Using a rational agent in an adaptive, web-based tutoring system

Matteo Baldoni, Cristina Baroglio, Viviana Patti, Laura Torasso

Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera 185, 10149, Torino, Italy
Tel.: +39 011 6706711 — Fax.: +36 011 751603
E-mail: {baldoni,baroglio,patti}@di.unito.it

Abstract. In this paper we describe an adaptive tutoring system, having a multi-agent architecture. The kernel of the system is a rational agent, whose behavior is programmed in the logic programming language DyLOG. In the prototype that we implemented the reasoning capabilities of the agent are exploited both to dynamically build study plans and to verify the correctness of user-given study plans with respect to the competences that the user wants to acquire.

Keywords: Adaptive curriculum sequencing, curricula verification, reasoning about actions, multiagent systems.

1 Introduction and motivation

In current days, there is a growing interest in developing web sites and portals where information is presented (and sometimes selected) according to the reader's preferences and interests. E-commerce, on-line news channels, music broadcasting, recommendation systems are a few examples of possible applications. This form of flexibility is commonly referred to as *adaptation*. Many of the most advanced solutions to web site adaptation [1, 2, 11, 10, 17, 9], assume that adaptation should focus on the user's characteristics (age, education, favorite topics), therefore they vary presentations according to a reference prototype, the "user model", to which the user is associated. By doing so, such approaches catch important aspects connected to the personality and the general interests of the user. Sometimes user models are refined according to observations about the users behavior.

In our work we have studied another kind of adaptation, which could be of great help especially in the case of recommendation systems: adaptation based on the user's *intentions* and *needs*. Often the reasons for which users connect to a web site depend on specific needs, which cannot be inferred from the user's past behavior. The user model approach could, therefore, be enriched by adding to the system the ability to reason about intention, belief and action. In this article we describe the most recent achievements of a work, that we have been

doing in the last years by presenting the ways in which our *virtual tutor* supports the definition of study plans.

Intention, as well as belief and action, has intensively been studied in *logics* and *logic programming* settings [18, 14]. In our approach to web site adaptation we use an agent logic programming language, called DyLOG [6, 5], for building cognitive agents, that exploit reasoning techniques for helping users and find ad hoc solutions to their needs. DyLOG is based on a *modal formal theory of actions*, so it can deal with reasoning about action effects in a dynamically changing environment. In this framework, adaptation is interpreted as a *reasoning problem*: thus, for instance, the agent can build a study plan that allows a student to acquire some desired competence, the same agent can also verify whether a study plan proposed by a student is correct (e.g. that course preconditions are respected and the plan will actually allow the student to acquire the desired competence), or diagnose why it is not correct. Independently from the kind of reasoning that is performed, web pages are generated from the system for communicating with the user: in a sense, most of the web site interaction could be considered a form of conversation.

Although our approach could recall some works on Natural Language cooperative dialogue systems, there are some differences. For instance, in 1997 Bretier and Sadek [8] proposed a logic of rational interaction for implementing the dialogue management components of a spoken dialogue system. This work is based, like DyLOG, on dynamic logic but while they exploit reasoning capabilities on actions and intentions to produce proper dialogue acts we use them to produce solutions to the user's problems. In this context, the novelty of our approach is that we exploit reasoning capabilities for building presentation plans guided by the user's goals rather than for dialogue act planning.

See the web site `http://www.di.unito.it/~alice` for technical information about the system.

2 Modeling a virtual tutor as a rational agent: adaptation through reasoning

2.1 The virtual tutor domain

The virtual tutor task is to help students by building (or verifying) study plans, where a study plan is a sequence of courses that the student will attend. Generally speaking, a study plan is supposed to allow a student to acquire a body of coherent and consistent knowledge; for instance, the student will become an expert about "web applications" or an expert about "bioinformatics". Such labels represent the *competence* that will be acquired. This competence, however, is likely to be structured into smaller pieces of knowledge, in different words competence is structured into a *hierarchy of competences*, see Figure 1.

We said that a study plan is a *sequence of courses*. Each course supplies a set of competences; on the other hand, each course can be attended with success only if the student's knowledge satisfies the prerequisites of the course, if any (either

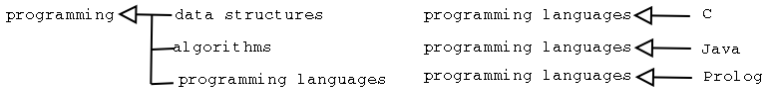

Fig. 1. As an example, this is a little excerpt from our competence hierarchy.

the student attended a set of preparatory courses or he acquired the necessary knowledge in alternative ways). For instance, let's suppose that a programming laboratory course where Java is tought, can be attended if a programming theory course has been attended (*precondition* to the action "attend the Programming Theory course"). When a student attends a course, he will gain competence (*unconditioned effect* of attending the course); some competences, however, may depend on *additional conditions* (*conditioned effect*). For instance, in order to attend our Java lab course it is not necessary to know the C programming language but if I attended a C programming language course I will also be able to understand the explanations about the relationship between C and Java, see Figure 2.

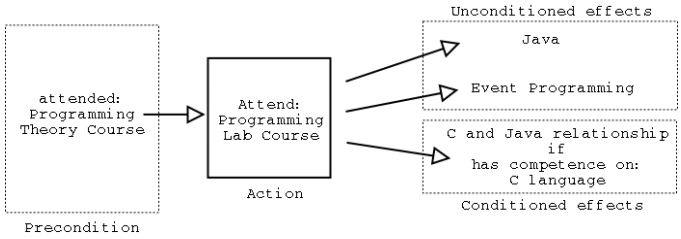

Fig. 2. Attending a course as executing an action.

The use of competence hierarchy-based representations allows a better description of prerequisites. Suppose that a course can be attended by students who know programming languages, independently from the specific course that they attended; it could be attended by students who studied the C programming language, by students who studied Prolog, and by students who studied Java (see Figure 1). Instead, in order to attend a Unix course it would be useful to know the C programming language.

Competence hierarchies can be used also to describe professional expertise independently from the specific courses that are offered and, as we will see, they can be exploited to build flexible study plans, that allow each student to achieve his/her intentions (goals), adapting to the taste and personal interests of each student.

2.2 Adaptation as reasoning about actions

In an *intention-based* adaptive system, *reasoning* is fundamental. In our approach adaptation is intended as an interaction between the user and an intelligent agent aimed at "presenting" and "discussing" study plan proposals. Such proposals are obtained after a reasoning process applied to the domain description (internal to the system) and the specific situations and interests obtained from the user. In the virtual tutoring system, once a student decided the professional expertise or the set of competences he wants to acquire, the system builds plans that lead to the acquisition of the desired knowledge. In general, many alternative study plans would allow the student to reach his goal; in this case, the system must tailor the plan according to the user's preferences. For instance, in the above mentioned case, in which a student should know a programming language in order to attend a certain course, the student will be asked which programming language course he would prefer to attend, among the available ones. Supposing that our student is interested in Artificial Intelligence, he will be likely to choose the Prolog course.

Building study plans is indeed interesting but the most interesting reason for using a reasoner is that it can actually perform *different* kinds of reasoning, based on the same knowledge base. Planning is just one possibility. Remaining on the same application, let's now consider this other situation: a student wrote his study plan, he thinks that by following it he will become a "bioinformatics expert". We still have the "bioinformatics expert" description, the difference w.r.t. the previous case is that now we have a sequence and we want to understand if it will lead the student where he would like to go. A similar case is when a student does not refer to a predefined expertise but collects an own set of desired competences and writes a study plan that is supposed to let him achieve them. Observe that the same kind of reasoning could be exploited by professors, who are defining new study curricula and want to check their consistency.

In order to obtain such a flexible behavior, the core of the system that we developed is a set of *rational agents*. We call our rational agents *reasoners*.

2.3 Implementing the virtual tutor behavior with DyLOG

The language that we used for implementing our reasoners is DyLOG [6, 5]. DyLOG is based on a logical theory for reasoning about action and change in a modal logic programming setting. It allows one to specify a rational agent behavior by defining both a set of simple actions, that the agent can perform (some of which are sensing and suggesting actions for interacting with the user) and a set of Prolog-like procedures, which build complex behaviors upon simple actions. The DyLOG intepreter allows both to execute the procedures, which define the agent behavior, and to reason about their execution, extracting (possibly) conditional plans [3]. The plan extraction process of the interpreter is a straightforward implementation of the proof procedure contained in the theoretical specification of the language.

In our virtual tutor *actions* correspond to *attending courses*. For each "attend course" action, we define its *preconditions* and *effects*: preconditions say when the course can be attended (in terms of passed courses and acquired competences), effects describe the competences that will be acquired. Effects can be further conditioned to more specific requirements (see Figure 2).

As we have seen, *competences* are organized into a set of hierarchies. Hierarchies are built upon sets of *causal implications*. For instance, the hierarchy shown in Figure 1 is represented as:

$has\_competence(programming)$ **if**
$has\_competence(data\_structures) \wedge$
$has\_competence(algorithms) \wedge$
$has\_competence(programming\_languages)$.
$has\_competence(programming\_languages)$ **if**
$has\_competence(c\_language)$.
$has\_competence(programming\_languages)$ **if**
$has\_competence(java\_language)$.
$has\_competence(programming\_languages)$ **if**
$has\_competence(prolog\_language)$.

In the above rules we state that the "programming" competence is achieved if we have competence about data structures, algorithms and we know a programming language (C, Java, or Prolog). *Professional expertise* is formalized as a set of implications that are built upon competence hierarchies. They just add some more level of abstraction.

The behaviour of our agent is described by a collection of procedures.

In the case of *study plan construction* the top level procedure, called *advice*, extracts a plan that will be executed (see Section 3).

(R1) $advice(Plan)$ **isp**
$ask\_user\_preferences \wedge ?requested(Curriculum) \wedge$
$generate\_goal(has\_competence(Curriculum)) \wedge$
$plan(has\_competence(Curriculum) \wedge credits(C)$
$\wedge\, max\_credits(B) \wedge (C =< B)$ **after** $combine\_courses, Plan)$.

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 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 metapredicate that actually builds the plan, in this case by extracting the executions of the procedure *combine_courses* that satisfy the user's goals as well as the further conditions that are specified (e.g. that the number of credits gained with the study plan is not bigger than a predefined maximum). [1]

[1] Note that the above formulation of the behaviour of the agent, has many similarities with agent programming languages based on the BDI paradigm such as dMARS

The way the agent combines the courses into a curriculum is specified by procedure *combine_courses* that, until the study plan is believed complete, tries to achieve the goal of acquiring still missing competences by adding a new course.

(R2) *combine_courses* **isp**
?*study_plan_complete*.
combine_courses **isp**
?(¬*study_plan_complete*) ∧
achieve_goal ∧
combine_courses.

Note that the main goal of having a study plan to propose to the student is reached and the curriculum is complete only when all of the goals of getting some piece of competence are fulfilled. Indeed, we assume the behaviour of a rational agent to be driven by a set of goals, which are represented as fluents[2] having form $goal(F)$. Our agent detects its goals based on student's explicit inputs and its expert competence about learning and courses combination. 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_preference* in (R1):

(R3) *ask_user_preferences* **isp**
$verify\_student\_competence$ ∧
$offer\_curriculum\_type$

$verify\_student\_competence$ allows to acquire knowledge about current curriculum studiorum of the student (if any), whereas $offer\_curriculum\_type$ allows the agent to acquire knowledge about the professional expertise the student would like to achieve. In DyLOG information is gathered from the user by means of special actions, called *sensing actions* [6]. Differently than "normal" actions, they increase (or revise) the knowledge of the agent but they do not change its environment. An example of sensing action is $offer\_curriculum\_type$:

(R4) $offer\_curriculum\_type$ **possible if** *true*.
$offer\_curriculum\_type$ **senses** $requested(Curriculum)$.

After executing $offer\_curriculum\_type$ the value of the fluent $requested(Curriculum)$ (used in (R1)) will be known and the main goal

$$goal(has\_competence(Curriculum))$$

will be inferred; here the actual *goal adoption* occurs. Then, the system will exploit the competence hierarchy-based representation for generating a set of

[12]. As in dMARS, plans are triggered by goals and are expressed as sequences of primitive actions, tests or goals.

[2] In action theory, *fluents* denote properties of the world whose truth value may change over the time.

subgoals that are to be achieved to compose a study plan for the selected curriculum (*generate_goal* in (R1)).

After adopting a goal $goal(F)$, the agent acts so to achieve it until it believes the goal is fulfilled.[3]

(R5) *achieve_goal* **isp**
$?goal(knows(X, \_)) \ \wedge \ (X \neq generic) \wedge$
$add\_course(X)$.
achieve_goal **isp**
$?goal(knows(generic, Keyword)) \wedge$
$?(u(has\_competence(Keyword))) \wedge$
$offer\_course\_on(Keyword) \wedge$
$?course\_on(Keyword, X) \wedge$
$add\_course(X)$.

The procedure *achieve_goal* allows the agent to select in a non deterministic way the goal of adding a course to the specific study plan that is being built. There are two cases. Either the goal requires to add a specific course among the available ones or, more generically, the goal requires to achieve a certain competence, given the student did not acquire it yet ($u(has\_competence(Keyword))$). In the latter case, the agent interacts again with the user to decide what specific course to add according to the user's preference ($offer\_course\_on$).

(R6) $offer\_course\_on(\_)$ **possible if** $true$.
$offer\_course\_on(Keyword)$ **suggests** $course\_on(Keyword, \_)$.

Note that $offer\_course\_on$ is a *suggesting action*. In [3] we introduced suggesting actions as a special case of sensing actions. As a difference with normal sensing actions, when performing this kind of actions the agent does not read an input in a passive way but it has an active role in selecting, after some reasoning, the possible values among which the user will make his choice. In particular only those values that lead to fulfill the goal will be selected. Figure 3 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. The other branches are cut during the reasoning phase and the corresponding courses are not offered to the student.

Formally, the above described kind of reasoning is called *temporal projection*. Given the description of a domain and an initial situation (e.g. the student intial competences), the temporal projection task consists in predicting the future effects of actions on the basis of (possibly incomplete) information on preceding states. Briefly, see [6] for details, we formalize the *temporal projection problem* "given an action sequence $a_1, \ldots, a_n$, does the condition *Fs* hold after the execution of the actions sequence starting from the initial state?" by the query *Fs*

[3] This corresponds to adopt a *blind commitment strategy*.

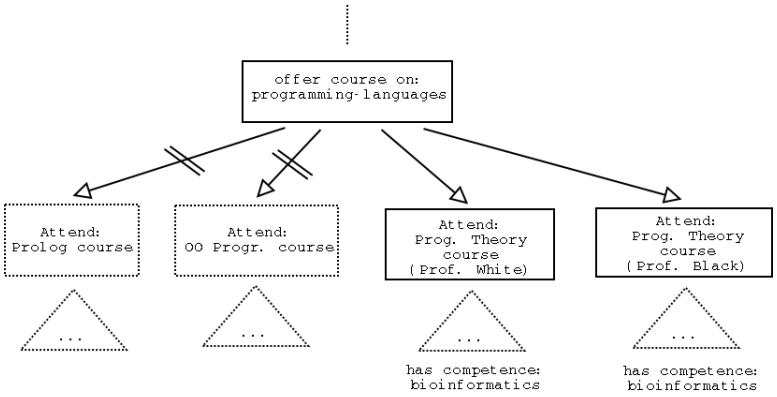

Fig. 3. An example of selection of courses to offer for consructing a bioinformatics curriculum.

after $a_1; \ldots; a_n$, where *Fs* is a conjunction of *fluents*. We can generalize this query to complex actions (procedures) $p_1, p_2, \ldots, p_n$ by:

$$Fs \textbf{ after } p_1; p_2; \ldots; p_n \tag{1}$$

where p_i, $i = 1, \ldots, n$, is either an atomic action (including sensing actions), or a procedure name, or a test. Query (1) succeeds if it is possible to find a (terminating) execution of $p_1; p_2; \ldots; p_n$ leading to a state where *Fs* holds. Intuitively, when we have a query *Fs* **after** p we look for those *terminating execution sequences* which are plans to bring about *Fs*. In this way we can formalize the *planning problem*: "given an initial state and a condition *Fs*, is there a sequence of actions that (when executed from the initial state) leads to a state in which *Fs* holds?". The procedure definitions constrain the search space of reachable states in which to search for the wanted sequence. In the virtual tutor application, the procedure p is "combine_courses" while *Fs* is the condition $has\_competence(Curriculum) \wedge credits(C) \wedge max\_credits(B) \wedge (C =< B)$.

The second kind of reasoning that we implemented is verifying whether a student-given plan will allow him to achieve some desired competence. In this case we still adopt temporal projection but in its simpler form: we have a sequence of actions and a condition that should hold after their execution. In the virtual tutor application, the sequence of actions is a sequence of courses to attend and the final condition is a set of desired competences: $has\_competence(c_1) \wedge \ldots \wedge has\_competence(c_n)$ **after** $attend(course_1); \ldots; attend(course_m)$:

(R2) $check\_plan(Plan, Competences)$ **isp**
$plan((Competences)$ **after** $(Plan), \_)$.

3 The virtual tutor as a multiagent system

WLog, the prototype system that we developed, has a *multi-agent system architecture*. Agent technology allows complex systems to be easily assembled by

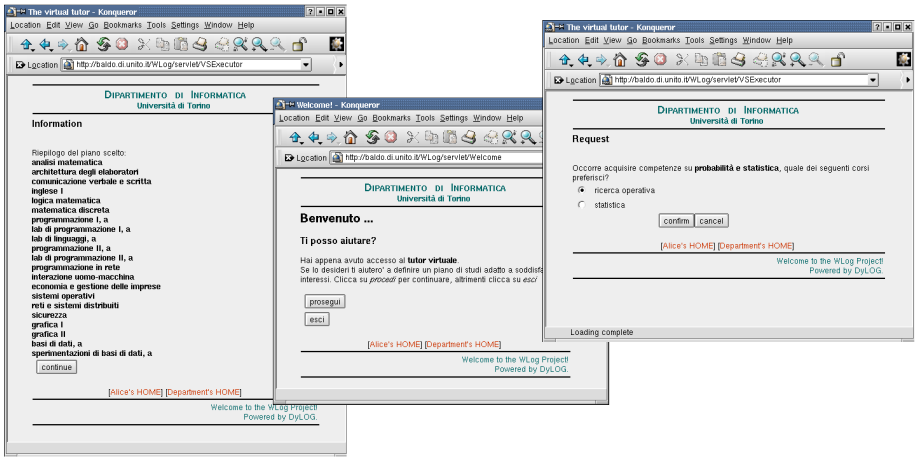

Fig. 4. Interacting with WLog.

means of the creation of distributed artifacts able to accomplish their tasks through cooperation and interaction. Systems of this kind have the advantage of being modular and, therefore, flexible and scalable. So, on one hand, each module can be developed by exploiting the best, specific technology for solving a given issue, 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*[4]. We already know that reasoners are written in DyLOG, executors, instead, are Java servlets embedded in a Tomcat web server.

In the case of study plan extraction, the reasoner task is to extract a conditional plan. Such a plan is a tree, where *each path* to a leaf is actually a study plan that will allow the student to acquire the desired competence. Branches represent alternative courses to acquire a same piece of competence. The execution of the conditional plan is aimed at helping the student to choose the path that better satisfies his/her personal interests. So the student will obtain a correct study plan that is also adapted to his preferences.

The interaction between a user and a reasoner is carried on by means of an *executor*. The task of an executor consists in showing one or more web pages to the user; in particular, when a page that corresponds to a branching point is shown, a feedback from the user is requested.

The communication among the agents has the form of message exchange in a distributed system; message exchange is FIPA-like [13]. Each agent is identified by its "location", which can be obtained by other agents from a facilitator. Each agent has a private mailbox where it receives messages from other agents (see Figure 5).

[4] For a more detailed description of the system's architecture we refer to [4].

Fig. 5. A sketch of the WLOG multiagent system.

3.1 Interaction between a tutor, an executor, and a user

Users access to the system by means of a normal web browser; until the end of the interaction, the interface between the user and the reasoner will be an *executor*. First the executor looks for a free reasoner, if any is available. At the moment reasoners are not differentiated (they all can perform the different kinds of reasoning).

Supposing that the previous step was successful, the interaction between the user and the system starts with the declaration of the user's goal ("I want to become an expert of bioinformatics"). The user's goal is adopted by the reasoner, that will start a conversation aimed at collecting information about the user initial situation. For instance, 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 evaluate and the competences to acquire. The leader of the conversation is the reasoner, which will send the information or the questions to the user with the help of the executor. At this point it is extremely interesting to understand how the interaction between the reasoner and the executor is carried on.

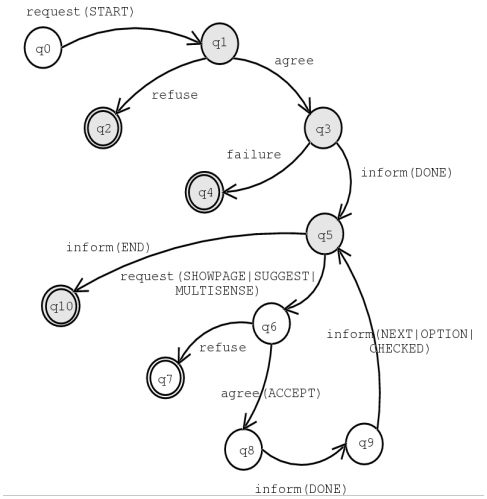

Fig. 6. Communication protocol between an executor and a reasoner.

In Figure 6, the interaction that occurs between a reasoner and an executor is specified by using a finite state automaton. Such an automaton represents the *interaction protocol* that is respected by the couples <reasoner, executor>. States are numbered and arcs are labeled with the speech act that causes the transition between states. Different shading on states are used for specifying which agent's turn it is to continue the conversation (white for the executor, grey for the reasoner). States with double border are terminating states.

The q_1-q_4 states concern the initialization phase for connecting a certain executor with a reasoner. The q_5-q_{10} states concern the *actual action execution cycle*. They state that when a reasoner executes an action (which can be part of a plan extracted after a reasoning process, or simply an atomic action in a DyLOG procedure that is being executed) the execution code associated to the action sends to the executor a request of showing a given HTML page. Both agents continually check the sender of the messages that they receive, so if, for instance, an executor receives a message from a reasoner which is not serving its user, it will refuse that execution. The same would happen if it were asked to perform an action that it is not supposed to perform in the current state. For the sake of clarity, suppose that the executor has just sent a form to the user's browser and is waiting for data. If meanwhile it receives from the reasoner the request to show another page, it will refuse to do it.

Supposing that the executor accepted to perform a requested action, it composes a proper HTML page and sends it to the user's browser. In some cases the page will contain a form to be filled. When the user finishes to consult/fill the page and asks to go on, the executor informs the reasoner that the page has been consulted. When requested, it also transmits to the reasoner the user's data, that can be used by the reasoner to update its knowledge about the user's goals (or background information); otherwise it only informs that it is possible to go on with the next action if there is any.

4 Conclusion and future work

In this paper, we have presented a web-based virtual tutor whose kernel is a reasoner, implemented in a logic programming language to reason about action and change, DyLOG. In such an approach we have, on one hand, a high-level description of the domain (e.g. the connection between two competences can be represented by means of a causality relationship), that is close to our intuition and the way in which we, human beings, handle this sort information. The advantage is that we can maintain or modify these descriptions in an easy way. On the other hand, the system interacts with the user in order to find out his goals in the context of the application framework and applies various kinds of reasoning, implemented by means of a logic derivation mechanism, for finding a solution that perfectly fits the specific user.

In our implementation, both planning (constructing a study plan) and verifying a student-given study plan are performed on-line. In the case of planning we could have followed an alternative approach: to build off-line the most gen-

eral conditional plan for each professional expertise and to prune (on-line) the resulting tree during the interaction with the user. This cannot be done efficiently neither in the case in which the user asks to build a plan for achieving a generic set of competences nor in the case of plan verification. In fact, finding out if a sequence is an instance of a schema by pruning the schema tree has a high computational complexity whereas verifying its correctness by applying *temporal projection*, that is to verify whether the linear plan is compatible with the domain description, is linear in the number of the elements in the sequence.

In the case in which the study plan that the system verified resulted to be wrong, it would be extremely useful to find out the reasons for which it is wrong, so to better help the user. Basically the reasons for which a study plan is wrong are of two kinds: either the sequentialization of courses is not correct or in the end the student will not achieve the desired competence. We are currently working on the application of other kinds of reasoning (diagnosis by abduction and postdiction) to achieve this goal in the line of [7] where a formal account of diagnostic problem solving is provided in term of an action language.

Adaptation in e-learning is a open problem and is being tackled by many researchers under many perspectives. In the terminology of works like [15, 16], our work focuses on "knowledge dependencies", separating them from the actual knowledge. As a difference, while works like [15] are based upon a partial order of "knowledge items" and Bayesian networks, we base our work on a *modal logic theory of action*. In the former knowledge dependencies of the form $K1 < K2$ express the fact that $K1$ should be learned in order to learn $K2$. Therefore, the transitive closure of the "$<$" relation is the inferencing mechanism that allows to the system to understand the dependencies between sets of knowledge items. In our case, we identify both a set of "knowledge items" (the courses) and a set of *hierarchical competences*. Competences are abstract descriptions about knowledge, which are not necessarily contained in the index (or program) of the course. In our course descriptions we have prediconditions to attending the course (they can consist of competences as well as of other courses) and (un)conditioned effects of attending the course. The dependencies between sets of "knowledge items"/competences are based on a *logic derivation* [6].

References

1. L. Ardissono and A. Goy. Tailoring the interaction with users in electronic shops. In *Proc. of the 7th Int. Conf. on User Modeling*, 1999.
2. L. Ardissono, A. Goy, G. Petrone, and M. Segnan. A software architecture for dynamically generated adaptive web stores. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence, IJCAI'01*, Seattle, Oregon, USA, 2001.
3. M. Baldoni, C. Baroglio, A. Chiarotto, and V. Patti. Programming Goal-driven Web Sites using an Agent Logic Language. In I. V. Ramakrishnan, editor, *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 60–75, Las Vegas, Nevada, USA, 2001. Springer.
4. M. Baldoni, C. Baroglio, and V. Patti. Structereless, Intention-guided Web Sites: Planning Based Adaptation. In *Proc. 1st International Conference on Universal*

Access in Human-Computer Interaction, a track of HCI International 2001, New Orleans, LA, USA, 2001. Lawrence Erlbaum Associates.
5. M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Modeling agents in a logic action language. In *Proc. of the Workshop on Rational Agents, FAPR'00*, London, September 2000.
6. M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Reasoning about complex actions with incomplete knowledge: a modal approach. In *Proc. of the 7th Italian Conference on Theoretical Computer Science, (ICTCS01)*, volume 2202 of *LNCS*, Torino, Italy, October 2001.
7. Chitta Baral, Sheila A. McIlraith, and Tran Cao Son. Formulating diagnostic problem solving using an action language with narratives and sensing. In *Principles of Knowledge Representation and Reasoning, KR 2000*, pages 311–322, 2000.
8. P. Bretier and D. Sadek. A rational agent as the kernel of a cooperative spoken dialogue system: implementing a logical theory of interaction. In *Intelligent Agents III, proc. of ECAI-96 Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, LNAI, 1997.
9. P. Brusilovsky. Adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 11:87–110, 2001.
10. B. De Carolis, S. Pizzutilo, and F. de Rosis. User manuals as animated agents. In *Proc. of the AI*IA Workshop su "Agenti intelligenti e internet: teorie, strumenti e applicazioni"*, Milano, Italy, 2000.
11. B.N. De Carolis. Introducing reactivity in adaptive hypertext generation. In *Proc. 13th Conf. ECAI'98*, Brighton, UK, 1998.
12. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Proc. of ATAL'97*, volume 1365 of *LNAI*, pages 155–176, 1997.
13. FIPA. FIPA 97, Specification part 2: Agent Communication Language. Technical report, Foundation for Intelligent Physical Agents, 1997.
14. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.
15. N. Henze and W. Nejdl. Bayesian modeling for adaptive hypermedia systems. In *Proc. of ABIS99, 7. GI-Workshop Adaptivität und Benutzermodellierung in Interaktiven Softwaresystemen*, Magdeburg, September 1999.
16. Nicola Henze and Wolfgang Nejdl. Extendible adaptive hypermedia courseware: Integrating different courses and web material. In Peter Brusilovsky, Oliviero Stock, and Carlo Strapparava, editors, *Adaptive Hypermedia and Adaptive Web-Based Systems, International Conference, AH 2000*, pages 109–120, 2000.
17. L. Marucci and F. Paternò. Designing an adaptive virtual guide for web application. In *Proc. 6th ERCIM Workshop UI4All*, Florence, Italy, 2000.
18. R. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *Proc. of the AAAI-93*, pages 689–695, Washington, DC, 1993.