Int. J. Human-Computer Studies (1997) **46,** 605–626

Active distributed framework for adaptive hypermedia

ANTONINA DATTOLO AND VINCENZO LOIA
Dipartimento di Informatica ed Applicazioni, Università di Salerno, 84081 Baronissi (SA), Italy. email: antos/loia@dia.unisa.it

(Received 10 May 1996 and accepted in revised form 11 December 1996)

Navigation through large hypermedia information spaces is complex and is an important application area for adaptive hypermedia systems. User navigation can be best supported when the design of the hypermedia system is embedded in an evolutionary process model that takes into account the decentralization of data sources and the variety of users. The paper deals with distributed frameworks for open hypermedia systems; it focuses on the design work done to make adaptive an existing actor-based architecture for hypermedia. The approach follows the initial design approach used in the definition of the hypermedia platform, i.e. the actor-based computational model. We present in detail the new actor classes and the cooperative schemes which allow adaptation within the resulting architecture. © 1997 Academic Press Limited

1. Introduction

The explosion of the World Wide Web (WWW) (Berners-Lee, Cailiau, Luotonen, Nielsen & Secret, 1994) platform, in which different media interact, has shown the potential of hypermedia to provide rapid and effective access to information. This situation explains, on the one hand, the race to create new geographically distributed knowledge sources, and, on the other, the challenge to increase the intelligence of the information providers.

The achievement of this latter goal depends crucially on the ability to define useful adaptive interfaces. Only recently, the research community has been investigating the problems of adaptive hypermedia systems (AHSs). Research has been directed at proposing metrics evaluating the user's cognitive state (Mathé & Chen, 1996), in defining different forms of adaptation (presentation and navigation) (Brusilovsky, 1966), in using normative user models [over-lay (De Rosis, De Carolis & Pizzutilo, 1993) and stereotype (Rich, 1989; Kaplan, Fenwick & Chen, 1993) models or the combination of the last two (Kobsa, Müller & Nill, 1996; Vassileva, 1996)]. Work has also been directed at prescriptive user models (Kobsa, 1991), to understand the problems of orientation and comprehension (Thüring, Hannemann & Haake, 1995) and implement user-oriented querying assistance (Aberer, Lkas & Furtado, 1994). Work remains to be done, however, in defining effective and general architectures to support the above-mentioned approaches. The usual trend is to separate the user modelling package from the hypermedia environment (Kobsa *et al*., 1996), centralizing the intelligence of the user model in a unique specialized module.

In the past, the team responsible for defining appropriate interfacing systems in a company has usually taken into account the following two fundamental aspects.

- The variety of the media used to access information.
- The model of the database used in the company.

1071-5819/97/050605 + 22$25.00/0/hc960114

At present, the strict boundaries of the "local" company have disappeared in the search for global information management. The freedom of navigating in a vast and evolving domain affects profoundly the user's exploration of non-traditional data models. The existence of a global information infrastructure complicates the problem of adaptive interfaces, and the distribution and variety of the final system must be taken into account at the design level. In this new situation, in order to simplify the creation of adaptive hypermedia systems for different applications, we need to consider alternatives to the traditional, centralized design methodologies and consider new proposals in which data and services are decentralized and a unique, general "shell" is used (Kobsa *et al*., 1996).

We present here an experimental, general, distributed user-model architecture for distributed hypermedia-based information systems. It is general in the sense that it allows the implementation of different user models. It is completely distributed and requires the hypermedia model and the user interface to be strongly coupled in order to obtain an effective user interface. The framework is viewed as an "open system" (Hewitt, 1991), i.e. an environment in which a continual flow of new information originates from numerous actors (Agha, 1986). Actors exploit large-scale concurrency in that they perform functions (*scripts*) concurrently but own local data (*acquaintances*). The decentralization of knowledge and tasks does not prevent the actors from managing global actions; in fact, designed cooperative and collaborative duties may be used to coordinate their local actions. Actors communicate via message-passing.

The paper is organized as follows. Section 2 discusses our motivation, pointing out the role of the object-oriented concurrent programming paradigm as a key issue for the design of distributed interactive systems. The computational model used to implement the high-level open hypermedia architecture is briefly described. Beginning from this brief description, we present the storage layer in Section 3. The extension of the architecture for adaptive interaction is then discussed in Section 4. The software platform and some practical experiments are given in Section 5, followed by related work in Section 6. Conclusions and future ideas are outlined in Section 7.

2. Software design methodologies for distributed interactive systems

One of the main objectives of software engineering is to develop embedded systems that provide interactive services over time that adapt to clients' needs. Object-oriented technology radically changes the traditional software engineering perspective, both in formal and practical aspects (Wegner, 1995*a*). This evolution transforms "classical" closed Turing machines into open systems "*that express on-line interaction with external processes as well as the passage of time*" (Wegner, 1995*b*).

Structured programming is based on software development through progressive refinements. Object-oriented programming follows this discipline but strengthens the abstraction level through a stronger use of abstract data types and information hiding concepts. In the last decade, several programming paradigms suitable for concurrent programming have been proposed but object-oriented concurrent programming requires a less radical break from the "conventional" programmer's mentality. Modelling software as a collection of autonomous cooperative agents is a natural evolution of object-level languages. In fact, an object-oriented program is already conceived in terms of autonomous objects which could be executed in parallel. However, the "classical" notion of object is too vague to support large-scale concurrency because it limits the amount of parallelism available.

The actor model (Agha, 1986; Agha & Hewitt, 1988) satisfies the double requirements of high-level programming and efficiency. Actors combine object-oriented and functional programming to make it easier to use the concurrency. Briefly, the actor model can be described as follows.

- The universe contains computational agents, called *actors*.
- Actors perform computation through asynchronous, point-to-point message passing.
- Each actor is defined by its state, mail queue and behaviour.
- An actor's state is defined by its internal data called *acquaintances*.
- An actor reacts to the external environment by executing its procedural skills, called *scripts*.

The actor model is the basis of object-oriented concurrent programming, one of the most important implementation paradigms in distributed artificial intelligence (DAI) level architectures (Briot & Gasser, 1992). Furthermore, actor-based languages are computationally practical; they can be efficiently compiled (Kim & Agha, 1992) and implemented realistically on distributed multiprocessor architectures (Agha, Houck & Panwar, 1992).

In Figure 1 we give a graphical representation of our actor-based computational model.

Each actor is represented as a frame composed of two parts, the data part (acquaintances) and the functional part (scripts). Each actor has a class name. Two different asynchronous communication strategies are supported.

- *Point-to-point*. This enables the communication between a sender/receiver couple. This protocol is depicted in Figure 1 by a single arrow crossing a rectangle in which the name of the message (that is a script name) is indicated. The script name identifies the task that will be accomplished by the receiver.
- *Multicast*. This enables the communication between a sender and a collection of receivers. This kind of message is represented by a set of arrows spanning in the direction of the addressed actors. The multicast message is an extension of the "classical" actor model currently adopted in some concurrent object-oriented languages (OOCP, 1993) to improve communication.

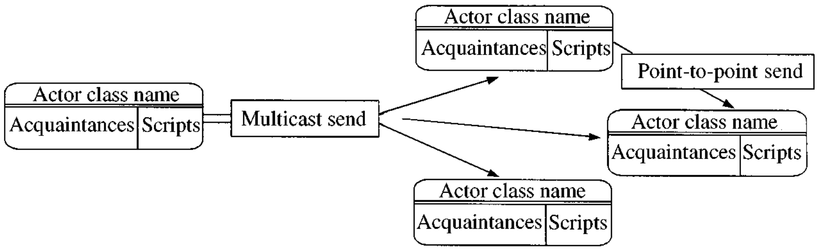


FIGURE 1. The actors and message types in our actor-based computational model.

3. Actor-based hypermedia model: storage layer

From a classical standpoint, a hypermedia (Nielsen, 1996) is a directed graph, composed of nodes containing basic data information and links which define the relationships between nodes. Users navigate from node to node by following links. Nodes can be either atomic or composite. The former contain data, text, graphics, sounds, images, whereas the latter provide alternative connections between the nodes or views. In either case, both atomic and composites nodes are *passive* objects in that they do not perform autonomous actions. In our model, we enrich the local competence of the nodes in such a way as to transform them into *active* objects, provided with enough knowledge to process internal as well as external tasks.

Each node is identified by an actor. An actor embodies passive information in its acquaintances, which are slots containing data. On receiving a stimulus, the actor may modify its internal status or interact with the external environment; these actions are performed by scripts, which are local functions associated with that actor. The social activity of an actor, i.e. the capability to establish collaborative goals, is possible because of the ability to contact "neighbour" entities.

In our model, we extend the usual point-to-point communication scheme of the pure actor model to allow multicasting and broadcasting message passing. This improves the flexibility and efficiency of distributed computation. Each actor, however, has only local knowledge and global tasks are achieved through collaborative cooperation. By decentralizing data and control, we achieve efficient task distribution management and we enforce the locality of the basic resources and, consequently, their use.

Figure 2 shows the storage layer of our model; this layer represents the structure of the hypermedia provided initially by its author. The main purpose of the storage layer is to manage the persistent storable objects that as a whole constitute the hypermedia. This layer is composed by two actor-based levels, as shown in Figure 2.

(1) *Structural level*. This first level contains the *HypActors*, which represent the "atomic" units of hypermedia information similar to well-known objects, such as cards (Halasz, 1988), frames (Garzotto *et al*., 1993) and slices (Isakowitz, Stohr & Balasubramanian, 1995). An important deviation from more traditional approaches is the dynamic nature of the HypActors. Conventional hypermedia objects are essentially data containers that delegate their operative functionality (e.g. user browsing) to external

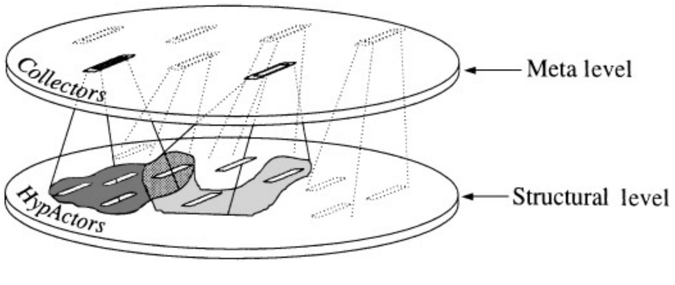


FIGURE 2. Our storage layer.

entities. This limitation is overcome in our model because each actor owns behavioural responsibility for itself and for the other members of the HypActor community.

(2) *Meta level.* This second level provides structure to the hypermedia model and the actors which compose the meta level are named *Collectors*. This level is necessary to allow the direct management and manipulation of HypActor collections. They allow access to non-linked hypermedia sections, provide a dynamic structuring mechanism based on the recursive composition of atomic (HypActors) and composite (Collectors) components (Willrich, Sénac, Diaz & de Saqui-Sannes, 1996) and maintain materialized views of the hypermedia, resulting, e.g. from a search or a query over node attributes. Together, the HypActors and the Collectors represent the most general, complete user perspective of the hypermedia.

In order to concentrate on the architecture, and specifically on the adaptive features of the model, in the following we do not discuss the problems associated with the effective representation of content embedded in complex multimedia data (tackled in Dattolo & Loia, 1996*b*). In the rest of the paper, we will use the term "StorActor" in order to indicate a generic actor belonging to the storage layer (HypActor or Collector).

3.1. THE HYPACTOR

Figure 3 illustrates a HypActor entity.

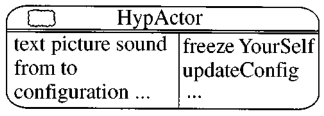


FIGURE 3. The HypActor class description.

The acquaintance part contains slots `text`, `picture`, `sound` to store textual, graphical and acoustic media information, while the slots `from` and `to` are used to contain the outgoing and incoming StorActors. Other slots, such as `configuration`, are used for configuration management. These slots allow the HypActor to fulfil the role of an atomic node in a traditional hypermedia. The HypActor differs from its traditional counterpart in its ability to perform actions by executing scripts. An example of such an action is versioning, which is carried out by the HypActor itself (discussed in Dattolo & Loia, 1995*a*, 1996*b*). `Freeze YourSelf` is used to freeze the content of a node when a new version of it is created, and `updateConfig` is used to update the node parameters related to a new configuration.

3.2. THE COLLECTOR

The Collectors provide a means of capturing non-link-based organizations of information, making structuring beyond pure networks an explicit part of hypermedia functionality. An author can use collections to allow a user to extract a portion of the hypermedia or to provide various browsing strategies. Collectors improve the modularity and, thus, reusability of the hypermedia since they allow data to be maintained separately. They

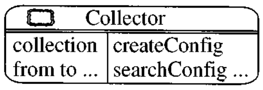


FIGURE 4. The Collector class description.

can also encapsulate other StorActors to address them more efficiently. An important acquaintance is `collection`; it contains the set of StorActor addresses on which the Collector exerts its role management and manipulation. For example, if the user activity produces a new configuration for a set of StorActors, the slot `collection` will contain their addresses. The task of creating a new configuration is carried out from a Collector, using the script `createConfig`, while the restoration of an old configuration is performed by the script `searchConfig`.

4. Actor-based hypermedia model: run-time layer

In this section we introduce the run-time layer which represents the part of our framework devoted to supporting the adaptive presentation of the storage components to the user. Figure 5 shows the general architecture; the run-time layer is identified by the teleological and the adaptive levels.

Teleological level. This level provides all the possible, dynamic user perspectives on the hypermedia; it is the interface between the data/services provided by a certain StorActor and the user. This level is composed of *TeleoActors*. As shown in Figure 5, each TeleoActor knows a unique StorActor and specializes its activity according to the evolution of the preferences shown by the user during browsing. The knowledge necessary to shape the functionality of the StorActor is obtained through cooperation with the adaptive level.

Adaptive level. This level contains the *InfoActors* and the *UserActors*. The InfoActors work as independent monitors of user behaviour observing human actions on each single hypermedia node. For each StorActor there exists a unique InfoActor associated with it. In this way, we have distributed and local user monitoring. The InfoActors learn new user habits and communicate this dynamic knowledge to the corresponding TeleoActor.

Each user is associated with a unique UserActor (see Figure 5) to coordinate the InfoActor reports and activities. The InfoActors inform the UserActor about all the recorded observations. This is done in a parallel and asynchronous way. The UserActor collects these different user perspectives and may thus recognize a new relevant user state. In this case, the UserActor sends this state update to selected InfoActors in order to permit them to acquire the new user behaviour and to update the related documentation.

It is clear that frequent and intensive cooperation activities among the different actor classes are required:

Primitive collaboration: *StorActors* $\Leftrightarrow$ *StorActors*. The structural and behavioural part of HypActors and Collectors establish a precise view of the hypermedia. At this level the cooperation activity provides "basic tasks" to do with the overall functioning of the hypermedia, such as configuration management (Dattolo & Loia, 1996*b*) or the search for a configuration (Dattolo & Loia, 1996*a*). These and other such tasks depend on information stored in the StorActor acquaintances, such as the slots `from` and `to` that provide a link structure between objects.

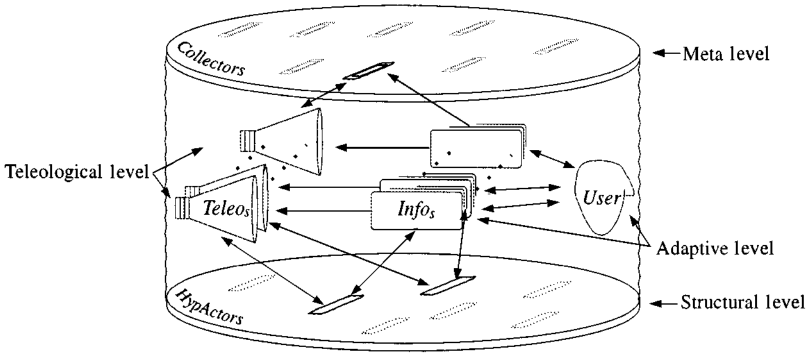


FIGURE 5. The hypermedia architecture.

Teleological collaboration: *StorActor ⇔ TeleoActor*, *InfoActor ⇒ TeleoActor*. The teleological level is the bridge between the users' goals and the hypermedia architecture. The interaction of the user with each hypermedia node is determined by the corresponding TeleoActor. Each TeleoActor is provided with a number of interface filters. Each filter modifies the functionality of the relative StorActor by pruning or adding views. The decision as to which filter to employ is taken by the TeleoActor using knowledge acquired from the user's behaviour. This knowledge is provided by the InfoActor that knows a user's particular habits. The collaboration activity is thus between StorActor–TeleoActor and InfoActor–TeleoActor. Figure 6 illustrates the teleological collaboration.

In Figure 6 the TeleoActor decides to activate the index-based interface from the three available interfaces.

Adaptive collaboration: *StorActor ⇔ InfoActor*, *InfoActor ⇒ UserActor*, *UserActor ⇒ InfoActors*. This collaboration involves three different tasks.

The first is the acquisition of local knowledge about the user. This task is achieved through a collaboration between the StorActor and its InfoActor. The InfoActor follows all the user activity on the current StorActor and keeps a trace of the user actions. This initial knowledge is used by the TeleoActor to apply a first local filtering.

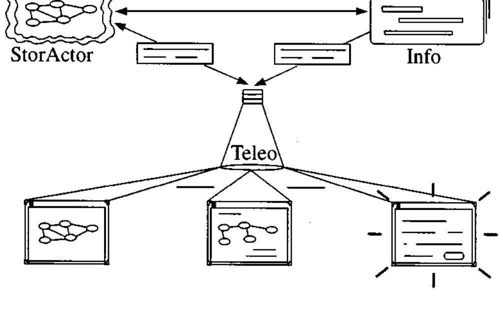


FIGURE 6. The teleological collaboration determines the current user view on each StorActor.

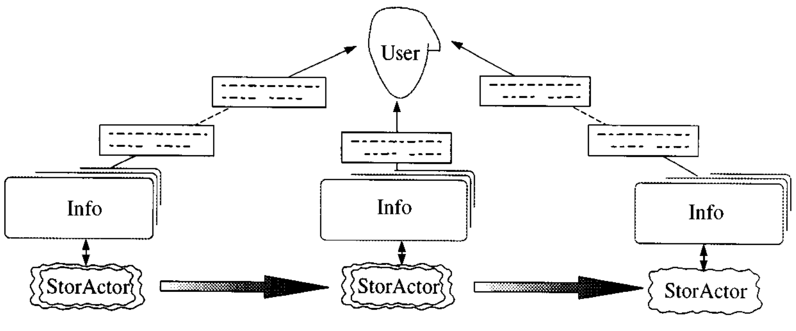


FIGURE 7. Asynchronous information flow from InfoActors to UserActor.

In the second task, the InfoActors forward to the UserActor the most significant observations on the user activity. The UserActor collects this information asynchronously so that during user monitoring the InfoActors and the UserActor work in an independent way. Figure 7 illustrates a simple example of our mechanism. The user navigates through the hypermedia by visiting different StorActors; the bold arrows show the navigation path. At each visit, the InfoActor, corresponding to the currently visited StorActor, gathers user habits, infers preferences and sends these to the UserActor. This result is a hypothesis (with a degree of confidence or trust) about a user's behaviour. All this is done in an asynchronous way because each InfoActor is completely autonomous.

In the third task the UserActor changes the current user model. The UserActor owns local duties that allows it to establish when and how the user model should change. To make a change, the UserActor informs the InfoActors about the need to update their user perspective. This action is performed using concurrent message passing in order to improve the adaptivity of the system. As stated previously, once an InfoActor acknowledges this message, it communicates to its TeleoActor the necessity to modify the user view of the corresponding StorActor. This knowledge allows each TeleoActor to filter data and services on the corresponding StorActor, so that the user update occurs in a deeper way that takes into account not only the local actions performed on the single node, but also the user's behaviour throughout the browsing. Figure 8 illustrates user-model updating. In this figure, the UserActor first sends, in multicast, the message `updateView` to the interested InfoActors. These are the so-called `futureInfos`.† The message contains the current global user-model updates. Each InfoActor that receives this model specializes it to forward a local user model to the related TeleoActor. To enable InfoActors to update their degree of trust (confidence) about a given user preference, a second kind of message is sent by the UserActor to those InfoActors (identified with the name `pastInfos`) that have generated a hypothesis about the current user preferences. This is done using the script `update Trust`.

The following sections describe in more detail the TeleoActor, InfoActor and UserActor entities.

† In our approach, a topological identification of this collection is obtained as the union of the currently active InfoActors and their close neighbours, extended by means of a recursive process k times (where k is a natural number dependent on the application).

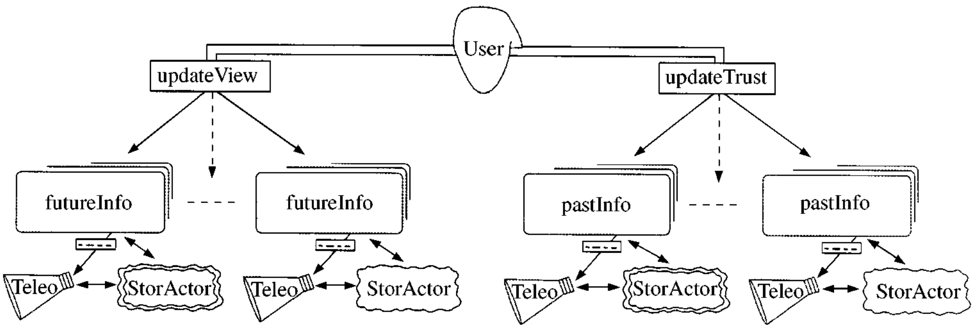


FIGURE 8. Concurrent, distributed update of the user model.

4.1. THE TELEOACTOR

The TeleoActors act as adaptive interface between the storage layer released by the hypermedia author and the user. These actors adapt the hypermedia functionality according to user behaviour. Each TeleoActor provides a given view of a corresponding StorActor which may be changed by adding or deleting data and services to that StorActor. Figure 9 illustrates a TeleoActor.

It is important to note that in the data part of the TeleoActor we have the link to the corresponding StorActor node. Whenever a StorActor instance is created, an instance of a TeleoActor is created automatically, and coupled with it via the local resource `storac` that contains the address of the StorActor. The same mechanism is applied to couple an instance of a TeleoActor with its corresponding InfoActor, identified by the slot `info`. All the services that may be used in the interaction between the user and the system are contained in the local resource `services`. This resource may be viewed as a frame containing knowledge.

When a TeleoActor is created, the complete list of services is present. This list may be altered by the TeleoActor during the interaction between the user and the system on the basis of information received from its InfoActor. The local resources `inSuggestion`, `brSuggestion` and `cnSuggestion` collect the user behaviour changes, in terms of three basic action categories: interface, browsing and content (detailed in Section 4.2). These resources are updated by the InfoActor `info`; the current, last user preferences are contained in the slot `image`.

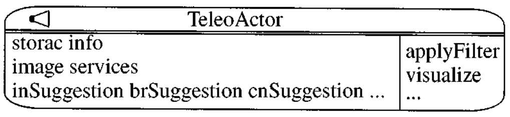


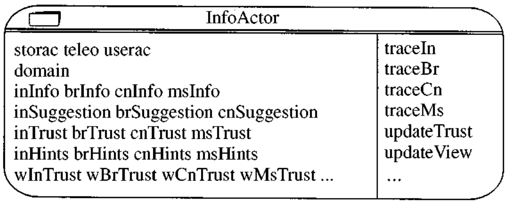
FIGURE 9. The TeleoActor class description.

4.2. THE INFOACTOR

Each InfoActor contains enough knowledge to recognize user actions. For good control and understanding of the user behaviour, the knowledge will depend on the domain content and is provided by the system designer. Figure 10 illustrates an InfoActor. As discussed before, each InfoActor knows a unique StorActor, a unique TeleoActor and the related UserActor, contained, respectively, in the slots `storac`, `teleo` and `userac`. The InfoActor's domain knowledge is stored in the acquaintance `domain`. The user monitoring is contained in the four resources `inInfo`, `brInfo`, `cnInfo` and `msInfo`; these objects can be viewed as frames containing information to identify the user actions, in terms of four basic categories: interface, browsing, content and measurements. Each of these classes depicts the local user activities. For instance, `msInfo` contains a sequence of numbers that quantify some user actions performed on the corresponding StorActor as follows.

- The number of visits by the user.
- The average time spent during the visits.
- The number of user help activations.

The user perspective evolves because of the cooperation with the UserActor. Essentially, the InfoActor establishes communication schemes with its TeleoActor and the UserActor. The messages from the InfoActor to its TeleoActor `teleo` enable the latter to be updated at run time to respond to recent user needs. The local InfoActor's acquaintances provide useful information about the changing user behaviour. `inSuggestion` specifies the last user-interface choices, `brSuggestion` represents the user browsing modalities and `cnSuggestion` represents the user content expectations. We note that `inSuggestion` details those aspects which in Dexter (Halasz & Schwartz, 1994) are stored in the *Presentation Specification* area of a component. The second communication activity is achieved through collaboration with the UserActor; here we discuss the messages from InfoActor to UserActor. The InfoActor possesses a local and temporary user perspective in the acquaintances `inInfo`, `brInfo`, `cnInfo` and `msInfo`, which are, respectively, the result of the tracing activities performed locally by the scripts `traceIn`, `traceBr`, `traceCn` and `traceMs`. These four typologies of information, together with the corresponding trust values `inTrust`, `brTrust`, `cnTrust` and `msTrust` are sent to the UserActor. As we will see in the next section, the UserActor collects this information asynchronously and establishes when and how the user model should change. The application of these changes induces the local InfoActor knowledge to adapt to the new user perspective. This updating consists of modifying the local



InfoActor	
storac teleo userac	traceIn
domain	traceBr
inInfo brInfo cnInfo msInfo	traceCn
inSuggestion brSuggestion cnSuggestion	traceMs
inTrust brTrust cnTrust msTrust	updateTrust
inHints brHints cnHints msHints	updateView
wInTrust wBrTrust wCnTrust wMsTrust ...	...

FIGURE 10. The InfoActor class description.

acquaintances `inTrust`, `brTrust`, `cnTrust` and `msTrust` in order to vary the relevance of the corresponding InfoActor observations. The rule used to change these values is similar to that used by Lashkari, Metral and Maes (1994). 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 the activated InfoActors. The formula used to compute the new trust values in `inTrust` (or respectively in `brTrust`, `cnTrust` and `msTrust`) is

$$trust = clamp(0, 1, trust + \delta_{s,p} * (\gamma * trust * (1 - wTrust))), \tag{1}$$

where

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

The second and third *trust* are the last trust levels maintained in `inTrust` (or, respectively, in `brTrust`, `cnTrust` and `msTrust`) of the InfoActor, *wTrust* represents the corresponding value in `wInTrust` (or, respectively, in `wBrTrust`, `wCn Trust` 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).

Formula 1 raises (lowers) the trusts related to the four local slots `inInfo`, `brInfo`, `cnInfo` and `msInfo`, when the information contained in them, corresponding to the suggestions sent previously to the UserActor, has been (has not been) effectively taken into account by the UserActor as meaningful for a new user habit. The amount by which the trust value rises or 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 one from 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 represents the more important part of the script `update Trust.`

4.3. THE USERACTOR

The UserActor organizes collections of InfoActors and the knowledge they provide. For this reason, whereas the goal of the InfoActors is to observe the local user actions, the UserActor studies such actions from a global standpoint in order to infer new user preferences. As each user preference is inferred, the UserActor delegates the InfoActors to consider such preferences in order to customize the TeleoActors for the new user preferences and habits. The involved InfoActors (`futureInfos`) are identified by their membership of the "focus of attention" area. This region contains all those InfoActors that are interested in this change. Various criteria (topological, node priority, etc.) may be adopted to define this area.

The cooperation activity that allows an adaptive evolution of the hypermedia is obtained by a continuous, asynchronous information flow between the InfoActors `pastInfos` and the UserActor. More in detail, the activated InfoActors in a parallel and asynchronous way return to the UserActor local user views (the slots `inInfo`, `brInfo`, `cnInfo` and `msInfo`) together with the related trust values (`inTrust`,

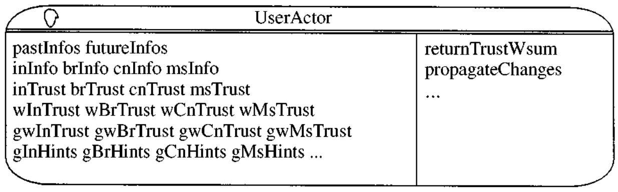

UserActor
pastInfos futureInfos
inInfo brInfo cnInfo msInfo
inTrust brTrust cnTrust msTrust
wInTrust wBrTrust wCnTrust wMsTrust
gwInTrust gwBrTrust gwCnTrust gwMsTrust
gInHints gBrHints gCnHints gMsHints ...
returnTrustWsum
propagateChanges
...

FIGURE 11. The UserActor class description.

brTrust, cnTrust and msTrust). The UserActor evaluates this information starting from the values contained in inTrust and obtains for each component inside inTrust a normalized weighted sum (wInTrust) by means of the execution of the script returnTrustWsum, according to the following general formula:

$$wTrust(x) = \frac{\sum_{i=1}^{n} w_i t_i}{\sum_{i=1}^{n} w_i}, \tag{2}$$

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 by the UserActor to the suggestion. Different criteria may be adopted to define the weighting strategy, such as time-based weighting or topic-based weighting. The highest trust sum determines the user preference taken into account. If this value is greater than the corresponding (current) meta-net threshold contained in 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 finished deducing meaningful user changes and can communicate them (the slots gInHints, gBrHints, gCnHints and gMsHints) to the interested InfoActors futureInfos (see Figure 11). This action is performed by the script propagateChanges. This script uses the multicast message passing protocol in order to gain concurrency in informing the InfoActor about the user-model changes. At its execution, a distributed, parallel update of the local user models takes placse, as shown in Figure 8. A local treatment of the update is delegated to each InfoActor belonging to futureInfos through the activation of the script updateView, while the update of InfoActor *trust* value is delegated to each InfoActor belonging to pastInfos through the activation of the script updateTrust.

5. Experimentation

Here we describe our experiments on a hypermedia environment designed to support logic object-oriented programming in OPLA (Loia & Quaggetto, 1996), a hybrid language developed from Prolog (Clocksin & Mellish, 1981) and CLOS (Bobrow, DeMichiel, Gabriel, Kiczales, Moon & Keene, 1988). We are not concerned with OPLA in itself but in the role played by our adaptive framework as an efficient user recognizer and dynamic user–machine interface model. We modified the "traditional" OPLA hypermedia programming environment, named Blue (Loia & Quaggetto, 1993*b*), using the new distributed model formalized in Dattolo and Loia (1996*b*) to obtain the distributed version of Blue, named DiBlue (Dattolo & Loia, 1996*a*). We then studied the effects of enhancing the system with the adaptive model described in this paper. The results are considered encouraging.

Figure 12 depicts the software modules used in the realization of DiBlue.

The first layer (corresponding to the Blue realization) consists essentially of CLOS and CLUE (Kimbrough & Lamott, 1990) (CLUE handles X-Windows objects). The extension of Blue towards a distributed framework has been made possible by using HyperClas, a concurrent, object-oriented programming language that supports the actor model (see Dattolo & Loia, 1995*a*, 1996*b*).

We give a feel for some of the features of the DiBlue environment by considering a scenario, illustrated in Figure 13, in which an OPLA user requires information about the OPLA class PRODUCT.

The interface in Figure 13 is organized in such a way as to provide meaningful information about the class, the superclasses and the subclasses, by specifying the data part and the procedural part, according to the inheritance mode. These data are shown in the upper pane, whereas the lower pane is dedicated to code editing. In particular, the

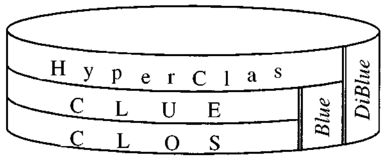
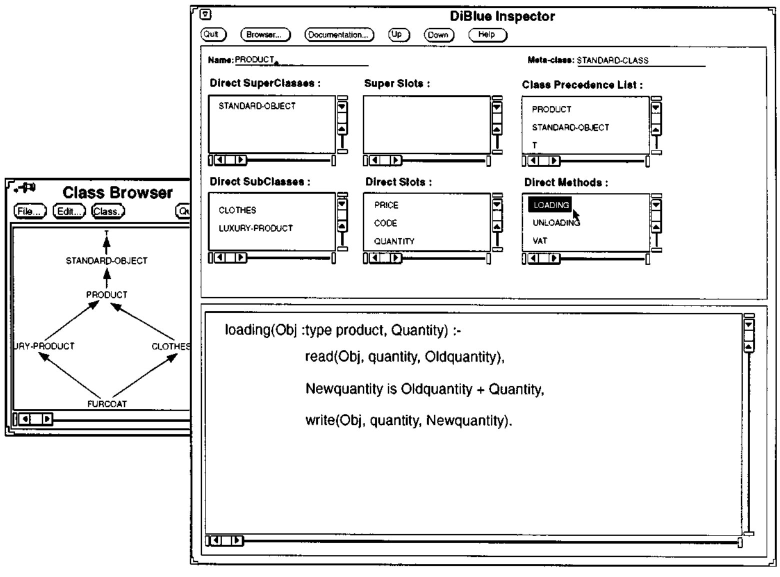


FIGURE 12. The DiBlue software platform.



Name: PRODUCT
Meta-class: STANDARD-CLASS

Direct SuperClasses : STANDARD-OBJECT

Super Slots :

Class Precedence List : PRODUCT, STANDARD-OBJECT, T

Direct SubClasses : CLOTHES, LUXURY-PRODUCT

Direct Slots : PRICE, CODE, QUANTITY

Direct Methods : LOADING, UNLOADING, VAT

```
loading(Obj :type product, Quantity) :-
        read(Obj, quantity, Oldquantity),
        Newquantity is Oldquantity + Quantity,
        write(Obj, quantity, Newquantity).
```

FIGURE 13. The DiBlue interface.

method `loading` in Figure 13 is highlighted when the user selects the method identifier in the upper `Direct Methods` area. If e.g. the user were to select the *documentation mode* (see top-level banner), different class-based information would be available in DiBlue; in Figure 13, on the leftmost side, we show a `Class Browser` window, in which a graph provides the inheritance ordering existing among the classes in relation with `PRODUCT`. During a session, the user may choose different browsing facilities. The `Class Browser` window may change by showing additional information, such as the slots, the methods or both. The adaptivity of the system makes it possible to recognize a user habit so that it can be provided automatically as a default mode. For the sake of brevity, let us denote by A, B and C the user choices corresponding to three habits regarding class browsing, viz. only classes (A), classes with methods (B) and classes with slots (C). We suppose that the user starts the programming activity with the class browsing mode A, shown in Figure 13, and that the UserActor local resources `gBrHints` and `gwBrTrust` contain the following values:

`gBrHints` = (A, …), `gwBrTrust` = (0.66, …).

The first resource corresponds to the user's preference for the current browsing mode, which is mode (A) since this mode has obtained the highest trust value (0.66) from the activated InfoActors. This trust value, which indicates the current relevant meta-net threshold value, is contained in the second resource. Now, let us suppose that the programmer's browsing modifies the user preferences. The InfoActors are responsible for detecting these meaningful operations and for conveying the new preferences to the UserActor. A possible flow of information is shown in Figure 14, which shows the following three important kinds of data.

- The identifiers of the InfoActors (`Info1`, `Info1`, … , `Info8`) that have sent messages to the UserActor.
- The pairs (preference, trust) which are received from the UserActor.
- The minutes (starting from 0, i.e. the first clock signal after the last threshold fixing) in which the UserActor receives the previous pairs.

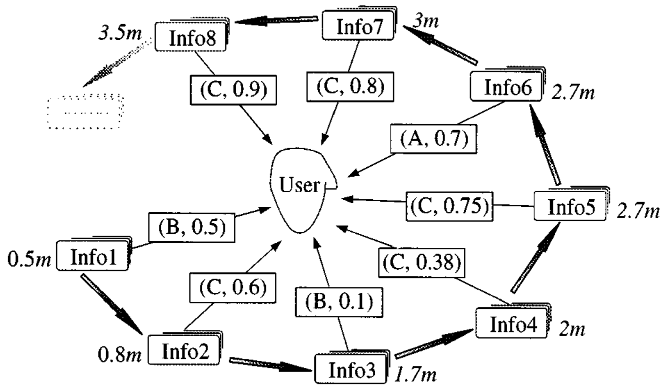


FIGURE 14. The second part of the adaptive collaboration.

Although other information is sent to the UserActor, we focus our attention only on the values reported in Figure 14.

On receipt of each of the previous messages, the UserActor computes a global estimation over the set of trust values corresponding to the current suggestion. This is accomplished according to formula 2, in which, in our example, the weights are determined using a time-based strategy. Table 1 provides the data related to each message sent to the UserActor.

From Table 1 it can be seen that as a function of the time expressed in minutes (first column), the InfoActors (second column) send their suggestions (third column) with the associated trust value (fourth column). The rightmost three columns show, for each of three different browsing modes A, B or C, the trust values obtained according to formula 2. Because the current choice was A (with meta-net threshold equal to 0.66), this preference remains the current one until, after 3.5 mins, a new threshold overcomes the previous limit. The graph in Figure 15 shows the progress of the trust functions pointing out the user preference change at 3.5 min.

TABLE 1
Data exchanges between InfoActors and UserActor. Upon reception, a new meta-net threshold is computed

Time	InfoActor	Suggestion	Trust	wBrTrust(A)	wBrTrust(B)	wBrTrust(C)
0				0.66	0	0
0.5	Info1	B	0.5	0.66	0.5	0
0.8	Info2	C	0.6	0.66	0.5	0.6
1.7	Info3	B	0.1	0.66	0.27338163	0.6
2	Info4	C	0.38	0.66	0.27338163	0.4753599
2.7	Info5	C	0.75	0.66	0.27338163	0.5847857
2.7	Info6	A	0.7	0.68584901	0.27338163	0.5847857
3	Info7	C	0.8	0.68584901	0.27338163	0.64908033
3.5	Info8	C	0.9	0.68584901	0.27338163	0.7119056

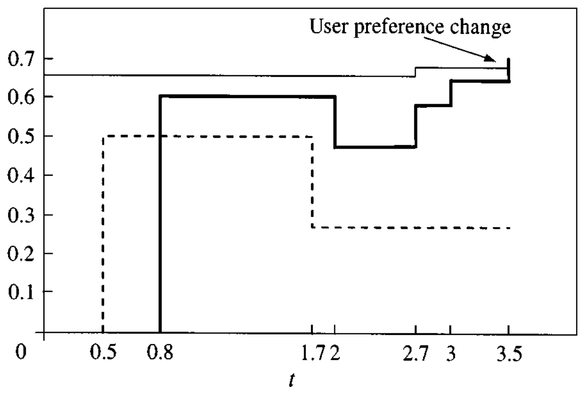


FIGURE 15. Three trust function plots; the function wBrSum(C) invokes the user preference change at time 3.5 min. —wBrTrust(A); --wBrTrust(B) and —wBrTrust(C).

This change provokes a decision, taken by the UserActor, to modify the user preferences, since now the two fundamental resources (`gBrHints` and `gwBrTrust`), associated with the UserActor, have the following new values:

$$\texttt{gBrHints} = (\mathrm{C}, \dots), \qquad \texttt{gwBrTrust} = (0.7119056, \dots).$$

For this reason, the UserActor activates the script `propagateChanges`, addressing it to the `futureInfos` and the `pastInfos` collections. Figure 16, with data values taken from our example, illustrates this action.

The InfoActors `Info1`, … , `Info8` represent the resource `pastInfos`, while the InfoActors `Infok`, … , `Infoj` are the `futureInfos`. The script `propagateChanges` sends the following.

- To `futureInfos`, the new user preferences (in the most general form, the slots `gInHints`, `gBrHints`, `gCnHints` and `gMsHints`) to each InfoActor in `futureInfos`. In this way, these InfoActors can apply the local script `updateView` and change the user preferences. In our example, the only resource interested in the change is `gBrHints`, the browsing mode C. As a result of this update, the class browsing mode changes from scenario A (leftmost window in Figure 13) to scenario C (Figure 17).

  The new habit C is now established and remains the "default" one until a different user habit is necessary.
- To `pastInfos`, the slots `gInHints`, `gwInTrust`, `inInfo`, `wInTrust`, `gBrHints`, … , `wMsTrust` necessary to update the local trust values; in our example, just the slots related to the browsing activity, the pairs (`gBrHints`, `gwBrTrust`) and (`brInfo`, `wBrTrust`) are sent, i.e. respectively, the values (C, 0.7119056) and (A, 0.68584901), (B, 0.27338163). As a result of this update, the `pastInfos` compute their new trust values, according to formula 1 given in Section 4.2, applying the local script `updateTrust`. Table 2 contains the result of this computation, showing for each InfoActor belonging to the collection `pastInfos`, the corresponding old and new trust values, as a function of the last UserActor preference. In this example, the trust learning rate γ was fixed at 1 for maximum reactivity.

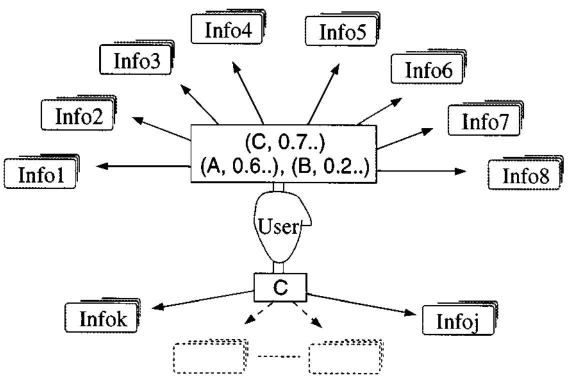


FIGURE 16. The third part of the adaptive collaboration.

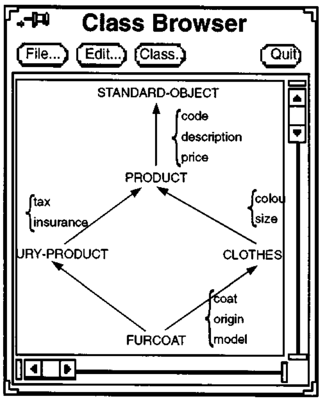


FIGURE 17. An example of the browsing mode C.

TABLE 2
Old and new trust values for the specified InfoActors

InfoActor	Old trust	New trust
Info1	0.5	0.13888889
Info2	0.6	0.77285664
Info3	0.1	0.02777778
Info4	0.38	0.48947587
Info5	0.75	0.9660708
Info6	0.7	0.48009431
Info7	0.8	1
Info8	0.9	1

6. Related work

AHSs have recently attracted considerable attention from the research community, as shown by a growing body of literature (Brusilovsky, 1996) and the existence of active research groups (Adaptive Hypertext & Hypermedia Web home page: `http://www.education.uts.edu.au/projects/ah/`). Initially, AHS appeared as an adaptive graphical interface able to support simple but frequent operations such as undo/redo strategies, active help and predefined plans-of-actions schemes. With the evolution of hypermedia models, and with the enormous diffusion of hypermedia applications to users from different social and professional classes, there is now a strong need for more efficient adaptive methods. Fortunately, the adaptive hypermedia community has benefitted from the research on user (typically student) modelling (Kobsa, 1993), especially in the field of artificial intelligence in education. The accomplishments in this area have been extremely useful in solving the difficult task of acquiring user's knowledge but little attention has been paid to the study of possible implications for the hypermedia architecture. In the current work in this area, two issues are emphasized as being relevant and underinvestigated.

- There is a need to design a general architecture for adaptive hypermedia. Leaving aside particular strategies, the research direction (Kay, 1994; Brusilovsky, 1996; Kobsa *et al.*, 1996) is towards a kind of shell that simplifies the creation of adaptive hypermedia systems for different applications.
- The relationship between AHS research and the actual dissemination of information on distributed environments, as the WWW (Brusilovsky, 1996; Thomas & Fischer, 1996).

Our work is directed at these goals and it should not be considered as a new user modelling method for hypermedia applications but a general, distributed, open actor-based framework for adaptive hypermedia systems. With this specification in mind, our approach differs from Kobsa *et al.* (1996), since our solution is not to conceive a "black box" which can be connected to an external application; our model is embedded in the hypermedia architecture and is difficult to "export" to hypermedia that do not share our design architecture. Nevertheless, due to our design perspective, our proposal can achieve the same level of abstraction and flexibility as can be achieved using black boxes. Our actor-oriented design perspective interprets adaptivity as an extension of the actor ontology. This paper recognizes as Maes (1994) and Thomas and Fischer (1996), i.e. the importance of an agent-based architecture as a useful model to support the dynamic customization of a system to a generic user, but is differentiates from both in the following two essential aspects.

- The use in our model of an abstract, completely distributed and modular approach in the hypermedia architecture and in particular for user modelling.
- The different, more general cooperation schemes among the agent populations.

In our approach, the strong task decentralization and the efficient interaction policies increase the dynamism and reactivity of the model, and is not limited to stereotype-based (Kaplan *et al.*, 1993; Boyle & Encarnacion, 1994) or overlay-based schemes (De Rosis *et al.*, 1993; Scott & Ardron, 1994). Furthermore, we underline that the two levels of possible adaptivity, adaptive presentation and adaptive navigation (Brusilovsky, 1996), are supported in few systems [Hypadapter (Boecker, Hohl & Schwab, 1990) is one of these]. In our model, these two types of adaptivity can be supported because of the cooperation activity between the adaptive and the teleological levels.

7. Conclusion and future work

Our goal was to enrich an existing hypermedia architecture (Dattolo & Loia, 1996*b*) with an appropriate user-modelling activity, conceived and developed with the same design perspective of the underlying system, i.e. the actor-level design approach. The organizational structure has been defined by using a previous platform. Within this scope, we have proposed a general, distributed architecture able to adapt hypermedia functions to the user behaviour. The bulk of our hypermedia architecture is composed of five categories of actors, three of them dedicated to user modelling. Previous experience in crafting-distributed cognitive diagnostic systems (Loia, 1994; Loia & Quaggetto, 1994) helped us in defining the overall architecture.

Our approach leads to various advantages.

Organizational structure. The user recognition is accomplished through a cooperation activity determined by an actor-based knowledge acquisition process. The specialization of InfoActors knowledge and duties for each hypermedia node allows us to handle local observations of the user better, personalizing metrics and strategies that depend on the various user characteristics. This organizational approach leads to a flexible architecture where the functionality can be extended without affecting its inner features.

Cooperation modes. Significant user-cognitive actions are detected by the InfoActors and sent to the UserActor. The UserActor collects the individual user profiles and works independently while InfoActors continue their activity. The UserActor applies specific learning strategies and takes a decision on when and where to adapt to user behaviour. In this case, it sends appropriate messages to InfoActors, that, in turn, communicate the new preferences to the relative TeleoActors. This method allows the system to be flexible and adaptive, while avoiding rigid schemes that do not support the dynamism and evolution of the user easily.

Conflicts. No conflict can arise between InfoActors, since the existence of a unique UserActor leads to the centralized management of the recognized preferences. This does not decrease the potential level of distributed operation. In fact, in order to use the concurrency of the model better, we are currently developing a collaborative, multiuser version of our architecture, where for each user we have an autonomous UserActor.

Distributed, decentralized computation. The StorActors are independent hypermedia nodes, the TeleoActors work in parallel, as do the InfoActors. Concurrency is introduced not only to use parallel technology but also as a metaphor for software design.

The architecture has been applied in the development of a hypermedia system that is used to support an object-oriented logic programming system (Dattolo & Loia, 1996*b*; Loia & Quaggetto, 1996). The results are sufficiently interesting to encourage further work although some problems exist in the current version.

There is too much work for the application engineer. The knowledge specification for each InfoActor requires considerable effort and we are investigating the possibility automating their construction by applying a meta-level definition scheme.

Few reasoning mechanisms are used by the UserActor. Currently, a threshold-based deduction strategy is supported. This is because our initial experimental effort was focused on providing a general and very flexible framework. The UserActor deduction strategies can be easily enriched by defining additional scripts (Lashkari, Metral & Maes, 1994; Maes, 1994).

The resolution of these issues are part of our ongoing research effort.

Another research direction is the generalization of our actor-based architecture to-wards open systems like the WWW. The WWW in its current form does not support distributed applications in an easy and direct way. This explains the latest trend devoted to overcoming this important deficit [see Web* (Almasi, Suvaiala, Muslea, Cascaval, Davis & Jagannathan, 1995), JOE (Sun Microsystems, 1996), PageSpaces (Ciancarini, Knoche, Tolksdorf & Vitali, 1996)]. A common feature of these different approaches is enforcing the role of Java as a middleware platform fully integrated in current Web technologies in order to allow really distributed applications. We intend to follow this direction; we have already designed a base class "Actor" which enables actor-computing in Java, and we plan to complete a port of the entire hypermedia architecture in short time. This facility is obtained via actor-based coordination policies that manage interaction activities in different layers of the platform.

We would like to thank Len Bottaci for providing valuable suggestions and useful corrections on the earlier version of this paper, and the anonymous reviewers for their useful and constructive comments.

Part of this work was supported by the EEC HCM project *VIM*; *a virtual Multicomputer for Symbolic Applications*, ref. ERB4050PL-0186.

References

ABERER, K., LKAS, W. & FURTADO, A. L. (1994). Designing a user-oriented query modification facility in object-oriented database systems. *Proceedings of the VI International Conference on Advanced Information Systems Engineering—CAiSE*94*, Utrecht, The Netherlands, June 1994, *Lecture Notes in Computer Science*, Vol. 811, pp. 380–393. Berlin: Springer Verlag.

Adaptive Hypertext & Hypermedia Web home page: `http://www.education.uts.edu.au/projects/ah/`.

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

AGHA, G. & HEWITT, A. (1988). Actors: a conceptual foundation for concurrent object-oriented program ming. *Research Directions in Object-Oriented Programming*, pp. 49–74. Cambridge, MA: MIT Press.

AGHA, G., HOUCK, C. & PANWAR, R. (1992). Distributed execution of actor programs. *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Vol. 586, pp. 1–17. Berlin: Springer Verlag.

ALMASI, G., SUVAIALA, A., MUSLEA, I., CASCAVAL, C., DAVIS, T. & JAGANNATHAN, V. (1995). Web*—a technology to make information available on the Web. *Proceedings of the 4th IEEE Workshop on Enabling Technology: Infrastructure for Collaborative Enterprise—WETICE'95.*

BERNERS-LEE, T., CAILIAU, R., LUOTONEN, A., NIELSEN, H. F. & SECRET, A. (1994). The World Wide Web. *Communications of the ACM*, **37,** 76–82.

BOECKER, H. D., HOHL, H. & SCHWAB, T. (1990). Hypadapter-individualizing hypertext. In D. DIAPER *et al.*, Eds. *Proceedings of the 3rd International Conference on Human-Computer Interaction, INTERACT'90*, pp. 931–936. Amsterdam: North-Holland.

BOYLE, C. & ENCARNACION, A. O. (1994). MetaDoc: an adaptive hypertext reading system. *User Modeling and User-Adapted Interaction*, **4,** 1–19.

BOBROW, D. G., DEMICHIEL, L., GABRIEL, R. P., KICZALES, G., MOON, D. & KEENE, D. (1988). Clos specification; X3J13 document 88-002R. *ACM-SIGPLAN Notices*, **23**. Distributed Artificial Intelligence.

BRIOT, G. P. & GASSER, L. (1992). Object-based concurrent computation and DAI. In N. M. AVOURIS & L. GASSER, Eds. *Distributed Artificial Intelligence: Theory and Praxis.* Drodrecht: Kluwer.

BRUSILOVSKY, P. (1996). Methods and techniques of adaptive hypermedia. *User Modeling and User Adapted Interaction*, **6**.

CIANCARINI, P., KNOCHE, A., TOLKSDORF, R. & VITALI, F. (1996). PageSpaces: an architecture to coordinated distributed applications on the Web. *Computer Networks and ISDN Systems*, **28,** 941–952.

CLOCKSIN, W. F. & MELLISH, C. S. (1981). *Programming in Prolog.* Berlin: Springer Verlag.

DATTOLO, A. & LOIA, V. (1995*a*). Hypertext version management in an actor-based framework. *Proceedings of the 7th International Conference on Advanced Information Systems Engineering-CAiSE*95*, 12–16 June, Jyväskylä, Finland, *Lecture Notes in Computer Science*, Vol. 932, pp. 112–125. Berlin: Springer Verlag.

DATTOLO, A. & LOIA, V. (1995*b*). Conceiving collaborative version control for agent-based conceived hypertext. *Proceedings of the Workshop on the Role of Version Control in CSCW Applications—ECSCW'95*, 10 September, Stockholm, Sweden, pp. 23–32.

DATTOLO, A. & LOIA, V. (1996*a*). Agent-based design of distributed hypertext. *Proceedings of the 11th International ACM Symposium of Applied Computing—SAC'96*, 17–18 February, Philadelphia, PN, pp. 129–136.

DATTOLO, A. & LOIA, V. (1996*b*). Collaborative version control in agent-based hypertext environment. *Information Systems*, **20,** 337–359.

DE ROSIS, F., DE CAROLIS, N. & PIZZUTILO, S. (1993). User tailored hypermedia explanations. *INTERCHI'93 Adjunct Proceedings*, 24–29 April, Amsterdam, pp. 169–170.

GARZOTTO, F., PAOLINI, P. & SCHAWBE, B. (1993). HDM—a model based approach to hypertext application design. *ACM Transaction on Information Systems*, **11,** 1–26.

GISOLFI, A. & LOIA, V. (1994). Designing complex systems within distributed architectures. *International Journal of Applied Artificial Intelligence*, **8,** 393–411.

HALASZ, F. (1988). Reflections on NoteCards: seven issues for the next generation of hypermedia systems. *Communications of the ACM*, **31,** 836–852.

HALASZ, F. G. & SCHWARTZ, M. (1994). The Dexter hypertext reference model. *Communications of the ACM*, **37,** 30–39.

HEWITT, C. (1991). Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, **47,** 79–106.

ISAKOWITZ, T., STOHR, E. A. & BALASUBRAMANIAN, P. (1995). RMM: a methodology for structured hypermedia design. *Communications of the ACM*, **38,** 34–44.

KAY, J. (1994). Lies, damned lies and stereotypes: pragmatic approximations of users. In *Proceedings of the IV International Conference on User Modeling, UM94*, 15–19 August, Hyannis, MA, USA, pp. 175–184.

KAPLAN, C., FENWICK, J. & CHEN, J. (1993). Adaptive hypertext navigation based on user goals and context. *User Models and User Adapted Interaction*, **3,** 193–220.

KIM, W. & AGHA, G. (1992). Compilation of highly parallel actor-based languages. *Proceedings of 5th Workshop on Languages and Compilers for Parallel Computing*, September, New Haven, CT, pp. 1–12.

KIMBROUGH, K. & LAMOTT, O. (1990). *Common Lisp User Interface Environment.* Texas Instruments Inc.

KOBSA, A. (1991). *Preface to User Modeling and User-adapted Interaction*, **1**.

KOBSA, A. (1993). User modeling: recent work, prospects and hazard. In *Adaptive User Interfaces: Principles and Practice*, pp. 111–128. Amsterdam: North-Holland.

KOBSA, A., MÜLLER, D. & NILL, A. (1996). KN-AHS: an adaptive hypertext client of the user modeling system BGP-MS. *Review of Information Science*, **1**.

LASHKARI, Y., METRAL, M. & MAES, P. (1994). Collaborative interface agents. *Proceedings of AAAI'94*.

LOIA, V. (1994). Distributed diagnostic reasoning: a new approach to student modeling. *Proceedings of the IV International Conference on User Modeling, UM'94*, 15–19 August, Hyannis MA, USA, pp. 37–41. *User Modeling and User Adapted Interaction*, to appear in an extended version.

LOIA, V. & QUAGGETTO, M. (1993*a*). High level management of computation history for the design and implementation of a Prolog system. *Software-Practice and Experiences*, **23,** 119–150.

LOIA, V. & QUAGGETTO, M. (1993*b*). CLOS: a key issue to bridge the gap between object-oriented and logic programming. *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, 16–18 June, San Francisco, pp. 62–69.

LOIA, V. & QUAGGETTO, M. (1994). Integrating object-oriented paradigms and logic program: the OPLA language. *Proceedings of the 9th International Conference on Knowedge-Based Software Engineering—KBSE'94*, 20–23 September, Monterey, CA, pp. 158–164. New York: IEEE Press.

LOIA, V. & QUAGGETTO, M. (1996). The Opla system: designing complex systems in an object-oriented logic programming framework. *The Computer Journal*, **39,** 20–35.

MAES, P. (1994). Social interface agents: acquiring competence by learning from users and other agents. In O. ETZIONI, Ed. *Software Agents-Papers from the 1994 Spring Symposium*, pp. 71–78. New York: AAAI Press.

MATHÉ, N. & CHEN, J. (1996). New user-driven and context-based centered adaptive information access. *User Modeling and User Adapted Interaction*, **6**.

NIELSEN, J. (1996). *Multimedia and Hypertext. The Internet and Beyond.* London: Academic Press.

OOCP (1993). In S. MATSUOKO & A. YONEZAWA, Eds. *Object Oriented Concurrent Programming.* Cambridge, MA: MIT Press.

RICH, E. (1989). Stereotypes and user modeling. In A. KOBSA & W. WAHLSTER, Eds. *User Models in Dialog Sytems*, Berlin: Springer Verlag.

SCOTT, P. J. & ARDRON, D. J. (1994). Integrating concept networks and hypermedia. *Proceedings of the World Conference on Educational Multimedia and Hypermedia—ED-MEDIA'94*, AACE, pp. 515–521.

Sun Microsystems. (1996). JOE: client/server applications for the Web. *White Paper*.

THOMAS, C. & FISCHER, G. (1996). Using agents to improve the usability and usefulness of the World Wide Web. *Proceedings of V International Conference on User Modeling, UM'96*, Kailua-Kona, Hawaii, pp. 5–12.

THÜRING, M., HANNEMANN, J. & HAAKE, J. M. (1995). Hypermedia and cognition: designing for comprehension. *Communications of the ACM*, **38,** 57–66.

VASSILEVA, J. (1996). A task-centered approach for user modeling in a hypermedia office documentation system. *User Modeling and User Adapted Interaction*, **6,** 185–223.

WEGNER, P. (1995*a*). Interaction as a basis for empirical computer science. *ACM Computing Surveys*, **27,** 45–48.

WEGNER, P. (1995*b*). *Models and paradigms of interaction. Technical Report* CS-95-21, Department of Computer Science, Brown University, September.

WILLRICH, R., SÉNAC, P., DIAZ, M. & DE SAQUI-SANNES, P. (1996). A formal framework for the specification, analysis and generation of standardized hypermedia documents. *IEEE Proceedings of the International Conference on Multimedia Computing and Systems*, 17–23 June, Hiroshima, Japan, pp. 399–406.

Paper accepted for publication by Editor, B. R. Gaines.