

A Multi-Agent Infrastructure for Developing Personalized Web-based Systems

LILIANA ARDISSONO, ANNA GOY, GIOVANNA PETRONE and MARINO SEGNAN
Dipartimento di Informatica, Università di Torino

Although personalization and ubiquity are key properties for on-line services, they challenge the development of these systems due to the complexity of the required architectures. In particular, the current infrastructures for the development of personalized, ubiquitous services are not flexible enough to accommodate the configuration requirements of the various application domains. To address such issues, highly configurable infrastructures are needed.

In this paper, we describe Seta2000, an infrastructure for the development of recommender systems that support personalized interactions with their users and are accessible from different types of devices (e.g., desktop computers and mobile phones). The Seta2000 infrastructure offers a built-in recommendation engine, based on a multi-agent architecture. Moreover, the infrastructure supports the integration of heterogeneous software and the development of agents that can be configured to offer specialized facilities within a recommender system, but also to dynamically enable and disable such facilities, depending on the requirements of the application domain. The Seta2000 infrastructure has been exploited to develop two prototypes: SeTA is an adaptive Web store personalizing the recommendation and presentation of products in the Web. INTRIGUE is a personalized, ubiquitous information system suggesting attractions to possibly heterogeneous tourist groups.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms: Personalization

Additional Key Words and Phrases: Infrastructures for developing personalized recommender systems, multi-agent architectures

1. INTRODUCTION

Personalization and ubiquity are key properties for Business to Customer services, as well as for other types of applications, such as e-learning and information systems; e.g., see [Resnick and Varian 1997; Riecken 2000; Maybury 2000; Maybury and Brusilovsky 2002]. However, the development of an ubiquitous, personalized system is complex because the adaptation to the end-user (henceforth, user) and to her device requires the integration of very different methodologies addressing the assessment of the user's preferences and the generation of the customized user interface. Although shells for the management of personalized services can be designed, they typically lack flexibility in the accommodation of domain-specific requirements.

Authors' address: L. Ardissono, A. Goy, G. Petrone and M. Segnan, Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2003 ACM 1529-3785/2003/0700-0001 \$5.00

Therefore, infrastructures for the development of systems, whose architecture can be suitably extended to offer new facilities, are needed.

In order to address such issues, we have designed Seta2000, an infrastructure for the development of highly configurable, Web-based recommender systems.¹ This infrastructure, based on a Multi-Agent System architecture, offers an engine that the service developer can use to create recommender systems displaying two main types of behavior:

- the personalized suggestion of items, depending on the user’s interests;
- the dynamic generation of a user interface whose layout and contents are targeted both to the user’s preferences and to the device she uses.

Moreover, the infrastructure provides software libraries implementing basic facilities, among which the communication between system components and the interaction with the user’s device. These libraries can also be used to extend the system architecture, if needed.

The present paper describes the architecture and the techniques applied to run the agents of Seta2000. This infrastructure supports the integration of heterogeneous agents, in order to facilitate the exploitation of specialized modules within the recommender system. Moreover, the infrastructure offers a set of libraries for the development of goal-based agents, which are suitable for the implementation of the components exhibiting autonomous behavior. We have exploited Seta2000 to develop two systems.

— SeTA (“Servizi Telematici Adattativi”) is a Web-based store that personalizes the recommendation and the presentation of telecommunication products to the customer’s expertise and preferences [Ardissono et al. 2002].

— INTRIGUE (INteractive Tourist Information GUIDe) is a prototype tourist information server that customizes the presentation and the suggestion of attractions to the tourists’ preferences. This system can be accessed via Web browser and WAP minibrowser [Ardissono et al. 2003].

The rest of this paper is organized as follows: section 2 sketches the architecture of the engine provided by the Seta2000 infrastructure. Section 3 describes the agent development infrastructure we have defined. Section 4 sketches how the Seta2000 infrastructure can be exploited to set up a recommender system: that section shows the flexibility of our approach, by addressing the creation of an individual recommender system and its extension to meet new architectural requirements. Section 5 briefly presents the SeTA and the INTRIGUE systems. Section 6 compares our proposal to the related work. Section 7 reports some examples of integration of external software in our systems and section 8 concludes the paper.

¹This paper extends the work presented in [Ardissono et al. 2001] by providing details about the infrastructure for the development of specialized agents and its exploitation in the development of systems presenting personalized information in e-commerce and tourist information applications.

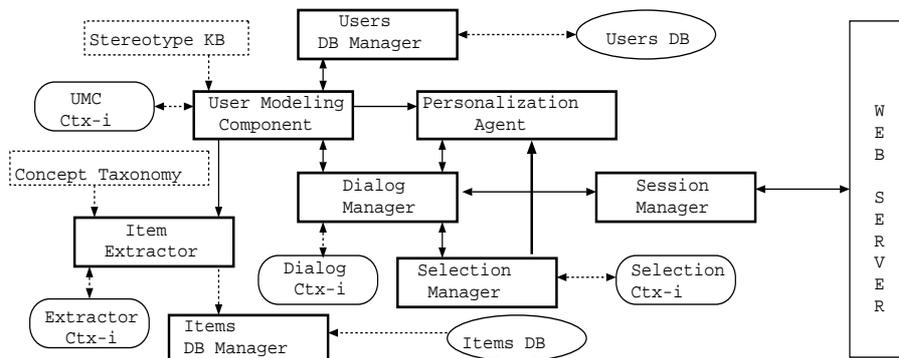


Fig. 1. Architecture of the Seta2000 recommender engine.

2. AN ENGINE FOR WEB-BASED RECOMMENDER SYSTEMS

2.1 Architectural requirements

The management of personalized recommender systems imposes requirements concerning the scalability of the underlying architectures and the integration of heterogeneous software. Several types of activities have to be carried out in parallel during the interaction with the user: e.g., the recognition of her preferences and interests, the personalized selection of the items to be recommended and the generation of the personalized content. Such activities have to be carried out by applying specialized techniques. For instance, probabilistic inference techniques are suitable for modeling the user's interests, while rule-based techniques are typically applied to customize layout, content and structure of the pages. Moreover, the same type of activity has to be carried out by applying different techniques, depending on the requirements of the application domain. For instance, collaborative filtering [O'Connor et al. 2001] efficiently supports the personalized recommendation of items in open environments. However, content-based filtering [Billsus and Pazzani 1999] is more suited to the cases where metalevel information about the items to be recommended is available, and the two techniques may be combined in other application domains [Cotter and Smyth 2000].

Agent-based technologies help managing this complexity because they support the seamless integration of heterogeneous components, and the cooperation between specialized and proactive modules [Genesereth and Ketchpel 1994; Sycara et al. 1996; Jennings et al. 1998]. Moreover, such technologies favor the parallel execution of the system activities by supporting a rich inter-agent communication based on the exchange of synchronous, asynchronous and multicast messages [Petrie 1996].

2.2 The Seta2000 recommender engine

The Seta2000 recommender engine personalizes the suggestion of items and their presentation to the characteristics of the user and of the output device. We have designed this engine by following the traditional role-based approach adopted in the Multi-Agent Systems research; see [Sycara et al. 1996] and [Bresciani et al. 2001]. As noticed by Sycara et al. [1996] and Wooldridge et al. [1999], the role-based design

supports the development of systems that harmonically integrate a set of specialized and possibly heterogeneous agents. In fact, a role isolates the responsibilities of a module from the rest of the system and favors a clean inter-module communication, based on well defined interaction protocols.

In the definition of the engine architecture, we have identified a minimal set of roles needed to manage personalized interactions with multiple users in the Web: communication with the user, management of the logical interaction flow, maintenance of the user models, generation of the user interface, selection and ranking of the items to suggest, access to user and item databases and management of the user's selections. We have associated a specialized agent to each role; Figure 1 shows the architecture of our recommender engine. The thick boxes represent agents and ovals represent the databases. The dotted rounded squares represent the knowledge bases used by the agents. The arrows between agents show the flow of the messages exchanged during a working session. The dotted arrows show the flow of the data retrieved and stored by the agents during their activity. Each agent handles multiple user sessions and the session environments are represented by means of solid rounded squares: e.g., "UMC Ctx-i" maintains state variables, active user models, and other similar information concerning the i-th user session of the User Modeling Component (UMC). Although a thorough description of the roles can be found in [Ardissono et al. 1999], we shortly present those that will be used in the rest of this presentation.

- *Web Communication.* This role, filled by a Servlet (the Session Manager) is devoted to catching user events and returning the responses to the client.
- *Management of the interaction flow.* The Dialog Manager handles the logical interaction with the user, by interpreting the generic events caught by the Session Manager and triggering the generation of the response pages.
- *Management of the user models.* A model of the user interacting with the system has to be managed to adapt the interaction accordingly; the UMC fills this role.

The roles identified within the architecture of our recommender engine differ not only in the internal behavior expected from their fillers, but also in the external interaction with the other components. For instance, some agents, such as the UMC, are proactive and autonomously carry out internal activities, by interleaving them with the provision of services; others, e.g., the Personalization Agent, only respond to service requests.

3. AGENT ARCHITECTURE PROVIDED BY THE SETA2000 INFRASTRUCTURE

In order to support the development and integration of heterogeneous agents, we have defined a general model supporting the inter-agent communication and the execution of parallel activities, autonomously, or in response to service requests. We have exploited this model to develop the agents of our recommender engine (henceforth, "Role Fillers"), which we have designed as structured entities composed of two main parts:

- An interface, called "Dispatcher", devoted to the management of the inter-agent communication. This interface is used as a reference by the other role fillers when they need to send messages. The Dispatcher also provides the role filler with

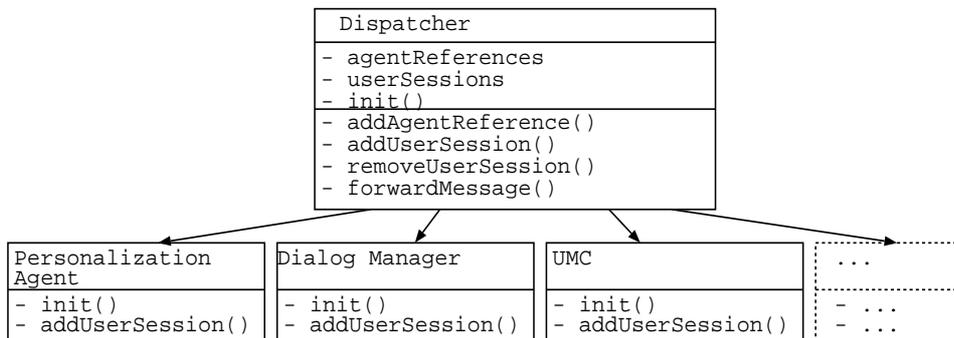


Fig. 2. Class hierarchy defining a role filler.

the communication capabilities needed to deliver messages belonging to the language supported by Seta2000 infrastructure. This language, described in [Ardissono et al. 1999], includes a subset of the speech acts specified in KQML [Finin et al. 1993]. For instance, the “tell” speech act is exploited to let the recipient know about a fact (e.g., a user datum acquired from the user interface). Moreover, the “ask-one” speech act is exploited to retrieve the value of a fact and to retrieve the pages of the user interface.

— The core of the role filler is composed of one or more instances, called “Role Filler Core”, each one devoted to the management of a user session. The role filler core provides the capabilities concerning the service provision and the execution of internal activities to be carried out. These activities are performed concurrently, thanks to the execution of parallel threads.

A role filler that has to display reactive behavior in the provision of services may have its own core designed as a traditional object. In contrast, a proactive role filler may need a core designed as an autonomous agent, which can take the initiative to perform specific activities. The Dispatcher hides this heterogeneity by wrapping the core and by translating the incoming requests to its internal format. In the following, we specify the design of the Dispatchers provided by the Seta2000 infrastructure. Later on, in section 3.2, we present the design of a specific type of core provided by our infrastructure: the “Action-based agent”.

3.1 Architecture of a role filler

The structure of a role filler, the information needed to manage the parallel user sessions and its communication capabilities are defined by a library class. Figure 2 shows the class hierarchy defining the role fillers: the “Dispatcher” class specifies that a role filler has a list of references to the role fillers it may send messages to (“agentReferences”). Moreover, it has a set of core instances (“userSessions”), devoted to the management of the activities related to the active user sessions. The class also offers the methods for initializing a role filler (“init()”) and setting references of other role fillers (“addAgentReference()”). The class also offers methods for creating and removing a core instance (“add/removeUserSession()”), in order to handle the opening and closure of user sessions. Finally, the class offers the fa-

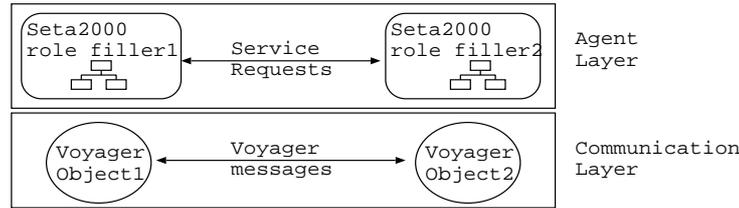


Fig. 3. Layered architecture of a Role Filler.

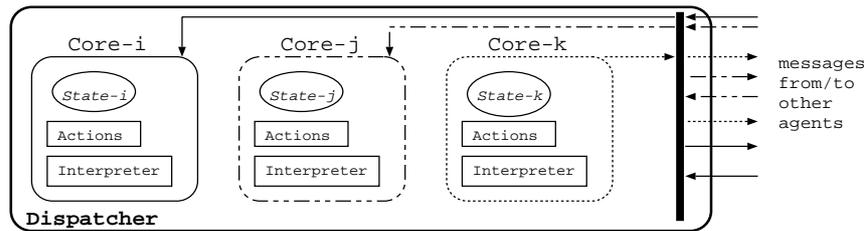


Fig. 4. Parallel sessions within a role filler.

cilities for forwarding the messages concerning a certain user session to the related core instance, in order to process a session-dependent request corresponding to a service possibly requested by another role filler (“forwardMessage()”). As shown in Figure 3, a Dispatcher is designed according to two layers.

— The upper layer describes the offered services and takes care of their execution. The facilities offered by the ObjectSpace Voyager infrastructure [ObjectSpace 2000] are applied to wrap the role fillers and enable them to run in parallel and to communicate by means of synchronous and asynchronous messages. Each Dispatcher is a Voyager Object and inherits the communication and distribution facilities offered by the Voyager Object class. The Dispatcher handles the incoming messages in parallel threads of execution, invoking the appropriate core instance to handle the requests. Moreover, Voyager takes care of load balancing issues by spawning the Dispatchers, when there is an overload in their activities.

— The communication between role fillers is implemented by relying on the message passing facilities offered by the lower layer of the architecture. The speech acts defined at the agent layer are implemented as messages between Voyager objects.

The architecture of the role fillers supports a simple management of parallel user sessions. The asynchronous communication enables the role filler to handle service requests and internal activities in parallel, with the only requirement that shared resources are accessed in mutual exclusion. Figure 4 shows a role filler, with three core instances, “Core-i,j,k”. The Dispatcher forwards each incoming message to the appropriate core instance, which handles the request. Then, the Dispatcher sends the results back to the role filler that delivered the initial message.

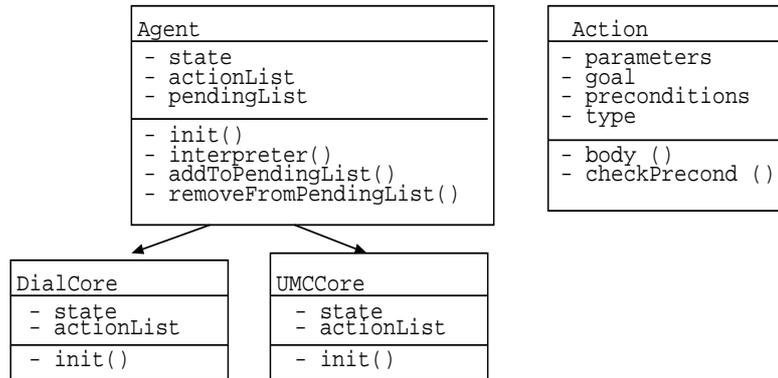


Fig. 5. Class hierarchy defining action-based role filler cores.

3.2 Design of an Action-based Role Filler Core: the “Agent” class

The Seta2000 infrastructure supports the development of a special type of core, which follows the Belief Desire Intention (BDI) model of agent behavior [Rao and Georgeff 1991]. The instances of this type of core (henceforth, *agents*) display reactive and proactive behavior. In the following, we describe the software libraries for the development of an action-based agent. We have developed the core of the User Modeling Component and Dialogue Manager of the Seta2000 recommender engine by extending such libraries.

The “Agent” class provides the basic data structures and methods for specifying the behavior of an action-based role filler core, including the selection of activities and their execution. An action-based formalism is used to describe the activities in a modular and declarative way and an interpreter is employed for their selection and execution. In order to create a concrete role filler core, the system developer has to extend the “Agent” class with specific information about the state of the agent, the initialization tasks and the actions it can perform, in response to service requests, or to carry out the agent’s internal activities. Figure 5 shows a portion of the class hierarchy defined for the Seta2000 recommender engine. In the following, we describe our framework by referring to the User Modeling Component (UMC).

3.2.1 The agent’s “State”. The *state* of an action-based role filler core describes the environment of its own role filler, within a specific user session, and evolves while the role filler sends or receives messages, or it performs actions.

Within a concrete agent representing the core of a role filler, the state extends the “State” class and specifies the actual instance variables of the agent. For instance, in the UMC, the state of the core instances contains the reference to the model of the current user, the information about whether the user’s personal data has been set, and so forth.

3.2.2 The agent’s “Actions”. The *actions* describe the tasks that the agent is able to perform. These actions represent generic behavior: specific action instances are created by the agent’s interpreter, by binding their parameters to concrete data. An action-based role filler core has an action list (“actionList” in Figure 5), storing

```

getUserDatum:
  parameters: datum;
  goal: [knowref(datum), temporary goal];
  preconditions: exists(directUser);
  body: directUser.get(datum);
  type: service provision;

```

Fig. 6. An example action defined in the UMC action-based core.

the whole list of actions defined for the agent.

We have designed an “Action” class defining the structure of an action and offering basic methods inherited by all the actions of a concrete agent. An action has the following slots:

- The *parameters* denote the arguments of the action.
- The *goal* is the information used to select the actions to be performed when a service is requested. There are *temporary* and *permanent goals*: the first ones can be satisfied by performing a suitable action, while the others persist in time and have to be satisfied by the agent during the whole interaction. Typically, service requests are associated to temporary goals. In contrast, internal activities can be triggered by temporary goals (e.g., initialization activities), as well as by permanent goals (e.g., the user model revision).
- The *preconditions* are applicability conditions that the state of the agent must satisfy for the action to be executable. For instance, the data used by an action must be available before performing it.
- The *body* represents the sequence of steps to be performed for completing the action. As a result of the execution of an action, the agent state changes; therefore, no postconditions are explicitly represented.
- The *type* distinguishes actions associated to services (invoked by role fillers) from internal activities, which the agent proactively performs during the user session. For instance, the initialization and the revision of the user model are internal activities of the UMC core.

The “Action” class offers a “checkPreconditions” method for checking the action preconditions and the “waitOnPreconditions” method to suspend the action execution until the preconditions are satisfied, if they are false when the action is examined for execution. Moreover, the class offers a “run” method, which invokes the sequence of steps (methods) in the action body.

The actions executable by a specific agent have to be defined by the developer by specifying, for each one, the parameters, preconditions, goal, body and type. For instance, the internal activities of the UMC include creating the model of the current user, initializing it with stereotypical information and updating it by applying dynamic user modeling techniques.

Figure 6 shows the “getUserDatum” action, which the UMC core instances execute to provide information about the user data. This action is a service provision and has only one parameter, “datum”, representing the requested piece of information (e.g., the user’s age). The “knowref(datum)” goal describes the fact that the role filler invoking the service wants to know the value of the requested datum.

```

Object processMessage(Message message) {
  sendAsynchMessage(thisAgent, "pendingTasks()"); /* activate pending tasks */
  Action action = findAction(message); /* identify requested action */
  if (null(action)) /* undefined requested action */
    return null;
  else {if (!action.checkPreconditions()) { /* action preconditions false */
    pendingList.add(action); /* put action into pending list */
    action.waitOnPreconditions(); /* suspend */
  }}
  return action.run(); } /* perform action body, return result */

```

Fig. 7. The interpreter of the “Agent” class.

The action precondition specifies that it can only be executed if the user model (“directUser”) has already been created. The action body consists of checking the current value of the datum in the user model (“directUser.get(datum)”)².

The preconditions of actions enable the developer to specify synchronization constraints on the execution of the agent activities; e.g., partial order relations between the execution of actions, applicability conditions, and so forth.

3.2.3 The agent’s “Pending list”. The *pending list* stores the action instances, which the agent has to perform in response to service requests, or to carry out its own internal activities, at each step of the interaction. The agent adds and removes action instances from the pending list by executing the “addToPendingList()” and “removeFromPendingList()” methods offered by the “Agent” class.

As described in section 3.2.4, the agent manages the pending list in order to satisfy the agent’s temporary and permanent goals. At the beginning of a user session, this list is initialized with action instances that satisfy the agent’s permanent and initialization goals. Then, this list is updated depending on the service requests that the agent receives. All the action instances are removed from the list as soon as they are selected for execution. However, those associated to permanent goals are reintroduced after their execution, in order to be considered again.

3.2.4 The agent’s “Interpreter”. The *interpreter* selects the actions to be performed, on the basis of the agent state and of the requests to be satisfied, and creates the action instances needed to perform the requested tasks. In the following we only deal with action instances; thus, for simplicity, we refer to them as “actions”. The interpreter is defined as the “processMessage” method of the “Agent” class (Figure 7 reports a Java-like representation of its code);³ “processMessage” is invoked by the Dispatcher each time it receives a message. On the basis of the goal specified in the message, the interpreter selects the action to be executed and checks its preconditions.

— If the preconditions are true, then the interpreter performs the action body and possibly returns the result to the caller, i.e., its own Dispatcher that, in turn,

²The figure shows an abstract representation of the action, which is a Java class and extends the “Action” class.

³The second line of the code, “sendAsynchMessage(thisAgent, pendingTasks())” refers to the management of the internal activities of the agent and is described in section 3.2.5.

returns the result to the sender of the original message.

— Otherwise the interpreter suspends, waiting to be activated (by another interpreter thread), when the preconditions become true. Before suspending, the interpreter stores the action into the pending list. When the interpreter resumes, it performs the suspended action and returns the result to the caller.

After the execution of an action and the production of its return value, the interpreter thread ends; another one will be generated by the Dispatcher to handle the next incoming message. The Dispatcher handles messages in parallel threads and remains idle when no requests (or pending activities) have to be satisfied.

3.2.5 Management of pending tasks. As both the internal activities and the service provision ones are represented as actions, they can be uniformly selected and performed by the interpreter. The crucial issue is how to trigger the internal activities, which have to be autonomously carried out, whenever their preconditions are true. The internal activities may be assimilated to suspended service requests: although, initially, the services are explicitly requested, in both cases the agent has to perform the related actions when their preconditions become true. For uniformity, the execution of internal activities and of that of suspended services are triggered by means of request messages, sent by each core to its own Dispatcher during the execution of the other services. Each time the agent processes a service request, it also launches its own pending tasks, which will be performed by another interpreter thread, if they are executable. Thus, an agent can:

- (a) receive service requests as messages from other role fillers;
- (b) send messages to its own Dispatcher for triggering the internal activities and suspended services.

The “pending tasks” message is sent by the “processMessage” interpreter as an asynchronous message, before handling the action specified in the message to be processed. See the second line of the interpreter code in Figure 7: “sendAsynchMessage(thisAgent, “pendingTasks()”)”.

When the Dispatcher receives the “pendingTasks” message, it generates a thread that selects and performs the “pendingTasks” action of the “Agent” class. The execution of this action requires that the interpreter thread selects an executable action from the pending list and processes it. If the action is an internal activity, the interpreter performs it.⁴ If the action is a service provision ones, the interpreter notifies the suspended thread.

The execution of the “pendingTasks” action also requires that the thread generates another “pendingTasks” message, to guarantee that the pending list will be inspected again. In this way, the agent can spawn a thread to handle the pending tasks every time it receives a message and every time an action is selected from the pending list. Moreover, no messages are sent when the list is empty, or it only contain actions whose preconditions are false, therefore limiting the attempts to perform such activities before the agent state changes.

Figure 8 shows a complete example, pictorially representing the management of suspended services and internal activities. The figure shows the situation of a role

⁴If the action has a permanent goal, the interpreter puts it into the “pending list” again.

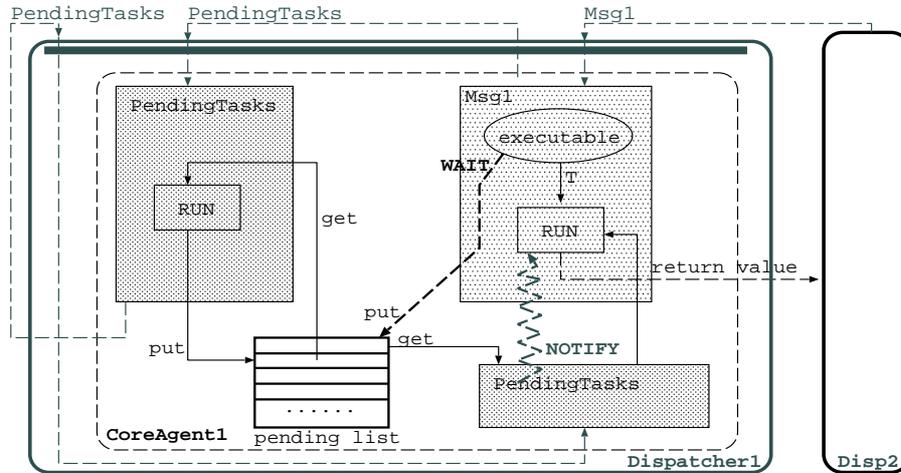


Fig. 8. Interpreter threads within an agent.

filler, “Dispatcher1”, with one core instance, “CoreAgent1”, and three interpreter threads, shown in the grey squares. “Msg1” has been generated to process a service request coming from another role filler, “Disp2”; the other two (“pendingTasks”) have been generated to handle the pending tasks.

To process “Msg1”, the interpreter selects a suitable action to be performed (either a service provision action, or the “pendingTasks” action). Then, it checks its preconditions. If they are true (“executable”), the action is performed (“RUN”) and the thread ends; otherwise, the thread stores the action into the pending list and suspends (“WAIT”). The thread sleeps until the state of the agent evolves to a situation where the preconditions of the action are true and some other thread wakes it up by means of a “NOTIFY” signal. As shown, the “notify” signals are generated by the interpreter threads invoked to perform the pending tasks.

4. USING THE SETA2000 INFRASTRUCTURE

In order to instantiate the Seta2000 recommender engine on a specific domain, the developer has to introduce a certain amount of domain-dependent information. This information is stored in declarative knowledge bases; for example, the Item Extractor retrieves the description of the categories of items to be recommended from a knowledge base called “Concept Taxonomy”; see Figure 1. This separation has the advantage that the engine can be instantiated on several domains by exploiting graphical acquisition tools that assist the specification of the domain-dependent information and hide the technical details related to the knowledge representation language adopted within the engine. Such tools also perform some simple consistency checks on the knowledge bases.

The Seta2000 recommender engine can also be configured to offer reduced personalization functionalities, or to support new facilities. Section 4.1 shows how the (optional) functionalities offered by the engine can be selected during the instantiation of the recommender engine. Section 4.2 addresses the revision of the default

functionalities to modify the behavior of a role filler. Section 4.3 sketches the extension of the engine to add a role filler performing activities that are missing in the current architecture.

4.1 Selection of the functionalities offered by the engine

The Seta2000 recommender engine offers a set of optional functionalities, which can be switched off at the engine instantiation time. This flexibility is useful because not all the systems need to exploit the full personalization capabilities offered by our engine. In the simplest cases, the engine should support the development of lighter systems, which reduce the overhead during the interaction with the user.

In particular, the developer may choose the user modeling techniques best suiting the requirements of the sales domain. For instance, stereotypical information is useful for immediate personalization, but it only works if the user population can be clearly segmented in classes with similar preferences and requirements. In contrast, dynamic user modeling enables the system to maintain precise user models, but it only makes sense if the typical interaction is long enough to acquire meaningful information about the user.

To provide this flexibility, we have developed the core of the User Modeling Component by using the Seta2000 libraries for the development of action-based agents. We represented the user model initialization and revision activities as actions that the core agent can perform. Moreover, we exploited the action preconditions to subordinate these actions to contextual conditions related to the configuration settings of the role filler. In this way, actions can be enabled or disabled, depending on the modality selected for the revision of the user model in each specific recommender system. This approach supports the definition of classes of role fillers offering the same services in different ways. For instance, the management of the user models can be filled by a hierarchy of alternative User Modeling Components.

- (a) The first and simplest one supports the construction of a generic user model, the same for every user.
- (b) The second role filler exploits stereotypical information about customer classes to predict the user's preferences. This is the configuration we selected for the INTRIGUE tourist information server.
- (c) The third one tracks the user behavior to dynamically update the user model during the interaction.
- (d) More complex role fillers can be obtained by merging these functionalities; for example, options (b) and (c) can be combined, as in our SeTA Web store.

4.2 Revision of the functionalities provided by a role filler

In some cases, the functionalities offered by the recommender engine cannot fully satisfy the domain requirements, and the problem may be solved by revising the individual role fillers. For instance, the Dialog Manager provides a default management of the logical flow of the interaction with the user, which can be directly applied within a new system. However, if the developer wishes, she can modify the interaction flow by revising the role filler core. In the following, we present the default implementation of the Dialog Manager core and then we sketch possible changes aimed at supporting different types of interaction with the user.

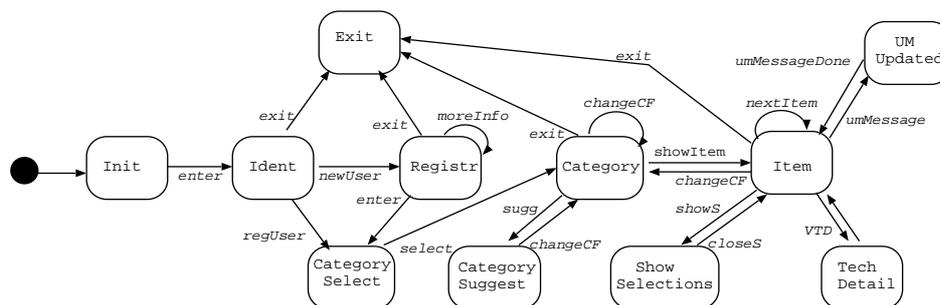


Fig. 9. Portion of the state diagram specifying the flow of the interaction with the user.

The management of the logical interaction can be represented as a sequence of turns in a dialogue between the user and the system. Each action performed by the user and page produced by the system represent turns and the admissible turn sequences can be described by means of finite state automata [Stein and Maier 1994]. As the Dialog Manager is responsible for managing the dialogue flow, it needs knowledge about the admissible turn sequences, which can be declaratively represented by means of State Diagrams [Fowler and Scott 2000]. Figure 9 shows a portion of the diagram designed for our recommender engine. The interaction is divided in three main phases:

- (1) The user identification: in this phase, information about the user is requested by means of a registration form.
- (2) The browsing of the catalog: the user enters the catalog specifying which categories of items she would like to consider. Then, she may follow links to get more information about items, and so forth.
- (3) The closure of the interaction, triggered when the user clicks on the “EXIT” button.

While the first and third phases are controlled by the system, both the system and the user can take the control during the middle phase. Normally, the system directly satisfies the user’s request by performing the requested action and showing the results; however, it may sometimes take the initiative and ask the user a question, or provide her with suggestions.

The state transitions are triggered by messages received by the Dialog Manager (shown in italics in Figure 9); such messages can origin from the actions performed by the user. For instance, from the “Ident” state, corresponding to the user identification, the Dialog Manager may:

- Move to state “Registr”, where a registration form is generated to let the user specify information about herself. This transition occurs if the user interacts with the system for the first-time (“*newUser*”).
- Move to state “Category Selection”, corresponding to the initial catalog page, where the user can specify which types of items she wants to inspect. This happens if the user has registered in a previous session (“*regUser*”).
- Go to the “Exit” state, if the user clicks on the “EXIT” button (“*exit*”).

Some transitions may be triggered by messages coming from the other role fillers and may not be related to the user's behavior.⁵

The core of the Dialog Manager has been created by exploiting the Seta2000 libraries for the development of an action-based agent, in order to make the revisions of the interaction flow as easy as possible. The state of the Dialog Manager core includes the specification of the current state of the diagram; moreover, the state transitions are implemented as actions. The preconditions of each action include the specification of the state of the diagram from which the action can be performed and the event triggering the transition associated to the action. For instance, when the agent is in state "Ident" and the "exit" event occurs, i.e., the user has clicked on the "EXIT" button, the agent can move to state "Exit", determining the closure of the interaction.

This diagram can be easily extended. For example, a new transition "T" from state "S1" to state "S2" can be added by defining a new action, whose preconditions require that the current state is "S1" and specify the event causing the transition. The action body will contain the activities to be performed in the new state (e.g., which type of page has to be displayed next) and the transition to state "S2". Notice that we have exploited this flexibility to develop different dialog strategies in the two recommender systems we have developed. In particular, the state diagram shown in Figure 9 is a subset of the diagram defining the interaction flow for the SeTA Web store. Instead, the Dialog Manager core running within the INTRIGUE system is based on a different state diagram.

4.3 Create a new role filler

If the developer needs to add functionalities to the recommender system, she may want to extend the engine by defining new roles. The following steps have to be performed to create a new role filler:

- (1) Creation of the Dispatcher.

The developer creates a wrapping class for the component. This is done by extending the "Dispatcher" class (see Figure 2) and implementing methods to add the references to the "user-session" objects.
- (2) Creation of a role filler core. This step can be performed in two ways:
 - (a) If the new role filler offers services associated to possibly complex, but deterministic activities and is not proactive (legacy software, such as a component accessing a database, might fall into this category), it can be handled by role filler cores developed as objects and responding to traditional method invocations. The "Role Filler Core" object corresponds to the component itself. The wrapping class contains references to it and will manage user-sessions.
 - (b) Other types of role fillers might include the autonomous management of internal activities. To satisfy this requirement, a role filler based on an

⁵For instance, when the Dialog Manager is in state "Item" (which corresponds to displaying the presentation page of a set of items) it may move to the "Category Suggest" state if it receives a "sugg" message from the UMC. During a transition, messages may also be sent to other role fillers. For example, from state "Item", the Dialog Manager may go to state "UM Updated" by sending a message ("umMessage") to the UMC, to specify the actions performed by the user.

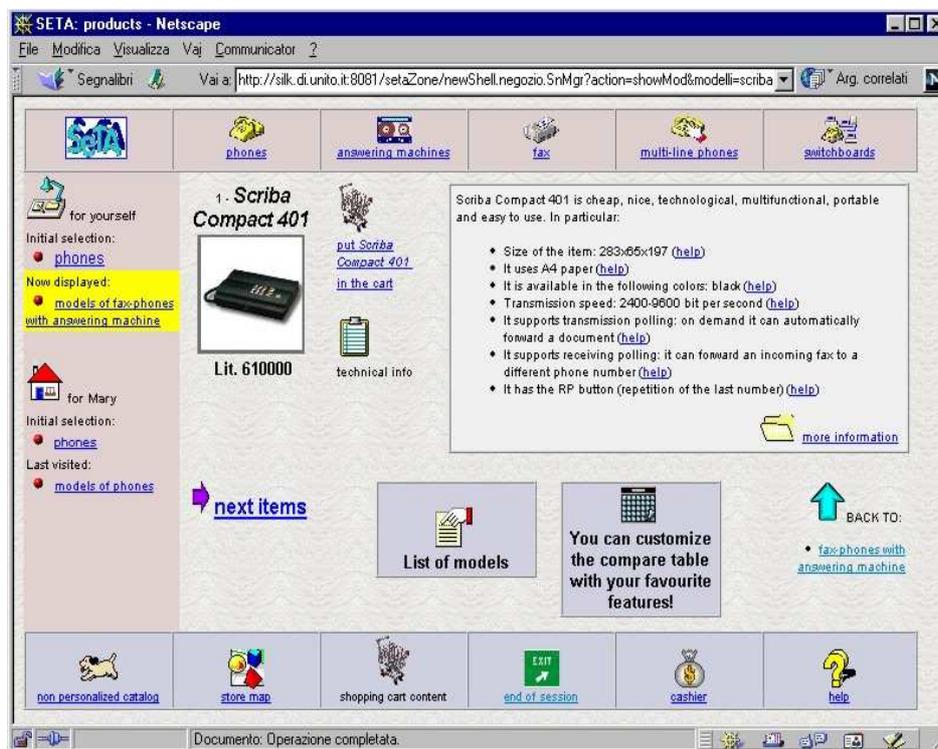


Fig. 10. Page generated by SeTA for the presentation of a telecommunication product.

explicit representation of the agent state and of the activities to be performed, declaratively represented as actions with preconditions and bodies, is needed. The developer creates a role filler core by extending the Agent class and by defining its preconditions and actions.

5. EXPLOITATION OF SETA2000

5.1 First exploitation: SeTA

SeTA (Servizi Telematici Adattativi) is an adaptive Web-based store that manages a personalized catalog of telecommunication products, such as phones and faxes [Ardissone et al. 2002].⁶ During the interaction with the customer, the Web store monitors her behavior and dynamically revises the user model to achieve a precise view of her expertise and preferences concerning the telecommunication products. Moreover, the system applies personalization techniques to dynamically generate the catalog pages, whose appearance and content depend on the user model. For instance, the amount of information displayed in each page is selected on the basis of the customer's receptivity. The description of the items is personalized, by selecting the features to be described on the basis of her interests. Furthermore, the (more

⁶Although this prototype is not available on the Web any more, a frozen demo of the system is available at the URL: <http://www.di.unito.it/~seta>.

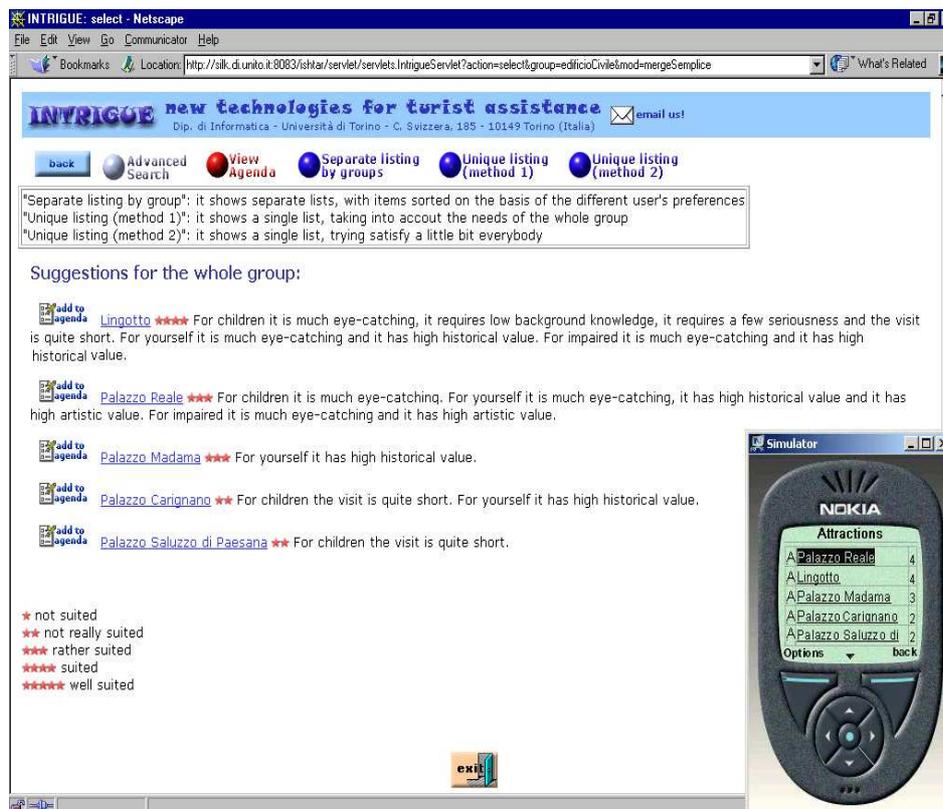


Fig. 11. Recommendation of tourist attractions generated by INTRIGUE for desktop and handset devices

or less technical) linguistic style of the descriptions is tailored to her expertise. As the user model evolves during the interaction, the system supports a reactive personalization of the interaction. Figure 10 shows a typical product presentation page generated the system.

The architecture of SeTA has been designed and developed by adapting the Seta2000 recommender engine to the requirements of Business-to-Customer e-commerce. For instance, some role fillers (e.g., the Selection Manager) have been specialized to handle an interactive shopping cart, which enables the customer user to add, remove or inspect the goods she has selected for purchase.

5.2 Second exploitation: INTRIGUE

INTRIGUE (INteractive TouRist Information GUIdE) provides information about tourist attractions and services, such as accommodation and food, in a restricted geographical area. The system helps the user to schedule a tour and can be accessed by using a standard Web browser, or a WAP minibrowser. An on-line demo of a prototype supporting the Web-based interaction is available at the URL: <http://silk.di.unito.it:8083/ishtar/intrigue.html>.

INTRIGUE manages a multilingual tourist catalog, dynamically generated by

employing efficient template-based NL generation techniques. Our current prototype presents information about the city of Torino and the surrounding Piedmont area, in Italian and in English. The system provides the user with personalized recommendations, by taking into account the possibly conflicting preferences of a group of people traveling together; e.g., a family with elderly and children. Moreover, the system offers an interactive agenda that enables the user to select the attractions she is interested in and helps her to define a tour schedule complying with her visiting preferences and other constraints, e.g., the opening times of the various attractions [Ardissono et al. 2003]. Figure 11 shows a recommendation page generated by INTRIGUE for desktop and handset devices. The suggestions are displayed as lists of items, sorted from the best to the worst. In the desktop User Interface, the recommended items are coupled with an explanation of the reasons for the suggestion.

6. RELATED WORK

Our model of agent behavior exploits the traditional action-based representation of agent activities adopted for Belief Desire Intention agents [Georgeff and Ingrand 1989], but differs in the internal management of activities. The behavior of traditional BDI agents is ruled by a cyclic interpreter that works in “busy waiting modality” and chooses the actions to be performed in a sequential way, depending on the agent state, at each cycle. In contrast, our agents perform actions in parallel, thanks to the thread-based interpreter we designed. Our agents handle autonomous actions posting goals to themselves, in a homogeneous way with respect to the service request messages they can receive from the other agents. Therefore, our agents uniformly carry out reactive and proactive behavior, balancing the two types of activities. Moreover, concurrent interpreters are created depending on the incoming requests, and the role fillers are idle when no requests have to be satisfied. More recently, the parallel execution of BDI agents has been introduced in tools such as JACK [AOS 2002], which supports posting parallel goals to the agents; however, JACK is focused on the development of homogeneous agents, while it is not so flexible as far as the integration of heterogeneous software is concerned.

The JADE environment for the development of MAS systems [Bellifemine et al. 2001], developed at the same time as Seta2000, offers a Java-based infrastructure for the specification of FIPA-compliant agent behavior [FIPA 2000] and communication; moreover, it provides a platform for managing and running the agents. The main difference between this infrastructure and the Seta2000 one concerns the abstraction level at which the agents have to be defined. In JADE, the activities to be performed by an agent are defined by “behaviors” supporting a rather low-level specification of the agent synchronization.

Other tools for the design and development of MAS systems offer high-level languages for the representation of the agent capabilities; e.g., DESIRE [Albers et al. 1999]. However, as discussed in [Shehory and Sturm 2001], they have problems in the generation of efficient code, which is essential in the development of a scalable system. For this reason, we did not exploit them to develop our recommender engine. In the development of the Seta2000 infrastructure, we decided to offer an action-based formalism for the declarative representation of actions, but, to increase

efficiency, we support a specification of the body of actions as Java code.

Several infrastructures for the development of multi-agent systems, e.g., DECAF [Graham and Decker 2000], seem to exceed our needs. In fact, they support complex activities such as the real time flexibility in the execution of tasks. In Seta2000, each role filler is an agent providing a specific set of services; due to this static association, we do not need to exploit schedulers for the distribution of activities among role fillers. Both Seta2000 and DECAF support a thread-based execution of parallel agent activities, but our approach assumes that the interpreter of an agent handles all the activities the agent is responsible for. Therefore, the agent can uniformly manage service requests and internal activities. Finally, some tools for the development of multi-agent systems focus on complementary aspects with respect to those addressed in Seta2000. For instance, RAJA [Ding et al. 2001] supports the development of resource adaptive multi-agent systems, whose configuration parameters can be dynamically tuned by controllers, in reaction to changes in the resource availability. RAJA is mostly focused on load balancing and run-time adaptation of parameters tuning the agent behavior (e.g., level of fidelity in video transmission). Instead, Seta2000 focuses on the configurability of the high-level system functionalities, such as dynamic versus static user modeling.

As *n-tier* architectures are a popular solution for complex Web-based systems, the relation between our architecture and the other approaches has to be discussed, with specific reference to frameworks like J2EE [Sun Microsystems 2002], which are industry standards for developing Web-based systems. The Seta2000 infrastructure provides the developer with an Agent-Oriented extension that can sit on top of a J2EE platform. In fact the underlying layer of Voyager can be substituted with the EJB + JMS layer, as the J2EE platform incorporated most of the advanced technologies⁷ offered by Voyager at the time the Seta2000 infrastructure was implemented. Exploiting the EJB allows interoperability in heterogeneous server environment. The standard mapping of the EJB architecture to CORBA enables a non Java platform CORBA client to access an Enterprise Bean object. On the other hand, Microsoft's .NET provides a framework for building, deploying, and running XML Web services and applications, but does not support action-based agents like Seta2000 and it supports less interoperability than J2EE, therefore it has not been selected as the underlying platform.

At a different level, the current Enterprise Portals developed for e-commerce applications represent an excellent example of specialized environments assisting the service development efforts. However, they can hardly be extended to fit specific domain requirements. Similarly, the architectures of the current personalization servers, such as [BroadVision 2002; Dynamo 2000] and [NetPerceptions 2002], are not open: they can be instantiated on a specific application domain, by they do not support the extension of their own architectures to offer new facilities.

7. INTEGRATION OF EXTERNAL SOFTWARE

The modularity and extensibility of the Seta2000 architecture enabled us to integrate in the role fillers several external software tools for the management of very

⁷For instance, asynchronous message support and the “container” concept, now offered by JMS and the J2EE Application Servers.

specific activities. For instance:

- The User Modeling Component exploits the JESS rule-based system [Sandia 2002] to maintain a structured representation of the interaction context and to produce a synthetic description of the user's behavior. Such description is then used within a Bayesian Network [Pearl 1988] to reason about the user's interests and preferences.

- The Personalization Agent exploits JTg2 [DFKI and CELI 2000] for the generation of the linguistic descriptions to be included in the Web pages. The JTg2 engine is a general-purpose generator: in its simplest conception, it takes objects as its input and returns strings. We used it to generate natural language descriptions. In both the SeTA and INTRIGUE systems, the use of NLG techniques has dramatically reduced the amount of pre-compiled information to be defined at configuration time. Moreover, it supported the generation of multilingual descriptions, tailored to different user characteristics.

8. CONCLUSIONS AND FUTURE WORK

We have presented the Seta2000 infrastructure for the development of personalized recommender systems supporting the interaction with users using different devices, such as desktop computers and mobile phones. The built-in facilities offered by the infrastructure are the adaptation to contextual parameters, e.g., the output device, and the user's preferences. However, Seta2000 offers software libraries supporting the extension of these services to satisfy specific domain requirements. Our infrastructure is based on the exploitation of Multi-Agent technologies and this paper has focused on two main topics: the management of the inter-agent communication and the design of the internal agent architecture.

The proposed infrastructure has clear pros and cons as far as scalability and applicability issues are concerned. On the one hand, we acknowledge that this architecture is less efficient than traditional distributed and component-based architectures, due to factors such as the communication overhead imposed by high-level communication languages, the thread-based agent management model, and the internal plan-based model of agent behavior.⁸

On the other hand, the same factors facilitate the development of complex ubiquitous and user-adaptive services. First of all, the introduction of high-level communication between agents abstracts from the details of the method invocation and facilitates the integration of heterogeneous (traditional and agent-based) software within the system agents. Moreover, the possibility to wrap heterogeneous components and to exploit a plan-based agent model supports the development of proactive components, which would be very expensive to develop otherwise.

Indeed, the Seta2000 infrastructure can be applied to develop social agents and components in rather different application domains, like peer-to-peer interaction between Web Service consumers and providers. As it well known, the current stan-

⁸It should be noticed that the use of threads in our framework can be seen as inefficient, but it is unlikely that in such complex systems it will be the most significant factor in the overall system performance. Moreover, action-based approaches to software development (see also planners and the like) are intrinsically heavier than procedural software.

dards for Web Service communication (e.g., WSDL [W3C 2002] and reference implementations) support one-shot interactions, but they fail to support conversations where consumers have to invoke several operations on the service provider before the service is completed (e.g., imagine the interaction with a web service supporting the configuration of a product). At the same time, the emerging Web Service standards for the management of workflow, e.g., BPEL [Curbera et al. 2002], are focused on the management of the service composition and assume a rather simple type of interaction between the provider and the individual consumer. In order to manage successful business interactions and, in the meantime, enable the consumers suitably match the provider's conversation requirements to their own business logic, a loosely coupled approach to the management of the interaction is needed. Moreover, the communication capabilities of service providers and consumers should be enhanced by means of a lightweight approach, especially on the side of the consumer, which may need to start several e-business interactions with heterogeneous providers and cannot be required to manage tightly coupled interactions with each of them, especially when the Web Service is exploited outside a complex and well established B2B relationship. To address these issues, we are currently applying the Seta2000 infrastructure to the development of an infrastructure supporting flexible, asynchronous and long-lasting conversations between Web Service consumers and providers; see [Ardissono et al. 2003] and [Petrone 2003] for details. In order to facilitate the consumer in the conversation management, we propose to exploit an action-based framework for the service invocation, which supports the consumer in the introduction of contextual conditions on the service invocation.

ACKNOWLEDGMENT

This work has been partially funded by MIUR and by the National Research Council CNR (grant number CNRG0015C3). We are grateful to Pietro Torasso for having contributed to this work with suggestions and fruitful discussions. We also thank Doug Riecken, Benjamin Pierce and the anonymous reviewers who helped us to improve this paper with very insightful comments.

REFERENCES

- ALBERS, M., JONKER, C., KARAMI, M., AND TREUR, J. 1999. An electronic market place: generic agent models, ontologies and knowledge. In *Proc. of the Agents'99 Workshop: "Agent-based decision-support for managing the Internet-enabled supply-chain"*. Seattle, WA, 71–80.
- AOS. 2002. JACK Intelligent Agents [tm]. <http://www.agent-software.com/shared/products/index.html>.
- ARDISSONO, L., BARBERO, C., GOY, A., AND PETRONE, G. 1999. An agent architecture for personalized Web stores. In *Proc. 3rd Int. Conf. on Autonomous Agents (Agents '99)*. Seattle, WA, 182–189.
- ARDISSONO, L., GOY, A., AND PETRONE, G. 2003. Enabling conversations with Web Services. In *Proc. 2nd Int. Joint. Conf. on Autonomous Agents and MultiAgent Systems*. Melbourne, Australia, to appear.
- ARDISSONO, L., GOY, A., PETRONE, G., AND SEGNAN, M. 2001. A software architecture for dynamically generated adaptive Web stores. In *Proc. 17th International Joint Conf. on Artificial Intelligence*. Seattle, WA, 1109–1114.
- ARDISSONO, L., GOY, A., PETRONE, G., AND SEGNAN, M. 2002. Personalization in business-to-consumer interaction. *Communications of the ACM, Special Issue "The Adaptive Web"* 45, 5, 52–53.

- ARDISSONO, L., GOY, A., PETRONE, G., SEGNAN, M., AND TORASSO, P. 2003. Intrigue: personalized recommendation of tourist attractions for desktop and handset devices. *Applied Artificial Intelligence, Special Issue on Artificial Intelligence for Cultural Heritage and Digital Libraries*, to appear.
- BELLIFEMINE, F., POGGI, A., AND RIMASSA, G. 2001. Developing multi agent systems with a FIPA-compliant agent framework. In *Software - Practice & Experience*. Vol. 31. John Wiley & Sons, Ltd., 103–128.
- BILLSUS, D. AND PAZZANI, M. 1999. A personal news agent that talks, learns and explains. In *Proc. 3rd Int. Conf. on Autonomous Agents (Agents '99)*. Seattle, WA, 268–275.
- BRESCIANI, P., PERINI, A., GIORGINI, P., GIUNCHIGLIA, F., AND MYLOPOULOS, J. 2001. A knowledge level software engineering methodology for agent Oriented Programming. In *Proc. 5th Int. Conf. on Autonomous Agents (Agents '01)*. Montreal, Canada, 648–655.
- BROADVISION. 2002. Broadvision. <http://www.broadvision.com>.
- COTTER, P. AND SMYTH, B. 2000. Personalisation technologies for the digital TV world. In *Proc. 14th European Conf. on Artificial Intelligence*. Berlin, 701–705.
- CURBERA, F., GOLAND, Y., KLEIN, J., LEYMAN, F., ROLLER, D., THATTE, S., AND WEERAWARANA, S. 2002. Business process execution language for web services, version 1.0. <http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/>.
- DFKI AND CELI. 2000. JTg2. <http://www.celi.it/english/tecnologia/tecLing.html/>.
- DING, Y., MALAKA, R., KRAY, C., AND SCHILLO, M. 2001. RAJA - a resource-adaptive Java agent infrastructure. In *Proc. 5th Int. Conf. on Autonomous Agents (Agents '01)*. Montreal, CA, 332–339.
- DYNAMO. 2000. Dynamo. <http://www.atg.com>.
- FININ, T., WEBER, J., WIEDERHOLD, G., GENESERETH, M., FRITZSON, R., MCGUIRE, J., SHAPIRO, S., AND BECK, C. 1993. DRAFT specification of the KQML agent-communication language. Tech. rep., The DARPA Knowledge Sharing Initiative.
- FIPA. 2000. Foundation for Physical Intelligent Agents. <http://www.fipa.org/>.
- FOWLER, M. AND SCOTT, K. 2000. *UML distilled*. ADDISON-WESLEY.
- GENESERETH, M. AND KETCHPEL, S. 1994. Software agents. *Communications of the ACM: Special Issue on Intelligent Agents* 37, 7.
- GEORGEFF, M. AND INGRAND, F. 1989. Decision-making in an embedded reasoning system. In *Proc. 11th International Joint Conf. on Artificial Intelligence*. Detroit, Michigan, 972–978.
- GRAHAM, J. AND DECKER, K. 2000. Tools for developing and monitoring agents in distributed multi agent systems. In *Proc. of the Agents'2000 workshop on Infrastructure for scalable multi-agent systems*. Barcelona.
- JENNINGS, N., SYCARA, K., AND WOOLDRIDGE, M. 1998. A roadmap of agent research and development. In *Autonomous Agents and Multi-agent Systems*. Kluwer Academic Publishers, Boston, 275–306.
- MAYBURY, M., Ed. 2000. *Special Issue on News on Demand*. Vol. 43. Communications of the ACM.
- MAYBURY, M. AND BRUSILOVSKY, P., Eds. 2002. *The adaptive Web*. Vol. 45. Communications of the ACM.
- NETPERCEPTIONS, INC. 2002. Net perceptions. <http://www.netperceptions.com>.
- OBJECTSPACE. 2000. Voyager. <http://www.objectspace.com/index.asp>.
- O'CONNOR, M., COSLEY, D., KONSTAN, J., AND RIEDL, J. 2001. PolyLens: a recommender system for groups of users. In *Proc. European Conference on Computer Supported Cooperative Work (ECSCW 2001)*. Bonn, Germany.
- PEARL, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Mateo, CA.
- PETRIE, C. 1996. Agent-based engineering, the Web, and intelligence. *IEEE Expert* December, 24–29.
- PETRONE, G. 2003. Managing flexible interaction with Web Services. In *AAMAS-03 workshop on Web-services and agent-based engineering*. Melbourne, Australia, 41–48.

- RAO, A. AND GEORGEFF, M. 1991. Modeling rational agents within a BDI-architecture. In *Proc. 2th Int. Conf. Principles of Knowledge Representation and Reasoning*. Cambridge, MA, 473–484.
- RESNICK, P. AND VARIAN, H., Eds. 1997. *Special Issue on Recommender Systems*. Vol. 40. Communications of the ACM.
- RIECKEN, D., Ed. 2000. *Special Issue on Personalization*. Vol. 43. Communications of the ACM.
- SANDIA NATIONAL LABORATORIES 2002. JESS, the Java Expert System Shell. <http://herzberg.ca.sandia.gov/jess/>.
- SHEHORY, O. AND STURM, A. 2001. Evaluation of modeling techniques for agent-based systems. In *Proc. 5th Int. Conf. on Autonomous Agents (Agents '01)*. Montreal, CA, 624–631.
- STEIN, A. AND MAIER, E. 1994. Structuring collaborative information-seeking dialogues. *Knowledge-Based Systems 8*, 2-3, 82–93.
- SUN MICROSYSTEMS, I. 2002. Java 2 Platform Enterprise Edition. <http://java.sun.com/j2ee/>.
- SYCARA, K., PANNU, A., WILLIAMSON, M., AND ZENG, D. 1996. Distributed intelligent agents. *IEEE Expert December*, 36–45.
- W3C. 2002. Web Services Definition Language. <http://www.w3.org/TR/wsdl>.
- WOOLDRIDGE, M., JENNINGS, N., AND KINNY, D. 1999. A methodology for Agent-Oriented analysis and design. In *Proc. 3rd Int. Conf. on Autonomous Agents (Agents '99)*. Seattle, WA, 69–76.

Received M Y; revised M Y; accepted M Y