A Multi-Agent Reflective Architecture for User Assistance and its Application to E-Commerce

Antonella Di Stefano[1], Giuseppe Pappalardo[2], Corrado Santoro[1], Emiliano Tramontana[2]

[1]Dipartimento di Ingegneria Informatica e delle Telecomunicazioni
[2]Dipartimento di Matematica e Informatica
Università di Catania
Viale A. Doria, 6 - 95125 Catania, Italy
{adistefa, csanto}@diit.unict.it, {pappalardo,tramontana}@dmi.unict.it

Abstract. Assisting an application can involve several tasks of a different nature; thus it can be a complex job which is better performed by several autonomous agents. Accordingly, in many scenarios, several small assistant agents, each dedicated to a single task, are employed to supply help and to enhance the same application.
This paper proposes a software architecture that allows multiple assistants to serve the same application and interact with each other as necessary, while working autonomously from each other. This architecture interfaces assistants with an existing application by means of computational reflection. The latter mechanism allows meaningful user activities to be intercepted by assistants, and the outcomes of their activity to be supplied to the application. No assumptions need to be made about the application or the assistants; assistants can be changed, added and removed as necessary to adapt the application to unforeseen scenarios, conversely an assistant can be employed to support several applications. The usefulness and applicability of the proposed architecture is demonstrated by an e-commerce case study: we show how a suitable assistant set can integrate with and enhance a bare web browser, making it fit to support e-commerce activities.

1 Introduction

Although common widely-known applications provide a user-friendly GUI and a set of functionalities that help users carrying out their work, they can be further improved with *assistant agents* [11, 17, 4] that facilitate user operations and/or add support for user-specific activities. In this field, one of the most famous example is the Microsoft Office assistant, which is an ActiveX object, integrated with the Office suite. Other research proposals provide assistants to help web browsing [14, 6], chatting [15], web mining [12], etc.

Up to now, the techniques used to connect assistant agents with an application need to be (re-)designed each time a new application is extended or a new assistance functionality is integrated; this is because the connection is generally

obtained by exploiting the access points provided by the application itself or by the operating system environment (e.g. scripting services). In any case, solutions are application- or environment-dependent and no general technique to interface applications and assistants exists. In addition (with few exceptions like [6, 12]), assistants proposed so far consist of a single component which embeds all the supplied functions, thus making it hard to modify assistance functionalities or add further ones.

To overcome the above limitations, this paper proposes a multi-agent architecture for the *modular* design of application assistance software based on a set of cooperative agents. It extends [4] by supporting coordination among various assistants working for the same application. The architecture is conceived not to be tied to a specific application. It exploits *computational reflection* [16] to interface with an existing application written in an object-oriented programming language. We adopt the *metaobject* model [8] to capture control from an application object whenever an operation is performed on it (e.g. a method is invoked), and to bring control within the associated *metaobject*, which can choose to modify the behaviour of the application object. In the proposed architecture, a set of assistant agents, each entrusted with a specific task, cooperate with a special agent, called *Coordinator*, which handles interactions between them and the application. This agent incorporates some metaobjects, which *intercept* control from application objects, and cooperates with assistant agents. It triggers assistant activities and uses their outcomes to change or enrich the behaviour of the application.

The proposed architecture affords a great degree of *flexibility* and *modularity* in the design and implementation of assistants. Each of these can be seen as a "plug-in", which can be added (even at run-time) if its functionality is needed, or can be removed in the opposite case, without affecting the functioning of the entire system.

The architecture has been employed to coordinate a set of assistants aimed at facilitating e-commerce activities. We envisage several types of assistants enhancing a web browser with functionalities for e-commerce and simplifying the steps that users must perform before purchase. An assistant is dedicated to each of the following tasks:

- understanding user preferences from visited web pages and from typed keywords;
- extracting data from web pages to collect features of interesting goods;
- creating *on the client side* a virtual cart that stores potential user's purchases;
- finding offers for user selected goods;
- monitoring the trend of prices of user selected goods.

The outcomes of assistant activities are used to change browser behaviour, by e.g. re-organising web pages and highlighting important pieces of information on web pages, as the user navigates.

The outline of the paper is as follows. Section 2 presents the reflective software architecture for coordinating multiple web assistants. Section 3 describes

in detail a set of assistants cooperating within the proposed architecture, in order to enhance a web browser. Section 4 analyses related work. Conclusions are presented in Section 5.

2 An Architecture for Coordinating Assistants

2.1 Reflection

A software system is said to be reflective when it contains structures, representing some of its own aspects, which allow it to observe, and operate on, itself [16]. A reflective system is typically a two-level system comprising a *baselevel*, intended to implement some functionalities, and a *metalevel*, which observes and acts on the baselevel. A widespread reflective model is the *metaobject model*, which associates each baselevel object for which this is deemed useful, with a corresponding metalevel object called *metaobject*. As Figure 1 shows, metaobjects *intercept* control from their associated objects whenever e.g. an object method is invoked (see (1) of Figure 1), or an object changes its state [16, 8]. Once control is within metaobjects (2), these are able to *inspect* and change the state of their associated objects, and to modify objects behaviour by activating operations, changing parameters, etc. After control has been captured by metaobjects, it is usually given back to the object invoked initially (3).

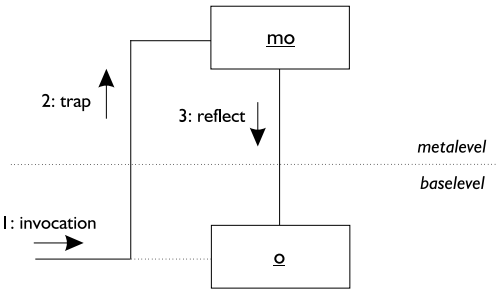

Fig. 1. Metaobject model

Reflective systems have been exploited to transparently provide software systems with synchronisation [22], adaptation to changing conditions of the environment [23], fine grained allocation of objects in a distributed environment [5], etc.

Metaobjects are associated with objects by means of reflective object-oriented languages, such as OpenC++ [2], Javassist [3], etc. The former language is a reflective version of C++ that relies on inserting keywords into the source code to provide an application with metaobjects. Then the OpenC++ code is transformed into executable code by a special pre-compiler. The latter language is a reflective version of Java that allows objects to be associated with metaobjects

by changing selected bytecode parts, and injecting into objects statements that notify some of their events to metaobjects.

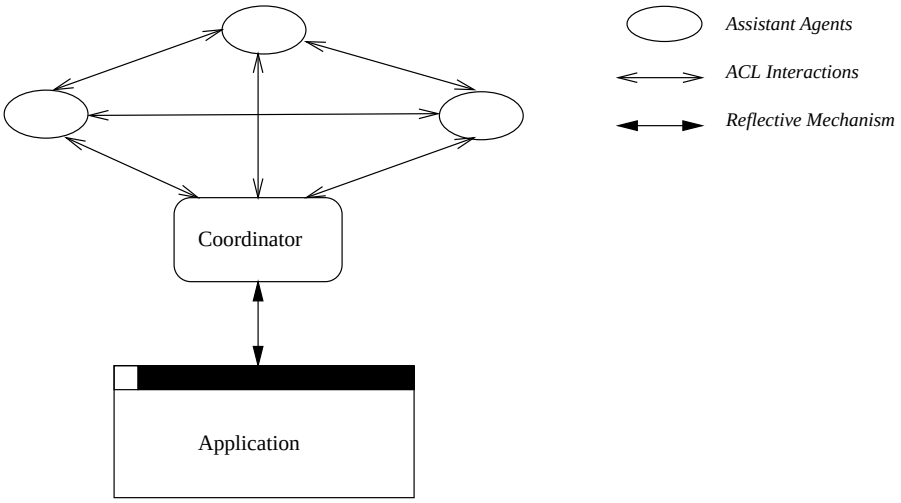

Fig. 2. Reflective software architecture coordinating several assistants

2.2 The Architecture

Assistants aim at providing additional information and functionalities to users while they work with an application. For this, assistants need to capture user activities and appropriately react by changing the application behaviour.

The software architecture that this paper proposes exploits computational reflection as a means to integrate assistants into an application. In such an architecture, complexity is handled and the support for modularity is effectively realised by using several autonomous and specialised (assistant) agents.

The architecture consists of an application (typically a web browser, however we are exploiting it also for other applications) at the baselevel, and various agents, i.e. a *Coordinator* and some assistants, at the metalevel (Figure 2). Several assistants enable achieving *modularity* since each of them is built as a small component dedicated to a single task.

Assistants interact with the application by means of the *Coordinator* agent, whose purpose is to communicate assistants the interesting events of the application, to use assistant outcomes to change some operations of the application, and to allow exchanging data between assistants. Communication between agents—both assistants and *Coordinator*— is performed by using messages, that is Agent Communication Language (ACL) speech acts [13].

Thanks to reflection, the *Coordinator* is able to detect the *effect* of user operations on the application (i.e. method invocations, changes of object attributes,

etc.). On the basis of this detection, the *Coordinator* deduces user actions and notifies the occurrence of these to other agents, according to their requests. In particular, each agent interested in being informed of a specific event sends a `request-whenever` speech act to the *Coordinator* defining, as the condition, the user action to intercept. Conversely, the Coordinator, each time that condition is met (i.e. the user action is intercepted), notifies requesting agents via an `inform` speech act, which carries additional parameters related to the action itself.

The *Coordinator* is also responsible to perform actions onto the application, derived from the outcomes of the reasoning process of assistant agents. For example, if an assistant wishes to highlight some words on the application (e.g. searched keywords), it can ask the *Coordinator* to change the colour of those words each time they occur in a loaded page. This is done by sending a `request` speech act, containing the action to be done, to the *Coordinator*.

The characteristics of the proposed reflective software architecture, and especially those of the *Coordinator*, allow various assistants to be plugged-in, according to the user needs. Reflection makes the application not aware of assistants changing its behaviour, whereas the *Coordinator* has to know only that some assistants need to be notified and that they can provide data to be used for the application, but has no knowledge at design time about their number and specific task. Indeed, assistants can be of any type and can be created incrementally. The only constraint for assistants, to be able to interact with each other and the application, is the type and meaning of events and data that they are able to exchange. To make it possible for the *Coordinator* to work with several unknown assistants, it exploits the *Observer* design pattern [9] and the *Blackboard* architectural style [20].

Figure 3 shows the proposed reflective software architecture allowing several assistants and an application to be interfaced, and the structure of the *Coordinator*.

2.3 Components of the Coordinator

In order to achieve/perform its operations, the *Coordinator* is based on the set of components shown in Figure 3, which also depicts the dynamic of interactions. These components are: metaobjects `Switcher`, implementing the interactions with application objects; and metalevel objects `Merger` and `Blackboard` realising the interactions with assistants, by means of an ACL. Each component is described in the following.

- Metaobjects `Switcher` are employed to detect events of the application. They are associated with those application objects that generate events which some assistants are interested in, and those application objects whose behaviour can be changed by assistants.
  Each metaobject captures all the method invocations and state changes of an application object (see (1) of Figure 3), thus it is able to detect a specific set of events and to intervene to modify the behaviour of the application object. E.g., for e-commerce assistants, the events that metaobjects capture

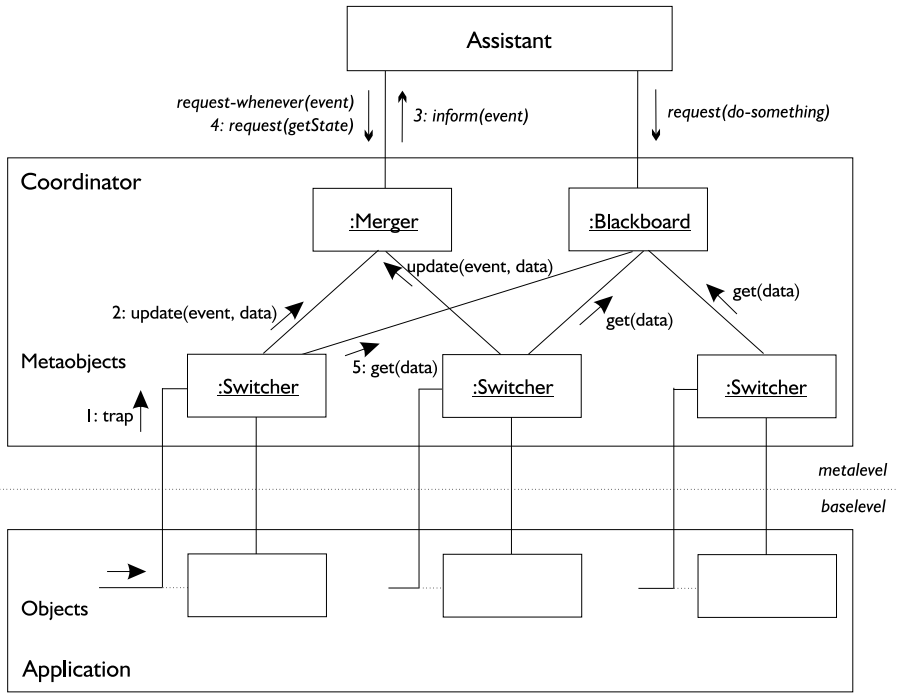

Fig. 3. Coordinator functionalities and interactions with assistant agents

include: downloading a new web page, rendering a web page, displaying a word on the screen, inserting a word in a web form.

- Metalevel object `Merger` receives information on events and data of the application by metaobjects `Switcher` (2). As in the *Observer* design pattern, it handles a list of *observers*—i.e. agents involved in the assistance activity—for each intercepted event and sends them notification (3). If needed, assistants may receive additional data, related with the event, using a suitable `request` speech act (4).
  The *Observer* design pattern allows updating the state of the assistants that need to know changes of the application state. It lets assistants change or increase their number without modifying the `Merger` (i.e. the *subject*, in the design pattern terminology) or other observers. It ensures loose coupling between the *Coordinator* and assistants.
- Metalevel object `Blackboard` is a repository for the outcomes of assistants and it constitutes a way to make results available to the application and to assistants. Outcomes derive from assistants deduction activity and are communicated asynchronously to this component in order to influence the behaviour of the application. Metaobjects access the `Blackboard` (5) and, according to retrieved outcomes, change the behaviour of their associated application object, by e.g. modifying its parameters, and synchronise assistance activities with the object operations.
  In our architecture, the `Blackboard` component is also a *shared knowledge base*, that is used by an assistant to obtain information inferred by other assistants during their activity. This architectural style allows independent assistants to work cooperatively and to share results [1]. Each assistant is not required to know about others, thus enhancing modularity.

Metaobjects are the means to access the application objects both to gather data and to modify their behaviour. They also provide coordination for the assistants activity and synchronisation between them and the application. The application changes its state for user activities and for its computation, and concurrently the assistants perform computation and provide their outcomes, from their deduction activity. Metaobjects exploit the moment when control is captured to pour the assistants (partial) outcomes to the application. This synchronisation ensures that the application is not badly affected by additional concurrent activities and it does not add complexity due to the handling of concurrency since it is very easy to be implemented.

2.4 A Case-Study: Changing the Behaviour of a Web Browser

In order to better understand the mechanisms of the architecture, in this Section we provide a simple example showing how a web browser is extended by adding user assistance functionalities. We consider a single assistant agent charged with the task of finding and highlighting keywords each time a new web page is loaded and displayed by the browser. The assistant builds a ranked list of keywords by means of a term-frequency algorithm (see Section 3.1), which is updated each

time a new page is loaded. The top elements of the list are used to determine the words to highlight.

As discussed in [4], the first step of a programmer wishing to build the connection between the assistance software and an application is *identifying* the *events* that trigger assistant activity. In this case, the relevant events are: downloading a new web page and rendering a web page, which are briefly sketched below.

1. *Downloading a new web page.* The assistant needs, from the *Coordinator*, a notification each time a new page is requested by the user and loaded by the browser. To this aim, at startup, the assistant contacts the *Coordinator* using a `request-whenever` speech act which expresses the assistant's interest in receiving this notification. Each time a new page arrives, the *Coordinator* answers with an `inform` speech act containing the downloaded HTML source page.
2. *Rendering a web page.* In order to perform keyword highlighting, the assistant executes a `request` speech act that supplies the *Coordinator* a ranked list of tuples, in the form `(keyword,colour)`, where the colour indicates how important a word is.

After identifying the set of events triggering assistance, the second design step to be performed is *understanding* which browser objects are involved with the events identified in the first step. Obviously, these objects depend on the implementation of the web browser. E.g., referring to the *Jazilla* Java browser [21], the above events are handled respectively by objects `SimpleLinkListener` and `JTextPaneRenderer`.

The third design step is *connecting* the identified application objects with the *Coordinator* by capturing their method calls by means of metaobjects. For this purpose, we use the Javassist [3] reflective extension of Java. In the example at hand, metaobject `SwitcherListen` is associated with object `SimpleLinkListener` and captures control when a new page is downloaded (i.e. when method `hyperlinkUpdate()` is invoked). This informs object `Merger` (see Figure 3) of this event, which, in turn, sends an `inform` message to the assistant. Once this message has been sent, control is returned to the application, which continues its normal execution.

As concerns keyword highlighting, application object `JTextPaneRenderer`, which displays web pages, is associated with metaobject `SwitcherRender`. The latter traps control before a page is displayed, searches the page for the the keywords of the ranked list, stored in the `Blackboard`, and modifies the page formatting to change the foreground colour of the keywords.

3 Web Assistants for E-Commerce

By exploiting the architecture proposed in the previous Section, we designed a multi-agent system aimed at assisting e-commerce activities performed through a web browser. The application extended is the web browser Jazilla. At this stage we have developed and tested the *Coordinator* and a simple version of

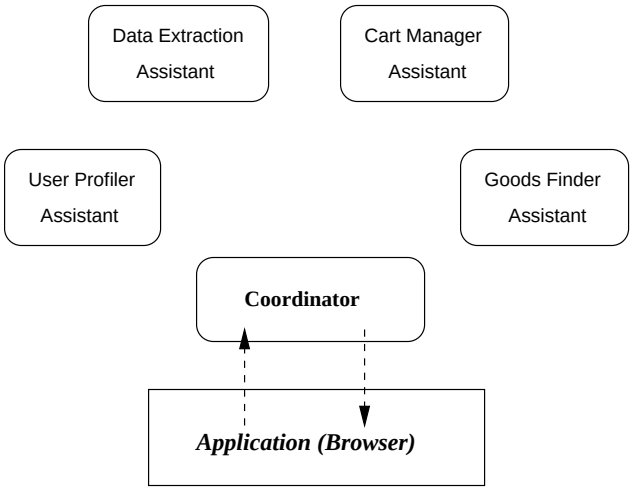

Fig. 4. E-commerce assistants for a web browser

the assistants described in the following. The set of assistant agents envisaged is depicted in Figure 4; they cooperate together to perform the following activities:

- understanding user preferences from visited web pages and typed keywords;
- extracting data about interesting goods from visited web pages;
- storing extracted data into a virtual cart;
- finding offers for some user selected goods.

3.1 User Profiler Assistant (UPA)

This assistant is entrusted with the task of profiling the user while he browses the Internet, in order to automatically determine user preferences and interests. For this purpose, it analyses and classifies the visited Web pages. This activity is triggered when a new page is loaded by the web browser and performed *autonomously* and asinchronously from the web browser and its user. UPA is informed by the *Coordinator* that a new page has been loaded.

Once the assistant is triggered, it employs a classification algorithm to characterise the current web page, by finding out its degree of "similarity" to categories of a predefined set. The adopted approach is analogous to the one described in [18]. The latter algorithm uses a set of page *categories* and a set of *weighted keywords* for each category generated in a training phase specialised for e-commerce activities. These two sets are stored into two appropriate local tables. New pages are classified by calculating and ranking the similarity value to each predetermined category.

This classification algorithm finds the frequencies of the words of a web page and normalises them by considering the length of the web page. To take into account user interests, in addition to the algorithm cited above, our assistant

changes the keyword weights of each category according to the most recurrent keywords of the visited pages. This allows tuning the keyword weights within a category to be tuned in order to adapt to the user's navigation activity. The outcome of the characterization of user preferences is a list of ranked keywords, called *WebPro requested pagesfile*, obtained from the keywords of each scored category normalised with the weight of the category itself. UPA sends the *WebProfile* list to the *Coordinator*, which stores it into the `Blackboard` in order to make it available to other assistants and to the browser application.

Additionally, a user can show his/her interest for some keywords by marking the appropriate text in the web pages. Consistently with the adopted methodology, the latter user operation is captured by the *Coordinator*, which identifies the marked words and notifies them to this assistant. As a result the latter changes the weight of the new keywords in its categories. The ranked list of keywords is used, when pages are displayed on the web browser, to determine which words to highlight, as described in Section 2.4.

Note that this kind of assistant is not e-commerce specific, except for its training phase. Its activities can be used also for other kinds of assistance, which shows that this assistant can be reused as a module in several contexts.

3.2 Data Extraction Assistant (DEA)

This assistant is responsible to search for the parts of a web page that refer to goods. It is notified by the *Coordinator* of the downloading of a new web page and *autonomously* performs its page analysis. It uses the keywords previously collected by UPA and stored on the `Blackboard` to find out whether some goods are interesting.

If any relevant goods are displayed on the web page, DEA gathers good names, prices, availability, features, links to other web pages where they are offered, etc. and stores them into a local list, called *GoodList*. By doing so this assistant builds a structured version of the data contained in an unstructured web page, so that data can be easily manipulated. This is achieved by a well-known algorithm [7]), using an ontology that describes the data of interest. The ontology allows producing a database scheme. Based on the ontology, it is possible to automatically extract data from web pages and structure them according to the generated database scheme.

Once new data are gathered, the assistant compares these with those previously stored and ranks requested pages goods so that the most accessed ones are on top of the *GoodList*. This list is sent to the *Coordinator* and stored into the `Blackboard` so that it can be read by other assistants when needed. When visiting a new web page any occurrence of a good that had been selected by DEA is highlighted. In order to carry out this service, the rendering of a web page is captured by a metaobject that searches the page for items occurring in the *GoodList* stored in the `Blackboard`.

A user can show his/her interest on specific goods by marking the appropriate text in a web page. Analogously to the previous assistant, this assistant is notified

about the marking action performed by the user and carries out its extraction algorithm to update the list of goods.

3.3 Cart Manager Assistant (CMA)

The goods collected by DEA are accessed by the Cart Manager Assistant and presented to the user on request. This assistant shows a new window that graphically compares for each good the prices on different web sites or, depending on the type of data available, the trend of prices over time. These data are transformed on-the-fly by the assistant for more effective presentation; thus, when necessary, currency conversion is performed, additional costs are considered (e.g. V.A.T., delivery fee), etc. The user can interact with the graphical representation of data to notify the assistant which goods are more relevant for her/him. This phase allows the assistant to have hints about user preferences and so to tune its activities.

This virtual cart has many benefits that accrue from the fact that data about goods are stored exclusively at the client side. To begin with, it enhances security and privacy, since such data may be sensitive and personal; thanks to this client side solution, remote web sites are cut off from their handling and only the proper user is given the opportunity to work on them. The second benefit is to simplify user operations on selected data, since the virtual cart handles data provided by several web sites. It provides a common repository that the user can easily access avoiding the fragmentation of data among several web sites and independently of the availability of the network connection. Other benefits of the virtual cart are: handling additional personal information, such as the budget for types of goods; organising goods inside categories, calculating requested pagesprice trends, since it keeps track of the offers for the same goods on different web sites.

By a user/assistant interface, the user can ask to see the list of items in the cart, or graphs comparing prices or displaying their trends.

Data organised by this assistant are converted to some suitable format (such as one a spreadsheet can read) and permanently stored. This allows other applications to further analyse data, separately from the assistant, and permits integration of the collected data into other applications.

3.4 Goods Finder Assistant (GFA)

For some user selected goods, this assistant carries out additional operations, such as seeking on the web further offers, or data. When viewing the data handled by CMA, the user can ask for more detail on a good simply by clicking on it. In response, GFA autonomously searches and accesses web pages where goods can be found, and analyses them in the background looking for the good of interest. If such a good is found, GFA asks DEA to extract the appropriate data from the web page and informs CMA of the new gathered data.

Addresses of Web pages and search engines where goods are searched are stored in a list handled by GFA. Each entry of this list contains both the web address, the categories of goods which can be found, and a weight for each

category. This weight is constantly updated taking into account whether a search of a good has been successful, thus tuning the effectiveness of a web address for a given category in accordance with the number of hits.

4 Related Work and Discussion

The literature reports many works dealing with agents that support user activities [11, 17, 14, 6, 15, 12]. Yet, the majority of them deals with a single agent that embeds all assistance activities and interfaces with the application using *ad hoc* techniques. We have already cited the Microsoft Office Assistant, which is an ActiveX object (MSAgent) catering only for user interactions, while assistance tasks proper must be addressed by the application [19]. The MSAgent is also used in [12] to provide the visual looks for a set of agents which assist the user in web mining activities; however this approach is based on JavaScript and thus assistants can only be used with a (JavaScript enabled) web browser and in a Win32 environment (since the MSAgent is a Win32-ActiveX object). Assistants proposed in [14, 15, 17] are interfaced with the application by means of AppleScript [10], and thus require an application to be controllable by means of this technology. In contrast to these approaches, our proposal exploits a general methodology—reflection—which does not require the application, nor the operating system (or GUI libraries) to provide special interface "hooks". It is platform-independent and can be applied to any application provided its source code, or only its Java bytecode, is available.

The second difference with other proposed approaches lies in the modular architecture: each agent is charged with a specific task and can be added or removed at run time without affecting the structure of the entire system. The *Coordinator* allows a complete separation between interfacing and assistance tasks, thus making the assistance activity independent of the particular application to be extended. For example, using the same e-commerce assistants presented here, we can extend different web browsers by simply adapting the *Coordinator* to the specific browser. Moreover, some assistants, originally designed to aid a particular type of application, can be later used to assist another type of application. E.g., a word processor user writing commercial letters advertising some goods for sale could be presented by GFA with a list of similar goods found in the Web.

Finally, the proposed architecture is also suitable to operate in a distributed environment; some agents, such as GFA or an assistant monitoring price trends and goods availability, could operate on some "reference server sites" in the background, irrespective of whether site users are browsing the web; they would provide search results as soon as a user opens his browser. In addition, if we include an agent capable of HTTP communication, we could be able to support the "personal mobility" [6], in order to offer search results also when the user is browsing the web from a PC different than his own.

5 Conclusions

This paper has described a reflective software architecture allowing multiple assistant agents to aid an application. The architecture handles the integration of assistants into the application, while enabling assistants to perform their tasks autonomously. Assistants need not be known at design time and can be added, when available, at run time.

Regarding performance issues, we have tackled it by making the interaction loose between application and assistants, e.g. giving assistants sufficient autonomy for carrying out their activities. This has been experimentally observed to avoid the application to be excessively delayed. Moreover, interception of application events, by metaobjects, can be carefully tuned to introduce a bearable overhead, e.g. capturing the rendering of the whole web page and introducing changes once for all is much faster than capturing the rendering of each word.

We have shown the usefulness and applicability of the architecture by means of a set of e-commerce assistants that enhance a web browser. However, the architecture can be easily used in other contexts varying the set of assistants, the application or both.

References

1. G. Cabri, L. Leonardi, and F. Zambonelli. Mobile-Agent Coordination Models for Internet Applications. *IEEE Computer*, 33(2), February 2000.
2. S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 285–299, 1995.
3. S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of the ECOOP 2000*, volume 1850 of *Lecture Notes in Computer Science*, 2000.
4. A. Di Stefano, G. Pappalardo, C. Santoro, and E. Tramontana. Extending Applications using Reflective Assistant Agents. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (Compsac'02)*, Oxford, UK, 2002.
5. A. Di Stefano, G. Pappalardo, and E. Tramontana. Introducing Distribution into Applications: a Reflective Approach for Transparency and Dynamic Fine-Grained Object Allocation. In *Proceedings of the Seventh IEEE Symposium on Computers and Communications (ISCC'02)*, Taormina, Italy, 2002.
6. A. Di Stefano and C. Santoro. NetChaser: Agent Support for Personal Mobility. *IEEE Internet Computing*, 4(2), March/April 2000.
7. D. W. Embley, D. M. Campbell, Y. S. Jiang, S. W. Liddle, Y.-K. Ng, D. Quass, and R. D. Smith. Conceptual-Model-Based Data Extraction from Multiple-Record Web Pages. *Data Knowledge Engineering*, 31(3):227–251, 1999.
8. J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326, New York, NY, 1989.
9. E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA, 1994.

10. D. Goodman. *Danny Goodman's AppleScript Handbook.* Random House, New York, 1994.
11. J. Bradshaw et al., editor. *Software Agents.* AAAI Press, Cambrigde, Mass., 1997.
12. Y. Kitamura, T. Yamada, T. Kokubo, Y. Mawarimichi, T. Yamamotom, and T. Ishida. Interactive Integration of Information Agents on the Web. In *Proceedings of CIA 2001*, volume 2182 of *Lecture Notes in Artificial Intelligence*. Springer, 2001.
13. Y. Labrou, T. Finin, and Y. Peng. Agent Communication Languages: the Current Landscape. *IEEE Intelligent Systems*, March-April 1999.
14. H. Lieberman. Letizia: An Agent That Assists Web Browsing. In *International Joint Conference on Artificial Intelligence*, Montreal, August 1995.
15. H. Lieberman, P. Maes, and N. Van Dyke. Butterfly: A Conversation-Finding Agent for Internet Relay Chat. In *International Conference on Intelligent User Interfaces*, Los Angeles, January 1999.
16. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, volume 22 (12) of *Sigplan Notices*, pages 147–155, Orlando, FA, 1987.
17. P. Maes. Agents that Reduce Work and Information Overload. In Bradshaw, J., editor, *Software Agents.* AAAI Press/The MIT Press, 1997.
18. H. Mase. Experiments on Automatic Web Page Categorization for IR system, 1998. Technical Report, Stanford University.
19. Microsoft Corporation. *Microsoft Developer Network Library*, 2000.
20. M. Shaw and D. Garlan. *Software Architecture - Perspective on an Emerging Discipline.* Prentice Hall, 1996.
21. SourceForge. Jazilla Home Page. WWW, 2002. http://jazilla.sourceforge.net.
22. E. Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science.* Springer-Verlag, June 2000.
23. E. Tramontana. Reflective Architecture for Changing Objects. In *Proceeding of the ECOOP Workshop on Reflection and Metalevel Architectures (RMA'00)*, Nice, France, June 2000.