Programming Goal-Driven Web Sites Using an Agent Logic Language

Matteo Baldoni, Cristina Baroglio, Alessandro Chiarotto, and Viviana Patti

Dipartimento di Informatica — Università degli Studi di Torino,
C.so Svizzera 185, I-10149 Torino (Italy)
{baldoni,baroglio,patti}@di.unito.it
http://www.di.unito.it/∼alice

Abstract. In this paper we show how an agent programming language, based on a formal theory of actions, can be employed to implement adaptive web applications, where a personalized dynamical site generation is guided by the user's needs. For this purpose, we have developed an on-line computer seller in DyLOG, a modal logic programming language which allows one to specify agents acting, interacting, and planning in dynamic environments. Adaptation at the navigation level is realized by dynamically building a presentation plan for solving the problem to assemble a computer, being driven by goals generated by interacting with the user. The planning capabilities of DyLOG are exploited to implement the automated generation of a presentation plan to achieve the goals. The DyLOG agent is the "reasoning" component of a larger system, called WLog, which is described in this paper.

1 Introduction and Related Works

Recent years witnessed a rapid expansion of the use of multimedia technologies, the web in particular, for the most various purposes: advertisement, information, communication, commerce and distribution of services are just some examples. One problem that arises because of the wide variety of users of such tools is to find a way for adapting the presentation to the particular user. Many of the most advanced solutions [20, 18, 1, 9, 10] start from the assumption that adaptation should focus on the user's own characteristics, thus, though in different ways, they all try to associate him/her with a reference prototype (also known as the "user model"); the presentation is then adapted to user prototypes. The association between the user and a model is done either a priori, by asking the user to fill a form, or little by little by inducing preferences and interests from the user's choices. In some cases, such as in the SeTA project [2], a hybrid solution is adopted where, first, an a priori model is built and, then, it is refined by exploiting the user's choices and selections.

In our view, such solutions lack one important feature: the representation of the user's "goals" (or intentions), which may change at every connection. Let us suppose, for example, that I access to an on-line newspaper for some time, always searching for sport news. One day, after I have heard about a terrible accident

I.V. Ramakrishnan (Ed.): PADL 2001, LNCS 1990, pp. 60–75, 2001.

in a foreign country, I access to the newspaper for getting more information. If the system has, meanwhile, induced that my only interest is sport, I could have difficulties in getting the information I am interested in or have it presented at a too shallow level of detail with respect to my current interest. This kind of inconvenient would not have happened if the system I interacted with tried to capture my goal for the current connection and only after it tried to adapt the presentation to me (not limiting its reasoning to my generic interests only).

Seller (asking):	-What do you need the computer for?
Client:	-I would use it for **multimedia** purposes.
Seller (thinking):	*-Well, let me think, he needs a configuration with a huge monitor, any kind of RAM, any kind of CPU, a sound-card, and a modem. But he may have some of these components. Let's list them.*
Seller (asking):	-Do you already have any of the listed components?
Client:	-Yes, I have a fast CPU and a sound-card.
Seller (asking):	-Do you have a limited budget?
Client:	-Yes, 650 Euro.
Seller (thinking):	*-He needs a monitor, RAM and a modem. I need to* **plan** *according to his needs, budget and current availability of the components in the store.*
Seller (asking):	-I have different configurations that satisfy your needs. Let me ask first which of the listed RAMs you prefer.
Client:	-I prefer to have 128MB of RAM.
Seller (informing):	I propose you this configuration. Do you like it?
	...

Table 1. Example of dialogue between a client and a seller

Our research aims at addressing this deficit, building a web system that, during the interaction with the user, adopts the user's goals in order to achieve a more complete adaptation. More technically, we study the implementation of web sites which are "structureless", depending their shape on the *goals* that the single users have. The structure of the web site depends on the interaction between the user and a server-side agent system. The notion of *computational agent* is central in artificial intelligence (AI), because it supplies a powerful abstraction tool for characterizing complex systems, situated in dynamic environments, by using mentalistic notions. In this perspective, we describe a system in terms of its beliefs about the world, its goals, and its capabilities of acting; the system must be able to autonomously plan and execute action sequences for achieving its purposes. When a user connects to a site managed by one of our agents, (s)he does not access to a fixed graph of pages and links but interacts with a program which, starting from a knowledge base specific to the site and from the requests of the user, builds an *ad hoc structure*. This is a difference with respect to current dynamic web sites, where *pages* are dynamically constructed but *not* the *site structure*. In our approach, such a structure corresponds to a

plan aimed at pursuing the user's *goals*. So, the user's goal is the bias that orients the presentation of domain-dependent information (text, images, videos and so forth). This approach is "orthogonal" to the one based on user models, which is already widely studied in the literature.

Our approach could also recall Natural Language cooperative dialogue systems but there are some differences. For instance, in [8] a logic of rational interaction is proposed for implementing the dialogue management components of a spoken dialogue system. This work is based (like ours, as we will see) on dynamic logic and reasoning capabilities on actions and intentions are exploited to plan dialogue acts. Our focus, however, is not on recognizing or inferring user's intentions. User's needs are taken as input by the software agent, that uses them to generate the goals that will drive its behaviour. The novelty of our approach stands in exploiting planning capabilities not for dialogue act planning but for building web site presentation plans guided by the user's goal. In this perspective the structure of the site is built by the system as a conditional plan, and according to the initial user's needs and constraints. The execution of the plan by the system corresponds to the navigation of the site, where the user and the system cooperate for building the configuration satisfying the user's needs. Indeed, during the execution the choice between the branches of the conditional plan is determined by means of the interaction with the user.

The language that we used for writing the server-side agent program is DyLOG [7, 5, 4]. DyLOG is based on a logical theory for reasoning about actions and change in a logic programming setting. It allows one to specify an agent's behavior by defining both a set of simple actions that the agent can perform (they are defined in terms of their preconditions and effects) and a set of complex actions (procedures), built upon simple ones. DyLOG has been implemented in Sicstus Prolog; a specification in DyLOG can be executed by an interpreter, which is a straightforward implementation of the proof procedure of the language.

This language is particularly interesting for agent programming, and in particular for our application, for two of its main characteristics. The first is that, being based on a formal theory of actions, it can deal with reasoning about action effects in a dynamically changing environment and, as such, it *supports planning*. Reasoning about the effect of actions in a dynamically changing world is one of the main problems that must be faced by intelligent agents, even in the case in which we consider the internal dynamics of the agent itself, i.e. the way it updates its beliefs and its goals, that can be regarded as the result of the execution of actions on its mental state. The second is that the logical characterization of the language is very close to the procedural one, and this allows to *reduce the gap* between theory and practical use. In the literature, other languages (e.g. GOLOG [16]) have been developed for reasoning in dynamic domains and for agent programming. However, there was an advantage in using DyLOG in the current work: it has a sound proof procedure, which practically allows to deal with the planning task in presence of sensing. Indeed conditional plans (not only linear plans) can be automatically extracted, as we better explain in Section 3.4.

To summarize, in this work we use DyLOG for building cognitive agents that autonomously reason on their own behavior in order to obtain web site adaptation at the *navigation level*, i.e. to dynamically generate a site being guided by the user's goals; such goals are *explicitly adopted* by the system throughout the entire interaction. In this domain, one of the most important aspects is to define the navigation possibilities available to the user and to determine which page to display, based on the dynamics of the interaction. The system that we realized and that uses the DyLOG agent is called WLog.

The approach that we propose brings along various innovations. From a *human-machine interaction* perspective, the user will not have to fill forms where pieces of information which (s)he does not feel as useful to explore the site are requested (for instance, his/her education). Moreover, the system will not restrict its answers to a user model which is either fixed or past-oriented; other advantages are expected on the web site *build-modify-update process*. In order to modify a classical web site, one has to change the contents of the pages and the links between them. In our case, the site does not exist as a given structure, there exist data contained in a data base, whose maintenance is much simpler, and a program (the agent's program), which is likely to be changed very rarely, since most of the changes are related to the data and to the structure by which they are presented, not in the way this structure is built. Last but not least, this approach allows a *fast prototyping* of sites as well as it allows the validation of how the information is presented.

2 A Case Study

The application that we will use as a case study deals with the construction of a *virtual computer seller*. Computer assembly is a good application domain because the world of hardware components is rapidly and continuously evolving so that, on one hand, it will be very expensive to keep a more classical (static) web site up-to-date; on the other hand, it is unlikely that clients are equally up-to-date and know the technical characteristics or even just the names of processors, motherboards, etc. It will also allow comparison with the literature because this domain has been used in other works, such as [17].

Furthermore, what a computer buyer wants, what (s)he often only knows, is what (s)he needs the computer for. Sometimes the desired use belongs to a category (e.g. world processing or internet browsing), sometimes it is more peculiar and maybe related to a specific job (e.g. use of CAD systems for design). In a real shop the choice would be taken thanks to a *dialogue* between the client and the seller (see Table 1 for an example), dialogue in which the latter tries to understand which function the client is interested in, proposing proper configurations. The client, on the other hand, can either accept or refuse the proposals, maybe specifying further details or constraints. Every new piece of information will be used for converging to a proposal that the client will accept.

In the case of on-line purchase, it is reasonable to offer a similar interaction; this is what our system tries to do. In this application, the seller and the client

have the *common goal* to build a computer, by joining their competences, which in the case of the seller is technical whereas in the case of the client is related both to the reasons for which (s)he is purchasing that kind of object and to his/her constraints (e.g. the budget). The seller leads the interaction, by applying a plan (see Figure 1 for an example) aimed at selling. The goal of such a plan is to find a configuration which satisfies the goal of the client. Observe that, although the final purpose is always the same, the variety of the possible situations is so wide that it is not convenient to build a single, general, and complete plan that can be used in all of the cases; on the contrary it is better to build the plan depending on the current conditions. In this way it is possible to develop the virtual seller in an incremental way, by augmenting its operational knowledge or by making it more sophisticate. Once a plan has been defined, the virtual seller follows it for making the proposals/requests that it considers as the most adequate.

3 The Agent Programming Language

In this section, we will recall the definition of the logic language DyLOG, referring to the proposal in [4], and its extension to deal with complex actions and knowledge-producing actions [5, 7]. DyLOG is based on a *modal action theory* that has been developed in [4, 14, 15]. It provides a nonmonotonic solution to the frame problem by making use of abductive assumptions and it deals with the ramification problem by introducing a "causality" operator. In [5, 7] the action language has been extended to deal with complex actions and knowledge producing actions. The formalization of complex actions draws considerably from dynamic logic. As a difference, rather than referring to an Algol-like paradigm for describing complex actions as in GOLOG [16], it refers to a Prolog-like paradigm: instead of using the iteration operator, complex actions are defined through (possibly recursive) definitions, given by means of Prolog-like clauses. The nondeterministic choice among actions is allowed by alternative clause definitions.

The adoption of Dynamic Logic or a modal logic to deal with the problem of reasoning about actions and change is common to many proposals, such as for instance [12, 19, 11], and it is motivated by the fact that modal logic allows a very natural representation of actions as state transitions, through the accessibility relation of Kripke structures. Since the intentional notions (or attitudes), which are used to describe agents, are usually represented as modalities, our modal action theory is also well suited to incorporate such attitudes.

In [5, 7] the *planning problem* has been addressed and a goal directed proof procedure for reasoning on complex actions and for extracting linear and conditional plans has been introduced. In [6] it has been described how to use our implementation of DyLOG as an *agent programming language*, for *executing* procedures which model the behaviour of an agent, but also for *reasoning* about them, by extracting from them linear or conditional plans.

3.1 Primitive Actions: The Skills of the Agent

In our action language each primitive action $a \in A$ is represented by a modality $[a]$. The meaning of the formulas $[a]\alpha$ is that α holds after any execution of action a. The meaning of the formula $\langle a\rangle\alpha$ is that there is a possible execution of action a after which α holds. We also introduce a modality $\Box$, which is used to denote those formulas that hold in all states, that is, after any action sequence.

A *state* consists of a set of *fluents*, i.e. properties whose truth value may change over the time. In general we cannot assume that the value of each fluent in a state is known to an agent, and we want to be able of representing the fact that some fluents are unknown and to reason about the execution of actions on incomplete states. To represent explicitly the unknown value of some fluents, in [7] we introduce an epistemic level in our representation language. In particular, we introduce an epistemic operator $\mathcal{B}$, to represent the beliefs an agent has on the world: $\mathcal{B}f$ will mean that the fluent f is known to be true, $\mathcal{B}\neg f$ will mean that the fluent f is known to be false. Fluent f is undefined in the case both $\neg\mathcal{B}f$ and $\neg\mathcal{B}\neg f$ hold. We will write $u(F)$ for $\neg\mathcal{B}f \wedge \neg\mathcal{B}\neg f$. In our implementation of DyLOG (and also in the following description) we do not explicitly use the epistemic operator $\mathcal{B}$: if a fluent f (or its negation $\neg f$) is present in a state, it is intended to be believed, unknown otherwise. Thus each fluent can have one of the three values: true, false or unknown. We use the notation $u(F)?$ to test if a fluent is unknown (i.e. to test if neither f nor $\neg f$ is present in the state).

Simple action laws are rules that allow one to describe direct and indirect effects of primitive actions on a state. Basically, simple action clauses consist of *action laws*, *precondition laws*, and *causal laws*:

- *Action laws* define *direct* effects of primitive actions on a fluent and allow actions with conditional effects to be represented. They have the form $\Box(Fs \rightarrow [a]F)$, where a is a primitive action name, F is a fluent, and Fs is a fluent conjunction, meaning that action a has effect on F, when executed in a state where the *fluent preconditions* Fs hold.
- *Precondition laws* allow *action preconditions*, i.e. those conditions which make an action executable in a state, to be specified. Precondition laws have form $\Box(Fs \rightarrow \langle a\rangle true)$, meaning that when the fluent conjunction Fs holds in a state, execution of the action a is possible in that state.
- *Causal laws* are used to express causal dependencies among fluents and, then, to describe *indirect* effects of primitive actions. They have the form $\Box(Fs \rightarrow F)$, meaning that the fluent F holds if the fluent conjunction Fs holds too[1].

In DyLOG we have adopted a more readable notation: action laws have the form "a **causes** F **if** Fs" precondition laws have the form "a **possible_if** Fs" and causal rules have the form "F **if** Fs".

[1] In a logic programming context we represent causality by the directionality of implication. A more general solution, which makes use of modality "causes", has been provided in [15].

As an example, one of the actions of our selling agent is the following:

add(monitor(X)) : add the monitor X to the current configuration
(1) $add(monitor(X))$ **possible_if** $true$.
(2) $add(monitor(X))$ **causes** $has(monitor(X))$.
(3) $add(monitor(X))$ **causes** $in\_the\_shopping\_cart(monitor(X))$ **if** $true$.
(4) $add(monitor(X))$ **causes** $credit(B1)$ **if** $get\_value(X, price, P) \wedge credit(B) \wedge (B1\ is\ B + P)$.

Rule (1) states that the action $add(monitor(X))$ is always executable. Action laws (2)-(4) describe the effects of the action's execution: adding the monitor of type X causes having the monitor X in the configuration under construction (2), having it into the shopping cart (3), and updating the current credit by summing the monitor price (4). Analogously, we define the other seller's primitive actions of adding to the configuration a CPU, a RAM, or a peripheral.

In other cases, an action can be applied only if a given condition is satisfied. For instance, a *motherboard* can be added only if a CPU is already available; furthermore, the CPU and the motherboard must be compatible:

add(mother(X)) : add the compatible motherboard X to the current configuration
(5) $add(mother(X))$ **possible_if** $has(cpu(C))$.
(6) $add(mother(generic))$ **causes** $has(mother(X))$ **if** $has(cpu(C)) \wedge get\_mother\_comp(C, X)$.

Intuitively, an action can be executed in a state s if the preconditions of the action hold in s. The execution of the action modifies the state according to the action and causal laws. Furthermore we assume that the value of a fluent *persists* from one state to the next one, if the action does not cause the value of the fluent to change.

3.2 The Interaction with the User: Sensing and Suggesting Actions

The interaction of the agent with the user is modeled in our language by means of actions for gathering inputs from the external world.

In [7] we studied how to represent in our framework a particular kind of informative actions, called *sensing actions*, which allow an agent to *gather knowledge from the environment* about the value of a fluent F, rather than to change it. In DyLOG direct effects of *sensing actions* are represented using *knowledge laws*, that have form "s **senses** F", meaning that action s causes to know whether F holds[2]. Generally, these effects are interpreted as inputs from outside that are not under the agent control but in a broad sense, executing a sensing action allows an agent to *interact* with the external world to determine the value of certain fluents. In this paper we are interested in modelling a particular kind of interaction: the *interaction of the agent system with the user*. In this case the user is

[2] See [7] for the translation of knowledge laws in the modal language.

explicitly requested to enter a value for a fluent, true or false in case of ordinary fluents, a value from the domain in case of fluents with an associated finite domain. The interaction is carried on by a sensing action. In our running example, for instance, we introduce the binary sensing action $ask\_if\_has\_monitor(M)$, for knowing whether the user has already a monitor of type M:

$ask\_if\_has\_monitor(M)$ **possible_if** $u(user\_has(monitor(M)))$.
$ask\_if\_has\_monitor(M)$ **senses** $user\_has(monitor(M))$.

Specifically for the web application domain, we have also defined a special subset of sensing actions, called *suggesting actions* which are useful when an agent has to find out the value of fluents representing the *user's preferences* among a finite subset of alternatives. The number and values of the alternatives depend on the particular interaction that is being carried on. Generally, the agent will suggest different subsets of choices to different users. When performing this kind of actions the agent does not read an input in a passive way, but has an active role in selecting (after some reasoning) the possible values among which the user chooses. In particular, only those values that lead to fulfill the goal will be selected. These are the intuitive motivations to introduce the *suggesting actions*. Formally, the difference w.r.t. normal sensing actions is that while those consider as alternative values for a given fluent its whole domain, suggesting actions allow to offer only a subset of it.

For representing the effects of such actions we use the notation "s **suggests** F", meaning that action s suggests a possibly selected set of values for fluent F and causes to know the value of F. As an example, our virtual seller can perform a suggesting action to offer to the user the choice among the available kinds of monitor:

$offer\_monitor\_type$ **possible_if** $true$.
$offer\_monitor\_type$ **suggests** $type\_monitor(X)$.

The range of X will be computed during the interaction with the user and will be a subset of the finite domain associated to $type\_monitor(X)$.

3.3 Procedures: The Agent's Behavior Strategies

Procedures are used to describe the behaviour of an agent. In particular, for each goal driving its behavior, a rational agent has a set of procedures (sometimes called plans) which can be seen as strategies for achieving the given goal.

In our language, *procedures* define the behavior of *complex actions*. Complex actions are defined on the basis of other complex actions, primitive actions, sensing actions and *test* actions. Test actions are needed for testing if some fluent holds in the current state and for expressing conditional complex actions. They are written as "(Fs)?", where Fs is a fluent conjunction. A *procedure* is defined as a collection of *procedure clauses* of the form

$$p_0 \textbf{ is } p_1, \ldots, p_n \quad (n \geq 0)$$

where p_0 is the name of the procedure and p_i, $i = 1, \ldots, n$, is either a primitive action, or a sensing action, or a test action, or a procedure name (i.e. a procedure call)[3]. Procedures can be recursive and can be executed in a goal directed way, similarly to standard logic programs.

From the logical point of view procedure clauses have to be regarded as axiom schemas of the logic. More precisely, each procedure clause p_0 **is** $p_1, \ldots, p_n$, can be regarded as the axiom schema[4] $\langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi \supset \langle p_0 \rangle \varphi$. Its meaning is that if in a state there is a possible execution of p_1, followed by an execution of p_2, and so on up to p_n, then in that state there is a possible execution of p_0.

A procedure can contain suggesting actions which are interpreted as inputs from the user and, as we will see, allow us to carry on a dialogue with him/her.

3.4 Planning and Execution

DyLOG programs are executed by an interpreter which is a straightforward implementation of the proof procedure. In general, the execution of an action will have an effect on the environment, such as, in our case, showing a web page to the user. This can be specified in DyLOG by associating with each primitive action some code that implements the effects of the action on the world (e.g. in our case the agent asks to the actual execution device, a web server, to send a given web page to the browser, see Section 4). Therefore, when the interpreter executes an action it must commit to it, and it is *not allowed to backtrack* by retracting the effects of the action. Thus procedures are deterministic or at most they can implement a kind of "don't care" non-determinism.

However, a rational agent must also be able to cope with complex or unexpected situations by *reasoning about the effects* of a procedure before executing it. We can deal with this case by using the language for reasoning about actions and, thus, for planning; the agent can do *hypothetical reasoning* on possible sequences of actions by exploring different alternatives.

In general, a *planning problem* amounts to determine, given an initial state, if there is a sequence of actions that, when executed in the initial state, leads to a goal state in which Fs holds. In our context, in which complex actions can be expressed as procedures, we can consider a specific instance of the planning problem in which we want to know if there is a possible execution of a procedure p leading to a state in which some condition Fs holds. In such a case the execution sequence is not an arbitrary sequence of atomic actions but it is an execution of p. In other words, the procedure definitions *constrain* the space in which the desired sequence is sought for. This can be formulated by the query $\langle p \rangle Fs$, which asks for a terminating execution of p (i.e. a finite action sequence) leading to a state in which Fs holds. The execution of the above query returns as a side-effect an answer which is an *execution trace* "$a_1, a_2, \ldots, a_m$"; such a trace is a

[3] Actually in DyLOG p_i can also be a Prolog goal.

[4] These axioms have the form of rewriting rules as in grammar logics. In [3] decidability results for subclasses of grammar logics are provided, and right regular grammar logics are proved to be decidable.

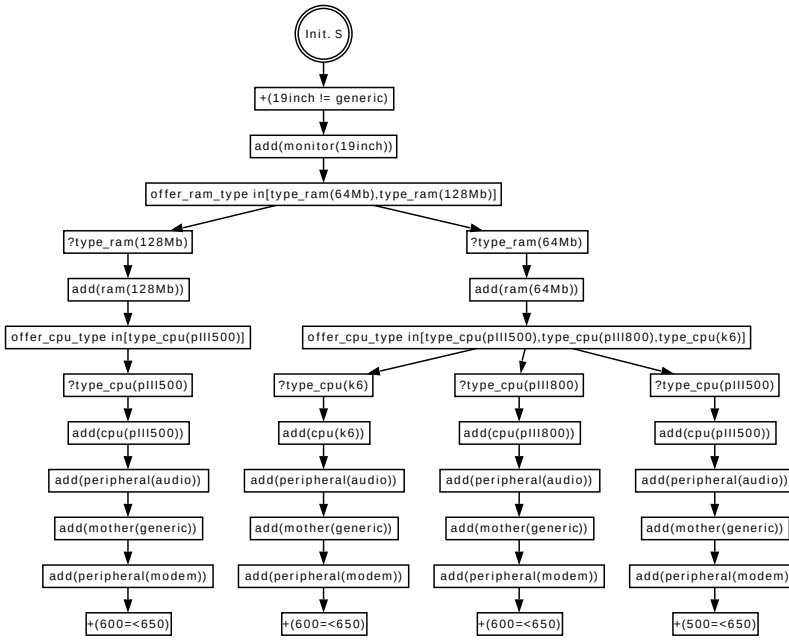

Fig. 1. The result of the planning process when the user does not have any component and has a budget of 650.

sequence of primitive actions that leads from the initial to the final state and it corresponds to a *linear plan*.

To achieve this, the DyLOG implementation provides a metapredicate *plan(p, Fs, as)*, where p is a procedure, Fs a condition on the goal state and as a sequence of primitive actions. The procedure p can be nondeterministic, and *plan* will extract from it a sequence as of primitive actions, a *plan*, corresponding to a possible execution of the procedure, leading to a state in which Fs holds, starting from the current state. Procedure *plan* works by executing p in the same way as the interpreter of the language, with a main differences: primitive actions are executed without any effect on the external environment, and, as a consequence, they are backtrackable.

Since procedures can contain sensing (or suggesting) actions, whose outcomes are unknown at planning time, all the possible alternatives are to be taken into account. Therefore, by applying DyLOG planning predicate to a procedure that contains sensing actions we obtain a *conditional plan* whose branches correspond to the possible outcomes of sensing (or suggesting).

4 The WLog System

In this section we describe the agent system that we developed and applied to the computer selling case: WLog.

4.1 Architecture

The architecture is sketched in Figure 2. In the current implementation, the system consists of two kinds of agents: *reasoners* and *executors*. Reasoners are programs written in DyLOG whereas executors are Java servlets embedded in an Apache web server. The connection between the two kinds of agents has the form of message exchange. Technical information about the system, the Java classes that we defined, and the DyLOG programs (as well as our virtual seller example) can be found at `http://www.di.unito.it/~alice`.

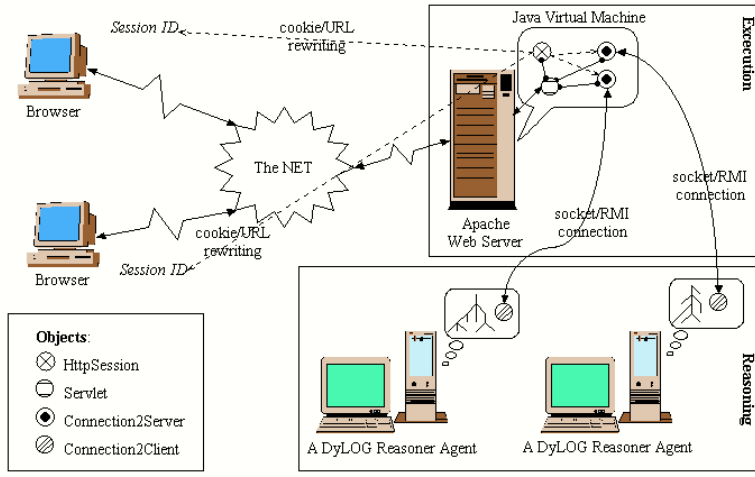

Fig. 2. A sketch of WLog architecture.

As explained in the previous sections, the interaction between the user and WLog starts with the declaration of the user's goal, that is, in our case study (see Section 2), what (s)he needs a computer for. The goal can either belong to a set of alternatives that was defined a priori (such as "multimedia") or they can be a DyLOG query. The web server (Apache) properly dispatches the requests to a free reasoner (if any is available), that from that moment till the end of the interaction will be dedicated to serve that specific client. The dispatch and the interaction are done by means of the java servlets, that work as an interface. The connection with the reasoners can currently be done either by means of sockets or by means of Remote Method Invocation but it can easily be extended with other kinds of communication systems.

Once the goal is passed on to the reasoner, the reasoner produces a conditional plan aimed at reaching that goal. In our example, it will produce a plan for assembling a computer whose main use will be "multimedia". Each path in the plan will correspond to a different computer configuration; however, all configurations in the conditional plan satisfy the user's intentions (see, for instance, Figure 1). The conditional plan is the web site that can virtually be navigated by the user.

Once built, the plan is executed. *Executing* a conditional plan implies following one of the paths; only the part of the site corresponding to this path will actually be built. The execution of some of the actions in the path consists in showing to the user one or more web pages; in particular, those actions (sensing and suggesting) that correspond to a branching point, require a feedback from the user. Other actions only affect the reasoner's mental state.

The actual execution (i.e. producing and showing web pages to the user) is a task of the *executor* (see Figure 2). The most general case is the one in which a set of alternatives for a given component is available. In our application domain the choice of which to buy is up to the user, so (s)he will be shown an HTML page containing the possible alternatives. Thus the interaction between the reasoner and the executor is as follows: the reasoner sends to the executor a command of the kind "offer CPU1, CPU2" and the executor produces some HTML code that contains the information related to the two CPU's identified by CPU1 and CPU2 plus the request to make a choice. Once an answer is returned from the client, this is passed on to the reasoner, which performs an action that is transparent to the client, that consists in adding the new fact to the knowledge base and take it into account for passing to the next step: sending to the executor the information about which page to show next. All unselected alternatives are forgotten.

4.2 The Virtual Seller: An Example of Agent Program

In this section we briefly report an example of *reasoner*, that we used in our case study. The behaviour of our selling agent is described by giving a collection of procedures. The top level procedure, *build_a_computer*, produces a plan and follows it.

$$\begin{array}{c} build\_a\_computer \ \textbf{is} \ get\_user\_preferences; get\_max\_value\_budget; \\ plan(assemble, credit(C) \wedge budget(B) \wedge (C \leq B), P); P. \end{array} \quad (1)$$

This is done by first interacting with the user in order to find (and adopt) his/her goals, by asking what kind of computer the user is interested in, by checking if the user has some of the needed components (*get_user_preferences*), and by getting information about budget limitations (*get_max_value_budget*). Second, it *plans* how to reach the goals, predicting also future interactions with the user. Planning is needed to find configurations by taking into accounts two interacting goals: the goal to assemble a computer satisfying the user needs and the goal to consider only configuration affordable by the user's

budget. Observe that the metapredicate *plan* in (1) corresponds to the query $\langle assemble \rangle (credit(C) \wedge budget(B) \wedge (C \leq B))$ in the modal logical language. Finally, the agent *executes* the conditional plan P resulted from the planning process.

The way the agent assembles a computer is specified by procedure *assemble* that, until the computer is believed assembled tries to achieve the goal of getting a still missing component.

$assemble$ **is** $assembled?.$
$assemble$ **is** $\neg assembled?; achieve\_goal; assemble.$

Note that only when all of the goals to get the necessary components are fulfilled, the main goal to have a computer to propose to the user is reached and the computer is considered assembled. Until there is still a goal to fulfill, the computer is considered not assembled (it is expressed by the causal rule: $\neg assembled$ **if** $goal(X)$).

We assume the behaviour of a rational agent to be *driven by a set of goals*, which are represented as fluents having form $goal(F)$. The system detects the goals based on user's inputs and its expert competence about computer configurations. Initially the reasoner does not have explicit goals, because no interaction with the user has been performed. The user's inputs are obtained after a first interaction phase (see (1)) and they generate a set of goals that the agent has to achieve to assemble the requested computer. In the language, we model this by means of *causal rules*, by describing the adoption of a goal as the indirect effect of requesting user's preferences. For instance, in $get\_user\_preference$ the first suggesting action, $offer\_computer\_type$, asks what kind of computer the user needs. This action has as an indirect effect the generation of the goal to have a computer having those characteristics:

$offer\_computer\_type$ **suggests** $requested(X)$
$goal(has(X))$ **if** $requested(X).$

Let us suppose the agent has been requested to assemble a computer for multimedia (the fluent $requested(computer(multimedia))$ is in the state), then, the causal rule above will generate the goal $goal(has(computer(multimedia)))$. This main goal will generate a set of sub-goals to get the needed components to built the requested computer by means of the appropriate instantiation of the following causal rule:

$goal(has(C))$ **if** $goal(has(computer(X)) \wedge component(computer(X), C)$

After adopting a goal $goal(F)$, an agent acts so to achieve it until it believes the goal is fulfilled (i.e. until it reaches a state where F holds). This corresponds to adopt a *blind commitment strategy*.

We can now get into the details of procedure *achieve_goal*, which allows the agent to select in a non-deterministic way the goal of adding a component (monitor, CPU, RAM or peripheral) to the specific computer that is being built.

When the agent has the goal to get a *generic* component, it has to choose among the available types, so it interacts again with the user to decide what specific component to add according to the user's preferences:

$$
\begin{array}{lll}
achieve\_goal & \textbf{is} & goal(has(monitor(generic)))?; offer\_monitor\_type; \\
 & & type\_monitor(X)?; add(monitor(X)). \\
achieve\_goal & \textbf{is} & goal(has(monitor(X)))?; (X \neq generic)?; add(monitor(X)). \\
achieve\_goal & \textbf{is} & goal(has(ram(generic)))?; offer\_ram\_type; \\
 & & type\_ram(X)?; add(ram(X)). \\
achieve\_goal & \textbf{is} & goal(has(ram(X)))?; (X \neq generic)?; add(ram(X)). \\
\ldots & & \\
\end{array}
$$

Note that the above formulation of the behaviour of the agent, has many similarities with agent programming languages based on the BDI paradigm such as dMARS [13]. As in dMARS, plans are triggered by goals and are expressed as sequences of primitive actions, tests or goals.

5 Conclusions and Future Work

In this paper we have presented a new perspective on interface adaptation by tackling the problem of the construction of adaptive web sites based on the user's intentions. This approach is orthogonal to the classical approach of focusing on the user model and it is our opinion that a real adaptive system should encompass both these aspects. We have shown how logic programming languages (and, in particular, DyLOG) can be used for this kind of application, that is to define the behavior of an agent that builds the web site on demand, according to the needs of each of its clients. We think that our approach to adaptation could have other interesting applications in the construction of automatic guides for virtual museums and help-on-line systems.

The work that we have presented is in progress. We are currently extending WLog by introducing a new agent, whose task is to interact with a Data Base that contains all the factual information about the domain, and that serves as an interface between such a Data Base and both the reasoners and the executors. We are also extending the logical framework in order to model the communication among the agents in the system in DyLOG itself. Indeed, a declarative specification of the communication would allow to prove correctness properties of the interaction among the agents. One last extension that we mean to study is to tackle the different kinds of failure that can occur during the interaction between a user and the system; in particular, the possibility that the user has to refuse the system's proposals. In these cases replanning should occur.

Acknowledgements The authors would like to thank prof. Alberto Martelli for his precious help and support.

References

[1] L. Ardissono and A. Goy. Tailoring the interaction with users in electronic shops. In *Proc. of the 7th International Conference on User Modeling*, 1999.

[2] L. Ardissono, A. Goy, R. Meo, G. Petrone, L. Console, L. Lesmo, C. Simone, and P. Torasso. A configurable system for the construction of virtual stores. *World Wide Web*, 2(3):143–159, 1999.

[3] M. Baldoni, L. Giordano, and A. Martelli. A tableau calculus for multimodal logics and some (un)decidability results. In H. de Swart, editor, *Proc. TABLEAUX'98*, volume 1397 of *LNAI*, pages 44–59, 1998.

[4] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming. In J. Dix, L. M. Pereira, and T. C. Przymusinski, editors, *Proc. of NMELP'96*, volume 1216 of *LNAI*, pages 132–150. Springer-Verlag, 1997.

[5] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. A modal programming language for representing complex actions. In L. Giordano A. Bonner, B. Freitag, editor, *Proc. of the Post-Conference Workshop on Transactions and Change in Logic Databases*, DYNAMICS'98, pages 1–15, 1998.

[6] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Modeling agents in a logic action language. In *Proc. of the Workshop on Rational Agents, FAPR'00*, London, September 2000. To appear.

[7] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Reasoning about complex actions with incomplete knowledge: a modal approach. Technical Report 53/00, Dipartimento di Informatica, University of Torino, 2000.

[8] P. Bretier and D. Sadek. A rational agent as the kernel of a cooperative spoken dialogue system: implementing a logical theory of interaction. In Proc. of *ATAL III*, LNAI, 1997.

[9] B. De Carolis, F. de Rosis, D. Berry, and I. Michas. Evaluating plan-based hypermedia generation. In *Proc. of European Workshop on Natural Language Generation*, Toulouse, 1999.

[10] B.N. De Carolis. Introducing reactivity in adaptive hypertext generation. In *Proc. 13th Conf. ECAI'98*, Brighton, UK, 1998.

[11] M. Castilho, O. Gasquet, and A. Herzig. Modal tableaux for reasoning about actions and plans. In S. Steel, editor, *Proc. of European Conference on Planning (ECP'97)*, LNAI, pages 119–130. Springer-Verlag, 1997.

[12] G. De Giacomo and M. Lenzerini. PDL-based framework for reasoning about actions. In *Topics of Artificial Intelligence, AI*IA '95*, volume 992 of *LNAI*, pages 103–114. Springer-Verlag, 1995.

[13] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dmars. In *Proc. ATAL'97*, volume 1365 of *LNAI*, pages 155–176, 1997.

[14] L. Giordano, A. Martelli, and C. Schwind. Dealing with concurrent actions in modal action logic. In *Proc. of ECAI'98*, pages 537–541, 1998.

[15] L. Giordano, A. Martelli, and C. Schwind. Ramification and causality in a modal action logic. *Journal of Logic and Computation*, 2000. to appear.

[16] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, (31), 1997.

[17] D. Magro and P. Torasso. Description and configuration of complex technical products in a virtual store. In *Proc. of ECAI2000, Workshop on Configuration*, Berlin, 2000.

[18] M. McTear. User modelling for adaptive computer systems: a survey on recent developments. *Artificial Intelligence Review*, 7:157–184, 1993.
[19] H. Prendinger and G. Schurz. Reasoning about action and change. a dynamic logic approach. *Journal of Logic, Language, and Information*, 5(2):209–245, 1996.
[20] W. Wahlster and A. Kobsa. *User models in dialog systems*. Springer-Verlag, 1989.