Configurability within a multi-agent Web store shell

L. Ardissono, A. Goy, G. Petrone, M. Segnan
Dipartimento di Informatica, University of Torino
Corso Svizzera 185; 10149 Torino, Italy
Phone: +39 - 011 - 7429111;
{liliana, goy, giovanna, marino}@di.unito.it

1. Introduction

The configurabilty of a Web store shell strongly benefits from the possibility to create simpler on-line stores, offering a subset of the functionalities supported by the shell: in fact, some functionalities may exceed the requirements of a specific sales domain and, at the same time, they may impose an overhead on the knowledge to be introduced at configuration time and on the interaction with the customer.

In this abstract, we sketch the approach we adopted in the design of SETA [1, 2], a prototype toolkit for the creation of adaptive Web stores. SETA is based on a multi-agent architecture and on the definition of a hierarchy of agents supporting progressively more sophisticated functionalities, for each main role of the architecture. This approach supports flexibility in the configuration of a Web store instance, allowing the store designer to choose, for each main role, the specific agent offering the desired functionalities.

The SETA architecture [1] includes specialized agents, devoted to the management of the front-end of an adaptive Web store: the SessionManager connects the store to the Web; the Dialog Manager handles the logical interaction with the customer, deciding which page has to be shown next. The User Modeling Component handles the user models; the Product Extractor selects the items to be suggested; the Personalization Agent generates the Web catalog pages.

2. Classes of agents

The agent-based techniques are very useful to enhance the configurability of systems: in fact, they support the extension of a system with new agents filling supplementary roles; moreover, they support the definition of agent hierarchies to exploit inheritance mechanisms for sharing data structures and behavior. In particular, we distinguish the *services* offered by an agent from its *functionalities*: services depend on the role filled by the agent and correspond to the requests which the agent can receive and reply to. Functionalities concern how the agent processes the incoming messages and carries out the related activities. Two agents offering the same services may support different functionalities; e.g., the management of the user models can be filled by a hierarchy of alternative User Modeling Components (UMCs):

- The first and simplest one supports the construction of a generic user model, the same for every user.
- The second agent exploits stereotypical information about customer classes to predict the user's preferences.
- The third agent tracks the user behavior to dynamically update the user model during the interaction.

More complex agents can be obtained by merging the functionalities offered by these basic agents; e.g., when stereotypical and dynamic user modeling have to be combined.

The agents of SETA are characterized by different behaviors, imposed by their roles; some agents autonomously carry on internal activities, by interleaving them with the provision of services (e.g., the UMC); others interact with other agents, as well as with the user (e.g., the Dialog Manager); others only work when a service is requested (e.g., the Personalization Agent). To obtain a uniform framework, where all these types of agent can be designed, we have defined a general model of agent behavior, which is inherited by each agent of the system and supports the selection and execution of the agent activities, autonomously, or in response to service requests. Our approach is integrated in the environment of an agent-building tool (Objectspace Voyager), which we have used to wrap our agents, enabling them to run in parallel and to communicate by means of synchronous and asynchronous messages.

3. A model of agent behavior

We have adopted an action-based model of agent behavior to describe our agents in a declarative way, supporting the inheritance of behavior and the possibility to represent types of agent like those mentioned above: an "Agent" class provides the basic data structures and methods for specifying the agent behavior; then, the individual agents can be designed by extending such class with specific information about the actions that the agent can perform, and so forth. The "Agent" class is characterized as follows:

a) The *state* describes the environment of the agent with respect to a specific user session[1] and evolves while the agent sends or receives messages or performs actions. For instance, the state of the UMC specifies whether the user model has been created or not, the user's personal data have been set, and so forth.

b) The *actions* include the tasks to be performed in response to the invocation of services by the other agents, as well as the internal activities to be carried on by the agent; e.g., revising the user model on the basis of the user behavior. The actions have the following slots: the *goal* of an action is the goal resulting from its execution and is the basic information exploited for selecting the actions to be performed when a service is requested. The *preconditions* are applicability conditions that the agent state must satisfy for the

[1] The agents handle parallel user sessions.

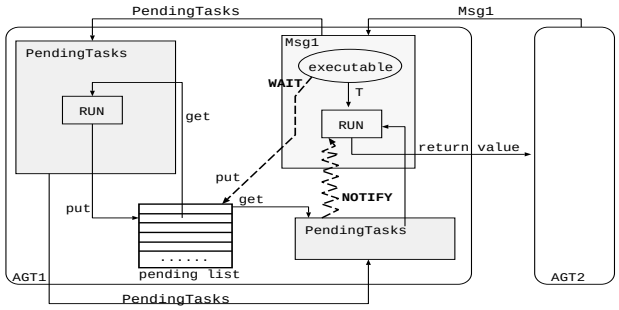

Figure 1: Interpreter threads within an agent.

action to be executable. The *body* represents the sequence of steps to be performed in order to complete the action. As a result of the execution of an action, the agent state changes (so, no postconditions are explicitly represented). The *type* distinguishes actions associated to public services from internal activities.

c) The *interpreter* describes the agent behavior: it selects the actions to be performed, on the basis of the agent state and of the requests to be satisfied. Notice that checking the preconditions of actions is the basis for the coordination of the activities that the agent has to carry on.

Since both the internal activities and the service provision tasks are represented as actions, they can be selected and performed by the interpreter in a uniform way. The crucial issue is how to trigger the internal activities: in fact, while the services are requested by the other agents via suitable messages, these activities have to be autonomously carried on, whenever their preconditions are true. For uniformity, the execution of internal activities is also triggered by means of request messages. Thus, an agent both receives service requests as messages from other agents and sends messages to itself for triggering its own internal activities.

We have integrated our agent model with the Voyager thread environment by exploiting asynchronous messages and defining a one-shot interpreter for processing their contents: when the agent receives a message, it runs the interpreter as a thread of the main agent process, to satisfy the request included in the message. At the beginning of the execution, each interpreter thread launches (by message invocation) the pending tasks that the agent is supposed to carry on: these tasks include the internal activities and the suspended service requests, which could not be processed because the preconditions of the related actions were false at the time the agent received the related messages. Figure 1 shows the situation of an agent ("AGT1") where three interpreter threads are active: "Msg1" has been generated to process a service request by "AGT2"; the other two ("pendingTasks") have been generated to handle the pending tasks.

Given a message to be processed, 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 a list of pending tasks ("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 ("NOTIFY"). The "notify" signals are generated by the interpreter threads invoked to perform the pending tasks. In particular, the execution of the "pendingTasks" action requires that an interpreter thread selects an executable action from the pending list and processes it, performing the action, if it is an internal activity, or notifying the suspended thread, if it is a service provision action. Moreover the thread generates another "pendingTasks" message, to guarantee that the agent will inspect the pending list again.[2]

4. Configurability issues

We have defined a model of agent behavior supporting the management of service provision and of internal agent activities, whose execution has to be carried on by the agent in a transparent way with respect to the rest of the system. Individual agents can be defined by extending this model, possibly overriding parts of it (e.g., an individual agent inherits the structure of the state, the interpreter and the "pendingTasks" action; moreover, it has to define all the actions implementing its own activities).

A declarative representation of the activities to be carried on by the agent as actions with goals and preconditions has several advantages: e.g., internal activities and service requests can be handled by means of the same message-interpretation mechanism, thus simplifying the coordination of the various tasks. Moreover, the agents can be easily updated to offer new functionalities: for instance, an individual agent activity can be introduced (removed) by defining (removing) the related action. Furthermore, constraints may be added to the preconditions of actions if supplementary constraints are needed to coordinate their execution (e.g., to impose partial orders in the execution of the various actions). Finally, hierarchies of agents inheriting certain actions and overriding other actions can be defined, in order to offer a family of alternative agents filling the same role.

We have exploited our agent model to enhance the configurability the SETA Web store shell, which we have revised by defining the state and the activities of all the system agents. For some of the main roles of the architecture, we have defined a hierarchy of alternative agents which can be selected to create a Web store instance providing only the desired functionalities. For instance, we have defined a hierarchy of UMCs inheriting the general agent behavior and implementing the activities to handle progressively sophisticated user models. When the store designer creates a new Web store, she can select a specific UMC on the basis of the needed type of user model (e.g., static, stereotypical, dynamic, etc.).

This work is developed in the project "Servizi Telematici Adattativi", (http://www.di.unito.it/~seta) carried on at the CS Department of the University of Torino in the initiative "Cantieri Multimediali", granted by Telecom Italia.

REFERENCES

[1] L. Ardissono, C. Barbero, A. Goy, and G. Petrone. An agent architecture for personalized web stores. In *Proc. 3rd Int. Conf. on Autonomous Agents (Agents '99)*, pages 182–189, Seattle, WA, 1999.

[2] L. Ardissono and A. Goy. Tailoring the interaction with users in electronic shops. In *Proc. 7th Int. Conf. on User Modeling*, pages 35–44, Banff, Canada, 1999.

[2] In this way, the agent can generate a thread to handle the internal activities every time it receives a message and every time an action is selected from the pending list for execution. Moreover, no messages are sent when the list is empty, or it does not contain any executable actions, in order to limit the attempts to perform them before the agent state changes.