Characterization of a Sequentially Consistent Memory and Verification of a Cache Memory by Abstraction[*]

Susanne Graf

VERIMAG[**], Avenue de la Vignate, F-38610 Gières[***]

Abstract. The contribution of the paper is two-fold. We give a set of properties expressible as temporal logic formulas such that any system satisfying them is a sequentially consistent memory, and which is sufficiently precise such that every reasonable concrete system that implements a sequentially consistent memory satisfies these properties.
Then, we verify these properties on a distributed cache memory system by means of a verification method, based on the use of abstract interpretation which has been presented in previous papers and so far applied to finite state systems. The motivation for this paper was to show that it can also be successfully applied to systems with an infinite state space.
This is a revised and extended version of [Gra94].

1 Introduction

We propose to verify the distributed cache memory presented in [ABM93] and [Ger94] by using the verification method proposed in [BBLS92,LGS$^+$94,CGL94,Lon93]. This method, based on the principle of abstract interpretation [CC77], proposes to verify a set of $\forall$CTL* [SG90] formulas on a composed program as follows: define an appropriate abstract program, obtained compositionally from the given program, and verify the required properties on it. Our way of computing abstract programs is similar to that proposed in [CGL94,Lon93,Cri95], but

- in the opposite to most other approaches, our approach allows to deal with arbitrary data types,
- our abstractions are harder to obtain, but possibly much more precise as the "standard" abstractions proposed in [Cri95] as they are property oriented,
- 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 usually compositionality consists in deducing properties of the composed system from properties of its components under some hypotheses on its environment — which must

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

[**] Verimag is a Research Laboratory affiliated to CNRS, Université Joseph Fourier and Institut Nationale Polytechnique of Grenoble

[***] e-mail: Susanne.Graf@imag.fr, URL: http://www-verimag.imag.fr/PEOPLE/Susanne.Graf

be proven to hold. The abstractions of components not directly involved in the property play the same role as the hypotheses on the environment, except that their correctness is obtained "by construction". 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 [GL93,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 for infinite state systems. 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 abstract program can be fully automatized.

Despite the fact that $\forall$CTL* contains also *liveness* properties, this method does in general not support directly the verification of liveness properties as they often do not hold on finite abstractions. Here, we succeed to verify the liveness property of the cache memory by applying variants of the induction rules given in [Pnu85,JPR94] which allow under some fairness assumptions to reduce a liveness property to a set of safety properties.

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],
- a method to compute abstract programs, which consists in defining for each operator occurring in the program a corresponding *abstract operator*,
- 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 which include 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 give a set of temporal logic formulas guaranteeing that, whenever a system satisfies all these properties, it is a "sequentially consistent memory" [Lam79]. This set of properties has been chosen in such a way that most reasonable implementations of sequentially consistent memories will satisfy it. In Section 4, we verify this set of properties on the distributed cache memory system. It turns out that, using our method, this verification is almost as simple as the verification of the tiny buffer, as we can use almost the same abstract types and corresponding operations, and this is the most time consuming part of the verification process.

2 A verification method using abstraction

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 system is a parallel composition of basic programs of the form

Name :	P
Variables :	$x_1 : T_{x_1}, ..., x_n : T_{x_n}$
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 P is an identifier used to refer to the program in a composition expression, x_i are variables of type T_{x_i} — defining its set of possible valuations — and $L_P = \{\ell_1, ..., \ell_p\}$ is a set of program labels. Each $action_i$ is an expression with free variables in the set of program variables and the corresponding set of primed variables which for each state variable x contains a variable x' with the same type as x. As e.g. in [Pnu86,Lam94], $action_i$ represents a transition relation on the set of valuations 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. For any set of variables $Y = \{y_1, ..., y_k\} \subseteq X$, we denote its set of valuations by $T_Y = T_{y_1} \times ... \times T_{y_k}$.

Semantics: A program P defines a transition system $S_P = (Q_P, R_P)$ where

- $Q_P = T_{X_P}$ 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: they are in general of the form $Init \Rightarrow \phi$ — where ϕ is a temporal logic formula — as we are only interested in properties of reachable states.

Variables representing inputs need not to be distinguished as they are not treated in a particular manner. However, we indicate in programs the variables which are meant as inputs as this makes them easier to read. We also indicate sometimes which variables are shared with other programs and which ones are used only locally, even if in the model, for simplicity, all variables are interpreted as globally.

Labels are used to name "events" or "actions". If ℓ_i is the label of $action_i$ and (v, v') a pair of valuations such that $action_i(v, v')$ evaluates to true, then the transition from state v to state v' is called an event ℓ_i. 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 type $elem$. 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.

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

At any moment the value of variable $\mathbf{E}$ contains the elements $e \in elem$ such that $push(e)$ has already occurred before, and for all elements e and sets E and E', $allowed(e, E, E')$ is true if $e \notin E$ and $E' = E \cup \{e\}$. This guarantees that the event $push(e)$ can occur at most once in every execution sequence. All other predicates have the intuitive meanings: $append(B, e, B')$ holds if the buffer B' is obtained by appending element e at the end of the buffer B; $tail(B, e, B')$ holds if B' is obtained by eliminating e from B if e is its first element ($first(B, e)$ holds); $empty(B)$ is true if B is the empty buffer. $unch(Y)$, where $Y = (y_1, ...y_k)$ is a tuple of program variables is a shorthand for $\bigwedge_{i=1}^{k}(y_i' = y_i)$, that means it holds if all variables in Y have the same value in the actual and in the next state.

We use predicates of the form $append(\mathbf{B}, \mathbf{e}, \mathbf{B}')$ instead of $\mathbf{B}' = Append(\mathbf{B}, \mathbf{e})$ where *Append* is a function, as abstract operations are in general non deterministic. This is the same way of representing operations which has been proposed in [MP91,CGL94,Lam94].

Composed programs: In [GL93] results for more general parallel composition operators are given, but here we need only composition obtained by interleaving of the actions of the composed programs. If P_1 and P_2 are programs defined on a tuple of state variables X_1, respectively X_2, then $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2 defining the transition system $S = (T_{X_1 \cup X_2}, R)$ where

$$R \;=\; R_{P_1} \land unch(X_2 - X_1) \quad \lor \quad R_{P_2} \land unch(X_1 - X_2)$$

Each transition of $P_1 \parallel P_2$ is either a transition of P_1 which leaves all variables not declared in P_1 unchanged or a transition of P_2 which leaves all variables not declared in P_2 unchanged.

2.2 Abstract programs

Let P be a program on the set of variables X, and let X_A be a set of abstract variables, defining a set of abstract states T_{X_A}. Then, a relation $\varrho \subseteq T_X \times T_{X_A}$ which is total on T_X, is called an abstraction relation (from T_X to T_{X_A}). For the ease of expression of properties, we suppose that ϱ is represented by a predicate on concrete and abstract variables denoted $\varrho(X, X_A)$. Furthermore, the abstraction relations we use in practice, are often (total) functions $\varrho : T_X \mapsto T_{X_A}$. In this case, we denote for $v \in T_X$ by $\varrho(v)$ the (unique) value $v_A \in T_{X_A}$ such that $\varrho(v, v_A)$ holds.

Definition 1. *(Abstract programs) Let, in addition to the above conventions, P_A be a program defined on X_A. P_A is an* abstraction*, or more precisely a ϱ-*abstraction *of P, if for every action act of P, there exists an action act_A of P_A (with the same label), such that*

$$\begin{array}{ll} \forall v, v' \in T_X \;.\; act(v, v') \;\;\Rightarrow\;\; \exists v_A, v'_A \in T_{X_A} \;.\; \varrho(v, v_A) \wedge \varrho(v', v'_A) \wedge act_A(v_A, v'_A) & \\ \textit{and} & (1) \\ \forall v \in T_X \;.\; init(v) \;\;\Rightarrow\;\; \exists v_A \in T_{X_A} \;.\; \varrho(v, v_A) \wedge init_A(v_A). & \end{array}$$

This condition ensures that ϱ defines a simulation in the sense of [Mil71] between the transition systems associated with the concrete and the abstract program.

Remark: Obviously, it is in principle sufficient, that the above conditions hold only in those states $v \in T_X$ which are reachable, which means that whenever one has a known invariant of the system, the conditions need only be checked on the states satisfying this invariant. For states outside this invariant, nothing is required. If one has a "desired" invariant (a property to be proved) of the system, a usual method consists in considering a trivial set of successors (such as *true*) of the states outside this desired invariant. This simplifies the definition and respects the conditions above.

For the verification of programs composed of several parallel components, it is interesting to compute an abstract program compositionally, i. e., as a parallel composition of abstract component programs. From a more general result [LGS+94], we deduce the following, sufficient for the verification of the distributed cache memory system.

Proposition 1. *(compositionality of abstraction) Let $\varrho^i(X^i, X^i_A)$ be abstraction functions represented by predicates on concrete and abstract variables. Let the program P^i_A be a ϱ^i-abstraction of P^i for $i \in \{1, 2\}$. If the predicate $\varrho^1 \wedge \varrho^2$ represents a total function $\varrho : T_{X^1 \cup X^2} \mapsto T_{X^1_A \cup X^2_A}$, then $P^1_A \parallel P^2_A$ is a ϱ-abstraction of $P^1 \parallel P^2$.*

Computation of abstract programs in practice: The idea of abstract interpretation [CC77] is to interpret every *function* on concrete values used in the program by a corresponding *abstract function* on the abstract values, and then to analyze the so obtained simpler abstract model instead of the concrete one.
Consider the program $Prog_A$ obtained by replacing every basic predicate *op* (such as *tail*, *first*,...) on the concrete variables by a predicate op_A on abstract variables X_A satisfying (1). If the expressions in *Prog* are negation free (as it is the case in the lossy buffer), then $Prog_A$ is in fact a ϱ-abstraction of *Prog*.

Our intention is to define for any predicate *op* — depending in general only on a small subset of the concrete variables — an abstract predicate on the "corresponding", hopefully also small, set of abstract variables. Also, in order to be able to verify interesting properties on $Prog_A$, the abstract predicate should be "reasonably close" to the "optimal" abstract predicate which is defined by the requirement

$$\forall v_A, v'_A \in T_{X_A} \;.\; (\; op_A(v_A, v'_A) \;\;\Rightarrow\;\; \exists v, v' \in T_X \;.\; \varrho(v, v_A) \wedge \varrho(v', v'_A) \wedge op(v, v') \;) \quad (2)$$

This approach makes no sense for arbitrary abstraction relations. We are interested in abstraction relations relating each variable of type T_x to a single abstract variable of type T^A_x, such that, e.g. each occurrence of expression $first(\mathbf{e}, \mathbf{B})$ in the concrete program can be replaced by an expression of the form $first_A(\mathbf{e}_A, \mathbf{B}_A)$ in the abstract program where $\mathbf{e}_A$ is a variable of some type "abstract element", $\mathbf{B}_A$ a variable of some type "abstract buffer" and $first_A$ is a predicate satisfying condition (1). That means, given a set of abstract variables, we are interested in abstraction functions such that

$$\forall (v_1, ..., v_n) \in T_X \ . \ \varrho(v) = (\varrho^1(v_{k_1}), ..., \varrho^p(v_{k_p}))$$

where p is the cardinality of X_A and the indices k_i are all different. That means that if $n = p$, every abstract variable is related to exactly one concrete variable. Otherwise, i.e., if $p < n$, there exist concrete variables related to no abstract variable; these variables are called *existentially abstracted* variables. For the verification of the cache memory, we use also an abstraction function mapping the values of a pair of variables (**a**,**d**) onto the value of a single variable **e**, but in this case, the two variables represented by a single abstract one are such that (almost) all predicates depend on both or on none of them.

The use of such an abstraction function allows to construct an abstract program in a very simple way: Each variable $x : T_x$ of the concrete program is either eliminated (existentially abstracted) or it is declared of an abstract type T_x^A instead of the concrete type T_x. Then, each basic predicate $op^{T_x, T_y, ...}(x, y, ...)$ is replaced by a predicate $op_A^{T_x^A, T_y^A, ...}(x, y, ...)$ depending on the same variables as the concrete predicate except the existentially abstracted ones. All these predicates must satisfy condition (1) which is simplified as it depends only on the (few) concrete variables occurring in predicate *op* and the corresponding abstract variables.

The guess of appropriate abstract types and the definition of abstract predicates is the only part of our verification method which in general cannot be automatized. The abstract predicates defined by condition (1) or (2) make reference to existentially quantified concrete variables. These quantifiers must be eliminated in order to explicitly construct the finite abstract model. If all concrete types are finite, this can always be done automatically [Loi94,Lon93].

Notice also, that for a *given* "guess" of an abstract predicate, the verification of condition (1) is often easy if ϱ is a function.

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 or constant parameters, and "buffers" with the usual operations *append*, *tail*, $first$,.. as in our example. As furthermore the properties to be verified are often similar, for the verification of many algorithms similar abstract types and corresponding operations (with adaptations to each particular case) may be used.

Example 2. : An abstract lossy buffer To illustrate the idea, consider again the lossy buffer of Example 1. In order to show that it has the property of "order preservation" (see Example 3), it is sufficient to show order preservation for any two of elements $e_1, e_2 \in elem$. To show order preservation for e_1, e_2, all the information we need about the value of the variable **B** is, if and in which order, it contains the elements e_1 and e_2. 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. In general, all the abstract types for the type *elem* that we need, distinguish n particular elements of *elem* and merge all others in a single abstract value. Suppose that we want to distinguish the elements $e\_obs = \{e_1, ..., e_n\} \subseteq elem$, we use as abstract type for variables of type *elem*:

$$elem_A = \{0\} \cup abs\_elem \ \text{ where } abs\_elem = \{1, ..., n\}$$

where the concrete type *elem* and the abstract type $elem_A$ are related by

$$\forall e \in elem \ . \ \varrho_{elem}^{e\_obs}(e) = \begin{cases} 0 & \text{if } e \notin e\_obs \\ i & \text{if } e = e_i \ (\in e\_obs) \end{cases}$$

We denote by $\varrho^{e\_obs}_{elem}$ also the pointwise extension of this function to a function from sets, sequences, ... containing concrete elements, to sets, sequences, ... where the concrete elements are replaced by the corresponding abstract ones.

The choice of the abstract type $elem_A$, determines the abstract types used for sets and buffers in an obvious way: as abstract type for variables of type "*set* of *elem*" we use the type "*set* of *abs_elem*", where the concrete and the abstract types are related by

$$\forall E \in set \text{ of } elem \ . \ \varrho^{e\_obs}_{set}(E) = \{\varrho^{e\_obs}_{elem}(e) \mid e \in E\} \cap abs\_elem$$

We use abstract sets subsets of *abs_elem* and not of $elem_A$ as the property to be verified depends only on information concerning distinguished elements. Finally, we use as abstract type for variables of type $buffer$ of *elem*

$$buffer^K_A \text{ of } elem_A = (sequences^K \text{ of } abs\_elem) \cup \{\bot\}$$

consisting of the sequences of abstract elements of length less or equal to some constant K which has to be chosen depending on the program under study. The element $\bot$ represents all buffers such that their restriction to elements in *e_obs* — denoted $B_{|e\_obs}$ — is of length greater than K. We denote the empty sequence by ϵ and the concatenation symbol of sequences by $\bullet$. Concrete and abstract buffers are related by

$$\forall B \in buffer \text{ of } elem \ . \ \varrho^{e\_obs,K}_{buffer}(B) = \begin{cases} \epsilon & \text{if } length(B_{|e\_obs}) = 0 \\ \varrho^{e\_obs}_{elem}(B_{|e\_obs}) & \text{if } 1 \leq length(B_{|e\_obs}) \leq K \\ \bot & \text{if } length(B_{|e\_obs}) > K \end{cases}$$

Thus, the abstract buffer $1 \bullet 2$ represents all concrete buffers containing any number of non-distinguished elements and the distinguished elements e_1, e_2, exactly once, and e_2 before e_1.

It remains to define abstract predicates for all the basic predicates used in the concrete buffer program, such as *allowed*, *append*, *tail*, *unch*, depending on the abstract types chosen for their parameters. The following abstract predicates satisfy condition (1). The proofs are omitted, but they are simple.

The abstract predicate associated with *unch* is obviously *unch* itself (where all the existentially abstracted variables are omitted). For the other predicates occurring in the lossy buffer, we have,

$$\forall e_A \in elem_A \ \ \forall E_A, E_A{}' \in set \text{ of } abs\_elem \ \ \forall B_A, B'_A \in buffer^K \text{ of } abs\_elem:$$

$$allowed_A(e_A, E_A, E_A{}') = (e_A = 0) \wedge (E_A{}' \equiv E_A) \ \vee \ (e_A \neq 0) \wedge (e_A \notin E_A) \wedge (e_A \in E_A{}')$$

$$\begin{aligned} append^K_A(B_A, e_A, B'_A) = {} & (e_A = 0) \wedge \ (B_A = B'_A) \vee \\ & (e_A \neq 0) \wedge ((length(B_A) < K) \wedge (B'_A = e_A \bullet B_A) \vee \\ & \qquad (length(B_A) = K) \wedge (B'_A = \bot) \ ) \end{aligned}$$

$$tail_A(B_A, e_A, B'_A) = (B_A = \bot) \ \vee \ (e_A = 0) \wedge (B_A = B'_A) \ \vee \ (e_A \neq 0) \wedge (B_A = B'_A \bullet e_A)$$

$$empty_A(B_A) = (B_A = \epsilon)$$

$$first_A(B_A, e_A) = (B_A = \bot) \vee \ (e_A = 0) \ \vee \ (e_A \neq 0) \wedge \exists B'_A \ . \ (B_A = B'_A \bullet e_A)$$

The concrete predicate *tail* defines a function, whereas the corresponding abstract predicate cannot be a function on the given abstract domain: $tail_A(\perp, 1, B'_A)$ must, in order to satisfy condition (1), hold for $B'_A = \perp$ and for all values of B'_A with length equal K, as after the concrete tail operation on a concrete buffer related with $\perp$ the obtained concrete buffer may contain K or more than K elements in e_obs. Here, we have chosen an approximation allowing any value for B'_A. According to the remark after Defintion 1, this is is a reasonable choice as the value of variable **B** should never become $\perp$. All other abstract predicates are optimal in the sense of condition (2) with respect to the chosen abstraction function ϱ.

Using a type $abs\_elem = \{1, 2\}$ and $elem_A = \{0\} \cup abs\_elem$ and all the above definitions, the program

```
Name :        Abstract lossy buffer
Variables :   $\mathbf{e} : elem_A$   (input)
              $\mathbf{E} : set$ of $abs\_elem$
              $\mathbf{B} : buffer^2$ of $abs\_elem$
Transitions:  $(push(\mathbf{e}))$  $allowed_A(\mathbf{e}, \mathbf{E}, \mathbf{E}') \wedge (append^2_A(\mathbf{B}, \mathbf{e}, \mathbf{B}') \vee unch(\mathbf{B}))$
              $(pop(\mathbf{e}))$   $first^2_A(\mathbf{B}, \mathbf{e}) \wedge tail_A(\mathbf{B}, \mathbf{e}, \mathbf{B}') \wedge\ unch(\mathbf{E})$
Init :                  $empty_A(\mathbf{B})$
```

represents a ϱ-abstraction of the lossy buffer where

$$\forall e \in elem_A \ \forall E \in set \text{ of } abs\_elem \ \forall b \in buffer^2 \text{ of } abs\_elem$$

$$\varrho(e, E, B) = (\varrho^{e\_obs}_{elem}(e), \varrho^{e\_obs}_{set}(E), \varrho^{e\_obs,2}_{buffer}(B))$$

This abstract program represents a transition system with at most 60 states on which any property can easily be verified.

The abstraction for the choice $e\_obs = \emptyset$, i.e., $elem_A = \{0\}$ defines an existential abstraction for all variables, and all the abstract predicates $append^K_A, \ldots$ are equivalent to *true* as $e_A \neq 0$ can never hold; the corresponding abstract program is the program "Chaos" which can produce any event at any moment and which is obviously not very interesting for the verification of properties. The abstraction function defined by

$$\varrho(e, E, B) = (\varrho^{e\_obs}_{elem}(e), \varrho^{e\_obs}_{set}(E))$$

for e_obs containing at least one element, defines an existential abstraction only for the variable **B**. The abstract program obtained for this abstraction function has no variable **B**; the corresponding abstract predicate $append^{ex}_A$ — which has a single parameter of type $elem_A$ — is equivalent to *true*, and analogously for the other predicates having variable **B** as parameter. This abstract program can be used to verify that for $e \in e\_obs$, the action $push(e)$ can be executed at most once in any execution sequence.

In [CGL94] a similar method is proposed and in [Lon93] particular abstraction schemes are proposed for bounded integers and operations on them.

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+94] are given for subsets of the more powerful branching time μ-calculus [Koz83] augmented by past time modalities. μ-calculus and CTL* can express both branching time and linear time properties; μ-calculus by using nested fixed points and CTL* by using explicitly state *and* path formulas. Our tool presented in [GL93,Loi94] only deals with state formulas; however formulas with nested fixed points are in general not very intuitive, so we prefer here for readability reasons to stick to CTL* even if it is less expressive.

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 $\mathbf{X}\phi$, $\phi\mathbf{U}\psi$ and $\phi\mathbf{W}\psi$ are path formulas.*

U is a "strong until" and **W** a "weak until" operator, a sequence satisfies $\phi\mathbf{W}\psi$ if ϕ holds as long no state satisfying ψ has been encountered, and $\phi\mathbf{U}\psi$ expresses the same property and moreover the obligation that such a state satisfying ψ exists. That means that **U** and **W** are related as follows: $\phi\mathbf{W}\psi = \neg(\neg\psi\mathbf{U}\neg(\phi \vee \psi))$ and, as usual, we use also the abbreviations $\phi_1 \Rightarrow \phi_2$ denoting implication, $\mathbf{F}\phi$ denoting $\mathit{true}\mathbf{U}\phi$ (expressing "eventually" ϕ) and $\mathbf{G}\phi$ denoting $\phi\mathbf{W}\mathit{false}$ (expressing "always" ϕ).

CTL is the subset of CTL* obtained by allowing in all rules only the choice (a) whereas PTL is the subset obtained by allowing only the choice (b) and restricting Rule 3 by allowing only the path quantifier **A**. $\forall$CTL and $\forall$CTL* [SG90] are the subsets of CTL respectively CTL* obtained by allowing negations only on atomic formulas and restricting Rule 3 by allowing only the universal path quantifier **A**; that means that PTL is contained in $\forall$CTL*.

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 $\mathcal{I} : \mathcal{P} \mapsto 2^Q$ is a function interpreting the propositional variables of $\mathcal{P}$ as sets of states of S.

Definition 3. *A* path *in a transition system $S = (Q, R)$ is an infinite sequence of states $\pi = q_1 q_2 ...$ such that for every $n \geq 1$, $R(q_n, q_{n+1})$. We denote by π_n the nth state of path π and by π^n the suffix of π starting in π_n.*

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

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

We say that $M \models \phi$ if and only if $q \models_M \phi$ for all states of M.

From the more general results given in [LGS$^+$94] we obtain the following proposition concerning preservation of properties of $\forall$CTL*. This proposition uses the notion of consistency that we define first.

Definition 5. *Let* $M = (S, \mathcal{I})$ *be a Kripke structure, and* $\varrho \subseteq Q \times Q_A$ *an abstraction relation, where* Q_A *is some abstract set of states. We say that* ϱ *is* consistent *with* $\mathcal{I}$ *for a set of propositional variables* $\mathcal{P}' \subseteq \mathcal{P}$ *if*

$$\forall p \in \mathcal{P}' \,.\, (Im[\varrho^{-1}] \circ Im[\varrho] \circ \mathcal{I})(p) \subseteq \mathcal{I}(p)$$

where $Im[\varrho] : 2^Q \mapsto 2^{Q_A}$ is the image function of ϱ, associating with any set of concrete states the set of abstract states related via ϱ with one of its elements. Consistency expresses the fact that for any atomic proposition $p \in \mathcal{P}'$, the set of abstract states $Im[\varrho](\mathcal{I}(p))$ represents no state in $\overline{\mathcal{I}(p)}$, which means in particular that $Im[\varrho](\mathcal{I}(p))$ and $\overline{Im[\varrho](\overline{\mathcal{I}(p)})}$, used as abstract interpretations of the formulas p, respectively $\neg p$, have an empty intersection.

Proposition 2. *(Preservation of* $\forall$CTL**)*
Let Prog be a program, ϱ *an abstraction relation from the set of states of Prog into some abstract set of states, and* $Prog_A$ *a* ϱ*-abstraction of Prog. Let be* $\phi \in \forall$CTL*, $\mathcal{P}$ *the set of atomic propositions in* ϕ *and* $\mathcal{I}$ *an interpretation function mapping* $\mathcal{P}$ *into sets of states of* S_{Prog}. *If* ϱ *is consistent with* $\mathcal{I}$ *for the set of propositions in* $\mathcal{P}$ *occurring non negated in* ϕ, *then*

$$(S_{Prog_A}, Im[\varrho] \circ \mathcal{I}) \models \phi \quad \Rightarrow \quad (S_{Prog}, \mathcal{I}) \models \phi$$

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 set of states, 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 $Im[\varrho^{-1}](\overline{Im[\varrho](\overline{\mathcal{I}(p)})}) \subseteq \overline{\mathcal{I}(p)}$ holds always. We conclude that, if ϕ holds on $Prog_A$ using the abstract interpretation $\overline{Im[\varrho](\overline{\mathcal{I}(p)})}$ of $\neg p$, then a stronger property than ϕ using the concrete interpretation $\overline{\mathcal{I}(p)}$ of $\neg p$ 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. Suppose that we want to show that the buffer of Example 1 has the property of *order preservation* — that means elements are taken out in the same order in which they are put into the buffer. This property can be expressed using atomic propositions in

$$\mathcal{P} = \{init, enable(push(x)), after(push(x)), enable(pop(x)), after(pop(x)) \mid x \in elem\}$$

by the following parameterized formula — that is a CTL* formula containing globally universally quantified rigid variables[1].

$$\forall e', e \in elem \,.\, init \;\Rightarrow\; \mathbf{A}(\; [\neg after(push(e))\mathbf{W}\; after(push(e'))] \;\Rightarrow\; [\neg enable(pop(e))\mathbf{W}\; after(pop(e'))] \;)$$

[1] a rigid variable or parameter never changes during the execution of a program

This formula can be transformed into a $\forall$CTL formula in which only the propositions of the form $after(push(e))$ and $after(pop(e'))$ occur non negated. The transformation into an $\forall$CTL* formula is immediate, due to the fact that for every operator there exists a dual one; in order to see that they are also in $\forall$CTL one can use a result given in [EH83].

In order to verify that the concrete buffer has the property of order preservation, it is sufficient to verify this property on the (finite) Kripke structure associated with the abstract buffer, provided that all the atomic propositions occurring non negated in the property are consistent (see Definition 5) with the abstraction relation ϱ relating the concrete and the abstract program.

How to define interpretations of atomic propositions for a program:

- The predicate *init* is interpreted as the predicate defining the set of initial states of the program.
- An atomic proposition of the form $enable(\ell)$ is interpreted as a predicate on program variables representing the set of states in which event ℓ is possible. Such a predicate is "$\exists X' \;.\; action_\ell(X, X')$" if ℓ is just a label and "$\exists X' \;.\; action_l(X, X')[E/Y]$" if $\ell = l(E)$ where l is a label and E a valuation of the vector of input variables Y.
- An atomic proposition of the form $after(\ell)$ is interpreted as a predicate on program variables representing the sets of states in which ℓ has just occurred. In order to make this predicate expressible as a predicate on program variables, one has in general to introduce a new boolean variable $after\_\ell$ for every proposition $after(\ell) \in \mathcal{P}$ which is set to $true$ whenever an event labeled ℓ occurs and to $false$ by all other events (by appropriate operations set_true and set_false). The so obtained program is equivalent to the original one as the values of the original variables do not depend of this new variable ($after\_\ell$ is added by superposition as defined in Unity [CM88]). In the sequel, we suppose that for every predicate $after(\ell) \in \mathcal{P}$ such a variable is defined, but we do not mention it explicitly in order to keep the programs simple. Usually, the set $\mathcal{P}$ of atomic propositions associated with a property is rather small such that just a few boolean variables have to be added.

Consistency: In the given example, the abstraction relation is not consistent with the interpretation $\mathcal{I}(init) = empty(\mathbf{B})$ as $Im[\varrho](empty(\mathbf{B})) = empty_A(\mathbf{B})$, but $Im[\varrho^{-1}](empty_A(\mathbf{B}))$ represents much more states than $empty(\mathbf{B})$ as it represents all states in which $\mathbf{B}$ contains any amount of non-distinguished elements. However, this is not a problem as in the property under study the atomic proposition *init* occurs negated.

The only atomic propositions of the property under study that occur non negated are of the form $after(\ell)$. It is easy to obtain the consistency of such a predicate by not abstracting the variable $after\_\ell$; that means $after\_\ell$ has in the abstract program the same type as the concrete one, and the abstract versions of the operations set_true and set_false are identical to the concrete versions. That is, the abstraction function is the identity.

Now it is easy to verify each instance of the formula above on the Kripke structure associated with the abstract buffer program by instantiating e_1 for e and e_2 for e'. It is clear that for each pair $e, e' \in elem$ this leads exactly to the same abstract transition relation and abstract interpretation of atomic propositions. That means that e_1 and e_2 represent an arbitrary pair of data values, and the verification of a single abstract property on a single abstract system is sufficient in order to prove the above set of formulas.

3 Abstract specification of a sequentially consistent memory

3.1 Characterization by a set of properties

In order to use the method presented in the previous section to verify that the distributed cache memory defined in [ABM93] is a "sequentially consistent memory" [Lam79], we need a characterization of this property in terms of a set of formulas of ∀CTL.

Consider a system with *observable* events of the form $read_i(a,d)$ and $write_i(a,d)$ and may be other (internal) events — where the index i indicates the process P_i performing the event, a is the address of a memory location and d a data element. The set *index* defining the size of the system is an integer interval of the form $[1..N]$. Such a system is a sequentially consistent memory if any of its computation sequences, projected on observable events, can be reordered — by respecting the order of the events with the same index — into a computation sequence of a central memory — that means a sequence in which $read_i(a,d)$ is only possible if the last *write* event concerning location a is of the form $write_j(a,d)$ for some index j.

For the exact characterization of this property — by using only observable, and not implementation dependent internal event names — one needs full second order logic, whereas we want to restrict ourselves to a set of propositional but parameterized temporal logic formulas which can be evaluated by model checking on a finite abstract model. Therefore, our characterization is necessarily stronger than required. For our convenience, we suppose that every pair of the form (a,d) can occur at most once as parameter of some *write* event. This assumption can be made without loss of generality as it is equivalent to adding (by superposition) an integer variable associating with each *write* event a unique index.

In implementations of a sequentially consistent memory (as the one we study here), a considerable amount of time may elapse, between the occurrence of the event $write_i(a,d)$ and the moment in which $read_i(a,d)$ is allowed; if $write_i$ events occur too often, some of the elements that have been written may never be readable in P_i (because they are "overwritten" before they are "available"). This makes the expression in terms of temporal logic difficult. However, suppose that for a given concrete system we can identify auxiliary predicates $avail_i(a,d)$ — the interpretation of which depends on the concrete system under study — which are weaker than $enable(read_i(a,d))$ (C1) but such that each event $write_j(a,d)$ is eventually followed by a state in which $avail_i(a,d)$ holds (C3), and — if $read_i(a,d)$ becomes possible in some future — from that moment on, until $avail_i(a,d)$ becomes "*false* forever", events of the form $read_i(a,d')$ for $d \neq d'$ are impossible (C2). Then, the expression of "sequentially consistent with a central memory" becomes possible.

In the sequel, instead of "$avail_i(a,d)$ holds", we write sometimes "(a,d) is available in process P_i".

Proposition 3. *(Properties guaranteeing sequential consistency)*
Let S be a transition system and $\mathcal{P}$ the set of predicates

$$\mathcal{P} = \{init, enable(read_i(a,d)), after(read_i(a,d)), \\ enable(write_i(a,d)), after(write_i(a,d))\}_{i:index,(a,d):address\times datum}$$

with the interpretation $\mathcal{I}$ defined as explained in the previous section. If it is possible to define an interpretation $\mathcal{I}_{aux}$ for the set of predicates

$$\mathcal{P}_{aux} = \{avail_i(a,d)\}_{i:index,(a,d):address\times datum}$$

such that $M = (S, \mathcal{I} \cup \mathcal{I}_{aux})$ satisfies the following set of properties, then the program generating model M is a sequentially consistent memory.

(C1) $\forall (a,d) \in address \times datum\ \forall i \in index$
$init \;\Rightarrow\; \mathbf{AG}(enable(read_i(a,d)) \;\Rightarrow\; avail_i(a,d))$

(C2) $\forall (a,d), (a,d') \in address \times datum\ .\ d \neq d'\ \forall i \in index$
$init \;\Rightarrow\; \mathbf{AG}((avail_i(a,d) \wedge \mathbf{EF}(enable(read_i(a,d))) \;\Rightarrow$
$\mathbf{A}[\neg avail_i(a,d') \mathbf{W}\ \mathbf{AG}(\neg avail_i(a,d))\ ])$

(C3) $\forall (a,d) \in address \times datum\ \forall i,k \in index$
$init \;\Rightarrow\; \mathbf{AG}[after(write_k(a,d)) \;\Rightarrow\; \mathbf{AF}(avail_i(a,d))\ ]$

(S1) $\forall (a,d) \in address \times datum\ \forall i \in index$
$init \;\Rightarrow\; \mathbf{AG}[\ after(write_i(a,d)) \;\Rightarrow\; \mathbf{A}(\neg enable(read_i) \mathbf{W}\ avail_i(a,d))\ ]$

(S2) $\forall (a,d) \in address \times datum\ \forall i \in index$
$init \;\Rightarrow\; \mathbf{A}(\neg avail_i(a,d) \mathbf{W}\ \bigvee_{k:index}\ after(write_k(a,d))\ )$

(S3) $\forall (a,d), (a',d') \in address \times datum\ .\ d \neq d'\ \forall i,k \in index$
$init \;\Rightarrow\; \mathbf{A}(\ [\neg after(write_k(a,d)) \mathbf{W}\ after(write_k(a',d'))\ ] \;\Rightarrow$
$[\neg avail_i(a,d) \mathbf{W}\ avail_i(a',d')\ ]\ )$

(S4) $\forall (a,d), (a',d') \in address \times datum\ .\ d \neq d'\ \forall i,k \in index$
$init \;\Rightarrow\; \mathbf{A}(\ [\neg avail_i(a,d) \mathbf{W}\ (avail_i(a',d') \wedge \neg avail_i(a,d)\ )] \;\Rightarrow$
$[\neg avail_k(a,d) \mathbf{W}\ avail_k(a',d')]\ )$

First a few remarks oncerning the choice of appropriate predicates $avail_i(a,d)$. In a central memory, $read_j(a,d)$ is enabled immediately after $write_i(a,d)$, that means $avail_i(a,d)$ and $enable(read_i(a,d))$ (the central memory holds datum d at address a) coincide. We will show that the distributed memory system that we want to verify satisfies the set of properties given above if we choose $avail_i(a,d)$ to be "the cache memory of process P_i holds datum d at address a"; for this choice, the condition (C1) is trivially satisfied in the system under study (given in Section 4.1).

Property (S1) expresses the requirement that in every process P_i as soon as an event $write_i(a,d)$ has occurred, $read_i$ events are impossible until (a,d) becomes available. This requirement looks very strong. However, the weaker and more intuive requirement that, after $write_i(a,d)$ only events $read_i(a)$ are forbidden until (a,d) is available in P_i, is not sufficient. Suppose that P_1 reads (a,d_1), then (a',d'_1), then (a,d_2) and then (a',d'_2) which guarantees by (S4) and (C2) that in all processes, (a,d_1) is available before (a,d_2) and analogously for the primed pairs. If in process P_2, $write_2(a,d_2)$ is followed by $read_2(a',d'_1)$ and in process P_3, $write_3(a',d'_2)$ is followed by $read_3(a,d_1)$, then these sequences cannot be merged and completed into a sequence of a central memory, but the system may satisfy all the above properties except that (S1) is replaced by the proposed weaker property.

Property (S2) expresses that in process P_i (a,d) cannot become available before $write_k(a,d)$ has occurred for some index k. This property is quite natural but could be weakened; what we need to express in order to guarantee sequential consistency is only that whenever (a,d) is available in P_i, then (a,d) must be written at some moment (earlier or later) by some P_k, and only if it is written by P_i it cannot be later. However, most concrete systems implementing a sequentially consistent memory satisfy property (S2).

Property (S3) expresses that the $write_k$ events become available in P_i in a compatible order, i.e., whenever $write_k(a', d')$ occurs before $write_k(a, d)$, then (a, d) cannot become available in P_i before (a', d'); they may become available at the same moment, except if the premiss of (C2) holds.

Property (S4) expresses that in any pair of processes pairs (a, d) become available in a compatible order; that means if $avail_i(a, d)$ holds strictly before $avail_i(a', d')$, then $avail_j(a, d)$ and $avail_j(a', d')$ may become true at the same moment but not in the opposite order.

Both (S3) and (S4) have the intended meaning only because of (C3). For example an execution sequence, in which process P_1 reads (a', d'_1) then (a, d_1) and then (a, d_2), process P_2 reads (a', d'_1) then (a', d'_2) and then (a, d_1), and process P_3 reads (a, d_2) and then (a', d'_1), can obviously not be merged and completed to a sequence of a central memory, but may be completed to a sequence satisfying all the above properties except (C3). In fact, pairwise compatibility of the order in which pairs (a, d) become available (required by (S3) and (S4)) implies global compatibility only then, when all pairs (a, d) which are read by some process become effectively available at some moment in all processes and not only in those in which a corresponding *read* event occurs (as required by (C1)).

However, despite the fact that the original abstract specification does not contain any liveness condition, a liveness property (slightly weaker than (C3)) is necessary in order to obtain a sufficient temporal logic characterization of a sequentially consistent memory.

Notice that, all the above formulas can easily be translated into $\forall$CTL formulas.

3.2 Proof of the correctness of this characterization

It remains to be shown that every system satisfying the requirement of Proposition 3 is a sequentially consistent memory. In order to do so, we show that the sequence of observable events associated with an arbitrary computation sequence π of a system satisfying properties (C1) to (S4), is a sequence of a sequentially consistent memory; that is, it can be finitely reordered respecting the order of the events of each individual process into a sequence π_{seq} of central memory.

In order to do so, we define for all indices i the sequence π_i of observable events of process P_i in π — that means, π_i is the sequence of events of the form $write_i(a, d)$ and $read_i(a, d)$ occurring in the order defined by π. We define also a sequence OWE, defining the order of *write* events in π_{seq}. Let

$$\mathcal{A} = \{(a, d) \mid \exists k.write_k(a, d) \in \pi\}$$

be the set of pairs (a, d) occurring as the parameter of some *write* event in π. Notice that, due to property (S1), this is in fact exactly the set of pairs (a, d) occurring as the parameter of *any* event in π. Property (C3) guarantees that for all indices $i \in [1..N]$,

$$\leq_i = \{((a, d), (a', d')) \in \mathcal{A}^2 \mid \text{first occurrence of } avail_i(a, d) \text{ is not after} \\ \text{first occurrence of } avail_i(a', d') \text{ in } \pi\}$$

is a preorder on $\mathcal{A}$. Denote by $<_i$ the corresponding strict relation (which is not necessarily an order on $\mathcal{A}$ as there may exist unordered pairs (a, d), (a', d') such that $(a, d) \leq_i (a', d') \wedge$

$(a', d') \leq_i (a, d)$ holds). Property (S4) guarantees that for all pairs of indices these preorders are *compatible* in the sense that

$$\forall i, k\,.\ <_i \cap >_k = \emptyset \text{ or equivalently } <_i \subseteq \leq_k$$

This guarantees that $\leq' = \bigcap \leq_k$ is a preorder on $\mathcal{A}$ and its corresponding strict relation is $<' = \bigcup <_k$. If $<'$ is not an order, then extend $<'$ to an order $<$ by ordering all unordered pairs according to the order of the corresponding *write* events in the sequence π. Let OWE be the sequence of elements of $\mathcal{A}$ defined by $<$ starting with the smallest element. Thus, OWE contains each pair $(a, d) \in \mathcal{A}$ exactly once.

Using these definitions, build a (possibly infinite) sequence π_{seq} in which *write* events occur in the order defined by OWE, that is by $<$, by means of the following procedure.

```
$\pi_{seq} := \epsilon;\ \forall a \in address\,.\,lw(a) := \epsilon;\ nw := first(OWE);$
$b := true;$
while $b$ do
    $b := false;$
    for $i := 1$ to $N$ do
        if $\exists a\,.\,first(\pi_i) = read_i(a, lw(a))$ then
          " $\pi_{seq} := append(\pi_{seq}, first(\pi_i));\ b := true;\ \pi_i := tail(\pi_i)$ ";
        if $first(\pi_i) = write_i(nw) \wedge \forall j.read_j(nw.a, lw(nw.a)) \notin \pi_j$ then
          " $\pi_{seq} := append(\pi_{seq}, first(\pi_i));\ b := true;\ \pi_i := tail(\pi_i);$
          $lw(nw.a) := nw.d;\ OWE := tail(OWE);\ nw := first(OWE)$ "
    endfor
endwhile
if $\exists i.\neg empty(\pi_i)$ then "error state" else "correct termination";
```

At any moment, for any address a, $lw(a)$ contains the last datum that has been written on address a, and nw contains the first element of OWE which defines the next *write* event to be appended to π_{seq}. That means, at any moment, any sequence π_i can only contain $write_i(a, d)$ events such that $nw \leq (a, d)$. We denote by $nw.a$ and $nw.d$ respectively the address and the data part of the pair nw.

At any moment, "the next event to be appended to π_{seq}" is one that satisfies one of the following two conditions:

(a) it is of the form $write_i(nw)$ and there remains no event of the form $read_j(nw.a, lw(nw.a))$ in π_j, (i.e. they have already been appended to π_{seq} and eliminated from π_j).
(b) or it is of the form $read_j(a, lw(a))$ for some a.

Naturally, there may be several events satisfying one of these conditions, one *write* and several *read* events. Notice that the conditions (a) and (b) remain true until the corresponding events are appended to π_{seq}. This guarantees — together with the fact that the algorithm looks at all sequences π_i in a round Robin manner — that every event satisfying (a) or (b) is appended to π_{seq} after a finite number of steps.

Whenever the procedure does not terminate, this means that it never gets stuck and continues forever to produce longer and longer prefixes of the infinite sequence π_{seq}.

What we have to show is that, under the condition that all properties of Proposition 3 hold,

(1) at any moment, (the prefix so far constructed of) π_{seq} is a sequence of a central memory consistent with the order of the events in each π_i,

(2) the procedure cannot terminate in the error state, that means, as long as there are still non-empty sequences π_i (containing events not yet appended to π_{seq}), there exists at least one sequence π_i such that its first event satisfies either condition (a) or (b).

(3) The (infinite) sequence π_{seq} is a finite reordering of the sequence of observable events associated with π, that means, every event of every π_i is appended to π_{seq} after a finite number of steps of the algorithm.

Proof of (1): At any moment, the prefix of π_{seq} so far constructed is a sequence of a sequentially consistent memory because during the whole execution of the algorithm, an event $read_i(a,d)$ can only be appended to the sequence π_{seq} if the most recent *write* event in π_{seq} concerning address a is of the form $write_j(a,d)$ for some index j. It is trivial to observe that the above algorithm appends each event of π at most once to π_{seq} and in an order consistent with the order of the events in each π_i.

Proof of (2): We want to show that it is not possible that the procedure can terminate in the error state because the first elements of all sequences π_i satisfy neither condition (a) nor (b). That means that,

- either OWE is empty — there are no more *write* events to be appended to π_{seq} — but there is at least one event of the form $read_i(a,d)$ not satisfying condition (b), i.e. such that $d \neq lw(a)$
- or the (unique) event of the form $write_j(nw)$ occurring in π, which is the next *write* event to be appended to π_{seq}, is preceded by events not satisfying condition (b)
- or condition (a) is not satisfied because in some sequence π_k there exist still events of the form $read_k(nw.a, lw(nw.a))$ preceded by events not satisfying condition (b).

Let us show that the first case is not possible. As nw is the greatest element of OWE and the last *write* event concerning address a is of the form $write_k(a, lw(a))$, we have necessarily $(a,d) < (a, lw(a))$. In this case we call $read_i(a,d)$ an "old" event. But there cannot be any old events in π_i as the algorithm allows to update $lw(a)$ only if all events of the form $read_k(a, lw(a))$ are eliminated from π_k.

Let us now show that the second case is impossible. Notice that the unique event $write_j(nw)$ occurring in π is still in π_j as the pair nw has exactly one occurrence in OWE and as soon as $write_j(nw)$ is appended to π_{seq}, nw is updated. Let us consider all events that could occur in π_j before $write_j(nw)$ and block the procedure.

- If $write_j(nw)$ is preceded by an event of the form $write_j(a,d)$ in π_i and therefore also in π, this implies on one hand by a remark made just after the definition of the algorithm that $nw < (a,d)$ (*). On the other hand, the fact that $write_j(a,d)$ occurs before $write_j(nw)$ implies by property (S3) $(a,d) \leq' nw$. The definition of $<$ implies that $(a,d) < nw$ either because $(a,d) <_k nw$ for some k or because of the above supposed order of the corresponding *write* events. This is in contradiction with (*).
- If $write_j(nw)$ is preceded by an event of the form $read_j(a,d)$ which cannot be appended to π_{seq}, then $(a, lw(a)) \leq' (a,d)$ as old *read* events are not possible. This implies that

- either $d = lw(a)$, implying that $read_j(a,d)$ satisfies condition (b).
- or $d \neq lw(a)$ and $(a, lw(a)) < (a,d)$. As the *write* events occur in π_{seq} in the order defined by OWE and $write(a, lw(a))$ is the most recent *write* event concerning address a before $write_j(nw)$, we deduce that $write(a,d)$ cannot occur before $write_j(nw)$ in π_{seq} and therefore $nw \leq_j (a,d)$ (**). This means that,
  - either $(a,d) = nw$ which is clearly in contradiction with (C1) and (S1) saying that (a,d) cannot be read before it has been written.
  - or $(a,d) \neq nw$. In this case, the fact that $read_j(a,d)$ occurs (strictly) before $write_j(nw)$ in π implies by properties (S2) and (C1) that $(a,d) <_j nw$ contradicting (**).

It remains to be shown that the third case is impossible, i.e., that events of the form $read_j(a,d)$ where $(a,d) = (nw.a, lw(nw.a))$ cannot be preceded by events not satisfying condition (b). Notice that $a = nw.a$ and property (C2) imply $(a,d) <_j nw$ (***) as the event $read_j(a,d)$ occurs in π.

- If $read_j(a,d)$ is preceded in π by $write_j(a',d')$, then this implies on one hand, exactly as in the second case, that $nw \leq_j (a',d')$, implying with (***), $(a,d) <_j (a',d')$ (****). On the other hand, the fact that $write_j(a',d')$ occurs before $read_j(a,d)$ implies by property (S1) that $(a',d') \leq_j (a,d)$ which contradicts (****).
- If $read_j(a,d)$ is preceded in π by $read_j(a',d')$, then as in the second case
  - either $d' = lw(a')$ implying that $read_j(a',d')$ satisfies condition (b).
  - or $nw \leq_j (a',d')$ by the same argument as in the second case. However, the fact that $a = nw.a$ implies by (C1) and (C2) that a soon as $avail_j(nw)$ holds, the event $read_j(a,d)$ is not possible anymore, and on the other hand $nw \leq (a',d')$ implies by (C1) that $read_j(a',d')$ cannot occur before $avail_j(nw)$ holds, making the above order of *read* events impossible.

That means that the procedure cannot terminate in the error state, as either there exists always at least one event satisfying (a) or (b) that can be consumed or all the sequences π_i are empty and the algorithm terminates correctly.

Proof of (3): If all the sequences π_i are finite and the procedure terminates (correctly), π_{seq} is necessarily a finite reordering of π. It remains to be shown that, also if the procedure never terminates, at any moment, the first element of each sequence π_i will be appended to π_{seq} after a finite number of steps of the algorithm.

By definition of OWE, the parameter (a,d) of any event occurring in π_i occurs at some (finite) position of OWE.

First, we show that at any moment the first element nw of OWE can be consumed after a finite number of steps appending *read* events to π_{seq}. From the proof of (2) we deduce that the only possibility that condition (b) does not hold for $write_j(nw)$ after a finite number of steps, is that there exists an infinite number of events of the form $read_k(nw.a, lw(nw.a))$ in π. However, the existence in π of an infinite number of $read_k(nw.a, lw(nw.a))$ events implies that π satisfies the property $\mathbf{GF}(enable(read_i(nw.a, lw(nw.a))))$ which by (C1) implies $\mathbf{GF}(avail_i(nw.a, lw(nw.a)))$. As nw occurs after $(nw.a, lw(nw.a))$ in OWE, property (C2) implies that in any process, nw can only become available when $(nw.a, lw(nw.a))$ has become unavailable forever, which due to $\mathbf{GF}(avail_i(nw.a, lw(nw.a)))$ means that nw can never become available, in contradiction with the fact that nw occurs in OWE. This implies that, if for some pair (a, d) there exists an infinite number of *read* events in π, then OWE cannot contain a pair of the form (a, d') occurring after (a, d) showing that the above situation is impossible.

This implies that the first element of OWE becomes always consumable — and therefore consumed — after a finite number of steps. This guarantees — using the fact that the procedure cannot terminate before all sequences π_i are empty — that at any moment, if the first event of a sequence π_i has parameter (a, d), then, after a finite number of steps, the value of the variable nw becomes (a, d), and either condition (a) or (b) will hold for this event and it will be appended to π_{seq} after another finite number of steps.

This terminates the proof of (3) and therefore that of Proposition 3. $\square$

4 Verification of a distributed cache memory

4.1 Definition of the concrete system

In our program formalism, the cache memory proposed by [ABM93] can be described as a system of the form $P_1 \parallel\!| P_2 \ldots \parallel\!| P_n$ where each process P_i is defined as follows:

Name :	P_i	
Variables :	Input :	$\mathbf{a} : address,\ \mathbf{d} : datum$
	local :	$\mathbf{E}_i : set$ of $address \times datum_i$, (already occurred $write_i$ events)
		$\mathbf{C}_i : memory$ of $address \times (datum \cup \{\epsilon\})$ (local cache memory)
		$\mathbf{Out}_i : buffer$ of $address \times datum_i$
	shared :	$\mathbf{M} : memory$ of $address \times (datum \cup \{\epsilon\})$ (global memory)
		$\mathbf{In}_k : buffer$ of $(address \times datum) \times Bool,\ k : index$
Transitions :		
$(write_i(\mathbf{a}, \mathbf{d}))$	$allowed((\mathbf{a}, \mathbf{d}), \mathbf{E}_i, \mathbf{E}'_i) \wedge append(\mathbf{Out}_i, (\mathbf{a}, \mathbf{d}), \mathbf{Out}'_i) \wedge unch(\mathbf{C}_i, \mathbf{M}, \mathbf{In}_1, \ldots, \mathbf{In}_n)$	
$(read_i(\mathbf{a}, \mathbf{d}))$	$holds(\mathbf{C}_i, (\mathbf{a}, \mathbf{d})) \wedge empty(\mathbf{Out}_i) \wedge empty\_true(\mathbf{In}_i) \wedge unch(\mathbf{E}_i, \mathbf{C}_i, \mathbf{Out}_i, \mathbf{M}, \mathbf{In}_1, \ldots, \mathbf{In}_n)$	
$(mw_i(\mathbf{a}, \mathbf{d}))$	$first(\mathbf{Out}_i, (\mathbf{a}, \mathbf{d})) \wedge tail(\mathbf{Out}_i, (\mathbf{a}, \mathbf{d}), \mathbf{Out}'_i) \wedge update(\mathbf{M}, (\mathbf{a}, \mathbf{d}), \mathbf{M}') \wedge \forall k \in index\ .\ append(\mathbf{In}_k, ((\mathbf{a}, \mathbf{d}), i = k), \mathbf{In}'_k) \wedge unch(\mathbf{E}_i, \mathbf{C}_i)$	
$(cu_i(\mathbf{a}, \mathbf{d}))$	$\exists b \in Bool\ .\ (first(\mathbf{In}_i, ((\mathbf{a}, \mathbf{d}), b)) \wedge tail(\mathbf{In}_i, ((\mathbf{a}, \mathbf{d}), b), \mathbf{In}'_i)) \wedge update(\mathbf{C}_i, (\mathbf{a}, \mathbf{d}), \mathbf{C}'_i) \wedge unch(\mathbf{E}_i, \mathbf{Out}_i, \mathbf{M}, \{\mathbf{In}_j, j \neq i\})$	
$(mr_i(\mathbf{a}, \mathbf{d}))$	$holds(\mathbf{C}_i, (\mathbf{a}, \epsilon)) \wedge holds(\mathbf{M}, (\mathbf{a}, \mathbf{d})) \wedge \neg isin(\mathbf{In}_i, (\mathbf{a}, \mathbf{d})) \wedge append(\mathbf{In}_i, ((\mathbf{a}, \mathbf{d}), false), \mathbf{In}'_i) \wedge unch(\mathbf{E}_i, \mathbf{C}_i, \mathbf{Out}_i, \mathbf{M}, \{\mathbf{In}_j, j \neq i\})$	
$(cl_i(\mathbf{a}))$	$clear(\mathbf{C}_i, \mathbf{a}, \mathbf{C}'_i) \wedge unch(\mathbf{E}_i, \mathbf{Out}_i, \mathbf{M}, \mathbf{In}_1, \ldots, \mathbf{In}_n)$	
Init :	$\forall a \in address\ .\ (holds(\mathbf{C}_i, (a, \epsilon)) \wedge holds(\mathbf{M}, (a, \epsilon))\ ) \wedge empty(\mathbf{Out}_i) \wedge empty(\mathbf{In}_i)$	

The predicates *append*, *tail*, *first*, *allowed* and *empty* are defined as in Example 1, where the type *elem* is replaced by the type $address \times datum$, respectively $(address \times datum) \times Bool$. Let B be a possible value of variable $\mathbf{In}_i$. Then, $empty\_true(B)$ holds if B contains no element with boolean parameter *true*, that means

$$empty\_true(B) = empty(B_{|(address \times datum) \times \{true\}})$$

The predicate $isin(B, e)$ for $e \in address \times datum$, evaluates to *true* if there exists some boolean value b such that the pair (e, b) is somewhere in B.
memory of $address \times (datum \cup \{\epsilon\})$ is a data type representing a memory with address space *address*. If M is such a memory and $(a, d) \in address \times (datum \cup \{\epsilon\})$, the predicate $holds(M, (a, d))$ expresses the fact that M contains datum d at address a; it has furthermore the property that $\forall a \in address$ there exists exactly one $d \in datum \cup \{\epsilon\}$ such that $holds(M, (a, d))$ is true. The predicates *update* and *clear* are defined by

$$\begin{array}{ll} update(M, (a, d), M') \equiv & holds(M', (a, d)) \wedge \\ & \forall b \in address\ .\ (b \neq a \ \Rightarrow\ (holds(M, (b, d')) \equiv holds(M', (b, d')))\ ) \\ clear(M, a, M') \equiv & holds(M', (a, \epsilon)) \wedge \\ & \forall b \in address\ .\ (b \neq a \ \Rightarrow\ (holds(M, (b, d')) \equiv holds(M', (b, d')))\ ) \end{array}$$

The only differences between our system and the one described in [Ger94] concerns

- the fact that each pair (a, d) can be the parameter of at most one *write* event. The way we obtain this, is by defining the type *datum* as $\bigcup_i datum_i$, such that each process "signs" the data it writes, and by using in each process a variable $\mathbf{E}_i$ of type *set* of $address \times datum_i$ containing all the pairs (a, d) such that the event $write_i(a, d)$ has already occurred, as in the example of the buffer.
- The additional condition $\neg isin(\mathbf{In}_i, (\mathbf{a}, \mathbf{d}))$ in the action $mr_i(\mathbf{a}, \mathbf{d})$ which is very reasonable in practice as otherwise too frequent mr_i events may fill the buffers $\mathbf{In}_i$ and delay the treatment of the *write* events waiting in buffer $\mathbf{Out}_i$. Here, we add this condition to be able to use a simple abstraction of the buffers $\mathbf{In}_i$, similar to the one presented in Section 2. We will also show how to verify the system without this restriction.

4.2 Construction of abstract systems

We verify the parameterized formulas of Proposition 3 on different abstract systems. Our aim is not necessarily to find the smallest abstract system that can be used for the verification of each formula, but we want to use, whenever possible, the already predefined abstractions in order to show that the application of the method is simple.

Definition of abstract types and predicates We use the same abstract types $elem_A$, *set* of *abs_elem* and $buffer_A^K$ of *abs_elem* and (almost) the same abstract predicates as for the verification of the lossy buffer, despite the fact that the variables in the cache memory system are not exactly of the same type as the variables of the lossy buffer.

Let us define $elem = address \times (datum \cup \{\epsilon\})$ and $elem_i = address \times datum_i$. As before, given a set of pairs $e\_obs = \{(a_1, d_1), ..., (a_n, d_n)\} \subseteq elem$, where we suppose that $\forall k \in \{1, ...n\}\ .\ d_k \neq \epsilon$, we use as abstract type for *elem* the type

$$elem_A = \{0\} \cup abs\_elem \text{ for } abs\_elem = \{1, ..., n\}$$

and relate the concrete and the abstract type by $\varrho^{e\_obs}_{elem}$.

The cache memory uses also a data type *memory*. Each variable of type *memory* is either existentially abstracted (i.e., omitted in the corresponding abstract program) or replaced by a variable of type *set* of *abs_elem*, and

$$\forall M \in memory \text{ of } elem\ .\ \ \varrho^{e\_obs}_{mem}(M) = \{\varrho^{e\_obs}_{elem}(a,d) \mid holds(M,(a,d))\} \cap abs\_elem\}$$

Then, it is obvious to define abstract predicates

$$\begin{aligned} &holds_A(M_A, e_A) = (e_A = 0) \vee (e_A \in M_A) \\ &clear_A(M_A, e_A, M'_A) = (e_A = 0) \wedge ((M_A = M'_A) \vee \exists e_A{}'.(M'_A = M_A - \{e_A{}'\})) \vee \\ &\qquad\qquad\qquad\qquad\qquad (e_A \neq 0) \wedge (M'_A = M_A - \{e_A\}) \end{aligned}$$

For the definition of the abstract predicate $update_A$ we need an auxiliary predicate on abstract elements, $same\_addr(e_A, e_A{}')$ that evaluates to *true* if its arguments are related via $\varrho^{e\_obs}_{elem}$ with concrete pairs with the same address. Using this auxiliary predicate, we can define

$$\begin{aligned} update_A(M_A, e_A, M'_A) =& (e_A = 0) \wedge ((M_A = M'_A) \vee \exists e_A{}'.(M'_A = M_A - \{e_A{}'\})) \vee \\ & (e_A \neq 0) \wedge (M'_A = M_A \cup \{e_A\} - \{e_A{}' \in M_A \mid same\_addr(e_A{}', e_A)\} \end{aligned}$$

Notice that for $e_A = 0$, the operations $update_A$ and $clear_A$ are the same.

For existentially abstracted memories, the abstract predicates $holds^{ex}_A, clear^{ex}_A, \ldots$ evaluate to *true* independently of the value of the argument of type $elem_A$.

In the processes P_i occur different types of sets and of buffers: variables $\mathbf{E}_i$ of type *set* of $elem_i$, variables $\mathbf{Out}_i$ of type *buffer* of $elem_i$, and variables $\mathbf{In}_i$ of type *buffer* of $elem \times Bool$.

Each variable $\mathbf{E}_i$ is either existentially abstracted or replaced by a variable of type *set* of *abs_elem* which is related with the concrete type via $\varrho^{e\_obs}_{set}$ (the same function as for the lossy buffer). However, as $e_A \in abs\_elem$ may or may not be related to some $e \in elem_i$ (it is always related to some $e \in elem$), we have to define abstract predicates $allowed^i_A(e_A, E_A, E_A{}')$ depending on the index i or more precisely on the the fact if e_A represents some pair in $elem_i$ or not. For this reason we need auxiliary predicates dat_i on abstract elements defined by

$$dat_i(e_A) = \exists e \in elem\ .\ (\varrho^{e\_obs}_{elem}(e) = e_A) \wedge (e \in elem_i)$$

Then, the abstract predicate for *allowed* can be defined as

$$allowed^i_A(e_A, E_A, E_A{}') = dat_i(e_A) \wedge allowed_A(e_A, E_A, E_A{}')$$

where $allowed_A$ is the predicate defined for the abstract lossy buffer. For existentially abstracted variables $\mathbf{E}_i$, we need abstract predicates defined analogously, that is,

$$allowed^{ex,i}_A(e_A) = dat_i(e_A) \wedge allowed^{ex}(e_A) \quad = \quad dat_i(e_A)$$

Similarly, each variable $\mathbf{Out}_i$ is either existentially abstracted or replaced by a variable of type $buffer^K_A$ of $elem_A$, related with the concrete type via $\varrho^{e\_obs,K}_{buffer}$; For these abstract buffers we need abstract predicates for *append* and *first* depending on the predicates dat_i:

$$\begin{aligned} append^{K,i}_A(B_A, e_A, B'_A) &= dat_i(e_A) \wedge append^K_A(B_A, e_A, B'_A) \\ first^i_A(B_A, e_A) &= dat_i(e_A) \wedge first_A(B_A, e_A) \\ append^{ex,i}_A(e_A) &= dat_i(e_A) \\ first^{ex,i}_A(e_A) &= dat_i(e_A) \end{aligned}$$

The abstract predicates for *tail* and *empty* do not depend on dat_i and we can use the abstract predicates $tail_A$, $empty_A$,... defined for the lossy buffer.

The variables $\mathbf{In}_i$ are all of the same type $buffer$ of $elem{\times}Bool$, and in the corresponding abstract buffers we cannot merge $((a,d), true)$ and $((a,d), false)$ for a pair $(a,d) \in e\_obs$ into a single abstract value without losing the preservation of the properties we want to verify — what we lose is in particular the consistency for the predicates $enable(read_i(a,d))$. Therefore, we define a slightly different abstract type

$$buffer^K \text{ of } abs\_elem \times Bool = (sequence^K \text{ of } abs\_elem \times Bool) \cup \{\perp\}$$

where the concrete and the abstract buffers are related by

$$\forall B \in buffer \text{ of } elem{\times}Bool \ . \ \varrho^{e\_obs,K}_{buf{\times}Bool}(B) = \begin{cases} \epsilon & \text{if } length(Obs) = 0 \\ \varrho^{e\_obs}_{elem}(Obs) & \text{if } 1 \leq length(Obs) \leq K \\ \perp & \text{if } length(Obs) > K \end{cases}$$

where $Obs = B_{|e\_obs{\times}Bool}$. The different associated abstract predicates, such as $append_A^{\times Bool,K}$, ... can be defined by an obvious systematic modification of the definitions given for the lossy buffer. In the processes P_i occur also predicates *empty_true* and *isin*. The abstract predicates for *empty_true* can easily be defined by

$empty\_true_A^{\times Bool}(B_A) = (B_{A|abs\_elem \times \{true\}} = \epsilon)$
$empty\_true_A^{\times Bool,ex} = true$

The predicate *isin* occurs negated in P_i. Therefore, we need, instead of an abstract predicate for *isin*, an abstract predicate for $\neg isin$ satisfying condition (2) of Section 2.2:

$not\_isin_A^{\times Bool}(B_A, e_A) = (B_A = \epsilon) \vee (B_A = \perp) \vee \nexists e_A, b, B_A^1, B_A^2 \ . \ (B_A = B_A^1 \bullet (e_A, b) \bullet B_A^2)$
$not\_isin_A^{\times Bool,ex}(e_A) = true$

Now, if we restrict ourselves to the abstraction functions and corresponding abstract types and predicates already defined, an abstract cache memory system is completely defined by

- its declaration part, where for each variable occurring in the concrete program we have the choice to omit it (existential abstraction) or to use the abstract type induced by the choice of the abstract type of the variable *elem*.
- the concrete set *e_obs* or alternatively the auxiliary predicates *same_addr* and dat_i which contain already all the useful information of *e_obs*.

This determines completely the abstract predicate to be used for every occurrence of a concrete predicate in the program.

We define for each property of Proposition 3 one or several abstract systems.

Definition of abstract systems Each instance of the properties to be verified involves only events of a few processes concerning at most two different pairs in $address{\times}datum$. However, even if the property involves only events of a few processes, it is not necessarily correct to verify the property on the system consisting only of the concerned processes as influences of all other processes may get lost using this approach. It is allowed to verify a property on the abstract system obtained by replacing all other processes by the process *Chaos*, but on this abstraction, the property under consideration does only hold if it holds in an (almost) arbitrary environment; for example, the event $mw_j(a,d)$ of a chaotic process P_j may allow $holds(\mathbf{C}_i,(a,d))$ to become true before any event $write_k(a,d)$ has occurred and therefore invalidate property (S2). For the verification of the Cache memory system under study, it is sufficient in the processes "not concerned with the property" to keep some information on global variables and to forget about all local variables. In practice, for global variables, the same abstract type is chosen in all processes.

Abstract system for property (S1): Each instance of property (S1) involves only events of a single process P_i concerning a single pair (a,d). Intuitively, (S1) is guaranteed by the fact that in process P_i after the occurrence of an event $write_i(a,d)$, $read_i$ events are impossible at least until (a,d) has traversed the buffers $\mathbf{Out}_i$ and $\mathbf{In}_i$ and has become available, that is, datum d has been written at address a in the cache memory $\mathbf{C}_i$. That means that we need to observe the cache $\mathbf{C}_i$ and all variables which may cause $enable(read_i(a,d))$ to hold. That is the buffers $\mathbf{Out}_i$ and $\mathbf{In}_i$ but also the global memory $\mathbf{M}$ which affects $\mathbf{In}_i$ and therefore also $\mathbf{C}_i$ via the action mr_i. It is not necessary to observe the buffers $\mathbf{Out}_j$ for $j \neq i$: for $d \in datum_i$ the action mw_j will never push (a,d) into $\mathbf{In}_i$ as it is not pushed into the buffer $\mathbf{Out}_j$ by action $write_j$. The same holds for the abstract action mw_j due to the definition of the predicate $first_A^{ex,j}(e_A)$. That means we need to distinguish a single pair in $address \times datum_i$ and define consequently the abstract element type $elem_A^1$ which is completely defined by

$abs\_elem = \{1\}$
$\forall e_A, e_A{}' \in elem_A^1 \,.\, same\_addr(e_A, e_A{}') = true$
$\forall j \in index \forall e_A \in elem_A^1 \,.\, dat_j(e_A) = (j = i) \vee (e_A = 0)$

The fact that we do not want to abstract existentially from the central memory and from all variables with index i, but from the local variables of all other processes, leads to the following abstract programs — by choosing everywhere $K = 1$, the number of elements in *abs_elem*.

Name :	P^1_{iA}
Variables :	abstract input : $\mathbf{e}$: $elem^1_A$
	local : $\mathbf{E}_i, \mathbf{C}_i$: *set* of abs_elem
	$\mathbf{Out}_i$: $buffer^1_A$ of abs_elem
	shared : $\mathbf{M}$: *set* of abs_elem
	$\mathbf{In}_i$: $buffer^1_A$ of $(abs\_elem \times Bool)$
Transitions :	
$(write_i(\mathbf{e}))$	$allowed^i_A(\mathbf{e}, \mathbf{E}_i, \mathbf{E}'_i) \wedge append^{1,i}_A(\mathbf{Out}_i, \mathbf{e}, \mathbf{Out}'_i) \wedge unch(\mathbf{C}_i, \mathbf{M}, \mathbf{In}_i)$
$(read_i(\mathbf{e}))$	$holds_A(\mathbf{C}_i, \mathbf{e}) \wedge empty_A(\mathbf{Out}_i) \wedge empty\_true^{\times Bool}_A(\mathbf{In}_i) \wedge unch(\mathbf{E}_i, \mathbf{C}_i, \mathbf{Out}_i, \mathbf{M}, \mathbf{In}_i)$
$(mw_i(\mathbf{e}))$	$first^{1,i}_A(\mathbf{Out}_i, \mathbf{e}) \wedge tail_A(\mathbf{Out}_i, \mathbf{e}, \mathbf{Out}'_i) \wedge update_A(\mathbf{M}, \mathbf{e}, \mathbf{M}') \wedge append^{\times Bool,1}_A(\mathbf{In}_i, (\mathbf{e}, true), \mathbf{In}'_i) \wedge unch(\mathbf{C}_i, \mathbf{E}_i)$
$(cu_i(\mathbf{e}))$	$\exists b \in Bool\ .\ first^{\times Bool,1}_A(\mathbf{In}_i, (\mathbf{e}, b)) \wedge tail^{\times Bool}_A(\mathbf{In}_i, (\mathbf{e}, b), \mathbf{In}'_i) \wedge update_A(\mathbf{C}_i, \mathbf{e}, \mathbf{C}'_i) \wedge unch(\mathbf{E}_i, \mathbf{Out}_i, \mathbf{M})$
$(mr_i(\mathbf{e}))$	$holds_A(\mathbf{M}, \mathbf{e}) \wedge not\_isin^{\times Bool}_A(\mathbf{In}_i, \mathbf{e}) \wedge append^{\times Bool,1}_A(\mathbf{In}_i, (\mathbf{e}, false), \mathbf{In}'_i) \wedge unch(\mathbf{E}_i, \mathbf{C}_i, \mathbf{Out}_i, \mathbf{M})$
$(cl_i(\mathbf{e}))$	$clear_A(\mathbf{C}_i, \mathbf{e}, \mathbf{C}'_i) \wedge unch(\mathbf{E}_i, \mathbf{Out}_i, \mathbf{M}, \mathbf{In}_i)$
Init :	$(\mathbf{C}_i = \emptyset) \wedge (\mathbf{M} = \emptyset) \wedge empty_A(\mathbf{Out}_i) \wedge empty^{\times Bool}_A(\mathbf{In}_i)$

Name :	$P^{1,ex}_{jA}$ for all indices $j \neq i$
Variables :	abstract input : $\mathbf{e}$: $elem^1_A$
	shared: $\mathbf{M}$: *set* of abs_elem
	$\mathbf{In}_i$: $buffer^1_A$ of $(abs\_elem \times Bool)$
Transitions :	
$(write_j(\mathbf{e}))$	$dat_j(\mathbf{e}) \wedge unch(\mathbf{M}, \mathbf{In}_i)$
$(read_j(\mathbf{e}), cu_j(\mathbf{e}), cl_j(\mathbf{e}))$	$unch(\mathbf{M}, \mathbf{In}_i)$
$(mr_j(\mathbf{e}))$	$holds(\mathbf{M}, \mathbf{e}) \wedge unch(\mathbf{M}, \mathbf{In}_i)$
$(mw_j(\mathbf{e}))$	$first^{1,j,ex}_A(\mathbf{e}) \wedge append^{\times Bool,1}_A(\mathbf{In}_i, (\mathbf{e}, false), \mathbf{In}'_i) \wedge update_A(\mathbf{M}, \mathbf{e}, \mathbf{M})$
Init :	$(\mathbf{M} = \emptyset) \wedge empty^{\times Bool}_A(\mathbf{In}_i)$

We have already eliminated all abstract operations that are equivalent to *true*, such as $append^{1,j,ex}_A$, $update^{ex}_A$,.... Notice that $mw_j(1)$ can never be executed as $first^{1,i,ex}_A(1) = dat_j(1) = false$.

For all indices $j \neq i$, the programs $P^{1,ex}_{jA}$ define the *same* transition relation (they depend on the same set of variables) which implies that also the parallel composition of an arbitrary number of these processes represents the same transition relation as a single one. Therefore, the system $P^1_{iA} \parallel P^{1,ex}_{jA}$ is equivalent to $P^{1,ex}_{1A} \parallel ... \parallel P^1_{iA} \parallel ... \parallel P^{1,ex}_{nA}$ which means that it is an abstraction for an arbitrary instance of a cache memory system.

Abstract systems for property (S2): Property (S2) expresses that any event $read_i(a,d)$ is preceded by an event $write_k(a,d)$ for some k. We verify a stronger property, saying that $\forall k \forall (a,d) \in elem_k$, $read_i(a,d)$ is preceded by $write_k(a,d)$. Thus, in order to define an appropriate abstract system, we distinguish a single element $(a,d) \in elem_k$, and we need a nonexistential abstraction for two processes P_i and P_k, whereas all other processes can be "existentially" abstracted.

As for (S1), we observe the global memory and buffer $\mathbf{In}_i$, for process P_i we observe the cache $\mathbf{C}_i$, but neither $\mathbf{Out}_i$ nor $\mathbf{E}_i$ as $(a,d) \notin elem_i$; for process P_k, we observe $\mathbf{E}_k$ and $\mathbf{Out}_k$, but neither $\mathbf{C}_k$ nor $\mathbf{In}_k$, as we are not interested in the events depending on the values of these variables. We could also existentially abstract from $\mathbf{E}_k$, but we use a nonexistential abstraction of this variable, as this allows us to reuse the definition of this abstract process for the verification of other properties.

The abstract system for the verification of property (S2) is completely defined by the abstract element type $elem^2_A$ defined by

$abs\_elem = \{1\}$
$\forall e_A, e_A{}' \in elem^2_A \ . \ same\_addr(e_A, e_A{}') = true$
$\forall j \in index \forall e_A \in elem^2_A \ . \ dat_j(e_A) = (j = k) \vee (e_A = 0)$

and by the declaration parts of all abstract processes.

Name :	P^2_{iA}	
Variables :	abstract input :	$\mathbf{e} : elem^2_A$
	local :	$\mathbf{C}_i : set$ of abs_elem
	shared :	$\mathbf{M} : set$ of abs_elem
		$\mathbf{In}_i : buffer^1_A$ of $(abs\_elem \times Bool)$

Name :	P^2_{kA}	
Variables :	abstract input :	$\mathbf{e} : elem^2_A$
	local :	$\mathbf{E}_k : set$ of abs_elem
		$\mathbf{Out}_k : buffer^1_A$ of abs_elem
	shared :	$\mathbf{M} : set$ of abs_elem
		$\mathbf{In}_i : buffer^1_A$ of $(abs\_elem \times Bool)$

Name :	$P^{2,ex}_{jA}$ for all indices $j \notin \{i,k\}$	
Variables :	abstract input :	$\mathbf{e} : elem^2_A$
	shared:	$\mathbf{M} : set$ of abs_elem
		$\mathbf{In}_i : buffer^1_A$ of $(abs\_elem \times Bool)$

P^2_{iA} is like P^1_{iA} where the predicate defining action for $write_i(\mathbf{e})$ is replaced by $true$ and that of action $mw_i(\mathbf{e})$ by $dat_i(\mathbf{e}) \wedge unch(..)$, whereas P^2_{kA} is like P^1_{iA} where the actions $read_k$, cu_k, mr_k and cl_k are simplified.

The processes $P^{2,ex}_{jA}$ define almost the same process as $P^{1,ex}_A$, and as before, the abstract system defined by $P^2_{iA} \parallel P^2_{kA} \parallel P^{2,ex}_{jA}$ defines an abstraction of a concrete system with an arbitrary number of processes.

This abstract system allows to verify property (S2) for $k \neq i$; for $k = i$, it can be verified on the abstract system constructed for the verification of (S1).

Abstract systems for properties (S3) and (C2): Property (S3) expresses due to (C3) that the $write_k$ events become available in any process P_i in an order compatible with their occurrence. For its verification, we need to observe events concerning two pairs $(a_1, d_1), (a_2, d_2) \in elem_k$. We can use the almost the same abstract system as for the verification of property (S2); the only difference is that we use a different abstract element type $elem_A^{3a}$ defined by

$abs\_elem = \{1, 2\}$
$\forall j \in index \,.\, dat_j(e_A) = (j = k) \;\vee\; (e_A = 0)$
$same\_addr(e_A, e_A') = (e_A = 0) \vee (e_A' = 0) \vee (e_A = e_A')$

That means, we consider abstract elements related to concrete pairs with different addresses written by the same P_k. In order to verify property (S3) also for pairs with the same address we need also the abstract system for type $elem_A^{3b}$ which is as $elem_A^{3a}$ except that *same_addr* is defined by

$same\_addr(e_A, e_A') = true.$

These two abstract systems allow to verify property (S3) for any $k \neq i$. In order to verify it also for $k = i$, we can use the same abstract system as defined for the verification of property (S1) where we define $\mathbf{e}$ of type $elem_A^{3a}$, respectively $elem_A^{3b}$. That means that for the exhaustive verification of property (S3), we need four different abstract systems.

The abstract systems defined for types $elem_A^{3b}$, that is for a set *e_obs* containing elements with the same address can also be used to verify property (C2).

Abstract systems for property (S4): In presence of properties (C3) and (S3), property (S4) expresses that all *write* events issued by two different processes P_{k_1} and P_{k_2}, become available in any two processes P_{i_1} and P_{i_2} in a compatible order. For its verification we observe events concerning two pairs $(a_1, d_1) \in elem_{k_1}$ and $(a_2, d_2) \in elem_{k_2}$ where $k_1 \neq k_2$. We define types $elem_A^{4a}$ (and $elem_A^{4b}$) differing from the types $elem_A^{3a}$ ($elem_A^{3b}$) only by the definition of the predicates dat_j:

$\forall j \in index \,.\, dat_j(e_A) = (j = k_1) \wedge (e_A = 1) \vee (j = k_2) \wedge (e_A = 2) \vee (e_A = 0)$

We define a system where four processes are not existentially abstracted; processes $P^4_{k_1 A}, P^4_{k_2 A}$ are defined exactly as P^2_{kA} and processes $P^4_{i_1 A}, P^4_{i_2 A}$ are defined as P^2_{iA} except that variable $\mathbf{e}$ is of type $elem_A^{4a}$ (respectively $elem_A^{4b}$).

This allows to verify property (S4) if the indices k_1, k_2, i_1, i_2 are all different for the two cases where that the two observed data element have the same address or not. We must also verify (S4) in the cases $k_1 = i_1$ and/or $k_2 = i_2$. For this, we need abstract systems in which we replace the pair of processes $(P^4_{k_1 A}, P^4_{i_1 A})$ (and/or $(P^4_{k_2 A}, P^4_{i_2 A})$) by a single process $P^4_{i_1 A}$ (and/or $P^4_{i_2 A}$) which are like P^1_{iA} except that variable $\mathbf{e}$ is of different type; that means that for the exhaustive verification of property (S4) we need six different abstract systems.

Here, we have defined for each property the smallest (most abstract) systems — with respect to the predefined abstract types and operations — that still allows to verify it. The systems defined for the verification of (S3) and those for the verification of (S4) are uncomparable (in the sense of abstraction), and are therefore all necessary — at least without using additional symmetry arguments allowing to eliminate some of them. However, all the abstract systems defined for the verification of (S1) and (S2) are abstractions of one of the systems defined for the verification of (S3) or (S4) and need not to be built.

Verification of properties on abstract systems using CAESAR/ALDEBARAN *:*
By Proposition 2, the satisfaction of the properties of Proposition 3 of one of the abstract systems allows to deduce their satisfaction on the given concrete system if we can show consistency for all atomic propositions used non negated in the positive normal form of the formulas expressing these properties. In the positive normal forms occur only predicates of the form $after(\ell)$ and $avail(a,d)$ non negated for which consistency is obvious.

That means that the above defined abstract systems allow to verify sequential consistency for the particular system in which the action $mr_i(a,d)$ is only allowed if (a,d) is not yet somewhere in $\mathbf{In}_i$. In order to verify the system without this restriction, we need a more complex abstraction for variables $\mathbf{In}_i$: without this restriction, $\mathbf{In}_i$ may contain for any pair (a,d) an arbitrary number of triples of the form $((a,d), \mathit{false})$ which means that using the above defined abstraction relations, we cannot choose a finite K without losing the satisfaction of properties we are interested in. But even if we let $(i, false)$ to represent an arbitrary number of consecutive occurrences of $((a,d), \mathit{false})$ in the concrete buffer restricted to elements in e_obs, this is not sufficient: there may be arbitrary alternations of occurrences of different triples with boolean parameter *false*. However, what we need for the verification of the above properties, is that $((a,d), \mathit{true})$ occurs in $\mathbf{In}_i$ always before $((a,d), \mathit{false})$, and also that an arbitrary amount of elements $((a,d), \mathit{false})$ after $((a,d), \mathit{true})$ cannot falsify the properties. That means, we can use exactly the same abstract type as before, but use a different abstraction function relating the concrete and the abstract type, and consequently, different abstract predicates $append_A^{\times Bool,K,alt}$, $tail_A^{\times B,K,alt}$, ...

$$\forall B \in buffer \text{ of } elem \times Bool \; .$$
$$\varrho_{buf \times Bool}^{e\_obs,K,alt}(B) = \begin{cases} \epsilon & \text{if } length(Obs) = 0 \\ \varrho_{elem}^{e\_obs}(Obs) & \text{if } 1 \leq length(Obs) \leq K \\ \bot & \text{if } length(Obs) > K \end{cases}$$

where Obs is $B_{|e\_obs \times \{\mathit{true}\} \cup \{(e,\mathit{false}) \;|\; (e \in e\_obs) \wedge (e,\mathit{true}) \notin B)\}}$ where furthermore all occurrences but the first one of elements of the form (e, false) are eliminated. The corresponding abstract predicates for $first$, *empty* and *empty_true* are unchanged, the abstract predicate for *append* can be defined by

$$append_A^{\times Bool,K,alt}(B,(e,b),B') = \begin{cases} b = \mathit{true} \wedge append_A^{\times Bool,K}(B,(e,b),B') \; \vee \\ b = \mathit{false} \wedge \exists b'.isin(B,(e,b')) \wedge B = B' \; \vee \\ b = \mathit{false} \wedge \; \nexists b'.isin(B,(e,b')) \wedge B' = (e, false) \bullet B \end{cases}$$

The abstract predicate for *tail* applied to a pair (e, true) eliminates this element, but it may also insert (e, false) at any position in B.

We have used the tool CAESAR/ALDEBARAN[FGM+92] in order to build all the necessary abstract systems and to verify the properties on them. CAESAR/ALDEBARAN verifies systems described in LOTOS[BB88]. In LOTOS, data types and all operations and predicates on them are described in form of abstract data types, whereas the control part is described by a process algebra term. LOTOS allows only local variables, but has a very powerful notion of synchronization by means of rendez-vous, allowing exchange of and agreement on values between an arbitrary number of processes: we define an additional process *MEMORY* that synchronizes with process P_j on the events mw_j and mr_j and updates the global memory $\mathbf{M}$. All processes synchronize on all events mw_i and each process updates its own local variable $\mathbf{In}$ by appending the right pair (e, b). All other actions are local to some process. This allows to define easily all the necessary abstract systems by modifying the type definitions of the concrete system given as a LOTOS program:

- For efficiency reasons, we use instead of a single process type P as in the concrete system, four different process types corresponding to
  - process P^1_{iA} with all variables, which is almost identical to the concrete process P
  - process P^2_{iA}, without variables $\mathbf{E}$ and $\mathbf{Out}$,
  - process P^2_{kA} without variables $\mathbf{C}$ and $\mathbf{In}$, and finally
  - process P^{ex}_{jA} which has only input variables

  All these process types are obtained by simplifying the concrete process type by eliminating all the predicates depending only on eliminated variables.
- We define a type $elem_A$ for each abstract element type defined earlier in this section. It includes also the definition of the predicates *same_addr* and dat_j and of the constant K.
- For the abstract memories, sets, and buffers — which are parameterized by the type of elements they can contain — we need a single definition (for each corresponding concrete type) which is also parametrized just by the type of elements it can contain. In LOTOS, type definitions include also the definitions of all associated predicates by means of sets of conditional equations. The abstract predicates are in general obtained from the corresponding concrete one by adding equations concerning the special values, such as abstract input 0 or abstract buffer $\perp$.

  The definition of abstract operations in terms of abstract data types makes the proof that they are abstractions of the concrete operations very easy.

Verification of Property (C3): As we have already mentioned, our verification does in general not allow to verify liveness properties directly: there exists no finite abstraction of the cache memory system that verifies (C3). Under the hypothesis that the system is fair with respect to the events mw_i and cu_i — a hypothesis that is made in the original description in [ABM93] — one can deduce (C3) due to the proof rules given in [JPR94] from the satisfaction of the following safety properties. Notice that these proof rules are given for a linear framework, but its adaptation to the branching time framework is straightforward.

- $after(write_i(a,d)) \Rightarrow in(\mathbf{Out}_i, (a,d))$
- $position(\mathbf{Out}_i, 1, (a',d')) \Rightarrow enable(mw_i(a',d'))$
- $\forall n > 1 \,.\, position(\mathbf{Out}_i, n, (a,d)) \wedge enable(mw_i(a',d')) \Rightarrow$
  $\mathbf{AX}(position(\mathbf{Out}_i, n, (a,d)) \wedge enable(mw_i(a',d')) \vee$
  $after(mw_i(a',d')) \wedge position(\mathbf{Out}_i, n-1, (a,d)))$
- $enable(mw_i(a,d)) \Rightarrow \mathbf{AX}(enable(mw_i(a,d)) \vee after(mw_i(a,d)) \wedge in(\mathbf{In}_j, (a,d))$
- $position(\mathbf{In}_i, 1, (a',d')) \Rightarrow enable(cu_i(a',d'))$
- $\forall n > 1 \,.\, position(\mathbf{In}_j, n, (a,d)) \wedge enable(cu_i(a',d')) \Rightarrow$
  $\mathbf{AX}(position(\mathbf{In}_j, n, (a,d)) \wedge enable(cu_i(a',d')) \vee$
  $after(cu_i) \wedge position(\mathbf{In}_j, n-1, (a,d)))$

- $enable(cu_j(a,d)) \;\Rightarrow\; \mathbf{AX}(enable(cu_j(a,d)) \vee after(cu_j(a,d)) \wedge avail(a,d))$

where *in* and *position* are predicates with obvious meanings.

All these safety properties can be verified using finite abstractions.

5 Discussion

What has been achieved? A first impression could be that this verification of a cache memory looks much like a handwritten proof. However, it is quite different: starting right from the beginning, it is in fact rather lengthy to define all the abstract types, abstraction relations and corresponding abstract predicates, even in order to verify a trivial buffer program. However, having done this once, in order to verify the much more complex cache memory system, we can reuse these definitions — for some of them by means of slight modifications — and have to come up with a few new definitions concerning the data type *memory* that was not used in the buffer program, Also, the definitions concerning abstract memories are already much easier to obtain using analogous reasonings. In fact, there are many examples of systems, for which we have to verify similar properties and which use similar data structures and operations on them, such that the same (or at least similar) abstract types and operations can be used. The abstract sets, buffers and memories given here are certainly not sufficient to build convenient abstractions for any system involving these data types but in many cases, the convenient abstractions can be obtained by slight modifications of the abstractions used here. In any case, it should be very useful to collect such definitions in a "library". A similar approach has been followed by P. and R. Cousot and more recently by D. Long concerning abstractions of integers and operations on them. In [DF95] a very interesting extension of our method has been proposed which allows to avoid to restart the whole process again if a property does *not* hold using the initially used abstract definitions.

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, depending on the particular system under study, as this has been proposed, e. g. by B. Kurshan [Kur94].

For the verification of the cache memory, an additional complexity comes from the fact that we also have to define the set of formulas to be verified as the original abstract specification is not given in these terms. We believe that this set of properties is interesting by itself as it can be used for the verification of other systems supposed to implement sequentially consistent memories. The advantage of this characterization is also that it can easily be modified in order to obtain weaker or stronger specifications which are frequently used in real implementations. This adaptability implies also that the fact that our characterization is slightly stronger than required is not a problem.

Another point which makes an abstract specification given as a set of properties so attractive, is the fact that the modification of a single property does not require to redo the whole verification process. Our method is also incremental with respect to modifications of the program, as long as they allow to use the same or at least very similar abstract types and operations, as we have seen when we modified the action $mr_i(a,d)$ in the cache memory. 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 already identical to the previous one, only the part of the verification process that can be automatized, i.e. the reconstruction of a model and the verification of the properties on it, must be redone.

Note at the moment of edition: time passing showed that the general approach presented is very useful in different domains. Since the development of tools like the Invariant Checker[GS97] and InVesT[BLO98] the kind of abstractions used without formal proofs in this paper, can be *computed algorithmically* just from the specification of the finite abstract domain and the abstraction relation ϱ. Also the use of the logical characterization of sequential consistency turned out to be very useful as it allows the use of very small abstract domains.

Acknowledgements: I would like to thank the anonymous referees for pointing out several errors, in particular the fact that the initial characterization of sequential consistency was not sufficient, Amir Pnueli for giving me some ideas how to get a satisfactory solution and Dennis Dams and Joseph Sifakis for many fruitful discussions.

References

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

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *ISDN*, 14(1):25–29, 1988.

[BBLS92] A. Bouajjani, S. Bensalem, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Workshop on Computer-Aided Verification (CAV), Montréal.* LNCS 630, 1992.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proceedings of CAV'98*, volume 1427 of *LNCS*, June 1998.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, 1977.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[Cri95] R. Cridlig. Semantic Analysis of Shared-Memory Concurrent Languages using Abstract Model-Checking. In *Symposium on Partial Evaluation and Program Manipulation*, La Jolla, California, June 1995.

[DF95] J. Dingel and Th. Filkorn. Model checking for infinite state systems using data abstraction, assumption-committment style reasoning and theorem proving. In *Proc. of 7th CAV 95, Liège.* LNCS 939, Springer Verlag, 1995.

[EH83] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: On branching versus linear time. In *10th ACM Symposium on Principles of Programming Languages (POPL 83)*, 1983. also in Journal of ACM , 33:151-178.

[FGM$^+$92] J.Cl. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A tool box for the verification of lotos programs. In *14th International Conference on software Engineering*, 1992.

[Ger94] R. Gerth. Introduction to sequential consistency and the lazy caching algorithm, 1994. same volume.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Conference on Computer Aided Verification CAV 93, Heraklion Crete.* LNCS 697, Springer Verlag, 1993.

[Gra94] S. Graf. Verification of a distributed cache memory by using abstractions. In *Conf. on Computer Aided Verification CAV'94, Stanford.* LNCS 818, Springer Verlag, 1994.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97, Haifa*, volume 1254 of *LNCS*, June 1997.

[JPR94] B. Jonsson, A. Pnueli, and C. Rump. Proving refinement using transduction, 1994. same volume.

[Koz83] D. Kozen. Results on the propositional μ-calculus. In *Theoretical Computer Science.* North-Holland, 1983.

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

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating processes, the automata theoretic approach.* Princeton Series in Computer Science. Princeton University Press, 1994.

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

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

[LGS$^+$94] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, Vol. 6, Iss 1, January 1995.

[Loi94] C. Loiseaux. *Vérification symbolique de programmes réactifs à l'aide d'abstractions.* PhD thesis, February 1994.

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

[MP91] Z. Manna,A. Pnueli. The temporal Logic of reactive and concurrent systems, Volume 1: Specification. Springer Verlag, 1991.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. Second Int. Joint Conf. on Artificial Intelligence*, pages 481–489. BCS, 1971.

[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, Springer Verlag, 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 *Conference on Automatic Verification (CAV), Rutgers, NJ.* LNCS 531, Springer Verlag, 1990.