Intention-guided Web Sites: A New Perspective on Adaptation

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

Dipartimento di Informatica – Università degli Studi di Torino
C.so Svizzera, 185 – I-10149 Torino (Italy)
Tel. +39 011 670 6711, Fax. +39 011 751603
E-mail: {baldoni,baroglio,patti,mrt}@di.unito.it
URL: http://www.di.unito.it/~alice

Abstract. Recent years witnessed a rapid growth of multimedia technologies for offering information and services on the internet. One of the many problems that are to be faced in this context is the great variety of possible users and the consequent need to adapt both the presentation of information and the interaction to the specific user's characteristics.
There is a general agreement that an adaptive web system should keep a model of the user's intentions, interests, and preferences, nevertheless, most of the research on adaptive web systems is based on the idea of associating each user with a "user model" and to dynamically build web pages, based on the profile given by that model. However, for the sake of a more complete adaptation -especially at the navigation level- it seems necessary to give a greater importance to the user's intentions both at the beginning of and during the interaction with the system. In our view this aspect of adaptation is currently underestimated. For achieving a dynamical site generation which is guided by the user's intentions, the system should keep track of such intentions and of their evolution during the interaction, and it should use them during its reasoning, when it plans dynamically personalized navigation routes. This is the topic of our current research, whose main lines are presented in this article.

1. INTRODUCTION

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 the distribution of services are just some examples. One serious problem due to the variety of the users is to find a way for adapting the presentation to the person who requested it. The most advanced solutions [Wahlster 89, McTear 93, Ardissono and Goy 99, De Carolis 98, De Carolis 99] start from the assumption that adaptation should focus on the user's own characteristics, thus, though in different ways, they all try to associate the user with a reference prototype (also known as the "user model"), and then adapt the presentation to such a prototype. The association 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 [Ardissono *et al.* 99b], a hybrid solution is used 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 "intentions" (or goals). In many systems the user is invited to write information about him/herself (education, job, etc.), which allows the system to build a user model and adapt the presentation to it but the user model lacks a component devoted to the representation of the user's intentions, which change at every connection. Even in the cases when the system builds (or refines) the model based on the choices that are taken in the various connections, it is not possible to have a complete adaptation. Let us suppose, as an 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 in some 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 both have difficulties in getting the information I'm interested in and have it presented at a too shallow level of detail, with respect to my current and real interest. If the system I interact with associated a model of my *intentions* with one of the user, this kind of inconvenient would not happen because the system would first of all try to capture my goal for the current connection (it would not limit its reasoning to my generic interests only) and only after it would be able to adapt the presentation to me.

Our research aims at addressing this deficit, by building a web system that, during the interaction with the user, adopts the user's goals in order to achieve a more complete adaptation.One fundamental characteristic of this approach is that the two actors of the interaction, the user and the system, “cooperate” to the solution of a problem which for them is now a “common problem”. In a way, it is as the system extended the user's competence with its own, which is related to the application domain. More technically, we study the implementation of web sites which are “structureless”, depending their shape on the *purposes* that the single users have. In different words, the structure of the web site depends on the interaction between the user and a server-side agent and such an interaction is finalized to satisfy a user's need or, better, his/her “intentions”. When a user connects to a site of this kind (s)he does not access to a fixed graph of pages and links but interacts with a program (it will be a software agent) which, starting from a knowledge base specific to the site (a data base) 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) which are contained in a knowledge base. This approach is “orthogonal” to the one based on user models, which is already widely studied in the literature.

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 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 build-modify-update process of the web site. 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 there exists a program (the agent's program), which is likely to be changed very rarely, since most of 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, in a way similar to what is normally done with programs.

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.

2. APPROACH AND CASE STUDY

We are currently tackling the construction of structereless intention-guided web sites, by exploiting, as a case study, 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 the 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 the names of processors, motherboards, buses, and so forth. It will also allow comparison with the literature because this domain has been used in other works, such as [Magro and Torasso, 00].

Often the client only knows 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), a dialogue in which the latter tries to understand the functionalities 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 by the seller to converg towards the optimal proposal.

In the case of on-line purchase, it is reasonable to offer a similar interaction. More formally, the seller and the client have a common goal: to build an object, in the specific case 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. The seller usually leads the pursuing process, by applying a plan aimed at selling. The goal of such a plan is to find a configuration which satisfies all the constraints. 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 the cases; on the contrary it is better to build the plan depending on the current conditions.

The virtual seller is implemented as a *software agent*, which can be seen as a program that creates the site at the moment, based both on an application-domain knowledge base and on the user's intentions. Formalisms for representing and reasoning on intentions have been studied in the field of Natural Language, with the aim of building cooperative and automatic dialogue systems, which recognize the intentions of the human interlocutor and use them as a guide for answering in an appropriate way (see, for instance [Bretier and Sadek, 97]).

Our focus 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 only for dialog act planning, but also for building web site presentation plans guided by the user's goal. In this perspective the structure of the site is build 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.

3. THE AGENT PROGRAMMING LANGUAGE

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 such as *beliefs*, *intentions*, and *desires* [Rao and Georgeff, 91]. 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.
Reasoning about the effect of actions in a dynamically changing world is one of the main problems that must be faced by intelligent agents. It is true, even if 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 execution of actions on the mental state.
In this section, we briefly describe DyLOG, a language for programming agents based on a logical theory for reasoning about actions and change in a logic programming setting, referring to the proposal in [Baldoni *et al.,* 97], and its extension to deal with complex actions and knowledge-producing actions [Baldoni *et al.,* 98a, Baldoni *et al.,* 00]. We will also show how to use it in order to develop web applications that require planning capabilities. As an example, we will specify our virtual seller as a software agent.
DyLOG allows one to specify an agent's behavior by defining both a set of simple actions that the agent can apply (they are defined in terms of their preconditions and effects) and a set of complex actions (procedures), built upon simple ones. However, the fundamental characteristic that makes this language particularly interesting for agent programming is that, being based on a formal theory of actions, it can deal with reasoning about actions' effects in a dynamically changing environment and, as such, it supports planning. We will better explain this point in Section 3.4.
The language is based on a *modal action theory* that has been developed in [Giordano *et al.,* 00, Baldoni *et al.,* 97, Baldoni *et al.,* 98a, Baldoni *et al.,* 00]. 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. 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 [Castilho *et al.* 97, De Giacomo and Lenzerini 95, Giordano *et al.* 98, Prendinger and Schurz 96, Schwind 97], 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.
The DyLOG interpreter (available at the address `http://www.di.unito.it/~alice`) has been implemented in Sicstus Prolog, thus a specification in DyLOG can be executed by this interpreter. Such an interpreter is a straightforward implementation of the proof procedure of the language. In this paper we use DyLOG as a language for building web applications which require planning capabilities. In particular our aim is to use the ability of a cognitive agent to autonomously reason on its own behavior in order to deal with *adaptivity at the navigation level*, i.e. to dynamically generate a site being guided by the user's goals that are explicitly adopted through the interaction. Indeed, 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 dynamic situation of the interaction with the user.

3.1. Primitive actions

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 $<a>\alpha$ 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 true 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 [Baldoni *et al.*, 00] we introduce an epistemic level in our representation language. In particular, we introduce an epistemic operator *B*, to represent the beliefs an agent has on the world: Bf will mean that the fluent f is known to be true, $B\neg f$ will mean that the fluent f is known to be false. Fluent f is undefined in the case both $\neg Bf$ and $\neg B\neg f$ hold. We will write $u(f)$ for $\neg Bf \wedge \neg B\neg f$. In our implementation of DyLOG (and also in the following) we do not explicitly use the epistemic operator *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 the fluent f is unknown (i.e. to test if neither f nor $\neg f$ is present in the state).

Simple action laws are rules that allow 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, a 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 $\Box(Fs \rightarrow <a>true)$, 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*".

Let us consider the example of the seller fully described in Section 3.5. 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*.

[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 [Giordano *et al*., 00].

(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. Analogously, we define the other seller's primitive actions of adding to the configuration a CPU, a RAM, or a peripheral.

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.

In general, the execution of an action will have an effect on the environment, such as moving a robot or, in our case, showing a web page to the user. This can be specified in DyLOG by associating with each primitive action some Prolog code which implements the effects of the action on the world (in our case the web server is requested to send a given web page to the browser). Therefore, when an agent executes an action it must commit to it, and it is *not allowed to backtrack* by retracting the effects of the action. This is the main difference with respect to the use of the language for reasoning about actions, where the agent can do hypothetical reasoning on possible sequences of actions by exploring different alternatives.

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 fact, in [3] we studied how to represent 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*. Direct effects of *sensing actions* are interpreted as inputs from outside that are not under the agent control. They are represented using *knowledge laws*, that have form "*s* **senses** *F*", meaning that action *s* causes to know whether *F* holds.[2] In our application domain we are interested in inputs coming from the user. Then, getting a user input, can be seen as the result of sensing actions executed by the agent, that, after requesting the user 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), senses the user's answer. Preconditions and indirect effects of sensing actions are expressed by precondition laws and causal rules, having the form defined above. In our running example, for instance, we introduce the 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(has(monitor(M)))*.

ask_if_has_monitor(M) **senses** *has(monitor(M))*.

Specifically for this application domain, we have also defined a special subset of sensing actions, called suggesting actions. The difference w.r.t. normal sensing actions is that while those offer as alternative values for a given fluent its whole domain, suggesting actions offer only a subset of it, that is those values that lead to fulfill the goals. 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:

[2] See [Baldoni *et al*., 00] for the translation of knowledge laws in the modal language.

offer_monitor_type **possible if** *true*.

offer_monitor_type **suggests** *type_monitor(X)*.

We use the results of the interaction with the user to generate the goals that will guide the selling agent behavior in order to build a computer satisfying the user's needs. We model it by causal rules, by describing the adoption of a goal as the indirect effect of requesting user's preferences. As an example, the answer the agent gets from the suggesting action *offer_computer_type* about the kind of computer that is requested (see Section 3.5 for a description), has as an indirect effect the generation of the goal to have a computer with those characteristics:

goal(has(X)) **if** *requested(X)*.

Let us suppose that the agent has to assemble a computer for multimedia (*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(X),C)*.

3.3. Procedures

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

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

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 $<p_1><p_2> \ldots <p_n>\varphi \supset <p_0>\varphi$.[4] 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.

Procedures can be used to describe the behaviour of an agent. In particular we assume the behaviour of a rational agent to be *driven by a set of goals*. For each goal, the agent has a set of procedures (sometimes called plans) for achieving the given goal.

The behaviour of our selling agent can be described by giving a collection of procedures for guiding the interaction with the user and for achieving various goals. The agent behaviour is driven by goals, which are fluents having form *goal(F)*. They are generated by the interaction with the user by means of causal rules. 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*.

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*?.

---

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

[4] These axioms have the form of rewriting rules as in grammar logics. In [Baldoni *et al.*, 98] decidability results for subclasses of grammar logics are provided, and right regular grammar logics are proved to be decidable.

assemble **is** *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 (see in Section 3.5 causal rules (5),(6),(7) and (11)). Until there is still a goal to fulfill, the computer is considered not assembled.

Procedure *achieve_goal* 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.

achieve_goal **is** *goal(has(monitor(generic)))*?; *offer_monitor_type*; *type_monitor(X)*?; *add(monitor(X))*.

achieve_goal **is** *goal(has(monitor(X)))*?; (*X* ≠ *generic*); *add(monitor(X))*.

achieve_goal **is** *goal(has(ram(generic)))*?; *offer_ram_type*; *type_ram(X)*?; *add(ram(X))*.

achieve_goal **is** *goal(has(ram(X)))*?; (*X* ≠ *generic*); *add(ram(X))*.

…

As pointed out before, whenever the interpreter executes an action, it commits to that action, and cannot backtrack. Thus procedures are deterministic, or, at most, they can implement a kind of don't care non determinism.

The above formulation of the behaviour of the agent, has many similarities with agent programming languages based on the BDI paradigm such as dMARS [d'Inverno *et al.*, 97]. As in dMARS, plans are triggered by goals and are expressed as sequences of primitive actions, tests or goals.

3.4. Planning

Up to now we have assumed that DyLOG procedures which specify the behaviour of an agent are executed by an interpreter as in usual programming languages. 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.

In general, a *planning problem* amounts to determine, given an initial state and a goal *Fs*, if there is a sequence of actions that, when executed in the initial state, leads to a 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*. This can be formulated by the query *<p>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, …, a_m", i.e. a primitive action sequence from the initial state to the final one, which represents a *linear plan*.

To achieve this, DyLOG provides a metapredicate *plan(p, Fs, as*), where *p* is a procedure, *Fs* a goal 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.

Instead of simply obtaining any plan, procedure *plan* could extract a *best plan*, according to some optimality criterion. In our case, *plan* uses iterative deepening to find the plan with the minimal number of actions, but it might be modified to make use of some domain dependent heuristics.
Since procedures can contain sensing 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.
The top level procedure, *build_a_computer*, describing the behavior of the selling agent, is reported hereafter. It makes use of the interleaving of planning and execution, by using the metapredicate *plan*.

build_a_computer **is** *get_user_preferences*; *get_max_value_budget*;
plan(assembly,credit(C) $\wedge$ *budget(B)* $\wedge$ *(C* $\leq$ *B), P)*; *P*.

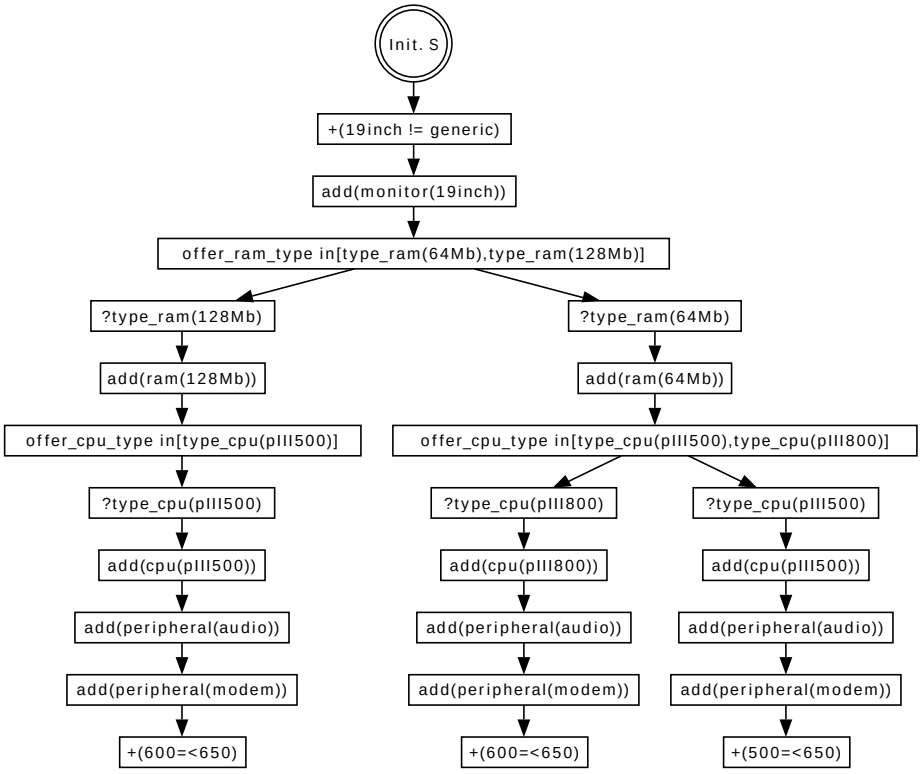

Figure 1 The result of the planning process when the user does not have any component and has a budget of 650 Euro.

First, the agent starts to interact with the user to adopt his goals, asking what kind of computer the user is interested in, checking if the user has some of the components that are needed to assemble a computer with those characteristics (*get_user_preferences*), getting information about budget limitations (*get_max_value_budget*). Second it starts *to plan* how reach the goals, i.e. how to assemble the missing components of the computer, predicting also future interactions to be guided by the users preferences in the assembling process. Planning is needed to find configurations 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. Finally, the agent executes the conditional plan resulted from the planning process. Let us assume to execute *build_a_computer* and to be in the case the user requested a computer for multimedia. In Figure 1 we show one possible result of the planning process, given specific user's needs and budget.

3.5. A virtual computer seller

Let us see, as an example, how it is possible to implement a program that creates a virtual computer seller. The example is, for our choice, simple and a computer is represented only in terms of its components: "cpu", "ram", "monitor" plus a few peripherals. It is possible to buy three different kinds of computer: one for CAD processing, one for multimedia and one for word processing. Each of them is characterized by a different configuration.

Finite domain specification

computer(X) **domain** *(X* **in** [*cad,multimedia,wordprocessing*]*).*
budget(X) **domain** (*X* **in** [*450,550,650,750,850,1000*]).
requested(X) **domain**
(*X* **in** [*computer(cad),computer(multimedia),computer(wordprocessig)*]*).*
type_monitor(X) **domain** (*X* **in** [*database:get_article(category(monitor),X*)]).
type_ram(X) **domain** (*X* **in** [*database:get_article(category(ram),X*)]).
type_cpu(X) **domain** (*X* **in** [*database:get_article(category(cpu),X*)]).
peripheral(X) **domain** (*X* **in** [*database:get_article(category(peripheral),X)*]).

Knowledge Base on application domain

component(computer(cad), peripheral(plotter)).
component(computer(cad), cpu('pIII800')).
component(computer(cad), monitor('19inch')).
component(computer(cad), ram('256Mb')).
component(computer(multimedia), peripheral(modem)).
component(computer(multimedia), peripheral(audio)).
component(computer(multimedia), monitor('19inch')).
component(computer(multimedia), ram(generic)).
component(computer(multimedia), cpu(generic)).
component(computer(cad), peripheral(printer)).
component(computer(cad), cpu(generic)).
component(computer(cad), monitor(generic)).
component(computer(cad), ram(generic)).

Main procedure The behavior of our selling agent is described at the high-level by the main procedure, reported hereafter.

build_a_computer **is** *get_user_preferences;get_max_value_budget;*
plan(assemble, credit(C) ∧ budget(B) ∧ (C ≤ B),P);P.
get_user_preferences **is** *ask_computer_type; goal(has(ram(T))?; ask_has_ram(T);*
goal(has(monitor(T))?; ask_has_monitor(T);
goal(has(ram(T))?; ask_has_ram(T).

Suggesting actions The following suggesting actions allow to acquire information about the user's needs and preferences: about the budget limitation the user has, the kind of computer

the user would like to buy and about the kind of components the user prefers among the available ones.

get_max_value_budget: offer a set of maximum budget and get the value of budget(B)
get_max_value_budget **possible_if** *true.*
get_max_value_budget **suggests** *budget(B).*
offer_computer_type: offer the available kinds of computer and get the preference about the requested computer
offer_computer_type **possible_if** *true.*
offer_computer_type **suggests** *requested(X).*
offer_monitor_type: offer the available monitors and get the preference about the monitor type
offer_monitor_type **possible_if** *true.*
offer_monitor_type **suggests** *type_monitor(X).*
offer_ram_type: offer the available RAM's and get the preference about the ram type
offer_ram_type **possible_if** *true.*
offer_ram_type **suggests** *type_ram(X).*
offer_cpu_type: offer the available CPU's and get the preference about the monitor type
offer_cpu_type **possible_if** *true.*
offer_cpu_type **suggests** *type_cpu(X).*

Sensing actions The following sensing actions allow the program to know whether the user already has some of the components the seller thinks to be necessary to assemble a computer satisfying the user's needs. The answer to these queries allows to revise the set of goals the seller wants to achieve for satisfying the user, taking in account new information about the user needs. In fact, in case of positive answer, the seller will update its goals, deleting, if necessary, the goal to “get the component” that the user already has.

ask_if_has_monitor(T): ask if the user has a monitor of type *T*
ask_if_has_monitor(T) **possible_if** *u(has(monitor(T))).*
ask_if_has_monitor(T) **senses** *has(monitor(T)).*
ask_if_has_cpu(T): ask if the user has a cpu of type *T*
ask_if_has_cpu(T) **possible_if** *u(has(cpu(T))).*
ask_if_has_cpu(T) **senses** *has(cpu(T)).*
ask_if_has_ram(T): ask if the user has a ram of type *T*
ask_if_has_ram(T) **possible_if** *u(has(ram(T))).*
ask_if_has_ram(T) **senses** *has(ram(T)).*

Causal rules The knowledge about the computer obtained by the first interaction generates a main goal of the form *goal(has(computer(T))),* (clauses (1), (2)). The main goal generates a set of sub-goals for getting the needed components (clause 3). A goal *goal(F)* is abandoned only when the agent reaches a state where *F* is believed (clause 4). When all the sub-goals to get the necessary components are fulfilled, also the main goal to have a computer to propose to the user is reached (clauses from (5) to (7)) and the computer is considered assembled (clause 11). Until there is still a goal to fulfill, the computer is considered not assembled (clause 12).

(1) *goal(has(X)))* **if** *requested(X)* $\wedge$ *u(has(computer(X))).*
(2) *goal(has(X)))* **if** *requested(X)* $\wedge$ $\neg$*has(computer(X)).*

(3) *goal(has(C))* **if** *goal(has(computer(X))* $\wedge$ *component(computer(X),C).*
(4) $\neg$*goal(has(X))* **if** *has(X).*
(5) *has(computer(cad))* **if** *has(peripheral(plotter))* $\wedge$ *has(monitor('19inch'))* $\wedge$
has(ram('256Mb')) $\wedge$ *has(cpu('pIII800')).*
(6) *has(computer(multimedia))* **if** *has(peripheral(modem))* $\wedge$ *has(peripheral(audio))* $\wedge$
has(monitor('19inch')) $\wedge$ *has(ram(generic))* $\wedge$
has(cpu(generic)).
(7) *has(computer(wordprocessing))* **if** *has(peripheral(printer))* $\wedge$ *has(monitor(generic))* $\wedge$
has(ram(generic)) $\wedge$ *has(cpu(generic)).*
(8) *has(cpu(generic))* **if** *has(cpu(X)).*
(9) *has(ram(generic))* **if** *has(ram(X)).*
(10) *has(monitor(generic))* **if** *has(monitor(X)).*
(11) *assembled* **if** *has(computer(X)).*
(12) $\neg$*assembled* **if** *goal(X).*

Achieving goals The way the agent behaves to assemble a computer is specified by the procedure *assemble* that, until the computer is believed assembled, tries to achieve the goal to get a still missing component.

assemble **is**	*assembled?.*
assemble **is**	$\neg$*assembled?; achieve_goal; assemble.*
achieve_goal **is**	*goal(has(monitor(generic)))?;offer_monitor_type ;* *type_monitor(X)?; add(monitor(X)).*
achieve_goal **is**	*goal(has(monitor(X)))?; (X* $\neq$ *generic)?;add(monitor(X)).*
achieve_goal **is**	*goal(has(ram(generic)))?; offer_ram_type;* *type_ram(X)?; add(ram(X)).*
achieve_goal **is**	*goal(has(ram(X)))?; (X* $\neq$ *generic)?; add(ram(X)).*
achieve_goal **is**	*goal(has(cpu(generic)))?; offer_cpu_type;* *type_cpu(X)?; add(cpu(X)).*
achieve_goal **is**	*goal(has(cpu(X)))?; (X* $\neq$ *generic)?;add(cpu(X)).*
achieve_goal **is**	*goal(has(peripheral(X)))?; add(peripheral(X)).*

Atomic actions The following simple action laws define the primitive actions that allows to add the single components (a monitor, a RAM a peripheral or a CPU) to the current configuration.

add(monitor(X)) **possible_if** *true.*
add(monitor(X)) **causes** *has(monitor(X))* **if** *true.*
add(monitor(X)) **causes** *credit(B1)* **if** *get_value(X,price,P)* $\wedge$ *credit(B)* $\wedge$
(B1 is B + P).
add(ram(X)) **possible_if** *true.*
add(ram(X)) **causes** *has(ram(X))* **if** *true.*
add(ram(X)) **causes** *credit(B1)* **if** *get_value(X,price,P)* $\wedge$ *credit(B)* $\wedge$
(B1 is B + P).
add(cpu(X)) **possible_if** *true.*
add(cpu(X)) **causes** has(cpu(X)) **if** *true.*
add(cpu(X)) **causes** *credit(B1)* **if** *get_value(X,price,P)* $\wedge$ *credit(B)* $\wedge$
(B1 is B + P).

add(peripheral(X)) **possible_if** *true.*
add(peripheral(X)) **causes** *has(peripheral(X))* **if** *true.*
add(peripheral(X)) **causes** *credit(B1)* **if** *get_value(X,price,P)* $\wedge$ *credit(B)* $\wedge$ *(B1 is B + P).*
add(C) **causes** *in_the_shopping_cart(C)* **if** *true.*

4. IMPLEMENTATION (WORK IN PROGRESS)

In this section we will describe the virtual seller as we are currently implementing it in order to work in a client-server environment. We suppose that clients are commercial web browsers.
As we said, the basic idea is to consider the web site as a plan for satisfying the user's intentions. Actually the plan will be a *conditional plan*, i.e. there will be nodes in which a set of alternative actions are possible. Each path will, in our case, correspond to a different computer configuration. All configurations in the conditional plan satisfy the user's intentions. The plan is built by the agent program, which, in the case of our computer seller is currently the one reported in Section 3.
After it was built, the plan is executed. *Executing* a conditional plan implies following one of the paths in the tree; only the part of the site corresponding to this path will actually be built. The execution of some of the actions in the path will affect only the data base that contains the information about the components that are currently contained in the store. For instance, if a certain processor was selected the agent has to decrease the number of available processors of that kind. Other actions, those that correspond to a branching point, require a dialogue with the user.
Actually, we handle a more generale situation in which many clients can connect to the server and start different, parallel purchases. Each client will be served by a dedicated selling agent. Given a client, the selling-agent responsible for that client builds the conditional plan starting from the client's expressed intentions. The nodes of the plan will correspond to the subsequent, foreseen alternative states and possible actions.
We have said that the interaction starts with the expression of the client's intentions, that is what the client is looking for (what it needs a computer for). Intentions can either belong to a set of alternatives that was defined a priori (such as “internet browsing”) or they can be composed as a DyLOG query on the web browser, and are sent to the web server (in our case Apache) of the server machine. The web server properly dispatches the requests to the right DyLOG selling agent running in a Sicstus Prolog environment by passing through a java servlet, that works as an interface. We used java servlets because it was not possible to directly embed Sicstus Prolog into the web server in such a way that sessions could be left open with the browsers. Our problem was that, as we have partly mentioned and we will better see further on, the agent does not simply send the result of its computation (the whole conditional plan) to the client but it sends step by step partial results of its computation, i.e. the single pages of the site, because the choice of which part of the site to build and explore is actually up to the user. Furthermore, we had to manage the multi-user situation in which many clients are parallely carrying on many conversations with as many prolog agents.
Let us now consider actions and, in particular, their execution. 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 choose is up to the user, so (s)he will be shown an HTML page containing the possible alternatives. For each item an auxiliary page containing its technical characteristics will also be produced, so that the user can be made aware of the details that can help him/her to make his/her mind. Such information is taken by the servlet from the

component data base. In different words, the agent sends to the servlet a command of the kind "visualize CPU1, CPU2" and the servlet produces some HTML code that contains the information related to the two CPU's identified by CPU1 and CPU2 in the data base plus the request to make a choice. Once an answer is returned from the client, this is passed on to the agent. So the execution of the agent's action consists of the following steps: 1. produce HTML code related to the choice; 2. show such a code to the user; 3. wait for feedback from the user. Afterwards the agent 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 servlet the information about which page to show next. All unselected alternatives are forgotten.

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's model and it is our opinion that a real adaptive system should encompass both these aspects. We have shown how modal logic programming languages can be used to define the behavior of an agent that builds the web site on demand, according to the needs of each client.
The work that we have presented is in progress. It started as an application of DyLOG as an agent programming language, and it is actually the core of a wider project in which the agent will be supplied also with replanning capabilities. Replanning may seem useless in a system that returns a plan only if it does not fail. However, one must take into account also the fact that the user may be unable to fully express its intentions or it may become aware during the dialogue of some consequences of its initials choices that (s)he would like to change. In these cases the client should be able to alert the system that (s)he is not interested in the options supplied by it and that maybe new facts should be taken into account.

REFERENCES

[Ardissono and Goy, 99a] L. Ardissono, A. Goy, Tailoring the interaction with users in electronic shops, in the *7th International Conference on User Modeling*, 1999.

[Ardissono *et al.*, 99b]L. Ardissono, A. Goy, R. Meo, G. Petrone, L. Console, L. Lesmo, C. Simone, P. Torasso, A configurable system for the construction of virtual stores, *World Wide Web Journal*, 2(3), 1999, pp. 143-159.

[Baldoni *et al.*, 97] M. Baldoni, L. Giordano, A. Martelli, V. Patti, An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming, in the *2nd International Workshop on Non-Monotonic Extensions of Logic Programming*, NMELP'96, J. Dix and L. M. Pereira and T. C. Przymusinski (eds), LNAI 1216, Springer-Verlag, 1997, pp. 132-150.

[Baldoni *et al.*, 98a] M. Baldoni, L. Giordano, A. Martelli, V. Patti, A Modal Programming Language for Representing Complex Actions, in the *Post-Conference Workshop on Transactions and Change in Logic Databases, DYNAMICS'98, Joint Int. Conference and Symposium on Logic Programming, IJCSLP'98,* A. Bonner, B. Freitag, L. Giordano (eds), Technical Report MPI-9808, 1998, pp. 1-15.

[Baldoni *et al.*, 98b] M. Baldoni, L. Giordano, A. Martelli, A Tableau Calculus for Multimodal Logics and Some (Un)Decidability Results, H. de Swart (ed), in *TABLEAUX'98*, LNAI-1397, 1998, pp. 44-59.

[Baldoni *et al.*, 00] M. Baldoni, L. Giordano, A. Martelli, V. Patti, Reasoning about Complex Actions with Incomplete Knowledge: a Modal Approach. Technical Report 53/2000, Dipartimento di Informatica, University of Torino, 2000.

[Bretier and Sadek, 97] P. Bretier, D. Sadek, A rational agent as the kernel of a cooperative spoken dialogue system: implementing a logical theory of interaction, in *ECAI-96 Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, LNAI series, Intelligent Agents III, 1997.

[Castilho *et al.*, 97] M. Castilho, O. Gasquet, A. Herzig, Modal tableaux for reasoning about actions and plans, in *European Conference on Planning (ECP'97)*, S. Steel (ed), LNAI, Springer-Verlag, pp. 119-130, 1997.

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

[De Carolis *et al.*, 99] B. De Carolis, F. de Rosis, D. Berry, I. Michas, Evaluating plan-based hypermedia generation, in *European Workshop on Natural Language Generation*, Toulouse, 1999.

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

[d'Inverno *et al.*, 97] M. d'Inverno, D. Kinny, M. Luck, M. Wooldridge, A Formal Specification of dMARS , in *ATAL'97*, LNAI-1365, 1997, pp. 155-176.

[Giordano *et al.*, 98] L. Giordano, A. Martelli, C. Schwind, Dealing with concurrent actions in modal action logic, in *ECAI'98*, 1998, pp. 537-541.

[Giordano *et al.*, 00] L. Giordano, A. Martelli, C. Schwind, Ramification and causality in a modal action logic, *Journal of Logic and Computation*, 2000, to appear.

[Magro and Torasso, 00] D. Magro, P. Torasso, Description and configuration of complex technical products in a virtual store, in *ECAI2000 Workshop on Configuration*, Berlin, 2000.

[McTear, 93] M. McTear, User modelling for adaptive computer systems: a survey on recent developments, *Journal of Artificial Intelligence Review*, 7, 1993, pp. 157-184.

[Prendinger and Schurz, 96] H. Prendinger, G. Schurz, Reasoning about action and change. a dynamic logic approach, *Journal of Logic, Language, and Information*, 5(2), 1996, pp. 209-245.

[Rao and Georgeff, 91] A.S. Rao, M.P. Georgeff, Modeling rational agents within a BDI-architecture, in *KR-91*, Morgan Kaufmann, 1991, pp. 473-484.

[Schwind, 97] C. B. Schwind, A logic based framework for action theories, *Language, Logic and Computation*, J. Ginzburg, Z. Khasidashvili, C. Vogel, J.-J. Levy, E. Vallduv (eds), CSLI publication, Stanford, USA, 1997, pp. 275-291.

[Wahlster and Kobsa, 89] W. Wahlster, A. Kobsa, User models in dialog systems, Springer-Verlag, 1989.