Asynchronous Construction of Consistent Global Snapshots in the Object and Action Model

Islene Calciolari Garcia
Luiz Eduardo Buzato

Relatório Técnico IC–98-16

Abril de 1998

Asynchronous Construction of Consistent Global Snapshots in the Object and Action Model*

Islene Calciolari Garcia
Luiz Eduardo Buzato

Abstract

The Object and Action Model (OAM) is well-known as an adequate paradigm to build fault-tolerant configurable distributed applications. The reconfiguration of an application depends on the construction of a consistent global snapshot of its global state. An atomic action that reads the states of all objects of the application is a simple and straightforward way to obtain such global snapshot, but reduces concurrency and interferes with the underlying computation. In the Process and Message Model (PMM) consistent snapshots can be constructed asynchronously by a component that passively receives process states. This paper presents OAM-based asynchronous global snapshot algorithms equivalent to PMM-based algorithms, built using a precedence relation defined for atomic actions. Arjuna, an object-oriented action-based distributed programming environment, has been used to implement these OAM-based global snapshot algorithms, allowing us to conclude that our approach is promising.

1 Introduction

The Object and Action Model (OAM) is well-known as an adequate paradigm to build fault-tolerant configurable distributed applications [3, 17]. In order to find out when an application can be reconfigured a consistent global snapshot [6] of its global state must be constructed. Informally, a global snapshot is "consistent" if it looks to the objects of the application as if it was taken at the same instant everywhere in the system. In the OAM, a straightforward way to obtain such a snapshot is to use an atomic action to read the states of all objects of the application, or at least of those that are relevant to the dynamic reconfiguration of the system. This synchronous solution, although very simple, reduces concurrency and interferes with the underlying computation. In contrast, in the Process and Message Model (PMM) consistent snapshots can be constructed asynchronously by a component that passively receives process states [1, 2, 13]. Fischer, Griffeth and Lynch [8] designed a global snapshot algorithm that is tailored for transaction-based systems, but

*Institute of Computing, UNICAMP, Caixa Postal 6176, 13083-970 Campinas, SP. This work has been supported by FAPESP under grant no. 95/1983-8 for Islene Calciolari Garcia and grant no. 96/1532-9 for the Laboratory of Distributed Systems. This paper was accepted for publication in the Proceedings of the 4th International Conference on Configurable Distributed Systems, May 4-6, 1998, Annapolis, Maryland, USA.

their solution requires the duplication of the atomic actions of the application, the duplicate actions are used to obtain the global snapshots. In this paper we show algorithms for global snapshot that do not require such duplication.

In this work we have used the duality of OAM and PCM [15] as a starting point for a detailed study of the precedence relations used to build consistent global snapshot algorithms in the OAM and in the PMM. The study of the duality OAM-PCM motivated us to explore the correlations between control and communication mechanisms found in OAM and PMM. The study of the three models can yield interesting mappings between algorithms developed for each of the models. This paper presents the *correlations* we have devised for the precedence relations as a result of our study, it also presents OAM-based asynchronous global snapshot algorithms equivalent to PMM-based algorithms. The OAM-based algorithms were obtained through the application of our correlation to well-known PMM-based global snapshot algorithms.

The paper is structured as follows. Section 2 introduces the object and action model (OAM). Section 3 introduces the concepts necessary to define the notion of asynchronous consistent global snapshots in the OAM. Section 4 focus on the construction of the correlation between OAM and PMM. Section 5 is devoted to the description of OAM-based consistent global snapshot algorithms. Section 6 brings a concise view of the object-oriented distributed reconfiguration environment we have used to test the algorithms. Section 7 brings some concluding remarks and considers future research.

2 Object and action model (OAM)

A distributed application in the OAM is defined as a set of n objects $(\mathsf{o}_1, \ldots, \mathsf{o}_n)$ that cooperate to compute and maintain relevant data. Every application object o_i is an instance of an abstract data type and has an internal state σ_i that represents the value of its attributes and relationships with other objects of the application and/or environment where it is embedded.

In the OAM, atomic actions are used to structure the computation of the application. The ACID properties of atomic actions (atomicity, consistency, isolation, and durability) provide a framework for building fault-tolerant applications. These properties allow the integrity of objects to be maintained in the presence of failures such as node crashes—fail silent nodes—and message loss [15, 17]. The implementation of the atomic action mechanisms imply that objects must be controlled by an action manager m_i that assigns read/write locks to objects.

We assume that atomic actions are organized according to *action structures* [16] determined by the application. The basic action structure is a single atomic action. More complex action structures can be built using sequential and concurrent combinations of action structures. Atomic actions can be nested [12], in the sense that they can also be composed of atomic actions. Nested atomic actions increase the potential for concurrency, making it possible to execute inner actions concurrently, it also makes crash recovery easier [15]. Action structures can be further extended through the use of nested actions.

The action manager ensures that locks are assigned only to objects that are consistent

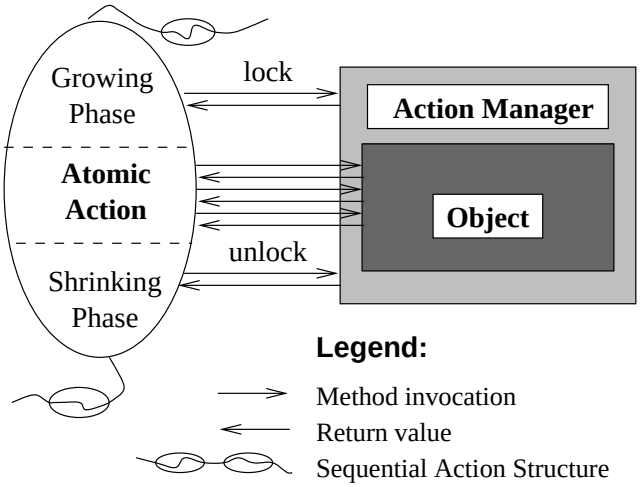

Figure 1: Action manager protocol

with respect to atomic actions. Usually, we say that locks are assigned to objects, but in this text, to avoid ambiguity, we say that locks are assigned to object states. Figure 1 shows the interaction pattern between an atomic action of a sequential action structure and the action manager of an application object. In this paper, we consider object states visible only after the commit of top-level actions.

Action structures notation

This Section introduces the notation used throughout the paper to present the correlation between OAM and PMM, and global snapshot algorithms.

We enumerate object states visible after commit of atomic actions, the enumeration creates a *state history* of an object, say o_i, $h_i = (\sigma_i^0, \sigma_i^1, \ldots)$.

An atomic action a acquires read/write locks on a set of object states during its execution. We define a function *Action* that when applied to object states returns the action that promoted the object to that state. We define the following functions to refer to subsets of object states:

- $LockR(\mathsf{a})$ — object states read-locked by a;
- $LockW(\mathsf{a})$ — object states write-locked by a;
- $LockRW(\mathsf{a})$ — $LockR(\mathsf{a}) \cup LockW(\mathsf{a})$;
- $Out(\mathsf{a})$ — object states promoted by a.

The execution of an application is represented by a diagram similar to the space-time diagram used traditionally in the PMM [1, 13] (Figure 2). In the diagram, horizontal lines represent sequential action structures with time flowing from left to right. Parallel horizontal lines represent concurrent action structures, which may fork or join depending on the global

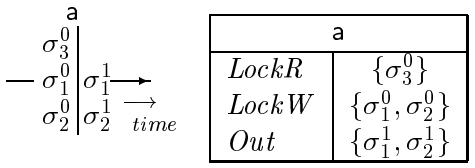

Figure 2: An execution diagram for the OAM

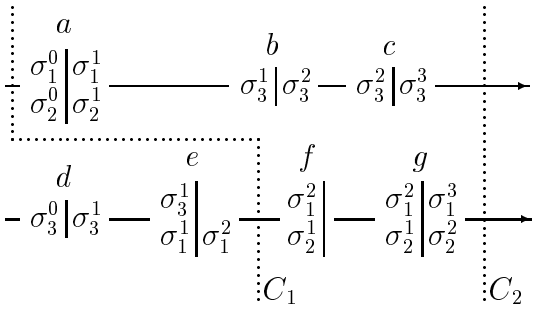

Figure 3: A scenario in the OAM.

action structure determined by the application. Atomic actions are represented by vertical bars. Object states that appear only at the left of a vertical bar (atomic action) represent read-locked objects, whereas object states that appear on both sides of the line represent objects write-locked by the atomic action.

3 Consistent global snapshots

The definition of consistent global snapshots for the OAM requires the mapping of the notion of casual precedence and consistent cuts from the PMM to their OAM equivalents. These concepts, once re-interpreted from within the OAM, are going to serve as the building blocks of the *correlation* used to map PMM-based asynchronous global snapshot algorithms into their OAM equivalents.

Structural precedence An action structure is a graph where nodes are atomic actions and two pair of nodes, say a and a', are joined by an edge if a must happen before a', as determined by the application semantics. An atomic action a precedes structurally a' if there is a path from a to a' in the graph. Two actions, say a and a', maintain a direct structural precedence whena precedes structurally a' and there is no other atomic action, say a'', so that, a precedes structurally a' and a' precedes structurally a''.

Dynamic precedence We establish a dynamic precedence relation between atomic actions that is analogous to the *happens before* binary relation defined by Lamport [11]; henceforth abbreviate as *precedence*.

Definition 3.1 *Precedence between atomic actions:* $\mathsf{a} \prec \mathsf{a}'$

1. $\mathsf{a} \prec_S \mathsf{a}'$, *or*
2. $Out(\mathsf{a}) \cap LockRW(\mathsf{a}) \neq \emptyset$, *or*
3. $LockR(\mathsf{a}) \cap LockW(\mathsf{a}) \neq \emptyset$, *or*
4. $\exists\, \mathsf{a}'' : (\mathsf{a} \prec \mathsf{a}'') \wedge (\mathsf{a}'' \prec \mathsf{a}')$.

The first part of the definition covers the case of a possible causal relationthip between atomic actions, as implied by the action structure. As an example, consider atomic actions a and b (Figure 3); although they involve disjunct sets of objects, their structural precedence may relate them casually.

The second part deals with a situation where an action reads an object state produced by another atomic action. In this case, we say that the former action precedes the latter. In Figure 3, action e reads state σ_1^1 produced by action a, so we can conclude that $a \prec e$.

The third part addresses the case when an action reads an object state promoted (written) by another action, the first one precedes the second one. In Figure 3, action e reads state σ_3^1 before action b promotes o_3 to σ_3^2. Thus, $e \prec b$, although there is no causal relationship between e and b.

The last part covers transitive precedence. In Figure 3, we can infer that $e \prec b$, and $b \prec c$, thus $e \prec c$.

Definition 3.2 *Concurrence between atomic actions:* $\mathsf{a} \parallel \mathsf{a}'$

- $(\mathsf{a} \not\prec \mathsf{a}') \wedge (\mathsf{a}' \not\prec \mathsf{a})$

We say that two atomic actions are concurrent if there are no precedence relations between them. In Figure 3 we can see that $a \| d$ and $b \| g$.

3.1 Consistent cuts

A cut C is the set of atomic actions formed by considering initial prefixes of action structures:

$$(\mathsf{a} \in C) \wedge (\mathsf{a}' \prec_S \mathsf{a}) \Rightarrow (\mathsf{a}' \in C).$$

A cut C is consistent if, for every pair of actions, the following relation is valid:

$$(\mathsf{a} \in C) \wedge (\mathsf{a}' \prec \mathsf{a}) \Rightarrow (\mathsf{a}' \in C).$$

In Figure 3, cuts are represented by dotted lines. Cut C_1 is inconsistent because $(e \in C_1)$ and $(a \prec e)$, but $(a \notin C_1)$; C_2 is a consistent cut.

We can associate a global state $\Sigma = (\sigma_1^{\iota_1}, \ldots, \sigma_n^{\iota_n})$ to a cut C using the following rules:

$$\forall i \in \{1, \ldots, n\}, \forall\, \mathsf{a} \in C :$$
$$\iota_i = max(\alpha : \sigma_i^\alpha \in (Out(\mathsf{a}) \cup LockR(\mathsf{a})))$$

using $\iota_i = 0$, if the set is empty. In Figure 3 we have $\Sigma_1 = (\sigma_1^2, \sigma_2^0, \sigma_3^1)$ associated to C_1 and $\Sigma_2 = (\sigma_1^3, \sigma_2^2, \sigma_3^3)$ associated to C_2.

Every cut is associated with only one global state, while the converse may not hold, since it is possible that atomic actions acquire read locks only. A global state Σ is guaranteed to be consistent if, and only if, it is associated with a consistent cut.

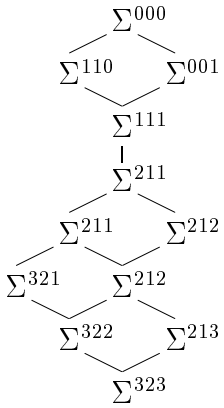

Figure 4: Lattice for the OAM scenario

3.2 Lattice of global states

The lattice of global states of an application represents the set of all sequential executions that conform to the dynamic precedence relation. Figure 4 is a graphical representation of the lattice of global states associated with Figure 3. The execution of each atomic action is represented by $\Sigma^{\iota_1\iota_2\iota_3}$ where $\iota_1\iota_2\iota_3$ represents the global state associated with objects o_1 to o_3.

The use of global state lattices can be useful in two situtations: (i) verifying a posteriori that the dynamic reconfiguration procedure has been applied and the maximum/minimum number of atomic ations executed until the reconfiguration action was executed, (ii) there are predicates whose validity cannot be asserted unless the lattice of global states of an application is constructed. For example, when debugging a system we may wish to monitor de lengths of two queues, and notify the user if the sum of the lengths is larger than some threshold. If both queues are dynamically changing, the predicate corresponding to the desired condition is not stable. In this case, the only solutioin is to evaluate the predicate using the lattice of global states [1].

4 Flow of precedence data in the OAM

In the PMM, the channels used to transmit the information (data) necessary to construct global states are maintained by each process of the application and exchanged through asynchronous message passing. This Section analyses objects and actions to determine where and how the transmission of the precedence data necessary to build global states occurs in the OAM.

In the OAM, the communication channels available for the transmission of data between objects are object states and action structures (application). Unfortunately, these channels do not allow the exchange of all precedence information needed to build consistent global snapshots. Definition 3.1, part 3, allows the construction execution scenarios where

Begin	$\mathcal{R} \leftarrow \emptyset$; $\mathcal{W} \leftarrow \emptyset$; $\phi(\mathsf{a}) \leftarrow \max(\phi(\mathsf{a}' \mid \mathsf{a}' \prec_S \mathsf{a}))$;
Read_lock(i)	$\phi(\mathsf{a}) \leftarrow \max(\phi(\mathsf{o}_i), \phi(\mathsf{a}))$; $\mathcal{R} \leftarrow \mathcal{R} \cup \{i\}$;
Write_lock(i)	$\phi(\mathsf{a}) \leftarrow \max(\phi(\mathsf{m}_i), \phi(\mathsf{a}))$; $\mathcal{W} \leftarrow \mathcal{W} \cup \{i\}$;
Commit	if $\mathcal{W} \neq \emptyset$ $\phi(\mathsf{a}) \leftarrow \mathrm{inc}(\phi(\mathsf{a}))$; $\forall i \in \mathcal{W}$ do $\phi(\mathsf{o}_i) \leftarrow \phi(\mathsf{a})$;
Unlock(i)$_{i \in \mathcal{R}}$	$\phi(\mathsf{m}_i) \leftarrow \max(\phi(\mathsf{m}_i), \phi(\mathsf{a}))$;
Unlock(i)$_{i \in \mathcal{W}}$	$\phi(\mathsf{m}_i) \leftarrow \phi(\mathsf{a})$;

Figure 5: Flow of precedence data in the OAM

an action precedes another but communicating the precedence data is impossible, if only the above mentioned channels are available. To fix this problem, we add communication channels to the action managers (Theorem 4.1).

The algorithm depicted in Figure 5 shows the modifications made to the atomic action primitives so that information is transmitted in accordance with the precedence relations. Actions, objects and action managers keep precedence data, denoted by $\phi(\mathsf{a})$, $\phi(\mathsf{o})$ and $\phi(\mathsf{m})$. Additionally, $\phi(\sigma)$ denotes the value of $\phi(\mathsf{o})$ in state σ. Furthermore the precedence data, each atomic action maintains two auxiliary variables, say $\mathcal{R}$ and $\mathcal{W}$, that accumulate the object identifications (OIDs) of objects locked during their execution. Precedence data percolates through the action structures. Action managers and atomic actions exchange precedence data by piggybacking them in the invocations of lock and unlock operations. Finally, atomic actions propagate precedence data through object states.

The following theorem proves that our algorithm is consistent with the precedence relation between atomic actions.

Theorem 4.1 *Let* a *and* a' *be atomic actions and* ϕ *be precedence data propagated using the algorithm depicted in Figure 5. In this case, we have:*

$$\mathsf{a} \prec \mathsf{a}' \Rightarrow \phi(\mathsf{a}) \leq \phi(\mathsf{a}')$$

Proof When the precedence relation ($\mathsf{a} \prec \mathsf{a}'$) holds, it can be expanded as:

$$\mathsf{a} = \mathsf{a}_0 \prec \mathsf{a}_1 \prec \cdots \prec \mathsf{a}_p = \mathsf{a}' \quad p \geq 1$$

where each relationship $\mathsf{a}_k \prec \mathsf{a}_{k+1}$ $(0 \leq k < p)$ is due to (1), (2) or (3) (Definition 3.1). We prove by induction on the size of the sequence that $\phi(\mathsf{a}) \leq \phi(\mathsf{a}')$.

Base $p = 1$

$$\mathsf{a} = \mathsf{a}_0 \prec \mathsf{a}_1 = \mathsf{a}'$$

We analyze cases (1), (2) and (3) of Definition 3.1:

1. $\mathsf{a}_0 \prec_S \mathsf{a}_1$

   In the Begin step, $\phi(\mathsf{a}_1)$ is initialized so that $\phi(\mathsf{a}_0) \leq \phi(\mathsf{a}_1)$. Next, the other steps apply to $\phi(\mathsf{a}_1)$ only an arbitrary number of maximum functions, and zero or one increment functions. Thus, $\phi(\mathsf{a}_0) \leq \phi(\mathsf{a}_1)$.

2. $Out(\mathsf{a}_0) \cap LockRW(\mathsf{a}_1) \neq \emptyset$

   For each object state σ_i a_1 reads from a_0, we know at the commit of a_0 that $\phi(\mathsf{o}_i) \leftarrow \phi(\mathsf{a}_0)$ (so, $\phi(\sigma_i) = \phi(\mathsf{a}_0)$) and $\phi(\mathsf{m}_i) \leftarrow \phi(\mathsf{a}_0)$. Besides, if $\sigma_i \in LockR(\mathsf{a}_1)$, then from Read_lock step we have $\phi(\sigma_i) \leq \phi(\mathsf{a}_1)$. Otherwise, $\sigma_i \in LockW(\mathsf{a}_1)$, and then from the Write_lock step we have $\phi(\mathsf{m}_i) \leq \phi(\mathsf{a}_1)$. In this case, $\phi(\mathsf{m}_i)$ might have been changed by the Unlock step of other atomic actions that read σ_i[1]. Nevertheless, even if it has actually changed, it can only have been assigned a greater value than $\phi(\mathsf{a}_0)$. Hence, $\phi(\mathsf{a}_0) \leq \phi(\mathsf{a}_1)$.

3. $LockR(\mathsf{a}_0) \cap LockW(\mathsf{a}_1) \neq \emptyset$

   For each σ_i read by a_0 and promoted by a_1, we know that $\phi(\mathsf{a}_0) \leq \phi(\mathsf{m}_i)$, if and only if a_0 has Commit'ted. Furthermore, from the Write_lock step of a_1, we know that the value of $\phi(\mathsf{m}_i)$ has been incorporated into $\phi(\mathsf{a}_1)$. Thus, $\phi(\mathsf{a}_0) \leq \phi(\mathsf{a}_1)$. In this case, because the value of $\phi(\mathsf{a}_1)$ is computed with the invocation of one or more maximums and, at last, one increment function, we have $\phi(\mathsf{a}_0) < \phi(\mathsf{a}_1)$.

Step Assume the hypothesis holds for a sequence of size p. We are going to prove it holds for a sequence of size $p+1$. We have:

$$\mathsf{a} = \mathsf{a}_0 \prec \mathsf{a}_1 \prec \cdots \prec \mathsf{a}_p \prec \mathsf{a}_{p+1} = \mathsf{a}' \text{ and } \phi(\mathsf{a}) \leq \phi(\mathsf{a}_p)$$

We have assumed $\mathsf{a}_p \prec \mathsf{a}_{p+1}$ not due to transitivity. We can, therefore, apply the base of the induction and conclude that $\phi(\mathsf{a}_p) \leq \phi(\mathsf{a}_{p+1})$. Hence, $\phi(\mathsf{a}) \leq \phi(\mathsf{a}')$.

Theorem 4.2 *Let σ and σ' be object states, and $Action(\sigma)$ and $Action(\sigma')$ the atomic actions that produced them. Let $\phi(\sigma)$ and $\phi(\sigma')$ be the control information associated with the object states, propagated using the algorithm depicted in Figure 5. In this case, we have:*

$$Action(\sigma) \prec Action(\sigma') \Rightarrow \phi(\sigma) < \phi(\sigma')$$

Proof We know from the proof of Theorem 4.1 that $\phi(\sigma) \leq \phi(\sigma')$. Additionally, from step (3) of the base of induction of Theorem 4.1, we have already shown that, when an action promotes a state, its precedence data is strictly greater than its predecessors. Thus, considering transitivity, we have $\phi(\sigma) = \phi(Action(\sigma)) < \phi(Action(\sigma')) = \phi(\sigma')$.

[1] Note that the algorithm guarantees that state of the manager $\phi(\mathsf{m}_i)$ remains monotonically crescent even in the presence of atomic actions that write-lock object o_i, since, in the Write_lock step, we can assure that $\phi(\mathsf{m}_i) \leq \phi(\mathsf{a})$ and, in the Unlock step, $\phi(\mathsf{m}_i)$ is assigned the value of $\phi(\mathsf{a})$, that may have been increased, but not decreased.

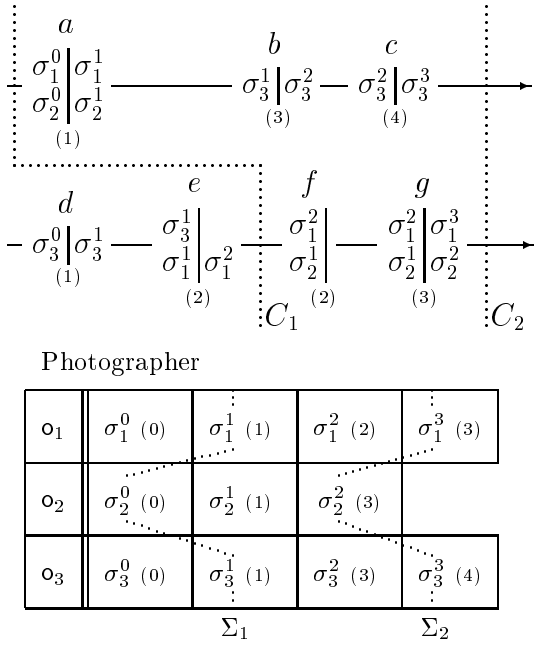

Figure 6: Time-stamps.

5 Snapshots algorithms

In this Section, we describe the software architecture of the monitoring system used to explain the OAM-based global snapshot algorithms of the Section: time-stamps and vector-clocks. These algorithms have been obtained through the application of the correlation between OAM and PMM (Section 1) and the algorithm for transmission of precedence data (Section 4) already described.

The external observer that builds consistent global states asynchronously is called the *photographer*. It receives object states from the application through reliable FIFO channels. The photographer is passive, that is, it waits for object states to be passed on to it instead of actively probing the application.

5.1 Time-stamps

The first algorithm to be mapped uses a natural number as precedence data. We denote this natural number by LC, in allusion to logical clocks [11]. The maximum and increment functions are used with their usual meaning.

In Figure 6, we present the same scenario introduced in Figure 3, adding time-stamps (in brackets) to atomic actions. We also add the photographer's view of each object, by showing the sequence of states and its time-stamps in each row of the table. Dotted lines indicate the global states Σ_1 and Σ_2 related, respectively, to cuts C_1 and C_2.

As in the PMM version, LC is consistent with the precedence relation but it does not characterize it [1, 13]. This means that the reciprocal of Theorem 4.2 is not always

valid, that is, if $\mathsf{LC}(\sigma) < \mathsf{LC}(\sigma')$ we can not derive whether $Action(\sigma) \prec Action(\sigma')$ or $Action(\sigma) \| Action(\sigma')$. In Figure 6, we can see that $b \prec c$ and $g \| c$ although $\mathsf{LC}(b) = \mathsf{LC}(g)$.

$$\mathsf{LC}(\sigma) < \mathsf{LC}(\sigma') \not\Rightarrow Action(\sigma) \prec Action(\sigma')$$

In the PMM, if two process states have the same LC, they are definitely concurrent. In OAM, however, they can be the result of the same atomic action or of concurrent ones. In Figure 6 we have that $\mathsf{LC}(\sigma_1^1) = \mathsf{LC}(\sigma_2^1) = \mathsf{LC}(\sigma_3^1)$ and σ_3^1 was promoted by an atomic action concurrent to the one that promoted σ_1^1 and σ_2^1. The photographer must have a conservative approach and incorporate all object states with the same LC in a single step to ensure that a consistent global snapshot is obtained.

Given a pair of object states $\sigma_a^\alpha, \sigma_b^\beta$ with $\mathsf{LC}(\sigma_a^\alpha) \leq \mathsf{LC}(\sigma_b^\beta)$, the photographer is unable to tell whether there exists some state σ_i^ι such that $\mathsf{LC}(\sigma_a^\alpha) \leq \mathsf{LC}(\sigma_i^\iota) \leq \mathsf{LC}(\sigma_b^\beta)$. This problem is analogous to the well-known PMM gap-detection problem [1]. In the PMM, the solution to this problem is possible only if we consider information in addition to the logical clock values: communication channels are FIFO. Once again, in the OAM, the photographer has to be conservative, by sometimes waiting for the next state of an object to be delivered before incorporating its previous state to a consistent global snapshot. The photographer uses the following formula to determine a consistent global snapshot Σ:

$$\forall \sigma_a^\alpha, \sigma_b^\beta \in \Sigma : \mathsf{LC}(\sigma_a^\alpha) \leq \mathsf{LC}(\sigma_b^\beta) \vee \mathsf{LC}(\sigma_a^\alpha) < \mathsf{LC}(\sigma_b^{\beta+1})$$

In the PMM, the speed of snapshots construction is limited by the slowest process. In absence of failures, we assume that a process stops execution just after notifying the photographer. In the OAM, however, it is acceptable that one object enters a state and remains in this state for a long time, possibly infinite, making it impossible for the photographer to build a new snapshot during this period. This fact makes the OAM-based time-stamp algorithm less attractive to build consistent global snapshots than in the PMM.

For an example, assume the photographer has received states σ_1^2 and σ_3^1, but not σ_2^1, as depicted by Σ_1. It will not be able to build a consistent global snapshot involving σ_1^2 and σ_3^1 before it receives σ_2^1. However, as soon as it receives σ_2^1, it can build a consistent global snapshot because all the logical clocks have the same value. On the other hand, when the photographer has $\sigma_1^3, \sigma_2^2, \sigma_3^3$, since $\mathsf{LC}(\sigma_1^3) = \mathsf{LC}(\sigma_2^2) = 3 < \mathsf{LC}(\sigma_3^3) = 4$, the photographer must wait for further information from both o_1 and o_2 in order to build consistent global snapshot Σ_2.

5.2 Vector clocks

The second algorithm uses precedence data that carries information about all the object states produced by preceding atomic actions. This information is organized as a vector clock VC, with one entry for each object of the application. We should redefine the maximum and increment functions in order to specialize the basic algorithm presented in Section 4.

$$\begin{gathered}\mathsf{VC} \leftarrow \max(\mathsf{VC}_a, \mathsf{VC}_b) \equiv \forall i : 1 \leq i \leq n : \\ \mathsf{VC}[i] \leftarrow \max(\mathsf{VC}_a[i], \mathsf{VC}_b[i]) \\ \mathrm{inc}(\mathsf{VC}) \equiv \forall i \in \mathcal{W} : \mathsf{VC}[i] \leftarrow \mathsf{VC}[i] + 1\end{gathered}$$

Figure 7 depicts the vector clocks assigned to atomic actions and object states (in square brackets), using a representation similar to the one of Figure 6.

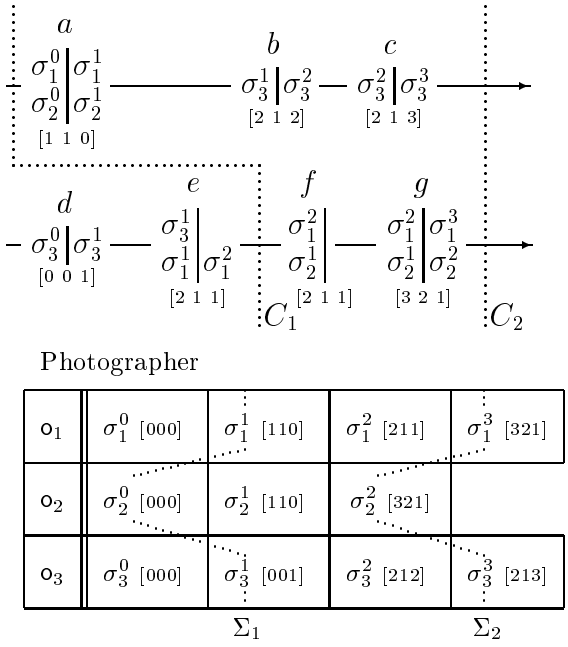

Figure 7: Vector clocks.

In the PMM version of the vector clock algorithm [1], we do not have the case where two process states have the same VC, since an event is related to just one process. In the OAM, however, all the object states that were produced by the same atomic action have the same VC, as σ_1^1 and σ_2^1 in Figure 7.

The information carried by vector clocks is sufficient to tell whether a pair of states was promoted in the same atomic action or in concurrent or preceding ones. Thus, the formula used by the photographer to verify if a snapshot is consistent does not need to involve any succeeding state.

$$\forall \sigma_a^\alpha, \sigma_b^\beta \in \Sigma : \mathsf{VC}(\sigma_a^\alpha)[b] \leq \beta$$

At global state Σ_1 (Figure 7), the photographer has σ_1^2 and σ_3^1, but not σ_2^1. It will be able to determine that this is not a consistent snapshot, because, comparing vector clocks of σ_1^1 and σ_2^0, it finds $\mathsf{VC}(\sigma_1^1)[2] = 1 > 0$. On the other hand, when the photographer has σ_1^3, σ_2^2 and σ_3^3, it will be able to build the consistent global snapshot Σ_2 immediately.

6 Dynamic reconfiguration system

From one perspective of dynamic reconfiguration, it is useful to abstract a distributed program as the superposition of two reactive programs: an application program and a monitoring program [3]. The application program implements functional aspects of the distributed program, the monitoring program implements the dynamic reconfiguration policies (Figure 8).

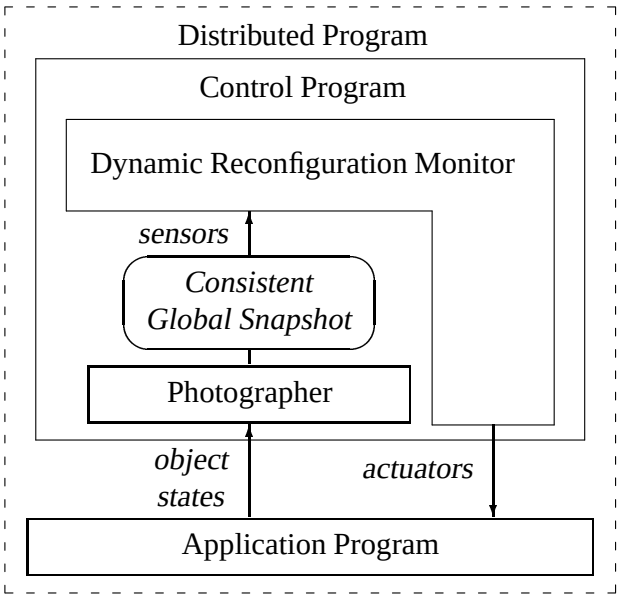

Figure 8: Monitoring system

Dynamic reconfiguration policies can be expressed as guarded actions (predicates) of the form *condition-action*, with conditions being implemented by methods called sensors. Actions are implemented by application methods called actuators (Figure 8). Consistent evaluation of guarded actions is meaningfull only if it takes place against consistent global snapshots of the state of the application objects [1]. In the architecture of the dynamic reconfiguration system the *photographer* is the component responsible for guaranteening that consistent global snapshots are available to the monitor (Figure 8).

From another perspective, the activity of the dynamic reconfiguration monitor can be seen as the processing of metainformation. The metainformation required for dynamic reconfiguration can be categorized as *structural* and *control* metainformation [3]. In an object-oriented system, classes and their relationships (inheritance, aggregation, and association) represent structural metainformation. Information concerning the states objects pass through during their execution is considered control metainformation; state transition diagrams can be used to represent this metainformation.

6.1 Experimental Environment

The dynamic reconfiguration system we have built derives its architecture from the perspectives drawn above and, thus, contains a subsystem for the management of each of the categories of metainformation. In the system, Stabilis (Figure 9) is responsible for the management of structural metainformation, and Vigil is responsible for the control metainformation. Stabilis and Vigil [3, 4] implement the mechanisms necessary to carry out dynamic reconfiguration of distributed applications based on object and atomic actions. Both systems have been built upon Arjuna [14], an object-oriented programming system which implements the mechanisms responsible for atomic actions, distribution, persistence, and

replication of objects (Figure 9). The programming language adopted by the systems is C++.

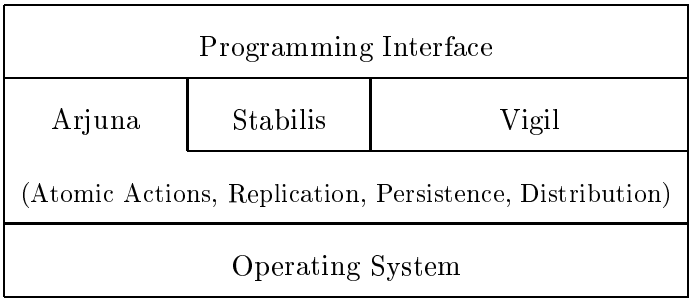

Figure 9: Dynamic reconfiguration system

Figure 10 shows how to build a distributed program using Stabilis and Vigil. The first step a programmer has to follow is the creation of an object model, divided in structural and control submodels. Statecharts [9] are used to represent control models. These submodels are captured and stored in repositories maintained by Stabilis and Vigil. This metainformation is used to generate part of the C++ code of both application and control programs. Subsequently, both programs are compiled and their object code is linked with code in the libraries of Arjuna, Stabilis, and Vigil (Figure 10). During the execution of the distributed program, the objects of the application program are constantly monitored by the objects of the control program. Changes in the state of the application program generate stimuli that trigger the execution of guarded actions, that is, reactions, in the monitor. Reactions cause further state changes in the application.

The architecture depicted in Figure 8 has been integrated into the dynamic reconfiguration system. The photographer implements the OAM-based algorithms described in this paper to build consistent global states using the object states received from the application. We have implemented the reliable channels between the photographer and the application program using Stabilis. This solution is simple, but costly. A more efficient approach would be to alter some classes of Arjuna in order to implement reliable channels using objects states mantained in stable memory by the atomic action protocol.

6.2 An example

We have implemented a variation of the Evolving Philosophers Problem [10] using the algorithms and the reconfiguration software architecture described. The Evolving Philosophers is an extension of the traditional Dining Philosophers [7]. Philosophers are arranged in rings with neighboring philosophers sharing a fork between them. A philosopher is either thinking, hungry or eating. To move from hungry to eating a philosopher must acquire both his left-hand and right-hand forks. To solve this problem is essentially to design a program to adjudicate conflicts: neighboring philosophers do not eat simultaneously and hungry philosophers eat eventually, given that no philosopher is allowed to eat forever.

Chandy and Misra [5] describe the behavior of the diners as follows: *"A fork is either*

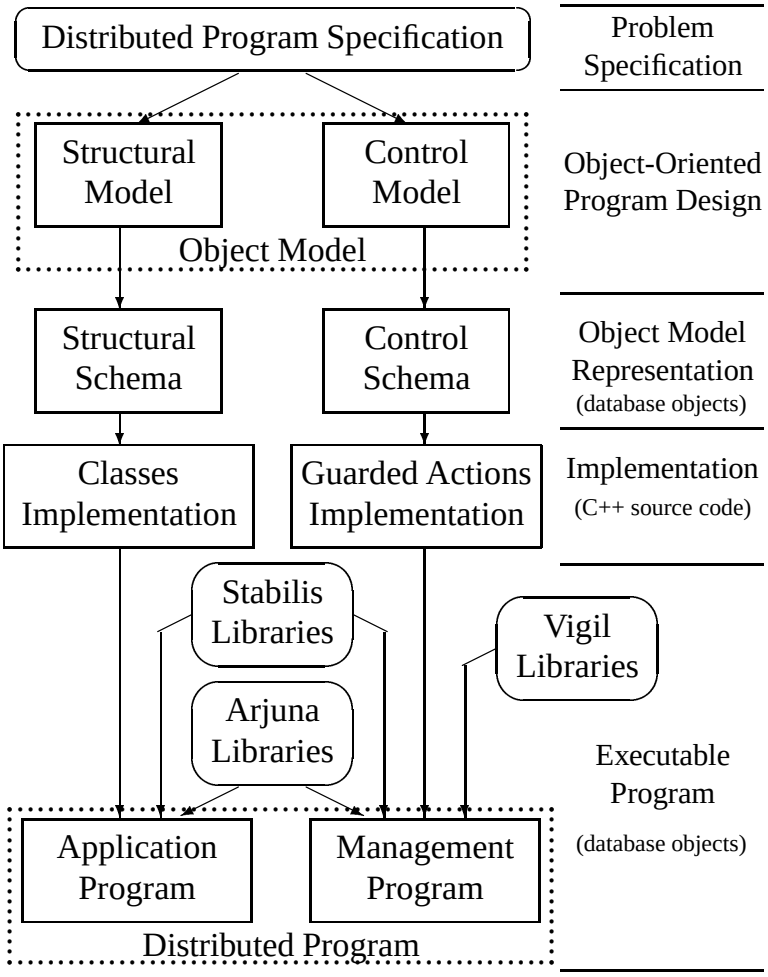

Figure 10: Implementation process

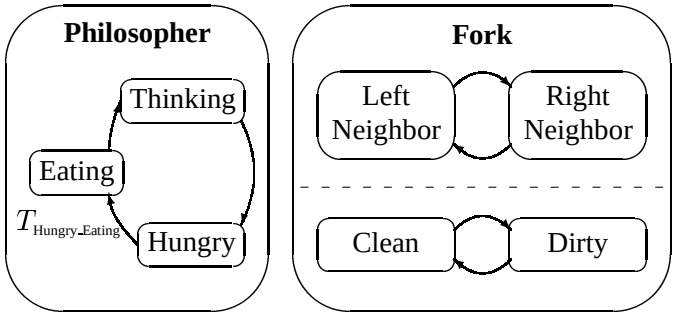

Figure 11: Philosopher's control model

clean or dirty. A fork being used to eat with is dirty and remains dirty until it is cleaned. A clean fork remains clean until it is used for eating. A philosopher cleans a fork when mailing it (he is hygienic). An eating philosopher does not satisfy requests for forks until he has finished eating."

In the Evolving Philosophers, philosophers are born, live and die without disturbing the lives of the other philosophers. In this variation, dynamic reconfiguration procedures must guarantee that rings of philosophers grow or shrink in a manner that is consistent with the correctness of the dining philosophers problem, that is, absence of starvation and deadlocks.

Each philosopher acts as a autonomous program with its own application and control. The application program, not shown here, is very simple, it implements the functions related to a philosopher's thinking activity and the sensors responsible for signalling the transitions from thinking to hungry and from eating to thinking (Figure 11). The fragment of the control program (guarded actions) shown below:

$$T_{\text{Hungry\_Eating}} \equiv (\text{left\_fork(RIGHT\_NEIGHBOR)} \wedge \text{right\_fork(LEFT\_NEIGHBOR)})$$
$$\{\ \text{set\_dirty(left\_fork);} \quad \text{set\_dirty(right\_fork);}\ \}$$

is responsible for the implementation of the transition from hungry to eating (Figure 11). The transition contains sensors and actuators that depend on the state of forks shared by this philosopher with its left and right neighbours. To validate this guarded action the monitoring program has to have access to the state of three objects: the philosopher himself and the two forks adjacent to him. Thus, the only way to guarantee a correct reaction is to assess the guarded action against a consistent global snapshot. This simple example presents evidence for the need of the algorithms developed so far.

7 Conclusion

Algorithms for obtaining consistent global snapshots are useful aid in a vast class of problems that can be formulated as the evaluation of a predicate over the global state of an application. In particular, dynamic reconfiguration of distributed systems can benefit from such algorithms to guarantee that reconfiguration occurs adequately.

There are many algorithms for the construction of consistent global snapshots in the Process and Message Model (PMM). In the Object and Action Model (OAM), it is also possible to obtain a consistent global snapshot, but the solution adopted usually involves the execution of a checkpoint atomic action that has lock all objects of the application, making the operation synchronous and costly. To the best of our knowledge, the consistent global snapshot algorithm proposed by Fischer et al. [8] is the only OAM-tailored asynchronous consistent global snapshot algorithm found in the technical literature. Despite being asynchronous, it demands the duplication of the action structure of the application for the construction of the consistent global snapshot, which makes it less efficient than ours.

In this paper, we have developed a correlation between PMM and OAM. To illustrate the practicality of the correlations found, we have used them to reason about and create OAM-based consistent global snapshots algorithms similar to PMM ones. We have implemented the OAM-based versions of vector clocks, and have used them as the basis for the implementation of a reconfiguration system. Next, we have assessed the algorithms by implementing the Evolving Philosophers' Problem [10].

The evidences derived from our experiments allow us to conclude that our approach to the construction of asynchronous consistent global snapshots in OAM is promising. It seems that our approach to map algorithms is general in the sense that it can be used to map into the OAM algorithms based upon the PMM.

We are now working on the development of a distributed programming environment based on Guaraná (`http://www.dcc.unicamp.br/~oliva`), a flexible reflexive architecture designed to ease reconfiguration and reuse of meta-level objects. This architecture has been implemented in a modified Java[2] virtual machine, based on Kaffe (a Free Java interpreter).

Acknowledgment

This work has been supported by FAPESP under grant no. 95/1983-8 for Islene Calciolari Garcia and grant no. 96/1532-9 for the Laboratory of Distributed Systems. We would like to thank Alexandre Oliva for his help with the paper. We would also like to thank our colleagues from the Institute of Computing, specially from the Laboratory of Distributed Systems, for providing us with interesting and motivating environment.

References

[1] O. Babaoglu. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Technical Report UBLCS-93-1, University of Bologna, 1993.

[2] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. Consistent Checkpointing in Message Passing Distributed Systems. Technical Report 2564, INRIA, June 1995.

[2] Java is a trademark of Sun Microsystems, Inc.

[3] L. E. Buzato. *Management of Object-Oriented Action-Based Distributed Programs.* Ph.D. Thesis, University of Newcastle upon Tyne, Department of Computer Science, Dec. 1994.

[4] L. E. Buzato and A. Calsavara. Stabilis: A Case study in Writing Fault-Tolerant Distributed Applications Using Persistent Objects. In A. Albano and R. Morrison, editors, *Proceedings of the Fifth International Workshop on Persistent Object Systems*, Workshops in Computing, pages 354–375, San Miniato, Italy, Sept. 1992. Springer-Verlag.

[5] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM TOPLAS*, 6(4):632–646, oct 1984.

[6] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computing Systems*, 3(1):63–75, Feb. 1985.

[7] E. W. Dijkstra. *Hierarchical Ordering of Sequential Processes*, pages 72–93. Academic Press, 1972.

[8] M. J. Fischer, N. D. Griffeth, and N. A. Lynch. Global States of a Distributed System. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, may 1982.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, Aug. 1987.

[10] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16:1293–1306, Nov. 1990.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[12] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Computing.* PhD thesis, MIT, Department of Computer Science, 1981.

[13] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report SFB 124-15/92, University of Kaiserslautern, Dec. 1992.

[14] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of Arjuna: A Programming System for Reliable Distributed Computing. Technical Report 298, University of Newcastle-upon-Tyne, Newcastle-upon-Tyne, England, Nov. 1989.

[15] S. K. Shrivastava, L. V. Mancini, and B. Randell. The Duality of Fault-tolerant System Structures. *Software — Practice and Experience*, 23(7):773–798, July 1993.

[16] S. K. Shrivastava and S. M. Wheater. Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coulored Actions. In *Proceedings of Tenth International Conference on Distributed Computing Systems*, pages 203–210, Paris, France, may 1990.

[17] S. M. Wheater and D. L. McCue. Configuring Distributed Applications using Object Decomposition in an Atomic Action Environment. In J. Kramer, editor, *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 33–44. IEE (UK), Imperial College of Science, Technology and Medicine, UK, March 1992. ISBN 0-85296-544-3.