Supporting Users in Adaptive Web-based Applications: Techniques from 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)
Tel. +39 011 6706711 — Fax. +39 011 751603
E-mail: {baldoni,baroglio,patti}@di.unito.it

***Abstract*—In this paper we discuss how different reasoning techniques can be applied to the development of adaptive web applications. We will describe the reasoning mechanisms that are used by an agent that is implemented in DyLOG, a logic programming language, whose main characteristic is to allow reasoning about actions in a dynamically changing environment. The applicative domain is curriculum sequencing for education. In this framework the reasoning mechanisms will be used for constructing study plans, validating user-given study plans, and, in case these are wrong, to explain why they are not correct.**

I. Introduction and motivation

Recent years witnessed a growing interest in the development of web sites where the information is selected and presented in a personalized way, so to fit each single user as much as possible. In this sense, the web site *adapts* to the user. Most of the solutions to web site adaptation rely on the user model approach [1], [2], [13], [12], [20], [10]. In this case each user is associated to a reference prototype which is tuned according to the user's behavior. Nevertheless, there are situations in which a different approach to adaptation is required. For example, in the case of recommendation systems, adaptation should also take into account the user's *intentions* and *needs*, which are usually related to the specific connection and cannot be inferred from the past user's behavior or from his/her general model. Therefore, the adaptive system should be enriched by allowing it to reason about the user's goals and intentions.

Let us focus on the case of *curriculum sequencing*, a specific adaptation technique in the area of intelligent tutoring systems and adaptive hypermedia [11]. *Curriculum sequencing* is an adaptive method for building reading or study paths (also known as trails) in a hyperspace of knowledge units and different approaches have been proposed on how to determine which path to generate in order to support the single user in an optimal way [9], [25], [23], [16]. In our work we dealt with a specific instance of the curriculum sequencing problem, i.e. the problem of building study plans for computer science students ("Laurea di Primo Livello"). In this context, a student is typically interested in acquiring some specific professional expertise; the task of the system is to help him/her to find a proper solution, i.e. a list of courses compiled taking into account both the student learning goals and some learning dependencies fixed by professors. Some students start from scratch, others, instead, want to change their study plan after attending one or two years, others propose an own study plan which is to be validated. In all these cases, the user model alone is not sufficient, but it can be useful for the adaptive tutoring system to be capable of reasoning about the user's learning goals and about the effect of the action of attending a given course on the user's knowledge.

Reasoning about actions and attitudes (i.e. goals and knowledge) is among the problems that are typically tackled by rational agents, especially in a logic and logic programming setting [22], [21], [19], [17], [8]. Our approach is to implement the adaptive services in our tutoring system by exploiting the deductive reasoning capabilities of a logic agent. Such an agent is meant as a *virtual tutor* that guides the interaction with the user, keeps a representation of the user's learning goal and current knowledge, and uses various reasoning techniques from the reasoning about actions field - such as planning, temporal projection and temporal explanation - for supporting him/her in the definition of a study plan for achieving the desired goal.

Starting from 1996 in our research group the problem of reasoning about actions and change was tackled leading to the design and implementation of a logic language for programming software agents: DyLOG [7], [8]. The language is based on a modal approach for reasoning about actions in a logic programming setting. It allows 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 external world) and a set of Prolog-like procedures, which build complex behaviors upon simple actions. The advantage of DyLOG is that its interpreter allows both to execute the procedures, which define the agent behavior, and to reason about their execution, extracting (possibly) conditional plans. The plan extraction process of the interpreter is a straightforward implementation of the proof procedure contained in the theoretical specification of the language. Besides planning, also the well-known temporal projection task is supported. Thanks to these features DyLOG has been successfully used for pro-

gramming the behavior of our virtual tutor. In fact, as we started to show in [4], [5], curriculum sequencing problems can be interpreted as reasoning about action problems. The reasoning process is applied to the domain description (internal to the system) and to the specific situations and interests obtained from the user.

In this paper we will present the *various* reasoning techniques that we applied to curriculum sequencing problems. In particular, we will sketch curriculum construction by means of planning techniques and, as a novelty with respect to previous work, we will describe an approach to curriculum validation where temporal projection and temporal explanation capabilities are exploited. In this latter case, if a curriculum is wrong, the system uses temporal explanation for giving to the user some feedback about the failure of the validation process.

In order to validate our approach we developed a prototype system, called WLog, which actually offers the tutoring services described above. The system has a multi-agent architecture, whose kernel is a set of rational agents implemented as DyLOG programs which control the adaptation to the user.

In the next section we introduce curriculum sequencing issues together with a description of the domain knowledge. In Section III we discuss the various reasoning mechanisms that we have used and show how we applied them to build adaptive web applications. A brief description of the multi-agent prototype system that we developed, WLog, follows.

II. The application domain

Curriculum sequencing techniques are used to handle a corpus of *information sources*, each information source being an atomic unit of information [3]. The granularity and the type of such units depend on the specific application domain and can widely vary. An atomic unit of information may, for instance, be a definition, a page, a book, a web site, the common characteristic being that it will in any case be considered as a single unstructured object. In our case, it is a *course description*.

Each information source is described by some meta-data. In particular, each unit has a set of outcomes, i.e. the pieces of knowledge that are acquired by attending that course, and a set of *prerequisites*. Prerequisites are the (knowledge) constraints that are to be satisfied so that the user can understand the information contained in the unit. We will call all the above mentioned pieces of knowledge *competences*.

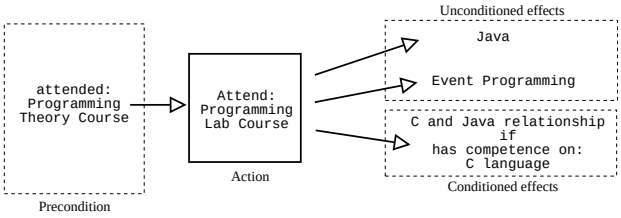

Fig. 1. Attending a course as executing an action.

As an example, consider a programming laboratory course where Java is taught, which can be attended only if a programming theory course has already been attended (*precondition* to the action "attend the Programming Theory course", see Figure 1). When a student attends this course, he/she will acquire competence about Java and Event Programming (*unconditional effect* of attending the course). Some competences, however, may depend on *additional conditions* (*conditional effect*). For instance, in order to attend our Java lab course it is not necessary to know the C programming language but if the student attended a C programming language course he/she will also be able to understand the explanations about the relationship between C and Java, see Figure 1.

In our approach competences are described separately and define an ontology, which is a *knowledge model*. Notice that, having an ontology is particularly interesting in open systems where there is a need of a common vocabulary. The association between the units and the concepts could either be done by hand or automatically by exploiting information retrieval techniques. Knowledge models can be defined in different ways by describing the relations among competences. We represent the knowledge model as a hierarchy. The knowledge model together with the associations of prerequisites and outcomes in each course defines a set of *learning dependencies*.

In curriculum sequencing applications, each user wants to acquire part of the information contained in the corpus; for this reason we will say that each user has a *learning goal*. Whenever the domain representation contains a knowledge model the learning goals can be expressed in terms of concepts that the user wants to acquire (competences).

III. Interpreting the tutoring adaptive services as reasoning about action tasks

The main target of action theories is to use a logical framework to describe the effects of actions on a dynamic world where *all* changes are caused by action execution. In general, a formal theory for representing and reasoning about actions allows us to specify:

(a) *causal laws*, i.e. axioms that describe domain's actions in terms of their preconditions and effects on the fluents;
(b) action sequences that are executed from the initial state;
(c) *observations* describing the fluent's value in the *initial state*;
(d) *observations* describing the fluent's value in later states, i.e. after some action's execution.

The term *domain description* is commonly used to refer to a set of propositions that express causal laws, observations on the fluents value in a state and possibly other information for formalizing a specific problem.

Given a domain description, the principal reasoning tasks are *temporal projection* (or prediction), *temporal explanation* (or postdiction) and *planning*. Intuitively, the aim of *temporal projection* is to predict action's future effects based on even partial knowledge about current state (reasoning from causes to effects), while the target of *temporal explanation* is to infer something on the past states of the world by using knowledge about the actual situation. The third reasoning task, *planning*, 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.

A. Interpreting courses as actions

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

A *state* consists of a set of *fluents*, i.e. properties whose truth value may change over the time. In general we cannot assume that the value of each fluent in a state is known to an agent, and we want to be able of representing the fact that some fluents are unknown and to reason about the execution of actions on incomplete states. To represent explicitly the unknown value of some fluents, in [8] we introduce an epistemic level in our representation language. In particular, we introduce an epistemic operator $\mathcal{B}$, to represent the beliefs an agent has on the world: $\mathcal{B}f$ will mean that the fluent f is known to be true, $\mathcal{B}\neg f$ will mean that the fluent f is known to be false. Fluent f is undefined in the case both $\neg\mathcal{B}f$ and $\neg\mathcal{B}\neg f$ hold. We will write $u(F)$ for $\neg\mathcal{B}f \wedge \neg\mathcal{B}\neg f$. In our implementation of DyLOG (and also in the following description) we do not explicitly use the epistemic operator $\mathcal{B}$: if a fluent f (or its negation $\neg f$) is present in a state, it is intended to be believed, unknown otherwise. Thus each fluent can have one of the three values: true, false or unknown. We use the notation $u(F)?$ to test if a fluent is unknown (i.e. to test if neither f nor $\neg f$ is present in the state).

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

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

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

In the curriculum sequencing system that we have implemented, each course is interpreted as the action of attending that course. As an example, consider the "Programming Lab" course , described in Figure 1. In DyLOG it can be represented as:

add(course('programming lab')) :
"add course *programming lab* to the current curriculum"
(1) $add(course('programming\ lab'))$ **possible if** $knows('programming\ theory')$.
(2) $add(course('programming\ lab'))$ **causes** $has\_competence('java')$.
(3) $add(course('programming\ lab'))$ **causes** $has\_competence('event\ programming')$.
(4) $add(course('programming\ lab'))$ **causes** $has\_competence('C\ and\ java\ relationship')$ **if** $?has\_competence('C\ language')$.
(5) $add(course('programming\ lab'))$ **causes** $credit(B1)$ **if** $get\_credits('programming\ lab', C) \wedge credit(B) \wedge (B1\ is\ B + C)$.

Rule (1) states that the action $add(course('programming\ \ lab'))$ can be executed if the programming theory course has already been added to the curriculum (or if the student already attended that course). Action laws (2)-(3) describe the unconditional effects of the action execution: adding the programming lab course causes to have competence about Java and event programming. Action law (4) describes the conditional effect of the action at issue. Finally, action law (5) updates the current credits by summing those related to the programming lab course.

B. Building personalized curricula by procedural planning

In [4], [5] we interpreted the problem of curriculum construction as a *procedural planning* problem. In this section we briefly sketch the proposed solution.

Besides simple actions, in DyLOG it is also possible to define *complex actions*, or procedures. Complex actions are defined on the basis of other complex actions, primitive actions, *sensing* actions and *test* actions. Test actions are needed for testing if some fluent holds in the current state and for expressing conditional complex actions. They are written as "$(Fs)?$", where Fs is a fluent conjunction. Sensing actions, instead, have the form "s **senses** F" and allow the agent to interact with the external world to determine the value of a fluent F, rather than to change it. Such a value may either be true or false or it may belong to a finite set of alternatives. A *procedure* is defined as a collection of *procedure clauses* of the form:

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

where p_0 is the name of the procedure and p_i, $i = 1, \ldots, n$, is either a primitive action, or a sensing action, or a test action, or a procedure name (i.e. a procedure call)[2]. Procedures can be recursive and can be executed in a goal directed way, similarly to standard logic programs. From the logical point of view procedure clauses have to be regarded as axiom schemas of the

[1] In a logic programming context we represent causality by the directionality of implication. A more general solution, which makes use of modality "causes", has been provided in [18].

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

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 an 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 to acquire *professional expertise*. For instance, in the following we report those procedures that describe how to acquire the body of competences necessary for a *web applications* curriculum:

$achieve\_goal(has\_competence(curriculum($
$\quad 'web\ applications'))))$ **is** ...
...
$achieve\_goal(has\_competence('programming'))$ **is**
$\quad achieve\_goal(has\_competence('datastructures')) \wedge$
$\quad achieve\_goal(has\_competence('algorithms)) \wedge$
$\quad achieve\_goal(has\_competence($
$\quad\quad 'programming\ languages')).$

$achieve\_goal(has\_competence($
$\quad 'programming\ languages'))$ **is**
$\quad achieve\_goal(has\_competence('c')).$
$achieve\_goal(has\_competence($
$\quad 'programming\ languages'))$ **is**
$\quad achieve\_goal(has\_competence('java').$
$achieve\_goal(has\_competence($
$\quad 'programming\ languages'))$ **is**
$\quad achieve\_goal(has\_competence('prolog')).$

Note that procedure definitions can be non-deterministic, as in Prolog.

In general, a *planning problem* amounts to determine, given an initial state, if there is a sequence of actions that, when executed in the initial state, leads to a goal state in which Fs holds. In our context, we consider a specific instance of the planning problem in which we want to know if there is a possible execution of a procedure p leading to a state in which some condition Fs holds. In such a case the execution sequence is not an arbitrary sequence of atomic actions but it is an execution of p. In other words, the procedure definitions *constrain* the space in which the desired sequence is sought for. This can be formulated by the query $\langle p \rangle Fs$, which asks for a terminating execution of p (i.e. a finite action sequence) leading to a state in which Fs holds. In DyLOG this query is written as: "Fs **after** p". The execution of this query returns as a side-effect an *execution trace*, which is a sequence of primitive actions that leads from the initial to the final state and it corresponds to a *linear plan* (or a *conditional* plan in case the program contains sensing actions). For instance, given a set of desired competences Fs the student may be interested to know which study plan he/she should follow within the family of "web applications" study plan for acquiring the Fs competences. The corresponding query is:

Fs **after** $achieve\_goal(has\_competence('web\ applications'))$

C. Validation of user-built curricula by temporal projection w.r.t. competences

In [5] we interpreted the problem of verifying whether a student-given plan will allow him/her to achieve some desired competence as a *temporal projection* task. Briefly, see [8] for details, in DyLOG 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 action sequence starting from the initial state?" by means of the query:

$$\langle a_1, \rangle \ldots \langle a_m \rangle Fs$$

where Fs is a conjunction of fluents [3]. If the sequence of actions is a sequence of courses to attend and the final condition Fs is a set of desired competences, such query can be naturally read as "Given the initial student background, does the student achieve the set of competences Fs after attending the course sequence $c_1, \ldots, c_n$?". The corresponding DyLOG query is:

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

where Fs is the set of desired competences.

D. Explanation of validation failure by temporal explanation

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 or because the sequencing does not respect some learning dependencies (i.e. the student does not have the necessary competence for attending the next course in the plan). Some feedback about the reasons of failure will support the user in modifying the wrong plan. A first feedback can be given by showing the student the list of the competences he/she should already have for the plan to be valid. This task can be interpreted as a *temporal explanation* task. In order to deal with temporal explanation, we adopt an *abductive approach* in the line of [6], by determining the assumptions on the initial state that are needed for explaining observations on later states. In the case of courses, they will be the competences that the student is supposed to have in the initial state in order to make the study plan applicable.

While the reasoning mechanisms of planning and temporal projection, used in sections III-C and III-B, are based on the proof procedure described in [8], *temporal explanation* is based on the work contained in [6]. 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 language. Given a query:

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

($G = \langle add(course(c_1)) \rangle \ldots \langle add(course(c_n)) \rangle Fs$), and a domain description Π, *support(*G*,*Π*)* returns an abductive support

[3] Notice that, since primitive actions defined in a DyLOG 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 actions a defined in the domain description, and then, the success of the existential query $\langle a_1 \rangle \ldots \langle a_n \rangle Fs$ entails the success of the universal query $[a_1] \ldots [a_n] Fs$.

for the query G in Π, that is, a set Δ of abducibles[4] such that

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

(see footnote 3). In this way, 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 [14], and is similar to the procedures proposed in [24] to compute the acceptability semantics. In this work, for the sake of brevity, we report only the support procedure, that was modified in the following way, while the abductive procedure remained unchanged (see [6], Section 4).

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

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 $\langle a_1 \rangle \dots \langle a_m \rangle Fs$ can be derived from a domain description Π with assumptions Δ if, using the rules above, we can derive $\varepsilon \vdash_{vs} [a_1] \dots [a_m]Fs$.

This failure explanation mechanism is, indeed, quite simple and it works because in the particular domain that we are tackling competences can only be added (new courses are not supposed to erase from the students' memory the concepts acquired in previous courses). Currently we have a prototype implementation of this service, that returns a subset of the possible abductive explanations that the abductive procedure would return (which is on its way of being implemented).

IV. The Architecture

Agent technology allows complex systems to be easily assembled by 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. For these reasons the prototype system that we developed, WLog (see [5] for details), has a *multi-agent* system architecture that is sketched in Figure 2.

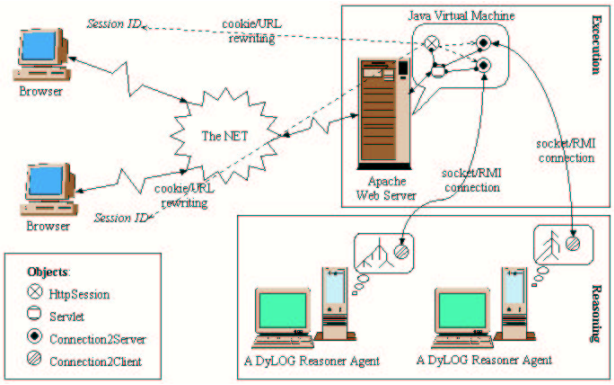

Fig. 2. A sketch of WLog architecture.

The WLog architecture mainly consists of two kinds of agents: *reasoners* and *executors*. Reasoners are the actual kernel of the system, they are written in DyLOG and perform the different kinds of reasoning that we have described so far. Executors carry on the interaction between a user and a reasoner. They are Java servlets which both display the information and the requests produced by a reasoner as web pages and return the user's choices to the reasoner.

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. The communication among the agents has the form of message exchange in a distributed system; message exchange is FIPA-like [15].

Users access to the system by means of a 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 [5], if any is available. If one is available, from that moment till the end of the connection that reasoner will be dedicated to serve that specific user.

Supposing that the previous step was successful, the interaction between the user and the system starts with the declaration of the user's goal (for instance "I want to become an expert in 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, 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 in input the sequence of courses to evaluate and the competences the student expect to acquire by attending such sequence. In this phase 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 interesting to understand how the interaction between the reasoner and the executor is carried on. In

[4] Abducibles are atomic propositions of the form $\mathbf{M}[a_1] \dots [a_n]F$. Notice that $\mathbf{M}$ is not to be regarded as a modality, this notation has been adopted in analogy to default logic and $\mathbf{M}[a_1] \dots [a_n]F$ means that F is consistent after the execution of the sequence of action $a_1, \dots, a_n$.

[5] Note that at the moment reasoners are not differentiated but they all can deal with the various reasoning tasks.

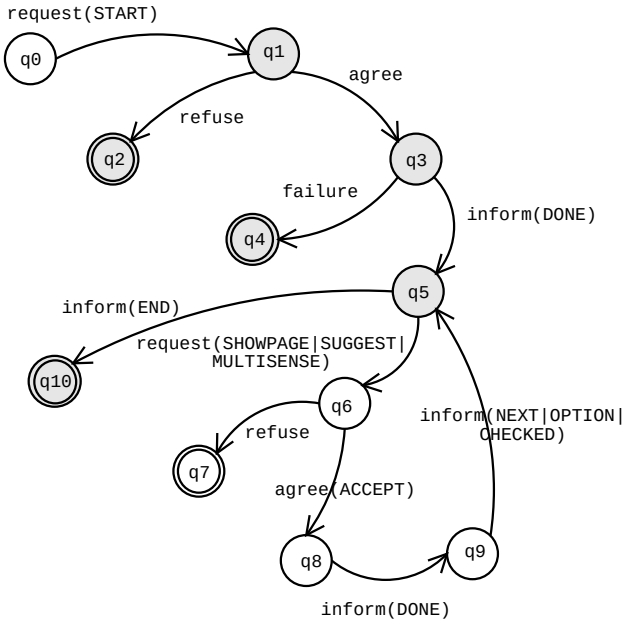

Fig. 3. Communication protocol between an executor and a reasoner.

Figure 3, 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 labelled with the speech acts that cause the transition. Different shading on states are used for specifying which agent will 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.

Notice that both agents continually check the sender of the messages that they receive. 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 be performed 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.

Assuming 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 (or fill) the page and asks to go on, the executor informs the reasoner that the page has been consulted. Thus, 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.

V. Conclusion and Future Work

In this paper we have shown how different reasoning techniques can be used for achieving adaptation in web applications. We have applied the proposed mechanisms to curriculum sequencing problems and, more specifically, to study plan construction and validation. We have also presented a solution to the explanation of validation failure for the applications at issue, that exploits temporal explanation. Nevertheless, we claim that the same approach can be used also in different domains, such as for building intelligent help on line systems and for hyperbook navigation.

In future work we mean to continue this study. In particular, we are planning to extend the actual framework in order to enhance the failure explanation mechanism. The idea is to make it possible to return to a student who submitted a wrong study plan, not only a feedback on the reasons of the validation failure, but also a *repaired plan*, that is a corrected version of the wrong student-given plan.

Acknowledgement

The authors thank *Laura Torasso* for her precious help and work.

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, N. Henze, and V. Patti. Setting up a framework for comparing adaptive educational hypermedia: First steps and application on curriculum sequencing. In *Proc. of the ABIS-Workshop 2002, Personalization for the Mobile World, Workshop on Adaptivity and User Modeling in Interative Software Systems*, pages 43–50, Hannover, Germany, October 2002.

[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*, volume 3, pages 237–241, New Orleans, LA, USA, 2001. Lawrence Erlbaum Associates.

[5] M. Baldoni, C. Baroglio, V. Patti, and L. Torasso. Using a rational agent in an adaptive web-based tutoring system. In P. Brusilovsky, N. Henze, and E. Millan, editors, *Proc. of the Workshop on Adaptive Systems for Web-Based Education, 2nd Int. Conf. on Adaptive Hypermedia and Adaptive Web-based Systems*, pages 43–55, Malaga, Spain, 2002.

[6] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming. In J. Dix et al., editor, *Proc. of NMELP'96*, volume 1216 of *LNAI*, pages 132–150. Springer-Verlag, 1997.

[7] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Modeling agents in a logic action language (extended abstract). In *Proc. of Workshop on Practical Reasoning Agents, FAPR2000*, 2000.

[8] 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*, pages 405–425, Torino, Italy, October 2001.

[9] P. Brusilovsky. Course sequencing for static courses? Applying ITS techniques in large-scale web-based education. In *Proceedings of the fifth International Conference on Intelligent Tutoring Systems ITS 2000*, Montreal, Canada, 2000.

[10] P. Brusilovsky. Adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 11:87–110, 2001.

[11] Peter Brusilovsky, Nicola Henze, and Eva Millan. *Adaptive Systems for Web-Based Education*. Universidad de Malaga, 2002. held on AH 2002, 2nd International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems.

[12] 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.

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

[14] K. Eshghi and R. Kowalski. Abduction compared with negation by failure. In *Proc. 6th ICLP'89*, pages 234–254, Lisbon, 1989.

[15] FIPA. FIPA 97, Specification part 2: Agent Communication Language. Technical report, Foundation for Intelligent Physical Agents, 1997.

[16] Nicola Henze and Wolfgang Nejdl. Adaptation in open corpus hypermedia. *IJAIED Special Issue on Adaptive and Intelligent Web-Based Systems*, 12, 2001.

[17] Koen V. Hindriks, F. de Boer, W. van der Hoek, and J.J. Meyer. Agent programming with declarative goals. In Cristiano Castelfranchi and Yves Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures and Languages*, volume 1986 of *LNAI*, pages 228–243. Springer-Verlag, 2001.

[18] L.Giordano, A.Martelli, and C.Schwind. Ramification and causality in a modal action logic. *Journal of Logic and Computation*, 10(5):625–662, 2000.

[19] J. Lobo, G. Mendez, and S. R. Taylor. Adding Knowledge to the Action Description Language $\mathcal{A}$. In *Proc. of AAAI'97/IAAI'97*, pages 454–459, Menlo Park, 1997.

[20] L. Marucci and F. Paternò. Designing an adaptive virtual guide for web application. In *Proc. 6th ERCIM Workshop UI4All*, Florence, Italy, 2000.

[21] A.S. Rao and M.P. Georgeff. Modeling rational agents whithin a bdi-architecture. In *Proc. of KR'91*, pages 473–484, 1991.

[22] 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.

[23] Mia K. Stern and Beverly Park Woolf. Curriculum sequencing in a Web-based tutor. *Lecture Notes in Computer Science*, 1452, 1998.

[24] F. Toni and A. Kakas. Computing the acceptability semantics. *LNAI*, 928:401–415, 1995.

[25] Gerhard Weber and Peter Brusilovsky. ELM-ART: An Adaptive Versatile System for Web-based Instruction. *IJAIED Special Issue on Adaptive and Intelligent Web-Based Systems*, 12, 2001.