Pergamon

Information Systems Vol. 21, No. 2, pp. 127–145, 1996
Copyright© 1996 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0306-4379/96 $15.00 + 0.00

S0306-4379(96)00008-7

COLLABORATIVE VERSION CONTROL IN AN AGENT-BASED HYPERTEXT ENVIRONMENT

ANTONINA DATTOLO and VINCENZO LOIA

Dipartimento di Informatica ed Applicazioni, Università di Salerno,
84081 Baronissi (SA), ITALY

(Received 31 July 1995; in final revised form 12 March 1996)

Abstract — In this work we discuss a number of issues for the design of hypertext systems in an agent-based model of computation. We examine how the "traditional" fundamental concepts which are at the basis of the design of hypertexts can be re-visited under a new perspective of collaborative expert agents. The paper presents how some principles of high-level concurrent programming are applied as new methodologies for the design and development of complex software, such as hypertext systems. By adopting an agent-based framework, we gain powerful control on version management that presents considerable difficulties for the development of hypertext systems; a general distributed version control mechanism is applied, without significant differences, both in single-user and in collaborative multi-user mode. In both cases, the underlying hypertext architecture is defined in terms of computational agents interacting each other in order to accomplish common goals. In this paper we present a first-level prototype implemented in a concurrent object-oriented language, realized on the top of the Common Lisp Object System.

Key words: Hypertext Systems, Version Control, Object-Oriented Concurrent Design, Computer Supported Collaborative Work, Common Lisp Object System.

1. INTRODUCTION

In the last few decades, we have witnessed the growing interest of the academic and industrial communities in hypertext and hypermedia systems [40]. The wide range of applications of these "new" technologies has imposed their importance as indispensable features for computer-based systems. The work presented in this paper is the result of our experience gathered in the last six years working on the design and implementation of complex programming environments. In particular, we developed a software platform, named OPLA, able to conceive and implement large object-oriented logic programming systems. The architecture consists of a fast interpreter and compiler [32], enriched by a programming environment [33] able to support the various activities which normally accompany the code production (debug utilities, browsing, documentation facilities, etc.). Owing to the fast growth of OPLA users and the strong object-orientation of the language, the need to strengthen the hypertext aspect in documentation management became a key issue for the survival of the OPLA platform itself. Before starting the implementation of this important requirement, we established the most important principles to follow in order to reduce the possible costs and risks of the project. Some of these guide-lines were:

- design a general-purpose hypertext framework, in order to re-use it for other software applications not strictly limited to programming environments;
- do not modify the underlying application and, consequently, the basic implementation choices adopted to develop the interpreter, the compiler and the other main components of OPLA;
- obtain something that could be specialized for program documentation, in particular for object-oriented logic programming;
- manage not only the hypertextual spatial data separation but also a spatial duties organization.

As regards the last point, we have that information-intensive applications refer to external materials [38], so that parts of the hypertext are decentralized in different computational settings. For this reason, a hypertext design environment should support efficient management of distributed information, exploiting (where possible) a corresponding distribution of services and tasks. As a matter of fact, in the hypertext field, the concurrent, or distributed approach is not new, but its use has been limited to solving problems which generally occur in multi-user hypertext systems, where we have a concurrent, collaborative access to databases shared across a net of workstations and file systems [36, 45, 49].
Our approach is different: we focused our attention on exploring a general design environment suitable for combining the decentralization of data and tasks; we have adopted the agent model as a general framework to design the basic components of a hypertext system, by sketching both internal and external activities through an extended communication of task requests sent along a web of intelligent agents. Even though in recent years agent-based systems [8] seem to be a cliché, we recognize that their effective utilization radically revolutionizes the way in which the software is conceived and works. In an agent-based approach the knowledge, intelligence, information and tasks are distributed over populations of computational agents. To reach global solutions, specific cooperation schemes are formulated and implemented in order to allow the agents to execute their goals autonomously and independently [17]. Generally, agent-based systems are realized by Object Oriented Concurrent Programming (OOCP for short) [41] languages. In fact, this choice frees the programmers from specific hardware requirements and facilitates the design of those software systems that suffer from a monolithic computational approach. Programming software in agent-based frameworks has stimulated research in re-visiting the foundations of Artificial Intelligence, by providing new schemes and methodologies to handle large-scale open systems [27]. Open Systems consist of a population of agents each of which is equipped with local knowledge and influence. The agents can share their knowledge and organize their activities to reach a common goal [13].
In this paper we focus our attention on the advantages that an agent-based design and implementation can provide to a fast high-level prototyping of open hypertext systems [14, 43]. The paper is organized as follows. In Section 2 we stress the benefits that high level distributed computing brings to the design of a concurrent architecture of hypertext systems. Here we present the software platform used in our implementation. The platform is an OOCP language available on top of the Common Lisp Object System, the ANSI specification of the object-oriented extension of Common Lisp [6]. Section 3 provides the agent-based model of hypertext, taking into account the different agent populations which have been defined. Section 4 discusses the versioning problem, one of the relevant issues in modeling and building hypertext systems. Our version control management is based on a collaborative activity among agents. Firstly, collaborative, single-user versioning is presented, differentiating the creation and selection phases. Secondly, in Section 5, we extend the previous versioning mechanism in order to support computer supported collaborative work (CSCW). We conclude the paper showing further research trends of our project.

2. OPEN HYPERTEXTS: THE AGENT-BASED APPROACH

Generally a concurrent system has a kind of module (object, guardian, actor, agent) which is invoked by (and only by) messaging. This module hides data from all other modules, i.e. there is no sharing of data between modules. By messaging we activate light-weight processes within an addressed module. Sometimes these modules are known as "active objects" and programming with active objects is nowadays named Object Oriented Concurrent Programming (OOCP) [41]. OOCP is represented by a large number of concurrent languages, which, despite their differences, allow to define active objects which interact each other through simple or complex communication models. Originally these objects were named *actors* because the underlying computational model was inherited from that defined by Hewitt [26] and successively deepened by Agha [2]. However the "pure" actor model suffers from rigid point-to-point communication protocol [3] that limits the design of efficient collaboration strategies suitable in organizational-based environments [16]. We overcome the classical actor-based model by introducing new communication strategies, such as multicasting, broadcasting, agent type-based message passing protocol. In order to avoid ambi-

guity, we refer to our computational entities with the generic name *agents*, being aware of that in our model the agents are not provided with hardwired skills for "social" reasoning [52]. For this reason, in our approach the designer of the distributed applications is responsible for implementing the wished cooperation activities by using the basic constructs of the language.
Before entering into implementation details, we would like to illustrate the general model of our architecture. Essentially, a hypertext is a collection of heterogeneous objects (cognitive fragments viewed as texts, images, sounds,...) connected together by conceptual links. The user navigates this graph of concepts by browsing the nodes and applying different actions. Owing to the richness of the resulting environment, hypertexts are characterized by numerous distinctive features, such as:

- strong fragmentation of information,
- high interactivity of information,
- continuous stimulus to modify the contents,
- natural approaches to retrieve information,
- opening towards other systems.

These important concepts generate several problems for the implementation of hypertext systems. The most common difficulty is known as "tyranny of the link" [24], which expresses the rigidity of the architecture which is not able to support, in an appropriate way, the cognitive activities offered by the hypertext. Recently, the need to improve the freedom of communication/handling of hypertext information has stimulated the research community into providing new strategies suitable for dynamic evolution of the system [14]. We argue that many difficulties remain unsolved in the implementation phase, because the underlying tools used as platforms for the design and implementation of hypertext do not support the concurrent, collaborative issues that exist at the basis of the system. In our proposal, the hypertext is conceived as an Open Information System [27], implemented by agents. The expertise, the knowledge, the actions of the system are sprinkled among different classes of agents that work simultaneously and independently. This "society of experts" is animated by exchanges of messages, by means of which the agents can communicate. The communication is asynchronous because each agent keeps a mail-box to receive enquiries and it keeps functioning even when receiving messages. Our model of agent programming has been realized using an object-oriented concurrent language based on the top of Common Lisp Object System, CLOS [6]. The concurrent extension of CLOS was named CLAS (standing for Common Lisp Actor System) and was initially tested and improved during the implementation of a significant application [18]. Once the functionality of CLAS was proven, we decided to specialize the new method as a tool for the design of hypertexts. The resulting system, named HYPERCLAS, is illustrated in the next subsection.

2.1. An Introduction to HYPERCLAS

HYPERCLAS allows the creation of populations of agents which specialize in accomplishing tasks that generally occur in hypertext systems. The complete architecture employed to design the hypertext system is depicted in Figure 1.

As the reader can note, HYPERCLAS has been built thanks to three main modules: CLOS, MTF-LCL (Multi-Tasking Facility of Lucid Common Lisp) [34] and CLUE [29]. CLAS provides object-oriented concurrent programming by using MTF-LCL in CLOS itself. CLUE is utilized to manage the graphical, window-based X11 interface in CLAS.
From the standpoint of agent handling, HYPERCLAS is composed of two levels:

- The first level consists of a module designed to solve all the problems which arise in concurrent programming, such as management of concurrent access to resources, scheduling of tasks, binding of local or global data to a process, locking or unlocking of a process, creation of processes and their destruction. To accomplish these goals we defined a super-class, named

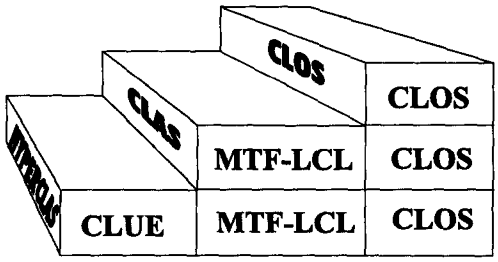

Fig. 1: The modular composition of our platform.

`atomic-object`, which specializes in handling mutual exclusion of processes. Thanks to this class, the legal access to fields of agents is guaranteed.

- The second level realizes the agent model. The main difference between objects and agents consists in the fact that objects are seen as passive entities that communicate with other objects via active messages whereas agents are seen as *active* objects that exchange passive messages. Each agent is composed of a passive and active part, each of which is an object of CLOS.

The active part of an agent is an instance of a class task that encapsulates the interface with the host multitask system. This class contains a slot called `jeckill` whose function is to address the passive part of the agent, i.e. the class `Agent`. Vice versa, in the class `Agent` we refer to the active part by means of a slot `hide`. In the code of Figure 2, we provide the definition of the most general object of HYPERCLAS, the `Agent` entity.

```
(defclass Agent(atomic-object)
( (hide    :initform () :reader get-hide :writer set-hide)
  (mbox    :initform () :reader get-mbox :writer set-mbox)
  (parcel  :initform () :reader get-parcel :writer set-parcel)
  (name    :initform "hector" :initarg :name
           :reader get-name :writer set-name)
  (agents  :allocation :class :initform () :initarg :agents
           :reader get-agents :writer set-agents) ) )
```

Fig. 2: The basic definition of the Agent entity.

The class `Agent` contains five slots:

- `hide` is meant to address the active part of the agent;
- `mbox` is used to support communication with other agents;
- `parcel` owns the current message;
- `name` identifies the agent;
- `agents` maintains abstract knowledge about the created agents.

As the reader can note, the definition of the agent embodies (as in CLOS) only the data part. For each of these slots, we have methods specialized in handling it. For example the methods `:initform`, `:reader` and `:writer` are automatically generated by HYPERCLAS in order to accomplish operations of initialization, reading and writing, respectively. Of course, we have other methods usable by `Agent` outside its data part definition. In fact, the remaining behavioral part is detached from the agent data description, i.e. the implementation of the *scripts* is done outside the class definition. This programming style, which derives from the initial CLOS proposal, leads to a more flexible model. In fact, the connection between data (agent/class definition) and operations (scripts/methods definition) is established by dynamic bindings and exploiting hierarchical structure. In this way, the user can specify the structure of the agents via the `defclass` command, whereas the management of the contents is realized via the scripts definition, giving flexibility and efficiency in the separation structure/content. Before focusing our attention on the behavioral description of the agent, we will provide more information about the message passing strategies possible in HYPERCLAS.

2.2. Message Passing Protocol in HYPERCLAS

In HYPERCLAS the message passing protocol is enriched with five schemes that fulfill the various needs to send a task to agents. These schemes are:

- `future`
  `future` is an answer to a message in the form of a promise; the semantic of `future` is equal to that of ACT-1 [30].
- `spray`
  We select this message passing when we want to apply multicasting messages.
- `express`
  `express` is a message with the highest priority. If an agent receives this message while it is treating a normal one, then it stops the current task in order to handle the `express` request.
- `broad`
  We select this message passing when we want to apply broadcasting messages.
- `all`
  HYPERCLAS allows the creation of clones. A clone is a perfect duplication of the original agent. The difference between the two agents consists in the fact that when an agent is cloned, then the clone replaces the existence of the original agent that becomes an inhibited entity, i.e. it remains deaf to any external stimulus. If we want to escape from this rule, we can use the keyword `all`. In this case, the message is sent in multicast to both the active and inhibited agents.

Message passing is supported by means of the construct `send`. A general form of send appears as:

```
(send destinator kind-of-message task arguments)
```

where:

- `destinator` identifies the agent(s) to which the message is addressed;
- `kind-of-message` specifies the strategy by which the message must be sent on the web (`future, spray, express, broad, all`);
- `task` determines (via the keyword `:selector`) the script that we want to trigger once the message is acknowledged by the addressed agent(s);
- `arguments` is meant to specify (via the keyword `:args`) the arguments (if existing), required by the script.

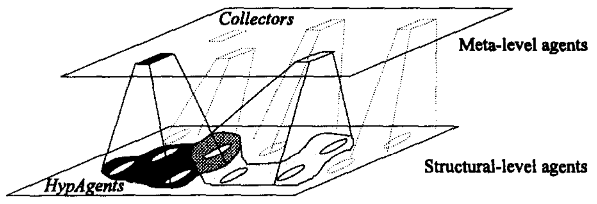

Fig. 3: Two different layers for Collectors and HypAgents agents.

3. AGENT-BASED HYPERTEXT MODEL

The framework of our hypertext model is completely distributed. All the information and the services are sprinkled over a web of autonomous agents. More precisely, our general framework is organized in two layers, as depicted in Figure 3.

The first, named *Structural-level agents*, corresponds to the architectural model provided by the hypertext authors. It is composed of the population of *HypAgents*. The HypAgent entity plays the same role of well-known objects, such as notecards, frames, nodes, entities [15, 25]. The second level, named *Meta-level agents*, provides a more "functional" hypertext perspective; on this layer we introduce a designed agent category, the *Collectors*, that allows visualization and manipulation of sub-sections of the underlying structural layer.

3.1. The HypAgent

This class of agents presents characteristics normally contained in nodes and links of classical hypertext models. The following code shows the definition of a generic `HypAgent` agent.

```
(defclass HypAgent(Agent)
( (title     :allocation :class :initform () :initarg :title
             :reader get-title :writer set-title)
  (version   :initform () :initarg :version :reader get-version
             :writer set-version)
  (text      :allocation :class :initform nil :initarg :text
             :reader get-text :writer set-text)
  (image     :allocation :class :initform nil :initarg :image
             :reader get-image :writer set-image)
  (sound     :allocation :class :initform nil :initarg :sound
             :reader get-sound :writer set-sound)
  (fromAgent :allocation :class :initform () :initarg :fromAgent
             :reader get-fromAgent :writer set-fromAgent)
  (toAgent   :allocation :class :initform () :initarg :toAgent
             :reader get-toAgent :writer set-toAgent)
  (deaf      :initform () :initarg :deaf :reader get-deaf
             :writer set-deaf) ...) )
```

Fig. 4: The data part description of the HypAgent object.

This single object is used to collect some fundamental data. Following the code of Figure 4, we specify the role of these attributes:

- `title` stores the frame topic.
- `version` is used for versioning management.

- `text`, `image`, `sound` contain the text, image and sound related to the topic, respectively.
- `fromAgent` stores the address of the HypAgents reachable from the current agent.
- `toAgent` stores the address of the HypAgents from which it is possible to reach the current agent.
- `deaf` maintains the address of the original node, if the HypAgent is a cloned entity.

The operational part of a HypAgent is represented by its scripts; scripts are like sub-programs representing all the possible actions that the agent can perform. A representative number of these scripts will be discussed in the rest of the paper.

3.2. The Collector

This class of agents allows us to manage alternative browsing structures and views of partial sections of hypertext. These agents are introduced to create and handle separate collections of HypAgents, by providing a more abstract treatment of browsing techniques. The Collectors, as well as the composites, ***provide a means of capturing nonlink-based organizations of information, making structuring beyond pure networks an explicit part of hypertext functionality*** [20].
The main task of this category is to support the following requests:

- allowing the author to structure the hypertext creating collections;
- allowing the user to require a portion of the hypertext; indirectly a collection is created. This collection represents an extraction of HypAgent sub-population selected from all the agents forming the hypertext, i.e. an already existing "traditional" sub-graph is returned;
- allowing the user to apply browsing strategies; a new sub-graph is created. In this case, the collection is still a sub-population of HypAgents, for which a specific browser is *dynamically* built and triggered.

To give an intuitive idea of this new category, we show the code of this class (Figure 5).

```
(defclass Collector(HypAgent)
( (collection    :allocation :class :initform () :initarg :collection
                 :reader get-collection :writer set-collection)
  (frontier ...;;code details omitted)
  (portion ...)
  (clones ...)  ...)  )
```

Fig. 5: The data part of a Collector.

Some of the most relevant acquaintances of the Collector data part are:

- `collection`: to store a set of addresses corresponding to a given collection;
- `frontier`: to establish the borders of a region of HypAgents in a certain collection;
- `portion`: to identify the collection on which new changes may occur;
- `clones`: to address those HypAgents belonging to a duplicated collection.

The next two sections discuss the version control management mechanism [24]; in particular, Section 4 is dedicated to explaining the version control strategy for a single-user hypertext environment. In Section 5, we extend the single-user strategy in order to treat the version control for CSCW architectures. Such extension is applied by adding new agents to our basic platform, without modifying the basic behavior of our methodology, which remains in both cases (single/multi-users) based on a distributed, collaborative approach.

4. COLLABORATIVE VERSION CONTROL MANAGEMENT IN SINGLE-USER MODE

The hypertext is a unstable resource. Users can create, destroy, modify nodes and links, changing small or large sections of the hypertext. In spite of these changes, some important laws must be observed:

- coherence of the information.
  For instance, when we cancel a node, we must reorganize the area of the hypertext in which this operation occurs. This task consists in eliminating the corresponding links, updating the contents, moving markers, testing the validity of such changes, etc..
- access to old versions of the hypertext.
  The return to previous cognitive states must always be available, in order to reuse the information and to maintain the derivation history [21]. In particular, in systems that utilize concurrency, version management is absolutely necessary to maintain consistency of data [36].

In the following, we use the term *version control* [5, 46] to indicate the ability to manage dependencies between subsequent instances of the same document, organize them into meaningful structures and allow such operations as navigation through or computation of them. In particular, the version control provides different control strategies suitable for handling the node-based version and the structure-based one [50]. In our model, the approach to version control is uniform, i.e. the node/structure distinction is broken, since in the agent-model each single entity acquires the global net not by accumulating data in a single entity, but by applying concurrent cooperation schemes among de-centralized entities in order to reach common goals. Thanks to this new perspective the node-based version becomes a particular aspect of the most general structure-based version. Hence, in this paper we use the terms *configuration* or *version* with the same meaning as the term configuration adopted in software engineering [28], i.e. to indicate a specific state of the hypertext structure as a whole.

The goal of this section is to show how versioning, one of the key issues concerning the functionality of a hypertext, is carried out by specialized scripts of agents. By providing chunks of HYPERCLAS code, we prove how the agent-based model is a powerful technique to handle the hard task of version management. In detail, we illustrate our approach to manage the configuration problem. Such versioning faces the problems of updating all the hypertext in its current form [9]. We focus our attention on version control, omitting details about databases memory management.

4.1. Version Creation

To better introduce the reader to our solution, we explain our main idea, discussing our version mechanism in the case of node versioning which characterizes a session-based[†] versioning. Successively, we tackle the application of our mechanism in the structure versioning showing that no difference occurs in this case.
In Figure 6a, we can observe a general situation that occurs when the user decides to create a new state of the hypertext with a new version mark. Figure 6a depicts the state of the hypertext associated with a version labelled with t_i. An original node is identified by Nk, whereas the notation Nkvj will identify the node Nk existing in a successive version *vj*.

Each node of the hypertext, i.e. each HypAgent, contains, as local information, the list of all the versions to which it belongs (for simplicity we suppose that the only existing version is t_i). The cognitive activity of the user is located on the node N2. The user modifies the node and stores the new content. This command provokes a session-based versioning operation, with a new storing of the hypertext indicated by t_{i+1}. This situation triggers the script `createConfig`. The action performed by such a script is shown in the code of Figure 7.

[†]By session-based versioning, we indicate, as in [36], the automatic creation of a new configuration at the end of each editing session for a given document.

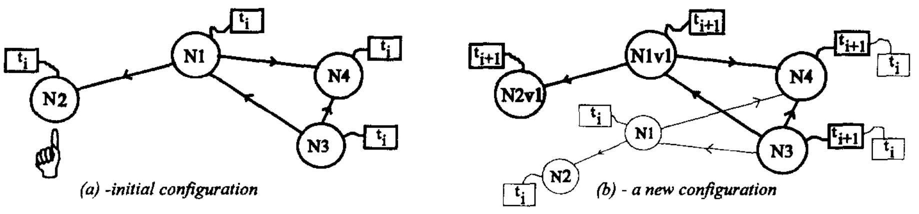

Fig. 6: The configuration of the hypertext fragment before the modification on the node N2 (a) and after (b).

```
(defmethod createConfig((self Collector) newConfig)
  (send (get-portion self) :spray :selector givemeYourFrontier)        1
  (set-collection (append portion frontier))                           2
  (send (get-collection self) :spray :selector cloneYourSelf)          3
  (send (get-collection self) :spray :selector freezeYourSelf)         4
  (send (get-clones self) :spray :selector changeYourName              5
  (send :broad :selector updateConfig :args newConfig) )               6
```

Fig. 7: The script **createConfig** creates a new configuration.

The basic operation is to store the previous state of the system, in order to recover it during a derivation history. The duplication is necessary to maintain old layers of configuration; we duplicate only the section of the hypertext which is probably submitted to change. This section is composed of two different entities:

- the current agent(s);
- the collection of neighbour agents named *frontier*.

As regards the neighbour agents, possible alterations concern only the so-named *frontier*, i.e. that HypAgents sub-population identified by all the incoming/outcoming neighbours that do not fall in the section to be modified. We underline the fact that the neighbours of the frontier are not cloned but just updated; the updating consists in adding in their local contexts the new version mark and the links to the cloned frontier in order to bind the bulk of the new configuration with the rest of the hypertext. In our example (see Figure 6a), this corresponds to duplicating the current node (N2), together with its frontier (N1) avoiding to duplicate the nodes N3 and N4.
A Collector is responsible for creating a new configuration. Let us illustrate all the basic steps of this activity, following the statements of the code in Figure 7.

As first action, the Collector establishes which HypAgents need to be duplicated. This decision is taken in line 1, where a multicast message is sent to each HypAgent belonging to the portion subject to modification, the local resource `portion` (in our example, N2). In this way the frontier (N1) is identified. As second step, the Collector assembles (line 2) both these resources (`portion` and `frontier`), in order to store in the acquaintance `collection` the complete area to be modified (N1, N2). Next, the Collector clones in parallel each element in `collection` (line 3). This action is necessary to substitute new HypAgents (the cloned ones, N1v1 and N2v1 in Figure 6b) with those which belong to past configurations (the original ones, N1 and N2). To guarantee coherence, after having cloned, the Collector freezes the original nodes (line 4). Frozen agents become inhibited entities. Though they exist, they are entities unknown to the rest of the system. Only a special message can resume frozen agents, as we will soon see. After all the HypAgents have generated the corresponding clones, these last ones concurrently change their name (line 5). Now, it is necessary that all the new cloned HypAgents and the rest of the active hypertext must be admitted to this new configuration by updating the configuration list with the new version label **newConfig** (t_{i+1}); this task is accomplished by sending, in broadcast on the net, the script **updateConfig** (line 6). During this broadcasting, the script takes into account the need to bind the cloned frontier (the

node N1v1) with the rest of the unchanged hypertext by adding this new link inside the contexts of the neighbours of the cloned frontier (the links N1v1-N3, N1v1-N4). In Figure 6b, we sketch the configuration after the cloning of the nodes N1 and N2. The reader can observe that the new state is singled out because the nodes N3 and N4 now belong to both the configurations marked with t_i and t_{i+1}, whereas all the active remaining nodes exist only in the latter version t_{i+1}. In order to distinguish the active from the inhibited entities, we use, in our graphical representation, bold objects to depict active agents. It is worth enphasizing that an agent can be active only in one version. Thus, in the configuration labelled with t_{i+1} of Figure 6b, the incoming links for N4 exist only from the nodes N1v1 and N3 (since the node N1 is invisible in such configuration).
The mechanism discussed before is normally adopted for session-based version creation. The difference with a user-decided[†] version creation lies in the fact that in this last case the user decides when and where (on what agents) the storing of the hypertext occurs. For instance, let us suppose that the user alters the state of the nodes N3 and N4 in Figure 6b and that, only after having modified N4, the user requires the creation of a new version. In this situation, following our mechanism, we obtain a new configuration, as shown in Figure 8.

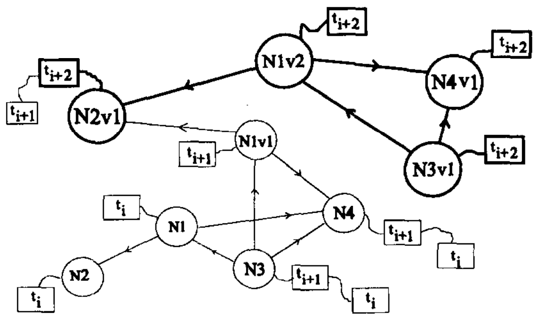

Fig. 8: The configuration identified by the mark t_{i+2}.

We stress the fact that only active objects are cloned. The implementation of our versioning mechanism fully utilizes the parallel computation to the full. As the reader can note, the code of Figure 7 adopts largely the `:spray` sending option, in order to benefit from concurrent processing. In more detail:

line 1.
The identification of the collection to be stored is accomplished in parallel, because each agent in the resource `portion` owns enough knowledge to recognize those neighbours that will compose the `frontier`. Each of the agents in `portion` performs this task concurrently.

lines 3, 4 and 5.
The cloning of original nodes, their freezing and the change of names for cloned agents occur in a concurrent and asynchronous way. Thus, the fragment of the hypertext that must be stored is built in parallel. Thanks to the semantic of the script `createConfig`, we handle different configurations of the hypertext in a high level way. The partition between active and inhibited societies of HypAgents allows efficient management of old contexts.

line 6.
In order to notify the new configuration of the hypertext, each interested node is informed concurrently. The script `updateConfig` broadcasts this message on the net until all the nodes acknowledge its reception. In this way, although we do not copy the whole network, we extend the version mark over the whole network.

[†] By user-decided versioning, we indicate, as in [36], the version management driven by the user preferences.

4.2. Optimization Process

The methodology previously described can be optimized by limiting the number of active HypAgents that need to be maintained. More precisely, when at the end of the modification process there are some nodes in which no change occurred, then we suppress these nodes and replace them with their corresponding original nodes. The optimization task is shown in the code of Figure 9.

```
(defmethod optimizeConfig ((self Collector))
  (send (get-clones self) :spray :selector optimizeYourself))               1

(defmethod optimizeYourself ((self HypAgent))
  (if unchanged                                                             2
  (progn (send (get-deaf self) [:all :express] :selector awake)             3
  (send (get-deaf self) :selector updateConfig)                             4
  (send (get-deaf self) :selector updateLinks)                              5
  (updateLinks)                                                             6
  (suicide)) ) )                                                            7
```

Fig. 9: The script to optimize a configuration.

The same Collector, which created the new configuration, is responsible for applying the optimization. This action is carried out by the script `optimizeConfig` shown in Figure 9. The role of this script consists in delegating, in a concurrent way, the optimization mechanism to each clone present in the local resource `clones` (line 1). Each clone compares itself with the corresponding HypAgent: if no difference is noted (line 2), the cloned HypAgent awakes its original one (line 3) using the options `:all` to access to an inhibited entity and using `:express` to force the termination of such message before considering the next message. Once the original HypAgent becomes active, it can restore its (unique) presence in the hypertext. To do this, its configuration (line 4) and links (line 5) are updated, taking into account the links of its neighbours. The same updating operation (line 6) is carried out on the neighbours of the clone and finally the clone suppresses itself (line 7)[†].

Let us consider again Figure 8. Supposing that the links N1v2-N3v1 and N1v2-N4v1 are unchanged, then it is necessary to maintain the frozen HypAgent N1v2. Consequently, this is suppressed and the agents N3v1, N4v1 directly point to the (now active) agent N1v1, as shown in Figure 10.

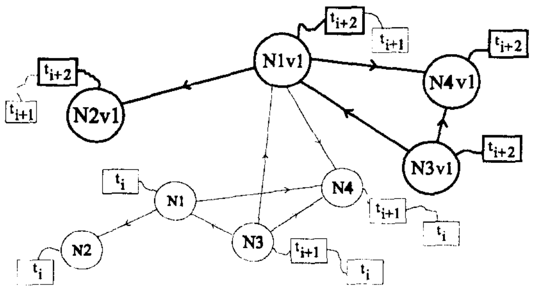

Fig. 10: The result of the optimization process.

[†]The optimization process can be executed only when there is no user cognitive action on the interested nodes. During its execution, the nodes remain inaccessible.

4.3. Version Selection

This activity enables the detection of a particular version of the hypertext. In our model the hypertext is represented by a population of both active and inhibited HypAgents. The active ones provide the current configuration. The remaining ones belong to suspended, past configurations still living in our agent-based universe. In this section we explain how we can gather all the agents belonging to a designated configuration. Let us suppose that the user requires access to a configuration by providing a version identifier (not strictly time-bounded). The search for such a configuration is executed by the Controller by triggering the script `searchConfig` shown in Figure 11.

```
(defmethod searchConfig((self Collector) version)
  (send [:broad :express] :selector freeze)                             1
  (send [:broad :all :express] :selector sameVersion :args version)     2
  (send (get-collection self) :spray :selector awake)                   3
  (send (get-collection self) :selector display) )                      4
```

Fig. 11: The script designed to select a configuration.

The main goal of this script is to make active all and only those HypAgents belonging to the designated version. To reach this goal, firstly all the nodes of the hypertext are frozen in parallel (line 1) and secondly a multicasting message is sent across the net, with the provision that also inhibited agents are to be addressed, in such a way that each HypAgent tries to match its version mark with the desired one (line 2). When this match is successful, the agent enters into a new collection. The script `searchConfig` requires this new collection to become active (line 3) and then to display itself (line 4). To explain this mechanism in more detail, let us re-consider Figure 10. The user requires to go back to the configuration identified by the mark t_{i+1}.

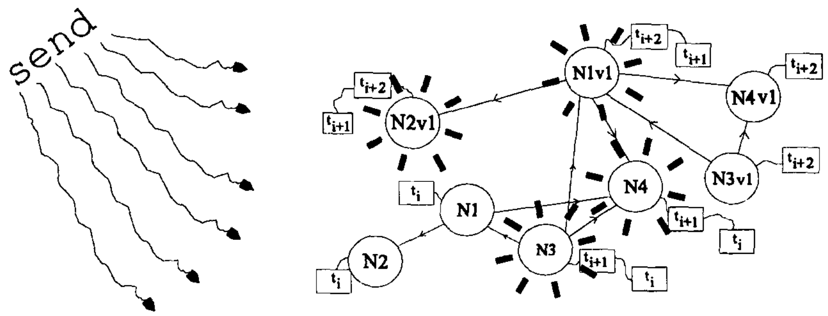

Fig. 12: The result of the configuration t_{i+1} selection.

Looking at Figure 12, a number of send messages is addressed on the net. The effect of the send is the awakening of those HypAgents that belong to the configuration t_{i+1} (N1v1, N2v1, N3 and N4). We can note that our approach is not restricted to a time-based management. In fact, a very similar implementation of the `searchConfig` script is adopted to handle situations in which the user requires access to versions of hypertext objects on the basis of their attribute values [11].

4.4. Related Works

Our agent-based model represents a new strategy to conceive of hypertexts. The novelty of our approach consists in a strong decentralization and distribution of control mechanism that operates on the nodes of hypertext. As a result of this approach, the node is no longer a passive container of knowledge [15], but it takes the role of an active entity that owns enough knowledge in order to establish communication with corresponding neighbours. The usual separation between node and structure versioning [22, 50] is broken: we treat both aspects of versioning, gaining in simplicity and uniformity, and avoiding the necessity to define separate resolution strategies. Our version group (that is, the set of all entities which are considered versions of the same entity [42]) is implicit, because any clone knows its original node. An other interesting consequence of the cloning is that the system maintains the consistency of knowledge [24, 36] and allows, differently from Neptune [12] for instance, to track the derivation history, by linking new version to an older one. Our configuration definition, in some aspects similar to the concepts of context described in PIE [19] or in [42], is characterized by a different composition law of the layers. In fact, in our approach, the configuration (context) is not the sum [42] or the combination [19] of layers: the partition of the hypertext is carried out by appropriate awakening of populations of agents existing in the net. Moreover, we do not suffer from particular strategies to handle the links [22], because they are seen as pure relations (acquaintances) between agents. As in [21], we recognize the importance to access versions of hypertext on the basis of their attribute values and we support user-decided versioning and session-based versioning, by differentiating their handling as stressed in [36].

5. COLLABORATIVE VERSION CONTROL MANAGEMENT IN MULTI-USER, COLLABORATIVE MODE

The importance of versioning for CSCW applications is stressed when versioning is combined with other collaborative techniques, such as semi-synchronous and synchronous collaboration. The integration of versioning into semi-synchronous and synchronous groupwork environments allows users to select a certain state of their work, to be aware of related changes and to cooperate with others either asynchronously or synchronously [44].

The problem is that the needs of a collaborative group are different from those of an individual user. For this reason, the support of collaborative work requires re-examining the design assumptions that have hitherto been used in building tools for individual use [47]. The necessity of redesigning the target architecture is avoided in our approach; in fact, the agent-based paradigm constitutes a natural and attractive technique to solve problems arising from collaborative-based domains. Thanks to this approach the extension of our model has not forced to change the basic hypertext platform, but only to enrich the agent population with new entities and to define new cooperation schemes. In the following section, after having sketched the extension of our framework, we discuss the cooperation strategies which enable us to support CSCW.

5.1. *Extending the Model for CSCW*

The extension has been applied in two directions:

- Improve the HypAgent with local intelligence to support lock/unlock operations on the acquaintances of the node. Essentially, it is necessary to introduce new script entities which control access to the resources, enabling one to distinguish the operation mode for each media `text, image, sound`. Furthermore, such scripts may apply on links (the acquaintances `fromAgent/toAgent`), since these resources are defined as local to the HypAgents.

- Define new agent populations to adapt our previous model for a CSCW environment.

In particular, this last extension has generated a new agent population that acts as an interface between the hypertext resource and the different users [31]: such new agents are named UserAgents [10]. The complete organization is depicted in Figure 13.

For each new user we have a corresponding UserAgent in order to manage the possible concurrent actions with the other users that may concurrently, or not, cooperate for reading/writing operations. Analogous to the HypAgents, there exists a meta-level object for the UserAgents. The Collector that organizes the UserAgents is named UserCollector. Its main role is to synchronize the different active UserAgents.

In the next section, we discuss the behavior of these new agents, focusing our attention on a CSCW environment.

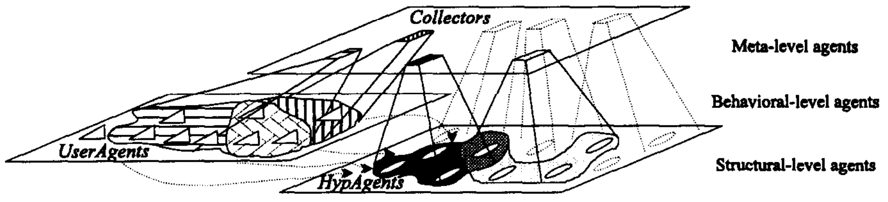

Fig. 13: The CSCW extension introduces the *behavioral-level agents*

5.2. Cooperation Schemes

To generalize our discussion we suppose that n users are active on the net in a given instant. In particular, k users are focused on the same node N1, while the remaining $n - k$ are located outside N1. In this paper we concentrate our discussion on these agent categories:

- $UA^{r/w}_{1..k}$ the set of UserAgents on N1 which execute r/w (read/write) operations;
- $UR^{r/w}_{k+1..n}$ the set of the remaining UserAgents.

To simplify our discussion, we treat only the concurrent reading/writing processes occurring on the HypAgent N1, taking into account that such restriction does not affect the generality of our method.

In the following subsections, we list the possible situations that demand the use of cooperation activities between the UserAgents, the HypAgents and the Collectors. For each of these cases, we provide the basic features of the collaborative version control management, showing how this is carried out in a concurrent way.

5.3. Case 1: Reading Activity.

Users are involved only in reading activity. Standard browsing demanded.

5.4. Case 2: a Single Writer UA^w_1.

This situation is depicted in Figure 14a[†]. The first task is to apply the versioning procedure on N1; this task is accomplished in the way discussed in section 4.

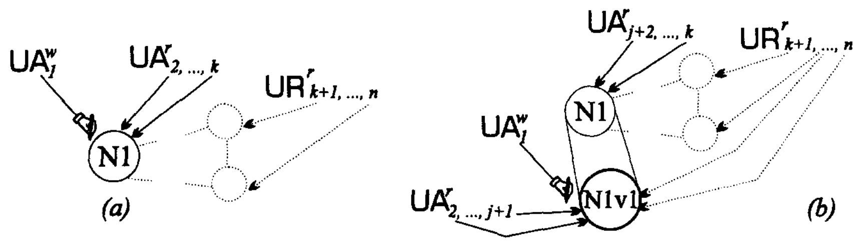

Fig. 14: One writer UA^w_1 and $k - 1$ readers on N1 and $n - k$ readers outside N1

Then, the acquaintances selected by the user for modification are locked; this process is executed in parallel since the HypAgent utilizes the multicast message passing facility. Afterwards, a notification mechanism must be executed. All the UserAgents must be informed about the writing on N1. At the meta-level object, the UserCollector knows all the UserAgents; this information enables the UserCollector to send, in multicast, the notification message. In this situation, the users behavior falls in two categories:

[†]To avoid graphical confusion, in the next figures, any time that we apply versioning on N1, the object N1vj, for a generic j, represents the cloning of N1 and its neighbours.

- those ($UA^r_{2..j+1}$) that want to share the view of the new N1 (N1v1) according to the WYSIWIS[‡] principle;
- those ($UA^r_{j+2..k}$) that do not want to share.

More precisely, if $UA^r_{2..j+1}$ belong to the first category, then the link between each $UA^r_{2..j+1}$ and N1 is removed and a new link is established between each $UA^r_{2..j+1}$ and N1v1. The remaining UR^r agents may acquire a new link for N1v1. No particular action is required for the second user category $UA^r_{j+2..k}$. This situation is depicted in Figure 14b.

5.5. *Case 3: p Writers $UA^w_{1..p}$ on Separate Acquaintances.*

Let UA^w_1 UA^w_2 $\ldots UA^w_p$ be the UserAgents wanting to write on the HypAgent N1 in different acquaintances, as shown in Figure 15a.

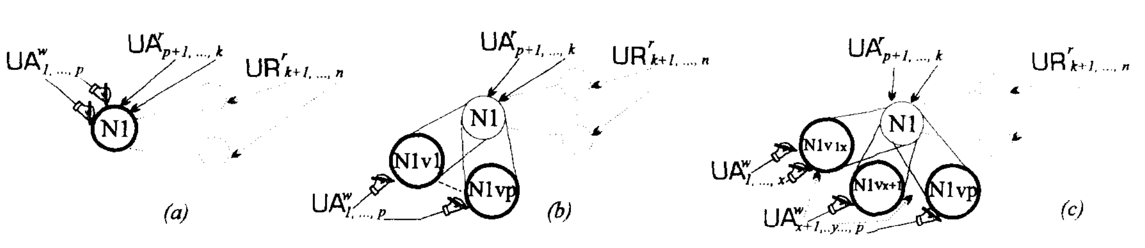

Fig. 15: p writers and $k-p$ readers on N1 and $n-k$ readers outside N1

Parallel versioning mechanisms are executed independently for each of the UA^w writers (see Figure 15b). The locking operation is then applied and notification is sent by the UserCollector on all the UserAgents in multicast. At this point, the $UA^w_1 \ldots UA^w_p$ may work according to the following schemes:

- $UA^w_{1..x}$ *want to work in WYSIWIS mode.* In this case, the relative cloned sub-sections (in Figure 15b, N1v1, ...) associated with the writing UserAgents ($UA^w_{1..x}$) are merged in a unique texture, i.e. a unique cloned subsection (N1v1x in Figure 15c) on which the different users work concurrently.
- $UA^w_{x+1..p}$ *want to work in separate mode.* In this case, no particular action is required. The different cloned HypAgents (N1vx+1, ..., N1vp in Figure 15c) are maintained for each UserAgent $UA^w_{x+1..p}$. At the end of the modification session, the cloned sub-sections could be collapsed in a unique texture.
- $UA^w_{x+1..y}$ *want to work in loose cooperation mode.* This means that some $UA^w_{x+1..y}$ users want to see the writing operation accomplished by other users on N1 without being observed in their writing activity. This situation is treated by adding a link between each $UA^w_{x+1..y}$ towards the selected cloned sub-sections. This action is represented in Figure 15c, where additional links are shown between the HypAgents $UA^w_{x+1..y}$ and the cloned sub-sections N1v1x, ..., N1vp .

The remaining UR^r that want to see the writing operation must express such intention to the UserCollector, that may allow this need by adding a new link.

5.6. *Case 4: More than One $UA^w_{1..k}$ Writer on a Same Acquaintance.*

The unique difference with respect to the previous state is on the need to synchronize the access to the same resource. The synchronization is monitored by the meta-level object UserCollector to better control the lock-unlock activities and the notification process. In particular, the synchronization is fundamental in WYSIWIS mode.

[‡]WYSIWIS is for What You See Is What I See.

5.7. *Case 5: There Exists at least One $UA^{w}_{1..k}$ Writer and at least One $UR^{w}_{k+1..n}$ Writer on a N1 Neighbour.*

Let N_{neib} be the node neighbour of N1. This situation must be considered during the version mechanism applied on N1. In fact, if UA^{w} and UR^{w} work on separate acquaintances, then the cloning is not applied on N_{neib}; in this way UA^{w} may refer to the N_{neib} HypAgent without affecting the consistency of the model. The sharing of resources may occur if both the UA^{w} and UR^{w} want to modify the same link between N1 and N_{neib}. In this case, the UserCollector provides appropriate synchronization mechanisms, according to the previously described cases.

5.8. *Considerations*

Let us focus our attention on some important points:

- Agent-based Systems handle problems disseminating tasks and knowledge among different autonomous entities. This allows us to treat the basic CSCW mechanisms [51] (notification, lock/unlock) in a more natural way, since the operational model supports directly collaboration issues.
- The Meta-level agents layer allows us to visualize and to browse configurations. In particular, if different alternative configurations exist in the same document, all of them are then visualized in such a way that the user may navigate them.
- Our model supports an easy management of alternative configurations (which is an important aspect of version control [23, 35, 37]); in fact, it is immediate to have them, since each past configuration is maintained by the system as a collection of (temporarily) inhibited entities.
- Our versioning mechanism does not support the user in an automatic merging operation. This feature is very important when different users establish (inter)dependence relationships on a same fragment of the hypertext. The community of hypertext research names "merging" the mechanism to treat this problem. Nevertheless the literature is rich in providing several (parallel) version models for hypertext applications, minor work has been spent on the merging [23]. A more recent approach in solving this problem consists in identifying the merging decisions that can be automatically taken by using as information the hypertext application data model and the group-work situation [39].
- In collaborative models, the notion of "current" configuration assumes subjective interpretations. That is, each user has its current configuration. In our model, this is possible thanks to the UserAgent, that allows us to obtain an different perspective for each user who interacts with the hypertext.

6. CONCLUSIONS AND FUTURE WORKS

Due to the richness of the design space of hypertext systems, the designers are faced with problems for the management of a large amount of heterogeneous data and control activities. This paper proposes a new methodology which is able to support the impact of this difficulty. Object-oriented concurrent programming offers high level tools to sketch software, characterized by strong dissemination of data and duties, as happens in hypertext systems. Moreover, OOCP allows a fine grained control on multi-tasking facility, that can be efficiently supported by multi-processor architectures. OOCP programming environments have the reputation of being computationally efficient [4, 41] in the universe of high level parallel languages. In our case, we adopted a "home-made" OOCP language instead of using a wide-known available languages [1, 7, 48]. This choice has inevitably led to lost of efficiency in time execution and memory allocation, as our developing target suffers of a prototypical status and further optimization techniques (compilation, optimal resource allocation, efficient process scheduling) are needed in order to improve the performances. The necessity of using a new OOCP language as developing tool was due essentially to our need to master inner features of the language, such as the message passing protocols. This problem has been easier to solve in our approach, where a high level language has been extended towards OOCP by meta-level programming strategies. The cost of building low-level structures and control strategies to handle knowledge and reasoning (our agent entity definition, our communication schemes, etc.) has been partially recovered in a more efficient prototyping activity, where for efficiency we mean faster designing phase, more effective ability to define and re-define the hypertext architecture, the availability to change the semantics of fundamental features of the language with minor efforts. The work presented in this paper has been focused on the discussion of the version management. The approach illustrated presents several advantages:

- it utilizes concurrent, asynchronous computation;
- it is described in a high-level fashion by means of specialization of the HypAgents behavior;
- it facilitates the management of complex CSCW features, such as the notification mechanisms, through appropriate collaboration policies defined among the agents;
- it is general and thus it can be adopted in hypertexts as well as in engineering databases;
- it supports efficiently dynamical linking.

These benefits stem from the underlying architecture, which is based on the agent-based programming paradigm. Our prototype has been tested with other versioning mechanisms [9], confirming the usefulness of our approach. Some open issues remain:

- improving HYPERCLAS with additional features in such a way as to define a complete language framework suitable for distributed hypertext design;
- interfacing HYPERCLAS hypertext applications with World Wide Web (WWW) which is nowadays the most important network hypertext resource.
- defining a flexible merging mechanism that allows a smart synthesis of the versions created by different users and maintained in this work as separate alternatives.

Acknowledgements — This work is supported partly by the the EEC HCM project VIM: a virtual Multicomputer for Symbolic application, ref. ERB4050PL-0186. We thank Anja Haake for her helpful remarks on a previous version of this paper, as well as the reviewers for their careful reading of this article and their thoughtful suggestions for modification.

REFERENCES

[1] ABCL. *ABCL: an Object Oriented Concurrent System.* Edited by A. Yonezawa, MIT Press (1990).

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA (1986).

[3] G. Agha A. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In *Research Directions in Object-Oriented Programming*, pp. 49–74, MIT Press, Cambridge, MA (1988).

[4] G. Agha S. Frølund W. Y. Kim R. Panwar A. Patterson D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel & Distributed Technology*, May 3–13 (1993).

[5] M. D. Beer. Versioning in Hypermedia. In *Proc. of the Workshop on Versioning in Hypertext Systems - ECHT'94*, pp. 7–12, September, Edinburgh (1994).

[6] D. G. Bobrow L. DeMichiel R. P. Gabriel G. Kiczales D. Moon S. Keene. Clos specification; x3j13 document 88-002r. *ACM-SIGPLAN Notices*, **23** (1988).

[7] J. P. Briot. Experience in Classification and Reuse of Synchronization Schemes. *Proc. 2nd International Symposium on Object Technologies for Advanced Software - ISOTAS'96*, Springer-Verlag's LNCS, March, Kanazawa, Japan (1996).

[8] DAI. *Distributed Artificial Intelligence.* L. Gasser and M. H. Huhns editors, Morgan Kaufman (1989).

[9] A. Dattolo A. Gisolfi. Analytical version control management in a hypertext system. In *Proc. 3th Int. Conf. on Information and Knowledge Management - CIKM'94*, pp. 132–139, Nov. 29 - Dec. 2, NIST, Gaithersburg, MD, ACM Press (1994).

[10] A. Dattolo V. Loia. Conceiving Collaborative Version Control for agent-based conceived Hypertext. In *Proc. on The Role of Version Control in CSCW Applications - ECSCW'95*, September 10, Stockholm, Sweden (1995).

[11] A. Dattolo V. Loia. Agent-based Design of Distributed Hypertext. *Proc. 11th Intern. ACM Symposium of Applied Computing - SAC'96*, February 17-18, Philadelphia, Pennsylvania (1996).

[12] M. D. Schwartz N. M. Delisle. Contexts - a partitioning concept for hypertext. *ACM Transactions on Office Information Systems*, **5**(2):168–186 (1987).

[13] L. Evans L. Anderson G. Crysdale. Achieving flexible autonomy in multiagent systems. *Applied Artificial Intelligence*, **6**(11):103–126 (1992).

[14] A. M. Fountain W. Hall E. Heath H. C. Davis. Microcosm: An open model for a hypermedia with dynamic linking. In *Proc. ACM Conf. on Hypertext - ECHT'90*, pp. 298–311, November, INRIA, France (1990).

[15] F. Garzotto P. Paolini B. Schawbe. HDM - a model based approach to hypertext application design. *ACM Transaction on Information Systems*, **11**(1):1–26 (1993).

[16] L. Gasser N. F. Rouquette R. W. Hill J. Lieb. Representing and using organizational knowledge in DAI systems. In *Distributed Artificial Intelligence*, **2**:55–78, L. Gasser and M. H. Huhns editors, Morgan Kaufman (1989).

[17] L. Gasser. Agent organizations for information retrieval and electronic commerce: the next frontier. In *Proc. Workshop on Heterogeneous Cooperative Knowledge-Base, International Symposium FGCS'94*, pp. 49–63, December 15-16, Japan (1994).

[18] A. Gisolfi V. Loia. Designing complex systems within distributed architectures. *International Journal of Applied Artificial Intelligence*, **8**(3):393–411 (1994).

[19] I. Goldstein D. Bobrow. A layered approach to software design. In D. Barstow H. Shrobe E. Sandewall, editor, *Interactive Programming Environments*, pp. 387–413. McGraw-Hill (1984).

[20] K. Grønbæk R. H. Trigg. Design issues for a Dexter-based hypermedia system. *Communications of the ACM*, **37**(3):40–49 (1994).

[21] A. Haake J. M. Haake. Take cover: Exploiting version support in cooperative systems. In *Proc. Conf. on Human Factors in Computing Systems - INTERCHI'93*, pp. 406–416, April, Amsterdam (1993).

[22] A. Haake. Under CoVer: The implementation of a contextual version server for hypertext applications. In *Proc. ACM Conf. on Hypertext - ECHT'94*, pp. 81–93, September, Edinburgh (1994).

[23] A. Haake J. Haake D. Hicks. On Merging Hypertext Networks. In *Proc. on The Role of Version Control in CSCW Applications - ECSCW'95*, September 10, Stockholm, Sweden (1995).

[24] F. G. Halasz. Reflections on Notecards: Seven issues for the next generation of hypermedia systems. *Communications of ACM*, **31**(7):836–852 (1988).

[25] F. G. Halasz, M. Schwartz. The Dexter hypertext reference model. (K. Grønbæk and R. H. Trigg Eds.) *Communications of the ACM*, **37**(3):30-39 (1994).

[26] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, **8**(3):323–364 (1977).

[27] C. Hewitt. Open information systems semantic for distributed artificial intelligence. *Artificial Intelligence*, **47**(1-3):79–106 (1991).

[28] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, **22**(4):375–408 (1990).

[29] K. Kimbrough O. Lamott. *Common Lisp User Interface Environment.* Texas Instruments Inc. (1990).

[30] H. Lieberman. Concurrent object-oriented programming in ACT-1. In *Proc. Conf. OOCP'87*, pp. 9–36 (1987).

[31] Y. Lashkari M. Metral P. Maes. Collaborative Interface Agents. *Proc. of AAAI'94*, (1994).

[32] V. Loia, M. Quaggetto. High level management of computation history for the design and implementation of a prolog system. *Software-Practice & Experiences*, **23**(2):119–150 (1993).

[33] V. Loia, M. Quaggetto. Integrating object-oriented paradigms and logic program: The opla language. In *Proc. 9th Int. Conf. on Knowledge-Based Software Engineering - KBSE'94*, pp. 158–164, September 20-23, Monterey, CA, IEEE Press (1994).

[34] The Multitasking Facility: Lucid Common Lisp Advanced User's Guide. Lucid Inc. (1988).

[35] H. Ludwig. Accessibility of Versions as Means of Handling Large Interdependent Object Spaces in Corporate Planning Environments. In *Proc. on The Role of Version Control in CSCW Applications - ECSCW'95*, September 10, Stockholm, Sweden (1995).

[36] C. Maioli S. Sola F. Vitali. Versioning issues in a collaborative distributed hypertext systems. *Technical Report UBLCS-93-6, Bologna* (1993).

[37] B. Magnusson U. Asklund S. Minör. Fine-Grained Revision Control for Collaborative Software Development. *Proc. of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Los Angeles, California, USA, December 7-10, *ACM Sofware Notes*, **18**(5):33–41 (1993).

[38] C. C. Marshall F. M. Shipman. Spatial hypertext: Designing for Change. *Communications of ACM*, **38**(8):99–97 (1995).

[39] J. P. Munson P. Dewan. A Flexible Object Merging Framework. In *Proc. of CSCW'94*, pp. 231–241, October, Chapel Hill, NC, ACM Press (1994).

[40] J. Nielsen. ***Hypertext and Hypermedia.*** Academic Press (1990).

[41] OOCP. *Object Oriented Concurrent Programming.* S. Matsuoko A. Yonezawa, editor, MIT Press (1993).

[42] K. Osterbye. Structural and cognitive problems in providing version control for hypertext. In *Proc. ACM Conf. on Hypertext - ECHT'92*, pp. 33–42, December, Milano, Italy (1992).

[43] A. Rizt L. Sauter. Multicard: An open hypermedia system. In *Proc. ACM Conf. on Hypertext - ECHT'92*, pp. 4–10, December, Milano, Italy (1992).

[44] T. Rodden. A survey of CSCW Systems. ***Interacting with Computers***, **3**(3):319–353, December (1991).

[45] D. E. Shackelford J. B. Smith F. D. Smith. The architecture and implementation of a distributed hypermedia storage system. In *Proc. ACM Conf. on Hypertext - ECHT'93*, pp. 1–13, Nov. 14-18, Seattle, Washington, USA (1993).

[46] L. F. G. Soares N. L. R. Rodriguez M. A. Casanova. Nested Composite Nodes and Version Control in Hypermedia Systems. In *Proc. of the Workshop on Versioning in Hypertext Systems - ECHT'94*, pp. 39–46, September, Edinburgh (1994).

[47] J. C. Tang. Findings from observational studies of collaborative work. *Int. J. Man-Machine Studies*, **34**(2):143–160 (1991).

[48] C. Tomlinson M. Scheevel V. Singh. Report on Rosette 1.1. *TR Number ACT-OODS-275-91, Microelectronics and Computer Technology, Austin*, TX, July (1991).

[49] V. Tschammer T. Magedanz M. Tschichholz A. Wolisz. Cooperative management in open distributed systems. *Computer Communications*, **17**:717–728 (1994).

[50] E. J. Whitehead K. M. Anderson R. N. Taylor. A Proposal for Versioning Support for the Chimera System. In *Proc. of the Workshop on Versioning in Hypertext Systems - ECHT'94*, pp. 51–60, September, Edinburgh (1994).

[51] U. K. Wiil J. J. Leggett. Concurrency Control in Collaborative Hypertext Systems. In *Proc. ACM Conf. on Hypertext - ECHT'93*, pp. 14–24, Nov. 14-18, Seattle, Washington, USA (1993).

[52] M. J. Wooldridge, N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, **10**(2), June (1995).