}_1)$ and $(\mathbf{a}_2, \mathbf{d}_2)$ by taking two abstract boolean variables X^1_A and X^2_A and an analogous abstraction relation $\varrho^2_{memory}(X, X^1_A, X^2_A)$.

Suppose the type *elem* to be *address*×*datum* and take an abstract variable e_A of type $elem^1_A = \{0,1\}$ already used in the buffer example and the abstraction relation

$$\varrho^1_{elem}((a,d), e_A) = (e_A = 0) \wedge ((a,d) \neq (\mathbf{a},\mathbf{d})) \;\vee\; (e_A = 1) \wedge ((a,d) = (\mathbf{a},\mathbf{d})),$$

exactly as in Example 2; then, it is easy to define an abstract predicate $update^1_A$ by

$$update^1_A(X_A, e_A, X'_A) = (e_A = 0) \wedge (X'_A \;\Rightarrow\; X_A) \;\vee\; (e_A = 1) \wedge X'_A$$

expressing that if $(a,d) \neq (\mathbf{a},\mathbf{d})$, $X[\mathbf{a}] = \mathbf{d}$ is only possible in the next state if already in the present state $X[\mathbf{a}] = \mathbf{d}$, and if $(a,d) = (\mathbf{a},\mathbf{d})$, then in the next state $X[\mathbf{a}] = \mathbf{d}$, independently of the value of $X[\mathbf{a}]$ in the present state.

Using these definitions (and analogous ones with superscripts ^{ex} and ^{2}) and those already given in Example 1, the definition of appropriate abstract finite state programs of the cache memory becomes simple.

Abstract programs for property (S3): Each instance of property (S3) involves only events of a single process P_i. However, even if we succeed to verify it on P_i we can *not* deduce their satisfaction on the composed system. In fact, if we replace all processes different from P_i by the process *"Chaos"*, (S3) does not hold any more on the composed abstract program. We use here another approach to compositionality: by Proposition 1, we can abstract each P_j individually and build a global model by composing these small abstract programs. We choose the abstraction relation for all P_j with $j \neq i$ in such a way that shared variables are abstracted in the same way as in P_i and we forget about all local variables; this has as effect to avoid adding certain changes of shared variables which are not allowed by the concrete programs P_j.

Intuitively, (S3) expresses that as soon as $write_i$(a,d) has occurred, only **d** may be read by P_i on address **a** until **d** has been put into $\mathbf{C}_i$ and afterwards been replaced by some other value; that means that we are interested in observing what happens on the buffers Out_i and In_i and the cache $\mathbf{C}_i$. The actions (mw_j) should not disturb the behaviour of P_i observed by (S3) because they cannot push (a,d) into In_i. This leads naturally to the following abstraction relation for P_i:

$$\begin{array}{llll}
\multicolumn{4}{l}{\varrho_i^{S3}((a,d), AD_i, \mathbf{C}_i, Out_i, \mathbf{M}, In_1, ...In_n,\ e_A, E_A, \mathbf{C}_{iA}, Out_{iA}, In_{iA}) \ =} \\
\quad \varrho^1_{elem}((a,d), e_A) & \wedge & \varrho^1_{set\_of\_elem}(AD_i, E_A) & \wedge \\
\quad \varrho^1_{memory}(\mathbf{C}_i, \mathbf{C}_{iA}) & \wedge & \varrho^1_{buffer}(Out_i, Out_{iA}) & \wedge \\
\quad \varrho^1_{buffer}(In_i, In_{iA}) & & &
\end{array}$$

and for P_j, $j \neq i$ we use the same abstraction as in ϱ_i for the shared variables and forget about all local variables

$$\begin{array}{lll}
\multicolumn{3}{l}{\varrho_j^{S3}((a,d), AD_i, \mathbf{C}_i, Out_i, \mathbf{M}, In_1, ..., In_n,\ e_A, In_{iA}) \ =} \\
\quad \varrho^1_{elem}((a,d), e_A) & \wedge & \varrho^1_{buffer}(In_i, In_{iA})
\end{array}$$

from which we obtain by replacing concrete by corresponding abstract predicates as defined before, the following abstract program P_i^A for index i,

Variables :	<u>abstract input</u> :	e_A : *Bool*
	<u>local</u> :	$E_A, \mathbf{C}_{iA}$: *Bool*
		Out_{iA} : $buffer^1_A$
	<u>shared</u> :	In_{iA} : $buffer^1_A$
Transitions :		
$(write_i(e_A))$	$allowed^1_A(e_A, E_A, E_A') \wedge add^1_A(Out_{iA}, e_A, Out'_{iA}) \wedge \mathbf{unch}(\mathbf{C}_{iA}, In_{iA}) \wedge$	
$(read_i(e_A))$	$(e_A \Rightarrow \mathbf{C}_{iA}) \wedge empty^1_A(Out_{iA}) \wedge empty^1_A(In_{iA}) \wedge$ $\mathbf{unch}(E_A, \mathbf{C}_{iA}, Out_{iA}, In_{iA})$	
$(mw_i(e_A))$	$first^1_A(Out_{iA}, e_A) \wedge tail^1_A(Out_{iA}, e_A, Out'_{iA}) \wedge$ $add^1_A(In_{iA}, e_A, In'_{iA}) \wedge \mathbf{unch}(\mathbf{C}_{iA}, E_A)$	
$(cu_i(e_A))$	$first^1_A(In_{iA}, e_A) \wedge tail^1_A(In_{iA}, e_A, In'_{iA}) \wedge update^1_A(\mathbf{C}_{iA}, e_A, \mathbf{C}'_{iA}) \wedge$ $\mathbf{unch}(E_A, Out_{iA})$	
Init :	$\neg \mathbf{C}_{iA} \wedge empty^1_A(Out_{iA}) \wedge empty^1_A(In_{iA})$	

and P_j^A for all indices different from i,

Variables :	abstract input : e_A : *Bool*	
	shared: $In_{iA} : buffer^1_A$	
Transitions :	$(write_j(e_A), read_j(e_A), cu_j(e_A))$	**unch**(In_{iA})
	$(mw_j(e_A))$	$first^{ex}_A(e_A) \wedge add^1_A(In_{iA}, e_A, In'_{iA})$
Init :	$empty^1_A(In_{iA})$	

in which we have already eliminated all abstract operations that are always *true*, such as add^{ex}_A, $update^{ex}_A$,.... Notice that the event $(mw_j(true))$ is never executed as $first^{ex}_A(true) = false$ because the buffer Out_j cannot contain the pair (a,d) as d $\in$ $datum_i$. Notice also that the size of the composed system $P^A_1 \parallel\!\!| \ldots \parallel\!\!| P^A_i \parallel\!\!| \ldots \parallel\!\!| P^A_n$ is the same, whatever the number of composed programs is, as for all $j \neq i$, the programs P^A_j are identical and $P \parallel\!\!| P$ and P represent the same transition system.

Abstract programs for property (S2): Property (S2) expresses the fact that any event $read_i$(a,d) is preceded by an event $write_j$(a,d) for some j. Thus, we are interested in observing the buffers Out_j and In_i and the cache C_i. This leads to similar abstraction relations as for the verification of (S3), except that we do not need the unicity of the *write* events and can forget about AD_i but we need abstract buffers Out_{jA} for all j. Thus, all abstraction relations ϱ^{S2}_j are the same:

$$\begin{aligned}\varrho^{S2}_j(A, D, AD_j, C_j, Out_j, M, In_1, \ldots, In_n,\ e_A, Out_{jA}, In_{iA}) \ &= \\ \varrho^1_{elem}((A, D), e_A) \quad \wedge \quad \varrho^1_{buffer}(Out_j, Out_{jA}) \quad &\wedge \\ \varrho^1_{buffer}(In_i, In_{iA})\end{aligned}$$

For this abstraction, the size of the obtained global abstract transition system does depend on the number n of processes as we have defined n abstract variables Out_{jA}. In order to obtain an abstract transition system such that its size is independent of n, we can define — instead of the set of local abstract buffers Out_{jA} — a *single global* abstract buffer Out_A defined by a relation of the form

$$\varrho^{1,glob}_{buffer}(\bigcup_{j:index} Out_j,\ Out_A)$$

which obliges however to redefine the abstract operations add_A, $tail_A$,...

Abstract programs for properties (S1) and (S4): For the verification of (S1) we need to observe events with two different parameters (a_1, d_1) and (a_2, d_2), such that $d_1, d_2 \in datum_j$; thus, we use the abstraction relations with superscript ^{2} as for the verification of order preservation in the preceding section. We define abstract variables E^1_A, E^2_A (in P^A_j) in order to guarantee uniqueness of the observed $write_j$ events, Out_{jA} (in P^A_j), C_{i_1}, C_{i_2} (in P^A_i) and a global variable In_{iA} and use the predefined abstraction relations and corresponding abstract operations.

The resulting global abstract transition system is again independent of the number of process as all the abstract programs with indices different from i, j are identical. We need only to consider the case where the indices i and the j are different, as the property for $i = j$ is implied by (S3).

Property (S4) expresses that the sequences of *read* events of any two processes P_{i_1} and P_{i_2} on the same address a are compatible, also when they have been written by two different processes P_{j_1} and P_{j_2}. Thus, for its verification we observe two pairs (a_1, d_1) and

$(\mathbf{a}_2, \mathbf{d}_2)$ such that $\mathbf{a}_1 = \mathbf{a}_2 = \mathbf{a}$ and $\mathbf{d}_1 \in datum_{j_1}$ and $\mathbf{d}_2 \in datum_{j_2}$. Consequently, we need abstract variables E^1_A, E^2_A (in $P^A_{j_1}$ respectively $P^A_{j_2}$), Out_{j_1A}, Out_{j_2A} (in $P^A_{j_1}$ respectively $P^A_{j_2}$), $\mathrm{C}_{i_11}, \mathrm{C}_{i_12}$ (in $P^A_{i_1}$), $\mathrm{C}_{i_21}, \mathrm{C}_{i_22}$ (in $P^A_{i_2}$) and global variables In_{i_1A} and In_{i_2A}. Here, we have to consider different cases, those where the indices i_1 and j_1 (respectively i_2 and j_2) coincide and those where not.

Now, the verification of the distributed cache memory is almost terminated. The actual construction of global abstract transition systems and the verification of the formulas on them could been done automatically by our tool [GL93b, Loi94]. By Proposition 5, it remains to verify the consistency of the atomic propositions with the used abstraction relations. Properties (S2) and (S4) pose no problem, as in the corresponding $\forall$CTL* formulas only predicates of the form $after(\ell)$ occur non negated. For (S3), in principle the consistency of $enable(read_i(\mathbf{a},\mathbf{d}))$ is required; however, it is used only within the predicate $(enable(read_i(\mathbf{a}))\Rightarrow enable(read_i(\mathbf{a},\mathbf{d}))$ which is equivalent to $\mathrm{C}_i[\mathbf{a}] = \mathbf{d}$ and consistent with the abstraction relation used for (S3). For (S1), it is slightly more complicated to show that the consistency of $enable(read_{i_2}(\mathbf{a}_2, \mathbf{d}_2))$ with the considered abstraction relation ϱ is not needed. The predicate obtained by translating $enable(read_{i_2}(\mathbf{a}_2, \mathbf{d}_2))$ forth and back by ϱ is $\mathrm{C}_{i_2}[\mathbf{a}_2] = \mathbf{d}_2 \wedge \ldots$ The property obtained from (S1) by replacing $enable(read_{i_2}(\mathbf{a}_2, \mathbf{d}_2))$ by this weaker predicate, implies nevertheless (S1) for our particular system, because $enable(read_{i_2}(\mathbf{a}_1)\Rightarrow((\mathrm{C}_i[\mathbf{a}_2] = \mathbf{d}_2)\Rightarrow enable(read_{i_2}(\mathbf{a}_2, \mathbf{d}_2)))$ holds. Notice also, that this additional condition is in fact necessary in order to obtain sequential consistency of the given system.

4 Discussion

What have we achieved? A first impression could be that our verification of a cache memory looks much like any other handwritten proof. However, it is quite different: starting right from the beginning, it is in fact rather lengthy to define all the abstraction relations and corresponding abstract predicates, even in order to verify some trivial buffer program. However, having done this once, in order to verify the much more complex cache memory system, we only need a few more definitions obtained a long the same line as the already given ones. In fact, there are many examples of systems, for which we have to verify exactly the same type of properties and which use analogous data structures and operations on them, such that the same abstract domains and operations can be used. Thus, we could build a "library" of useful abstract domains and operations in which new definitions can be added when necessary. A similar approach has been followed by P. and R. Cousot and D. Long concerning "standard" abstractions of integers and operations on them.

The fact that for the verification of an individual property a large part of the system can be abstracted existentially is often necessary in order to obtain tractable global models. If the system is too large or the property is "too global" one can often get results by decomposing the property, as this has been proposed, e. g. in [Kur89].

It can also been observed that our verification method is incremental: it is obviously incremental with respect to changes in the abstract specification, like every method based on the fact that abstract specifications are expressed by a set of properties. But also certain changes of the program allow to use the same or at least very similar abstraction relations and abstract operations. That means that exactly the time consuming and difficult part of the verification process need not to be redone. In the case that the obtained abstract program is not identical to the previous one, the reconstruction of a model and the verification of the properties on it by means of some model checker poses no problem.

An important point for the use of our method in practice, is the formalisms used for the description of programs. We either need a formalism allowing to express nondeterminism or we have to use more complex abstract domains allowing to represent certain sets of classes of concrete values. For example, LOTOS is a specification formalism for which this method can be applied: all the data types and operations on them are specified separately from the control part by means of some "abstract data type" language. This means that, by coding whatever should be abstracted in the data part, to construct an abstract program consists simply in replacing the original data type definitions by simpler ones, whereas the control part of the program remains completely unchanged.

Acknowledgements: I would like to thank the referees who pointed out that my initial characterization of sequential consistency was not sufficient.

References

[ABM93] Y. Afek, G. Brown, and M. Meritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1), 1993.

[CC77] P. and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[CGL91] E. Clarke, O. Grumberg, D. Long. Model checking and abstraction. In *POPL*, 1991.

[DGJ+93] J. Davis, R. Gerth, W. Jannsen, B. Jonsson, S. Katz, G. Lowe, A. Pnueli, and C. Rump. Verifying sequentially consistent memory. Preliminary report, 1993.

[EH83] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: On branching versus linear time. In *POPL*, 1983. also in Journal of ACM, 33:151-178.

[GL93a] S. Graf and C. Loiseaux. Program verification using compositional abstraction. In *TAPSOFT 93, joint conference CAAP/FASE.* LNCS 668, Springer Verlag, April 1993.

[GL93b] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *CAV, Heraklion Crete.* LNCS 697, Springer Verlag, 1993.

[JPR93] B. Jonsson, A. Pnueli, and C. Rump. Proving refinement using transduction. Technical report, Weizmann Institute, 1993.

[Kur89] R.P. Kurshan. Analysis of discrete event coordination. In *REX Workshop on Stepwise Refinement of Distributed Systems, Mook.* LNCS 430, Springer Verlag, 1989.

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690-691, 1979.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, 1991.

[LGS+92] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *To appear in Formal Methods in System Design*, also in *CAV'92.*

[Loi94] C. Loiseaux. Vérification symbolique de programmes réactifs à l'aide d'abstractions. Thesis, Verimag, Grenoble, January 1994.

[Lon93] D. E. Long. Model checking, abstraction and compositional verification. Phd thesis, Carnegie Mellon University, July 1993.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models for Concurrent Systems.* NATO, ASI Series F, Vol.13, 1985.

[Pnu86] A. Pnueli. Specification and Development of reactive Systems. In *Conference IFIP, Dublin.* North-Holland, 1986.

[SG90] G. Shurek and O. Grumberg. The modular framework of computer-aided verification: motivation, solutions and evaluation criteria. In *CAV*, LNCS 531, 1990.