International Journal of Software Engineering and Knowledge Engineering
Vol. 10 No. 3 (2000) 345–369

DISTRIBUTED INFORMATION AND CONTROL IN A CONCURRENT HYPERMEDIA-ORIENTED ARCHITECTURE

ANTONINA DATTOLO

Dipartimento di Matematica ed Applicazioni, "R. Caccioppoli",
Universitá degli Studi di Napoli "Federico II",
Via Cinthia 45 (Monte S. Angelo), 80126 Napoli, Italy
E-mail: dattolo@matna2.dma.unina.it

VINCENZO LOIA

Dipartimento di Matematica ed Informatica,
Universitá di Salerno, via S. Allende, 84081 Baronissi (SA), Italy
E-mail: loia@unisa.it

Received 1 October 1998
Revision 1 June 1999
Accepted 19 October 1999

The market for parallel and distributed computing systems keeps growing. Technological advances in processor power, networking, telecommunication and multimedia are stimulating the development of applications requiring parallel and distributed computing. An important research problem in this area is the need to find a robust bridge between the decentralisation of knowledge sources in information-based systems and the distribution of computational power. Consequently, the attention of the research community has been directed towards high-level, concurrent, distributed programming. This work proposes a new hypermedia framework based on the metaphor of the actor model. The storage and run-time layers are represented entirely as communities of independent actors that cooperate in order to accomplish common goals, such as version management or user adaptivity. These goals involve fundamental and complex hypermedia issues, which, thanks to the distribution of tasks, are treated in an efficient and simple way.

Keywords: Distributed hypermedia design, open hypermedia systems, configuration management, adaptive hypermedia systems, object-oriented concurrent design, actor-based models.

1. Introduction

The World Wide Web [1] (WWW or just Web for short) started at CERN as a project to connect a heterogeneous collection of information using a hypermedia document metaphor. In spite of the large diffusion, in its current form the WWW suffers from two drawbacks, it is static and it has a weak distributed architecture. Due to a strong evolution of technologies from centralised to decentralised systems, it is important to also change the software engineering perspective in interface design [2]. The current availability of information highways and of inexpensive network technologies modifies [3] the traditional perspective of hardware (see for instance *network computer*) and software (*agent-based software*). From a software standpoint, there is an interest in viewing software as an "intelligent" collection of agents that interact by coordinating knowledge-based processes [2,4,5]. In this way, software can be conceived as an open system, *large-scale information systems that are always subject to unanticipated outcomes in their operation and new information from their environment* [6]. Open systems realise the "manager paradigm"; the manager reuses and coordinates the work of expert individuals without necessarily understanding it.

In this paper, we present a complete concurrent distributed hypermedia model designed according to an object-oriented concurrent paradigm [7]. Distributed hypermedia are not a novelty in the literature. Obviously, the decentralisation of media and the co-operation of several users working simultaneously have stimulated scientists in investigating efficient and suitable tools and models to provide distributed processing in hypermedia. In conventional hypermedia systems [8–10], all the operations are carried out by an active central unit which exerts control on a set of passive components. In our model this situation is reversed. There is no main resource responsible for the global management but an aggregation of autonomous and independent actors, each of them embodying a behavioural responsibility and a partial perception of the other members of the actor community. This design perspective enables to formulate new software design approaches, but it raises complex questions about the effective construction of software. This paper reports a research project which aims to define and realise a new hypermedia framework by adopting the actor model as reference design model.

The structure of the paper is as follows. Section 2 introduces the abstract language used to formally describe our actor-based model of hypermedia, HyDe (acronym of Hypermedia Distributed Design). In Sec. 3, the storage layer of our model is presented in detail. The fundamental issue of version management is examined in Sec. 4, emphasizing how it offers a uniform treatment for atomic and composite components. Section 5 is dedicated to the run-time layer: the basic architecture of the hypermedia is extended with new actor classes in order to support efficiently adaptive navigation and presentation. A discussion and comparison with related works are followed by the conclusions.

2. Actor Formalism

In order to formally describe our actor-based model of hypermedia we use a simple notation, named ESAL. ESAL (Extended SAL) is an extension of SAL (Simple Actor Language), an abstract language defined in [11] to formalise basic aspects of actor oriented programming. We adopt ESAL as a descriptive tool to define our actor entities and their behaviours. The construct `Def` is used to define an abstract actor, myactor, according to this form:

```
(Def myactor
    {inherit-from-this-class}
    (acquaintance list)
    [communication list] )
```

An actor is described by specifying three elements: its superclass, its data part and its script part, respectively put between braces, parentheses and brackets. In particular, the communication list is a sequence of scripts which can be executed by myactor. The communication between a sender and one or more receivers is accomplished by the "send" command types:

- `send` allows an actor to send a point-to-point message;
- `send-multicast` allows an actor to send multicasting messages on the net;
- `send-now-multicast` is similar to the previous `send-multicast`, but it requires an "acknowledge" message from the receiver actors.

A general form of the send construct is the following:

send − type (script-name argument-list) to destination-list

where send − type . . . to is one of the send commands; script-name argument-list determines the script (with its arguments, if any) that the destination actors trigger once they have received the message, while destination-list, introduced by the keyword to, identifies the actor(s) to which the message is addressed.

3. The Storage Layer Model

The storage layer model constitutes the structure of the hypermedia as provided by its author. The main purpose of this layer is to maintain the persistent objects, the collection of which defines the hypermedia in terms of dynamic internal mechanisms. The storage layer is organised in two levels:

- The first level, named *Structural Level*, contains atomic nodes (named HypActors) and links (named HypLinks).
- The second level is named *Meta level* and constituted by composites (named Collectors).

In the following, we will use the term "StorActor" to indicate a generic actor belonging to the storage layer (HypActor, HypLink or Collector).

In the rest of this section we discuss all these actor classes in detail by comparing them with traditional counterparts known in other popular hypermedia models, in particular with Dexter [9] and Dexter-based models [12–14].

3.1. *HypActors*

HyDe overcomes the traditional concept of "node" by proposing an active perspective, the HypActor. The main idea is to introduce inside the node a number of important functionalities for the management of the node itself and for the control of external interactions. For this reason, the fundamental issues normally handled in traditional models by separate layers and functions, are directly accomplished by the nodes in our model.

```
(Def HypActor
  {Actor}
  (text picture sound
   to from toAnch fromAnch toConf fromConf currConf confRange
   cloneOf unaltered keys danglnk whoIncludesMe)

  [(accessor ...), (cloning ...), (freezing ...), (unfreezing ...),
   (find-frontier ...), (take-new-conf ...), (update-conf ...),
   (awakening ...), (change-references ...), (optimize-yourself ...), ...]  )
```

Fig. 1. HypActor class definition.

The ESAL code in Fig. 1 defines the HypActor class.

The HypActor contains the data part (in parentheses) and the control part (in brackets). These slots are added to the basic information (such as name, mbox, ...) inherited from the primitive class `Actor`. The meaning of some acquaintances follows:

- text/picture/sound: these slots are used to maintain pointers to media objects.
- to/from: these slots store the addresses of a particular class of actors, the HypLinks. HypLinks serve as actors which support link-based operations. to maintains all the links leaving the node, while from denotes the links entering the same HypActor.
- toAnch/fromAnch: they mark a region, an item or a substructure of a component as an end-point of a link. Anchors have no direction; the prefixes to and from are used only to create a correspondence between these acquaintances and to/from.
- toConf/fromConf: these slots maintain the configuration of the related anchors and links. Each element of toAnch has a corresponding to anchored object in a given configuration, present in the resource toConf.
- currConf: this slot maintains the current configuration of the HypActor.
- confRange: this resource dynamically updates and stores all the possible con-

figurations to which the actor may belong.

- cloneOf: if the actor is a clone, then this slot contains the address of the actor from which the clone has been originated.
- whoIncludesMe contains the list of Collectors that address the current HypActor.

The script section defines the possible task which the actor can accomplish. In the next sections, we will provide details of some of the most important scripts.

3.2. *HypLinks*

Links are entities that manage relations between other components. They represent a sequence of two or more "end-point specifications", each of which refers to a hypermedia component, or to a section of it. In particular, when a link refers to something more complex than an entire component, it is called a "span-to-span" link (like in Intermedia [15]). In our model, links are actors, i.e. HypLinks, whose definition is given in Fig. 2.

```
(Def HypLink
   {Actor}
   (from to fromAnch toAnch
    fromConf toConf currConf confRange
    dnglFlg dnglAnch
    cloneOf unaltered whoIncludesMe)

   [(resolver ...), (cloning ...), (freezing ...), (unfreezing ...),
    (find-frontier ...), (take-new-conf ...), (update-conf ...),
    (awakening ...), (change-references ...), (optimize-yourself ...), ...]  )
```

Fig. 2. HypLink class definition.

The information contained in the data part is similar to the HypActor's. For example, the slots to and from will contain two lists of end-points, the addresses of HypActor and/or Collector. These slots suggest the direction of the HypLinks, but they can be traversed in both directions. The HypLinks support very general multi-headed links and a variety of link subtypes, as *one-to-one* and *one-to-many* links. Specific HypLink acquaintances are:

- dnglFlg. This is a flag that signals whether the link is dangling or not.
- dnglAnch. This data contains the "dangling" anchor list.

In order to better discuss the semantics of the HypLink entity and to underline the difference with the Dexter model, in Fig. 3, we give a representation of the link component.

Three HypActors, i.e. A01, A02, and A03, communicate with the link entity L01. The HypLink entity contains useful information to identify the addressed anchors in the corresponding HypActors.

In particular, the slot from contains the address of the HypActor A01 in which an end-point is given by the textual anchor txt[i,i'] in the corresponding resource fromAnch. The slot to is a list which addresses two HypActors A02 and A03 in the corresponding internal parts identified by the acquaintances toAnch; more precisely, the first anchor is a graphical end-point (img[j,j']), while the second is a reference to the whole HypActor A03. This last anchor is similar to the *whole-component anchors* supported by the DHM model [12]. With reference to Fig. 3, let us note that our approach differs from [16] with respect to anchor management, since we do not need to introduce distinct objects in order to identify anchors, but we simply add designed internal resources and scripts.

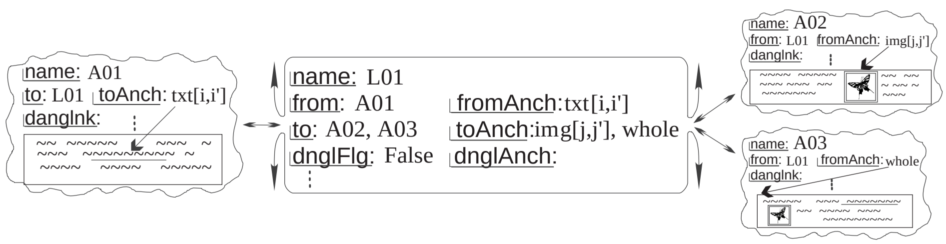

Fig. 3. Hyplink and HypActors.

In Fig. 3, observe that inside HypLink and HypActors there is enough knowledge to construct the link net; for instance, the slot from of the HypActor A02 contains information about the link (L01) having the anchor img[j,j'] (present in the slot fromAnch) as end-point. The active knowledge and control containing important context information [13] are used to increase the local computational power of the HypActors as well as the HypLinks.

The effect of this autonomy is particularly interesting during the execution of the resolver and accessor functions. According to Dexter's model, the resolver is the function responsible for "resolving" component specifications into the corresponding UIDs. Once the UIDs of the components are returned, the function accessor makes them accessible. In Dexter this process is accomplished in a centralised way using functions which are not local to the involved objects, while, in our approach, locality is protected because resolver and accessor are respectively scripts of the HypLink and HypActor communities. Moreover, the distributed concurrent facility of the model allows parallel handling of these functions.

For example, considering Fig. 3, the request to follow the link L01 in the HypActor A01, starting from the anchor txt[i,i'], provokes the call to the script resolver;

thus, the message is sent to the HypLink L01 which, in its turn, sends in multicast a message to each HypActor contained in the slot to (in our example, A02 and A03), asking to make accessible themselves. From a programming standpoint, this means to trigger the script accessor local to the HypActors. Once the execution of this script takes place, by addressing the specific anchors, then the span-to-span link process may be considered as terminated.

In Fig. 3, the slot danglnk related to the HypActors, and the slots dnglFlg and dnglAnch of the HypLink L01, are useful to manage possible dangling links. A dangling link can occur when modifications are applied to links: more precisely, a link is dangling if it has not at least two endpoints (source and destination). The treatment of the dangling link is an intrinsically dynamic process and imposes important aspects which are not present in the pure Dexter model. In Dexter-based hypermedia, it is possible to introduce dangling link management only by modifying basic issues of the Dexter models; for instance, in DHM [12] dangling link treatment is limited to the detection and re-link option in cases when the end-points component has been deleted.

In our approach dangling link management is supported in full without sacrificing the original model organisation. The first aspect to discuss is how a dangling link is represented.

In Fig. 4 the HypActor A01 of Fig. 3 has been modified. In the slot to the information related to the link L01 has been deleted. This deletion leads to a dangling link.

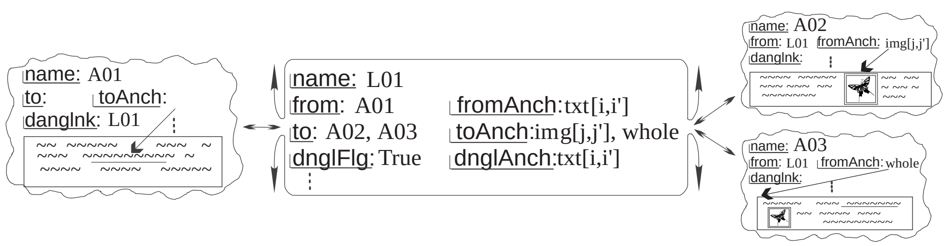

Fig. 4. Dangling link and anchors.

In our approach, this situation is treated by updating the local data of the corresponding HypLink, the slot dnglAnch in L01 is set to txt[i,i']. Since the slot from of the HypLink L01 contains a single reference, this means that no HypActor may access the HypLink in a standard way. This situation is represented by the value True in the slot dnglFlg; on the contrary, the value False (present in Fig. 3) reflects the fact that the link L01 was not dangling. This discussion describes only the data part useful to describe a dangling link. In fact, the effect of a dangling link is reported in different crucial contexts of the hypermedia model, namely:

- The strategy to follow a link is obviously affected and thus requires more complex control.

- In order to avoid the expensive proliferation of redundant nodes the version control mechanism should be applied. This means that the version control strategy must take into account the dangling link as an object belonging to different configurations.
- The dangling link has an important role even in information retrieval if we wish to acquire finer grained information associated with detached end-points.

3.3. *Collectors*

As pointed out by Halasz [17] "*composites would provide a means of capturing nonlink-based organisations of information, making structuring beyond pure networks an explicit part of hypermedia functionality*". Composites improve the modularity and thus the reusability of the hypermedia since they oblige the data to be maintained separately. In Dexter the composites encapsulate components. This means that composites act essentially as data containers; thus they lack the ability to provide more efficient organisation strategies not strictly reduced to composition by "copy" [14]. In HyDe the composites are represented by the class of actors called Collectors. This class:

- allows the author to structure the hypermedia by creating collections;
- allows the user to retrieve a collection of HypActors, HypLinks and/or (eventually other) Collectors, by search or by query; this collection will represent a direct reference to the (already existing) hypermedia portion. This type of composite is known in the literature as a *computed composite* [17];
- supports the user during browsing strategies. When the user browses, a collection is built on demand at runtime and provided to the user by activating an appropriate versioning mechanism in order to avoid unnecessary copies in the database. A similar collection is known in the literature as *virtual composite*.

In our model, the Collector plays the traditional role of container of atomic entities, i.e. HypActors, HypLinks, or of other composite entities, i.e. Collectors. This role of container is assumed only from a logic standpoint, in the sense that the Collector addresses HypActors, HypLinks and Collectors which are not necessarily encapsulated. In Fig. 5, the ESAL definition of the Collector class is provided.

According to the ESAL definition, a Collector is a kind of HypActor supplied with additional resources which are useful to gather more control and information on sections of the hypermedia. The main features of this class consist of the following data and services:

- collection/linkCollection: these slots are used to store a set of HypActor and Collector/HypLink addresses corresponding to a given collection;
- frontier: this slot contains the addresses of the incoming and outgoing HypLinks from the HypActors (and Collectors) in collection. In general, the frontier is the union of all the addresses contained in the acquaintances to and

from (inherited from the HypActor class) of the Collector and of the actors in collection.

- create/optimize/search-config: these scripts handle and improve the configuration management;
- explicit-query: this script supports the information retrieval facility.

```
(Def Collector
  {HypActor}
  (collection linkCollection frontier)

  [(create-config ...), (optimize-config ...), (search-config ...),
   (explicit-query ...), (take-new-conf ...), ...]  )
```

Fig. 5. Collector class definition.

At creation time the Collector sends a multicast message to the actors contained in its to and from slots; then it notifies the contacted actors that it maintains a reference to them and allows them to update the acquaintance whoIncludesMe. In this way, the important function WhoIncludesMe?, considered in [14] as an important issue not supported by Dexter, is easily supported in our model.

Now let us discuss in detail the version management, stressing the dynamic issues and their treatment in HyDe.

4. Version Management

Version management is important because it allows one to handle past states that can be re-used in future decisions, to keep track of the historical progress of the system and to support concurrent facilities in multi-user architectures. Its use also provides consistency support for the construction of widely distributed, open and interactive hypermedia models. Version management is viewed at two levels: versioning of node and versioning of structure. The need of distinguishing these levels arises from an obvious tendency to consider local, node-focused activities differently from those typically associated with a net-based structure (the same distinction is applied in software engineering [18] where, for single modules, *version control*, whereas for complete programs *configuration management*). Traditionally in hypermedia design the node is responsible for internal information [9] and, in some proposals [19], for its closer neighbours. For complex, external operations it is necessary to abandon the node entity and to rely on additional modules, which are designed to store a model of the net and to follow and maintain the evolution of the overall hypermedia. Our approach to version management is *uniform*, i.e. the node/structure distinction is broken, since in the actor model each single entity is able to obtain global information not by accumulating data in a single entity, but by applying concurrent cooperation schemes among de-centralised entities in such a way as to accomplish common goals. Thanks to this new perspective the version of node becomes a particular aspect of the most general version of structure. Hence, in this paper we focus our attention in describing the *configuration management* as the process that enables one to handle designed states of hypermedia evolution.

4.1. *Creating a configuration*

Hypermedia nodes can be created, deleted or modified by the user. The set of changes will produce a new configuration relative to the involved entities. This process of creating a configuration requires more attention if it is necessary to save the old configuration of the system. In our model, the process is made up of the following steps:

- focus which objects may change (as we will see these objects are not only those directly selected by the user);
- create copies of such objects, the so-called clones;
- freeze these objects in such a way as to transform them in passive entities;
- finally allow the user to apply the changes to the clones, to get the new current configuration.

In Fig. 6, the script create-config, defined in the ESAL language, provides a formal description of this process.

```
1  (create-config (coll linkColl newConfig)
2  (let* ((closure [send-now-multicast find-closure to linkColl])
3        (collToModify (append coll closure))
4        (clones [send-now-multicast] cloning to collToModify])
5        (linkClones [send-now-multicast cloning to linkColl])
6        (front [send-now-multicast find-frontier to collToModify])
7  (setq collection clones linkCollection linkClones frontier front))
8  [send-multicast freezing to collToModify linkColl])
9  [send-now-multicast (take-new-conf newConfig) to collection LinkCollection]
10 [send-now-multicast (update-conf newConfig) to frontier])
```

Fig. 6. The script to create configuration.

The section of hypermedia on which the user requires changes is identified by the two local resources coll and linkColl representing, respectively, the set of selected HypActor/Collector and HypLink objects (see row **1**). Other objects may be modified as a side effect: when the user modifies a HypActor, then its internal structure may change but this alteration does not affect the link information; if the user wants to modify a link then the involved entities are both HypActor and HypLink objects. The collection of the HypActors, addressed by the HypLinks which may be modified and do not belong to coll, defines the closure. Since closure depends on the current link selection, in row **2** it is explicitly computed by executing a multicasting message sent to the HypLinks addressed by the parameter linkColl. Finally, the resource collToModify, representing the complete area to duplicate (without links), is established (row **3**). Now the copying operation may start: a multicast message (row **4**) is addressed to the whole area composed of HypActors and Collectors which must be duplicated; the same action is repeated (row **5**) for HypLinks actors. As an effect of these messages only the actors that can be modified are cloned (see [20] for a more detailed discussion of this mechanism). The execution of the successive multicast (row **6**) serves to identify the HypLinks bordering on the cloned area. This frontier is assigned to the resource front. In row **7**, the clones' addresses and the frontier are added in local acquaintances of the Collector (collection, linkCollection and frontier). In this way, the user will access and modify the requested area acting on the Collector. Cloned actors are hence frozen (row **8**) and the clones replace the original ones in the new configuration. In more detail, as shown in row **9**, the execution of the script take-new-config updates the name (the clone, even though identical to the cloned, is a new different entity and hence has new address and name), the current configuration (the clone belongs to the just created configuration newConfig), and the configuration range. Different updating operations must be applied for the HypLinks in frontier. In fact, the HypLinks inside the frontier refer to a number of frozen actors; additional correct references must be established with the clones of the frozen actors. This operation corresponds to the statement of row **10**. Let us point out that the frontier is not cloned, but just updated; this action serves to bind the bulk of the new configuration with the rest of the hypermedia.

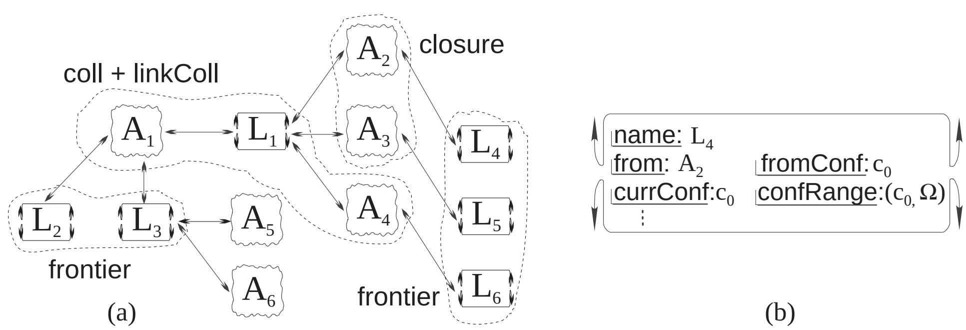

Fig. 7. Closure and frontier.

In Fig. 7(a) we show a simple case in which the user decides to modify the nodes A_1, A_4 and the link L_1.

In this example, A_1 and A_4 individuate the resource coll, L_1 the resource linkColl.

The closure is given by A_2 and A_3. Hence, the whole area to duplicate, i.e. collToModify, is the set of actors A_1, A_2, A_3, A_4, L_1. According to our definition, the frontier is composed by L_2, L_3, L_4, L_5, L_6, while the actors A_5 and A_6 remain unchanged and are not duplicated. Figure 7(b) provides the local environment of the HypLink L_4. The slot fromConf specifies the configuration of the entity A_2, i.e. the configuration labeled c_0. The dynamic evolution of the hypermedia extends the membership of unaltered actors to the sequence of next configurations created after c_0. This information is contained in the slot confRange where the value (c_0, Ω) establishes the membership scope, namely from c_0 up to the last created configuration labeled with Ω. The cloning mechanism is now applied to the actors in coll, linkColl and closure of Fig. 7(a). The effect of the cloning is shown in the next Fig. 8(a).

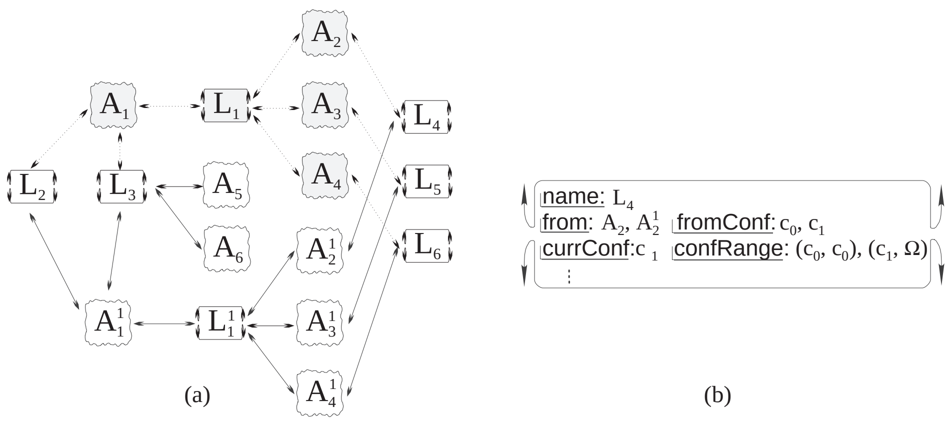

Fig. 8. Clones and cloned.

The cloned actors (A_1, A_2, A_3, A_4, L_1), shown in grey, become *suspended*, i.e., they become inaccessible from the standard external stimulus and messages, transforming themselves into static, frozen entities. A_j^i (or L_j^i) denotes the new name of the actor A_j (or L_j) in the i-th version. For simplicity, we suppose that the new occurring version is labeled with the superscript *1*. This notation is used to identify the clones which substitute the original actors. In Fig. 8(a), we note how the actors L_2, L_3, L_4, L_5, L_6 have now new bindings with the peripheral area of the copied hypermedia (A_1^1, A_2^1, A_3^1, A_4^1). In particular, in Fig. 8(b), we show the context of HypLink L_4 after the cloning. The labels c_0 and c_1 denote respectively the configuration related to Fig. 7 and the new configuration occurring in Fig. 8. Figure 8(b) shows the change of the internal knowledge: from contains a new HypActor A_2^1 and, similarly, fromConf contains c_1. The new values in currConf and confRange characterise the new configuration. Now, while the configuration c_1 represents a range of configurations, c_0 identifies only itself: for this reason, A_2 belongs exclusively to c_0 is exclusive, whereas A_2^1 belongs to each configuration created after c_1 (c_1 included).

We note that the duplication is applied to the actors which may potentially be modified. If after the cloning, the user does not perform modifications on a clone, then it is not necessary to maintain the clone itself. This situation may occur for several clones and thus has an impact on the overall process. For this reason, it is important to avoid useless copies, deleting unchanged clones but preserving the consistency of the hypermedia. In fact, a direct deletion of a clone cannot be applied since the clone exists in the hypermedia together with a number of connections. Hence, the deletion must be preceded by an updating operation which involves clones and cloned.

The same mechanism is applied also to Collectors (versioning occurs also for Collector entities in order to guarantee a uniform treatment of basic hypermedia issues).

5. The Run-Time Layer Model

In this section, we discuss the presentation of the storage layer components to the user, i.e. the run-time layer. Figure 9 shows the complete architecture including the new actor classes belonging to the run-time layer.

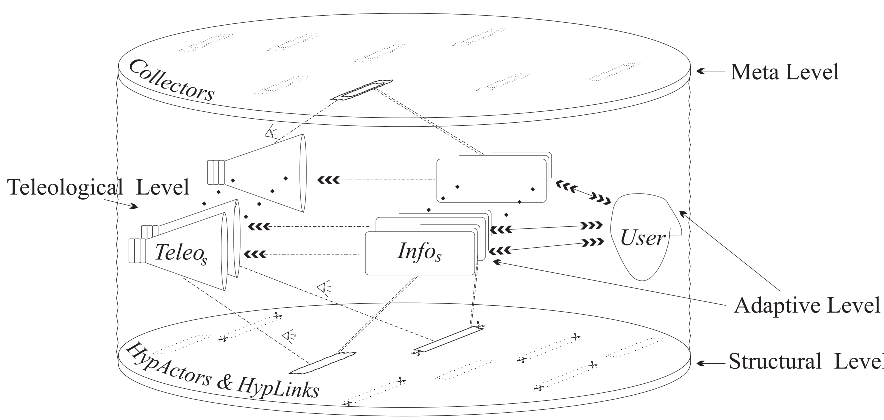

Fig. 9. The complete hypermedia architecture.

Two extra actor levels are added, the teleological and the adaptive levels. Here we briefly discuss their role.

- The *Teleological level* provides all the possible dynamic user perspectives of the hypermedia and interfaces the data/services provided by a certain StorActor and the user. This level contains a population of actors named *TeleoActors*. TeleoActors compose the front-end between the complete, anonymous hypermedia run-time layer and the user's expectations. Thanks to a distributed problem-solving strategy, TeleoActors specialise the contents of the related StorActor according to the user's profile.

- *Adaptive Level* contains InfoActors and UserActors. The *InfoActors* work as independent monitors of user behaviour, observing the human actions for each single hypermedia node: their main task is to form a web of prompt analysers specialised in recording user's actions on the hypermedia nodes. Furthermore, to each user corresponds a *UserActor* that plays the role of coordinator of the various InfoActors.

In the following sections we provide details about these new actor classes.

5.1. *TeleoActors*

The TeleoActors act as an adaptive interface between the storage layer and the user. In a classical approach, the instantiation of a component consists of a "copy" in cached memory space. When the user writes or edits this component then the cached instance is replaced with a new copy and then restored in the storage layer. A TeleoActor offers a more flexible mechanism of instantiation since it acts as a mediator between the storage level and the end-user by providing customised views on storage layer entities. Each view consists in adding/deleting data and services to the corresponding actor. Let us describe in Fig. 10 the ESAL definition of TeleoActor.

```
(Def TeleoActor
   {Actor}
   (stor info
     hypServices image
     inSuggestion brSuggestion cnSuggestion)

   [(apply-filter ...), (visualise ...), (tree-brws ...), (grph-brws ...), ...]  )
```

Fig. 10. TeleoActor class definition.

We detail some of the local acquaintances:

- stor. It is important to note that in the data part of the TeleoActor we have the connection with the corresponding StorActor. More precisely, when a StorActor instance is generated, automatically an instance of a TeleoActor is created, and coupled with the former via the stor acquaintance, containing the address of the instance.
- info. As a side effect, the previous mechanism couples an instance of a TeleoActor with its corresponding InfoActor, addressed by this slot.

- hypServices. This acquaintance may be viewed as a frame which depicts all the possible usable services on the stor. At the creation of a TeleoActor, the complete list of services is present. Successively, during the interaction between the user and the system, each TeleoActor may alter these services on the basis of information received from its InfoActor.
- inSuggestion/brSuggestion/cnSuggestion. These resources collect the user perspective and preferences, and are updated by the InfoActor. These three different data collect the user behaviour changes, in terms of three basic action categories: interface, browsing and contents.

The main role of a TeleoActor consists in specialising the use of its stor, according to the evolution of the preferences shown by the user during the browsing activity. The ability to shape the functionalities of the stor is given through a cooperation with the adaptive level: in fact, the knowledge about the user behaviour is acquired by an external entity, the InfoActor. It constitutes the main source of information useful to the TeleoActor in order to define which view must be applied on its stor; on the basis of the inSuggestion, brSuggestion and cnSuggestion information received by the InfoActor, the TeleoActor performs the script apply-filter on its stor. The execution of this script consists of three actions:

- copy the cached instance of the corresponding StorActor into a new temporary memory space;
- modify this instance by applying the specified view;
- replace in its local acquaintance image the old reference to the cached image with the new address of the last cached instance.

The script visualise carries out the visualization of the image containing the last user preferences.

5.2. *InfoActors*

InfoActors work as autonomous monitors of user behaviour. Each InfoActor monitors the user actions on the current StorActor. The InfoActor is created with enough knowledge to recognise the user actions. Of course, this knowledge is related to the domain content. The code in Fig. 11 shows its definition.

The existence of InfoActors leads to a simple and efficient organisational structure that enables the distribution of the user modelling activity viewed as a collaborative effort on a decentralised net. Here we describe the local acquaintances of the InfoActor:

- stor. This slot addresses the corresponding StorActor.
- teleo. This slot contains the address of the corresponding TeleoActor.
- usrAct. This resource identifies the UserActor.
- domain. In this slot the context knowledge given by the hypermedia author is stored.

```
(Def InfoActor
   {Actor}
   (stor teleo usrAct
     domain
     inSuggestion brSuggestion cnSuggestion
     inInfo brInfo cnInfo msInfo
     inTrust brTrust cnTrust msTrust
     inHints brHints cnHints msHints)

   [(notify-changes ...), (update-trust ...),
     (trace-in ...), (trace-br ...), (trace-cn ...), (trace-ms ...), ...]  )
```

Fig. 11. InfoActor class definition.

- inSuggestion/brSuggestion/cnSuggestion. These slots include the user preferences in term of three categories of actions: interface, browsing, contents. Their format is specific for TeleoActors.
- inInfo/brInfo/cnInfo/msInfo. These resources contain information necessary to qualify the user actions in terms of four basic categories: interface, browsing, contents, measurements. Each of these slots contains a sequence of identifiers representing the features used during the evaluation of the user behaviour. For instance, msInfo contains a sequence of numbers which quantify some user actions performed on the corresponding HypActor/Collector, such as:
  (a) the number of visits done by the user;
  (b) the average value of the time spent during the visits;
  (c) the number of help activations required by the user.
- inTrust/brTrust/cnTrust/msTrust. These four slots are used to record the trust values of the InfoActor.
- inHints/brHints/cnHints/msHints. These resources serve to receive the new behaviours sent by the UserActor.

Essentially, the InfoActor establishes two different communication schemes with its TeleoActor and the UserActor.

- The messages from the InfoActor to its TeleoActor enable the latter to be updated to the more recent user needs. This is possible thanks to the local

InfoActor's acquaintances which provide useful information about user behaviour changing, i.e. inSuggestion specifies the last user interface choices, brSuggestion represents the user browsing modalities and cnSuggestion represents the user content expectations. The former slot details those aspects which in the Dexter approach are stored in the *Presentation Specification* area of any component [9]. The distinction between these three acquaintances derives from the need to consider three different knowledge sources: the first source (inSuggestion) is used to personalise the interface by modifying the way in which the contents are displayed; the second resource (brSuggestion) depends strictly on the browsing style of the single user; finally, the last slot (cnSuggestion) indicates the views to apply to the contents of the HypActor/Collector but it is independent of the presentation modalities. These three resources are sent from the InfoActor to the corresponding TeleoActor by means of the script notify-changes.

- A communication activity also exists between InfoActor and UserActor. When the user navigates through the hypermedia by visiting different StorActors, the InfoActor, corresponding to the currently visited node, gathers the local and temporary user perspective in the acquaintances inInfo, brInfo, cnInfo and msInfo; these acquaintances are respectively the result of the tracing activities locally performed by the scripts trace-in, trace-br, trace-cn, and trace-ms. These four typologies of information, together with the corresponding trust values inTrust, brTrust, cnTrust and msTrust, are sent to the UserActor. The UserActor collects this information asynchronously and establish when and how the user model changes. The application of these changes adapts the local InfoActor knowledge to the new user perspective. This updating consists in modifying the local acquaintances inTrust, brTrust, cnTrust and msTrust in order to dynamically vary the relevance of the corresponding InfoActor observations. Let s be the suggestion inInfo (or respectively brInfo cnInfo msInfo), provided by the InfoActor to the UserActor and let p be the suggestion chosen by the UserActor and considered as relevant amongst all the suggestions received by all the activated InfoActors. The formula used to compute the new trust value inTrust (or respectively brTrust, cnTrust and msTrust) will be:

$$trust = clamp(0, 1, trust + \delta^{\star}_{s,p}(\gamma^{\star} trust^{\star}(1\text{-}wTrust)))\,. \qquad (1)$$

where

$$\delta_{s,p} = \begin{cases} +1 & \text{if suggestion } s = \text{UserActor preference } p \\ -1 & \text{if suggestion } s \neq \text{UserActor preference } p \end{cases}$$

and the *trust* maintains the old trust level in inTrust (or respectively in brTrust cnTrust msTrust) of the InfoActor, *wTrust* represents the corresponding value in wInTrust (or respectively in wBrTrust, wCnTrust and wMsTrust) provided

by the UserActor, γ is the trust learning rate, and the function $clamp(0, 1, v)$ ensures that the value of v always lies in $(0, 1]$.
The rationale behind the modelling above is the following. Formula (1) works in such a way as to increase (or decrease) the trust related to the local slots inInfo, brInfo, cnInfo and msInfo, when the information contained in them, corresponding to the suggestions sent to the UserActor previously, has (or has not) been effectively taken into account by the UserActor as meaningful to establish the current user behaviour. The amount the trust value rises and falls depends on the confidence of the other InfoActors in the suggestion provided by the current InfoActor. That is, if the suggestion of the InfoActor is not taken into account by the UserActor, and the average trust ($wTrust$) expressed by the other InfoActors is high, then the trust value should be penalized less heavily than an incorrect suggestion but with a lower average trust value. This inverse ratio is captured by the value (1-*wTrust*).
The formula to update the trust values of the single InfoActors constitutes the more relevant part of the script update-trust.

5.3. *UserActors*

In contrast to the locality of user observation made by the InfoActors, the UserActor is designed to reason globally about the user. In fact, its main goal is to infer new, general user preferences or needs in order to communicate them to the InfoActors which, in their turn, will be responsible for customising such general information in specific local targets. The UserActor is hence a collector-like actor, since it must organise the knowledge provided by InfoActor collections. The general mechanism used by the UserActor to deduce meaningful user changes is based on the concept that the user actions, observed by the InfoActors, modify a global trust level associated with the preferences/needs of the user. In this way, whenever the trust of a certain feature exceeds a meta-net threshold, then the corresponding feature is elected as a global user preference. Figure 12 shows the ESAL description of the UserActor class.

Considering the code of Fig. 12, we have the following semantics of the local acquaintances:

- pastInfos. This slot addresses the InfoActors that collaborate with the UserActor, providing it with observations on the user activity.
- futureInfos. This slot addresses the InfoActor collection interested in updating the user model; in particular, it is defined as the union of the current active InfoActors and their frontier extended by means of an iterative process k times (where k is a natural number dependent to the application).
- inInfo/brInfo/cnInfo/msInfo. These slots receive the different local user actions detected by the pastInfos.
- inTrust/brTrust/cnTrust/msTrust. These slots are used to store the sequence of trust values corresponding to the previous slots.

```
(Def UserActor
  {Actor}
  (pastInfos futureInfos
   inInfo inTrust wInTrust gwInTrust gInHints
   brInfo brTrust wBrTrust gwBrTrust gBrHints
   cnInfo cnTrust wCnTrust gwCnTrust gCnHints
   msInfo msTrust wMsTrust gwMsTrust gMsHints)

  [(return-wTrust ...), (return-gwTrust ...), (propagate-changes ...), ...] )
```

Fig. 12. UserActor class definition.

- wInTrust/wBrTrust/wCnTrust/wMsTrust. These slots contain, in correspondence to the previous data, the normalised weighted sum of the related trust values.
- gwInTrust/gwBrTrust/gwCnTrust/gwMsTrust. These slots contain the globally highest trust values selected from those contained in the previous slots. These values determine the UserActor choice of the current user view.
- gInHints/gBrHints/gCnHints/gMsHints. These slots dynamically maintain the user features corresponding to the previous highest trust values. They represent the hints (with the highest probability of interest) provided by the UserActor to the futureInfos InfoActors. These slots offer the current global user view.

As previously discussed, the InfoActors in a parallel and asynchronous way return to the UserActor local user views (the acquaintances inInfo, brInfo, cnInfo and msInfo) together with the related trust values (inTrust, brTrust, cnTrust and msTrust). The UserActor processes such information starting from the values contained in inTrust and obtains for each component inside inTrust a normalised weighted sum (wInTrust) by means of the execution of the script return-wTrust, according to the following general formula;

$$w\text{Trust}(x) = \frac{\sum_{i=1}^{n} w_i t_i}{\sum_{i=1}^{n} w_i} \quad (2)$$

where x may be one of the components in inTrust, t_i is the trust value related to the InfoActor that has sent the suggestion x, and w_i is the related weight given to the suggestion by the UserActor. Different criteria may be adopted to define the weighting strategy, such as time-based weighting or topic-based weighting. The highest trust value determines the relevant user preference to take into consideration. If this value is greater than the corresponding (current) meta-net threshold, i.e. gwInTrust, then this slot is updated with the new higher value. This corresponds to a new current meta-net threshold. This action is repeated for the remaining brTrust, cnTrust and msTrust slots. At the end of this execution, the UserActor has terminated its activity of deducing meaningful user changes and can propagates the changes to the interested InfoActors. Figure 13 shows this process.

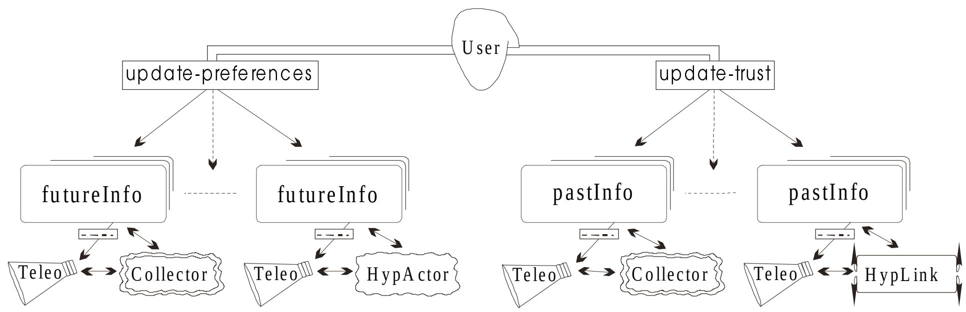

Fig. 13. Distributed update of the user model in multicasing.

The new user preferences (the slots gInHints, gBrHints, gCnHints and gMsHints) are sent to the interested InfoActors futureInfos. Now, if a new threshold is established, then it is necessary to update the local, distributed trusts contained in the sender InfoActors. This is done by sending the multicasting message update-trust to each InfoActor specified in pastInfos. propagate-changes is the script responsible for the described propagation of preferences and trusts.

6. Related Works

The actor-based model presented here modifies the way in which complex software systems, such as hypermedia, are generally conceived. The main difference compared to other significant proposals [8,9,21] is in the *distribution* of not only data, but also *of control*, and in the fundamental role of *communication*. As a result, the storage and run-time layers are composed of *active* entities, that embody enough knowledge to solve global goals by cooperative activities.

The role played by the HyDe actors is similar to that defined for the TAO entities [22], even though our architectures stresses in depth the issues of the communication and of the decentralization of the tasks and duties.

This section has been structured in two subsections which focused respectively on the storage and runtime layers of HyDe.

6.1. *Storage layer*

Some important hypermedia issues, which are hard problems to solve in Dexter [14], are successfully addressed in our framework:

- **Composites**
  The Collectors provide the visualisation and browsing of configurations. In our approach, they are dynamic objects, multi-versioned (in contrast to Dexter) and they are able, thanks to the strong interconnectivity realised by the communication, to support important functions, such as information retrieval and the WhoIncludesMe? [14].

- **Link service**
  As pointed out by Davis [23], *hypermedia systems may be classified by how they store both links and the information indicating their destination nodes and which areas within a node's content are "hotspots"*.
  In our approach the requested information is both embedded in the node (HypActor or Collector) and stored externally (in the HypLink). In this way, we gain several advantages:

  (a) StorActors are self-contained objects;
  (b) the dangling link is a natural instance of a generic link;
  (c) the presence of autonomous HypLinks permits the building of tools which navigate the links as well as tools to identify dangling links.
  (d) the users may select in which of a number of alternative webs to store a particular link;
  (e) thanks to communication and version control, changes on StorActors do not cause inconsistency in the data and bindings.

- **Version Control**
  Many hypermedia models [8,9] do not have the notion of configuration; this seems to be a strong restriction, since the basic principle of the hypermedia is the continuous evolution of its data. Few systems manage the version control [24–26]. Our approach realises in a uniform way version control on StorActors and offers a model which adheres to important trends [27].

- **Configuration as context**
  The concept of configuration is not time-restricted: more properly, the configuration corresponds to a context [13], characterised by a number of different features, among them, time.

- **Alternative configurations**
  Our model supports an easy management of alternative configurations, an important aspect of the version control [25,28]). In fact, they can be restored immediately since each past configuration is maintained by the system as a collection of (temporarily) suspended entities.

6.2. *Run-time layer*

Non-adaptive hypermedia system provides the same hypermedia pages and the same set of links to all users, even though different users need different information. This restriction is overcome by a few hypermedia models [29–32] but adaptive hypermedia systems are recently attracting considerable attention from the research community, as shown by a growing body of literature [33] and the existence of active research groups [34]. Of course, if a model of hypermedia is to be general, then it must support adaptivity.

- **Adaptive presentation and navigation**
  Adaptivity can be realised on two levels: adaptive presentation and adaptive navigation. Hypadapter [35] is one of the few systems that supports these two different ways of adaptivity. In our model, these two types of adaptivity are managed by the adaptive level and explicitly displayed by the teleological level; in particular, the suggestions provided by the InfoActors to the TeleoActors allow them to realise adaptive presentation and adaptive navigation. We point out that our process of instantiation of a component is a generalization of that proposed by [9]. In fact, it is not simply a copy of the component content, but a customised view of that component.
- **Generality of the architecture**
  The work of [30,33,36] emphasizes the need to design a general architecture for adaptive hypermedia, leaving aside particular strategies: the research direction is toward a kind of shell which simplifies creating adaptive hypermedia systems for different applications. Our model offers an interesting proposal in this sense: it is very general, independent from the usable adaptive strategies, and principally is not affected by the underlying storage layer.
- **Dynamic valuation of the user**
  The decentralisation and the communication are the basis for the high reactivity of our model, that is not limited to stereotype-based [29,37] or overlay-based schemes [38,39]. The *concurrent interaction* provides the continuous update of the actor knowledge in order to customise it to individual user habits and preferences. The *threshold-based mechanism* [40] makes possible the evaluation of the user behaviour changes.

7. Conclusions

In this paper we have modelled a complete hypermedia framework using the actor model as efficient paradigm of high-level distributed, concurrent programming. Actor-based languages may be viewed as an extension of script-based languages towards concurrent computing. This extension increases the benefits derived from the script languages, recently used as target tools for advanced hypermedia/multimedia architectures [16], in supporting efficient interaction with the user. The actor choice is due to the necessity to handle a simple, essential model of distributed computing, in order to highlight, as much as possible, the most important aspects of data and communication abstraction at the basis of a computational architecture rather than exploring new models of human reasoning. Using an extended actor-model, we have described, in a formal way, the details of our architecture; using a concurrent extension of CLOS, different prototypes have been realised [20,41,42]. This practice allowed us to learn much on the high-level distributed concurrent design methodology and to stabilise our model. This is confirmed by the fact that "hard" enhancements/extensions, such as a first proposal for CSCW environment, have been possible without reformulating the basic platform design concepts [20].

An going research activity consists in porting our distributed architecture on the WWW. The WWW in its current form does not support distributed applications in an easy and direct way. This lack has stimulated different proposals (see Web* [43], JOE [44], PageSpaces [2]). These efforts are characterised by a common feature: the role of Java as a middle- ware platform fully integrated in current Web technologies in order to allow really distributed applications. We intend to investigate this issue by considering recent directions that have been proposed as new guide-lines to develop interactive multimedia application for the Web [45].

Acknowledgments

This work has been partially funded by the European Social Fund of European Communities.

References

1. T. Berners-Lee, R. Cailiau, A. Luotonen, H. F. Nielsen, and A. Secret, "The World Wide Web", *CACM* **37** (1994) 76–82.
2. A. P. Ciancarini, D. Rossi and R. Tolksdorf, "Redesigning the web: From passive pages to coordinated agents in PageSpaces", in *Proc. 3rd Int. Symp. on Autonomous Decentralized Systems — ISADS97*, Berlin, Germany (April 1997).
3. M. Bieber, F. Vitali, H. Ashman, V. Balasubramanian and H. Oinas-Kukkonen, "Some hypermedia ideas for the WWW", in *Proc. of the 30th Annual Hawaii Int. Conf. on System Sciences*, Wailea, Hawaii (January 1997), (IEEE Press), pp. 309–319.
4. L. Gasser, "Social conceptions of knowledge and action: DAI foundations and open systems semantics", *Artificial Intelligence* **47** (1991) 107–138.
5. S. Goose, J. Dale, G. Hill, D. De Roure and W. Hall, "An open framework for integrating widely distributed hypermedia resources", *Proc. of the Int. Conf. on Multimedia Computing and Systems*, Hiroshima, Japan (June 1996), pp. 364–374.
6. C. Hewitt, "Open information systems semantic for distributed artificial intelligence", *Artificial Intelligence* **47** (1991) 79–106.
7. OOCP, *Object Oriented Concurrent Programming*, eds. S. Matsuoko and A. Yonezawa (MIT Press, Cambridge, MA, 1993).
8. F. Garzotto, L. Mainetti and P. Paolini, "Hypermedia design, analysis, and evaluation issues", *CACM* **38** (1995) 74–87.
9. F. Halasz and M. Schwartz, "The Dexter hypertext reference model", *CACM* **37** (1994) 30–39.

10. J. Nielsen, *Multimedia and Hypertext: The Internet and Beyond* (Academic Press, London 1996).
11. G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems* (MIT Press, Cambridge, MA, 1986).
12. Gronbæk and R. H. Trigg, "Design issues for a Dexter-based hypermedia system", *CACM* **37** (1994) 40–49.
13. L. Hardman, D. C. A. Bulterman and G. Van Rossum, "The Amsterdam hypermedia model: adding time and context to the Dexter model", *CACM* **37** (1994) 50–62.
14. J. J. Leggett and J. L. Schnase, "Viewing Dexter with open eyes", *CACM* **37** (1994) 76–86.
15. Yankelovich, B. Haan, N. Meyrowitz and S. Drucker, "Intermedia: The concept and the construction of a seamless information environment", *IEEE Computer* **21** (1988) 1–96.
16. MHEG-6, "Information technology — Coding of multimedia and hypermedia information — Part 1: MHEG object representation — Base notation (ASN.1)", ISO/IEC 13522-1:1997 (1997).
17. F. Halasz, "Reflections on NoteCards: Seven issues for the next generation of hypermedia systems", *CACM* **31** (1988) 836–852.
18. R. H. Katz, "Toward a unified framework for version modeling in engineering databases", *ACM Computing Surveys* **22** (1990) 375–408.
19. Y. Shibata and M. Katsumoto, "Dynamic hypertext and knowledge agent systems for multimedia information networks", in *ACM Hypertext '93 Proc.*, Seattle, Washington, USA (November 1993), pp. 82–93.
20. A. Dattolo and V. Loia, "Collaborative version control in an agent-based hypertext environment", *Information Systems* **21** (1996) 127–145.
21. P. Stotts and R. Furuta, "Petri net based hypertext: Document structure with browsing semantics", *ACM Trans. on Inf. Systems* **7** (1989) 3–29.
22. H. Chang, S. Chang, T. Hou and A. Hsu, "Tele-action objects for an active multimedia system", in *Proc. of 2nd Int. IEEE Conf. on Multimedia Computing and Systems*, Washington, D.C. (May 1995), pp. 106–113.
23. H. Davis, "To embed or not to embed ...", *CACM* **38** (1995) 108–109.
24. A. Haake and D. Hicks, "VerSE: Towards hypertext versioning styles", in *Proc. 7th ACM Conf. on Hypertext — Hypertext'96*, Washington, DC (March 1996), pp. 224–234.
25. 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*, Stockholm, Sweden (September 1995), pp. 57–61.
26. K. Osterbye, "Structural and cognitive problems in providing version control for hypertext", in *Proc. of the ACM Conf. on Hypertext*, Milano, Italy (November 1992), pp. 33–42.
27. L. F. G. Soares, N. L. R. Rodriguez and M. A. Casanova, "Nested composite nodes and version control in hypermedia systems", in *Proc. of the Workshop on Versioning in Hypertext Systems — ECHT'94*, Edinburgh (September 1994), pp. 39–46.
28. B. Magnusson, U. Asklundxi and S. Minör, "Fine-grained revision control for collaborative software development", in *Proc. 1st ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Los Angeles, California (December 1993), *ACM Software Notes* **18** (1993) 33–41.
29. C. Boyle and A. O. Encarnacion, "An adaptive hypertext reading system", *User Modeling and User-Adapted Interaction* (1993).

30. A. Kobsa, D. Müller and A. Nill, "KN-AHS: An adaptive hypertext client of the user modeling system BGP-MS", *Review of Information Science* **1** (1996).
31. N. Mathé and J. Chen, "New user-driven and context-based centered adaptive information access", *User Modeling and User Adapted Interaction* **6** (1996) 225–261.
32. J. Vassileva, "A task-centered approach for user modeling in a hypermedia office documentation system", *User Modeling and User Adapted Interaction* **6** (1996) 185–223.
33. P. Brusilovsky, "Methods and techniques of adaptive hypermedia", *User Modeling and User Adapted Interaction* **6** (1996).
34. AHS home page, (http://www.education.uts.edu.au/projects/ah/) (1999).
35. H. Hohl, H. D. Böcker and R. Gunzenhäuser, "Hypadapter: An adaptive hypertext system for exploratory learning and programming", *User Modeling and User-Adapted Interaction* **6** (1996) 131–155.
36. J. Kay, "Lies, damned lies and stereotypes: pragmatic approximations of users", in *Proc. 4th Int. Conf. on User Modeling*, *UM94*, Hyannis, Massachusetts (August 1994), pp. 175–184.
37. C. Kaplan, J. Fenwick and J. Chen, "Adaptive hypertext navigation based on user goals and context", *User models and User Adapted Interaction* **3** (2).
38. F. De Rosis, N. De Carolis and S. Pizzutilo, "User tailored hypermedia explanations", in *INTERCHI'93 Adjunct Proceedings*, Amsterdam (April 1994) pp. 169–170.
39. P. J. Scott and D. J. Ardron, "Integrating concept networks and hypermedia", in *Proc. World Conf. on Educational Multimedia and Hypermedia*, AACE, pp. 515–521.
40. Y. Lashkari, P., Maes and M. Metral, "Collaborative interface agents", in *Proc. of AAAI'94*.
41. Dattolo and V. Loia, "A concurrent, distributed model for hypermedia-based information systems", *Technical Report* (http://www.dia.unisa.it/antos/papers.htm) (1998).
42. Dattolo and V. Loia, "Active distributed framework for adaptive hypermedia", *International Journal of Human-Computer Studies* **26** (1997) 605–626.
43. G. Almasi, A. Suvaiala, I. Muslea, C. Cascaval, T. Davis and V. Jagannathan, "Web* — A technology to make information available on the Web", in *Proc. 4th IEEE Workshop on Enabling Technology: Infrastructure for Collaborative Enterprise* (1995).
44. Sun Microsystems, "JOE: Client/server applications for the web", *White Paper* (1996).
45. S. Bugaj, D. Bulterman, L. Hardman, J. Jansen, R. Lanphier, N. Layaida, J. Marsh, A. Rao, W. ten Kate, J. van Ossenbruggen, M. Vernick and J. Yu, "Synchronized multimedia integration language", *W3C Working Draft 09-November-97* (1997).