Extending Applications using Reflective Assistant Agents

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

Assistant agents are software systems that help users during their activities by carrying out some task as a reaction to the events of their environment. This paper proposes a software architecture that allows assistant agents to extend applications by autonomously giving users suggestions and activating useful application functionalities. The connection between applications and assistant agents is realised by means of computational reflection, which allows applications to evolve essentially without changes to their source code. The proposed approach is general in that it does not depend on a specific application nor platform. Its application is demonstrated by two examples of reflective assistant agents for a web browser, supporting data presentation and e-shopping respectively.

Keywords: Computational reflection, Agent technology, Component-based software development, Object-oriented technology, Software architecture, Electronic commerce, Internet and web-based systems.

1 Introduction

Even using the most common applications (such as word processors, electronic sheets, web browsers, etc.), users face a considerable amount of data and functionalities. In order to simplify their work, it may be helpful to manipulate some data before presentation to them or to suggest or activate certain application functionalities at a convenient time. For example, data could be manipulated, in accordance with user-determined criteria, so that a selection of incoming e-mails, or the highlighting of parts of a web page, is performed. Likewise, a word processor could format the text and suggest synonyms as the text is typed. Such activities, which extend the functionalities of applications, can be provided by *assistant agents*. These are software systems that carry out an autonomous activity, can recognise their environment and perform inferences to determine their behaviour [5, 28]. Thanks to their basic characteristics—*autonomy*, *reactivity* and *proactiveness* [27]—agents are particularly suited to build software assistants: reactivity is used to trigger assistance tasks when the user performs certain operations, autonomy allows agents to establish/decide whether a user action needs assistance, and proactiveness can be used to adapt the assistant's behaviour to the evolution of user abilities or new assistance requirements. For this reasons, several works [9, 14, 15, 16, 19] make a wide use of agent technology for software assistants.

Some existing programs, like the Microsoft Office Suite, already provide an assistant agent capable of giving suggestions on what action the user could take and explaining program operation. Nevertheless, such an application must be aware of the existence of the assistant, and implement both the policy and the mechanisms whereby the assistant is triggered; what is left for the latter to do, then, is merely the task of interacting with the user [21]. This arrangement increases the complexity and size of the application, whose design is strongly and possibly adversely affected by its close relationship with the desired assistance functionalities. As for the assistant, its scope for adaptation to user needs or separate development is rather limited, since it is only entrusted with user interface services.

In contrast, we advocate a software architecture that completely isolates applications from the presence of assistant agents: applications should not even know that assistants exist, while these unobtrusively observe any activity by the application and autonomously decide when to step in[1]. This approach simplifies the separate *development* and *evolution* of both the application and the assistant agents. The latter can be easily developed as plug-ins by third parties, can provide wide range of services, which e.g. handle data and activate functionalities, and can be trained by

[1]Of course the user can predetermine the degree of alertness and extent of intervention the assistants should exhibit.

users or learn to vary their behaviour adaptively. The separation between applications and assistants diminishes the complexity of applications, by relieving them of the burden to provide the services which have been delegated to the assistants.

Moreover, the proposed separation allows a *high degree of modularity*. To begin with, an application could be endowed with multiple assistants, each providing a service different from (albeit compatible with) the others. Furthermore, assuming an appropriate degree of interface standardization, an assistant can be designed to serve a whole set of applications rather than a specific one; e.g. the same spelling assistant could aid various word processors and spreadsheets.

Admittedly, the literature reports other solutions which provide separation between assistant agents and applications, which can even (in some cases) be separately implemented [9, 13, 14, 15, 16]. However, these approaches achieve integration by exploiting either OS services (e.g. scripting services allowing the interception of some GUI events only) or specific access points provided by the application itself (e.g. scripting capabilities or network protocols as in the case of web assistants). Neither technology, however, allows for a full inspection of the application, useful to build more effective assistant agents.

Our proposal, instead, overcomes these limitations since it smoothly achieves integration between applications and assistants by means of *computational reflection*. A reflective system is typically a two-level system whose first level, called *baselevel*, implements some functionalities of interest, and whose second level, called *metalevel*, observes and possibly influences the behaviour of the baselevel [18]. A widespread reflective model is the *metaobject model*, which associates certain objects at the baselevel with suitable *metaobjects*, viz. objects belonging to the metalevel; thus, metaobjects are aimed at observing and controlling the behaviour and state of the associated baselevel objects.

In the proposed reflective software architecture, the application represents the baselevel, and the assistant agent(s) represent the metalevel. An assistant agent is activated by trapping activities carried out by the user of the application. Since the baselevel need not be aware of the presence of the metalevel, reflection gives a means to achieve separation and a noticeable degree of independence of applications and assistants both in the design and implementation phases; this greatly simplifies development, reuse and evolution of both applications and assistants [4]. Last but not least, reflective mechanisms allow applications to be extended even when only their bytecode is available.

The outline of the paper is as follows. Section 2 presents the model of an assistant agent. Section 3 describes the reflective software architecture integrating applications and assistant agents. Section 4 presents two examples of reflective assistant agents. Section 5 discusses issues related with the deployment of the reflective architecture. Section 6 analyses some related work. Finally, the authors' conclusions are presented in Section 7.

2 Model of an Assistant Agent

Figure 1 depicts the reference model of an assistant agent. The environment where the agent acts is composed of a *user* and an *application*. Note that in turn these also interact with each other using the application GUI. In the said environment, the assistant agent can be modeled by the following components (these are the basic building blocks, common to many assistant agents):

1. *Application Control Interface* (in the following ACI),
2. *Inference Engine*,
3. *Knowledge Base*,
4. *User/Agent Interface*.

First of all, the agent must be able to communicate with the application *in a transparent way*, i.e. without requiring user intervention; hence the need for the ACI component within the assistant agent. The ACI provides ways to "sense" and "act on" the application without involving the user. Indeed, the task of the ACI is twofold:

- to inspect the state of the application and intercept what happens inside it, so that it is possible to track user operations such as mouse movement, clicks or double-clicks, menu choices, keytyping, etc.;
- to offer methods allowing the agent to perform operations on the application as though it were the user her/himself.

Regarding the first item, since each user action has a precise meaning, when related to a particular application, the task of the ACI is not only to simply capture events (such as mouse-down, key-press, etc.), but also to understand the context and semantics of that event inside the application. For example, for a word processor, to click within a document means the position of the cursor should be changed unless the region clicked represents a link, in which case the link should be opened.

Besides being able to sense and intervene on the application, the agent must also be provided with an ability to reason. Thus, the other main component of an assistant agent is the *Inference Engine*. This is intended to give suggestions, organise data and activate operations of the application to be extended. Thus, the Engine is built taking into account the functionalities of the application, and yet usually it is developed separately from it. Basically the Engine works on the

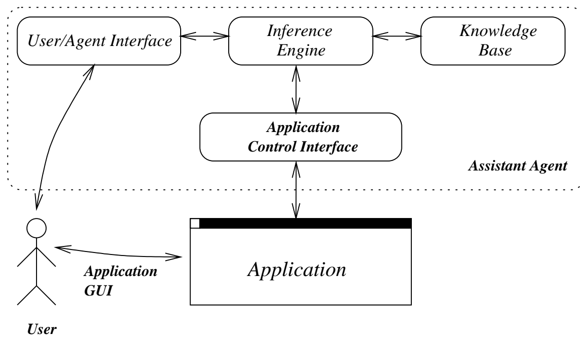

Figure 1. Reference model of an agent

basis of a set of rules which, starting from the current state of the environment and current facts (stored in the *Knowledge Base*), generates and stores new facts and/or triggers suitable actions. The internal structure of the Inference Engine depends on the behavioural model used to implement the agent. As reported in the literature, many schemes can be used, such as purely reactive, hierarchical, BDI, etc. [27].

Finally, in order to provide suggestions, an assistant must interact with the user through a proper *User/Agent Interface*, which can be graphical or of another type (e.g. an audio interface based on a speaker and a microphone).

2.1 Choosing an Implementation Methodology

The design and implementation of the components of an assistant agent should be based on a general methodology. Instead, solutions supported by specific technologies that are application- or platform-dependent should be avoided, as far as possible. For platform independence favors portability, while application independence permits the modular use of the same assistant with different applications. Accordingly, the rule-based Inference Engine and the Knowledge Base can be built using a production system such as CLIPS [1], Jess [2] or CLOS. Of necessity, instead, the User/Agent Interface is implemented using either the graphical API provided by the operating system or a speech synthesis and recognition package.[2]

For the ACI (or analogous components), instead, no general methodology has been suggested so far in the literature on assistant agents; current approaches try to exploit the access points provided by either the operating system/environment or the application to interface with. For example, some web assistants [9] act as "proxies" placed between the browser and the Web, thus intercepting user's navigation actions by recognizing HTTP requests. Other approaches to build the ACI, for more general purpose applications, are based on OS scripting services [14, 10, 22]

[2]If platform independence is paramount in this realm too, the User/Agent Interface can be implemented in Java.

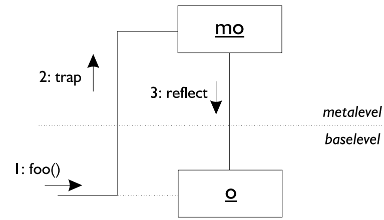

Figure 2. Metaobject model

or on wrapping OS libraries [3] (for details, see Section 5).

The situation outlined above is unsatisfactory. In short, when the ACI exploits specific access points provided by the application, the solution devised is application-dependent. On the other hand, when ACI design exploits OS support, it is of necessity platform-dependent. In any case, the lack of a generally valid solution forces designers to undertake a new analysis effort each time an assistant has to be developed for or integrated with an application.

Being based on reflection, our proposal, instead, is tied neither to specific platform services nor to specific applications (these are only required to satisfy a reasonable set of hypotheses discussed in Section 3.2). This approach is advantageous and original in the field of assistant agent design and development.

3 A Reflective Architecture for Assistant Agents

3.1 Reflection

A software system is reflective when it contains structures, representing some of its own aspects, which enable it to observe and perform operations on itself [18]. Reflection provides a principled means of accessing the implementation of a system. According to the *metaobject model*, some objects of an application, belonging to the baselevel of a reflective system, are associated with metaobjects, which are part of the metalevel. This association provides the metaobjects with the ability to *inspect* the state of the associated objects, *intercept* invocations of their methods or changes of state variables, etc.[3], and hence to gain control before baselevel operations. This allows activities to be performed or conditions to be checked transparently, either before or after executing the methods invoked at the baselevel.

As an example, Figure 2 shows a metaobject mo that traps invocations of the method `foo()` of the associated object o and then hands control to it. Thus, if the metaobject is used, say, for debugging, it can print the name of the method `foo()` invoked at the baselevel, either before or after the execution of `foo()`, without any change to the code

[3]This twofold ability is called *reification*.

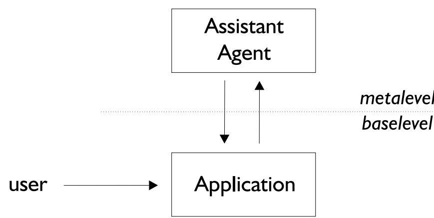

Figure 3. Reflective architecture for agents

of the baselevel object. This shows that the added value of the metalevel is delivered *transparently*. In the last decade reflective systems have been proposed for transparently providing software systems with fault-tolerance [23], synchronisation [24], distribution [20], etc.

To associate objects of the metalevel with objects of an application, the source or bytecode of the application must be available. Reflective languages such as OpenC++ [6], OpenJava [8], etc. rely on inserting keywords into source code to add reflective abilities. Then the resulting code is transformed into executable code by a special pre-compiler. Other reflective languages such as Kava [26] and Javassist [7] allow objects to be associated with metaobjects by simply changing selected parts of their bytecode and injecting suitable control transfer mechanisms just before execution; thus, in this case, source code is not required.

3.2 The Architecture

The proposed reflective architecture consists of applications and an assistant agent, implemented at the baselevel and metalevel, respectively (see Figure 3). The advantage of using a reflective architecture is that applications are not forced to be aware of any assistant agents that are collaborating with them. This achieves separation of concerns [11] and reduces the complexity of both application and assistant. As a result, it is possible to extend the functionalities of an application without modifying its code, thus greatly simplifying its development, reuse and evolution.

To construct the reflective architecture, two goals have to be attained: first, building the Inference Engine and the rules that describe the agent's behavior, and second, connecting this Engine with the application. The latter task is entrusted to the ACI component, which relies for this purpose on the reflective mechanisms described in Section 3.1. To achieve such a connection, the target application and environment should satisfy the following hypotheses.

H1 *The application consists of a collection of interacting objects implemented in an object-oriented (OO) language.*

In such a context, the most popular reflective approach is the metaobject model which, accordingly, is the one adopted here.

H2 *Depending on the language, the source or the bytecode of the application is available, as required by the adopted implementation of the metaobject model.*

This enables the necessary hooks to be inserted to capture operations (such as method invocations), thus bringing control within the metalevel.

H3 *Some knowledge of the application is available in order to determine the objects and the methods that handle the events of interest.*

This provides the programmer with the architectural knowledge necessary to decide how to apply the agent's reflective abilities to the application.

As noted in Section 2.1, the Inference Engine is first developed in a suitable rule-based language (like Prolog, CLOS, CLIPS, Jess, etc.). In the OO framework assumed, the Inference Engine is then encapsulated within a class of the OO language employed. This solves the problem of interfacing the OO language used for the application with the rule-based language. In practice, several classes can be employed to encapsulate each a different Inference Engine, in order to activate that best suited for the type of event detected and the context where the event is generated.

Given an application meeting the hypotheses **H1**, **H2** and **H3**, the steps which follow describe how the programmer may approach the design of the ACI component connecting the Inference Engine with the application (Figure 1).

Identifying a set of *events* (i.e. user actions) that should trigger the assistant agent. These events are generated by the usual operations of the user on the application. For example, an assistant agent for a web browser could provide help when the user generates the following events: opening a new page, typing a keyword in a form, opening a dialog window, etc.

Understanding how the application handles the selected events. Each event is related with some application objects or some interaction between them. In this step, the programmer should establish the correspondence between the identified events and the methods of the application that handle them. This correspondence is obtained from the knowledge of the application code or a part of it, as stated in hypothesis **H3**. This knowledge can arise from various sources, e.g. from UML diagrams or proper comments in the source code, or from the analysis of the interaction between applications and well-known GUI libraries (such as AWT or Swing for a Java application). In the latter case, the knowledge can be obtained even when the source code is unavailable, by using Java de-compilers [17] or bytecode analysers [25].

At the end of this step, the objects involved with the events identified in the previous step will have been determined.

Connecting the application objects that handle the identified events with the assistant agent. To achieve this connection, some metaobjects are associated with the appropriate application objects. Metaobjects intercept baselevel method invocations in order to capture events generated by the application and bring control within the metalevel. This allows some checks to be inserted before the execution of baselevel methods, and gives a means to start the operations of the Inference Engine. Metaobjects can also carry out several additional tasks, such as:

- recognising the context of the baselevel invocation, in order to pass the proper information to the Inference Engine;
- handling the event intercepted, by themselves.

Tasks like the above mean that metaobjects are involved with some computation and not simply trapping each call and delegating all the necessary computation to the Inference Engine. However, to avoid activating the Engine at every trap helps achieving better performance, since deduction can be a computationally costly activity.

Mapping the output of the Inference Engine onto actions on the application. The Inference Engine may suggest the activation of a functionality of the application as a result of its deduction. This functionality is mapped onto method invocations of appropriate application objects, which must be identified at this stage. Carrying out this mapping is the responsibility of the same metaobject that had trapped control from the baselevel and will, in due course, hand it back. In some cases, the outcome of the Inference Engine can be honoured by carrying out a change of some arguments of the trapped method. For example, an assistant that should highlight some keywords in a page could simply change the text colour by setting the proper arguments of the method called to draw the text.

Of course, care must be taken when invoking operations of baselevel objects from the metalevel, for this could interfere with the application in subtle ways [12].

Using the inspection capability available at the metalevel in order to construct a set of structures encoding the metalevel's runtime knowledge of the baselevel application. This should include at least the references to some relevant application objects, in order to simplify further accesses to them. Furthermore, the metalevel can exploit its awareness of the state of the application to decide upon an intervention policy that does not interfere harmfully with current operations of the application.

4 Example Applications

4.1 A Web Presentation Assistant

We have developed a Java prototype assistant agent that interfaces with a web browser following the proposed reflective architecture. This assistant captures the keywords which the user asks to be found in the current page, and highlights those keywords when pages are displayed. Also, the assistant proposes pages which it has autonomously found and selected for their relevance to the keywords.

The application events that have to be detected are: (1) displaying a page, and (2) typing (in the relevant text field) a keyword to be found in the current page. To capture these events we associate metaobjects with the relevant objects of the application. Metaobjects are also responsible to connect the activities of the Inference Engine with the web browser. Figure 4 shows, using the UML notation, the reflective software architecture employed for this case study.

After analysing the application, we find out that the objects that handle the events we are interested in are: object `:DialogBox`, which is responsible for displaying a dialog window that prompts for a keyword; and object `:Text`, which is responsible for drawing some text on the screen. Thus, two metaobjects, `:DialogMO` and `:TextMO`, are associated with the latter two objects respectively. Note that these metaobjects belong to different classes, since they perform different operations once control is trapped at the metalevel.

Application object `:DialogBox` is associated with metaobject `:DialogMO`, in such a way at run time control is trapped at the metalevel whenever a method of the former object is invoked (see (1) in Figure 4). Once metaobject `:DialogMO` gains control (2), it checks whether the invocation and the context are those in which the assistant should intervene, i.e. whether the dialog box is actually being used to type a keyword[4]. If the required conditions are satisfied, control is passed to the Inference Engine `:InferenceEngine` (3).

Once the Inference Engine hands a result (4), `:DialogMO` lets the application resume execution of the method it had originally invoked (5). Meanwhile it passes the keyword to metaobject `:TextMO`, which modifies the aspect of displayed pages (6), and invokes a method of the application object `:Page` in order to start a search of rele-

[4]This can be determined, for example, by inspecting the title attribute of the object `:DialogBox`.

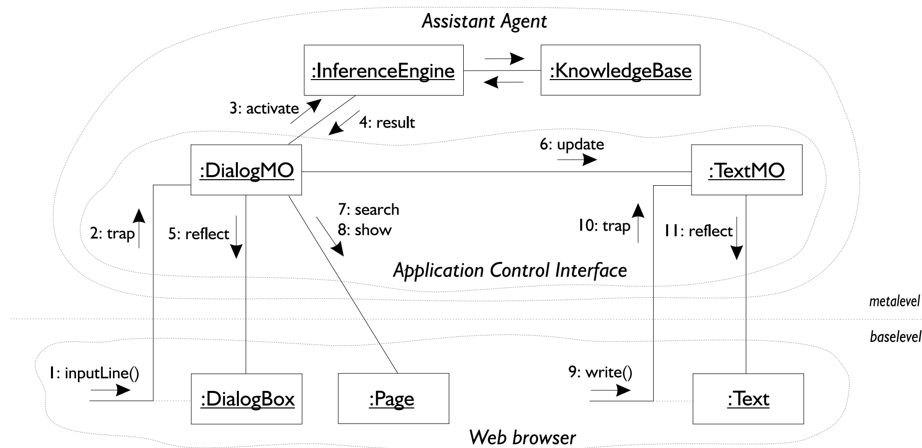

Figure 4. Reflective architecture for a web browser and an assistant agent

vant pages (7). Later the interesting pages are shown to the user (8).

When a new page is shown, methods of object `:Text` are called to display some text (9), but metaobject `:TextMO` traps control before the text is drawn on the screen (10). This metaobject intervenes to highlight the previously defined keywords, by modifying the arguments of the trapped method, or by calling other methods that change the style of the text. Then the text is actually displayed. Note that metaobject `:TextMO` had been notified the relevant keywords by metaobject `:DialogMO`.

4.2 A Web Assistant for E-Commerce

The previous example aimed at showing in some detail how the proposed methodology can be applied. Many other types of assistants can be built based on the proposed architecture.

As a further experiment, we have developed a Java prototype of a web assistant for e-commerce that simplifies the comparison of good offers found in the web. This assistant analyses data of web pages as they are visited by the user, and displays in a separate window more effective presentations (e.g. overviews, price trends) of some user-selected goods. Moreover, the extracted data (e.g. prices, product names, availability, features) are transformed on-the-fly by the assistant and stored into a database or spreadsheet for further comparative analysis.

Two techniques are employed to allow the assistant to extract data from the web pages visited. First, the user can delegate the task of analysing HTML code to the assistant, which exploits the intelligence of the Inference Engine. Alternatively, the user himself can submit significant data to the assistant, by simply marking the relevant text on the web pages. Consistently with the adopted methodology, the latter user operation is captured by an appropriate metaobject.

5 Deployment

In principle, assistant agents conforming to the proposed reflective architecture can be developed by any third party that knows the application, in accordance with and up to the hypotheses **H1**, **H2** and **H3** in Section 3.2.

In order to extend a C++ application (provided source code is available), OpenC++ [6] can be used as a reflective platform, and the Inference Engine can be implemented by means of the CLIPS system [1], which is provided as a library that can be linked into C++ code.

Java applications can be extended, as in the examples of Section 4, using a reflective language like Javassist [7] for metaobjects, and the rule-based language Jess [2] for the Inference Engine. In this case, metaobjects signal the occurrence of an event by storing an appropriate *fact* in the knowledge base (e.g. "`(ok-click keyword-search-dialog)`") by invoking the `assert()` method of the Inference Engine. Such a fact is then used by the Jess engine to activate the programmed rules. On the other hand, actions triggered by rule processing inside the Inference Engine (also leading to asserting or retracting facts) are communicated to metaobjects through Jess "listeners"[5].

When the application is developed in Java, the assistant agent is bound to the application in a configuration stage where bytecode is modified as appropriate in order to associate metaobjects with objects. During configuration the user selects the events s/he is interested in from a pre-determined list. Each event of the list corresponds to a metaobject and determines the objects that have to be made reflective.

[5]Metalevel objects whose method is invoked automatically by Jess when a specific fact is asserted or retracted.

For instance, with reference to the example of Section 4.1, the user could choose to avoid searching pages related with the keywords typed in the `:DialogBox`. In this case, instead of metaobject `:DialogMO`, `:DialogMO1` would be used, which only captures keywords (see (2) in Figure 4) and communicates with the Inference Engine ((3) and (4) in Figure 4).

As another example of configuration choices, the user could associate `:DialogMO` with `:Form` objects, thus gaining the ability to collect, highlight and search keywords typed in any web form.

6 Related Work

As observed in Section 2.1 and highlighted in [15], the literature reports no general methodology to connect assistant agents with existing applications; each time, instead, an *ad-hoc* solution has been devised. Most of the proposals make wide use of AppleScript/AppleEvents [14, 10], thus requiring the application to be *scriptable*, i.e. externally controllable by a program through a script, and *recordable*, i.e. capable of reporting user actions to an external program (for this terminology see [15, 10]).

Other approaches [13, 22] to the integration of assistant agents are based on flexible multi-layered event handling architectures. These require that applications be built using the library implementing such an architecture.

The work [3] is based on "instrumented connectors", i.e. on changing the connections between applications and the operating system by building code that wraps the OS libraries. This transparently enables code extending applications to perceive user actions and interact with the application proper. This approach is practicable, yet platform-dependent and intrusive (since parts of the operating system are changed). Moreover, it limits the possibility of extending applications since it does not allow object inspection but only intercepts interactions between application modules and system libraries. As a consequence, only some user actions can be detected.

NetChaser [9] provides user assistance for Internet services. It performs web assistance (i.e. keyword classification, prefetching of interesting pages, cache maintenance, etc.) by acting as a "proxy" between the user's browser and the web, thus trying to recognize user's interests by analyzing each page in transit. Similarly, the IRC assistant in [16] exploits the IRC protocol to interface with the application. However, these approaches are valid only for that kind of assistants and do not represent a general solution.

Finally, one of the most famous assistants, the Microsoft Agent (MS-Agent [21]) widely used in the Microsoft Office Suite, is an ActiveX object that only provides methods aimed at user interaction by means of animation and dialog boxes. Therefore, it does not contain any reasoning ability, which must be provided by the application and cannot be viewed as a proper "agent" [27]; rather, in our reference architecture of Figure 1, it corresponds to the User/Agent Interface component.

With respect to the cited works, the methodology proposed in this paper is platform- and application-independent. It does not require the application to be written (or re-written) using specific libraries or providing access points for scripting. Any object-oriented application can be extended based only on the knowledge assumed with the hypotheses **H2** and **H3** in Section 3.2.

7 Conclusions

In this paper we have presented an approach that allows extending applications with assistant agents. The proposal is general in the sense that it is not aimed to a particular type of applications and does not impose on it any specific design requirement or interfacing ability. The software architecture proposed as a solution identifies the components that constitute an assistant agent, and uses computational reflection as a mechanism to connect assistant agents with applications.

In our approach, the development of an assistant agent can be clearly separated from that of applications, thus facilitating development, reuse and evolution of both. In particular, in order to enrich applications with new functionalities (which is an example of evolution), the latter can be implemented as assistants and then integrated with the applications without changing their code. Assistants are designed so that they can perform their tasks autonomously from the application.

The performance penalty introduced by the assistant consists of the cost for its computation and the cost of jumping to the metalevel. We have lowered the former cost by caching results of the inference engine, so avoiding that each intercepted operation is handed to it; and by giving the assistant ability to work asynchronously from the application for costly operations, e.g. searching new web pages. The cost of jumping to the metalevel has been reduced by carefully choosing when interception has to be performed, capturing the rendering of the whole web page to update it (e.g. for changing colours of selected words) once for all is much faster than capturing the rendering of each word.

As a future work, we plan to exploit our reflective software architecture to build a multiple-agent system that provides assistance for an application by taking advantage of the collaborative ability of specialised assistants.

References

[1] CLIPS: A Tool for Building Expert Systems. WWW, 2002. www.ghg.net/clips/CLIPS.html.

[2] JESS: The rule engine for Java Platform. WWW, Sandia National Laboratories, (Livermore, CA, USA), 2002. herzberg.ca.sandia.gov/jess/.

[3] R. Balzer. Instrumenting, Monitoring, Debugging Software Architectures. WWW, 1998. www.isi.edu/software-sciences/papers/instrumenting-software-architectures.doc.

[4] K. Bennett, S. Glover, X. Li, and S. Rank. Designing Software for Change: Evolvable Architectures. In *Proceedings of the Workshop on Software Change and Evolution (SCE'99)*, Los Angeles, CA, May 1999.

[5] Bradshaw, J., editor. *Software Agents*. AAAI Press/The MIT Press, 1997.

[6] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA*, pages 285–299, 1995.

[7] S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of ECOOP*, volume 1850 of *LNCS*, 2000.

[8] S. Chiba and M. Tatsubori. Yet Another java.lang.Class. In *Proceedings of the ECOOP Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, 1998.

[9] A. Di Stefano and C. Santoro. NetChaser: Agent Support for Personal Mobility. *IEEE Internet Computing*, 4(2), March/April 2000.

[10] D. Goodman. *Danny Goodman's AppleScript Handbook*. Random House, New York, 1994.

[11] W. L. Hürsh and C. V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, 1995.

[12] G. Kiczales, J. des Riviéres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[13] D. Kosbie and B. Myers. A System-Wide Macro Facility based on Aggregate Events. In Cypher, A., editor, *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, Mass., 1993.

[14] H. Lieberman. Letizia: An Agent That Assists Web Browsing. In *International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

[15] H. Lieberman. Integrating User Interface Agents with Conventional Applications. *Knowledge-Based Systems Journal*, 11(1):15–24, Sept. 1998.

[16] 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.

[17] Linux Documentation Project. Java-decompilers-HOWTO. WWW, 2002. www.linuxdoc.org.

[18] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA*, volume 22 (12) of *Sigplan Notices*, Orlando, FA, 1987.

[19] P. Maes. Agents that Reduce Work and Information Overload. In Bradshaw, J., editor, *Software Agents*. AAAI Press/The MIT Press, 1997.

[20] J. McAffer. Meta-Level Programming with CodA. In W. Olthoff, editor, *Proceedings of ECOOP*, volume 952 of *LNCS*, 1995.

[21] Microsoft Corporation. *Microsoft Developer Network Library*, 2000.

[22] P. Piernot and M. Yvon. The Aide Project: An Application-Independent Demonstrational Environment. In Cypher, A., editor, *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, Mass., 1993.

[23] R. J. Stroud and Z. Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In C. Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.

[24] 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 *LNCS*. Springer-Verlag, 2000.

[25] R. J. Walker, G. C. Murphy, B. N. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing Dynamic Software System Information through High-Level Models. In *Proceedings of OOPSLA*, Vancouver, Canada, 1998.

[26] I. Welch and R. J. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *LNCS*. Springer-Verlag, 2000.

[27] M. Wooldridge. Intelligent Agents. In Weiss, G., editor, *Multiagent Systems*. The MIT Press, Cambridge, 1999.

[28] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.