Artificial Intelligence Review **22:** 3–39, 2004.

Web-Based Adaptive Tutoring: An Approach Based on Logic Agents and Reasoning about Actions

MATTEO BALDONI, CRISTINA BAROGLIO and VIVIANA PATTI
Dipartimento di Informatica, Università degli Studi di Torino, C.so Svizzera, 185, I-10149 Torino, Italy (E-mails: {baldoni; baroglio; patti}@di.unito.it)

Abstract. In this paper we describe an approach to the construction of adaptive tutoring systems, based on techniques from the research area of Reasoning about Actions and Change. This approach leads to the implementation of a prototype system, having a multi-agent architecture, whose kernel is a set of rational agents, programmed in the logic programming language DyLOG. In the prototype that we implemented the reasoning capabilities of the agents are exploited both to dynamically build study plans and to verify the correctness of user-given study plans with respect to the competence that the user wants to acquire.

Keywords: adaptive systems, curriculum sequencing, curricula validation, logic programming, multiagent systems, reasoning about actions, web-based tutoring

1. Introduction

This work investigates the use of an agent logic language, based on reasoning about action effects, for performing adaptive tutoring tasks in the context of a Web-based educational application. Adaptation in Web-based educational systems is a hot research topic attracting greater and greater attention (Brusilovsky, 2001), especially after the spreading of web technologies. Many research teams have implemented Web-based courseware and other educational applications based on different, adaptive and intelligent technologies, with the common goal of using knowledge about the domain, about the student and about the teaching strategies in order to support flexible, personalized learning and tutoring (Weber and Brusilovsky 2001; Henze and Nejdl 2001; Carro et al. 1999; Macías and Castells 2001).

The problem that we faced is to provide personalized support to users (students, in our case) in the definition process of *study plans*, a study plan being a sequence of courses that the student should attend. The idea that we explored is to base adaptation on the *reasoning capabilities* of a *rational agent*, built by means of a logic language. In particular, we focused on the possible uses of three different reasoning techniques, namely *planning*, *temporal projection*, and *temporal explanation*, which have been developed for allowing software agents to build action plans and to verify whether some

properties of interest hold after the application of given sequences of actions. In both cases actions are – usually – not executed in the real world but their execution is simulated "in the mind" of the system, which has to foresee their effects (or their enabling causes) in order to build solutions. In our application framework, a group of agents, called *reasoners*, works on a dynamic domain description, where the basic actions that can be executed are of the kind "attend course X" and where also complex professional expertise can be described. Effects and conditions of actions (courses) are essentially given in terms of a set of abstract *competences*, which are connected by causal relationships. The set of all the possible competences and of their relations defines an *ontology*. This multi-level description of the domain bears along many advantages. On the one hand, the high modularity that this approach to knowledge description manifests allows course descriptions as well as expertise descriptions to be added, deleted or modified, without affecting the system behavior. On the other hand, working at the level of competences is close to human intuition and enables the application of both goal-directed reasoning processes and explanation mechanisms.

The reasoning process that supports the *definition* of a *study plan*, aimed at reaching a certain learning goal, either computes over the effects of attending courses (given in terms of competence acquisition, credit gaining, and the like) or over those conditions that make the attendance of a course reasonable from the educator point of view. The logic approach also enables the *validation* of student-given study plans with respect to some learning goal of interest to the student himself. Basically, the reasons for which a study plan may be wrong are two: either the course sequence is not correct or by attending that plan the student will not acquire the desired competence. In both cases our reasoning mechanism allows the system to detect the weak points in the plan proposed by the student and to return a precious feedback.

Summarizing, the adaptive services currently supplied by our rational agents are: study plan construction, student-given study plan validation, and, in case of validation failure, explanation of the reasons for which a student-given plan is not correct. Notice that our tutoring system is not required to monitor the students progress, its only target being study plan definition. In this perspective, our task resembles *curriculum sequencing* problems, where an "optimal reading sequence" through a hyper-space of information sources is to be found. We can consider in fact courses (or, at least, course descriptions) as information sources. The main difference with respect to many applicative domains, where curriculum sequencing is used, is that in our framework the whole path is to be constructed before the student begins to attend the courses. Therefore, we cannot apply methods that construct the solution one step at a time, always returning the *next best step*. We actually

propose a *multi-step methodology*, that builds *complete*, personalized study plans (see the conclusions for a deeper comparison with other curriculum sequencing techniques).

A key feature that allows an agent of the kind described above to *adapt* to each single user is its capability of tackling mental attitudes, such as *beliefs* and *intentions*. In fact, the agent can adopt a student's *learning goal* and find a way for achieving it, which fits that specific student's interest and takes into account the student's current knowledge. As we will see, the identified solution can, then, be further adapted by interacting with the user during the solution presentation phase. Intention, as well as belief and action, has intensively been studied in *logics* and in *logic programming* (Scherl and Levesque 1993; Gelfond and Lifschitz 1993). We used an agent logic programming language, called DyLOG (Baldoni et al. 2001b; Patti 2002), that is based on a *modal formal theory of actions* and allows reasoning about action effects in a dynamically changing environment. The language will be described in the following sections, while its setting in the logic programming literature is given in the conclusions.

Users interact with the system that we implemented, Wlog,[1] by means of a web browser. All the communication with the system is performed by means of a set of dynamically generated web pages, in a process that can be considered a very simple form of *conversation*. In this perspective, the use of mental attitudes could recall some works on cooperative dialogue systems in the field of Natural Language, and in particular (Bretier and Sadek 1997); however, there are some differences. Bretier and Sadek proposed a logic of rational interaction for implementing the dialogue management components of a spoken dialogue system. This work is based, like the DyLOG language, on dynamic logic; nevertheless, while they exploit the capability of reasoning on actions and intentions in order to produce proper *dialogue* acts, we use them to produce *solutions* to the users' problems.

The article is organized as follows. Section 2 describes the application framework and informally introduces some key elements of our work: the notion of course, the notion of competence, the tasks that we have tackled. Section 3 introduces the DyLOG language and explains how both the domain knowledge and knowledge about the student are represented; moreover, this section explains how courses can be interpreted as actions and the forms of interaction that the language supports. Section 4 explains our interpretation of adaptive tutoring services as "reasoning about actions" tasks. Three kinds of reasoning processes are introduced: temporal projection, temporal explanation, and planning. We will see how rational agents, that can perform these kinds of reasoning, can accomplish the tutoring tasks that are described in Section 2. In Section 5, we show how DyLOG can be used not only for

representing the domain knowledge but also for programming the reasoners that perform the various tasks. Finally, Section 6 describes the implemented system; conclusions follow.

2. The Virtual Tutor Domain

In Italian universities students must list the courses that they want to attend in their years as undergraduate students. Every year they have the possibility to change this list (called *study plan*) according to their recent experiences and personal taste. Study plans should be compiled according to given guidelines and their consistency is verified by university professors.

Both the definition and the validation of study plans are time-consuming, difficult tasks. For instance, students tend to be attracted by courses whose names recall graphics, multimedia, or the web, disregarding those that supply the necessary theoretical backgrounds. One of the main reasons of this behavior is that when asked to build a study plan, students tend to be attracted by those courses that are described by keywords, which can intuitively be associated to the professional expertise they are interested in (the student's *learning goal*). On the other hand, also study plan verification is a difficult and time-consuming task for professors, which requires knowledge about each student situation (which courses have been attended and passed, if and what the student studied abroad, and so forth) as well as knowledge about each course (prerequisites, topics that are taught) and the general guidelines that constrain the possible alternatives. In order to resolve the possible inconsistencies of a student-defined study plan, it is often necessary for students to meet a tutor and discuss with him/her both their intentions and some of their specific choices.

For all these reasons it would be very useful to have a *software assistant* that helps both students and professors in all of the different phases of plan construction and validation. Taking a look at the literature, study plan construction can be interpreted as a special case of curriculum, or page, sequencing. *Curriculum sequencing* is a well-known technique in the field of Adaptive Educational Hypermedia (AEH) (Stern and Woolf 1998; Brusilovsky 2000; Weber and Brusilovsky 2001; Henze and Nejdl 2001; Baldoni et al. 2002); it is commonly used in this field to personalize, in an intelligent way, the navigation of a student in a hyperspace of information sources. Although the granularity and the type of information source depend on the specific application domain and can widely vary (an information source may be a definition, a page, a web site), they are usually considered as being *atomic* and thus they are handled as single unstructured objects. In the special case of our application domain, study plan construction can be described

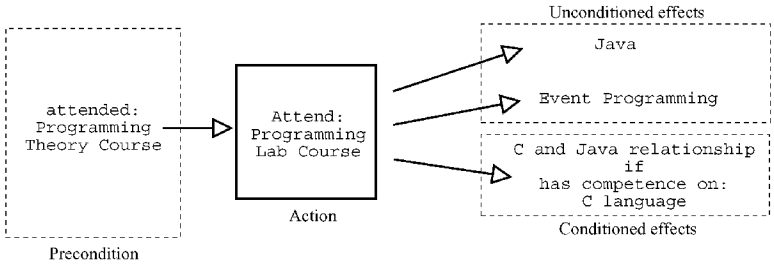

Figure 1. The action of attending the Programming Lab course.

Figure 2. As an example, this is a little excerpt from our competence-based knowledge model.

as a curriculum sequencing task where the atomic units of information that compose the hyperspace are *course descriptions*.

Our approach to study plan construction is based on the observation that it is quite natural to represent each course as an *action*: the action of attending the course. For instance, Figure 1 describes a Programming Lab course as the action of "attending a Programming Lab course", which can be executed if an action "attend a Programming Theory course" has already been executed. The execution of the action "attend a Programming Lab course" has as an effect the acquisition of knowledge about Java and Event programming. Part of an action effects can be subject to more specific conditions; in the example a student can understand comparisons with C programming if he already knows the C language; observe, however, that it is not necessary to know C for attending the course. As underlined by the example, in this perspective a course can be fully described by two sets of conditions: the conditions that are to be satisfied before the course can be attended and the conditions that ideally become true by attending it (i.e., the knowledge that the student is supposed to acquire by attending the course). Besides references to explicit courses, all preconditions and outcomes are given in terms of knowledge elements, that in our approach are described separately: we call them *competences*. Generally, competences are not atomic and can be seen as composed of smaller pieces of competence, related to one another. The set of all the pieces of competence and the set of their relations define an ontology, which is a *knowledge model* of the learning domain. The ontology also defines the vocabulary of the terms used to write the course descriptions which belong to our hyperspace of information sources; last but not least, competences are also used to describe the students' learning goals.

Knowledge models can be defined in different ways by describing the relationships among the various elements; our ontology is built upon a set of causal relations. Figure 2 shows an example: the two-level tree on the left represents the fact that having competence about "data structures", "algorithms", and "programming languages" causes to have competence about "programming". The rules on the right say that competence about "programming languages" can be achieved by (alternatively) acquiring competence either about the "C", the "Java", or the "Prolog" programming language.

In this framework, study plan construction can be interpreted as a *reasoning about actions* task: the task of finding a sequence of courses that, once attended, will allow a student to achieve his learning goal. Interpreting courses as actions, described in terms of a common vocabulary, has another advantage: it enables the application of other reasoning mechanisms that are very helpful in the construction of an AEH system. For example, it is possible to validate the correctness of a student-given plan in an automatic way or, as we will explain in the following sections, to build systems that to some extent "discuss" with a student explaining what is wrong in a submitted study plan.

3. Representing the Domain in DyLOG

In our work, we have developed a multi-agent system, Wlog, whose architecture will be described in Section 6, that performs the tasks described in the previous section. The system's kernel is a set of rational agents, called *reasoners*, that have been implemented in the agent programming language DyLOG. In the current and in the following section we will respectively explain how it is possible to represent both the domain knowledge and the agents' behaviour by using the DyLOG language and how it is possible to interpret adaptive tutoring services as reasoning about actions tasks.

The reader who would like to learn more about the DyLOG language will find a thorough description of it in (Baldoni et al. 2001b; Patti 2002).

3.1. *The* DyLOG *agent programming language*

DyLOG is a language for programming agents based on a logical theory for reasoning about action and change in a *modal logic programming* setting. Agents are entities that read, understand, modify, or more generally interact with their environment by performing actions. Therefore, an agent's behavior can be fully described in a non-deterministic way by giving the *set of actions* that it can perform.

Actions have preconditions to their application, which may be conditions either about the external world or about the agent internal state, and they produce expected effects. This situation recalls what we said about courses: a course has preconditions and it has effects on the knowledge of the student. An action effect may also cause further effects, which do not directly derive from the execution of that action but from a set of causal rules which depend on the domain and that are triggered automatically when some conditions become true. For instance, let us think to a robot that plugs in an iron: an indirect effect of this action is that the iron becomes hot. The robot did not heat the iron but it caused its increase of temperature by plugging it in.

3.2. *States: Representing the knowledge of a student*

We have mentioned that actions can be applied only if their preconditions are true. Once applied, actions produce changes either to the world or to the agent's knowledge. We can, then, think to the whole reasoning process as a sequence of *transitions* between *states*. Our states represent the *knowledge* that the agent has about the world and not the state of the world itself (which we do not model). A state can also be seen as the result of an action sequence, applied to some initial state. Technically speaking, 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, so we want to have both the possibility of representing that some fluents are unknown and the capability of reasoning about the execution of actions on incomplete states. To explicitly represent the unknown value of some fluents, in (Baldoni et al. 2001b) we introduced an epistemic level in our representation language. In particular, we introduced an epistemic operator $\mathcal{B}$, to represent the beliefs an agent has about the world: $\mathcal{B}f$ means that the fluent f is known to be true, $\mathcal{B}\neg f$ means that the fluent f is known to be false. A fluent f is undefined when both $\neg\mathcal{B}f$ and $\neg\mathcal{B}\neg f$ hold at the same time $(\neg\mathcal{B}f \wedge \neg\mathcal{B}\neg f)$. For expressing that a fluent f is undefined, we write $u(f)$. Thus each fluent can have one of the three values: *true*, *false* or *unknown*.

Nevertheless, in our *implementation* of DyLOG (and, for the sake of simplicity, 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. This choice is due to the fact that operator $\mathcal{B}$ is indeed very useful when an agent induces some information (fluent values) about the world and, thus, it cannot be certain about it. Uncertainty is not present in the case that we currently tackle, although the possibility of dealing with uncertainty will allow us, in future work, to enrich our agents with the capability of carrying on complex dialogues with the user, in the line of the works of (Bretier and Sadek 1997).

In our application, a state contains fluents that capture those pieces of information about a student's education, that a rational agent believes to be true at a certain time. In particular, the fluents that we use represent:

1. the set of already attended *courses*: for each attended course and for each competence, that is a direct effect of attending that course, there is a fluent **knows(course, competence)** that records the way in which the competence was acquired. Observe that if a course supplies many competences the state will contain as many fluents *knows*(*course*, *competence*). We call the competences that are direct effects of actions *basic* competences;
2. the *competences* that the student has acquired: the state also contains a fluent **has_competence(competence)** for each competence that the student has, independently from the way it has been acquired;
3. the *learning goal* of the student: in this case we use the fluent **requested(goal)**. Notice that we do not strictly partition the set of all the competences in a set of derived competences that is disjunct from a set of the basic competences (related to courses); actually, a same competence may be acquired in different ways: it could either be obtained by learning the smaller pieces of competence it is made of or it could be obtained by attending a single course, if any is available, aimed at teaching that topic. This characteristic is particularly important in the perspective of building an open system, in which courses can be added or removed along time.

As an example, a state may contain the following fluents:

(a) *knows*(*'programming lab', 'java'*).
(b) *knows*(*'programming lab', 'event programming'*).
(c) *has_competence*(*'programming languages'*).
(d) *requested*(*has_competence*(*curriculum*(*'web application'*)).

Fluents (a) and (b) mean that a student has passed the programming course lab exam, thus acquiring the basic competences *java* and *event programming*. The student also acquired the derived competence *programming languages*, see fluent (c). Eventually, fluent (d) states that the student's goal is to acquire the competences supplied by the "web applications" curriculum.

3.3. *Course as actions*

In DyLOG, *primitive actions* are the basic building blocks for defining an agent's behavior. From a modal logic point of view, each primitive action a is represented by a modality $[a]$ (box a). The meaning of the formula $[a]\alpha$ is that α holds after any execution of action a. We can also write: $\langle a\rangle\alpha$ (possible a), whose meaning is that there is a possible execution of action a after which

α holds. The special modality $\Box$ (box) is used to denote those formulas that hold in all states, i.e., after any action sequence.

The direct and indirect effects of primitive actions on states are described by *simple action laws*, which consist of *action laws*, *precondition laws*, and *causal laws*. Intuitively, an action can be executed in a state s if the preconditions to the action hold in s; the execution of the action modifies the state according to the action and causal laws. We also assume that the value of a fluent *persists* from one state to the next one, if the action does not cause it to change. In the following we will define the simple action laws giving also the syntax that we use in the language for defining them and, last but not least, their representation in the modal logic framework.

1. *Action laws* define *direct* effects of primitive actions on a fluent and can also be used for representing action conditional effects. In the language they have the form:

$$a \textbf{ causes } F \textbf{ if } Fs \tag{1}$$

   where a is a primitive action name, F is a fluent, and Fs is a fluent conjunction, meaning that action a has effect F, when executed in a state where the fluent precondition Fs holds. The modal logic representation for this rule is $\Box(Fs \rightarrow [a]F)$.
2. *Precondition laws* allow action *preconditions*, i.e., those conditions which make an action executable in a state, to be specified. In DyLOG they are written as:

$$a \textbf{ possible if } Fs \tag{2}$$

   meaning that when the fluent conjunction Fs holds in a state, the execution of action a is possible in that state. The modal logic representation of precondition laws is $\Box(Fs \rightarrow \langle a \rangle true)$.
3. *Causal laws* are used to express *causal dependencies* among fluents and, then, to describe *indirect* effects of primitive actions. In the language they are written as:

$$F \textbf{ if } Fs \tag{3}$$

   meaning that the fluent F holds if the fluent conjunction Fs holds too. The modal logic representation of such rules is $\Box(Fs \rightarrow F)$.

In the tutoring system that we have implemented, each course is interpreted as the *action* of attending that course, therefore it is represented as a set of simple action laws. As an example, consider Figure 3, in which the representation in DyLOG of the "Programming Lab" course described in

(a) *attend*(*course*('*programming lab*')) **possible if**
knows('*programming theory*',_).
(b) *attend*(*course*('*programming lab*')) **causes**
knows('*programming lab*','*java*').
(c) *attend*(*course*('*programming lab*')) **causes**
knows('*programming lab*','*event programming*').
(d) *attend*(*course*('*programming lab*')) **causes**
knows('*programming lab*','*C and java relationship*') **if**
has_competence('*C language*').
(e) *attend*(*course*('*programming lab*')) **causes** *credit*(*B*1) **if**
get_credits('*programming lab*',*C*) ∧
credit(*B*) ∧ (*B*1 *is B* + *C*).
(f) *has_competence*('*programming*') **if**
has_competence('*data structures*') ∧
has_competence('*algorithms*') ∧
has_competence('*programming languages*').
(g) *has_competence*('*programming languages*') **if**
has_competence('*java language*').
(h) *has_competence*(*Competence*) **if**
knows(_,*Competence*).

Figure 3. Precondition, action, and causal laws.

Figure 1 is shown. Rule (a) states that the action *attend(course('programming lab'))* can be executed if the action of attending the "programming theory" course has already been executed. Action laws (b)–(c) describe the unconditional effects of the action execution: adding the "programming lab" course causes to have competence about *Java* and *event programming*. Action law (d) describes the conditional effect of the action at issue. Finally, action law (e) updates other fluents (credit) that control the length of the desired study plan. Rules (f) and (g) describe the *indirect effects* of having a competence; they are used for inferring the higher-level competences of a student based on his known competences. Finally, since the agent reasons on the student's competences independently from how they were obtained, rule (h) states that if a competence is a direct effect of attending a course (the underscore in *knows*(_,*Competence*) means that we do not care which course was actually attended) it will be a student's competence (*has_competence*(*Competence*)).

3.4. *Curriculum schemas as procedures*

So far, we have seen the basic building blocks of the DyLOG language. However, in order to be able to represent behaviour strategies we need

some compound syntax element. This is given by procedure clauses, which allow us to define complex actions. More precisely, in our language *complex actions* are defined as procedure clauses on the basis of primitive actions, sensing actions, *test* actions[2] and other complex actions. A *complex action* is a collection of *procedure clauses* of the form:

$$p_0 \textbf{ is } p_1, \ldots, p_n (n \geq 0) \tag{4}$$

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

From a theoretical 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:

$$\langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi \supset \langle p_0 \rangle \varphi.$$

Its meaning is that if in a state there is a possible execution of p_1, followed by a possible execution of p_2, and so on up to p_n, then in that state there is a possible execution of p_0.

In the curriculum sequencing application, procedures schematize the way for acquiring *professional expertise*. For instance, in Figure 4 we report a little part of the procedures that describe how to acquire a *web applications* expertise. In particular, we have expanded only part of the procedures for acquiring competence about programming. This example will be further discussed in the following sections.

3.5. *Interaction as sensing and suggesting actions*

In the previous sections we have explained that in our approach agents keep a mental state of the situation that they are currently tackling, which is modified by action execution. Generally speaking, however, the agent's knowledge may be incomplete so, in order to understand which actions can be applied, it is sometimes necessary to acquire new information from the outer world. To this aim, we have studied and integrated in the formal account of the language some informative actions, whose outcome is not under the agent's control but depends on the environment; such actions are called *sensing actions*. The difference with respect to the other kinds of actions, that we have seen so far, is that they allow an agent to acquire knowledge about the value of a fluent f rather than to change it.

(a) *achieve_goal*(*has_competence*(*curriculum*(*'web applications'*))) **is**
achieve_goal(*has_competence*(*'first year competences'*)) ∧
achieve_goal(*has_competence*(*'database'*)) ∧
achieve_goal(*has_competence*(*'ai'*)) ∧
achieve_goal(*knows*(*'distributed systems'*_)) ∧
achieve_goal(*has_competence*(*'web technology'*)).

(b) *achieve_goal*(*has_competence*(*'first year competences'*)) **is**
...
achieve_goal(*has_competence*(*'programming'*)) ∧
...

(c) *achieve_goal*(*has_competence*(*'programming'*)) **is**
achieve_goal(*has_competence*(*'data structures'*)) ∧
achieve_goal(*has_competence*(*'algorithms'*)) ∧
achieve_goal(*has_competence*(*'programming languages'*)).

(d) *achieve_goal*(*has_competence*(*'programming languages'*)) **is**
achieve_goal(*has_competence*(*'c'*)).

(e) *achieve_goal*(*has_competence*(*'programming languages'*)) **is**
achieve_goal(*has_competence*(*'java'*)).

(f) *achieve_goal*(*has_competence*(*'programming languages'*)) **is**
achieve_goal(*has_competence*(*'prolog'*)).

Figure 4. Procedure clauses.

(a) *achieve_goal*(*knows*(*Course,Competence*)) **is**
Course ≠ *generic* ∧
attend(*Course*).

(b) *achieve_goal*(*knows*(*generic,Competence*)) **is**
?*u*(*knows*(*generic,Competence*)) ∧
offer_course_on(*Competence*) ∧
?*course_on*(*Competence,Course*) ∧
attend(*Course*).

(c) *offer_course_on*(_) **possible if** *true*.

(d) *offer_course_on*(*Keyword*) **suggests** *course_on*(*Keyword,*_).

Figure 5. Suggesting actions.

In DyLOG direct effects of *sensing actions* are represented by using *knowledge laws* that have form:

$$s \textbf{ senses } f \tag{5}$$

meaning that action s causes to know whether f holds.

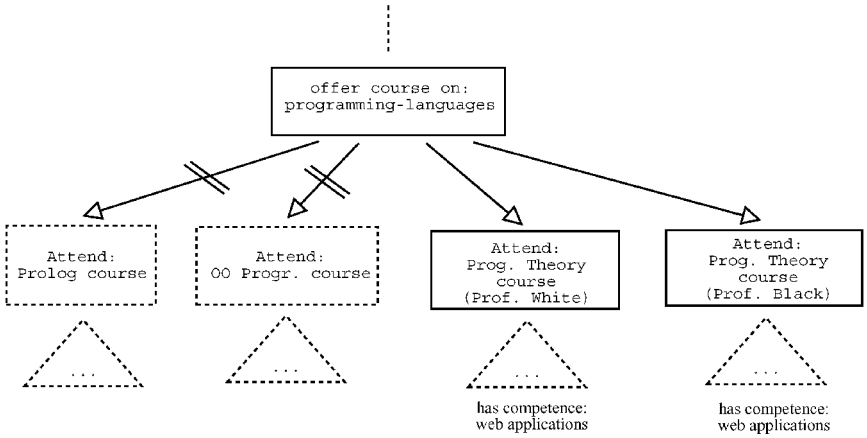

Figure 6. An example of selection of courses to offer for constructing a web applications curriculum.

In the current application, sensing actions are used also for allowing the agent to *interact with the user*. In the simplest case, the user is explicitly requested to enter the truth value of a fluent. This kind of interaction, however, is not sufficient, because rather than asking for a fluent's truth value, it is often more useful to offer a set of alternatives to the user, among which he will make a choice. To this aim, we have defined a special subset of sensing actions, called *suggesting actions*. For representing the effects of such actions we use the notation:

$$s \textbf{ suggests } f \tag{6}$$

meaning that action s suggests a (restricted) set of values for fluent f and causes to know the value of f.

Formally, the difference w.r.t. standard sensing actions is that while those consider as alternative values for a given fluent its whole domain, suggesting actions offer only a subset of it. The agent has an active role in selecting the possible values among which the user chooses: only those values that lead to fulfill the goal will be selected. Such values are identified by means of a reasoning process. After reasoning about a given problem, the agent finds a set of alternative items that are *equivalent* w.r.t. the task of achieving a given goal. Since using one or the other is equivalent, the reasoner may decide to leave the choice up to the user.

For example, let us consider again our application. We have seen that procedures are used to schematize the way to achieve a certain professional expertise in terms of simpler competences to acquire. We have also introduced basic competences, saying that they are all those competences that

are supplied by single courses; more than one course could supply the same competence but the system does not necessarily present all of the alternatives to the student. Figure 6 shows an example where the agent is helping a student to build a bioinformatics study plan: at a certain point of the plan construction, the agent finds four alternative courses that give competence about "programming languages", however, only two subtrees allow the student to get competence about imperative languages, necessary for a bioinformatics curriculum (i.e., the actual learning goal). The other branches are cut during the reasoning phase and the corresponding courses are not offered to the student. In different words, only those alternatives that open paths that will lead to the fulfillment of the user's learning goal will be selected. Afterwards, the choice can be left to the user: in fact, whatever the choice, the goal will be reached.

Let us now consider our agent implementation. Professional expertise definitions that, as we will see, are used for accomplishing the task of building a study plan, are expressed by means of the procedure *achieve_goal*, as reported in Figure 4. The *achieve_goal* definition encompasses also two cases, that are described in Figure 5:

rule (a) – some specific course is requested: *knows*(*Course,Competence*)
rule (b) – some competence is requested: *knows*(*generic,Competence*).[3]

Rule (a) states that if a specific course is requested, this is to be added to the study plan (action *attend*(*Course*)). Rule (b), instead, formalizes the case in which we are not interested in a specific course, therefore, the agent searches for all of the possible alternatives and suggests them to the user, waiting for his choice. In the agent implementation the *suggesting action* aimed at offering a set of alternative courses is *offer_course_on*, see rules (c) and (d). In particular, the possible alternatives are extracted from the domain knowledge by *course_on* in rule (d).

4. Tutoring Adaptive Services as Reasoning about Action Tasks

In Section 3 a representation of the virtual tutor domain in terms of a DyLOG domain description has been introduced. On this basis, we can interpret the adaptive services described in Section 2 – building and validating personalized curricula – as "reasoning about actions" tasks.

In general, given a domain described in a logic action framework, the main kinds of reasoning tasks that can be performed are *temporal projection* (or prediction), *temporal explanation* (or postdiction) and *planning* (Sandewall, 1994). Intuitively, temporal projection is a method for reasoning from causes to effects; its aim is to predict the effects of actions, that have not been

executed yet, based on (even partial) knowledge about the current state. On the contrary, by performing temporal explanation an agent considers some facts as effects of *already executed* actions and reasons about the possible causes that produced them. In different words, the aim of temporal explanation is to infer knowledge about the past states of the world, starting from some knowledge about the current state. The third reasoning task, *planning*, is probably the best known of the three; it is aimed at finding an action sequence that, when executed starting from a given state of the world, produces a new state where certain desired properties hold. All such reasoning tasks are supported by DyLOG.

In the following, we show how we interpreted the problem of personalized curricula construction as a procedural planning problem. Then, we describe how we interpreted the problem of validating a user-given plan as a temporal projection problem and how to exploit a simple temporal explanation mechanism for helping the student to understand the reasons of validation failure.

4.1. *Building personalized curricula by procedural planning*

Generally speaking, the planning problem amounts to determine, given an initial state s, if there is a sequence of actions that, when executed in s, leads to a goal state, in which a desired condition Fs holds.

In the DyLOG framework we consider a specific instance of the planning problem, in which we wonder if there is a possible execution of a *procedure* p, leading to a state in which some condition Fs holds, Fs being a set of fluents. In the modal language, this problem is expressed by the query $\langle p \rangle Fs$, which is to be read: "given a procedure p, is there a terminating execution of p (i.e., a finite sequence of primitive actions), leading from an initial state, that corresponds to the current situation, to a state in which Fs holds?". In DyLOG we refer to this query with the English-like notation:

$$Fs \textbf{ after } p \tag{7}$$

Intuitively, the *terminating executions* of p that lead to the goal state are *plan*s to bring about Fs. Indeed, the procedure *constrains* the space in which the plan is sought for; in the literature, this reformulation is known as *procedural planning*. In the curriculum sequencing application, procedures schematize how to acquire *professional expertise* (see the previous section), whereas Fs expresses the set of competences that a student would like to acquire. The student, who asked for support in the construction of a study plan, may have already acquired some of the necessary competences before the study plan construction; in that case, the fluents that express his current expertise will be part of the initial state.

The execution of the above query returns as a side-effect an *execution trace* of p, i.e., one of the terminating sequences $a_1, \ldots, a_n$ of primitive actions leading to the final state. Such a trace can either be a *linear* or, when the procedure contains sensing or suggesting actions, a *conditional* plan. In fact, if some of the p_i's include sensing or suggesting actions, the obtained execution trace contains also the foreseen communication acts with the user. Due to the fact that their outcomes are unknown at planning time, all the possible alternatives are to be taken into account; therefore, we will obtain a conditional plan, whose branches correspond each to one of the alternative values. It is important to remark that only those values that lead to success will be taken into account and will produce a branch in the conditional plan.

In Figure 4, a set of procedures describing how to acquire the body of competence for a *web applications* curriculum is shown. Let us suppose that our student asked to be supported in the design of a study plan for becoming an expert of *web applications* with the further requirement of acquiring competence about *graph theory*. Furthermore, suppose the student also added some constraints on the length of the plan: it should not exceed 132 credits. This request can be expressed by the query:

$$\{\mathit{has\_competence}(\text{`}\mathit{graph\ theory}\text{'}), \mathit{credit}(C), (C \leq 132)\}$$
$$\textbf{after}\ \mathit{achieve\_goal}(\mathit{has\_competence}(\text{`}\mathit{web\ applications}\text{'}))$$

where the set of conditions $\{\mathit{has\_competence}(\text{`}\mathit{graph\ theory}\text{'}), \mathit{credit}(C), (C \leq 132)\}$ is the set *Fs*, which the student wishes to hold after the execution of the procedure $\mathit{achieve\_goal}(\mathit{has\_competence}(\text{`}\mathit{web\ applications}\text{'}))$. If such an execution trace exists, it will correspond to a personalized study plan because, besides achieving the main learning goal $\mathit{has\_competence}(\text{`}\mathit{web\ applications}\text{'})$, it will also fulfill the additional requirements contained in *Fs* by supplying the competence *graph theory* and not exceeding 132 credits. Let us suppose that our student is not a beginner because he already attended the *first year* university studies and also some courses that supply the *database* competence. The system will suggest a course sequence for achieving only the missing competences, which (with reference to Figure 4) are *artificial intelligence*, *distributed systems* and *web technology*. Figure 7 sketches the conditional plan that results from planning in the described situation. The plan specifies a set of courses that the student is recommended to attend. The branching points of the conditional plan are to be interpreted as questions that will be posed to the student at execution time; by answering all the questions the student will select one of the alternative study plans. Queries are inserted in the conditional plan during its construction. In fact, when the planning process finds that different courses supply a required competence, it introduces an interactive action for offering all of the alternatives to the student.

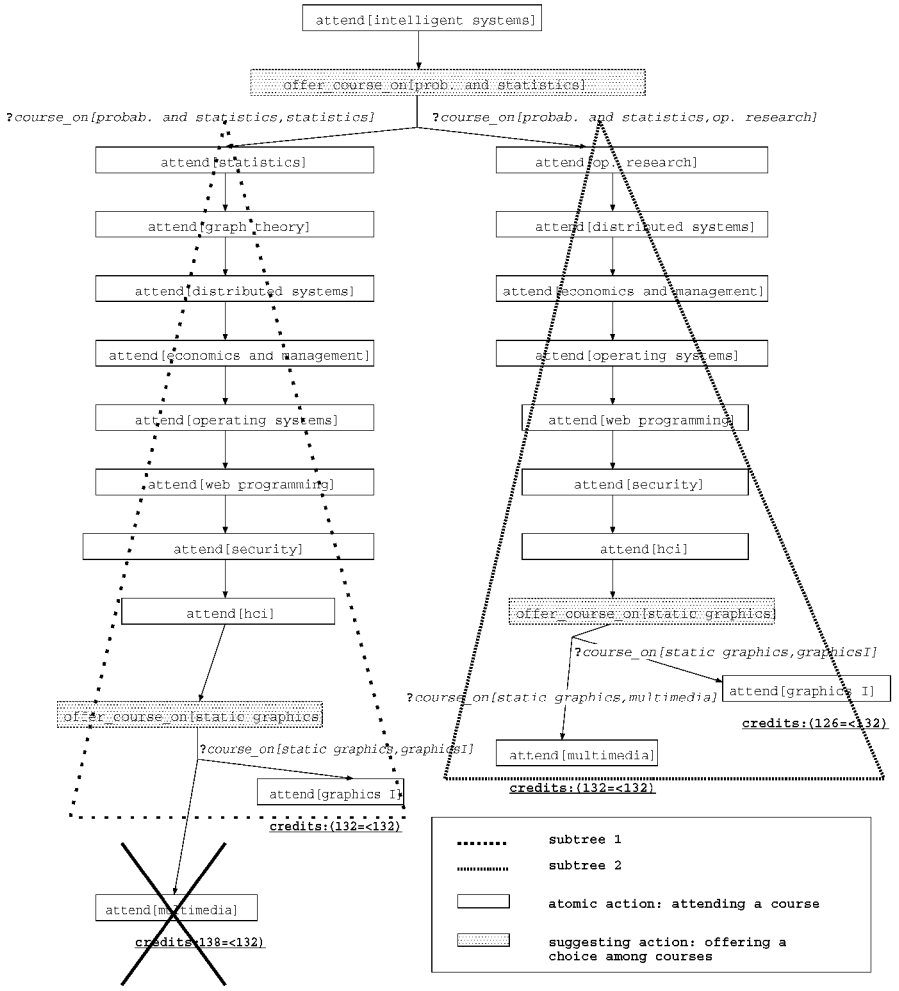

Figure 7. An example of conditional plan for a personalized curriculum in "web applications", that does not exceed the requested 132 credits and guarantees to acquire competence on graph theory.

Further suggestions are committed to the user's choice. For instance, in the plan above after recommending the student to attend the "intelligent systems" course, the system offers the student the choice between two different courses – i.e., "statistics" and "operations research" – which supply competence on *probability and statistics*. Further recommendations depend on the student's decision. For instance, when the preferred option is "operations research" (right subtree, Figure 7), the "graph theory" course is not added to the study plan, because the "operations research" course already provides the compet-

ence that this course supplies. At the end of that path another branching point corresponds to a query where the student must choose between two courses ("multimedia" and "graphics I") that supply competence on *static graphics*. Notice that such a choice is not given in case the student previously selected the "statistics" course. This is because of the user's constraint on credits. In fact, at the bottom of the left subtree of Figure 7 only one of the courses allows the system to build a study plan that does not exceed 132 credits. The "multimedia" course is too long (the study plan would be 6 credits longer than requested), therefore, it is not taken into account.

Finally, let us stress that all of the branches of the conditional plan lead to a state where the condition to have competence on *graph theory* is satisfied. Intuitively, it means that, no matter the alternatives preferred by the user during the actual interaction, it is guaranteed that the resulting study plan leads to the fulfillment of the learning goal. The addition of further requirements about the competence to achieve affects the generated conditional plan, causing the cut of some branches. If, for instance, our student asks also for competence about *audio*, since such competence is provided only by the "multimedia" course, the system would insert in the conditional plan only those paths that contain the "multimedia" course; with reference to Figure 7, those paths that contain the "graphic I" course would be cut.

4.2. *Validation of curricula by temporal projection*

Another reasoning process that is extremely interesting in the context of personalized course sequencing is the *validation of a student-given study plan*. In this case, after defining the course sequence according to his personal taste and interests, the student asks the system if it satisfies the learning dependencies of the domain, allowing to achieve some desired learning goal (some specific expertise).

This task can easily be interpreted as a *temporal projection* problem. In spoken words, the temporal projection problem is defined as: "given an initial state s and an action sequence $a_1, \ldots, a_n$, does the condition Fs hold after the execution of the action sequence?". Differently than in planning, in this case the action sequence is given. In the logical framework that we defined, we formalize the temporal projection task by means of the query:

$$\langle a_1 \rangle \ldots \langle a_n \rangle F_s$$

where each a_i is a primitive action and Fs is a conjunction of fluents. Notice that, since the primitive actions defined in our domain descriptions are *deterministic* w.r.t. the epistemic state, the equivalence $\langle a \rangle Fs \equiv [a]Fs \wedge \langle a \rangle \top$ holds for all the actions a that are defined in the domain description. There-

fore, the success of the existential query $\langle a_1 \rangle \ldots \langle a_n \rangle F_s$ entails the success of the universal query $[a_1] \ldots [a_n] Fs$.

In DyLOG we represent the query that formalizes the temporal projection task by the English-like notation:

$$Fs \textbf{ after } a_1, \ldots, a_n$$

In the curriculum sequencing context, the sequence of actions is a sequence of courses $c_1, \ldots, c_n$ to attend while the final condition Fs is a set of desired competences. The temporal projection task can then be read as "Given the initial student background, can the student acquire the set of competences Fs by attending the course sequence $c_1, \ldots, c_n$?". In the implementation this query becomes:

$$Fs \textbf{ after } attend(course(c_1)); \ldots; attend(course(c_n)) \tag{8}$$

where Fs is the student's learning goal.

As a first, simple example, let us consider a student, who asks his tutor if, by attending the course sequence "programming theory" followed by "programming lab" – in the given order –, he will acquire competence about *event programming* (which is the student's learning goal). Let us also suppose that the student has no previously acquired competence about programming or other related topics. The corresponding DyLOG query will be:

has_competence('*event programming*') **after**
attend(*course*('*programming theory*'));
attend(*course*('*programming lab*'))

Since all the learning dependencies are respected and attending the "programming lab" course causes the acquisition of competence about *event programming* (see Figure 3), the validation of this query will succeed.

Indeed in our framework *validation may fail* for *two reasons*: either the preconditions to one of the actions do not hold in the state in which the action is executed (sequencing problem) or the learning goal is not achieved. For instance, the query:

has_competence('*event programming*') **after**
attend(*course*('*programming lab*'));
attend(*course*('*programming theory*'))

would fail because the "programming lab" course cannot be attended if the student does not have competence about *programming theory*, which is the effect of attending the "programming theory" course. On the other hand, the query:

has_competence(‘*multimedia*’) **after**
attend(*course*(‘*programming theory*’));
attend(*course*(‘*programming lab*’))

would fail because none of the two courses in the sequence allows the student to acquire competence in *multimedia*, although the action sequencing is correct.

4.3. *Explanation of validation failure by temporal explanation*

As we have seen above, the validation of a user-built plan may fail for different reasons: the plan may be wrong because it does not allow to reach the final desired competence (the action sequence is correct but it does not lead to the learning goal) or because the sequencing does not respect some of the learning dependencies (the student does not have the necessary competence for attending the next course in the plan). In either case, it is extremely important to return a feedback to the user about the reasons of validation failure, in order to support plan correction.[4]

Course sequencing is, actually, quite a special application domain. Its peculiarity is that competences can only be added. Intuitively, no new course will ever erase from the students’ memory the concepts acquired in previous courses. More formally, the domain is *monotonic*. This consideration is very helpful in the definition of a failure explanation mechanism; the one that we propose is based on the notion of state *completion* and exploits a mechanism known as *temporal explanation*. The information that we compute to return as a feedback is the set of competences that the student should already have in order for the plan to be valid. So, for instance, if a student adds to his study plan the course “programming lab” but, standing to the information that the system has, the student does not have notions about “programming theory”, the system will tell the student that he can attend his plan only if he already has notions about programming theory, otherwise he will not be able to understand the contents of the “programming lab” course.

More formally, a task of temporal explanation amounts to reconstruct, starting from some given observations, what has happened (more generally, what should have happened) in order for those observations to be true. For dealing with temporal explanation, we adopt an *abductive approach* by determining the assumptions on the initial state that are needed for explaining observations on later states. In the case of courses and study plans, such assumptions will be a set of competences: all those competences that the student did not declare to have and that are not supplied by the courses in the sequence up to a state in which they result to be necessary. The intuitive idea is that a study plan is always applicable, given that the student has as background knowledge the missing competences.

While the reasoning mechanisms of planning and temporal projection, used in Sections 4.2 and 4.1, are based on the proof procedure described in (Baldoni et al. 2001b), *temporal explanation* is based on the work contained in (Baldoni et al. 1997). In that work an abductive proof procedure is defined in terms of an auxiliary nondeterministic procedure *support*, which carries out the computation for the monotonic part of the action language. Given a domain description Π and a query of form:

$$Fs \textbf{ after } attend(course(c_1)); \ldots; attend(course(c_n))$$

let us pose, for the sake of simplicity, $Q = \langle attend(course(c_1))\rangle \ldots \langle attend(course(c_n))\rangle Fs$, then, the procedure *support*(Q,Π), described hereafter, returns an abductive support for the query Q in Π, which is a set Δ of abducibles that, when added to the domain description, allows us to derive our query:

$$\Pi \cup \Delta \vdash_{vs} [add(course(c_1))] \ldots [add(course(c_n))]Fs$$

Briefly, abducibles are atomic propositions of the form $\mathbf{M}[a_1]\ldots[a_n]F$ (F being a fluent), where $\mathbf{M}$ is not to be regarded as a modality: this notation has been adopted in analogy to default logic and $\mathbf{M}[a_1]\ldots[a_n]F$ means that F is consistent after the execution of the sequence of actions $a_1, \ldots, a_n$.

All the details concerning the implementation of the monotonic part of the language are hidden in the definition of the procedure *support*, and they can be ignored by the abductive procedure. The abductive procedure is defined in the style of Eshghi and Kowalski's abductive procedure for logic programs with negation as failure (Eshghi and Kowalski 1989), and it is similar to the procedures proposed in (Toni and Kakas 1995) to compute the acceptability semantics. Here we report only the support procedure, that was modified in the following way, while the abductive procedure remained unchanged (see (Baldoni et al. 1997, Section 4):

1. $a_1, \ldots, a_m \vdash_{vs} \top$ with $\emptyset$;
2. $a_1, \ldots, a_m \vdash_{vs} F$ with Δ if
   a) $a_1, \ldots, a_{m-1} \vdash_{vs} Fs'$ with Δ, where $m > 0$ and $\Box(Fs' \supset [a_m]F) \in \Pi$; or
   b) $a_1, \ldots, a_{m-1} \vdash_{vs} F$ with Δ_1 and $\Delta = \Delta_1 \cup \{\mathbf{M}[a_1, \ldots, a_m]F\}$; or
   c) $m = 0$ and $\Delta = \emptyset$ if $F \in S_0$, $\Delta = \{\mathbf{M}F\}$ otherwise;
3. $a_1, \ldots, a_m \vdash_{vs} Fs_1 \wedge Fs_2$ with $\Delta_1 \cup \Delta_2$ if $a_1, \ldots, a_m \vdash_{vs} Fs_1$ with Δ_1 and $a_1, \ldots, a_m \vdash_{vs} Fs_2$ with Δ_2;
4. $a_1, \ldots, a_m \vdash_{vs} \mathcal{M}\ell$ with Δ if $a_1, \ldots, a_m \vdash_{vs} \mathcal{B}\ell$ with Δ;
5. $a_1, \ldots, a_m \vdash_{vs} [a'_1, a'_2; \ldots; a'_n]Fs$ with $\Delta_1 \cup \Delta_2$ if $a_1, \ldots, a_m \vdash_{vs} Fs'$ with Δ_1 and $a_1, \ldots, a_m, a \vdash_{vs} [a'_2; \ldots; a'_n]Fs$ with Δ_2.

(a) *attend*(*course*('*database*')) **possible if**
has_competence('*matrices*') ∧
has_competence('*dynamic structures*').
(b) *attend*(*course*('*database*')) **causes**
knows('*database*','*db*').
(c) *attend*(*course*('*database*')) **causes** *credit*(*B*1) **if**
get_credits('*database*',*C*) ∧ *credit*(*B*) ∧ (*B*1 *is B* + *C*).

Figure 8. Action and precondition laws for the course *database*.

To prove a fluent F, we can either select a clause in the domain description Π, rule 2(a), or add a new assumption to the assumption set Δ, rule 2(b) and 2(c). A query $\langle a_1 \rangle \ldots \langle a_m \rangle Fs$ can be derived from a domain description Π with assumptions Δ if, using the rules above, we can derive $\varepsilon \vdash_{vs} [a_1] \ldots [a_m] Fs$.

As an example, let us consider again a student who would like to acquire competence about *event programming*, this time by attending the courses "database" and "programming lab" in the given order. Let us suppose that, standing to the information that the system has, the student has no notion about *programming theory*, which is a prerequisite for attending the *programming lab* course (Figure 3, clause (a)), nor about *dynamic structures* and *matrices*, that is necessary in order to attend the "database" course (Figure 8). The request of the student is represented by the following query:

(q_1) *has_competence*('*event programming*') **after**
attend(*course*('*database*'));
attend(*course*('*programming lab*'))

Given a domain description Π, that includes both the simple action laws for "programming lab" and the simple action laws for the course "database", the proof procedure returns a support Δ for the query (q_1), that contains the following assumptions on the initial state: $\mathbf{M}\mathcal{B}$ *has_competence* ('*matrices*'), $\mathbf{M}\mathcal{B}$ *has_competence* ('*dynamic structures*') and $\mathbf{M}\mathcal{B}$ *knows* ('*programming theory*', '*object languages*'). Intuitively, Δ is the tutor feedback about the proposed plan, which can be read as: "The plan that you proposed would allow you to learn *event programming* if you already had notions about programming theory, matrices and dynamic structures".

This failure explanation mechanism is quite simple and we would like to remark that it works due to the monotonicity of the domain that we are tackling, where new courses are not supposed to erase from the students' memory the concepts acquired in previous courses.

5. The Virtual Tutor as a Logic Agent

In our work we used the DyLOG language not only for representing the knowledge domain (as described in details in the previous sections) but also for programming the agents that use such a knowledge, i.e., for implementing the reasoners. In the specific application that we are presenting, at a very high-level reasoners have the following behavior:

1. acquire a problem definition from the user;
2. acquire the initial situation;
3. solve the problem and present a solution to the user;
4. further adapt the solution by interacting with the user.

Sometimes the solution can be achieved with no further interaction; more generally, however, the agent will ask the user for further information or for choosing among equivalent (w.r.t. the goal) alternatives, when it is the case. The whole communication between the two actors (the reasoner and the user) takes place by means of web pages that are constructed on the fly for presenting (requesting) information to (from) the user. Of course, although at an abstract level the different kinds of problem are fixed ("help me to build a study plan" or "validate this study plan"), there may be a wide variety of specific interests and interactions depending on the user and on his/her goals and situation. For each triple $\langle user,goal,situation \rangle$, a specific interaction will occur and, therefore, an ad hoc web page sequence will be generated.

Our reasoners are executed by the DyLOG interpreter, which is a straightforward implementation of the language proof procedure (Baldoni et al. 2001b). Every *primitive action* has some *code* associated to it, that is to be performed when the action is executed (the association is done by means of the keyword **performs**); such a code actually produces the effects of the action in the world. For instance, when the reasoner must show some information to the user, it executes a *showpage* action, which has associated some code for asking another agent, the actual execution device (see Section 6), to show an appropriate web page to the user. As a consequence, when the interpreter *executes* an action it must commit to it and it is not allowed to backtrack by retracting the effects of the executed action.[5]

However, reasoners perform rational tasks by *reasoning about* actions effects. In Section 4 we have seen that all the different kinds of reasoning exploit a query of the form Fs **after** p; the language interpreter provides a few meta-predicates for reasoning about actions in order to answer to this kind of query. More specifically, the meta-predicate:

$$plan(Fs \textbf{ after } p,\ as)$$

extracts a primitive action sequence *as* that, given a specific initial state, is a possible execution of procedure *p* that leads to a state in which *Fs* holds. Procedure *plan* works by executing *p* in the same way as the language interpreter with a main difference: primitive actions are executed "in the mind of the reasoner", without any effect on the external environment and, as a consequence, they are backtrackable. The meta-predicate *plan* is used both to perform *study plan construction* (see Section 4.1) and for *validation* (see Section 4.2). The explanation of validation failure, instead, is accomplished by means of the meta-predicate *explain*(*Fs* **after** *as,d*), that collects in *d* all the fluents that should be true in the initial state in order for *Fs* to hold after the execution of the sequence *as* of primitive actions.

5.1. *Implementing the virtual tutor in* dyLOG

The *behaviour* of a reasoner is described by a collection of procedures. In the case of *study plan construction*, see Section 4.1, the top level procedure, called *advice*, extracts a plan that will be executed. In the following, a question mark in front of a fluent means that the value of that fluent is to be checked. So, for instance, ?*requested*(*Curriculum*) will check which professional expertise the student declared to be interested in. In this case the fluent has a predefined finite domain.

(R1) *advice*(*Plan*) **is**
ask_user_preferences $\wedge$?*requested*(*Curriculum*) $\wedge$
plan(*credits*(*C*) $\wedge$ *max_credits*(*B*) $\wedge$ ($C \leq B$)**after**
achieve_goal(*has_competence*(*Curriculum*),*Plan*) $\wedge$ *Plan*.

Intuitively, the reasoner asks the student what kind of final expertise he wants to achieve and his background knowledge (e.g., if he already attended some of the possible courses). Afterwards, it adopts the user's goals and builds a *conditional plan* for reaching them, predicting also the future interactions with the user. That is, if it finds different courses that supply a same competence, whose prerequisites are satisfied, it plans to ask the user to make a choice. *plan* is the *meta-predicate* that actually builds the plan, in this case by extracting those executions of the procedure *achieve_goal* that satisfy the user's goals as well as the further conditions that are possibly specified (e.g., that the number of credits gained by following the study plan is not bigger than a predefined maximum).[6]

Eventually the conditional plan that is returned by the reasoning process is executed. This means that the code that is associated to every primitive or suggesting action, that is part of the returned plan, is executed, possibly modifying the environment. In our application a plan can only consist of two different kinds of actions: the primitive action *attend* and the suggesting

action *offer_course_on* (see Section 3.5). Rules (R2) and (R3) contain the code associated to such actions:

(R2) *attend*(*Course*) **performs** (
showCourse(*Course*)).

(R3) *offer_course_on*(*Keyword*) **performs** (
build_question(*Keyword_Question*) $\wedge$
ask_choice(*Question,Choice*)).

showCourse is a Prolog predicate that performs a FIPA-like communication with the executor (see next section) for commanding the visualization of a web page containing all the information about the course *Course*. The predicate *build_question* composes a question that suggests a set of alternative courses to the user asking for his preference; the question is stored into variable *Question*. Last but not least, *ask_choice* takes care of asking the composed question to the user, and then waits for the answer (which is stored in the variable *Choice*).

Initially the agent does not have explicit goals, because no interaction with the student has been performed. The student's inputs are obtained after the first interaction phase, carried on by the procedure *ask_user_preferences*:

(R4) *ask_user_preferences* **is**
verify_student_competence $\wedge$
offer_curriculum_type.

verify_student_competence is an action that allows the system to acquire knowledge about the current student's situation: mainly, which courses have been attended and successfully passed. *offer_curriculum_type*, instead, is used to acquire knowledge about the professional expertise the student would like to achieve. In Section 3.5, we have seen that in DyLOG information is acquired by means of special actions, called *sensing actions*. Differently than "normal" actions, they increase (or revise) the knowledge of the agent but they do not change its environment; indeed, *offer_curriculum_type* is an example of sensing action:

(R5) *offer_curriculum_type* **possible if** *true*.
(R6) *offer_curriculum_type* **senses** *requested*(*Curriculum*).

It is defined by means of both a precondition law that states that this action can always be executed (R5) and a sensing action law (R6), which states that, after the execution of *offer_curriculum_type*, the value of the fluent *requested*(*Curriculum*) – used in (R1) – will be known. Here the *goal adoption* occurs: the goal of the user becomes the goal of the reasoner.

In the case of *study plan validation*, see Section 4.2, the top level procedure is *check_study_plan*. This procedure – see (R7) –, after executing *verify_student_competence* that we have already explained, first interacts with the student so to get the study plan that he built (*ask_curriculum*(*Plan*)) and then asks him to input the competences he is interested in (*ask_desired_competence*(*Competence*)). Afterwards, it executes *Check*(*Plan, Competence*), which performs the actual validation.

(R7) *check_study_plan* **is**
verify_student_competence ∧
ask_curriculum(*Plan*) ∧
ask_desired_competence(*Competence*)∧
check(*Plan,Competence*).

(R8) *check*(*Plan,Competence*) **is**
plan(*Competence* **after** *Plan*,_) ∧
showpage("Your plan is OK").

(R9) *check*(*Plan,Competence*) **is**
showpage("Your plan is not OK") ∧
explain(*Competence* **after** *Plan*,*Delta*) ∧
showpage("Explanation:",*Delta*).

(R8) uses again the meta-predicate *plan*, which executes the query *Competence* **after** *Plan*; however, in this case, we are only interested in checking if the sequence of actions contained in *Plan* allows to achieve the requested competences: for this reason we discard the meta-predicate return value (second argument). The agent will return to the user an appropriate feedback by using the primitive action *showpage*, according to the result of the validation procedure. If the plans turns out to be wrong, rule (R9) executes the metapredicate *explain*, which, according to the approach described in Section 4.3, collects in variable *Delta* a list of competences that the student should already have in order to acquire the target *Competences* by following *Plan*.

6. The Multiagent System

Wlog, the prototype system that we developed, has the *multi-agent architecture* that is sketched in Figure 10. Agent technology allows complex systems to be easily assembled by means of the creation of distributed artifacts, that can accomplish their tasks through cooperation and interaction. Systems of this kind have the advantage of being modular and, therefore, flexible and

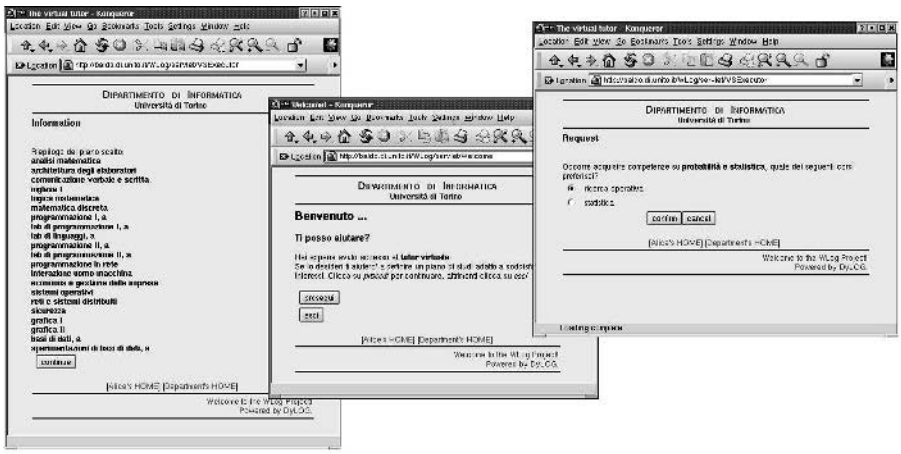

Figure 9. Interacting with Wlog.

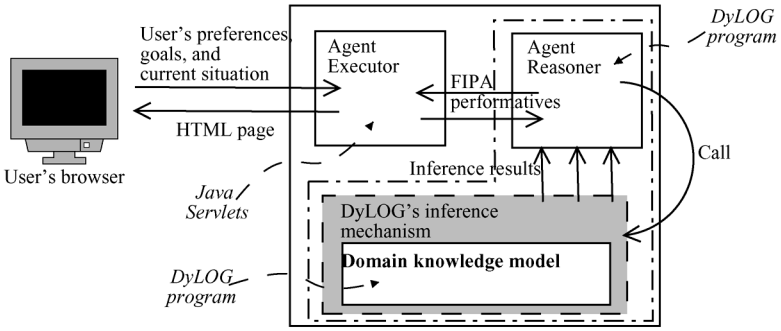

Figure 10. A sketch of Wlog architecture.

scalable. So, on one hand, each module can be developed by exploiting the best, specific technology for the service that it supplies, on the other, new components can be added for supporting either new functions or a wider number of users.

Wlog consists mainly of two kinds of agents: *reasoners* and *executors*. Reasoners are written in DyLOG, whereas executors are Java servlets embedded in a Tomcat web server. Executors are the interface between the rational agents and the users; they mainly produce HTML pages, driven by the directives sent by reasoners, and they forward the collected data to the reasoners themselves. Reasoners collect inputs from the users (preferences, goals, information about the current educational situation) and invoke the inference mechanism of the DyLOG language (see Section 4) on the domain knowledge model in order to accomplish one of the possible adaptive services, i.e., building a study plan or validating a student-given study plan.

As we have seen, also the domain knowledge model is defined in the DyLOG language. We would like to remark that while the use of DyLOG for representing the knowledge model and performing inferences is fundamental, the agent implementation described in Section 5 is written in DyLOG for convenience and it could, actually, be written in other programming languages, such as Java, the important thing being that the implementations call the DyLOG meta-predicates *plan* and *explain*, which perform the actual reasoning process.

The communication among the agents has the form of a FIPA-like message exchange in a distributed system (FIPA 1997). Each agent is identified by its location, which can be obtained by other agents from a facilitator, and has a private mailbox where it receives messages from other agents.

6.1. *Interaction between a tutor, an executor, and a user*

A user accesses the system by means of a normal web browser (Figure 9); from this moment until the end of the interaction, the user will be served by an *executor*. First the executor looks for a free reasoner by consulting the facilitator; since at the moment reasoners are not differentiated and can all perform all the different kinds of reasoning, that we have described in the previous sections, if any is available the interaction will begin.

Supposing that the previous step was successful, the user selects the service he is interested in and starts his interaction with the system. The next step will be the declaration of the user's goal, e.g., "I want to become an expert of web applications". The user's goal is adopted by the reasoner, that will start a conversation aimed at collecting information about the user's current situation. For instance, in the case of study plan construction the user will be asked about successfully passed exams. In the case of study plan validation, instead, the system will ask the study plan to validate. Of course, if our reasoning system were integrated in a wider system that is, for instance, connected with the secretariat databases, part of the information would be available without asking and the resulting interaction with the user would be simpler, although the kernel of the system would not change. At this point it is extremely interesting to understand how the interaction between the reasoner and the executor is carried on.

Figure 11, reports a finite state automaton, that represents the *interaction protocol* between the members of each couple ⟨reasoner, executor⟩. States are numbered and arcs are labelled with the speech acts that cause the various transitions. Different shading on states are used for specifying which agent will continue the conversation (white for the executor, gray for the reasoner). States with double border are terminating states.

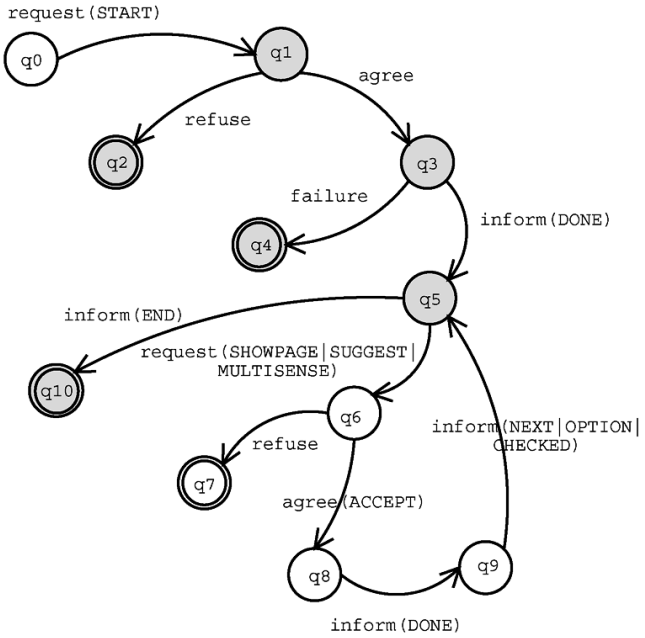

Figure 11. Communication protocol between an executor and a reasoner.

The part of graph that contains the states q_1 through q_4 encodes the connection of an executor with a reasoner (initialization phase). The part of graph consisting of the $q_5 - q_{10}$ states rules the actual *action-execution* cycle, i.e., the execution of primitive actions, commanded by the reasoner and performed by the executor. As we have seen in the previous Sections, in the application hereby described only a little number of primitive actions are defined: those necessary for sensing the inputs from the user plus *attend*(*Course*) and *offer_course_on*(*Keyword*). The former causes the description of a course to be displayed to the user, the latter causes an interaction in which, first, some alternatives are shown and, then, the user's choice is expected. Whenever a primitive action is executed an appropriate web page is to be produced and sent to the user's browser. The action-execution cycle takes care of this phase. The reasoner sends the executor the request of showing an HTML page by means of the *request* FIPA speech act from q_5 to q_6, completed with values that specify what to show, suggest or sense. The executor composes a proper HTML page and sends it to the user's browser; sometimes the page will contain a form to be filled. In either cases, when the user finishes to consult/fill the page/form, he asks the system to continue by clicking a button. The executor, then, informs the reasoner that the page has been consulted and, if necessary, also transmits to the reasoner the user's

data (*inform*() speech act from q_9 to q_5). Afterwards, it waits for the next command.

Both agents perform various controls on the messages that they receive, for guaranteeing the integrity of the interaction. For instance, if an executor receives a command from a reasoner, which is not serving its user, it will refuse to execute it. The same would happen if it were asked to execute an action that is not allowed in the current state. So if it has sent to the user's browser a form and it has not received any information in return, it will refuse to send to the browser any other page.

7. Conclusion and Related Work

In this article, we have presented an approach to adaptive tutoring, based on the use of a logic programming language that supports reasoning about actions and change. In our approach a group of agents, called reasoners, works on a real-world domain description, given in terms of a set of competences, which are connected by causal relationships. The set of all the possible competences and of their relations defines an ontology. This multi-level description of the domain bears along many advantages. The most straightforward is the simplicity of use of the system: on the one hand, no initialization phase is required (differently than in other, e.g., statistical, approaches); on the other hand, we can add, delete, modify course descriptions as well as expertise descriptions without affecting the system behavior because of the high modularity that this approach to knowledge description manifests. Last but not least, working at the level of competences is close to human intuition and enables both goal-directed reasoning processes and explanation mechanisms.

The logic approach also enables the validation of student-given study plans with respect to some learning goal of interest to the student himself. Basically the reasons for which a study plan may be wrong are two: either the sequentialization of courses is not correct or by attending that plan the student will not acquire the desired competence. In both cases we can detect all the weak points in the plan and return a precious feedback to the user. An interesting extension would be to automatically build what, according to the terminology proposed in (Baral et al. 2000), is known as a *repair plan*: an automatic correction of the wrong proposal. At the moment, however, we do not enact repair-planning policies. In fact, although at a first glance, it could seem that repairing a study plan means to complete it by adding some missing courses, the problem is actually not trivial. For instance, what to do if the patched plan violates some constraint (e.g., it is too long)? Should the system eliminate courses that the student chose but that are not really necessary for

acquiring the declared learning goal? What about adaptation in this case? We believe that repairing requires a close interaction between the system and the user, whose dynamics are yet to be investigated.

In our implementation, both the study plan construction and the study plan validation tasks are performed on-line. In the case of planning we could actually have followed an alternative approach: to build off-line the most general conditional plan for each professional expertise and to limit the on-line phase to a tree pruning, according to the inputs given by the user. However, this solution would not be efficient in the case in which the user asks to build a plan for achieving a generic set of competences (rather than a professional expertise out of the set offered by the system) nor in the case of plan validation. In fact, discovering whether a sequence of actions is an instance of a schema by matching the schema tree has a higher computational complexity than verifying its correctness by applying *temporal projection*, which is linear in the number of the elements in the sequence.

The approach that we proposed can generally be adopted for building recommendation systems. For instance, besides the application that we presented in this article, we used procedural planning also for building a prototype system that helps users to assemble personal computers according to their needs (Baldoni et al. 2001a). However, it is possible to widen the set of possible applications moving to the design of virtual supervisors. Presently we are, actually, working at a new application in which a student learns how to use an application software in a learning by doing framework. A virtual tutor silently monitors the user by verifying if and how he/she reaches a learning goal proposed by the system. One of the problems to solve in this context is to ignore useless actions, that the user performs either because he/she has little acquaintance with the software or because he/she is actually exploring menus and commands. Systems for helping the users to familiarize with softwares are already being developed, the problem is that usually they are simulators, whose design and implementation are very expensive. By reasoning on the user's actions, we think that the production of such systems would be simplified.

7.1. *Other approaches to curriculum sequencing*

In the Information Technology society, the field of adaptive hypermedia applied to educational issues is attracting greater and greater attention (Brusilovsky 2001). In the last few years considerable advancements have been yield in the area, with the development of a great number of systems, like ELMArt (Weber and Brusilovsky 2001), the KBS hyperbook system (Henze and Nejdl 2001), TANGOW (Carro et al. 1999), the authoring tool for course designing ATLAS (Macías and Castells 2001) and many others. Among the

technologies used in Web-based education for supporting student adaptation and guidance, *curriculum sequencing* is one of the most popular. Different methods have been proposed on how to determine which reading (or study) path to select or to generate in order to support in an optimal way the learner navigation through the hyperspace of knowledge items, see e.g. (Brusilovsky 2000; Weber and Brusilovsky 2001; Stern and Woolf 1998; Henze and Nejdl 2001).

In the last sections we described the usefulness of three techniques for reasoning about actions, based on logic, in a curriculum sequencing applicative framework; in the following we compare our application with some other Adaptive Educational Hypermedia systems which also implement curriculum sequencing even if in slightly different application frameworks. Our analysis will not be exhaustive – we have focused on a set of representative systems – and it is inspired by the concept-driven comparison framework defined in (Baldoni et al. 2002).

Let us start with the *KBS Hyperbook System* (Henze and Nejdl 1999, 2000), an AEH system which personalizes the access to information according to the particular needs of a student. KBS implements various adaptational functionalities, among which the generation and proposal of reading sequences through a hyperspace of information sources about Java programming. As in our case, in the KBS Hyperbook framework knowledge and actual information units are kept separate. The learning dependencies, used by the adaptation component of the system for the sequencing task, are expressed at the knowledge level. They are stored in a so called *knowledge model*, which contains the knowledge prerequisites required for understanding some concept, as well as the resulting knowledge. Curriculum sequencing allows the KBS system to compile a multi-step sequence of pages for helping a user to reach a certain learning goal. Such a sequence is compiled by following a stochastic approach that performs a depth first traversal of the knowledge model.

Our work also focuses on dependencies among knowledge elements (competences), even though, when necessary, also dependencies among the actual information items (the courses) can be expressed. As a difference, while sequencing in Henze and Nejdl (1999) is based upon a Bayesian approach, producing a partial order of knowledge elements, we adopted a symbolic approach based on a *modal logic theory of action*. In KBS dependencies between knowledge elements have the form $K1 < K2$, expressing the fact that $K1$ should be learned before $K2$. Therefore, the inferencing mechanism that enables the system to understand the dependencies between sets of knowledge elements is the transitive closure of the "$<$" relation. In our case, since information items are represented as "attend course" actions

that require or produce competences (our knowledge elements), the dependencies between information items and knowledge elements emerge by logical reasoning about "attend course" actions, using all the information modelled in the action theory. Indeed learning dependencies are inferred by logical derivation not only from the knowledge elements, which are in the precondition and effects of the courses, but also from the hierarchical structure among knowledge elements, encoded by causal laws, and from the specification of schematic professional expertises expressed by procedures.

One characteristic of our approach, the decoupling of knowledge from the set of the courses that are available at a specific time, makes it suitable to extensions to a more open framework and, in particular, to the development of open systems, where different sources of information are integrated. An example application could be supporting those students that apply to the Erasmus (or other) interchange program in choosing a set of courses to attend abroad. In this context an advantage of our approach is that, due to the fact that by means of DyLOG procedures we can express different composition strategies, we could specify different teaching policies (organizations of information presentation).

In the ELM-Art system (Weber and Brusilovsky 2001) curriculum sequencing is used for compiling a sequence of hypermedia documents that a student will follow for reaching a certain learning goal. As a difference with respect to KBS, in ELM-Art there is no distinction between knowledge elements and information items, thus the learning dependencies used by the adaptation component are coded at the level of the information units. Based on this model of dependencies, the system does not produce a multi-step reading sequence but suggests to the student the *next best page* to visit, which is calculated based on the reading path actually followed by the student and on the page prerequisites. A similar approach is taken in ACE (Specht and Oppermann 1998), a WWW-based tutoring framework that influenced the development of the recent WIND project (Specht et al. 2002). In ACE the domain model is built on a conceptual network of learning units: it describes a set of learning units and their interrelations and dependencies, without drawing a distinction among knowledge elements and information units. Besides prerequisite relationships among units, that specify a partial order of units in the learning space, the model can contain also a default curriculum sequence. The combination of these elements is used for adapting the student learning sequence step by step, according to the student's current knowledge. In particular ACE computes the *next best unit* to work on, depending on the probabilistic overlay model of a learner's knowledge and the prerequisites of the possible next units.

MetaLinks (Murray 2002), an authoring tool for adaptive hyperbooks, implements a functionality, "the narrative flow", that allows it to suggest step by step a reading path in a hyperspace of documents. Instead of coding learning dependencies in the usual way, i.e., by associating preconditions and outcomes to information units, MetaLinks represents *decompositional dependencies* by structuring the documents hierarchically in a way that parents are introductions or summaries of their children, while children detail the matter introduced by the parents. Based on this structure, the next page to visit is suggested by adopting a breadth first visit strategy, that exploits the concept of sibling, allowing a horizontal reading of the hierarchy: the next page to visit must be at the same level in the hierarchy of the current one, which intuitively means that they contain information at the same level of granularity.

7.2. DyLOG *in the context of the literature about agent programming languages*

The language that we used for programming our reasoners and for implementing the adaptive intelligent services provided by our tutoring system is DyLOG, a logic language developed in (Baldoni et al. 2001b; Patti 2002) with the aim of modelling and programming agents with reasoning capabilities. Formalizing rational agents by means of logic languages is one of the main topics of interest in the AI community (Levesque et al. 1997; Hindriks et al. 2001; Herzig and Longin 2002) and recent years have witnessed a growing interest in non-classical logics, such as modal and non-monotonic logics, because of their capability of representing and reasoning about structured and dynamic knowledge. Nonetheless, there is a gap between the expressive power of agent logical models and the practical implementation of agent systems, mainly due to the computational difficulties to verify that properties granted by the models hold also in the implemented systems. Indeed, the leading idea in developing DyLOG was to integrate the expressive capabilities of modal logic and non-monotonic reasoning techniques, within the logic programming framework, in order to define a language which can be used both for specifying and for programming agents, bridging the gap mentioned above.

DyLOG is based on a *modal action theory* that has been developed in (Baldoni et al. 1997, 2001b; Giordano et al. 2000; Patti 2002). The logical framework allows us to deal with complex actions as well as with sensing actions, and to address the most classical reasoning about actions tasks, such as planning, temporal projection and postdiction. In general the framework allows the user to specify the behavior of an intelligent (goal directed or reactive) agent, that chooses a course of actions conditioned on its beliefs on

the environment and that can use sensors and communication for acquiring or updating its knowledge on the real world. The reasoning capabilities supported by the language were essential in the implementation of the adaptive intelligent services provided by our virtual tutor. Moreover, there was a major advantage in using DyLOG in the current work, rather than other languages, developed in the literature for reasoning about dynamic domains and for agent programming, such as GOLOG (Levesque et al. 1997): DyLOG has a sound proof procedure, which practically allows reasoners to perform the planning task in presence of *sensing*. The consequence, in our application framework, is that we can treat the problem of *interactively* generating adapted study plans as a *conditional plan* extraction problem.

The adoption of modal logic in order to tackle reasoning about actions and change, is common to many proposals, such as (De Giacomo and Lenzerini 1995; Prendinger and Schurz 1996; Castilho et al. 1997), and it is motivated by the fact that modal logic allows a very natural representation of actions as *state transitions*. Since the mental attitudes used for describing agents are usually represented as modalities, our modal action theory is also well suited to incorporate such attitudes. The formalization of complex actions draws considerably from dynamic logic, and it refers to a Prolog-like paradigm: complex actions are defined through (possibly recursive) definitions, given by means of Prolog-like clauses. The nondeterministic choice among actions is allowed by defining sets of alternative clauses.

Acknowledgements

We would like to thank prof. Alberto Martelli, Alessandro Chiarotto, Andrea Molia and Laura Torasso for their precious support.

Notes

[1] Technical information about the Wlog system can be found at the URL: `http://www.di.unito.it/~alice`.

[2] 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.

[3] The atom *generic* is used to express that we do not care about which course supplies a given competence.

[4] Currently we do not tackle the problem of building repair plans, aimed at fixing a student-given, wrong study plan.

[5] Thus procedures are deterministic or at most they can implement some "don't care" determinism.

[6] Note that the above formulation of the behaviour of the agent, bears many similarities with agent programming languages based on the BDI paradigm such as dMARS (d'Inverno et al. 1997). As in dMARS, plans are triggered by goals and are expressed as sequences of primitive actions, tests or goals.

References

Baldoni, M., Baroglio, C., Chiarotto, A. & Patti, V. (2001a). Programming Goal-driven Web Sites using an Agent Logic Language. In Ramakrishnan, I. V. (ed.) *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages*, Vol. 1990 of *LNCS*. Las Vegas, Nevada, USA, 60–75. Springer.

Baldoni, M., Baroglio, C., Henze, N. & Patti, V. (2002). Setting up a Framework for Comparing Adaptive Educational Hypermedia: First Steps and Application on Curriculum Sequencing. In *Proc. of ABIS-Workshop 2002: Personalization for the Mobile World, Workshop on Adaptivity and User Modeling in Interative Software Systems*. Hannover, Germany, 43–50.

Baldoni, M., Giordano, L., Martelli, A. & Patti, V. (1997). An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming. In Dix J. et al. (eds.) *Proc. of NMELP'96*, Vol. 1216 of *LNAI*, 132–150. Springer-Verlag.

Baldoni, M., Giordano, L., Martelli, A. & Patti, V. (2001b). 'Reasoning about Complex Actions with Incomplete Knowledge: A Modal Approach. In Restivo, A., Ronchi Della Rocca, S. & Roversi, L. (eds.) *Proc. of Theoretical Computer Science, 7th Italian Conference, ICTCS'2001*, Vol. 2202 of *Lecture Notes in Computer Science*, 405–425. Springer.

Baral, C., McIlraith, S. A. and Son, T. C. (2000). Formulating Diagnostic Problem Solving Using an Action Language with Narratives and Sensing. In *Principles of Knowledge Representation and Reasoning, KR 2000*, 311–322.

Bretier, P. & Sadek, D. (1997). A Rational Agent as the Kernel of a Cooperative Spoken Dialogue System: Implementing a Logical Theory of Interaction. In Müller, J., Wooldridge, M. & Jennings, N. (eds.) *Intelligent Agents III, Proc. of ECAI-96 Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, Vol. 1193 of *LNAI*. Springer-Verlag.

Brusilovsky, P. (2000). Course Sequencing for Static Courses? Applying ITS Techniques in Large-ScaleWeb-Based Education. In *Proceedings of the fifth International Conference on Intelligent Tutoring Systems ITS 2000*. Montreal, Canada.

Brusilovsky, P. (2001). Adaptive Hypermedia. *User Modeling and User-Adapted Interaction* **11**: 87–110.

Carro, R., Pulido, E. & Rodriguez, P. (1999). Dynamic Generation of Adaptive Internet-Based Courses. *Journal of Network and Computer Applications* **22**: 249–257.

Castilho, M., Gasquet, O. & Herzig, A. (1997). Modal Tableaux for Reasoning about Actions and Plans. In Steel, S. (ed.) *Proc. ECP'97*, 119–130.

De Giacomo, G. & Lenzerini, M. (1995). PDL-based Framework for Reasoning about Actions. In *Proc. of AI∗IA '95*, Vol. 992 of *LNAI*, 103–114.

d'Inverno, M., Kinny, D., Luck, M. & Wooldridge, M. (1997). A Formal Specification of dMARS. In *Proc. of ATAL'97*, Vol. 1365 of *LNAI*, 155–176.

Eshghi, K. & Kowalski, R. (1989). Abduction Compared with Negation by Failure. In *Proc. 6th ICLP'89*. Lisbon, 234–254.

FIPA (1997). FIPA 97, Specification Part 2: Agent Communication Language. Technical report, Foundation for Intelligent Physical Agents.

Gelfond, M. & Lifschitz, V. (1993). Representing Action and Change by Logic Programs. *Journal of Logic Programming* **17**: 301–321.

Giordano, L., Martelli, A. & Schwind, C. (2000). Ramification and Causality in a Modal Action Logic. *Journal of Logic and Computation* **10**(5): 625–662.

Henze, N. & Nejdl, W. (1999). Bayesian Modeling for Adaptive Hypermedia Systems. In *Proc. of ABIS99, 7. GI-Workshop Adaptivität und Benutzermodellierung in Interaktiven Softwaresystemen*. Magdeburg.

Henze, N. & Nejdl, W. (2000). Extendible Adaptive Hypermedia Courseware: Integrating Different Courses and Web Material. In Brusilovsky, P., Stock, O. & Strapparava, C. (eds.) *Adaptive Hypermedia and Adaptive Web-Based Systems, International Conference, AH 2000*, 109–120.

Henze, N. & Nejdl, W. (2001). Adaptation in Open Corpus Hypermedia. *IJAIED Special Issue on Adaptive and Intelligent Web-Based Systems* **12**: 325–350.

Herzig, A. & Longin, D. (2002). Sensing and Revision in a Modal Logic of Belief and Action. In van Harmelen, F. (ed.) *Proc. of 15th European Conference on Artificial Intelligence, ECAI 2002*. Lyon, France, 307–311. IOS Press.

Hindriks, K. V., de Boer, F., van der Hoek, W. & Meyer, J. (2001). Agent Programming with Declarative Goals. In Castelfranchi, C. & Lespérance, Y. (eds.) *Intelligent Agents VII. Agent Theories, Architectures and Languages*, Vol. 1986 of *LNAI*, 228–243. Springer-Verlag.

Levesque, H. J., Reiter, R., Lespérance, Y. Lin, F. & Scherl, R. B. (1997). GOLOG: A Logic Programming Language for Dynamic Domains. *J. of Logic Programming* **31**: 59–83.

Macías, J. A. & Castells, P. (2001). Interactive Design of Adaptive Courses. In Ortega, M. & Bravo, J. (eds.) *Computer and Education – Towards an Interconnected Society*, 235–242. Kluwer Academic Publishers.

Murray, T. (2002). MetaLinks: Authoring and Affordances for Conceptual and Narrative Flow in Adaptive Hyperbooks. *International Journal of Artificial Intelligence in Education*, to appear.

Patti, V. (2002). Programming Rational Agents: a Modal Approach in a Logic Programming Setting. Ph.D. thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy. Available at http://www.di.unito.it/∼patti/.

Prendinger, H. & Schurz, G. (1996). Reasoning about Action and Change. A Dynamic Logic Approach. *Journal of Logic, Language, and Information* **5**(2): 209–245.

Sandewall, E. (1994). *Features and Fluents. The Representation of Knowledge about Dynamical Systems*, Vol. I. Oxford University Press.

Scherl, R. & Levesque, H. J. (1993). The Frame Problem and Knowledge-producing Actions. In *Proc. of the AAAI-93*. Washington, DC, 689–695.

Specht, M., Kravcik, M., Klemke, R., Pesin, L. & Hüttenhain, R. (2002). Personalized eLearning and eCoaching in WINDS. In *Proc. of Workshop on Integrating Technical and Training Documentation, ITS 2002*. San Sebastian, Spain.

Specht, M. & Oppermann, R. (1998). ACE – Adaptive Courseware Environment. *The New Review of Hypermedia and Multimedia* **4**: 141–162.

Stern, M. & Woolf, B. (1998). Curriculum Sequencing in a Web-Based Tutor. In *Proc. Of Intelligent Tutoring Systems 1998*, Vol. 1452 of *LNCS*. Springer.

Toni, F. & Kakas, A. (1995). Computing the Acceptability Semantics. *LNAI* **928**: 401–415.

Weber, G. & Brusilovsky, P. (2001). ELM-ART: An Adaptive Versatile System for Webbased Instruction. *IJAIED Special Issue on Adaptive and Intelligent Web-Based Systems* **12**(4): 351–384.