ABSTRACTION OF REASONING FOR PROBLEM SOLVING AND TUTORING ASSISTANTS

by

Vu Le

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____	Dr. Gheorghe Tecuci, Dissertation Co-Director
_____	Dr. Mihai Boicu, Dissertation Co-Director
_____	Dr. James Chen, Committee Member
_____	Dr. Nada Dabbagh, Committee Member
_____	Dr. Harry Wechsler, Committee Member
_____	Dr. Daniel Menascé, Associate Dean for Research and Graduate Studies
_____	Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: February 20th, 2008

Spring Semester 2008
George Mason University
Fairfax, VA

Abstraction of Reasoning For Problem Solving and Tutoring Assistants

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Vu Le
Master of Science
George Mason University, 1999

Co-Director: Gheorghe Tecuci, Professor,
Department of Computer Science
Co-Director: Mihai Boicu, Assistant Professor,
Department of Applied Information Technology

Spring Semester 2008
George Mason University
Fairfax, VA

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS/SYMBOLS

?O1, ?O2,..., ?N1,... - variables in problems, reductions or solutions

$\forall$ - for all

$\exists$ - there is at least one

$\exists!$ - there is only one

$\mathcal{V}_t$ - set of vertices

δ_t - argument function of the tree t

$t = (\mathcal{V}_t, \delta_t)$ - tree with finite vertices set $\mathcal{V}_t$ and argument function δ_t

$v_t(x)$ – the valence of x in tree t

$st = (\mathcal{V}_{st}, \delta_{st})$- sub-tree with finite vertices set $\mathcal{V}_{st}$ and argument function δ_{st}

$\mathcal{N}(t)$ – natural notation of a tree t

$\mathcal{P}_t$ – the set of problem nodes in tree t

$\mathcal{Rd}_t$ – the set of reduction nodes in tree t

$\mathcal{S}_t$ – the set of solution nodes in tree t

Root(t) – the root of the tree t

Leaves(t) – the leaves of the tree t

δ_{tP} - problem node argument function

δ_{tS} - solution node argument function

δ_{tRd} - reduction node argument function

SN – the sub-node

$Partition_t$ – partition of tree *t*, a set of sub-trees of tree *t*

$t^{(i)}$- a tree at abstract level *i*

$\alpha_t(st)$ – abstraction function of a sub-tree of a tree *t*

$\alpha_P(st)$ – problem node abstraction function of a sub-tree *st*

$\alpha_{Rd}(st)$ – reduction node abstraction function of a sub-tree *st*

$\alpha_S(st)$ – solution node abstraction function of a sub-tree *st*

$\mathcal{PC}_\Lambda$ – set of problem classes of abstraction mapping Λ

$\mathcal{SC}_\Lambda$ – set of solution classes of abstraction mapping Λ

$\mathcal{RdR}_\Lambda$ – set of reduction rules of abstraction mapping Λ

$\mathcal{rRdR}_\Lambda$ – set of root reduction rules of abstraction mapping Λ

$\mathcal{APC}_\Lambda$ – set of abstract problem classes of abstraction mapping Λ

$\mathcal{ASC}_\Lambda$ – set of abstract solution classes of abstract mapping Λ

$\mathcal{ARdR}_\Lambda$ – set of abstract reduction rules of abstract mapping Λ

$\Lambda(\sigma)$ – abstraction mapping of a class $\sigma \in \mathcal{PC}_\Lambda \cup \mathcal{SC}_\Lambda \cup \mathcal{RdR}_\Lambda$

$\Lambda_P(\sigma)$ – problem abstraction mapping of a problem class σ

$\Lambda_S(\sigma)$ – solution abstraction mapping of a solution class σ

$\Lambda_{Rd}(\sigma)$ – reduction abstraction mapping of a reduction rule σ

$\Lambda_{rRd}(\sigma)$ – root reduction abstraction mapping of a reduction rule σ

ABSTRACT

ABSTRACTION OF REASONING FOR PROBLEM SOLVING AND TUTORING ASSISTANTS

Vu Le, PhD

George Mason University, 2008

Dissertation Directors: Dr. Gheorghe Tecuci and Dr. Mihai Boicu

This dissertation presents an approach to the abstraction of the reasoning of a knowledge-based agent that facilitates human-agent collaboration in complex problem solving and decision-making and the development of systems for tutoring expert problem solving to non-experts.

Effective human-agent collaboration requires an ability of the user to easily understand the complex reasoning generated by the agent. The methods presented in this dissertation allow the partition of a complex reasoning tree into meaningful and manageable sub-trees, the abstraction of individual sub-trees, and the automatic generation of an abstract tree that plays the role of a table of contents for the display, understanding and navigation of the concrete tree.

Abstraction of reasoning is also very important for teaching complex problem-solving to non-experts. This dissertation presents a set of integrated methods that allow the abstraction of complex reasoning trees to define abstract problem solving strategies for

tutoring, the rapid development of lesson scripts for teaching these strategies to non-experts, and the automatic generation of domain-specific lessons. These methods are augmented with ones for learning and context-sensitive generation of omission, modification, and construction test questions, to assess a student's problem solving knowledge.

The developed methods have been implemented as an extension of the Disciple learning agent shell and have led to the development of the concept of learning and tutoring agent shell. This is a general tool for building a new type of intelligent assistants that can learn complex problem solving expertise directly from human experts, support human experts in problem solving and decision making, and teach their problem solving expertise to non-experts. The developed learning and tutoring shell has been used to build a prototype tutoring system in the intelligence analysis domain which has been used and evaluated in courses at the US Army War College and George Mason University.

1. Introduction

1.1. Knowledge-Based Agents

An important goal of Artificial Intelligence is to develop knowledge-based agents that represent the subject matter expertise of human experts in particular domains, such as engineering design, emergency response planning, intelligence analysis, medical diagnosis and treatment, etc. These agents could act as "interactive, user-adaptive problem solving aids that understand what they do, accept goals being set as input rather than instructions or deduce such goals, and, once these goals are identified, aim at solving them independently from their user" (Kaschek, 2006).

A knowledge-based agent may be used by a subject matter expert as a decision-making assistant, or by a non-expert user as an expert system, or by a student as a tutoring system. In the words of Edward Feigenbaum (1993), "Rarely does a technology arise that offers such a wide range of important benefits of this magnitude. Yet as the technology moved through the phase of early adoption to general industry adoption, the response has been cautious, slow, and 'linear' (rather than exponential)."

There are several explanations of this situation. One is the difficulty of acquiring and representing the subject matter expertise of human experts (Buchanan and Wilkins, 1993). Knowledge acquisition for tutoring purposes, which also involves building lessons and exercises, is even more difficult (Murray, 1999). Anderson (1992) estimated that "it

takes at least 100 hours to do the development that corresponds to an hour of instruction for a student." Other difficulties are related to the actual use of such systems. Solving complex, real-world problems involves reasoning trees with thousands or tens of thousands of reasoning steps. A user must be able to understand and work with this complex reasoning if he or she is to use the system as a decision-making assistant. Similarly, a student has to be able to learn from such a complex reasoning.

A general objective of this PhD dissertation is to investigate how abstraction of reasoning may advance the state of the art in the development and use of knowledge-based agents. In particular, we investigate the development of a specific type of intelligent assistant (see Figure 1) that can:

- learn complex problem solving expertise directly from human experts;
- support human experts in complex problem solving and decision making;
- teach their complex problem solving expertise to non-experts.

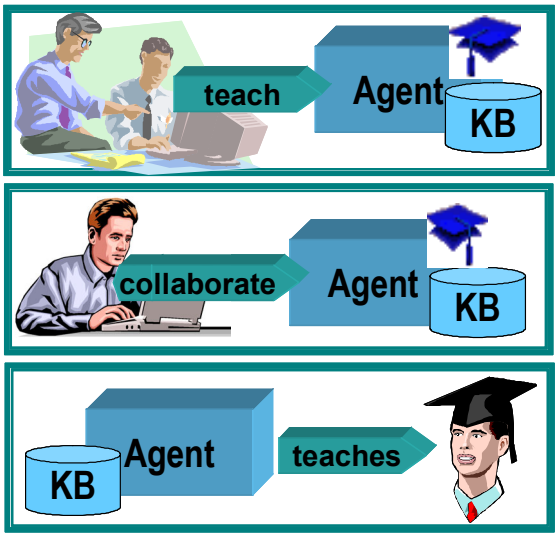


Figure 1: Intelligent Learner, Assistant and Tutor

For this type of agents, we investigate how abstraction of complex reasoning, viewed as a type of simplification that removes less important details, may facilitate human-agent collaboration in complex problem solving and decision-making, teaching complex problem-solving to non-experts, and rapid development of intelligent tutoring systems for complex problem solving.

This dissertation builds upon three areas, Expert Systems, Machine Learning, and Intelligent Tutoring Systems, which are briefly reviewed in the following sections.

1.2. Expert Systems

An expert system is a knowledge-based system which represents the human expertise in some specialized area and uses that knowledge to solve problems in that area. The expert system behaves as a human expert during the problem solving process to find solutions to problems and present the appropriate explanations of the problem solving process.

The input problems for an expert system are usually complex and difficult enough to require significant human expertise for their solutions (Feigenbaum, 1982). These problems demand a substantial body of knowledge with different levels of uncertainty (Waltz, 1983).

The main modules of an expert system are the *knowledge base* and the *inference engine*. The knowledge base stores the knowledge of a certain expertise domain, acquired by the knowledge engineer from a given subject matter expert, and encoded in production rules, heuristics, facts, etc. The inference engine implements a general method for solving

problems by using the knowledge from the knowledge base. A main architectural principle in the development of an expert system is *the separation between the inference engine and the knowledge base* (Davis, 1982), as shown in Figure 2. These two modules are usually built separately so that the same inference engine can potentially be used with different knowledge bases. Apart from reusing the inference engine (Whitley, 1990), this makes the knowledge in knowledge base more easily identifiable, more explicit and more accessible.

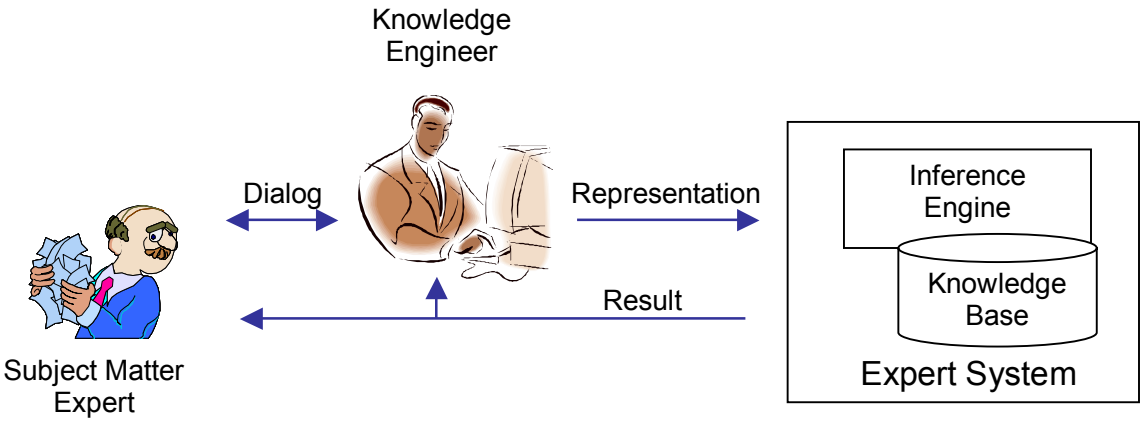


Figure 2: Expert System Development

The communication between the subject matter expert and the expert system is a difficult issue. Each side speaks a different language and the common understanding is usually vague. The knowledge engineer has to interact with the subject matter expert to understand how a problem is to be solved, then uses some representation to encode the expert's knowledge into the system. This process is time consuming, difficult and error prone, being well known as the *knowledge acquisition bottleneck* (Buchanan and Wilkins, 1993).

Due to the separation between the inference engine and the knowledge base, a generic inference engine can be developed and used with different knowledge bases to create expert systems for different purposes. This approach, *Expert System Shell,* revolutionizes the way the expert systems are built. The expert system shell contains a generic inference engine and an empty knowledge base with a pre-defined knowledge representation. Now the developers of the expert systems are no longer concerned with building the problem solving engine. Building an expert system reduces to building a knowledge base following a pre-defined syntax. In addition, most shells provide useful utilities that can do some additional tasks such as knowledge base integrity checking and debugging (Whitley, 1990).

A critical characteristic of expert systems that are used as decision-support assistants is the ability to make very clear their reasoning process. For very complex problems, however, the reasoning trees are very large, making their browsing and understanding difficult. This problem can be alleviated by abstracting the reasoning process, as proposed in this PhD dissertation. This allows the user of an expert system to both get a general understanding of the reasoning strategy (at an abstract level) and investigate the details of the reasoning (when needed).

1.3. Machine Learning and Learning Agent Shells

The knowledge acquisition bottleneck plagues the development of expert systems. One approach to alleviate this problem is to automatically acquire domain knowledge through learning. Knowledge acquisition can be based on several *Machine Learning* strategies (Tecuci, 1998):

- *Empirical inductive learning from examples* learns the definition of a concept from a set of positive and negative examples. The inductive process generates the generalized description for that concept (Mitchell, 1978).
- *Explanation-based learning* learns by observing a single example to improve system's performance. However, this technique requires complete and correct knowledge of the domain under study (Mitchell, 1997).
- *Analogical learning* learns by transferring knowledge from a source entity to a target entity (Winston, 1980).
- *Abductive learning* hypothesizes the causes of observed effects (Josephson et al., 1987).
- *Conceptual clustering* classifies a set of objects into concepts and learns the descriptions of these concepts (Kodratoff and Tecuci, 1988; Fisher, 1987).
- *Quantitative discovery* discovers quantitative laws that relate to the values of variables characterizing objects (Langley et al., 1987).
- *Reinforcement learning* learns by using the feedback on an agent's performance from the environment (Sutton, 1988).
- *Genetic algorithms* learn by evolving a population of individuals over a sequence of generations (DeJong, 2006).
- *Neural networks* learn by evolving a network of connected nodes which simulates the brain's dendrites and axons (Rumelhart and McClelland, 1986).

- *Multistrategy learning* integrates complementary machine learning approaches to solve learning problems that are beyond the capabilities of the integrated methods (Michalski and Tecuci, 1994).

A significant advance in the use of machine learning for knowledge acquisition was the development of the concept of learning agent shell (LAS), as an extension and generalization of the concept of expert system shell (Tecuci, 1998). A learning agent shell is a tool for building expert systems. It contains a general problem solving engine, a learning engine and a generic knowledge base structured into object ontology and a set of rules (see Figure 3).

The main purpose of the learning agent shell is to enable rapid development of the knowledge base, directly by the subject matter experts, with limited assistance from the knowledge engineers. A specific type of learning agent shell which was used as an experimentation platform for this dissertation research is the Disciple shell (Tecuci et al., 1998; Boicu, M., 2004). It consists of:

- A problem solving component based on problem reduction. This component includes a modeling agent that helps the user to express his/her contributions to the problem solving process, a mixed-initiative (step-by-step) problem solving agent, and an autonomous problem solving agent.
- A learning component for acquiring and refining the knowledge of the agent, allowing a wide range of operations, from ontology import and user definition of knowledge base elements (through the use of editors and browsers), to ontology learning and rule learning.

- A knowledge base manager which controls the access and the updates to the knowledge base. Each module of Disciple can access the knowledge base only through the functions of the knowledge base manager.
- A domain-independent, graphical user interface.

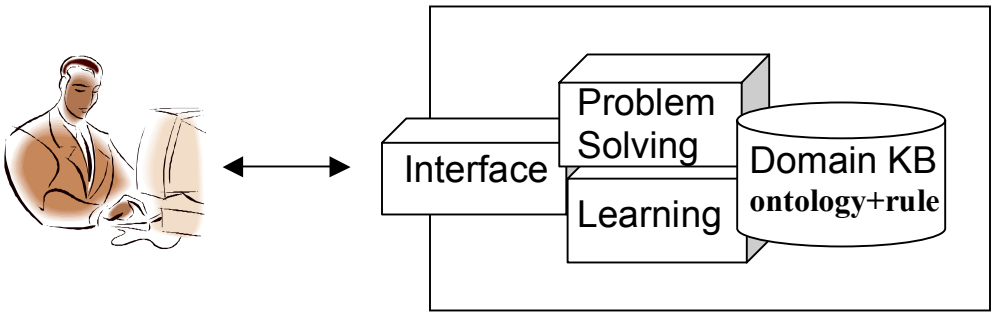


Figure 3: Learning Agent Shell Architecture

Building an agent for a specific application consists of customizing the shell for that application and developing the domain knowledge base. The learning engine (which uses various learning strategies, such as learning from examples, from explanations, and by analogy) facilitates the building of the knowledge base by subject matter experts. It reduces the involvement of the knowledge engineers who otherwise would play a very important role in acquiring knowledge from the expert and encoding it in the knowledge base. This leads to a significant speed-up of the process of building a knowledge-based system.

The methodology to build an end-to-end knowledge-based agent with a Disciple shell is the following one (Tecuci et al., 1999):

- *Specification of the problem*: The subject matter expert and the knowledge engineer usually accomplish this step to identify the types of problems to be solved by the system.
- *Modeling the problem solving process as problem reduction*: The expert and the knowledge engineer work together to model the problem solving process as problem reduction and, in the process, define: a) an informal description of the agent's problems, b) instances and concepts are defined, and (3) conceptual problem reduction trees to guide the training of the agent by the subject matter expert.
- *Developing the customized agent*: Add auxiliary components (as needed) such as graphical viewer for a reasoning tree, special report generation capabilities, etc.
- *Importing and developing the ontology*: There are many available ontologies that can be imported partially. The imported ontology is then extended by using the different tools for different knowledge elements, such as feature editor, problem editor, object editor, etc.
- *Training the agent for its domain-specific problems*: During this step, the expert teaches the agent to solve problems in a cooperative, step-by-step problem solving scenario. The expert defines an initial problem and asks the agent to reduce it. The agent will try different methods to reduce the current problem. If the solution was defined or modified by the expert, then it represents an initial example for learning a new reduction rule. To learn the rule, the agent will first try to find an

explanation of why the reduction is correct. Then the example and the explanation are generalized to a rule which becomes part of the agent's knowledge base.

- *Testing and using the agent*: The agent is tested with additional problems. The agent will solve the problems alone. The solutions are then inspected by the expert. If the agent generated wrong solutions then the expert will identify the errors and will help the agent to fix them.

An important characteristic of a Disciple-type learning agent shell is that it allows the subject matter expert to teach the agent in a very natural way, similar to how the expert would teach a student. As a consequence, the reasoning of the agent will be very natural, similar to that of the expert who has taught it. This will facilitate the understanding of the agent's reasoning by an end-user. But it also opens a significant opportunity with respect to tutoring, which is exploited by our dissertation research. It may make possible for such an agent to teach students in a way that is similar to how it was taught by the subject matter expert. This is important for two different reasons. First, a new user of the agent should become familiar with how the agent solves problems, if the user is to use the agent for decision-support. Second, teaching expert problem solving strategies is an important application area and easily building such tutoring systems would have a significant economic impact.

As discussed in Chapter 6, we have developed the concept of learning and tutoring agent shell, as an extension of the concept of learning agent shell. A learning and tutoring agent shell allows rapid development of intelligent tutoring systems for problem solving knowledge.

1.4. Intelligent Tutoring Systems

Intelligent tutoring systems (ITS) emulate the human tutors in teaching the students. Unlike *the computer-based training* (CBT) or *computer-aided instruction* (CAI) systems (Carbonell, 1970) which drive the students inflexibly following predefined scripts, an ITS focuses on individualized curriculum that suits the student's need. In order to do that, the ITS needs: 1) the representation of the domain knowledge which is handled by the *expert module*; 2) the tutoring knowledge which is stored in the *tutoring module*; and 3) the knowledge of the student's capability and progress which is stored in the *student module* (Polson and Richardson, 1988). These three modules constitute the backbone of the tutoring system. The other modules such as the *user interface* play supportive roles in preparing and constructing the curriculum customized to the student ability as shown in Figure 4 (Polson and Richardson, 1988).

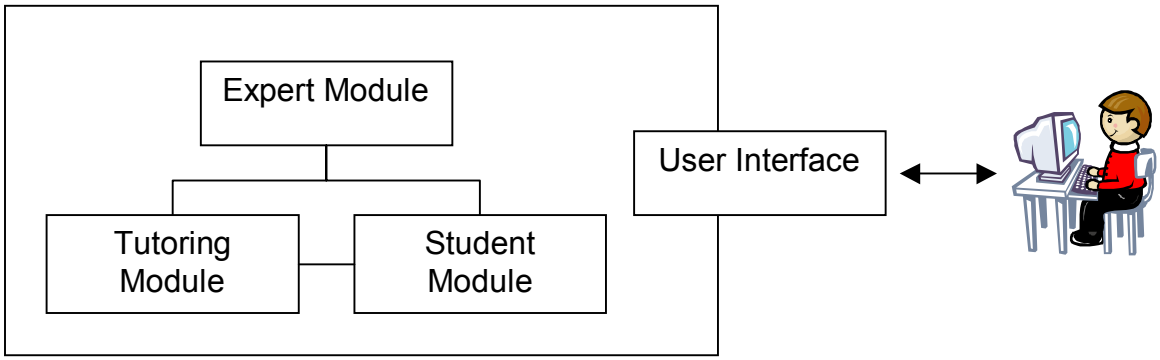


Figure 4: General Architecture of an Intelligent Tutoring System

The expert module contains the domain knowledge and has the problem solving capability necessary for the subjects that the tutoring system is designed for. Acquiring

the expert knowledge is both time consuming and difficult. For instance, Anderson (1998) estimated that for the applications to programming and mathematics, over 50% of the effort goes into encoding the domain knowledge (Anderson, 1998).

There are several models that are typically used by the expert module. The black box model encapsulates the domain knowledge and delivers the output based on the given input without explanations of why the problem is solved that way (Anderson, 1998). This type of behavior can be used to judge the correctness of student's performance while executing the same tasks. A typical example of the use of a black box model in a tutoring system is SOPHIE (Brown et al., 1982) which teaches students how to troubleshoot faulty electronic circuits. An alternative model is based on expert systems and is widely used in teaching the expert knowledge (Anderson, 1998). A classic and well-known tutoring system which teaches students how to diagnose the bacterial infection is GUIDON (Clancey, 1987). GUIDON is based on the MYCIN expert system and generates explanations of how the results have been obtained. Yet another type of expert model is the cognitive model. This model simulates the "*human problem solving in a domain in which the knowledge is decomposed into meaningful, human-like components and deployed in a human-like manner*." (Anderson, 1998). A typical example of this type is the LISP Tutor (Anderson and Reiser, 1985) which teaches the students how to program in LISP. Because the cognitive systems simulate the human problem solving knowledge, the understanding of different types of knowledge that need to be tutored is useful. There are 3 levels of knowledge: procedural, declarative and qualitative. The procedural knowledge relates to how a task is performed (Anderson, 1998). The LISP tutor is a cognitive tutoring system which uses procedural knowledge. The declarative

knowledge is a set of facts appropriately organized to be reasoned with. The tutoring systems which use the declarative knowledge are designed to teach the students the basic principles and facts of the domain and how to reason with them in general (Anderson, 1998). An example of this type of tutoring system is SCHOLAR which teaches the students the South American geography (Carbonell, 1970). Qualitative knowledge is "*any kind of knowledge that does not always allow a correct and consistent match between the represented objects and the real world, but can nevertheless be used to get approximate characterizations of the behavior of the modeled domain*" (Furnkranz, 1992). This type of knowledge therefore underlies the human capability of simulation and reasoning. Thus it is essential in the troubleshooting process. SOPHIE (Brown et al., 1982) uses this type of knowledge as well to teach a student how to troubleshoot a faulty circuit.

The student module (see Figure 4) evaluates the student's performance to determine his/her knowledge of the domain and reasoning skills (Ong and Ramachandran, 2000). The student model built and maintained by this module uses that understanding to help the student in many ways. It can advance the student to a higher level if it is determined that the student succeeded in answering most of the questions or seemed to master the presented topics. The tutoring system can give explanations to the student based on the concepts and definitions that have been previously presented to the student and are recorded in the student model. Or the system can give some advice during runtime when the student model can "feel" that the student does not know how to proceed further based on his/her suboptimal performance or misconceptions (Digangi, A. S., et al., 1999).

There are 3 types of student models: overlay model, differential model and perturbation model (Smith, 1998; Tsinakos and Margaritis, 2000). In the overlay model the student's knowledge is a subset of the expert's knowledge. The student knowledge will expand when more knowledge is acquired and eventually becomes the expert knowledge. The differential model is an extension of the overlay model where it focuses on two types of learner knowledge: the knowledge that the student must have, and the knowledge the student is not exposed to. The student knowledge may never be the expert knowledge and is limited by the knowledge that the student is not supposed to be exposed to. Neither the overlay model nor the differential model supports the correction of the faulty knowledge of the student. The perturbation model is an overlay model with such faulty knowledge which is called "bug library" (Tsinakos and Margaritis, 2000).

The student performance is evaluated by the student model. This model keeps track of student progress. The tutoring module interacts with the student module to define the curriculum which is appropriate, based on the student's capability. This module must possess at least the following three characteristics (Halff, 1988):

- It should control the generation of the curriculum (which is the selection and sequencing of the material to be presented).
- It should be able to answer the questions posted by the student during the tutoring process.
- It should have a mechanism to determine when the student needs help and what type of help the student should receive.

The tutoring module can define different tutoring strategies to deal with different student skills. For the beginner, the tutoring module can guide the student with step-by-

step procedures. For the advanced student, it can decide to have the student work on her/his own unless the student needs help. In other words, the tutoring module adapts to the student performance and skill to ensure the effective learning.

In our dissertation research we have developed an abstraction-based approach for tutoring expert problem solving knowledge, as discussed in Chapter 4.

1.5. Sample Application Area: Intelligence Analysis

The purpose of intelligence analysis is to analyze available partial and uncertain information in order to estimate the likelihood of one possible outcome, given the many possibilities in a particular scenario. An intelligence analyst has to solve complex problems such as

- *Assess whether Location-A is a training base for terrorist operations.*
- *Assess whether Agent-B has nuclear weapons.*
- *Assess whether Agent-C is pursuing nuclear energy for peaceful purposes.*

Solving such problems involve analyzing large amounts of uncertain, incomplete and/or incorrect information in the form of pieces of evidence whose relevance and believability have to be evaluated and correlated. They result in large reasoning trees of thousands or even tens of thousands of reasoning steps.

Therefore, this application domain is very appropriate for demonstrating and testing the abstraction-based methods proposed in this dissertation

1.6. Dissertation Overview

The rest of this dissertation is organized as follows. Chapter 2 presents the research problem addressed along with other related research. Chapter 3 presents the developed

theory for abstracting a complex reasoning tree generated by a knowledge-based agent, in order to facilitate human-agent collaborative problem solving, and tutoring expert problem solving to non-experts. Chapter 4 presents the abstraction-based methods developed to facilitate a human's browsing and understanding of a complex reasoning tree generated by an agent. The methods were also evaluated. Then, Chapter 5 presents the application of our theory of abstraction to the tutoring of expert problem solving strategies. It describes a set of integrated methods that allow the abstraction of complex reasoning trees to define abstract problem solving strategies for tutoring, the rapid development of lesson scripts for teaching these strategies to non-experts, and the automatic generation of domain-specific lessons. It also describes the developed methods for learning and context-sensitive generation of omission, modification, and construction test questions, to assess a student's problem solving knowledge. It also includes the evaluation of these methods. Chapter 6 presents the concept of learning and tutoring agent shell, the architecture of the prototype shell developed, and the methodology of building a learning and tutoring agent with such a shell. Chapter 7 concludes this dissertation with a summary of my research contributions and some of the most promising directions for future research. The dissertation also includes several Appendices with more details on several aspects presented in the dissertation.

2. Research Problem

2.1. Problem Definition

Research progress in Artificial Intelligence has led to the development of knowledge-based agents that can solve complex real-world problems requiring large amounts of human subject matter expertise. In principle, such an agent can be used by a subject matter expert as a decision-making assistant, or by a non-expert user as an expert system, or by a student as a tutoring system.

A critical requirement for such a knowledge-based agent is the transparency of its reasoning process. To accept a decision suggested by an agent, its user has to be able to easily understand how that decision has been reached. Similarly, to teach a student, the reasoning of the agent has to be natural and easily understood. This requirement becomes increasingly difficult to be achieved when the reasoning trees generated by the agent are very complex, with thousands of reasoning steps. This also makes it very difficult, not only to teach a student, but also to build the necessary tutoring knowledge.

Abstraction of complex reasoning, viewed as a type of simplification that removes less important details, may be the key to both facilitate human-agent collaboration and teach students complex problem-solving.

Consequently, the problem addressed by this dissertation research is to develop an approach to the abstraction of complex reasoning processes that facilitates:

- *human-agent collaboration in complex problem solving and decision-making;*
- *rapid development of intelligent tutoring systems for complex problem solving;*
- *teaching complex problem-solving to non-experts.*

Figure 5 shows the three main processes that are addressed by the researched approach to abstraction: human-agent collaboration, instructor authoring of tutoring knowledge, and agent teaching of a student.

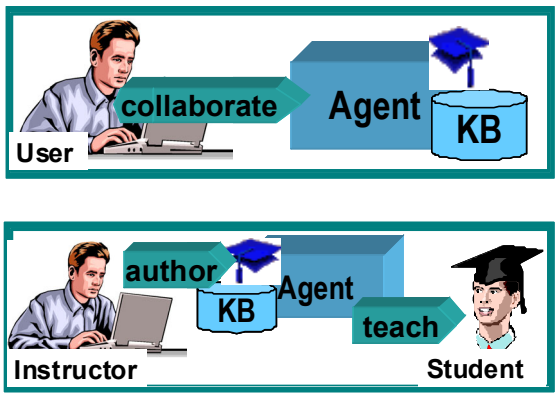


Figure 5: Processes Facilitated by the Proposed Approach to Abstraction

A fourth process, related to those from Figure 5, is that of knowledge acquisition from a subject matter expert. This process is critical because it is the knowledge acquired from the subject matter expert that is used in problem solving, and it is this knowledge that has to be taught to a student.

One of the most advanced and successful approaches to knowledge acquisition is to use a learning agent that can be taught directly by a subject matter expert how to reason and solve problems, as illustrated by the family of Disciple systems (Tecuci et al., 1998;

Boicu 2002). One advantage of this knowledge acquisition approach to the research problem we are investigating is that the reasoning of the agent is already natural, as it emulates the reasoning used by the expert when teaching the agent. Thus our efforts can concentrate on how abstraction can deal with the complexity of the reasoning trees, and not with reformulating this reasoning to make it more natural. Moreover, the agent might be able to teach a student similarly to how it was taught by the subject matter expert.

This creates the opportunity to develop a new type of intelligent assistant that integrates the three complementary capabilities shown in Figure 1:

- can learn complex problem solving expertise directly from human experts;
- can support human experts in complex problem solving and decision making;
- can teach their complex problem solving expertise to non-experts.

The addressed research problem includes:

- the development of a theory of the abstraction of complex reasoning processes for collaborative problem solving and tutoring;
- the development of methods for abstracting concrete reasoning trees to facilitate collaborative problem solving;
- the development of methods for abstracting concrete reasoning trees to facilitate the tutoring of expert problem solving strategies;
- the development of abstraction-based methods for authoring lessons to teach students;
- the development of methods to teach the agent to generate test questions;

- the development of the concept of learning and tutoring agent shell as a powerful tool for building learning, problem solving and tutoring agents for complex application domains.

The next section discusses the related research, pointing to existing limitations that are addressed by our work.

2.2. Related Research

There are two major issues presented in this dissertation. One is the abstraction theories and the other is the intelligent tutoring systems. The two will be discussed in details in Section 2.2.1 and Section 2.2.2 respectively.

2.2.1. Abstraction Related Research

Abstraction has been widely used in human perception, reasoning and problem solving. Its benefit has motivated the Artificial Intelligence theorists and practitioners to capture the underlying principles and characteristics of abstraction and apply them to building intelligent systems that can reason and solve problems. The theories of abstraction were needed for three reasons: to understand different abstraction approaches that have been used in the past, to justify the need to use abstraction in terms of computational complexity, and to construct the intended abstractions automatically (Zucker, 2003).

There are several existing theories of abstractions. In essence, they can be classified into four categories (Zucker, 2003): abstraction as predicate mapping (Plaisted, 1981), (Tenenberg, 1987), abstraction as mapping between formal systems (Giunchiglia and Walsh, 1992), abstraction as semantic mapping of interpretation models (Giordana and

Saitta, 1990), (Nayak and Levy, 1995), and perception-based abstraction (Saitta and Zucker, 1998). We will present the frameworks for each of the categories and find the relations between them and our abstraction of reasoning presented in Chapter 3.

Abstraction as Predicate Mapping

Abstraction as predicate mapping is the class of abstractions that maps a set of predicates in one first-order language to those of another language $f: P_1 \rightarrow P_2$ where P_1 is a set of predicates of language L_1, and P_2 is a set of predicates of language L_2. The mapping f is not a one-to-one relationship. It is possible that more than one predicate $p_i \in P_1$ can be mapped to the same predicate $p_j \in P_2$. The mapping f then can be extended to map the literals in L_1 to those in L_2 (Tenenberg, 1987).

The predicate mappings are in fact the subclass of *abstraction mapping* defined in (Plaisted, 1981), quoted by Tenneberg (1987).

Definition 1.1 (Abstraction Mapping – Plaisted, 1981): "An abstraction is an association of a set $f(C)$ of clauses with each clause C such that f has the following properties:

[1] If clause C_3 is a resolvent of C_1 and C_2 and $D_3 \in f(C_3)$, then there exist $D_1 \in f(C_1)$ and $D_2 \in f(C_2)$ such that some resolvent of D_1 and D_2 subsumes D_3.

[2] $f(\varnothing) = \{\varnothing\}$.

[3] If C_1 subsumes C_2, then for every abstraction D_2 of C_2 there is an abstraction D_1 of C_1 such that D_1 subsumes D_2." ■

If f is a mapping with these properties, then we call f an abstraction mapping of clauses. The set of clauses C is called *original theory* and $f(C)$ is called *abstract theory.*

The mapping however could lead to undesirable false proof - discovered in (Plaisted, 1981), as quoted by Zucker (2003). To solve this problem Tenneberg proposed the *Restricted Predicate Mappings.* The restriction interprets an abstract predicate of the abstract theory as the union of the predicates from the original theory that are mapped to it (Tenneberg, 1987).

This type of abstraction however is not applicable to our abstraction of reasoning due to the fact that it does not take into account the reason why the abstraction is justified, i.e., the semantics of the abstraction.

Abstraction as Mapping between Formal Systems

Giunchiglia and Walsh (1992) defines a formal system Σ as a triple (Λ, $\triangle$, Ω) where Λ is the language, $\triangle$ is the deductive engine of the system Σ and Ω is the set of axioms.

Definition 1.2 (Formal System Abstraction – Giunchiglia and Walsh, 1992): "An abstraction, written as f: $\Sigma_1 => \Sigma_2$ is a pair of formal systems (Σ_1, Σ_2) with language Λ_1 and Λ_2 respectively, and an effective total function f_Λ: $\Lambda_1 \rightarrow \Lambda_2$. "■

Σ_1 is called "*ground space"* and Σ_2 "*abstract space"*, the effective total function f_Λ is an abstraction. The function f_Λ is called "*total*" because all the well-formed formulas (wff) of the system Σ_1 are mapped to Σ_2.

According to Giunchiglia and Walsh (1992), there are three types of abstraction: *theorem increasing (TI), theorem decreasing (TD), and theorem complete (TC).* They are defined as follows:

Definition 1.3 (T* Abstraction – Giunchiglia and Walsh, 1992): "An abstraction f: $\Sigma_1 => \Sigma_2$ is called

- TC abstraction iff, for any wff α, $\alpha \in TH(\Sigma_1)$ iff $f_\Lambda(\alpha) \in TH(\Sigma_2)$.
- TD abstraction iff, for any wff α, if $f_\Lambda(\alpha) \in TH(\Sigma_2)$ then $\alpha \in TH(\Sigma_1)$.
- TI abstraction iff, for any wff α, if $\alpha \in TH(\Sigma_1)$ then $f_\Lambda(\alpha) \in TH(\Sigma_2)$"

where $TH(\Sigma_1)$ is the set of theorems of Σ_1 and $TH(\Sigma_2)$ is the set of theorems of Σ_2. T^* abstraction is either of the types. ■

In TC abstraction, all members of $TH(\Sigma_1)$ are mapped to all members of $TH(\Sigma_2)$, as shown in middle of Figure 6. In TD abstraction, only a subset of $TH(\Sigma_1)$ is mapped to the members of $TH(\Sigma_2)$ as shown in top part of Figure 6. An example of such abstraction is the dropping axioms and/or inference rules. TD abstraction is therefore called weak abstraction, because not all members of $TH(\Sigma_1)$ are mapped to $TH(\Sigma_2)$. Oppositely, the TI abstraction maps all members of $TH(\Sigma_1)$ to a subset of $TH(\Sigma_2)$ (bottom part of Figure 6). TI abstraction is preferable in problem solving because all the ground problems can have solutions once their abstract problems are solvable (Giunchiglia and Walsh, 1992). An example of TI abstraction is *Abstrips* which builds STRIPS plan (Giunchiglia and Walsh, 1992). Abstrips's operators together with pre-condition apply to the current state to generate new states. The TI abstraction can be applied to it. For example, the operator for climbing an object with a condition of being climbable

$$at(z, x, s) \Lambda climbable(y, z, s) \rightarrow at(z, x, climb(y, z, s))$$

can be abstracted to

$$at(z, x, s) \rightarrow at(z, x, climb(y, z, s))$$

with the condition of *climbable* being dropped.

This theory is useful in terms of classification of different types of abstractions. This theory of abstraction, however, is just a syntactic abstraction that does not take into account the semantics of the abstraction. Therefore, it is not qualified as our desirable theory of abstraction of reasoning where the underlying justification is too important to ignore.

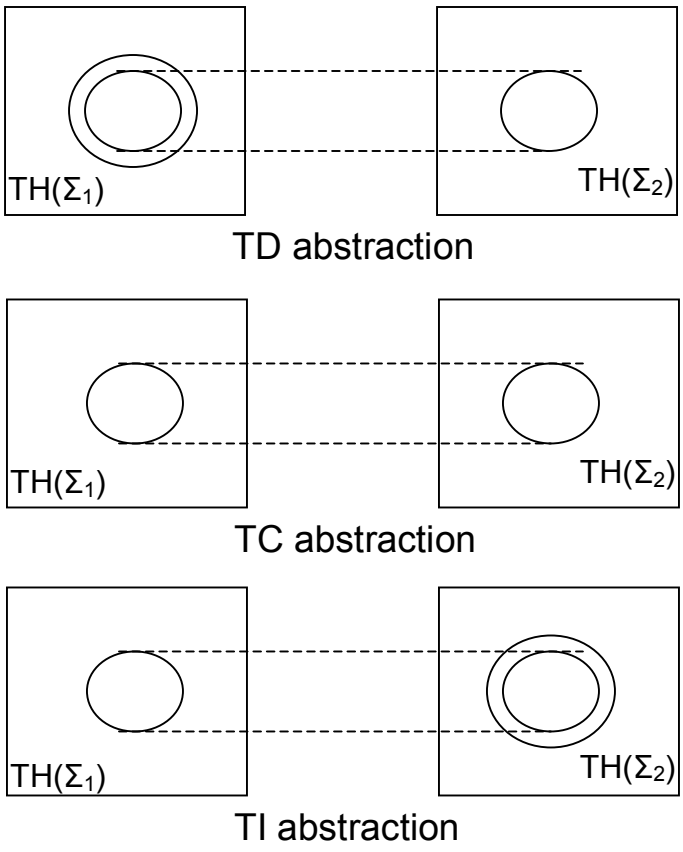


Figure 6: TD, TC and TI Abstractions

Abstraction as Mapping between Models

What we have presented so far is the syntactic abstraction. This type of abstraction does not take into account the underlying justifications that lead to the abstraction

(Zucker, 2003). Nayak and Levy (1995) proposes the theory of semantic abstraction. This theory defines the abstraction on the model level rather than the predicate level as the syntactic approaches. The semantic abstraction consists of two steps: the first step is to abstract the intended domain model and the second step is to construct the abstract formulae to capture the abstracted domain model. The abstract formulae are indeed the justification of the syntactic abstraction of the first step.

Nayak and Levy (1995) base their abstraction theory on the model which is defined as "an interpretation, I, is a model of a set of sentences, Σ, (denoted $I \models \Sigma$) if and only if I satisfies each sentence in the set" (Nayak and Levy, 1995).

Definition 1.4 (Model Increasing Abstractions – Nayak and Levy, 1995): "Let T_{base} and T_{abs} be sets of sentences in languages L_{base} and L_{abs}, respectively. Let π: $Interpretations(T_{base}) \rightarrow Interpretations(T_{abs})$ be an abstraction mapping. T_{abs} is a model increasing abstraction of T_{base}, with respect to π, if for every model M_{base} of T_{base}, $\pi(M_{base})$ is a model of T_{abs}."

One important notion that Nayak and Levy (1995) propose is the simplifying assumption. This notion can be used to prevent false proofs and can be used to evaluate the usefulness of the abstraction by the assessment of the reliability of the simplifying assumption (Zucker, 2003). Let us consider two railroad cars that are linked by a linkage. The linkage is modeled as a spring with a very large sprint constant, i.e., the spring is very stiff. The simplifying assumption sets the linkage as infinitive which makes the two railroad cars become one single rigid body. According to Nayak and Levy (1995), viewing abstraction as a combination of MI abstraction and simplifying assumption has

two advantages: one is that the simplifying assumption is made explicit and therefore it is very useful in reasoning or modeling. The other advantage is the MI abstraction admits false proof only when the simplifying assumption is inappropriate.

The theory of semantic abstraction now is equipped with the semantic underlying justification. It constructs the abstract formulae as the justification of the syntactic abstraction. The theory is based on models instead of on predicates as the other two. With these two properties, the theory of semantic abstraction can be a starting point in our formulation of abstraction of reasoning. The reasoning that is embedded in intelligent assistants is the product of a multi-step process from modeling the expert knowledge to learning the reasoning rules. We expect to have a theory that can capture such a complicated process. The next theory of abstraction comes closer to what we anticipate.

Perception-Based Abstraction

Perception-based abstraction was developed based on the observation that the conceptualization of a domain involves at least four different levels. They are perception, structure, language, and theory levels (Zucker, 2003). The concrete level is the world *W* where the concrete objects exist. The objects are perceived by the observer through his/her physical sensors. The perception *P(W)* is what the observers "feels" about the world, not the world per se. The perception is the *internal representation* of the perceived world. The perception however decays over time; the memorization of the perception into a structure *S* must be implemented to preserve the perception. The structure is the *external representation* of the perceived world. So far *P(W)* and *S* exist with respect to the observer only. To be able to reason about the perceived world, there must be a

language *L* to communicate with other agents. Now the perceived world can be described *intensionally*. Finally, the theory *T* is established to embed the properties and the knowledge of the world (Saitta and Zucker, 1998). Figure 7 shows the four level model with the general background knowledge providing inputs at all levels. Saitta and Zucker (1998) define $R = < P(W), S, L, T >$ as a *Reasoning Context.*

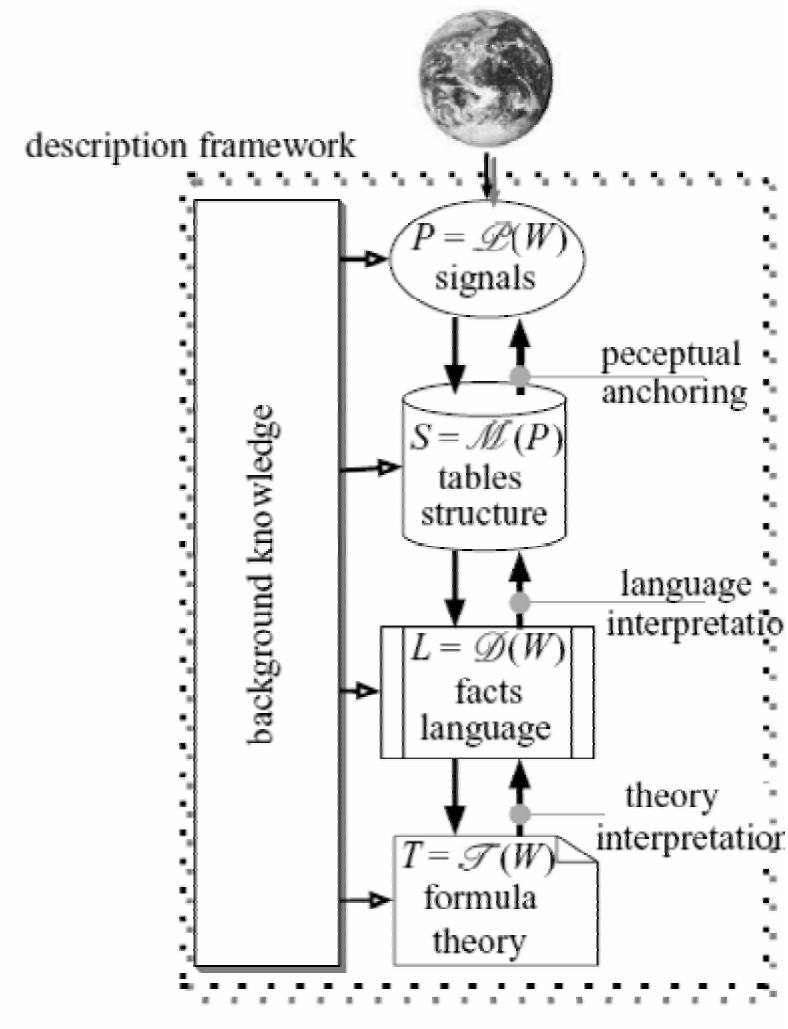


Figure 7: Four Levels of Representing and Reasoning about the World – from (Zucker, 2003)

The abstraction process starts from the perception level and propagates through all the levels. Figure 8 displays the models of abstraction that occur on the four levels. For each level, there is a corresponding abstraction operator. Specifically, $P_a = \omega(P_g(W))$, $S_a = \sigma(S_g)$, $L_a = \lambda(L_g)$ and $T_a = \tau(T_g)$.

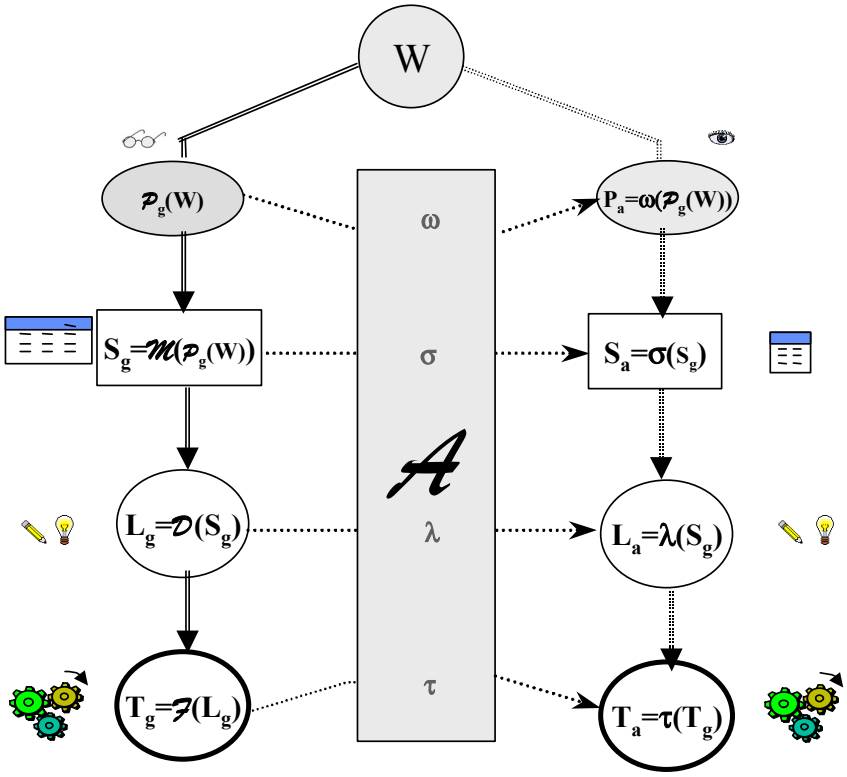


Figure 8: Knowledge Abstraction and Representation - from (Mustière et al., 2000)

Figure 9 presents an example of perception-based abstraction in cartography. This example concerns two aspects: the modeling of the knowledge acquisition of the map design process and partial automation of the process named cartographic generalization

(Zucker, 2003). The horizontal axis shows the abstraction process and vertical one shows the reformulation process.

As for the modeling process, the world W is perceived by aerial photographs or satellite images $P_g(W)$. The abstraction occurs at the perception level to map the captured images with appropriate resolutions $P_a = \omega(P_g(W))$ (step 1 in Figure 9). Step 2 involves the expert – photogrammetrist - who extracts a Digital Landscape Model (DLM) that contains the coordinates of all the objects in the images. This is the process of determining $S_g = \mathcal{M}(P_g(W))$. This step involves the abstraction and reformulation of an image to have it structured in some recognizable form and associated with categories such as road, building, rivers, etc - $S_a = \sigma(S_g)$. In the third step, a language L is selected to assign symbols to map objects, such as houses, roads, etc. $L_g = \mathcal{D}(S_g)$. The abstraction of the language level is not applied in the modeling process, but it will be used in the cartographic generalization process. Finally, the theory level is achieved by the use of maps in different areas, such as space and landscape analysis, direction guidance, or geographic theory, $T_g = \mathcal{T}(L_g)$ (Zucker, 2003).

With regard to the cartographic generalization process, the abstraction involves repetitive scaling, reorganization of the map objects, and arrangement of different levels of details, $L_a = \lambda(L_g)$. The basic operations that the expert uses in this process are the applications of transformation algorithms to the GDB (Zucker, 2003).

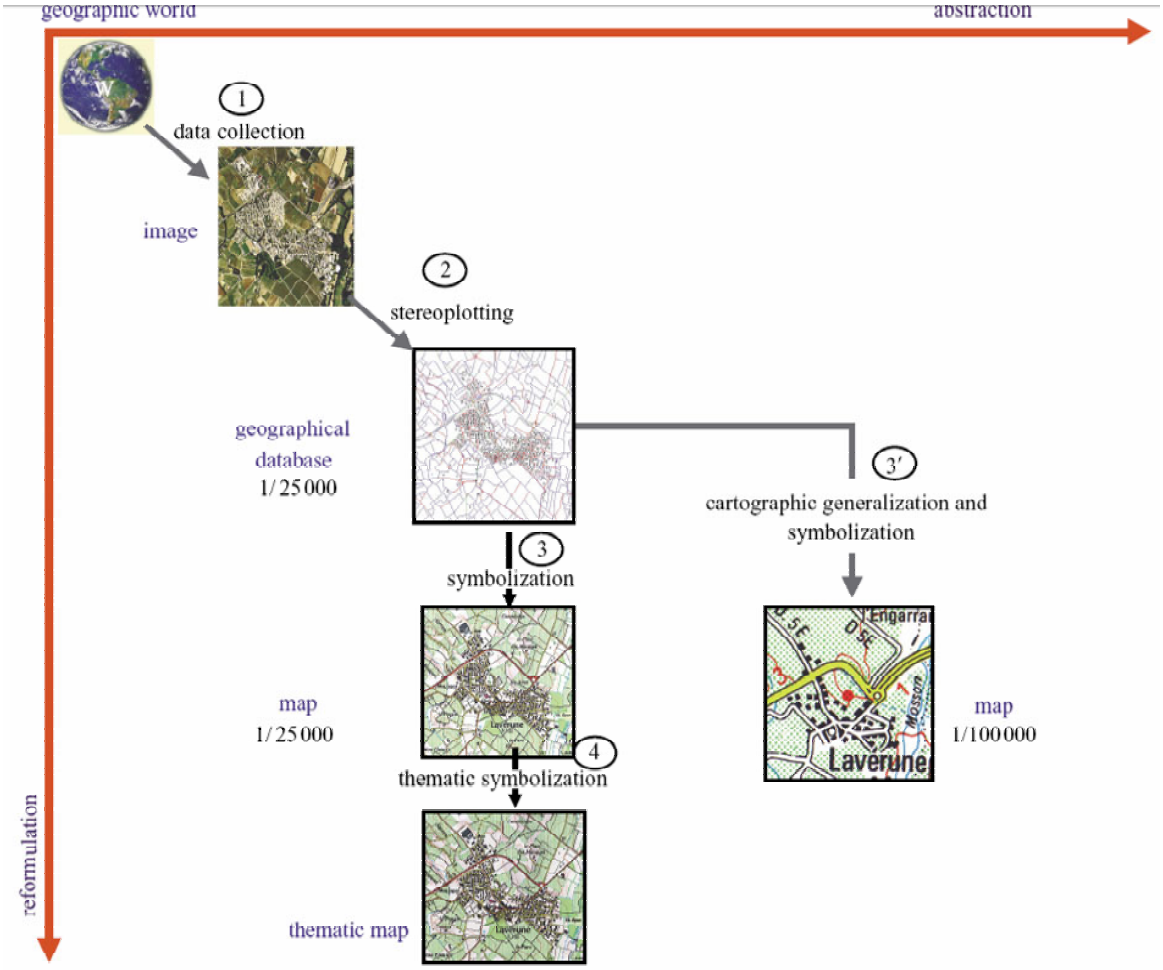


Figure 9: Application of Perception-Based Abstraction in Cartography - from (Zucker, 2003)

This view of abstraction is appropriate to what we have been doing in our research. At the first level, the expert knowledge is acquired and modeled. At the second level, the knowledge is structured into knowledge base components such as problem classes, reduction rules, solution classes and so on. At the third level, the knowledge components are the set of symbols and the operators are defined upon the symbols to construct an instantiated reasoning tree. At each level, there is a corresponding abstraction, but we are

interested of the abstraction of reasoning trees, i.e., the abstraction at the third level. Chapter 3 presents a formal definition of the abstraction of the reasoning trees.

2.2.2. ITS Related Research

The rapid development of an intelligent tutoring system (ITS) has been an important research area. Developing an ITS is notoriously costly and time consuming. In addition to that, the ITS development requires high skills in programming and cognitive science. Therefore it is hard for teachers who do not have experience or skills in computer science to develop such systems. The ITS authoring systems are intended to provide tools that can ease the process of developing an ITS. Murray (2003) classified the authoring systems into seven types:

- *Curriculum sequencing and planning authoring* which focuses on organizing instructional units into a hierarchy of courses, lessons, presentations. Each instructional unit typically has an instructional objective. The content of the tutoring system built by this type usually consists of canned texts and graphics, which is applicable for computer-based learning. The limit of this type of authoring systems is the shallow skill representation (Murray, 2003).
- *Tutoring strategies authoring* which presents diverse tutoring strategies. This type of systems is similar to the curriculum sequencing authoring above in the sense that the content consists of canned texts and graphics. However, it has sophisticated tutoring strategies and "meta-strategies" that select the appropriate tutoring strategies in a given situation. The weakness of this category is also the shallow skill representation (Murray, 2003).

- *Simulation-based learning authoring* which builds a simulation system for tutoring purposes. The expert knowledge in the systems belonging to this category consists of the component locations and operational scripts. The performance monitor and feedback are straightforward such as "You should have checked the safety valve as your next step." The most difficult task in building the tutoring system of this category is building the device simulation. The limits of this type of authoring systems are limited instructional strategies and limited student model (Murray, 2003).
- *Expert systems authoring* which uses rule-based expert system to construct the tutoring systems. The expert systems provide relatively deep domain knowledge and can solve problems. Such systems not only teach but can also help students when stuck to continue next steps or to complete the solution for the entire problem. The weaknesses of this type of authoring systems are the difficulty of building the expert systems, limited to procedural and problem solving expertise and limited instructional strategies (Murray, 2003).
- *Multiple-knowledge types authoring* treats knowledge into different types: facts, concepts and procedures. The tutoring systems built by this type of authoring system tend to treat the knowledge differently. The limits are relatively simple facts, concepts and procedures. It is also limited by the predefined tutoring strategies (Murray, 2003).
- *Special purpose authoring* specializes in particular tasks or domains. The authors are usually given the templates to fill them in. The examples of how to fill in the

blank are given to help the author doing the task. Once the tutor is built by using this type of authoring system, it is only used for that particular task. The limits of this type of systems are each tool is confined in specific type of tutor and the inflexibility of representation and pedagogical knowledge (Murray, 2003).

- *Intelligent/adaptive hypermedia authoring* which builds the web-based tutoring systems. These systems have limited interactivity and student model. These systems are constrained by the bandwidth (Murray, 2003).

In this dissertation, we focus only on the expert system type of authoring tools. Compared to other types of systems, authoring an expert system is particularly complex and time-consuming task (Murray, 1999). Due to that reason, there are only a few such systems available for evaluation or usage. Among them are Demonst8 (Blessing, 1997), Simulated Students (Matsuda et al., 2005), Assistment Builder (Turner et al., 2007), DIAG (Eugenio, 2005) and CTAT (Aleven et al., 2006). We will review them in following subsections to identify their strengths and weaknesses as compared to our new approach.

Demonstr8

One approach to rapid development of a cognitive tutoring system is using the programming by demonstration technique. The basic idea behind this approach is that the demonstrations of how to solve particular problems from the creator are generalized to become the rules for teaching the students how to solve that problem. Demonstr8 is the authoring tutoring system that employs that technique (Murray, 1999). The author can create different tutoring systems by using the provided toolkit to create the user interface

for each system. The higher-order working memory elements (WME) must be defined for that particular interface. Each interface can be associated with multiple WMEs. Each WME can be responsible for a particular feature such as name, number of columns in a subtraction interface, etc. The WME can be created by grouping several interface components or by building the table of values. The most important WME is the problem WME which semantically describes a problem. In the subtraction problem, the WME problem is a table of values, as shown in Figure 10.

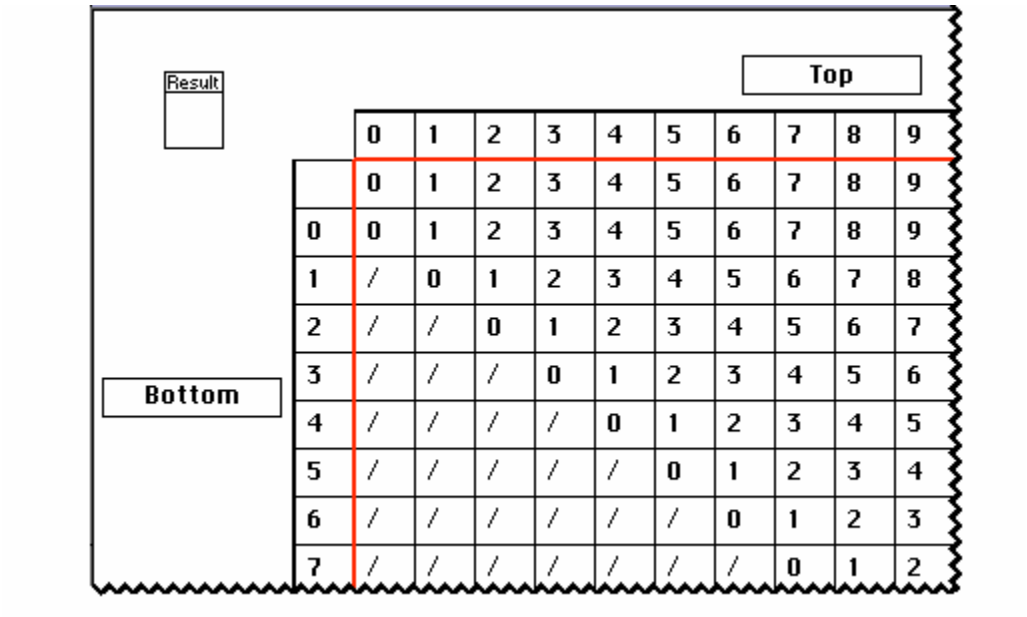


Figure 10: The Subtract Knowledge Function - from (Blessing, 1997)

Once the WMEs associated with the desired user interfaces are constructed, the author can demonstrate the skill to be tutored, and have Demonstr8 induces the underlying production rules (Blessing, 1997). The author first creates an example by using the newly built interface. Then he/she starts solving the problem. This can be done

in two ways: either the author interacts directly with the Knowledge Function, or he/she enters input and output values. The system will induce the production rules based on the WMEs.

Demonstr8 is useful in creating a simple tutoring system to teach simple problems in arithmetic or algebra, but it is difficult to deal with more complex problems. The reason is the system totally relies on the WMEs which are constructed from the interface toolkit. It is difficult to define and solve complex problems by merely manipulating the interface elements. Another reason is that Demonstr8 induces simple production rules from a single example (Jarvis, 2004). That makes it hard to have the production rules cover a broad set of examples, and tends to make them overly general. It also causes the problem for rule refinement, the important process in learning agent. The rule refinement requires the production rules are able to be modified either manually or automatically to cover the exceptions or new conditions. As Matsuda (2005) pointed out, Demonstr8 hard codes pre-defined predicate symbols to specify conditions to fire the rules; hence it is difficult to add conditions or exceptions to these rules. My research is based on a Learning Agent Shell (Tecuci, 1998) which facilitates the knowledge acquisition and refinement. The research also presents a new approach that overcomes the difficulty to deal with complex problems by using the abstraction of reasoning to construct the tutoring lessons for complex domains (see Section 5.1.1).

Simulated Student

Another authoring system named Simulated Student (Matsuda et al., 2005 a), a machine learning agent, is using the programming by demonstration technique. Simulated

Student observes the author's demonstrations of solving a task and induces a set of production rules that replicate the author's performance.

The instructor starts the content construction by building the desired GUI for the Cognitive Tutor using the system's toolkit. The instructor then specifies all the necessary predicate symbols and operator symbols which will be represented in production rules. The operator symbol represents the function which takes parameters as input and produces a single value. The predicate symbol functions as a test for a specific feature. All the symbols are task-dependent and have to be crafted carefully to produce the desired results. Once all the symbols are defined, the instructor presents a few demonstrations by solving a certain numbers of problems. The demonstrations are fed into the Simulated Student and generalized into production rules. The production rules are then loaded back into the Cognitive Tutor with the GUI components. The resulted production rules are tested by trying to solve different problems. Some erroneous rules may be generated and will be corrected by the tutors either by using the GUI component or by modifying the rules directly. Figure 11 illustrates the interface of the Cognitive Tutor.

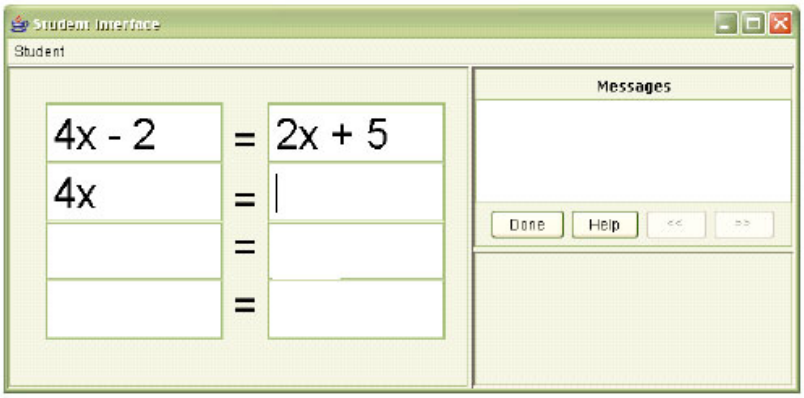


Figure 11: The Interface of a Cognitive Tutor - from (Matsuda, 2005 a)

The production rules define a way to manipulate objects such as buttons, text fields, etc. A production rule consists of three main components: working memory element path, feature tests (the left hand side), and a set of operators (the right hand side) (Matsuda et al., 2005 b). The working memory element (WME) path is the sequence of WMEs from the problem to the current WME. Each WME is associated with a GUI element, so the sequence of GUI elements in solving a problem is captured in the WME path.

The Simulated Student is easy to use when building the small and simple tutoring systems. The learning agent learns the production rules from the demonstrations. This system, however, is plagued by the limitation of the available GUI elements to capture the complex actions during the demonstration. Furthermore, the number of demonstrated problems required to induce the production rules are high. Matsuda et al. (2006) stated that solving ten problems generated nine production rules for algebra equation solving. Therefore to solve a real world problem, there must have been a larger number of problems to be used for demonstration. The other drawback of this system is the machine learning techniques being used in the system. Simulated System uses only the inductive generation which is limited compared to multi-strategies learning mentioned in Section 1.3. The rule revision is also simple, only the manual refinements are available either by GUI manipulation or directly on the production rules. This type of refinement limits the agent to apply the revision to similar production rules in the system. Matsuda et al. (2007) acknowledged that having training on twenty problems, the correctness of the production rules were just 82% which is a slow progress. So far the Simulated Student

has been tested only on some simple domains such as algebra equation solving, long division, multi-column multiplication, fraction addition, chemistry and Tic-Tac-Toe (Matsuda et al., 2007). In other words, it is almost impossible to build an intelligent tutoring system for complex domains such as planning using Simulated Student.

My research is able to overcome the problems presented above. Based on its learning capability, the tutoring system is able to acquire expert knowledge to solve complex problems. Using the abstraction of reasoning facilitates the process of building the curriculums that cover the strategies that has been used in problem solving, even in a complex domain such as Intelligence Analysis (see Section 5.1).

Assistment Builder

The purpose of the Assistment Builder is to help the teacher who has little or no computer science and cognitive psychological background to build a cognitive tutoring system in a relatively short time (Turner et al., 2005). The system is built based on the state graphs which are finite graphs with arcs representing the student's actions and nodes representing the states of the problem's interface. The student's actions change the states of the system which are stored in the state graph. The state graph models the expected behaviors in problem solving, and can predict behaviors as well as provide feedback on them. The instructional scaffolding – a technique to promote learning at different levels, providing sufficient supports at first and reduce them gradually when the students develop their own cognitive or learning skill - is implemented in the system and used to provide the appropriate questions or feedback for students at different levels.

The Assistment Builder builds cognitive tutoring systems over the web. The system implements different tutoring strategies, the scaffolding strategy being one of them. The underlying content representation, the XML schema, defines the problem. The problem consists of an interface definition and a behavior definition, plus some metadata such as problem id and comments (Turner, 2005). The interface definition includes a set of selected widgets, images and texts to present to the students. The behavior definition is the state graph with all the transitions between the nodes. One important type of interface elements is the answerable element. This element is able to capture the student's actions and pass them onto the behavior component in the system. The student's actions then are analyzed against the state graph which represents the expected behaviors. The difference between the expected behavior and student behavior is used to provide appropriate actions.

The behavior acts as the tutoring logic of a problem. It interprets the student's actions which are translated into high-level actions before passing them to the behavior component. Depending on the action and tutoring strategies, different messages can be invoked such as hints, explanations via buggy messages or even scaffolds. If a student's answer is correct, then the problem's state transitions to a new state. Otherwise, the student's actions are mapped to the tutoring strategies and no state transition takes place.

One can use the Assistment Builder to rapidly build a simple tutoring system for a particular problem. There are five types of interface elements that are used to build the content: radio buttons, pull-down menus, checkboxes, text-fields, and algebra text fields that automatically evaluate mathematical expressions. The content builder can assign

only two states to each question in the state graph. An arc that connects the two states allows the student to be moved to the next state if the answer is correct. Along with the interface elements, messages are added to provide the scaffolding questions, hints and feedback. Figure 12 presents a snapshot of a lesson preparation interface.

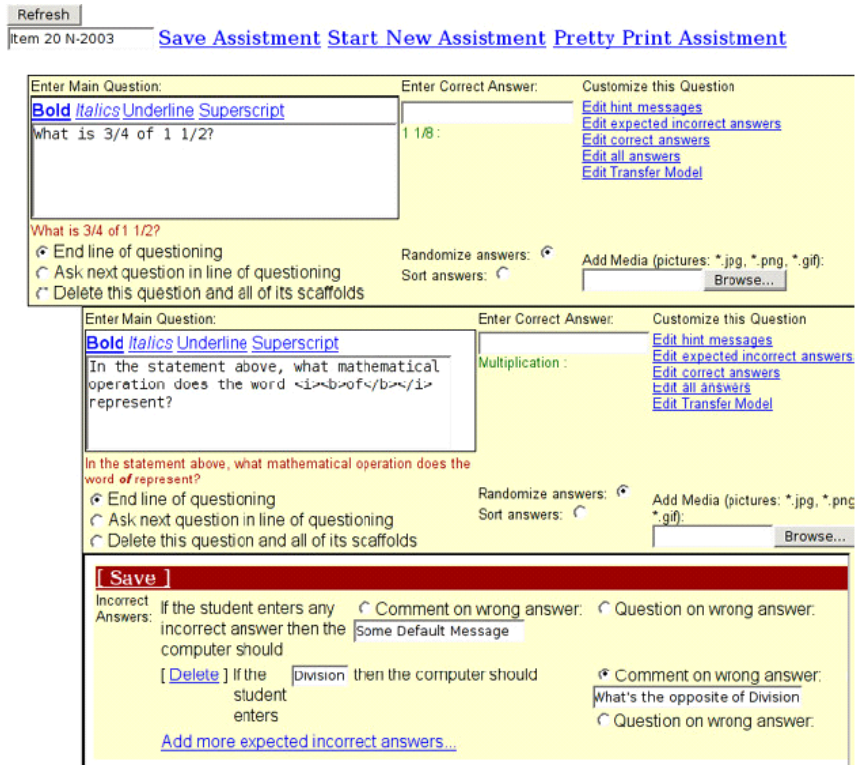


Figure 12: Initial Question, One Scaffold, and Incorrect Answer in Assistment Builder - from (Turner, 2005)

The Assistment Builder is deployed in several schools. Its domains are usually from mathematics such as algebra, geometry. Due to the limit of using simple interface components - radio buttons, pull-down menus, checkboxes, text fields and algebra text fields to evaluate mathematical expressions - to generate the problem, the system is only able to construct a simple pseudo-tutor (Turner et al., 2005). The state graphs for each

question have only binary values and therefore cannot represent complex reasoning. The Assistment Builder does not have a capability to learn rules, and thus cannot construct the tutoring system for problems other than the ones that have been developed specifically by the instructor. In other words, the system does not learn from one problem to generate other problems. Therefore it requires a lot of effort from the instructor who designs the lessons.

The Learning and Tutoring Agent Shell concept (LTAS) that is part of my contribution can learn from the instructor how to generate lessons automatically and adaptively from the content of the knowledge base. That capability can ease the burden of the instructor who designed the lessons (see Section 5.1.2). Furthermore, the tutoring system is constructed based on the abstraction of a complex reasoning tree that is appropriate for complex domains.

DIAG

Diagnostic Instruction and Guidance (DIAG) is an authoring system that uses graphical models to build interactive user interfaces and the lessons. The tutoring systems built with DIAG – also called DIAG applications - are specialized in troubleshooting complex systems, such as heating or circuitry, as shown in Figure 13.

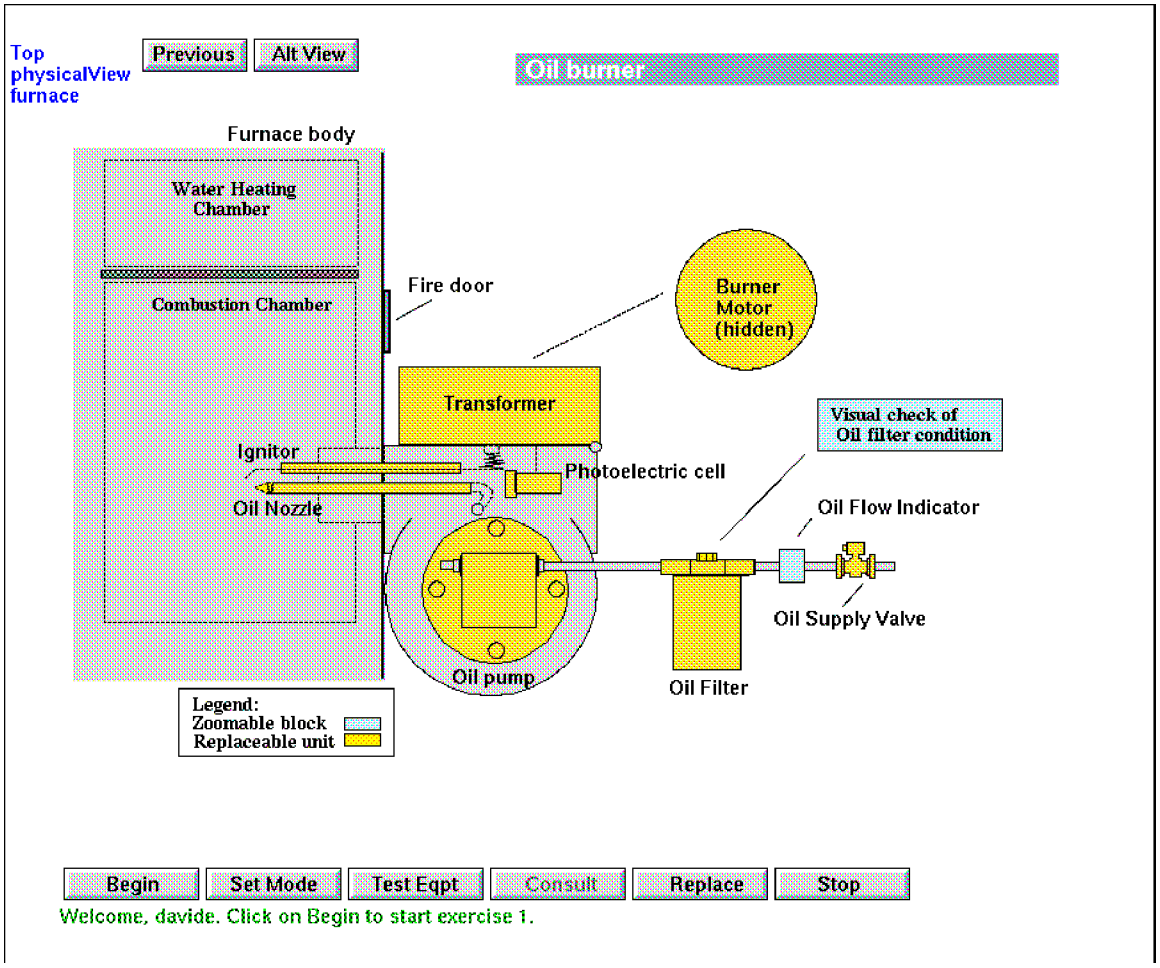


Figure 13: A Scene from a DIAG Application to Oil Burner - from (Eugenio, 2005)

The DIAG application presents a set of scenarios of faulty systems to the students. The student has to figure out the defective components by testing the indicators and inferring what components are not working. Once the problem is identified, the student has to fix the system by using the graphical model. The set of scenarios is presented in a sequence ordered by level of difficulty. The student can ask for help via the Consult menu and the system will generate a context-sensitive hint (Eugenio, 2002).

The main feature of DIAG is the automatic generation of the diagnostic instructions. The steps to author a new diagnostic ITS are (Towne, 1997):

- Create a graphical model of the target system (to be diagnosed) and establish its modes of operation. The graphical model is developed such that it responds to the user's actions and to the failures that are introduced during the troubleshooting exercises. This is structured in a hierarchy fashion of subsystems. This structure allows user to go down to different levels of details. The modes of operation are set by using different combination of switch settings.
- Define the replaceable units (RU) with names, replacement times, and their reliabilities.
- Define the faults in a pool. Specify what impacts the faults can have on the system and provide these statements to the student at the end of the exercises. The details of the statement can vary from simple facts to complex technical details.
- Specify an exercise by selecting a fault from the pool, writing a statement for the beginning of the exercise, setting up the mode that the system is initially in, and setting the time limit to troubleshoot the problem. The author can provide multiple exercises for each fault.
- Produce the symptom data that support the diagnostic reasoning process. DIAG first generates a provisional set of fault effect statements. It does this by simulating each fault and recording the frequency of the various outcomes. The author then refines these statements to reflect his or her own view. Figure 14

illustrates the symptom specifications when a faulty RU occurs. The author can indicate visual symptoms such as the alarm goes off when the fault happens.

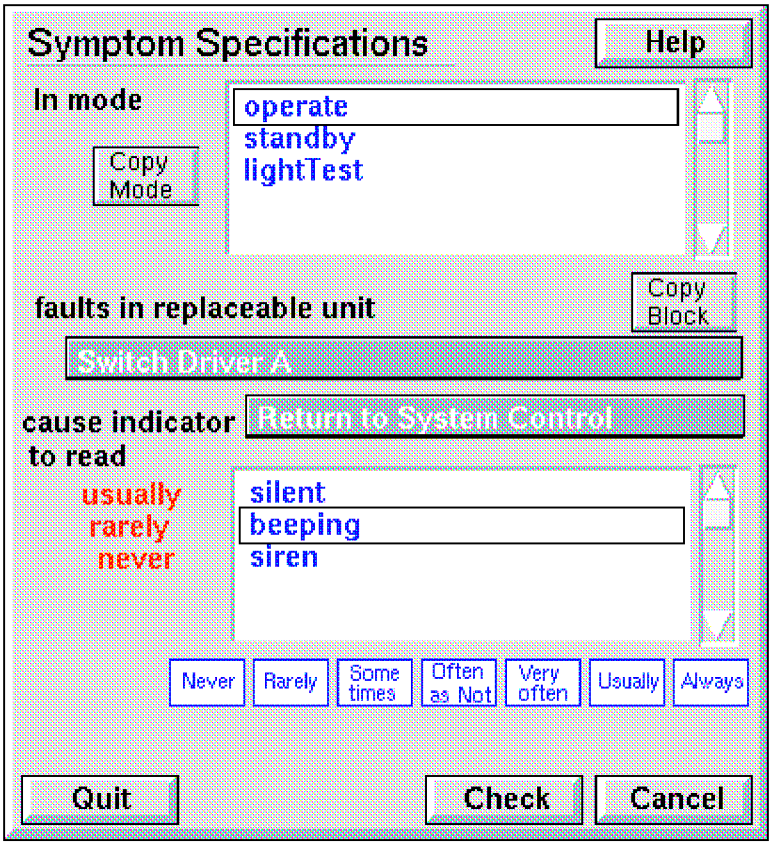


Figure 14: Authoring Interface for Specifying Fault Effects - from (Towne, 1997)

DIAG teaches a student to diagnose faulty systems using clarifications and a highly interactive graphical model. Such a system is useful in occupational training. The problem with this approach is, however, that for each type of the system, a graphical model must be defined with details of faulty statements. No learning capability is

implemented in this approach to save time for lesson preparation. That drawback is addressed in this dissertation (see Section 5.1).

CTAT

Cognitive Tutor Authoring Tools (CTAT) is a tool suite for rapid development of ITSs. CTAT has two types of tutors: *example-tracing tutors* – also named *Pseudo Tutor* - which can be constructed without programming but require problem specific authoring, and *cognitive tutors* which, on the contrary, require AI programming to build a cognitive model but can be used across a range of problems from the same domain. In this research we review the cognitive tutors which can be compared to our approach.

CTAT is based on the ACT-R cognitive theory (Anderson, 1993). This system involves creating a cognitive model of a student's problem solving by using production rules that governs the misconceptions and the different reasoning strategies that a student may use. CTAT consists of the set of tools presented in Figure 15 (Koedinger et al., 2003) such as:

- *GUI Builder* builds the student interface where the student interacts with the tutor. The author uses the tool to build the user interface by dragging-and-dropping the interface widgets on the canvas.
- *Behavior Recorder* records the solution paths of a given problem demonstrated by the author. It has three main functions. First, it builds the *Behavior Graph* which captures the correct or the incorrect demonstrated behavior. Second, it builds the example-tracing function which belongs to the first type of the tutor that CTAT authors. Third, it supports planning and testing of the cognitive model.

- *Working Memory Editor* inspects and modifies the contents of the cognitive model.
- *Production Rule Editor* generalizes the production rules based on the demonstration given by the author.
- *Cognitive Model Visualizer* debugs the production rules.

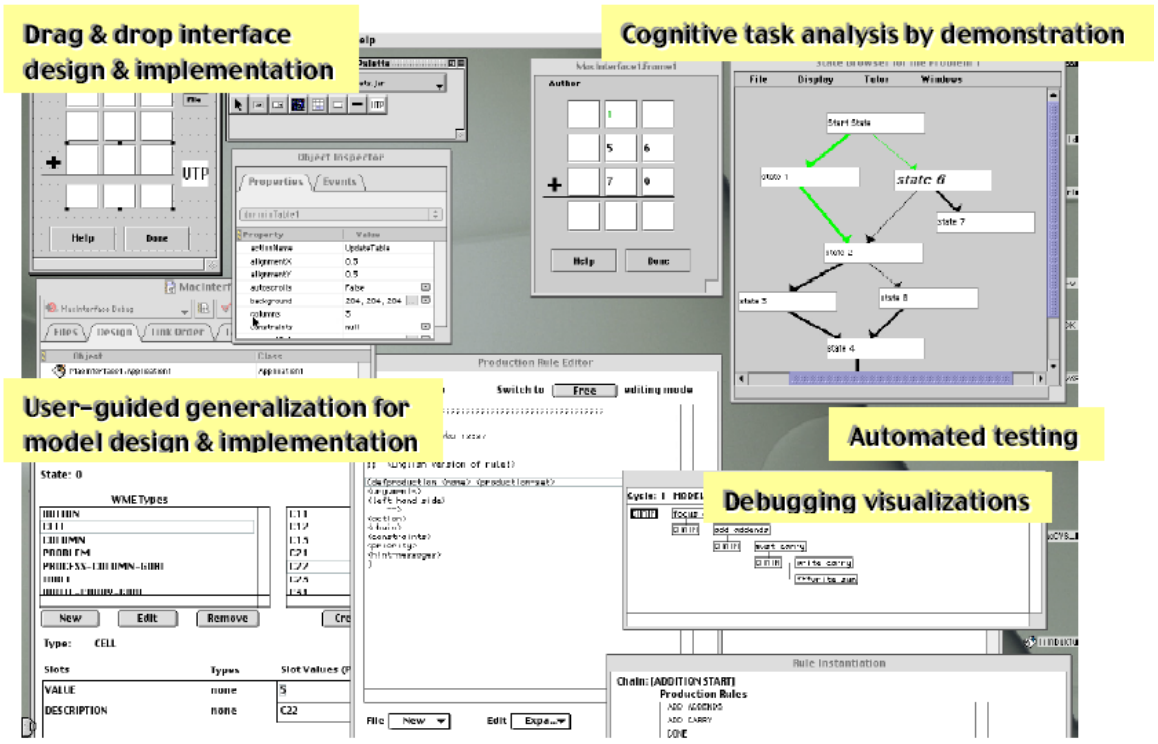


Figure 15: CTAT - from (Koedinger et al., 2003)

The production rule model plays an important role in constructing a Cognitive Tutor. It handles general categories of problems in a specific domain. The module consists of a specification of the objects in "working memory" representing the initial state of the

problem and a set of production rules to transit the objects from one state to the other until the solution of the problem is reached. The development of the production rules is supported by Production Rule Editor which uses the user-guided generalization process. The generalization process starts by the author entering the concrete production rule, and then the editor generalizes the rule by replacing the constants by the variables and adding list matching patterns (Koedinger et al., 2003).

CTAT is quickly generating the Pseudo Tutors which tutor only specific problems. Cognitive Tutors is more interesting in which it can cover similar problems in the same domain, but writing the production rules for CTAT is a time consuming process due to no help from the tool. The author must know Jess – the Java Expert System Shell – and write Jess by hand. Koedinger (2004) stated that it took roughly 200 hour development of Cognitive Tutor for one hour of instruction. CTAT does not have the capability of rule refinement. The production rules once written can not be refined unless the author has to modify the code by himself/herself.

The problem with the CTAT production rules is addressed in our Learning and Tutoring Agent Shell which allows the instructor to build the lessons without writing a single line of code. The shell generates the lesson scripts that underlie the lesson structures designed by the instructor and uses the scripts to construct the lessons automatically (see Section 5.1.3).

3. Abstraction of Reasoning Trees

For real world domains, the formalization of the reasoning process for a given problem is very complex and involves thousands (or even hundred of thousands) of specific reasoning steps. It is very difficult to fully review, understand and verify completely such huge reasoning trees. This section describes an innovative approach of using an abstraction of the reasoning trees in order to significantly simplifying their browsing and understanding.

For instance, the problem "*Assess whether Al Qaeda has nuclear weapons"* from the Intelligence Analysis domain has a reasoning tree of over 1700 nodes. A large tree is hard to be rendered intelligibly on a computer display and therefore hard to comprehend. An abstraction of such a large reasoning tree would help facilitate its browsing and understanding by a user. In addition, a reasoning tree can be partitioned into several sub-trees, based on the corresponding abstract problem solving strategies involved. The abstraction of the reasoning tree in that sense can help identify the abstract reasoning strategies that the expert had used in teaching the agent. Those learned strategies can be reused in solving problems or teaching the students how to solve similar problems. The subsequent sections will explain in detail the methodology of the abstraction of the reasoning tree and its application for problem solving and tutoring assistants.

3.1. Reasoning Tree

A reasoning tree is a special type of tree. Therefore, before defining it, we should take a look at the definition of a tree, as stated by Meyers (1971):

Definition 3.1 (Tree): A tree is a pair $t = (\mathcal{V}_t, \delta_t)$ where $\mathcal{V}_t$ is a finite set of vertices of t and $\delta_t(x)$: $\mathcal{V}_t \rightarrow \mathcal{V}_t^*$ is the *argument function*[1] of t, $\delta_t(x)$ *represents the sequence of children of the vertex* $x \in \mathcal{V}_t$ satisfying the following axioms:

[a] A vertex x cannot be a child of itself and cannot have same child twice: $\forall x \in \mathcal{V}_t$, $\delta_t(x)$ is a (possibly empty) sequence, without repetitions, of elements of $\mathcal{V}_t \setminus \{x\}$

[b] There is only one root vertex, which has no parent: there is one and only one point $r \in \mathcal{V}_t$ (called the root of t) such that for no $x \in \mathcal{V}_t$ is r an argument of x in t. Formally: $\exists! r \in \mathcal{V}_t$, $!\exists x \in \mathcal{V}_t, r \in \delta_t(x)$.

[c] Each node has at most one parent: for $\forall x, y \in \mathcal{V}_t$, $x \neq y$ then $\delta_t(x)$ and $\delta_t(y)$ have no elements in common.

[d] There is only a single path from a node x to the root r: $\forall x \in \mathcal{V}_t, \exists!$ sequence $S = x_k \ldots x_2 x_1$ such that $x_1 = x$, $x_k = r$, $x_i \in \delta_t(x_{i+1})$, $1 \leq i < k$. ■

Notation: If x has n children then $\delta_t(x)$ is defined as $\delta_t(x) = x_1 x_2 \ldots x_n$, in this case, x_i is called the i^{th} argument of x in t (i.e. the i^{th} children); if x has no child then $\delta_t(x) = \lambda$, where λ represents the empty sequence.

[1] The *argument function* of a vertex x is in fact the edge that links x with its n children $x_1 x_2 \ldots x_n$

Definition 3.2 (Valence): For any node $x \in \mathcal{V}_t$, we define $v_t(x) \in Z$ - the *valence* of x in t – to be the length of $\delta_t(x)$ (i.e., the number of children of x in t). A vertex $x \in \mathcal{V}_t$ is called an "endpoint" (*leave*) of t if and only if $v_t(x) = 0$ or $\delta_t(x) = \lambda$, and a "node" of t if and only if $v_t(x) > 0$.■ (Meyers, 1971).

Example of a tree: Figure 16 presents a simple tree which has only six nodes. The tree is denoted as follows: $t = (\mathcal{V}_t, \delta_t)$ where $\mathcal{V}_t = \{a,b,c,d,e,f\}$, $\delta_t(x) = \{a \rightarrow bc,\ b \rightarrow de,\ c \rightarrow f,\ d \rightarrow \lambda,\ e \rightarrow \lambda,\ f \rightarrow \lambda\}$, specifically, $\delta_t(a) = bc$, $\delta_t(b) = de$, $\delta_t(c) = f$, $\delta_t(d) = \lambda$, $\delta_t(e) = \lambda$ and $\delta_t(f) = \lambda$.

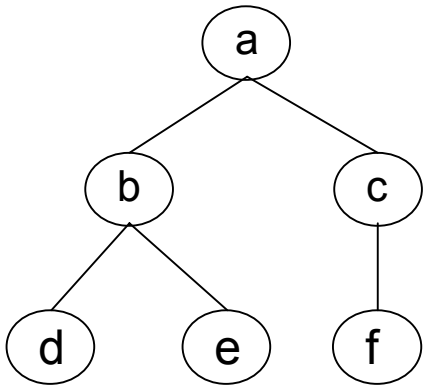


Figure 16: A Simple Tree

Meyers (1971) also defines the sub-tree as follows:

Definition 3.3 (Sub-tree): Given a tree $t = (\mathcal{V}_t, \delta_t)$, and $st = (\mathcal{V}_{st}', \delta'_{st})$, *st* is a sub-tree**,** denoted as $st \in t$ if and only if $\mathcal{V}_{st}' \subset \mathcal{V}_t$ and $\forall x \in \mathcal{V}_{st}'$, $\delta_{st}'(x)$ is obtained from the sequence $\delta_t(x)$ with all elements of $\mathcal{V}_t \setminus \mathcal{V}'_{st}$ deleted. ■ (Meyers, 1971).

Definition 3.4 (Singleton Sub-tree): A singleton sub-tree is a sub-tree that has only one node.

The following definition introduces the natural notation of a tree, which is a refinement of the *isotone* notation developed by (Meyers, 1971):

Definition 3.5 (Natural Notation of a Tree): Let $t = (\mathcal{V}_t, \delta_t)$ be a tree. A natural notation of *t* is a sequence $\mathcal{N}(t) \in (\mathcal{V}_t \times Z)^*$ that satisfies the following properties:

[a] A natural notation is a sequence of the vertices powered by their valences: $\mathcal{N}(t) = x_1^{v_1} x_2^{v_2} \ldots x_n^{v_n}$, where $x_i \in \mathcal{V}_t$, $v_i = v_t(x) \in Z$ is the valence of x_i for i = *1,..,n.*

[b] The parent appears before its children: $\forall x, y \in \mathcal{V}_t$, if $y \in \delta_t(x)$ then $x^{v_t(x)}$ precedes $y^{v_t(y)}$ in $\mathcal{N}(t)$. This property enforces the *prefix notation*.[2]

[c] The children appear from left to right in the sequence: $\forall x, y, z \in \mathcal{V}_t$, if *y* and *z* are the i^{th} and j^{th} arguments of $\delta_t(x)$ and $j=i+1$ (i.e. *y* is the left sibling of *z)* then $y^{v_t(y)}$ is the left sibling of $z^{v_t(z)}$ in $\mathcal{N}(t)$. This property enforces the order of the children from left to right.■

[2] Prefix notation presents the parent before the children

As an example, the natural notation $\mathcal{N}(t)$ of the tree in Figure 16 is $\mathcal{N}(t) = a^2b^2c^1d^0e^0f^0$.

Lemma 3.1: For a given tree *t* there is a unique natural notation $\mathcal{N}(t)$.

The reasoning tree structure utilizes the problem-reduction/solution-synthesis paradigm. A brief overview of this paradigm is presented in the next section.

3.1.1 Problem-Reduction/Solution-Synthesis Paradigm

A general problem solving paradigm is the problem-reduction/solution-synthesis paradigm – this paradigm is also known as "divide and conquer" or "problem decomposition" (Durham, 2000; Powel and Schmidt, 1998; Tecuci, 1998). In this paradigm, which is illustrated in Figure 17, a complex problem is successively reduced to simpler problems via the reduction operators. The reduction continues until elementary problems are reached for which there are known solutions. Then the synthesis process begins to synthesize all the solutions successively from the simplest problems upwards via the synthesis operators, until a solution is found for the original problem.

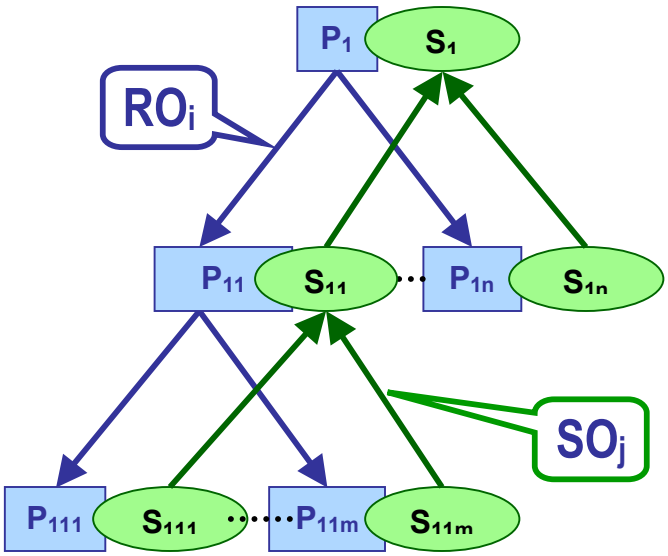


Figure 17: Problem-Reduction/Solution-Synthesis Paradigm

In the illustration from Figure 17, the solution of problem P_1 is obtained by reducing that problem into *n* simpler problems P_{11}… P_{1n}, via the reduction operators RO_i. Each problem then is reduced into simpler problems. For instance, P_{11} is reduced to P_{111}… P_{11m}. P_{1n} is not reduced further because it has its solution S_{1n}. Once the solutions S_{111}…S_{11m} of the sub-problems P_{111}…P_{11m} are obtained, the synthesis starts by combining S_{111}…S_{11m} into the solution S_{11} of problem P_{11} via the synthesis operators SO_j. The process continues until the final solution S_1 of original problem P_1 is found.

This paradigm has been successfully applied in a wide variety of problems such as course of action critiquing (Tecuci et al., 2001), intelligence analysis (Tecuci et al., 2005), planning (Sebastia et al., 2006), requirements engineering (Maiden and Sutcliffe, 1996), to name a few. As demonstrated in (Barr et al., 1998) the problem reduction

representation of the problem solving process is equivalent with the state-space search representation, and most of the problems can be represented using the state-space representation.

3.1.2 Question-Answering Based Problem-Reduction

In order to facilitate the knowledge acquisition and problem solving processes, the problem reduction paradigm was refined by introducing a question and an answer to guide each reduction. The question considers relevant aspects of the problem to be reduced and the answer suggests how to reduce it (Bowman et al., 2001), as shown in Table 1.

Table 1: A Question-Answering Based Reduction Step

- Assess the credibility of Osama Bin Laden as the source of EVD-Dawn-Mir01-02c.
- Q: What factors determine the credibility of Osama Bin Laden as the source of EVD-Danw-Mir01-02c?

  A: The veracity, objectivity and observational sensitivity of Osama Bin Laden because EVD-Dawn-Mir01-02c is testimonial evidence based upon the direct observation.
- Therefore one has to:
  - Assess the veracity of Osama Bin Laden with respect to the information provided in EVD-Dawn-Mir01-02c.
  - Assess the objectivity of Osama Bin Laden with respect to the information provided in EVD-Dawn-Mir01-02c.

- Assess the observational sensitivity of Osama Bin Laden with respect to the information provided in EVD-Dawn-Mir01-02c.

In this question-answering based problem-reduction paradigm, an application domain is modeled based on the following main types of knowledge elements: objects (concepts and instances), features and facts, problems, solutions, examples, explanations and rules (Tecuci et al. 1999).

- Concepts represent sets of individuals. An example of the concept is "*evidence"*.
- Instances are the instantiations of concepts in a specific scenario. For example, an instance of *evidence* in *Intelligence Analysis* is "*EVD-Dawn-Mir01-02c*[3]*"*.
- Objects represent individuals or set of individuals in the application domain that are organized hierarchically in an ontology. An object can be a concept or an instance.
- Features are to describe further the objects, problems and other features. Each feature has a domain and a range. The domain of a feature is the set of objects that can have that feature and the range is the set of possible values of that feature. The features are hierarchically organized. An example of a feature is "*has as description"* whose domain is "*evidence"* and range is "*any string"*.

[3] EVD-Dawn-Mir01-02c is a fragment of an article by Hamid Mir, published in Dawn, a Pakistani magazine.

- Facts are features with specific values. An example of fact is "*EVD-Dawn-Mir01-02c has as description 'We have chemical and nuclear weapons as a deterrent and if America used them against us we reserve the right to use them."*
- Problems represent anything that the agent attempts to solve. An example of a problem is:
- *Assess the credibility of ?O1 as the source of ?O2 (?O1* and *?O2* are variables that can be instantiated to a person and an evidence, respectively).
- A problem with instantiated variables is called an *instantiated problem*. The problem that is illustrated in the top part of Figure 18 is a part of a problem reduction rule.

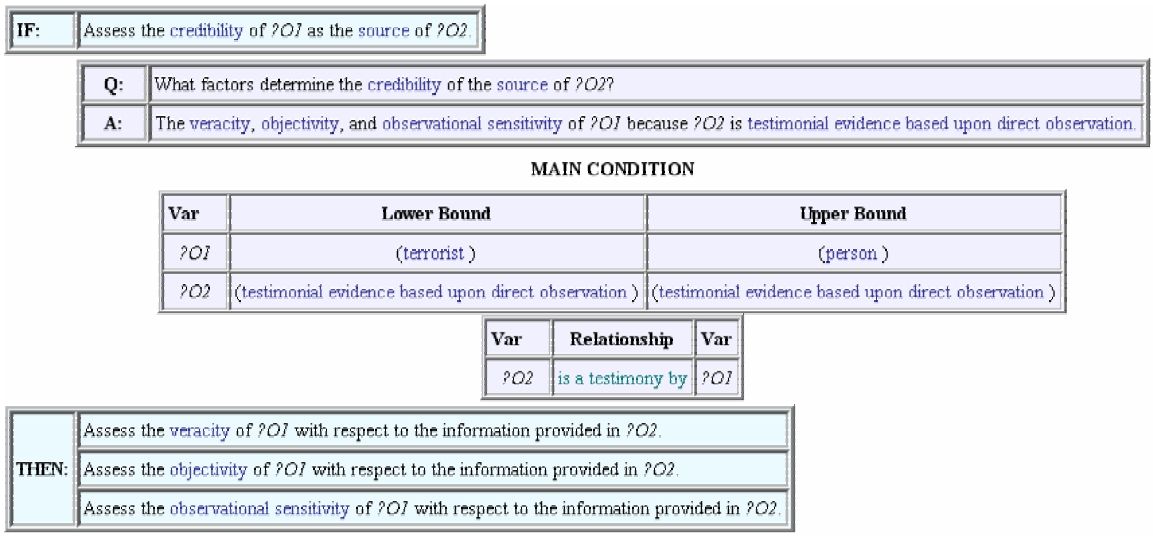


Figure 18: Reduction Rule

- Solutions are associated with the problems. An example of a solution is "*The credibility of Osama Bin Laden as the source of EVD-Dawn-Mir01-02c is an even chance"*.
- Examples are the instances of problem reduction and solution synthesis steps. An example can be negative or positive. A negative example represents an incorrect problem reduction step and a positive example represents a correct problem reduction step. A positive example of problem reduction step is the one from Table 1.
- Explanation is the justification of why a problem reduction step or a solution synthesis steps is correct or incorrect. An explanation is expressed as a set of facts, called explanation pieces. The explanation pieces for the problem reduction example in Table 1 are:
  - *EVD-Dawn-Mir01-02c is testimonial evidence based upon direct observation.*
  - *EVD-Dawn-Mir01-02c is a testimony by Osama Bin Laden.*
  - *Osama Bin Laden is a terrorist.*
- Rules are generalizations of problem reduction or solution synthesis steps. For instance, Figure 18 shows the rule which is a generalization of the problem reduction step in Table 1. As with a general problem, a rule can be instantiated to different reduction steps.

3.1.3 Reduction and Synthesis Process

During problem solving, a reasoning tree is created by using the knowledge elements described in the previous section. This tree is "*a natural and explicit representation of the*

thread of logic of the analyst, as if he or she would be thinking aloud" (Tecuci et al., 2005). The reasoning tree hierarchically represents the discrete steps in the problem solving process based on the problem reduction paradigm. The root of the tree indicates the problem to be solved. The tree is basically composed of successive sequences of problem – reduction – sub-problems, which are represented by corresponding sequences of problem nodes - reduction nodes – sub-problem nodes. The reasoning tree consists of instantiated problems and instantiated reduction rules or reduction examples. Therefore the reasoning tree represents an instantiated reasoning process.

The Figure 19 illustrates a fragment of an instantiated reasoning tree for assessing the credibility of *Osama Bin Laden* as the source of testimonial evidence *EVD-Dawn-Mir01-02c*. (a statement made by *Osama bin Laden* in an interview). The reasoning tree leads to the assessing of three main components of the credibility. A solution for each of them is found. Then these solutions are composed, from bottom up, as illustrated in Figure 19 and Figure 20.

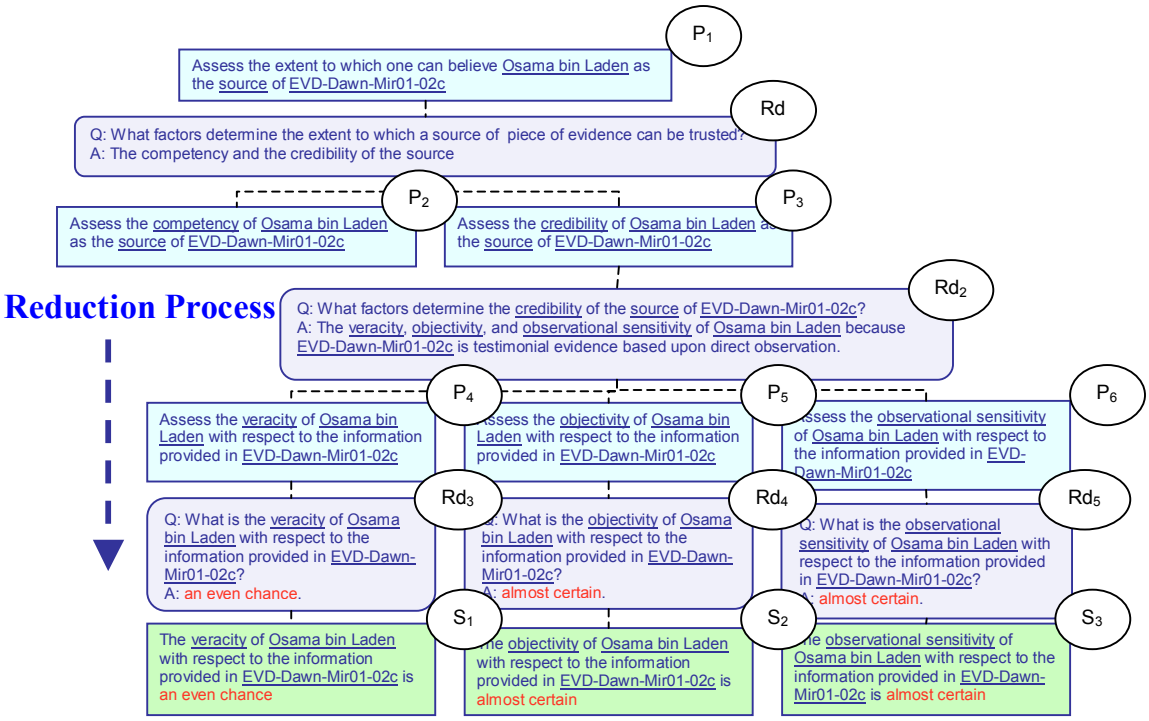


Figure 19: Hypothesis Analysis through Problem Reduction

The solution of a problem is obtained from the synthesis of the solutions of its sub-problems. The synthesis starts from the assessed veracity, objectivity and observational sensitivity of *Osama bin Laden* (i.e. an *even chance, almost certain* and *almost certain*, respectively). The process goes upward until the solution of the top problem is found (which is the assessed believability of *Bin Laden*). The synthesis of the solutions is based on certain synthesis rules acquired from a subject matter expert. In the example from Figure 20, the credibility of *Osama bin Laden* (i.e. *"an even chance"*) is obtained as the minimum of his veracity, objectivity and observational sensitivity (i.e. "*an even chance"*). Similarly, the believability of *Bin Laden* as the source of *EVD-Dawn-Mir01-*

02c is obtained as *"an even chance"*, the minimum between his competence and his credibility[4].

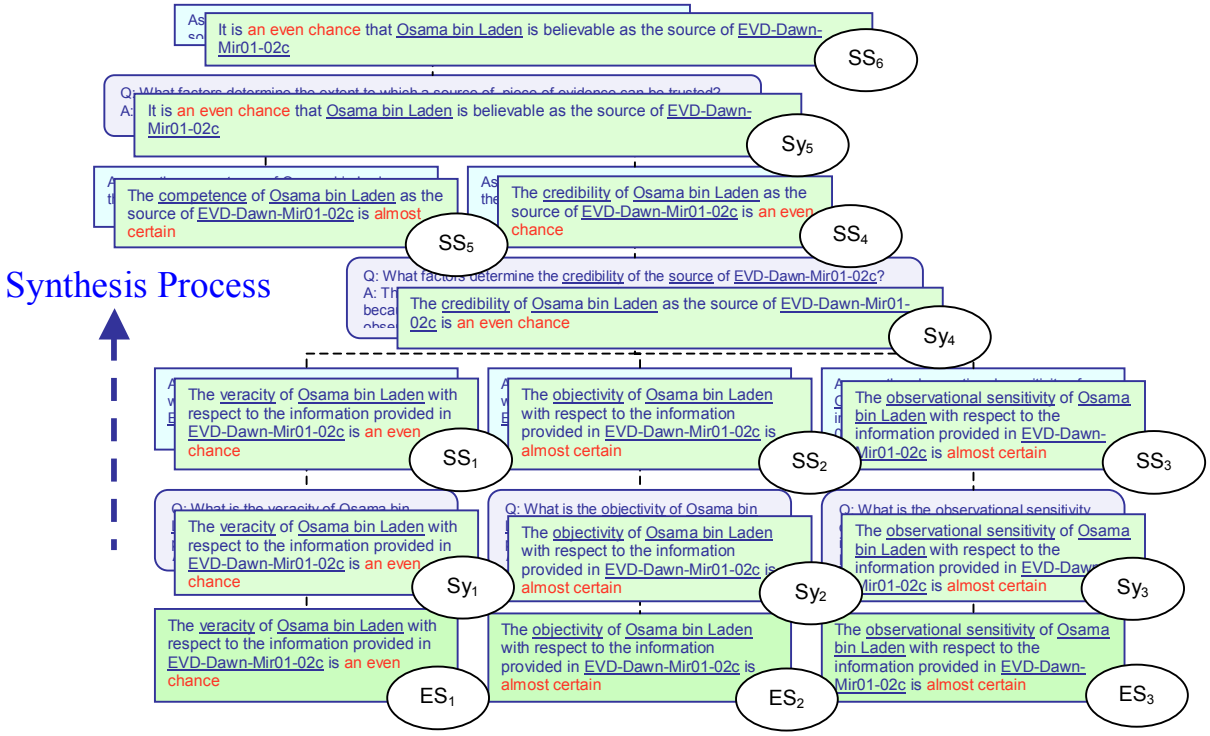


Figure 20: Hypothesis Analysis through Solution Synthesis

Due to the fact that the reduction and synthesis processes are synchronized, Figure 20 also indicates the correlation between the reduction process and the synthesis process. Each problem in the tree (cyan rectangle) is associated with a synthesized solution (light green rectangle). The question/answer pair from a reduction step (round cyan rectangle) is associated with a synthesis from a synthesis step which synthesizes the sub-solutions to a solution (sub-solution is a solution of a sub-problem).

[4] The synthesis of the solutions can be performed through different strategies as indicated by the expert who teaches the agent.

From what we presented above, a reasoning tree in the problem reduction/solution synthesis paradigm can be seen as consisting of two isomorphic trees: the reduction tree and the synthesis tree. The reduction tree shows how the top-level problem is reduced to simpler sub-problems until the elementary solutions are found for the simplest problems. The synthesis tree shows how the elementary solutions are composed to the solution of the original problem. Because they are isomorphic to each other, we will provide only the definitions for a reduction tree. The definitions for a synthesis tree are similar to those for a reduction tree.

A reduction tree *t* is formally defined as follows:

Definition 3.6 (Reduction Reasoning Tree): A tree $t = (\mathcal{V}_t, \delta_t)$ is a reduction reasoning tree – a.k.a. *reduction tree* - if the following properties are satisfied:

[a] There are three types of reasoning nodes named *problem nodes, reduction nodes* and *solution nodes.* We denote the reasoning nodes as follows:

- $\mathcal{P}_t$ is the set of problem nodes in the tree *t.*
- $\mathcal{Rd}_t$ is the set of reduction nodes in the tree *t.*
- $\mathcal{S}_t$ is the set of solution nodes in the tree *t.*

By definition, $\mathcal{V}_t = \mathcal{P}_t \cup \mathcal{Rd}_t \cup \mathcal{S}_t$. A vertex $v \in \mathcal{V}_t$ is also called reasoning node or simply node.

[b] The root is a problem node: $Root(t) \in \mathcal{P}_t$. It represents the top level problem.

[c] The reasoning nodes are connected together by the argument function $\delta_t(x)$ which is defined using the following functions: δ_{tP} represents the connection from a problem node to its reduction children nodes, and δ_{tRd} represents the connection from a reduction node to its problem or solution children nodes.

$$\delta_t(x) = \begin{cases} \delta_{tP}(x) & , x \in \mathcal{P}_t \\ \delta_{tRd}(x) & , x \in \mathcal{R}d_t \\ \lambda & , x \in \mathcal{S}_t \end{cases}$$

Where:

- δ_{tP}: $\mathcal{P}_t \rightarrow \mathcal{R}d_t$* indicates that a problem node can be either a leaf of a tree or can be further reduced to reduction nodes.
- δ_{tRd}: $\mathcal{R}d_t \rightarrow [\mathcal{P}_t \cup \mathcal{S}_t]^+$ indicates that a reduction node can be reduced further to problem nodes and/or solution nodes.
- $\delta_t(x) = \lambda$ for $x \in \mathcal{S}_t$: indicates that the solution node is the leaf of the reasoning tree.■

Notations:

- If there are more than one tree, the superscript *(i)* is used where $0 \leq i \leq n$ for trees and their components. For example, a list of n trees are denoted as $t^{(0)} = \left(\mathcal{V}_t^{(0)}, \delta_t^{(0)}\right)$, $t^{(1)} = \left(\mathcal{V}_t^{(1)}, \delta_t^{(1)}\right)$, ..., $t^{(n)} = \left(\mathcal{V}_t^{(n)}, \delta_t^{(n)}\right)$.
- A tree t with root r can be notated as t_r.
- A tree t with root r and leaves $\{n_1, n_2, ..., n_3\}$ can be notated as $t_{[r|\{n1,n2,...n3\}]}$.

- A node x_1 which is the parent of node x_2 is denoted as $x_1 = Parent(x_2)$.

Remark: The root of a sub-tree *st* of a tree *t* is not necessary a problem node, it can be any type of node.

A reduction reasoning step consists of a problem, a question/answer pair and one or several sub-problems, as shown in Table 1. Similarly, a synthesis reasoning step consists of a set of sub-solutions, a question/answer pair and a solution synthesized from the sub-solutions. The association between a reduction step and its counterpart synthesis step is a one-to-one relationship. The reduction and synthesis reasoning steps can partition a tree into several smaller sub-trees which are as functional as the original tree. For instance, Figure 20 shows a tree which is by itself a sub-tree of a larger reasoning tree. This sub-tree contains five reduction reasoning steps and five corresponding synthesis reasoning steps. The sub-trees are trees themselves. This observation is the foundation of the operations of the reasoning tree abstraction.

Definition 3.7 (Reduction Reasoning Step): A reduction reasoning step in a reasoning tree *t,* is a sub-tree $rs = (\mathcal{V}_{rs}, \delta_{rs})$ of *t*, satisfying the following properties:

[a] The root node of the reduction step is a problem node, named the problem node of the reduction step, and denoted with $P_{rs} \in \mathcal{P}_t$.

[b] The reduction step must contain only one reduction node, child of the reasoning step problem node, named the reduction of the reasoning step, and denoted with $Rd_{rs} \in \mathcal{Rd}_t : Rd_{rs} \in \delta_{tP}(P_{rs})$.

[c] The reduction step will contain all the children of the reasoning step reduction node, named sub-nodes of the reduction step and denoted with $SN_{(i)rs}$, $i=1,n$: $\delta_{tRd}(Rd_{rs})=SN_{(1)rs}\ SN_{(2)rs}\ ..\ SN_{(n)rs}$. A sub-node can be a sub-problem node or a solution node.

[d] There are no other nodes in a reasoning step: $\mathcal{V}_{rs}=\{ P_{rs}, Rd_{rs}, SN_{(1)rs}, SN_{(2)rs}, ... SN_{(n)rs}\}$.■

Example (Reduction Reasoning Step) Figure 21 shows an example of a reduction reasoning step. Its formalization is:

$\mathcal{V}_{rs}=\{P_0, R_0, P_1, P_2, P_3\}$

$\delta_{rs}(x) = \{P_0 \rightarrow R_0,\ R_0 \rightarrow P_1P_2P_3,\ P_1 \rightarrow \lambda\ ,\ P_2 \rightarrow \lambda\ ,P_3 \rightarrow \lambda\ \}.$

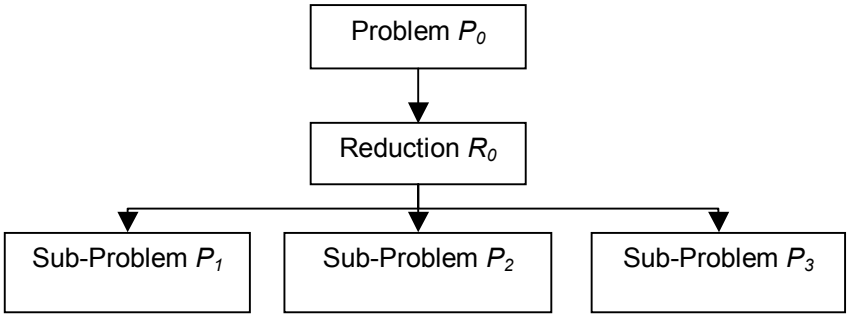


Figure 21: Reduction Reasoning Step

Until now, the definitions of the tree in general and reasoning tree in particular have been presented thoroughly. Next we will present the abstraction of a tree. This will be the foundation of two types of abstraction that are applied to collaborative problem solving and to tutoring.

According to Giunchiglia and Walsh (1992), the abstraction is "the process of mapping a representation of a problem, called the "*ground*" representation, onto a new representation, called the "*abstract*" representation" (Giunchiglia and Walsh, 1992). In this dissertation we focus on the abstraction of the reduction tree. Based on Giunchiglia definition, we will use the term reduction tree at "*ground level"* as the initial (concrete) reduction tree and reduction tree at "*abstract level"* as the abstracted reduction tree. Reduction tree is a specific representation of tree (see Definition 3.6). It is possible to have numerous ways to abstract a reduction tree; each type of abstraction will result in different abstract reduction tree. We will consider two types of abstractions that are suitable for our considered representations. Both types share a common definition of abstraction as presented below.

3.2. Abstraction of a Tree

Definition 3.8 (Partition): A partition of tree t, $Partition_t$ is a set of sub-trees st of tree t for which $\forall x \in V_t$, $\exists !\ st \in Partition_t$ such that x $\in V_{st}$.

Definition 3.9 (Singleton Partition): A singleton partition is a partition that has only singleton sub-trees.

Definition 3.10 (Root of Partition): A sub-tree st_r is a root of a partition $Root(Partition_t) = st_r$, if and only if $st_r \in Partition_t$ and $Root(t) \in V_{st}$.

Definition 3.11 (Parent Sub-tree): A sub-tree st_1 is a parent sub-tree of sub-tree st_2 $st_1 = Parent(st_2)$ if and only if $\exists x \in V_{st1}$, $Root(st_2) \in \delta_t(x)$.

Definition 3.12 (Tree Abstraction): We define the abstraction of a tree at ground level $t = (V_t, \delta_t)$ (*ground tree*) to be the tree at abstract level $t_a = (V_{ta}, \delta_{ta})$ (*abstract tree*),

if there is a partition of t $Partition_t$ and an abstraction function α such that α: $Partition_t \rightarrow Vt_a \cup \{\lambda\}$. The abstraction function α satisfies the following properties:

[a] The abstraction of the root of the ground tree must be the root of the abstract tree. For the root sub-tree $st_r = Root(Partition_t)$, $\alpha(st_r) = Root(t_a)$.

[b] The parent-child relationships of nodes of the ground tree are preserved in the abstract tree. If $st_1, st_2 \in Partition_t$ such that st_1 is the parent sub-tree of st_2 then

- if $\alpha(st_1) \neq \lambda$ then $\alpha(st_2) \in \delta_{ta}(\alpha(st_1))$ or $\alpha(st_2) = \lambda$
- if $\alpha(st_1) = \lambda$ then $\alpha(st_2) = \lambda$

[c] The sibling relations of the nodes of the ground tree are partially preserved in the abstract tree, i.e., $\forall v_1, v_2 \in \mathcal{V}_{ta}$, v_1 is left sibling of v_2 if and only if $\exists st_1 \in Partition_t$ such that $v_1 = \alpha(st_1)$ and $\forall st_2 \in Partition_t$, $v_2 = \alpha(st_2)$, st_1 is left sibling of st_2.

[d] Any abstract node is the abstraction of at least one concrete sub-tree. $\forall x \in V_{ta}, \exists st \in Partition_t, \alpha(st) = x$. ■

Definition 3.13 (Complete Abstraction): An abstraction is called *complete abstraction* if and only if all the sub-trees of the ground tree t have abstractions in the abstract tree t_a. $\forall st \in Partition_t, \alpha(st) \neq \lambda$.

In the next two sections, we will focus on two different types of abstraction of reasoning trees that are suitable for two different purposes: collaborative problem solving and tutoring problem solving strategies. We will introduce the concepts of the two abstractions and then will provide the detailed definitions for both of them.

3.3. Abstraction of Reasoning Trees for Collaborative Problem Solving

As mentioned above, a very large reasoning tree is difficult to view and understand. An abstraction of complex reasoning tree that partitions the complex tree into meaningful and manageable sub-trees is desirable. Once the tree is partitioned into smaller but manageable sub-trees, the browsing of the concrete tree now is facilitated by its abstract tree. Figure 22 shows how a complex tree can be partitioned, abstracted and presented as table of contents.

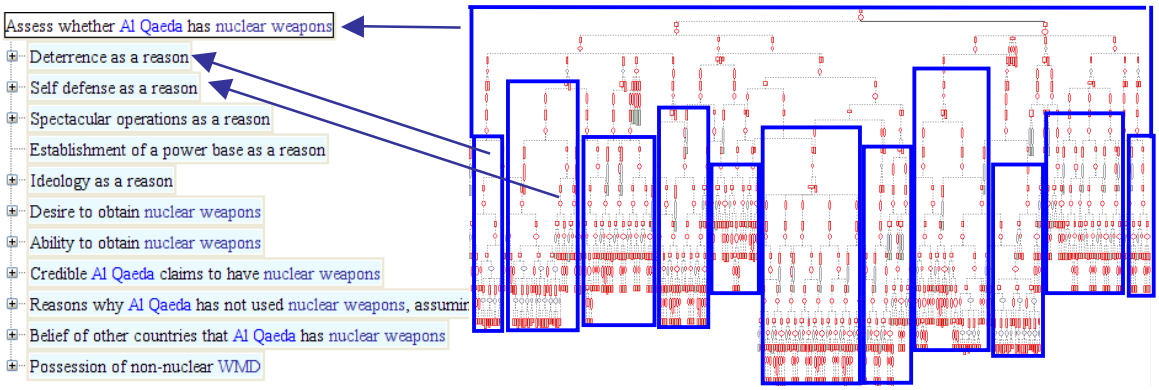


Figure 22: Partition of a Reduction Tree

In order to abstract a reasoning tree for collaborative problem solving, the tree must be partitioned into several distinct sub-trees. Each sub-tree is abstracted into an abstract node in abstract reasoning tree. Consider the example in Figure 23, where a fragment of a concrete reasoning tree is partitioned into five sub-trees:

- $st_1^{(0)}$ where $V_{st1}^{(0)} = \{P_1^{(0)}, Rd_1^{(0)}, P_3^{(0)}, Rd_2^{(0)}\}$, $Root(V_{st1}^{(0)}) = P_1^{(0)}$,
- $st_2^{(0)}$ where $V_{st2}^{(0)} = \{P_2^{(0)}\}$, $Root(V_{st2}^{(0)}) = P_2^{(0)}$,

- $st_3^{(0)}$ where $V_{st3}^{(0)} = \{P_4^{(0)}, Rd_3^{(0)}, S_1^{(0)}\}$, $Root(V_{st3}^{(0)}) = P_4^{(0)}$,
- $st_4^{(0)}$ where $V_{st4}^{(0)} = \{P_5^{(0)}, Rd_4^{(0)}, S_2^{(0)}\}$, $Root(V_{st4}^{(0)}) = P_5^{(0)}$, and
- $st_5^{(0)}$ where $V_{st5}^{(0)} = \{P_6^{(0)}, Rd_5^{(0)}, S_3^{(0)}\}$, $Root(V_{st5}^{(0)}) = P_6^{(0)}$.

These five partitions are abstracted into five abstract nodes $P_1^{(1)}$, $P_2^{(1)}$, $P_3^{(1)}$, $P_4^{(1)}$ and $P_5^{(1)}$ respectively. The abstract nodes form an abstract tree which represents an abstraction of the concrete reduction tree.

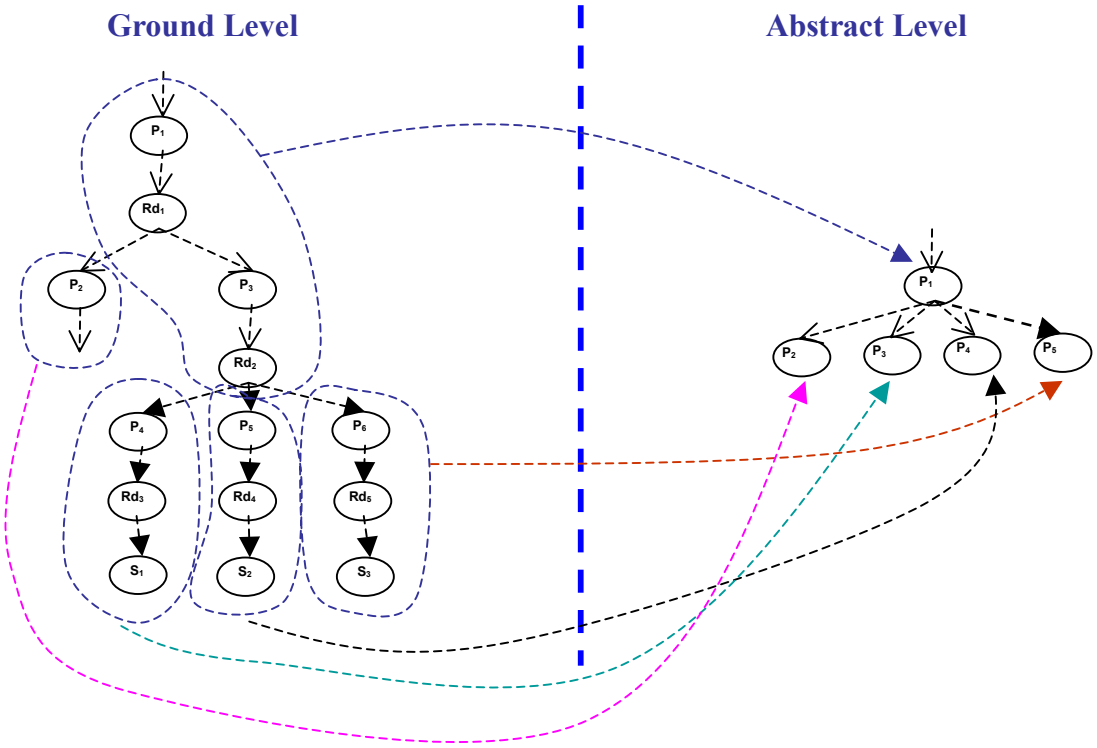


Figure 23: Abstraction of a Reduction Tree for Collaborative Problem Solving

From the example presented above, we can define the abstraction of a reasoning tree for collaborative problem solving based on the common Definition 3.12. This is a special

type of abstraction where concrete sub-trees are abstracted into abstract nodes of the abstract tree. The sub-trees are defined to have the problem nodes as their roots. The definition of this abstraction is formally presented as follows:

Definition 3.14 (Tree Abstraction for Collaboration Problem Solving): We define the abstraction for collaborative problem solving of a reasoning tree at ground level $t = (\mathcal{V}_t, \delta_t)$ (*ground tree*) to be the tree at abstract level $t_a = (\mathcal{V}_{ta}, \delta_{ta})$ (*abstract tree*) is the abstraction for collaboration problem solving, if the $Partition_t$ will contain sub-trees having problem nodes as roots ($\forall st \in Partition_t, Root(st) \in \mathcal{P}_t$) and the abstraction function is a bijective complete abstraction function $\alpha: Partition_t \rightarrow V_{ta}$.

3.4. Abstraction of Reasoning Trees for Tutoring

The abstraction of a reasoning tree for tutoring purpose is different from that for collaborative problem solving presented above. The purpose of this type of abstraction is to present the problem solving strategies that are used to reduce the top problem to the simplest problems in the reasoning tree.

The abstraction of a reasoning tree results in an abstract reasoning tree. The abstract reasoning tree is simpler to view quantitatively and more organized semantically. Each node of the abstract reasoning tree is the abstraction of a set of related nodes of the concrete reasoning tree. Figure 24 shows a concrete reasoning tree on the left panel and the corresponding abstract reasoning tree on the right panel. The former has more than 1700 nodes and the latter has only over 130 nodes which is a 92.5% reduction in number of nodes. Furthermore, the content of an abstract node – a node of the abstract reasoning tree – is problem solving strategy oriented. For example, the yellow node in the abstract

reasoning tree describes the strategy "*Reduce the hypothesis to simpler hypothesis"* which is essentially the principle of the problem reduction paradigm. This yellow node is the abstraction of a set of 54 nodes bordered by the broken blue line. The set of concrete reasoning nodes include different types of reasoning nodes (such as problem nodes, reduction nodes) and different hypotheses (such as "*Assess whether Al Qaeda has reason to use the nuclear weapon"* and the opposite one "*Assess whether Al Qaeda has reason not to use nuclear weapons, assuming that it has them.")* The hypotheses are further reduced to simpler ones, according to the content of the abstract yellow node of the abstract reasoning tree.

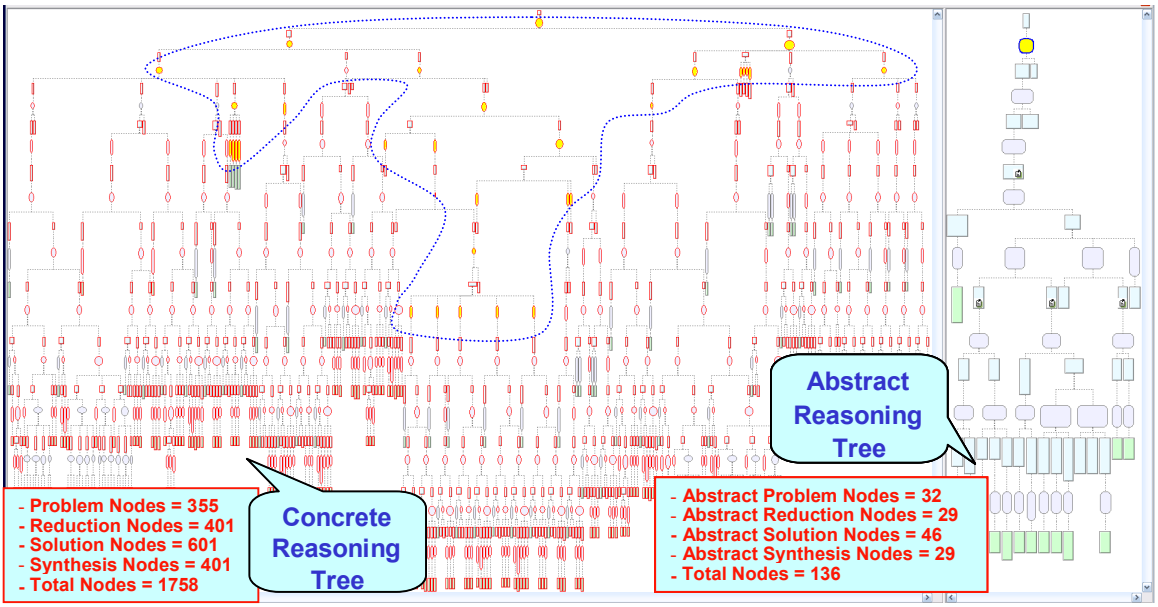


Figure 24: Concrete Reasoning Tree and Its Abstraction for Tutoring

3.4.1. Abstract Problem

The abstract reasoning tree organizes the problem solving strategies in such a way that the tree itself becomes an explicit elucidation of the problem solving methods based on the problem-reduction/solution-synthesis paradigm. An abstract problem node of the abstract reasoning tree represents an abstract problem. The root of the abstract tree is an abstract problem node which specifies the most general problem to solve such as "*Assess a hypothesis"*. The most general problem is also reduced further using the problem reduction paradigm. The reductions in the abstract reasoning tree correspond to those of the concrete reasoning tree. For instance, the abstract problem "*Assess a hypothesis"* which corresponds to the first problem (root) of the reasoning tree "*Assess whether Al Qaeda has nuclear weapons*", is reduced to the more specific problem "*Assess a hypothesis through evidence analysis* which corresponds to the problem "*Assess whether Al Qaeda has desire to obtain nuclear weapons*" in the concrete reasoning tree.

Let us consider the two problems in the reasoning tree:

- "*Assess to what extent the piece of evidence EVD-Dawn-Mir01-01a favors the hypothesis that Al Qaeda considers self defense as a reason to obtain nuclear weapons"* and
- "*Assess to what extent the piece of evidence EVD-Glazov01-01c favors the hypothesis that Al Qaeda considers the use of nuclear weapons in a spectacular operation as a reason to obtain nuclear weapons*".

The two problems use two different pieces of evidences to assess two different hypotheses. The former uses the piece of evidence "*EVD-Dawn-Mir01-01a"* to judge its

support of the hypothesis "*Al Qaeda considers self defense as a reason to obtain nuclear weapons*". The latter assesses how supportive the piece of evidence "*EVD-Glazov01-01c"* is for the hypothesis *"Al Qaeda considers the use of nuclear weapons in a spectacular operation as a reason to obtain nuclear weapons".* The abstract problem of these two problems can be defined as "*Assess to what extend the piece of evidence favors the hypothesis",* as illustrated in Figure 25. In essence, an abstract problem is the abstraction of all the concrete problems that are solved by using the same abstract problem solving strategy. There is no limit to the number of concrete problems corresponding to a given abstract problem. The abstract problem therefore can reduce a large number of problems in the reasoning tree.

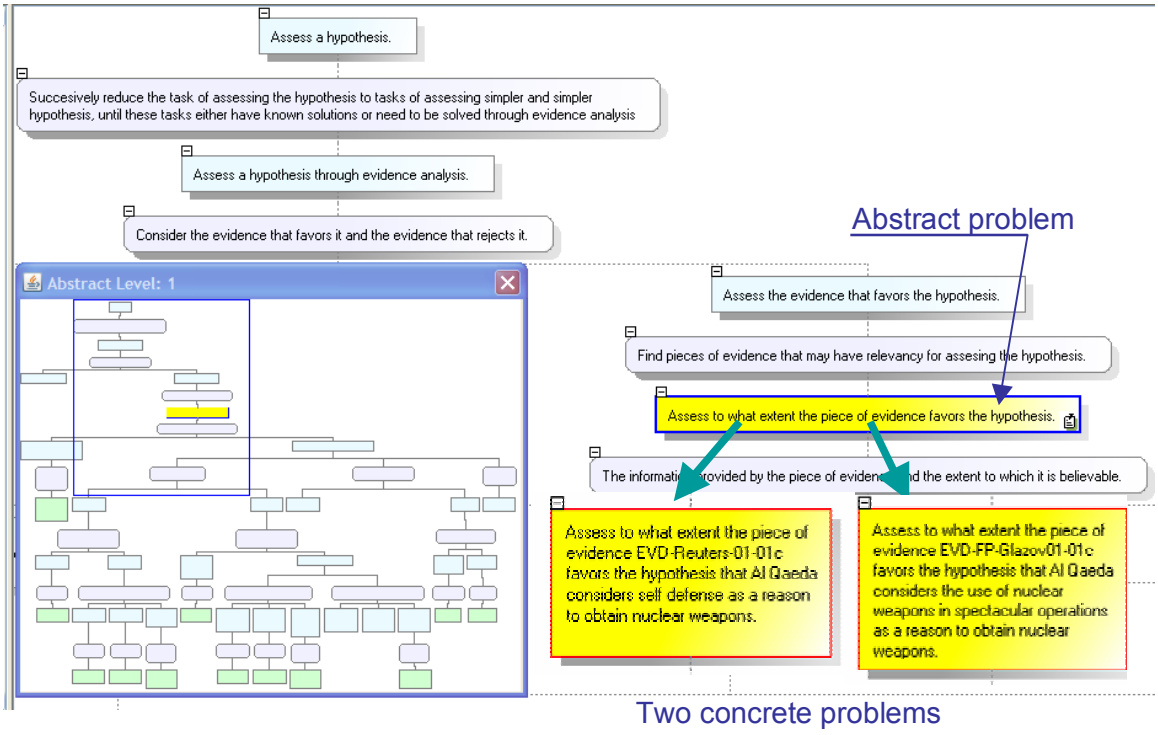


Figure 25: Abstract Problem

3.4.2. Abstract Reduction

The abstract reduction focuses on the problem solving strategies. Each abstract reduction is a reasoning strategy that reduces an abstract problem to its abstract sub-problems. The concrete components of an abstract reduction are the reductions that use the same problem solving strategy.

Let us consider the abstract problem

- "*Assess a hypothesis"* and its abstract sub-problem
  - "*Assess a hypothesis through evidence analysis"*

which correspond to the top problem of a concrete reasoning tree

- "*Assess whether Al Qaeda has nuclear weapons"* and its sub-problems
  - "*Assess whether Al Qaeda considers deterrence as a reason to obtain nuclear weapons."*
  - *"Assess whether Al Qaeda considers self-defense as a reason to obtain nuclear weapons."*
  - *"Assess whether Al Qaeda considers the use of nuclear weapons in spectacular operations as a reason to obtain nuclear weapons."*
  - And so on…

Between the top problem and the sub-problems listed above, there is a sub-tree which successively reduces the first problem to different sub-problems, as indicated in Figure 26.

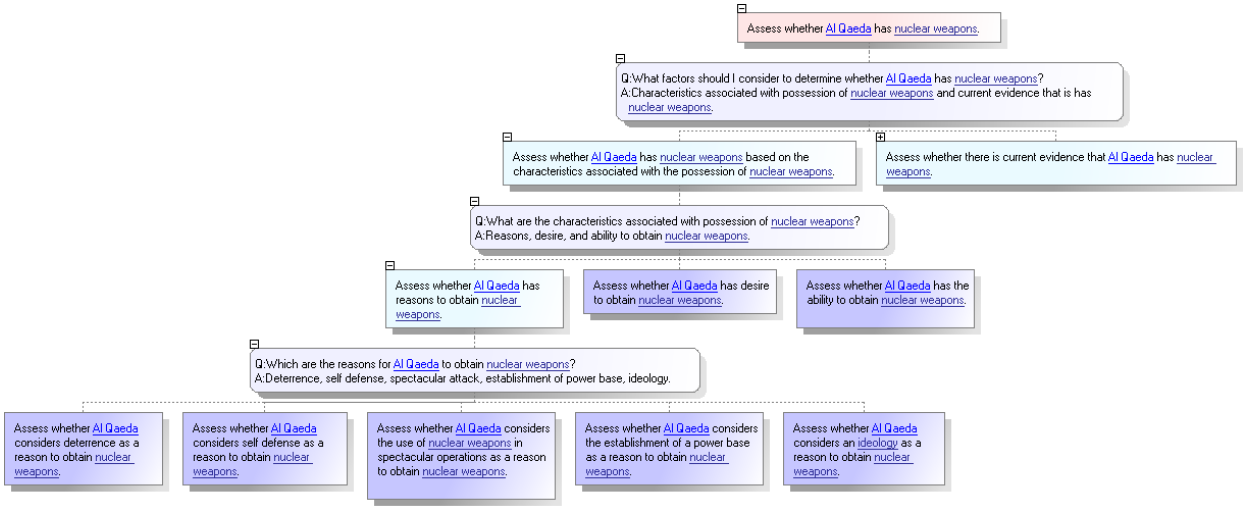


Figure 26: Top Level of a Concrete Reasoning Tree

In other words, there is a sub-tree that plays the role of a reduction strategy that makes it possible for the first problem to be reduced to simpler sub-problems. That sub-tree in the concrete reasoning tree can be abstracted to an abstract reduction "*Reduce the hypothesis to simpler hypothesis."*

A reduction is always associated with a problem and its direct or indirect sub-problems, because it indicates how a problem is reduced to several sub-problems. An abstract reduction therefore can abstract a large sub-tree of a concrete reasoning tree whose root is the problem and leaves are sub-problems that are mentioned above. For instance, the yellow abstract reduction in the right panel of Figure 27, which states "*Reduce the hypothesis to simpler hypotheses"* abstracts several yellow sub-trees in the left panel.

The abstract reasoning tree represents an abstract way to solve a problem in the problem reduction paradigm.

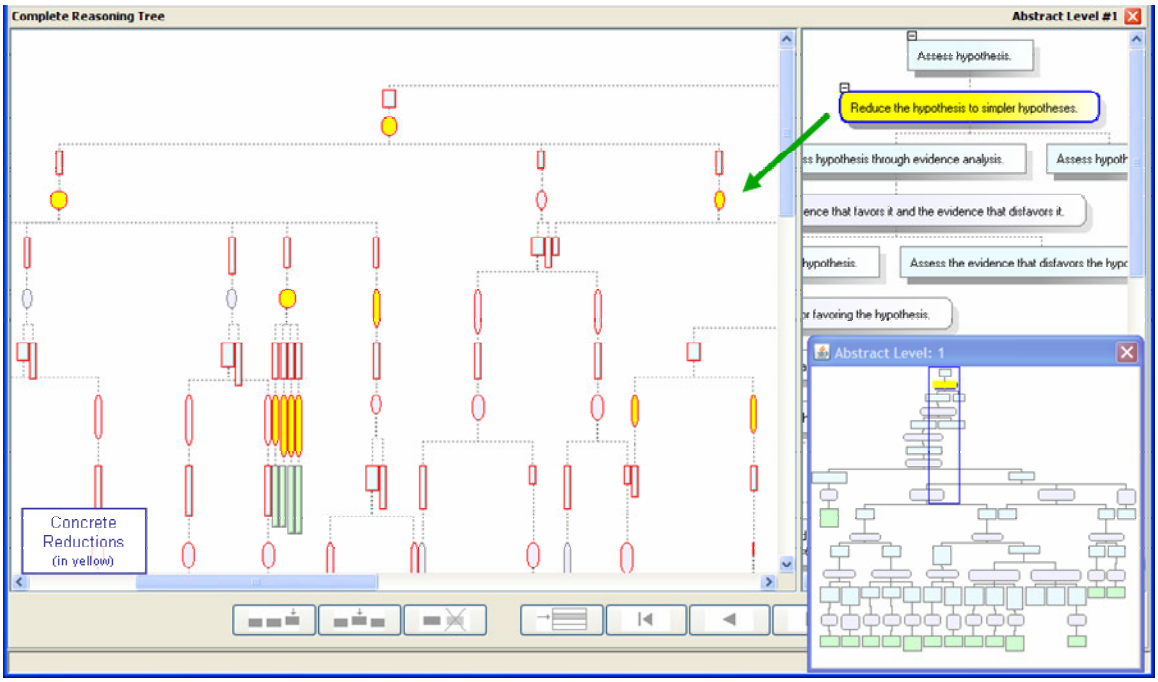


Figure 27: Abstract Reduction and Its Concretions

3.4.3. Abstract Solution

In the problem reduction paradigm, a problem is reduced to simpler sub-problems until the sub-problems have known solutions. Then the synthesis process starts to combine the solutions of the sub-problems to get the synthesized solution of the initial problem. Due to the synthesis process, each problem in the reasoning tree has an associated solution, either a direct solution or a synthesized one. Figure 20 shows a synthesis process whose color is cyan which starts from the solutions at the bottom such as "*The objectivity of Osama Bin Laden with respect to the information provided in Dawn-Mir01-02c is almost certain"* and climbs up the tree to synthesize the solution of

the original problem "*The credibility of Osama Bin Laden with respect to the information provided in Dawn-Mir01-02c is an even chance.*"

The abstract solution is the abstraction of the solutions of all the problems that are solved using the same reasoning strategy. The abstract solution is associated with an abstract problem. Thus the abstract solution of an abstract problem is in fact the abstraction of all the solutions of the concrete problems of that abstract problem. Figure 28 shows several abstract problems and their abstract solutions (represented as cyan sticky notes attached to the abstract problems). For example, the abstract problem:

- *Assess to what extent the piece of evidence favors the hypothesis.*

Has the following abstract solution:

- *Assessed support of hypothesis from the piece of evidence.*

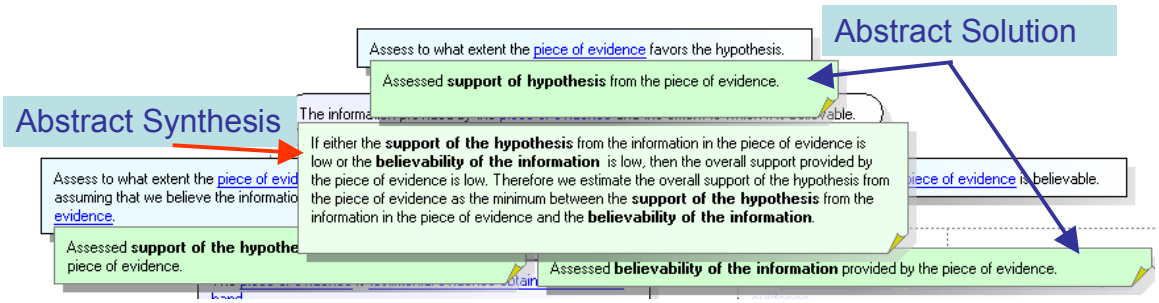


Figure 28: Abstract Solutions and Abstract Synthesis

3.4.4. Abstract Synthesis

In the problem reduction process, the abstract reductions are the bridges connecting abstract problems to their abstract sub-problems. Similarly, in the solution synthesis process, the abstract syntheses connect the abstract sub-solutions to their abstract synthesized solution. An abstract synthesis abstracts the concrete syntheses from the

concrete reasoning tree. While an abstract solution is associated with an abstract problem, the abstract synthesis is associated with an abstract reduction. Thus the abstract solutions and abstract syntheses depend on the corresponding abstract problems and abstract reductions.

The abstract syntheses provide the guidance of how to synthesize the abstract solutions. There are multiple ways to synthesize the abstract solutions. It is up to the subject matter expert who teaches the agent to specify what strategy to be applied. Figure 28 illustrates a way to synthesize an abstract solution from abstract sub-solutions.

3.4.5. Abstract Reasoning Tree

The abstraction of a concrete reasoning tree is essentially the abstractions of its problem nodes, reduction nodes, solution nodes and synthesis nodes. The abstractions of these reasoning tree components are abstract problem nodes, abstract reduction nodes, abstract solution nodes and abstract synthesis nodes respectively. The abstract tree shows the problem solving strategies that are repeatedly used in the concrete reasoning tree. These strategies are the contents of the abstract reduction nodes.

Each abstraction corresponds to one or several concretions. These concretions are the components of the concrete reasoning tree. The many-to-one relationship from the concrete reasoning tree components to their abstract tree components makes the resulting abstract reasoning tree much smaller. The abstract reasoning tree is a semantic representation of the different types of reasoning strategies used in the concrete reasoning tree. Figure 24 shows a concrete reasoning tree and the corresponding abstract tree. The

simplicity of the abstract tree in terms of the number of nodes can be seen in the comparison between the numbers of nodes of the two trees.

Qualitatively, the abstract reasoning tree is a hierarchical organization of the problem solving strategies to solve the problems. For instance, an abstract problem "*Assess to what extent the piece of evidence favors the hypothesis*" is solved by reducing it to simpler abstract problems by using the reasoning:

- *Consider the relevance and the believability of the piece of evidence.*

That strategy leads to simpler abstract problems:

- *Assess to what extent the piece of evidence favors the hypothesis, assuming that the piece of evidence is believable.*
- *Assess the believability of the piece of evidence.*

Each abstract reduction in the abstract tree provides a guideline for how to solve a problem. In other words, the whole abstract tree is a large recipe of problem solving strategies. As in the concrete reasoning tree, each abstract reduction step is associated with an abstract synthesis step. An abstract synthesis step contains several abstract sub-solutions, an abstract synthesis and a synthesized abstract solution. The abstract synthesis indicates how to compose abstract sub-solutions into an abstract solution. To illustrate an abstract synthesis, let us consider the solutions of the two simpler abstract problems above. They are

- *Assessed* ***support of the hypothesis*** *from the information in the piece of evidence.*
- *Assessed* ***believability of the information*** *provided by the piece of evidence.*

The abstract synthesis is

- *If either the **support of the hypothesis** from the information in the piece of evidence is low or the **believability of the information** is low, then the overall support provided by the piece of evidence is low. Therefore we estimate the overall support of the hypothesis from the piece of evidence as the minimum between the **support of the hypothesis** from the information in the piece of evidence and the **believability of the information.***

And that allows us to obtain the assessed **support of hypothesis** from the piece of evidence.

Our proposed abstraction process of the reduction tree begins by grouping the similar problem nodes in a concrete reasoning tree into an abstract problem node. Consider the sub-tree in Figure 19 and the sub-tree in Figure 29. They have the same structures and similar problem nodes, reduction nodes and solution nodes. Both sub-trees can be abstracted into the abstract reduction reasoning tree as shown in Figure 30.

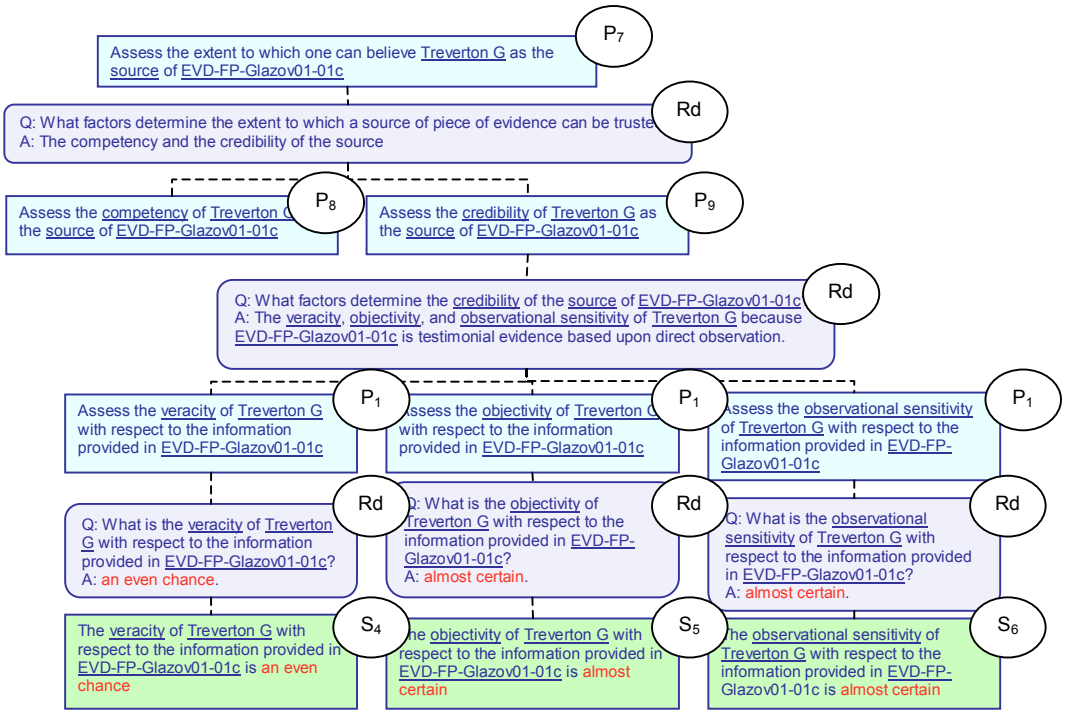


Figure 29: Reduction Sub-tree

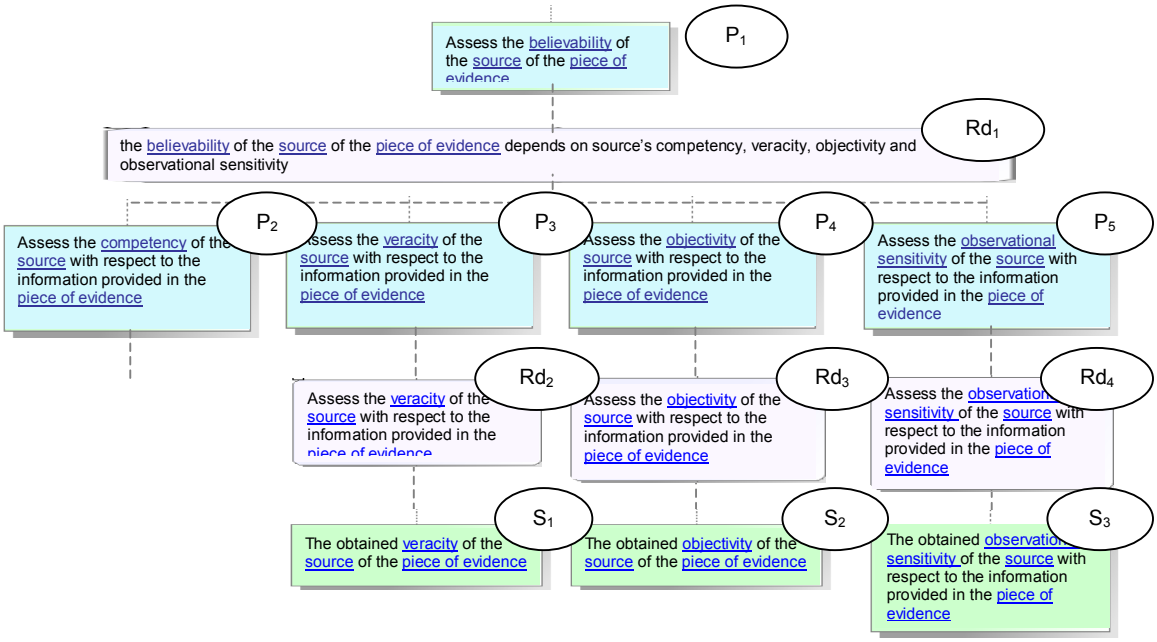


Figure 30: Abstract Reduction Sub-tree

Figure 31 displays the abstraction process of the two sub-trees. We will discuss first how the problem nodes are abstracted. The problem node $P_1^{(0)}$ "*Assess the extent to which one can believe Osama Bin Laden as the source of EVD-Dawn-Mir01-02c"* and $P_2^{(0)}$ "*Assess the extent to which one can believe Treverton G as the source of EVD-FP-Glazov01-01c"* of the sub-trees at ground level (superscripted as *(0)*) are abstracted into abstract problem node $P_1^{(1)}$ "*Assess the believability of the source of the piece of evidence"* of the sub-tree at abstract level (superscripted as *(1)*).

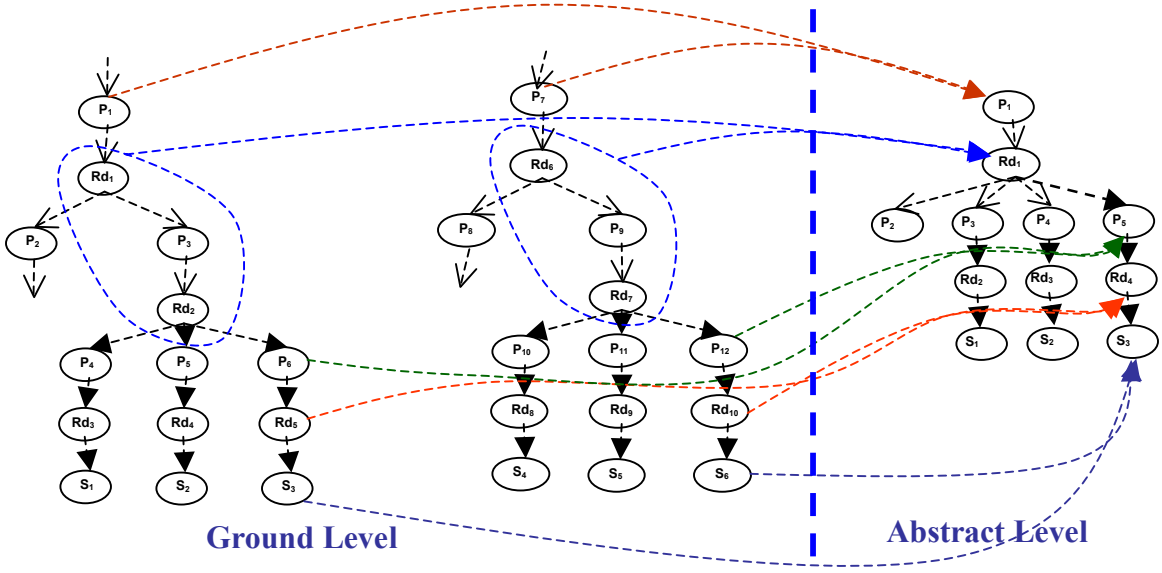


Figure 31: Abstraction of Reduction Trees for Tutoring

Similarly, the abstraction of the solution nodes at one abstract level to abstract solution nodes at the next higher level is called solution abstraction. Figure 30 shows the abstraction of solution node "*The observational sensitivity of Osama bin Laden with respect to the information provided in EVD-Dawn-Mir01-02c is almost certain*" and the

solution node "*The observational sensitivity of Treverton G with respect to the information provided in EVD-FP-Glazov01-01c is almost certain*" to abstract solution node "*The obtained observational sensitivity of the source with respect to the information provided in the piece of evidence*."

The abstraction of the reductions is more complex. In this case, the sub-trees from reasoning tree at ground level are abstracted into an abstract reduction node at the abstract level. As seen in Figure 31, the two sub-trees rooted in reduction node $Rd_1^{(0)}$ and $Rd_6^{(0)}$ and bordered by broken blue lines are abstracted into the abstract reduction node $Rd_1^{(1)}$. The reduction abstraction involves the abstraction of sub-trees.

To be able to define an abstraction for tutoring, we need to define the partition of reduction tree for tutoring purpose.

Definition 3.15 (Partition of Reduction Tree for Tutoring): A partition of a reduction tree $t = (\mathcal{V}_t, \delta_t)$ $Partition_t = P_{st} \cup Rd_{st} \cup S_{st}$ where P_{st} is a set of problem sub-trees $P_{st} = \{st_i = (\{P \in \mathcal{P}_t\}, \varnothing), i=1,n\}$, Rd_{st} is a set of reduction sub-trees $Rd_{st} = \{st_j \mid Root(st) \in \mathcal{Rd}_t, j=1,m\}$, and S_{st} is a set of solution sub-trees $S_{st} = \{st_k = (\{S \in \mathcal{S}_t\}, \varnothing), k=1,l\}$. It has the following properties:

[a] There is only one problem sub-tree that contains the root of the ground tree. $\exists!$ st $\in P_{st}$ such that $Root(t) \in V_{st}$.

[b] Each problem sub-tree except the root sub-tree has as parent a reduction sub-tree. $\forall st \in P_{st}, st \not\subset Root(Partition_t), Parent(st) \in Rd_{st}$.

[c] Each reduction sub-tree has as parent a problem sub-tree. $\forall st \in Rd_{st},\ Parent(st) \in P_{st}$.

[d] Each solution sub-tree has as parent a reduction sub-tree. $\forall st \in S_{st}, Parent(st) \in Rd_{st}$.■

From Definition 3.15, a definition of the abstraction for tutoring is formed based on the abstraction of different types of sub-trees as its basic components. The definition is an extension of the common abstraction function defined in Definition 3.12. This abstraction governs how a tree is abstracted for tutoring purposes.

Definition 3.16 (Abstraction for Tutoring): The abstraction of a reasoning tree at ground level $t = (\mathcal{V}_t, \delta_t)$ (*ground tree*) to the reasoning tree at abstract level $t_a = (\mathcal{V}_{ta}, \delta_{ta})$ (*abstract tree*) is named *abstraction for tutoring*, if there is a partition for tutoring $Partition_t = P_{st} \cup Rd_{st} \cup S_{st}$ and an abstraction function α_t: $Partition_t \rightarrow Vt_a \cup \{\lambda\}$ such that $\alpha_t(st) = \begin{cases} \alpha_P(st), \forall st \in P_{st} \\ \alpha_{Rd}(st), \forall st \in Rd_{st} \\ \alpha_S(st), \forall st \in S_{st} \end{cases}$ where

- α_P: $P_{st} \rightarrow \mathcal{P}_{ta}$ is a surjective function, i.e, $\forall P_{ta} \in \mathcal{P}_{ta}, \exists st \in P_{st}$ such that $\alpha_P(st) = P_{ta}$. α_P is called *problem node abstraction function*.
- α_{Rd}: $Rd_{st} \rightarrow \mathcal{Rd}_{ta}$ is a surjective function, i.e, $\forall Rd_{ta} \in \mathcal{Rd}_{ta}, \exists st \in Rd_{st}$ such that $\alpha_{Rd}(st) = Rd_{ta}$. α_{Rd} is called *reduction node abstraction function.*
- α_S: $S_{st} \rightarrow \mathcal{S}_{ta}$ is a surjective function, i.e, $\forall S_{ta} \in \mathcal{S}_{ta}, \exists st \in S_{st}$ such that $\alpha_S(st) = S_{ta}$. α_S is called *solution node abstraction function.*■

Remark:

[a] A problem sub-tree cannot have more than one abstraction. $\forall p_1, p_2 \in \mathcal{P}_{ta}, p_1 \neq p_2,$

$\alpha_P^{-1}(p_1) \cap \alpha_P^{-1}(p_2) = \varnothing.$

[b] A reduction sub-tree cannot have more than one abstraction. $\forall rd_1, rd_2 \in \mathcal{Rd}_{ta}, rd_1$

$\neq rd_2, \alpha_{Rd}^{-1}(rd_1) \cap \alpha_{Rd}^{-1}(rd_2) = \varnothing.$

[c] A solution sub-tree cannot have more than one abstraction. $\forall s_1, s_2 \in \mathcal{S}_{ta}, s_1 \neq s_2,$

$\alpha_S^{-1}(s_1) \cap \alpha_S^{-1}(s_2) = \varnothing.$

The problem node abstraction functions corresponding to the abstractions in Figure 31 are:

- $\{st^{(0)}_{t[P1|\{P1\}]}, st^{(0)}_{t[P7|\{P7\}]}\} \xrightarrow{\alpha_P} P_1^{(1)}$
- $\{st^{(0)}_{t[P2|\{P2\}]}, st^{(0)}_{t[P8|\{P8\}]}\} \xrightarrow{\alpha_P} P_2^{(1)}$
- $\{st^{(0)}_{t[P4|\{P4\}]}, st^{(0)}_{t[P10|\{P10\}]}\} \xrightarrow{\alpha_P} P_3^{(1)}$
- $\{st^{(0)}_{t[P5|\{P5\}]}, st^{(0)}_{t[P11|\{P11\}]}\} \xrightarrow{\alpha_P} P_4^{(1)}$
- $\{st^{(0)}_{t[P6|\{P6\}]}, st^{(0)}_{t[P12|\{P12\}]}\} \xrightarrow{\alpha_P} P_5^{(1)}$

The reduction node abstraction functions corresponding to the abstractions in Figure 31 are:

- $\{st^{(0)}_{t[Rd1|\{Rd2\}]}, st^{(0)}_{t[Rd6|\{Rd7\}]}\} \xrightarrow{\alpha_{Rd}} Rd_1^{(1)}$
- $\{st^{(0)}_{t[Rd3|\{Rd3\}]}, st^{(0)}_{t[Rd8|\{Rd8\}]}\} \xrightarrow{\alpha_{Rd}} Rd_2^{(1)}$
- $\{st^{(0)}_{t[Rd4|\{Rd4\}]}, st^{(0)}_{t[Rd9|\{Rd9\}]}\} \xrightarrow{\alpha_{Rd}} Rd_3^{(1)}$
- $\{st^{(0)}_{t[Rd5|\{Rd5\}]}, st^{(0)}_{t[Rd10|\{Rd10\}]}\} \xrightarrow{\alpha_{Rd}} Rd_4^{(1)}$

The solution node abstraction functions corresponding to the abstractions in Figure 31 are:

- $\{st^{(0)}_{t[S1|\{S1\}]}, st^{(0)}_{t[S4|\{S4\}]}\} \xrightarrow{\alpha_S} S_1^{(1)}$
- $\{st^{(0)}_{t[S2|\{S2\}]}, st^{(0)}_{t[S5|\{S5\}]}\} \xrightarrow{\alpha_S} S_2^{(1)}$
- $\{st^{(0)}_{t[S3|\{S3\}]}, st^{(0)}_{t[S6|\{S6\}]}\} \xrightarrow{\alpha_S} S_3^{(1)}$

Lemma 3.2 (Lower Bound of Abstraction for Tutoring): The lower bound of the abstraction of a reduction tree is the reduction tree itself.

Proof: An abstraction $\alpha_t(st)$: $Partition_t \rightarrow V_{ta}$ reduces a sub-tree of a reduction tree into a node of an abstract reduction tree. If $Partition_t$ is a singleton partition then the abstraction $\alpha_t(st)$ does not reduce the number of nodes at all. Furthermore, according to Definition 3.12 b, the parent-child relationships and sibling relations of the reduction tree are preserved. Therefore for a singleton partition $Partition_t$, the abstract reduction tree is as same as the reduction tree.■

Lemma 3.3 (Upper Bound of Abstraction for Tutoring): The upper bound of abstraction of a non-singleton reduction tree is an abstract reduction tree which has two nodes: an abstract problem node as root and an abstract reduction node as leaf.

Proof: Because $Partition_t$ is, in general, a set of sub-trees (not always singleton sub-tree), the abstraction tends to makes the number of nodes of the abstract reduction tree smaller than that of the concrete reduction tree. The smallest number of nodes that a tree can have is one. Assume that an upper bound of the abstraction is a single node tree. That implies the domain $Partition_t$ of the abstraction α_t is a partition that has only one sub-tree

which is the reduction tree itself and the co-domain $\mathcal{P}_{ta}$ has only one abstract problem node because it is the abstraction of the root of the reduction tree. Based on the Definition 3.15 and Definition 3.16, $\alpha_t = \{\alpha_P: P_{st} \rightarrow \mathcal{P}_{ta}\}$ and $P_{st} = \{st_i = (\{P \in \mathcal{P}_t\}, \varnothing), i=1,n\}$ and reduction tree contains the reduction nodes $\{Rd \in \mathcal{P}_t\}$ as well which contradicts the assumption.

Let us assume we have an abstraction function $\alpha_t(st)$: $Partition_t \rightarrow V_{ta}$ defined as follows:

$\alpha_P(\{st^{(0)}{}_{t[root|\{root\}]}\}) = Pr^{(1)}$ where $Pr^{(1)}$ is root of the abstract reduction tree.

$\alpha_{Rd}(\{st^{(0)}{}_{it[Rd1|\{Vtj\}]} \mid i=1,n, j=1,m\}) = Rd^{(1)}$ where $\{st^{(0)}{}_{it[Rd1|\{Vtj\}]} \mid i=1,n, j=1,m\}$ is the rest of the reduction tree from root and $Rd^{(1)}$ is the only abstract reduction node of the abstract tree. Such abstraction yields an abstract reduction tree that has only two abstract nodes: one is the abstract problem node as root and the other is the abstract reduction node. This abstract reduction tree satisfies the properties of Definition 3.15 and Definition 3.16. Therefore the upper bound of an abstraction for tutoring of a non-singleton reduction tree is a two node abstract tree.■

Remark: Any reduction tree can have different levels of abstractions for tutoring. Their complexity will be in between the upper bound and the lower bound of the abstraction. In other words, there are different ways to tutor the domain knowledge based on an abstract reasoning tree of that domain.

3.4.6 Abstraction Mapping

Once the abstraction of a reasoning tree is built within the stated constraints, the abstraction operations are learned by generalization to become abstraction mapping. The generalization of the abstraction operations is a two-step process. The first step is the generalization of the concrete components and of the abstract component of an abstraction function (top two blocks of Figure 32). The second step is the construction of the abstraction mapping for the abstraction function based on the generalizations (bottom two blocks of Figure 32).

The generalization of the concrete components of an abstraction function is a complex process. In essence, generalization is a process that transforms an expression into a more general expression. It may be done by applying generalization rules, such as replacing a constant with a variable, a concept with more general one, a number with an interval, and so on (Tecuci, 1998). A problem node is generalized to a problem class, a reduction node is generalized to a reduction rule and a solution node is generalized to a solution class. For example, in Figure 19, the problem node $P_1^{(0)}$

- *"Assess the extent to which one can believe Osama Bin Laden as the source of EVD-Dawn-Mir01-02c"*

is generalized to the problem class

- "*Assess the extent to which one can believe ?O1 as the source of ?O2"*

by replacing the constant *Osama Bin Laden* with the variable ?O1 and the constant *EVD-Dawn-Mir01-02c* with the variable ?O2.

The solution node $S_3^{(0)}$

- "*The observational sensitivity of Osama bin Laden with respect to the information provided in EVD-Dawn-Mir01-02c is almost certain"*

is generalized to the solution class

- "*The observational sensitivity of ?O1 with respect to the information provided in ?O2 is ?SI1."*

The reduction node $Rd_1^{(0)}$

- *"Q: What factors determine the extent to which a source of piece of evidence can be trusted? A: The competency and the credibility of the source"*

is generalized to the reduction rule

- *"If assess the extent to which one can believe ?O1 as the source of ?O2 then assess the competence of ?O1 as the source of ?O2 and assess the credibility of ?O1 as the source of ?O2."*

Similarly, the generalization of an abstract problem node is an abstract problem class, the generalization of an abstract reduction node is an abstract reduction class, and the generalization of an abstract solution node is an abstract solution class. For example, the abstract problem node $P_1^{(1)}$ is generalized to the abstract problem class "*Assess the believability of the source of the piece of evidence."*

The second step of the generalization of the abstraction operations consists in creating an abstraction mapping that links the concrete classes of the concrete components to the abstract classes of the abstract components. For example, one instance of the abstraction mapping can be stated as follows: "If a concrete class is *Assess the extent to which one can believe ?O1 as the source of ?O2* where *?O1* is the source and *?O2* is the piece of

evidence then that class can be mapped to an abstract class *Assess the believability of the source of the piece of evidence."*

As recalled from the Definition 3.16, the domain of the abstraction function for tutoring is the partitions of the reasoning tree. For the problem node abstraction function and the solution node abstraction function, the domain is the single partitions as defined in Definition 3.9. The single partitions contain the single sub-trees, so there is no concept of root node and non-root node in single sub-trees. However, the domains of reduction node abstraction functions are not singleton partitions. As a matter of fact, the role of root nodes in the sub-trees is important in the abstraction. It guides how a reduction occurs. Therefore, when we build the abstraction mapping, the root nodes of the reduction sub-trees have to be taken into account.

Figure 32 presents the relation between the reduction tree and its abstract reduction tree. In this figure, the concrete reduction tree is abstracted to the abstract reduction tree. The reduction tree is the instantiation of the problem classes, reduction rules, and solution classes from the knowledge base. The abstract reduction tree generates from abstract problem classes, abstract reduction classes and abstract solution classes. The abstract classes are, in turn, generated from the problem classes, solution classes and reduction rules of the knowledge base of the existing reduction tree via the abstraction mapping. The abstraction mapping is saved and restored to generate an abstract reduction tree given a concrete reduction tree. The abstraction mapping, in fact, governs how a reduction tree should be abstracted.

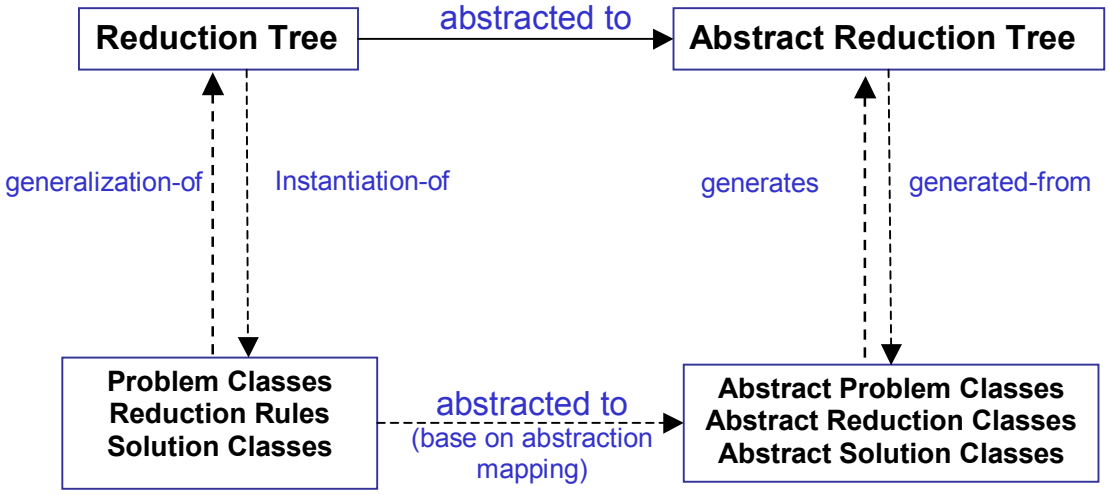


Figure 32: The Relation between Reduction Tree and Its Abstract Reduction Tree

Definition 3.17 (Reduction Abstraction Mapping): Given a set of problem classes $\mathcal{PC}_{\Lambda}$ at ground level, a set of solution classes $\mathcal{SC}_{\Lambda}$ at ground level, and a set of reduction rules $\mathcal{RdR}_{\Lambda}$ at ground level, the abstraction mapping Λ is defined as follows:

$$\Lambda(\sigma)=\begin{cases}\Lambda_{P}(\sigma) & ,\sigma\in\mathcal{PC}_{t}\\ \Lambda_{S}(\sigma) & ,\sigma\in\mathcal{SC}_{t}\\ \Lambda_{Rd}(\sigma) & ,\sigma\in\mathcal{RdR}_{t}\\ \Lambda_{rRd}(\sigma) & ,\sigma\in\mathcal{RdR}_{t}\end{cases}$$

Where:

- $\Lambda_P(\sigma)$: $\mathcal{PC}_{\Lambda} \rightarrow \mathcal{APC}_{\Lambda}$ where $\mathcal{APC}_{\Lambda}$ is a set of abstract problem classes at abstract level, indicates that for each problem class selected from the ground level there is a corresponding abstract problem class at the abstract level. $\Lambda_p(\sigma)$ is named

problem abstraction mapping which is surjective (similar to the problem node abstraction function α_P).

- $\Lambda_S(\sigma)$: $\mathcal{SC}_\Lambda \rightarrow \mathcal{ASC}_\Lambda$ where $\mathcal{ASC}_\Lambda$ is a set of abstract solution classes at abstract level, indicates that for each solution class selected from the ground level there is a corresponding abstract solution class at the abstract level. $\Lambda_S(\sigma)$ is named *solution abstraction mapping* which is surjective (similar to the solution node abstraction function α_S).
- $\Lambda_{Rd}(\sigma)$: $\mathcal{RdR}_\Lambda \rightarrow \mathcal{ARdC}_\Lambda$ where $\mathcal{ARdC}_\Lambda$ is a set of abstract reduction classes at abstract level, indicates that for each reduction rule that is not the root of its sub-tree selected from the ground level there is at least a corresponding abstract reduction class at the abstract level. $\Lambda_{Rd}(\sigma)$ is named *reduction abstraction mapping* which is surjective (similar to the reduction node abstraction function α_{Rd}).
- $\Lambda_{rRd}(\sigma)$: $\mathcal{rRdR}_\Lambda \rightarrow \mathcal{ARdC}_\Lambda$ where $\mathcal{rRdR}_\Lambda \subset \mathcal{RdR}_\Lambda$, $\mathcal{rRdR}_\Lambda$ is a set of root reduction class and $\mathcal{ARdC}_\Lambda$ is a set of abstract reduction classes at abstract level, indicates that for each reduction rule that is root of its sub-tree selected from the ground level there is a corresponding abstract reduction class at the abstract level. $\Lambda_{rRd}(\sigma)$ is named *root reduction abstraction rule* which is surjective (similar to the reduction node abstraction function α_{Rd}).

The abstraction mapping must satisfy the following properties:

[a] A problem class cannot have more than one abstraction. $\forall PC \in \mathcal{PC}_{\Lambda}$, $\exists!\ APC \in \mathcal{APC}_{\Lambda}$ such that $\Lambda_P(PC) = APC$. That makes the problem abstraction mapping a function.

[b] A reduction rule that is root of its sub-tree cannot have more than one abstraction. $\forall rRdR \in \mathcal{rRdR}_{\Lambda}$, $\exists!\ ARdC \in \mathcal{ARdC}_{\Lambda}$ such that $\Lambda_{rRd}(rRdR) = ARdC$. That makes the root reduction abstraction mapping a function.

[c] A solution class cannot have more than one abstraction. $\forall SC \in \mathcal{SC}_{\Lambda}$, $\exists!\ ASC \in \mathcal{ASC}_{\Lambda}$ such that $\Lambda_S(SC) = ASC$. That makes the solution abstraction map a function.■

Theorem 3.4 (Existence and Uniqueness of Abstract Reduction Tree): Given a set of problem classes $\mathcal{PC}$, a set of solution classes $\mathcal{SC}$, a set of reduction rules $\mathcal{RdR}$ and an abstraction mapping Λ defined for the previous classes, there is a construction method such that for each reduction tree generated at the ground level there is one and only one corresponding abstract reduction tree constructed based on the abstraction mapping.

Proof: The proof contains two parts. First, we need to prove that given an abstraction mapping, for each reduction tree at ground level we can construct an abstract reduction tree. Second, we prove that the newly constructed abstraction tree is unique, given the reduction tree and the abstraction mapping.

Part 1: Existence.

Given the classes at the ground level and an abstraction mapping $\Lambda(\sigma)$, we need to show that we can develop a construction method that will generate a unique abstract tree

for any reasoning tree at the ground level. Let us consider a reasoning tree t generated at the ground level. The tree t was generated based on the problem classes $PC \in \mathcal{PC}$, solution classes $SC \in \mathcal{SC}$, and reduction rules $RdR \in \mathcal{RdR}$.

First we will partition the reduction tree t into sub-trees. The sub-trees are built based on their root nodes.

We define the set of abstracted root nodes as being all the nodes in the reasoning tree t that are generated based on problem classes from $\mathcal{PC}_\Lambda$, solution classes from $\mathcal{SC}_\Lambda$, or root reduction rules from $\mathcal{RdR}_\Lambda$: $RootNodes = \{P \in \mathcal{P}_t | P$ is instantiation of $PC \in \mathcal{PC}_\Lambda\} \cup \{S \in \mathcal{S}_t | S$ is instantiation of $SC \in \mathcal{SC}_\Lambda\} \cup \{rRd_i \in \mathcal{Rd}_t | rRd$ is instantiation of $rRdR \in \mathcal{rRdR}_\Lambda\}$.

From the set of root nodes, we build the sub-trees:

- singleton sub-trees for the problem nodes that are in the *RootNodes* set.

  $ST_P = \{st_i = (\{P\}, \lambda) \mid P \in RootNodes \cap \mathcal{P}_t\}$

- singleton sub-trees for the solution nodes that are in the *RootNodes* set.

  $ST_S = \{st_j = (\{S\}, \lambda) \mid S \in RootNodes \cap \mathcal{S}_t\}$

- sub-trees for the reduction nodes, with the root node *rRd* in *RootNodes*, and also containing all the reduction nodes Rd_i that are generated based on reduction rules RdR_i that have the same abstract reduction class *ARdC* as that of the reduction rule *rRdR* on which the root node *rRd* is generated, (i.e.,

$\Lambda_{rRd}(rRdR) = ARdC = \Lambda_{Rd}(RdR_i) \forall i$) and there is no other root node in the sub-tree.

ST_{Rd} = {*st* sub-tree *of t* | $V_{st} \cap RootNodes = \{Root(st)\}$, $Root(st) \in RootNodes \cap$ $\mathcal{Rd}_t$, *Root(st)* is instantiation of *rRdR, and* $\Lambda_{rRd}(rRdR) = ARdC = \Lambda_{Rd}(RdR_i)$ for any reduction rule RdR_i for which there is an instantiation node Rd_i in *st*}

One may notice that the problem nodes that are located between the selected reductions are also added to those sub-trees. Having constructed the previous sub-trees, there will remain some concrete reasoning nodes that are not included in the sub-trees. These nodes do not have abstraction.

Next, we construct the abstract tree, by specifying the abstraction function. The construction method is given below.

Given the partitioned reasoning tree we will construct the abstract tree as its abstraction as follows: from the root of the reasoning tree, go top-down and left-right, taking the sub-trees one by one. For the current sub-tree *st*, we consider the root *R* = *Root(st)*. As stated, the nodes of the reasoning tree were generated by instantiating corresponding problem classes, reduction rules or solution classes. Let us consider *RC* as being the corresponding class for the root *R* (if *R* is problem node, *RC* is a problem class; if *R* is solution node, *RC* is a solution class and if *R* is a reduction node, *RC* is a reduction rule). If there is an abstraction mapping defined for *RC* based on properties [a], [b], [c] of Definition 3.17, then for each *RC* there is a unique abstraction mapping. We apply the abstraction mapping to the class and obtain an abstract class *ARC* (an abstract problem

class for a problem class, an abstract solution class for a solution class, and an abstract reduction class for a reduction rule).

At this point, we will either use an existing instantiation of *ARC* already created in the abstract sub-tree or we will create a new instantiation. If there is a left sibling at the current location in the abstract tree of the same *ARC* we will reuse that instantiation (abstract node). If not, then we create the abstract node *AN* as instantiation of the abstract class *ARC*. This abstract node is the abstraction of the sub-tree *st*, i.e., $\alpha_t(st) = AN$ where $AN \in \mathcal{V}_{ta}$. For each abstract node *AN*, link it to another abstract node *AN'* as its parent, where *AN'* is the abstraction of the parent sub-tree of the current sub-tree in the concrete reasoning tree – to preserve the parent-child relationship.

If there is no abstraction mapping defined, this sub-tree will not be abstracted, i.e., $\alpha_t(st) = \lambda$.

The process will continue until all the sub-trees in the partition will be considered. At the end of this process we obtain the abstract tree and the abstraction function α_t.

Part 2: Uniqueness.

Let us assume that there are two abstract trees t_{a1} and t_{a2}, constructed based on the same abstraction mapping applied to one concrete reasoning tree *t*. We will traverse the two abstract trees from the root in the top-down, left-right manner, to find the first different nodes.

Let us assume that the first different nodes are the abstract problem nodes P_{1ta} in the first tree and P_{2ta} in the second tree.

Because they need to be different (except an instantiation isomorphism) the nodes need to be generated by different abstract problem classes PC_{1ta} and PC_{2ta}. Due to the property [a] of Definition 3.17, there must be a problem class PC_{1t} abstracted to PC_{1ta} and a problem class PC_{2t} abstracted to PC_{2ta}. Because of the fact that $PC_{1ta} \neq PC_{2ta}$ and property [a], then $PC_{1t} \neq PC_{2t}$. Based on the previous construction method, and the conservation of parent-child relationship we must have two different sibling nodes P_{1t} and P_{2t} in the reasoning tree instantiating PC_{1t} and PC_{2t}. Let us assume P_{1t} is the left sibling of P_{2t}.

Because of the construction method, we must have in the second abstract tree a left sibling of P_{2ta} corresponding to P_{1t}: P'_{1ta}. Moreover this must be also in the first abstract tree (because we considered the first place where the abstract trees are different). This means that in the first abstract tree we will have P'_{1ta} left sibling of P_{1ta} and both of them are instantiations of the same class PC_{1ta}. This contradicts the construction method which will reuse the P_{1ta} and not create another instantiation.

If the first different nodes are abstract solution nodes or abstract reduction nodes a similar contradiction is obtained.■

3.4.7 Algorithm for Generation of Abstract Reduction Trees

Based on Definition 3.17 and Theorem 3.5, there is a construction method that maps a concrete reduction tree to a unique abstract reduction tree. In the implementation of the construction method, it would be efficient to treat the abstraction mapping as a set of abstraction rules, each of which is a pair of a concrete class and its corresponding abstract class. In other words, there are three types of abstraction rules, listed as follows:

- Problem abstraction rule: (problem class, abstract problem class), storing the mapping $\Lambda_P(PC) = APC$, where $PC \in \mathcal{PC}_\Lambda$ and $APC \in \mathcal{APC}_\Lambda$. The set of all problem abstraction rules are a representation of Λ_P.
- Reduction abstraction rule: (reduction rule, abstract reduction class), storing the mapping $\Lambda_{Rd}(RdR) = ARdC$, where $RdR \in \mathcal{RdR}_\Lambda$ and $ARdC \in \mathcal{ARdC}_\Lambda$. The set of all reduction abstraction rules are a representation of Λ_{Rd}.
- Solution abstraction rule: (solution class, abstract solution class), storing the mapping $\Lambda_S(SC) = ASC$, where $SC \in \mathcal{SC}_\Lambda$ and $ASC \in \mathcal{ASC}_\Lambda$. The set of all solution abstraction rules are a representation of Λ_S.

In general, an abstraction rule (concrete class CC, abstract class AC) will associate a concrete class with its corresponding abstract class. With the abstraction rules, the task of abstracting a reduction reasoning tree can be done automatically. Table 2 presents the algorithm of associating the abstraction rules to the reduction nodes of the concrete reduction tree. Table 3 presents the retrieval of the associated abstraction rule from a reduction node of a concrete reduction tree. Table 4 presents the algorithm of generation of the abstract reduction tree given the concrete reduction tree and a set of abstraction rules.

Table 2: Associate Abstraction Rule

Given: • AbstRules - *the set of all the abstraction rules* **Return:**

- none

AssociateAbstractionRule(AbstRules)
1. **for** each AbstRule = (CC, AC) from AbstRules **do**
2. associate AbstRule to CC
3. **end for**

end AssociateAbstractionRule

The algorithm does not return any but creates a link between a concrete class and the abstraction rule, if any. For a given reasoning tree, each node has an associated concrete class (e.g. problem class, reduction rule). Therefore, each node will be indirectly associated with an abstraction rule, if any.

Table 3: Get Abstraction Rule

Given:
- Node - *a node to search for its abstraction rule*

Return:
- NodeAbstRule - *abstraction rule associated with given node Node – NULL if none*

GetAbstractionRule(Node**)**
1. **if** Node is problem node **then**
2. ProblemClass ← retrieve the problem class of Node
3. **return** the associated abstraction rule of the ProblemClass or NULL if none
4. **else if** Node is solution node **then**
5. SolutionClass ← retrieve the solution class of Node
6. **return** the associated abstraction rule of the SolutionClass or NULL if none
7. **else if** Node is reduction node **then**
8. ReductionRule ← retrieve the reduction rule of Node
9. **return** the associated abstraction rule of the ReductionRule or NULL if none
10. **end if**

end GetAbstractionRule

The algorithm in Table 3 retrieves the abstraction rule associated with a node in the reasoning tree, if any. The node can be problem node, reduction node or solution node. For any node, it retrieved the class which generated that node and the abstraction rule associated with it, if any.

Table 4: Generation of Abstract Reduction Tree

Given:

- t - *concrete reasoning tree*
- AbstRules - *a set of abstraction rules*

Return:

- ta - *the abstract reduction tree*

```
GenerateAbstractReductionTree(t, AbstRules)
1.  AssociateAbstractionRules(AbstRules)
2.  Queue ← ∅  nodes in the tree waiting to be abstracted
3.  FoundFlag ← false
4.  add root of tree t to Queue
5.  while Queue is not empty do
6.      Node ← pop a node from queue
7.      NodeAbstRule ← GetAbstractionRule(node)
8.      if Node is problem node then
9.          AbstProblemClass ← retrieve abstract problem class from NodeAbstRule
10.         ParentNode ← get parent of Node
11.         if ParentNode is not null then
12.             FoundFlag = false
13.             AbstParentNode ← get abstract node from ParentNode
14.             if AbstProblemClass is null and Node's children have abstractions then
15.                 add Node to AbstParentNode concrete components
16.                 Set AbstParentNode as abstraction of Node
17.             else
18.                 AbstChildrenNodes ← get children of AbstParentNode
19.                 for each AbstChildNode of AbstChildrenNodes do
20.                     AbstChildNodeClass ← retrieve abstract class from
                            AbstChildNode
21.                     if AbstChildNodeClass = AbstProblemClass then
22.                         add Node to AbstChildNode's concrete components
23.                         set AbstChildNode as abstraction of Node
24.                         FoundFlag = true
```

```
25.                     end if
26.                 end for
27.             end if
28.             if FoundFlag is false then
29.                 APN ← create abstract problem node from AbstProblemClass
30.                 add Node to APN's concrete components
31.                 set APN as abstraction of Node
32.                 link APN to AbstParentNode as its parent
33.             end if
34.         else                    ParentNode is null
35.             APN ← create abstract problem node from AbstProblemClass
36.             add Node to the list of concrete components of APN
37.             set APN as abstraction of Node
38.             add APN to Vta and set APN as Root of ta
39.         end if
40.     else if Node is reduction node then
41.         AbstReductionClass ← retrieve abstract reduction class from NodeAbstRule
42.         ParentNode ← get parent of Node
43.         if ParentNode is not null then
44.             AbstParentNode ← get abstract node from ParentNode
45.             FoundFlag = false
46.             if AbstReductionClass is not null then
47.                 AbstChildrenNodes ← get children of AbstParentNode
48.                 for each AbstChildNode of AbstChildrenNodes do
49.                     AbstChildNodeClass ← retrieve abstract class from
                            AbstChildNode
50.                     if AbstChildNodeClass = AbstReductionClass then
51.                         add Node to AbstChildNode's concrete components
52.                         set AbstChildNode as abstraction of Node
53.                         FoundFlag = true
54.                     end if
55.                 end for
56.             end if
57.             if FoundFlag is false then
58.                 ARN ← create abstract reduction node from AbstReductionClass
59.                 add Node to ARN concrete components
60.                 set ARN as abstraction of Node
61.                 link ARN to AbstParentNode as parent
62.             end if
63.         end if
64.     else if  node is solution node then
65.         AbstSolutionClass ← retrieve abstract solution class from NodeAbstRule
66.         ParentNode ← get parent of Node
67.         FoundFlag = false
```

```
68.          if ParentNode is not null then
69.              AbstParentNode ← get abstract node from ParentNode
70.              if AbstSolutionClass is not null then
71.                  AbstChildrenNodes ← get children of AbstParentNode
72.                  for each AbstChildNode of AbstChildrenNodes do
73.                      AbstChildNodeClass ← retrieve abstract class from
                             AbstChildNode
74.                      if AbstChildNodeClass = AbstSolutionClass then
75.                          add Node to AbstChildNode's concrete components
76.                          set AbstChildNode as abstraction of Node
77.                          FoundFlag = true
78.                      end if
79.                  end for
80.              end if
81.              if FoundFlag is false then
82.                  ASN ← create abstract solution node from AbstSolutionClass
83.                  add Node to ASN's concrete components
84.                  set ASN as abstraction of Node
85.                  link ASN to AbstParentNode as parent
86.              end if
87.          end if
88.      end if
89.      if Node has abstraction or at least one of Node's children has abstraction then
90.          Children  ← get children of Node
91.          add Children to Queue
92.      end if
93. end while
94. return ta
end GenerateAbstractReductionTree
```

Table 4 provides the algorithm to generate the abstract reduction reasoning tree given a concrete reduction reasoning tree and a set of abstraction rules. The algorithm starts by associating the abstraction rules to the reasoning nodes of the concrete reasoning tree (line 1). Then it uses breadth-first search to enumerate all the nodes in the concrete reasoning tree. Line 5 starts the breadth-first search. For each node of the concrete reasoning tree, the associated abstraction rule NodeAbstRule is retrieved at line 7. The algorithm distinguishes three different types of nodes: problem node, reduction node and

solution node. For the problem node, the abstract problem class AbstProblemClass is retrieved given its abstraction rule NodeAbstRule (line 9). The problem node Node is tested if it is a root or not, based on its reduction parent node ParentNode (lines 10 and 11). A root does not have a parent and is taken care of at line 34. Line 11 presents the case where the problem node Node is not the root. Line 13 presents the case where the AbstProblemClass is null but some of its children have abstractions, which means the Node does not have its own abstraction but implicitly abstracted to its parent's abstraction (abstract reduction node). In this case it is added to the concrete component list of its abstraction of its parent (lines 14-16). Lines 18 and 19 retrieve the children AbstChildrenNodes of the abstraction of the parent of the problem node AbstParentNode. Each child AbstChildNode is supposedly an abstract problem node or an abstract solution node. Lines 19 to 25 are the FOR loops to enumerate all children. Their abstract problem classes or solution classes AbstChildNodeClass are, in turn, compared against the abstract problem class AbstProblemClass of the problem node Node. If AbstChildNodeClass retrieved from the child problem node AbstChildNode is the same as the abstract problem class AbstProblemClass (line 21), then the problem node Node is added as one of the concrete components of the child abstract problem node AbstChildNode (line 22). The flag FoundFlag is set to true (line 24) to indicate that an abstract problem node AbstChildNode in the abstract reasoning tree was found. If no appropriate abstract problem node was found (line 28) then a new abstract problem node APN is created from the abstract problem class AbstProblemClass (line 29). The problem node Node is added to the concrete component list of APN (line 30). APN is linked to its

parent AbstParentNode (line 32). If the problem node is the root of the concrete reasoning tree (line 34), the abstract problem node APN which is created based on the abstract problem class AbstProblemClass (line 35) is also the root of the abstract tree. The concrete component of the newly created abstract problem node APN is the root Node itself (line 37).

In case of Node as reduction node, lines 40 to 63 present the similar algorithm to find an existing abstract reduction node or make a new abstract reduction node in the abstract reduction tree. Similarly, lines 64 to 88 present the algorithm to find an existing abstract solution node or make a new abstract solution node in abstract reduction tree. The only difference between the algorithm for abstraction of problem nodes and the other types of nodes is the problem node may have an implicit abstraction which is the abstract reduction node. In this case, that problem node is located in the reduction sub-tree – the sub-tree whose root and leaves are reductions.

Line 89 indicates that if Node and its children do not have the abstraction then there is no need to go down further; because there cannot be any abstraction below the Node; it would violate the parent child relationship, if there were.

3.4.8 Complexity Analysis of Generation of Abstract Reduction Trees

According to Cormen (1997), the cost of traversing a tree $RT = (V, \delta_t)$ using the breadth-first strategy is $O(N_V + N_\delta)$ where $N_V = |V_t|$ is number of nodes in the tree and $N_\delta = |\delta_t|$ is number of edges in the tree. The algorithm uses the breadth-first search to enumerate all nodes of the concrete reasoning tree. Therefore there are N_v WHILE loops

(line 6) and enumeration of them costs $O(N_V + N_\delta) = O(N_v + N_v - 1) = O(N_v)$. Each WHILE loop consists of a sequence of statements which abstract the current node in the loop. There are three types of nodes, so there are three corresponding conditions (line 8 for problem node, line 40 for reduction node and line 64 for solution node). For each type of node, there are similar operations in sequence such as:

[a] retrieve the class of the current node, which is the concrete class,

[b] retrieve the associated abstraction rule for that concrete class,

[c] abstract the concrete class to the abstract class,

[d] build an abstract node out of the abstract class,

[e] link the abstract node to the existing abstract reduction tree.

For a problem node, there is one exception:

[f] For the root node of the concrete reasoning tree, there is no need to link the abstract node to the abstract tree, because the abstract tree does not exist at that time, and the abstract node becomes the root of the abstract tree.

[g] get all direct children and add to the queue (if needed)

From the specifications given above, we can compute the complexity of the algorithm of generation of the abstract reduction tree. First of all, we want to compute the complexity of AssociateAbstractionRule. The algorithm contains one FOR loop which enumerate all the abstraction rules. Let N_{ar} be the number of abstraction rules, the algorithm will cost $O(N_{ar})$.

Next we compute the complexity of GetAbstractionRule. This method calls the statements that cost *O(1)*. Therefore, the complexity of GetAbstractionRule is *O(1)*.

Now we compute the complexity of GenerateAbstractReductionTree. For each WHILE loop (line 5), all statements (see [a] to [g]) cost a constant *O(1)*. In other words, N_v WHILE loops cost $O(N_v)$. The whole algorithm thus costs $\boldsymbol{O(N_{ar}) + O(N_v)}$ which is linear with number of abstraction rule and linear with number of reasoning nodes**.**

4. Abstraction-Based Collaborative Problem Solving

Based on the definition of abstraction of a reasoning tree for collaborative problem solving (see Section 3.3), we have developed a new approach to facilitate the problem of viewing and understanding a very large reasoning tree. The approach is called Abstraction-Based Table of Contents. The table of contents (TOC) is, in fact, the abstract reasoning tree of a concrete reasoning tree. The user who wants to view the complex tree can browse it by navigating the abstract tree.

4.1. Abstraction-Based Table of Contents

Figure 33 shows the TOC of the large tree displayed on the left hand side panel of Figure 24. In Figure 33, the right hand side panel displays the smaller sub-tree presenting the logic that reduces a main problem "*Assess whether Al Qaeda has nuclear weapons*" to its main sub-problems such as

- *"Assess whether Al Qaeda considers deterrence as a reason to obtain nuclear weapons"*
- *"Assess whether Al Qaeda considers the use of nuclear weapons in spectacular operations as a reason to obtain nuclear weapons"* and so on

The first level of the abstract subtree in the TOC, which is shown on the left-hand side of Figure 33, is the abstraction of the concrete tree shown in the right hand side of Figure 33. From the user's point of view, the top of the tree in the TOC corresponds to

the top node in the concrete tree, and the sub-nodes in the TOC correspond to the leaf nodes of the concrete tree (as indicated by the arrows from the figure). Moreover, the names of the leaf nodes in the concrete tree (such as those shown above) are abstracted into the names of the sub-nodes in the TOC (e.g. "*Self defense as reason", "Spectacular operations as reason"* and so on).

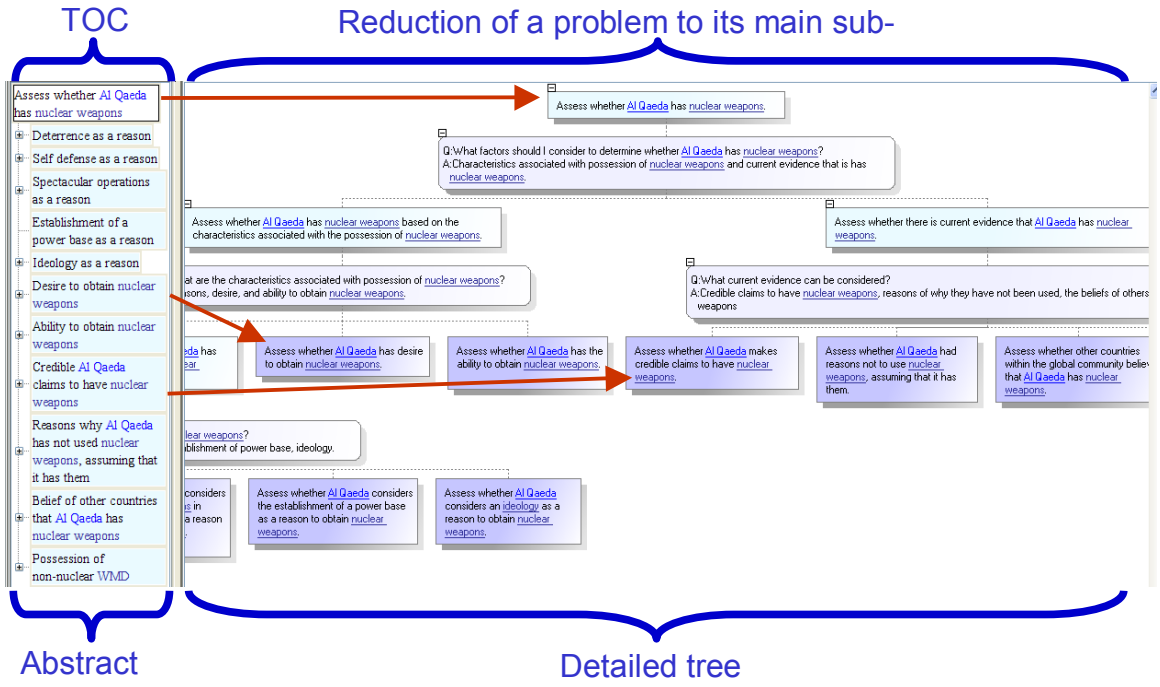


Figure 33: Abstraction of Reasoning Tree as Table of Contents

The leaves of the sub-tree on the right hand side panel of Figure 33 are also the roots of sub-trees in the TOC, as illustrated in Figure 34. This figure presents a fragment of abstraction-based TOC. It shows an abstract tree of the reasoning tree generated by the agent. We can see the top problem is *Assess whether Al Qaeda has nuclear weapons* and

sub-problems are *Deterrence as a reason, self-defense as a reason.* Each of the sub-problems is assessed by favoring and disfavoring evidences. Each evidence is assessed by the relevance, believability of the reporter and the source and so on. When the user clicks on the node, the right hand side shows a reasoning tree for that particular problem to explain the logic of reduction.

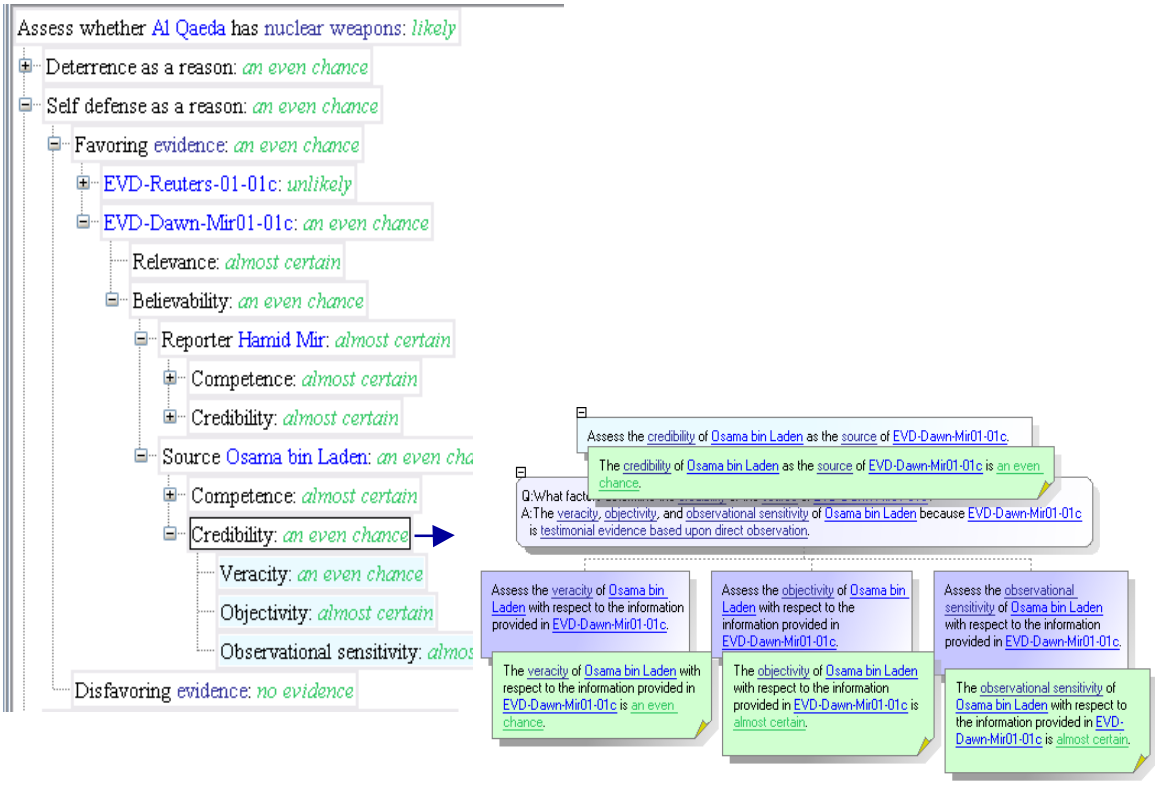


Figure 34: An Expanded Fragment of TOC

This type of abstraction is context dependent where the content of the abstract node is dependent on the context where it is located. For instance, the TOC item "*Favoring evidence"* implicitly indicates the evidence to support the hypothesis "*Self defense as a reason"* which is its parent TOC item.

Given the abstract tree as TOC, the browsing of the large reasoning tree becomes easier. Viewing the TOC gives the user the summary of the content of the concrete reasoning tree at different levels of abstraction. The higher level is presented first and the drill down of the TOC item gives more detailed information. For example, Figure 34 presents the drill-down of the "*Self defense as a reason"* node which is "*Favoring evidence"* and "*Disfavoring evidence"*. The "*Favoring evidence"* has two evidence pieces, "*EVD-Reuters01-01c"* and "*EVD-Dawn-Mir01-01c"*. Each of the supporting evidence's characteristics such as "*Relevance"* and "*Believability"* are also presented. By clicking on a TOC item one can view the corresponding concrete reduction sub-tree.

The abstraction of reasoning tree for interactive problem solving also supports the synthesis process. Figure 35 presents the synthesis view of Figure 33. The solutions are abstracted in the TOC together with their reduction counterparts. For instance, the node *"Assess whether Al Qaeda considers an ideology as a reason to obtain nuclear weapons"* and its solution "*It is likely that Al Qaeda considers an ideology as a reason to obtain nuclear weapons"* are abstracted to "*Ideology as reason: likely."*

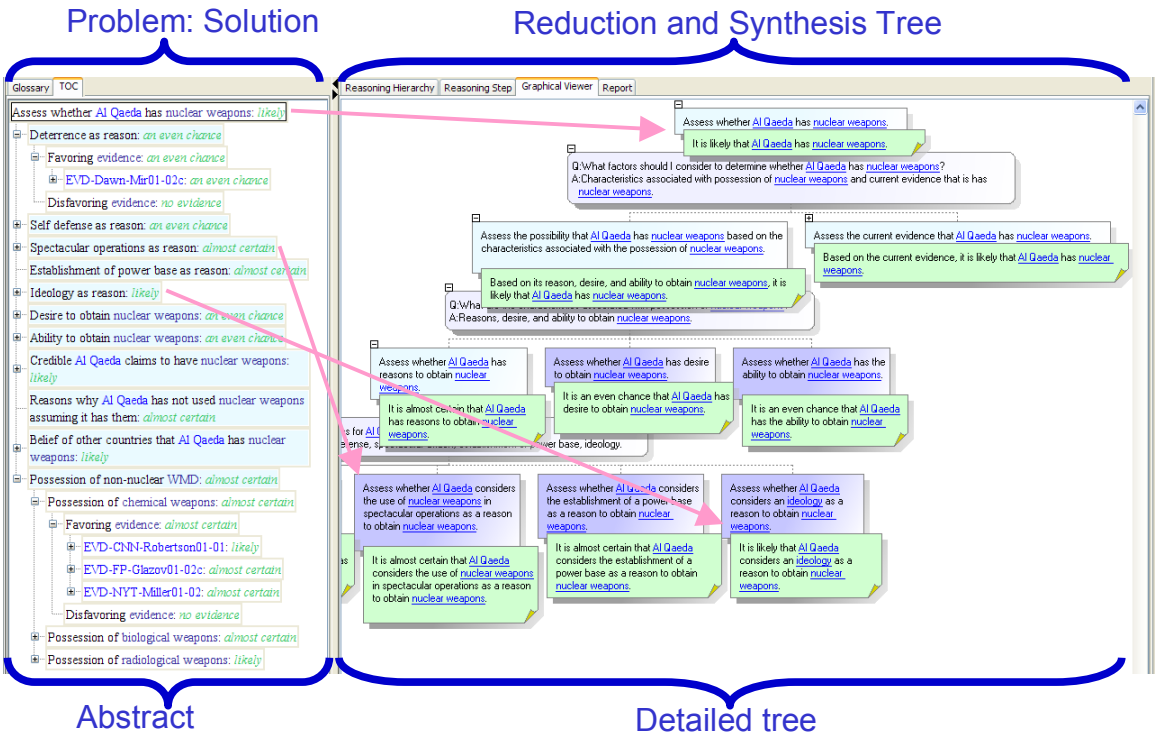
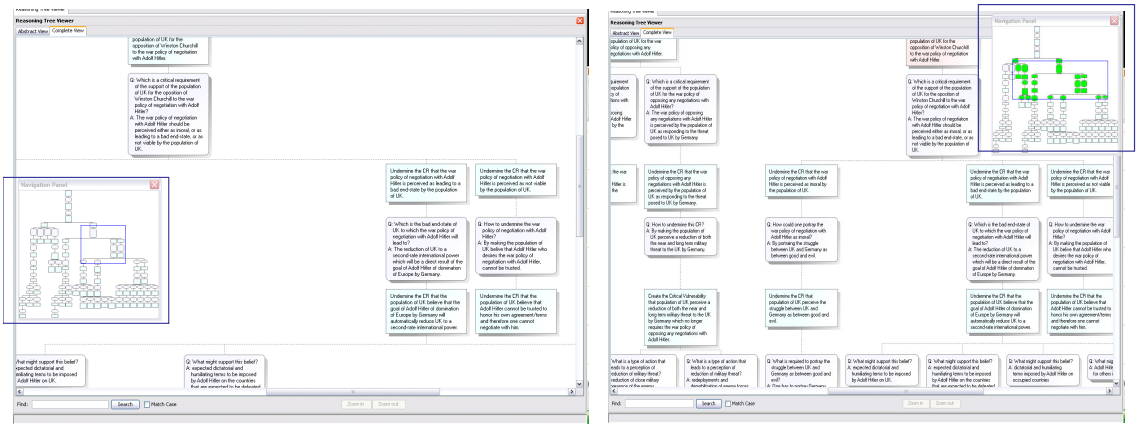


Figure 35: Abstract and Concrete Reduction and Synthesis Tree

4.2. Optimization of the Reasoning Tree Display

Even with the help of abstraction, the display of a reasoning tree on a small screen is difficult (Nguyen et al., 2000). We have therefore developed a technique to optimize this display, as illustrated in Figure 36. The left-hand side picture in Figure 36 displays a reasoning tree with a navigator showing the small part of the tree which is visible. The tree by itself is compactly displayed, but the view port at different locations is still spacious. In other words, the density of the tree is not evenly distributed. The right-hand side picture shows an optimized view which can display 150% more nodes in the same view port. This allows more nodes to be viewed in the same view port by reducing the white space between nodes while still preserving the characteristic of a hierarchical tree.



Before Optimization After Optimization

Figure 36: Optimization of the Display of a Large Reasoning Tree

4.3. Evaluation of Abstraction for Collaborative Problem Solving

In Fall 2006 and Spring 2007 we performed two experimentations with using the abstraction-based TOC. One was in the course CS681-2006 *Designing Expert Systems* at George Mason University and the other was in the course MAAI-2007 *Military Application of Artificial Intelligence* at the US Army War College. Both used the same abstraction-based TOC to browse and modify a large reasoning tree. At the end of the class, they were asked to agree or disagree on some certain statements. A sample of the students' subjective evaluations is presented in Figure 37. With one exception, all the students agreed or strongly agreed that the abstraction-based TOC facilitates the browsing and understanding of the reasoning trees.

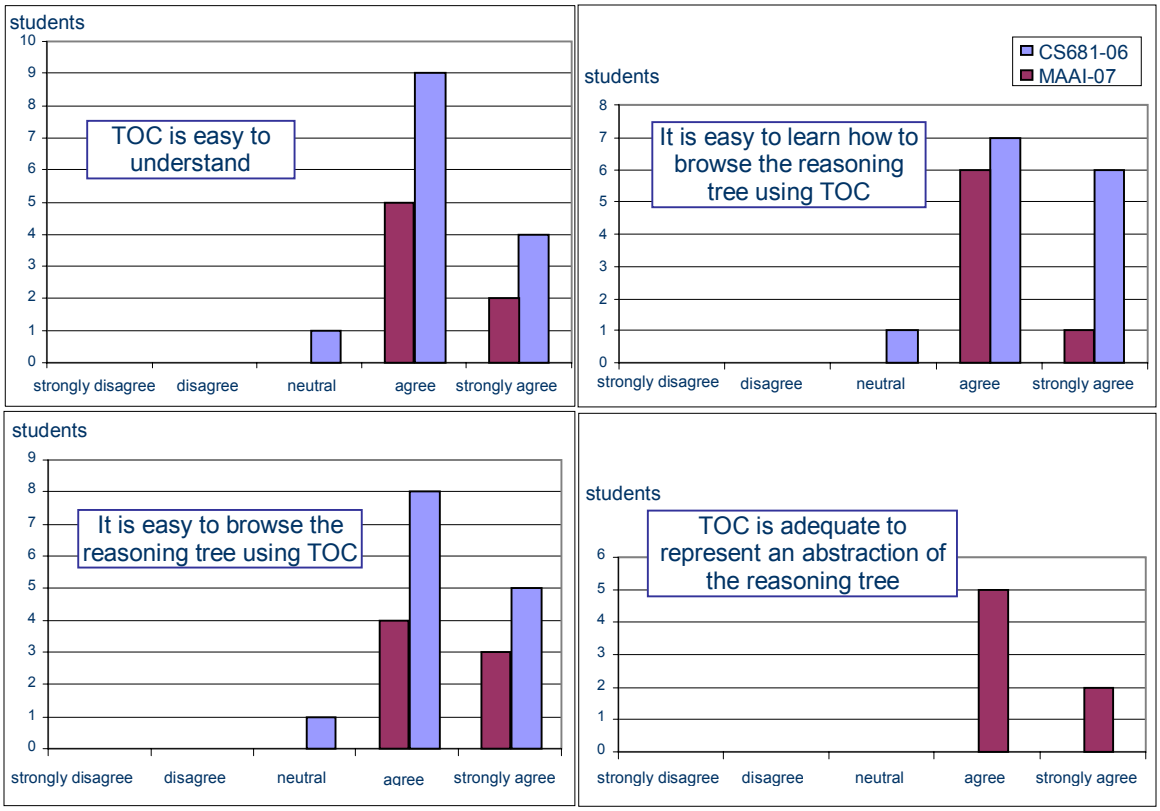


Figure 37: Evaluation of Abstraction for Collaborative Problem Solving

5. Abstraction-Based Tutoring

The intelligent tutoring systems (ITS) are valuable educational tools. They are used to assist the teachers in teaching as well as to support students in learning. These tools, however, are not widely available because the process of building them is very complex and time-consuming. This chapter presents several methods that facilitate the process of developing systems for tutoring expert problem solving. First we present an abstraction-based approach to lesson design and generation. Then we present several methods for learning and generation of exercises to test the students.

5.1. Lesson Design and Generation

Lesson creation is one of the most difficult and time consuming tasks in developing intelligent tutoring systems. Anderson estimated that "it takes at least 100 hours to do the development that corresponds to an hour of instruction for a student" (Anderson, 1992). According to Aleven and Rose (2004) "A recent estimate puts development time at 200 hours per hour of instruction". This activity puts a difficult burden on the instructor who designs and builds the lessons. The more complex the domain, the harder and longer it takes to build the curriculum for the tutoring system. GUIDON (Clancey, 1987), a classic tutoring system based on an expert system, took a subject matter expert and a full-time knowledge engineer six years to make it work. The enormous labor that is required to build the lessons for a tutoring system is one of the reasons the ITSs have not been

widely developed and used, in spite of their obvious benefits. We have developed a new, abstraction-based, approach to teach expert problem solving to students. *The corresponding abstraction-based lesson design and generation methods reduce the complexity and time for building the curriculum. They not only reduce the time to develop the tutoring system, but also generate the lessons automatically*.

The abstraction-based lesson design and generation process uses the abstraction of the reasoning trees of the application domain as the resource to build the lessons. An abstract reasoning tree is much smaller than its corresponding concrete reasoning tree and consists of precisely those abstract concepts and reasoning strategies that need to be learned by a student. This makes the task of the instructor who has to build the lessons out of the tree much easier. The detailed description of the abstraction of a reasoning tree for tutoring is presented in Chapter 3. Once the abstract lessons are built from the abstract tree, the examples for the lessons are generated automatically by concretizing the abstract components of the tree. The concretization of the abstract tree allows the reuse of the abstract lessons for different domain knowledge bases. Another feature of the abstraction-based lessons is the assured consistency between the expert's knowledge from the system's knowledge base and the knowledge used in constructing the curriculum to teach expert problem solving to the students.

5.1.1 Abstraction-Based Lesson Design

An abstract reasoning tree is a representation of some of the problem solving strategies used by a subject matter expert. Capturing that knowledge systematically and presenting it pedagogically is required in order to develop a tutoring system that can

teach the students the expert knowledge to solve problems in a particular domain. The way the tutoring system teaches the student is also similar to how it was taught by the expert because the reasoning tree is the representation of how the expert has taught the system in the first place. The abstract reasoning tree serves as a guide to construct the lessons. As mentioned in Chapter 3, the abstract reasoning tree (which includes the abstract reduction tree and the abstract synthesis tree) consists of hierarchies of four types of abstract nodes: abstract problem nodes, abstract reduction nodes, abstract solution nodes and abstract synthesis nodes.

A lesson can be defined to cover a part of an abstract reasoning tree. In general, a lesson teaches a strategy to solve a particular type of problem. Therefore the lesson is associated with an abstract problem node. This association constitutes a one-to-one mapping between the knowledge learned from the expert and the knowledge to be taught by the tutoring system. In order to solve a problem, the problem reduction paradigm guides the system to successively reduce that problem to simpler and simpler problems. That reduction strategy must be captured in a lesson. Depending on the complexity of the problem, the sequence of the reductions needed to solve the problem can be short or long. The lesson that teaches how to solve that type of problem must present the necessary reasoning steps. Each reasoning step may correspond to a lesson as a lesson section. Therefore a lesson can contain one or more lesson sections, depending on the complexity of the problem at hand. The relations among the lesson sections can vary. They can be sibling relations, cousin relations or parent-children relations. Thus, the lesson can be

used to represent and teach the knowledge that reduces a problem to simpler sub-problems via multiple reasoning steps.

To illustrate the lesson design process, we will use the abstract reasoning tree from Figure 38. The top level problem is successively reduced to simpler problems, as follows:

We need to

- *Assess the believability of the reporter of the piece of evidence.*

The believability of the reporter of a piece of evidence is determined by the reporter's competency and credibility.

Therefore we need to

- *Assess the competency of the reporter of the piece of evidence.*
- *Assess the credibility of the reporter of the piece of evidence.*

The credibility of the reporter of a piece of evidence depends on reporter's veracity, objectivity, and observational sensitivity.

Therefore, to assess reporter's credibility we need to:

- *Assess the veracity of the reporter of the piece of evidence.*
- *Assess the objectivity of the reporter of the piece of evidence.*
- *Assess the observational sensitivity of the reporter of the piece of evidence.*

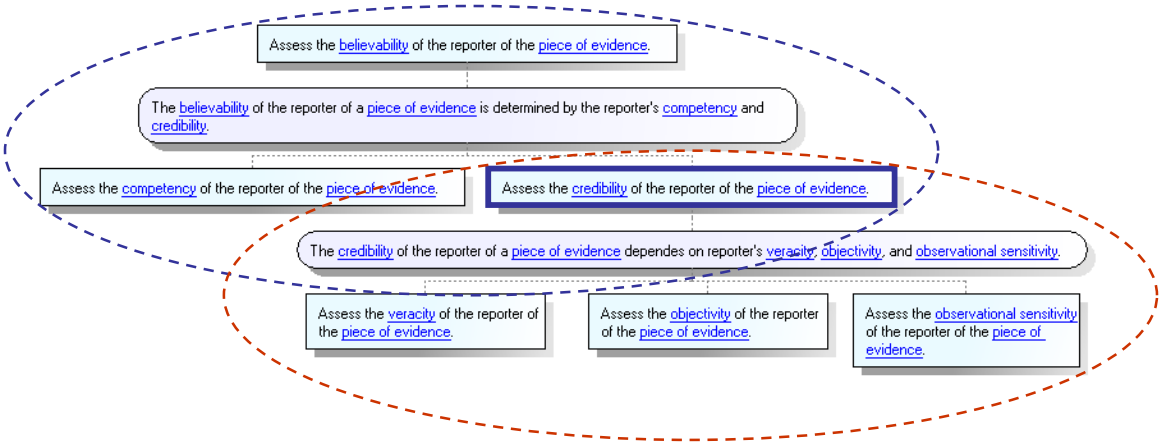


Figure 38: Lesson Sections

A lesson that teaches how to assess the believability of the reporter can be defined based on these two reductions. As a result, a two-section lesson is defined. The first section covers the first reasoning step and the second section covers the other one. The two sections share one problem, as depicted by the blue-border problem in Figure 38. The shared problems are required to link the sections together to ensure the continuity of the lesson's flow.

The lesson designer or instructor uses the abstract reasoning tree as guidance in designing the lessons of a curriculum. Once the instructor has decided what sections to include in a lesson and how they are linked together, he or she can elaborate more on how to teach that lesson to the students. Showing the entire lesson to the student is not always desirable because it can be confusing and misleading. A long lesson which has multiple sections needs to be shown one part at a time and follows some natural logic. The lesson sections can be presented in multiple ways: breadth-first, depth-first, a combination of breadth-first and depth-first, or any way that the instructor deems fit to the student's

knowledge or to his/her own taste. The breadth-first strategy introduces the problem in a broad way that helps the student appreciate the big picture before going into details. On the contrary, a depth-first strategy may help train the students the capability to focus on one particular problem and narrow down the problem to find the suitable solutions.

A lesson is not complete without the examples to illustrate the points being taught at an abstract level. Using the abstract reasoning tree, the instructor is able to avoid the burden of creating the examples for the lessons. As described in detail in Section 3.3, the abstract reasoning tree consists of hierarchical abstract nodes. Each abstract node is an abstraction of a set of concrete reasoning nodes in a concrete reasoning tree, as shown in Figure 39. The concrete reasoning nodes are illustrations of the abstract node. Therefore, the concrete reasoning nodes are the sources of the examples for the lessons built upon abstract node.

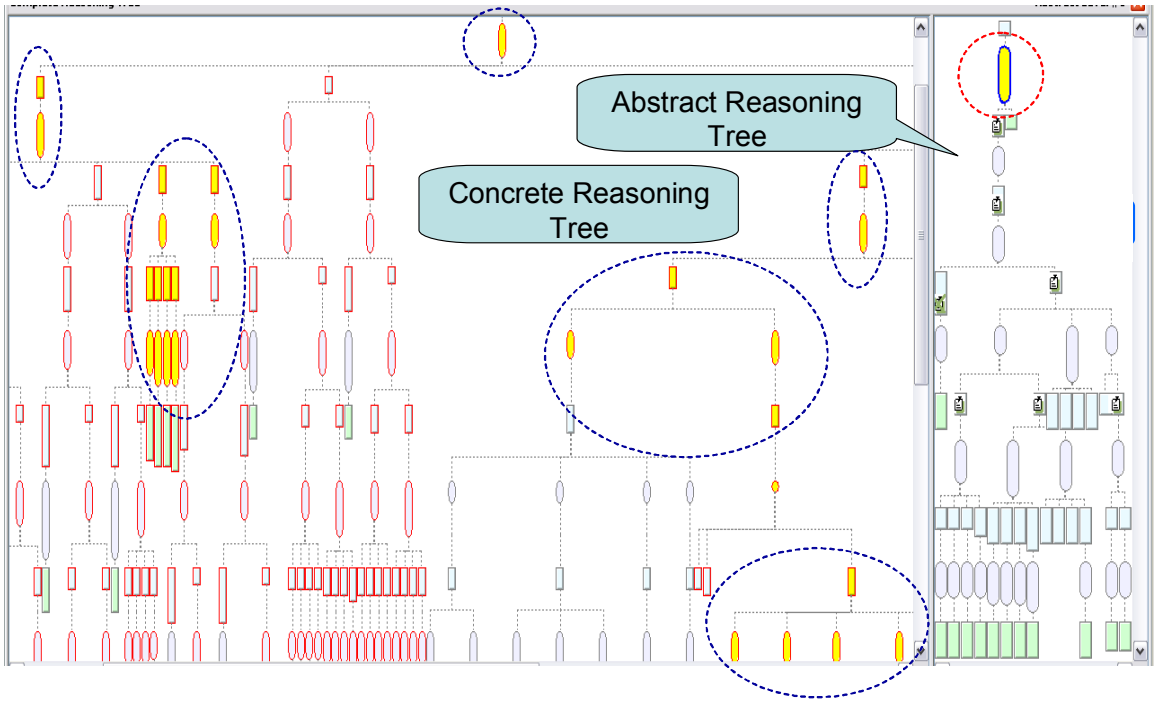


Figure 39: Abstract Node and Its Concretions

For instance, the following abstract problem (see Figure 38)

- *Assess the believability of the reporter of the piece of evidence.*

is an abstraction of the following concrete problems:

- *Assess the extent to which one can believe Hamid Mir as the reporter of EVD-DawnMir-01-02.*
- *Assess the extent to which one can believe Glazov J. as the reporter of EVD-FP-Glazov01-01.*

EVD-DawnMir-01-02 is a fragment of an article by *Hamid Mir,* published in the *Dawn* magazine. *EVD-FP-Glazov01-01* is a fragment of an article by Glazov J, published in the Front Page magazine.

The abstract reduction

> *The believability of the reporter of a piece of evidence is determined by the reporter's competency and credibility.*

is the abstraction of the concrete reductions:

> *What factors determine the extent to which Hamid Mir a reporter EVD-DawnMir-01-02 can be trusted?*
>
> *The competency and the credibility of Hamid Mir.*

and

> *What factors determine the extent to which Glazov J. a reporter of EVD-FP-Glazov01-01 can be trusted?*
>
> *The competency and the credibility of Glazov J.*

The abstract sub-problem

- *Assess the competency of the reporter of the piece of evidence.*

is the abstraction of the concrete sub-problems:

- *Assess the competency of Hamid Mir as the reporter of EVD-DawnMir-01-02.*
- *Assess the competency of Glazov J. as the reporter of EVD-FP-Glazov01-01*

and so on. Therefore, the lesson will have a set of examples, two of which are shown in Figure 40.

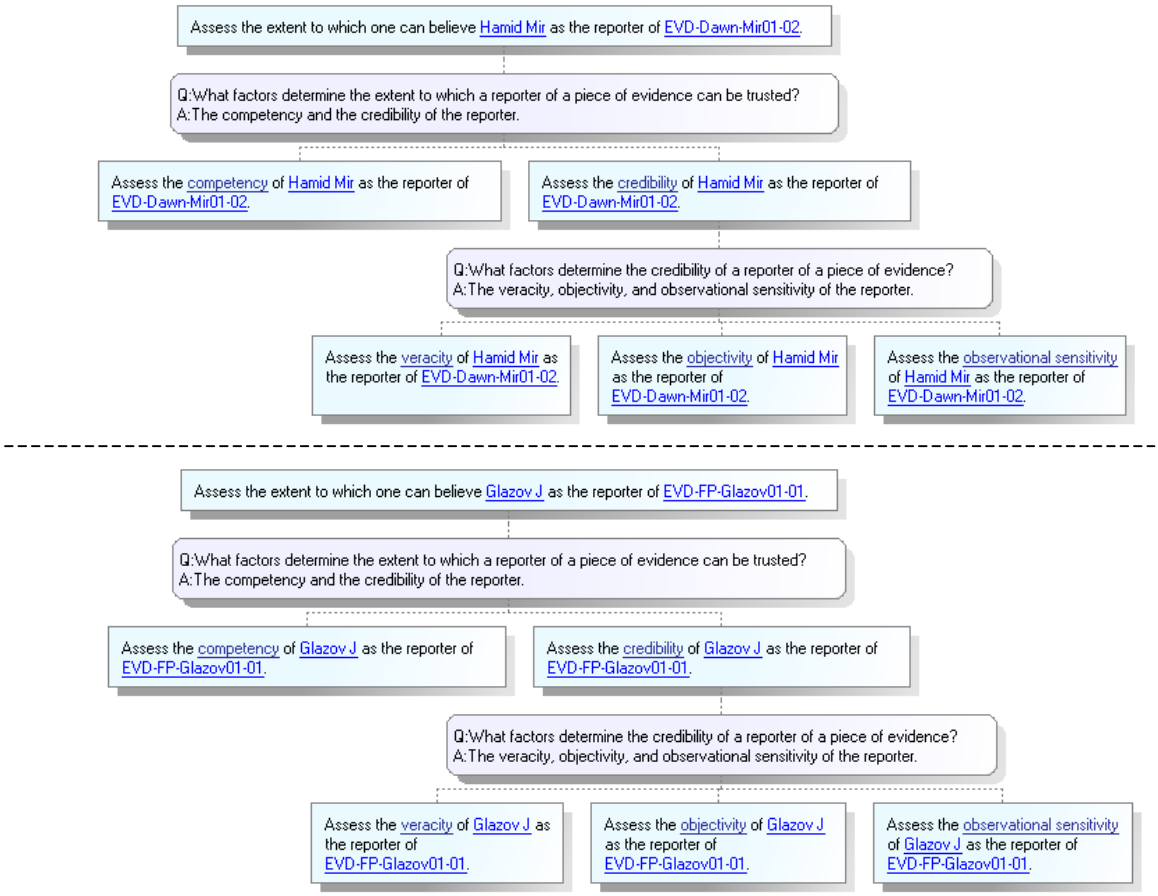


Figure 40: Examples Illustrating the Abstract Reduction in Figure 38

The lesson designer may also enhance the lesson with hyperlinks (as shown in Figure 40) that connect to the knowledge base to provide descriptions of important concepts and instances. These descriptions are generated automatically from the system's knowledge base. They provide an unintrusively help to the students. For example, Figure 41 shows the description of *EVD-FP-Glazov01*.

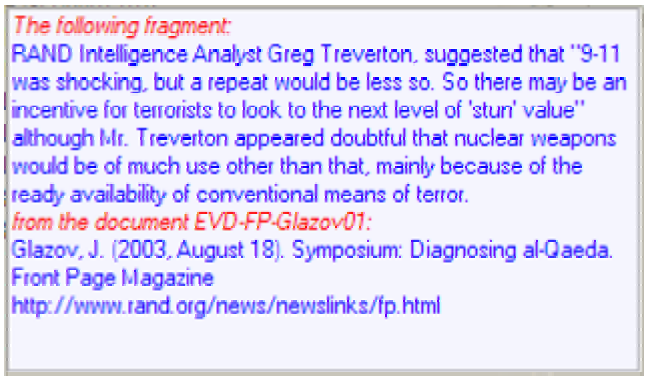


Figure 41: Description of a Piece of Evidence

What we have discussed so far is the reduction part of the problem reduction / solution synthesis paradigm. The other half is the synthesis process to find the solution of the original problem. According to this problem solving paradigm, the synthesis follows from bottom up: the solutions of the sub-problems are successively composed upward into the solutions of their parent problems. Similar to the reduction examples, the synthesis examples of the abstract syntheses are obtained from the corresponding concrete reasoning trees. Figure 42 shows a two-panel window. The upper panel shows the lesson's abstract synthesis steps (as green sticky notes) and the lower panel shows the corresponding concrete synthesis steps.

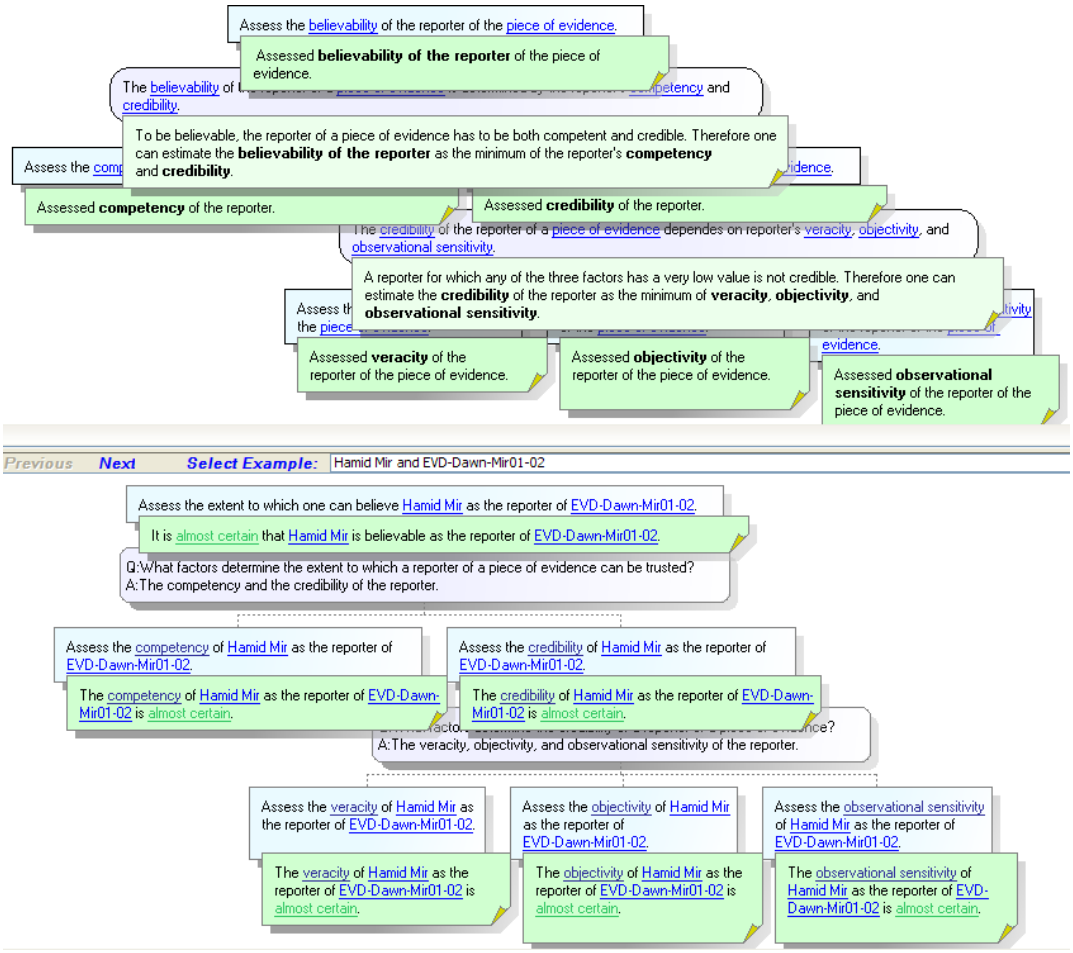


Figure 42: Lesson's Abstract Syntheses and their Concretions

An abstract synthesis teaches the student how to combine the solutions of some sub-problems to obtain the solution of their parent problem:

Let us consider the following solutions:

- *Assessed **veracity** of reporter of the piece of evidence.*
- *Assessed **objectivity** of the reporter of the piece of evidence.*
- *Assessed **observational sensitivity** of the reporter of the piece of evidence.*

*A reporter for which any of the three factors has a very low value is not credible. Therefore one can estimate the **credibility** of the reporter as the minimum of **veracity**, **objectivity**, and **observational sensitivity**.*

We thus obtain the:

- *Assessed **credibility** of the reporter*.

Similarly with the reduction part, the system can automatically generate concrete examples of the abstract synthesis process, as illustrated in the bottom part of Figure 42.

The abstraction-based lesson design is important in the sense that it partitions an abstract reasoning tree into multiple segments. Each segment conveys a separate topic and is captured into a separate lesson. Different ordered collections of lessons reflect different ways the lesson designer may direct the transfer of problem solving knowledge to the students, the goal being to find the most pedagogical way.

5.1.2 Lesson Script and Its Language

A lesson contains the lesson header and multiple sections. The lesson header includes the lesson's title and objectives. The lesson's objectives are the summary of what the lesson tries to convey and how the information can be used. Each section teaches a strategy to solve a particular problem. In other words, each section contains one abstract problem, one reduction strategy and its abstract sub-problems derived by the reduction. In addition to the reduction strategy, the synthesis strategy is taught as well. Therefore a lesson section also contains an abstract synthesis, abstract solutions of the abstract sub-problems and the synthesized abstract solution of the abstract problem. Both reduction examples and synthesis examples are added to illustrate the topic being taught. The

lesson section may also contain annotations and descriptions. These are optional components used to introduce certain components or for explanation purposes. Last but not least, the lesson also includes the long and short descriptions for certain concepts to enhance the understanding of the lesson content. All the descriptions are shown to the students upon request and unintrusively.

Once the lesson design phase is completed, the system automatically generates the lesson script whose content is based on the design. Each lesson has a lesson script. The entire curriculum consists of multiple ordered lessons. Abstraction-based lesson generation relies on the lesson scripts to build the lessons when they are needed. The lesson script is managed through the Abstraction-Based Lesson Emulation (ABLE) scripting language. ABLE allows the instructor to design and build the abstraction-based lessons in a very flexible manner. In fact, the instructor does not have to write a single line of ABLE to build the lesson script. The graphical interface helps him/her to generate the lesson script underlined by ABLE. ABLE is described in Appendix A

Each token of the lesson has an optional feature, named *LifeCycle,* which indicates the display timing of that token. The feature allows the lesson components to be displayed in different orders and with different durations. The grammar of this feature is presented in the Table 14. In *LifeCycle,* the two components *Order* and *Duration* indicate when and for how long to display a component on the screen. They are dynamically computed based on the current configuration of the lesson components.

The relative values of *Order* and *Duration* serve two purposes. First of all, the relative orders allow the lesson components to be easily added and deleted without

significantly affecting the orders and durations of the rest of the components. For instance, if *Objective's Duration* depends on the *Problem's Order* and the *Problem* is deleted for some reason, then the *Objective's Duration* will relies on the component *Token i* that is displayed right before *Problem*, i.e. *Objective's Duration = before(Token$_i$)* where *Problem's Order = after(Token$_i$)*. The other purpose that the relative values serve is to maintain the integrity of the orders and durations of the lesson components when the abstract reasoning tree is realized in different scenarios. Different scenarios may result in different abstract reasoning trees. No matter the configuration of the abstract reasoning tree, the lesson components that are hosted by that abstract tree can connect to each other by using the relative values of orders and durations. This characteristic is suitable for applying the same lesson script to different scenarios. More on this issue is discussed in the Lesson Generation section.

The *LifeCycle* feature is also used in implementing the tutoring strategies. A lesson can be a large hierarchical collection of sections. The displaying of the whole lesson at one time may become confusing and hard to understand. The instructor can design the displaying order of the lesson components in several ways, to emphasize the focal points of the lesson. For example, in Figure 38, a lesson with two sections is presented. The instructor may wish to introduce first the top reduction (as in Figure 43), or the bottom reduction (as in Figure 44)

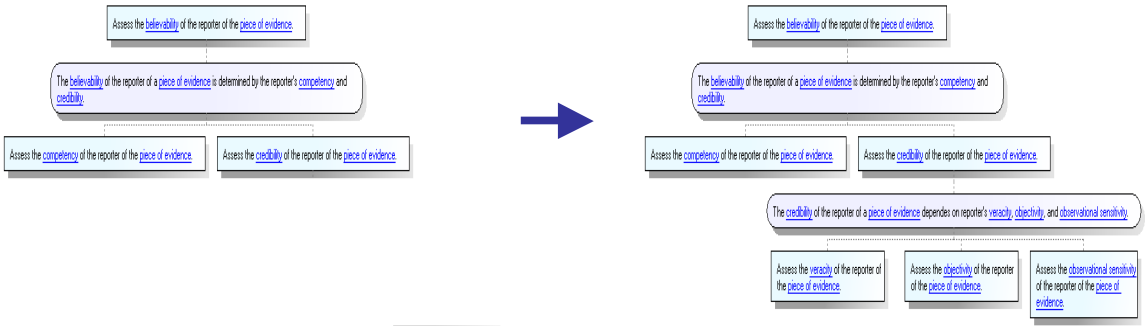


Figure 43: Top-down Tutoring Strategy

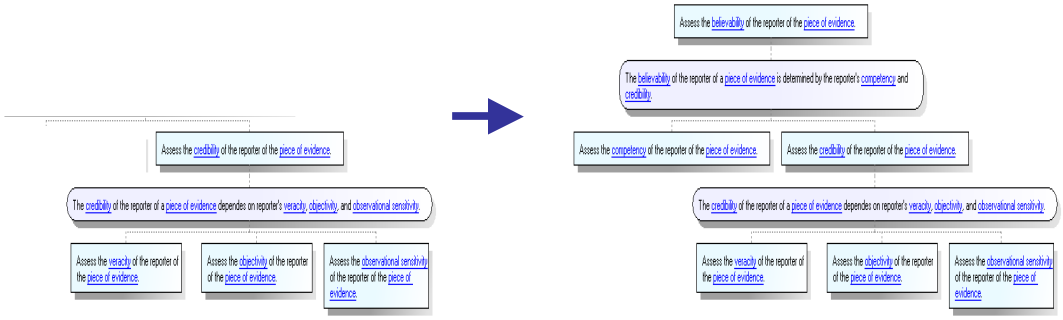


Figure 44: Bottom-up Tutoring Strategy

In short, there are various ways to arrange the display of a problem reduction process, to fit one's preferences. The tutoring system, however, has a default configuration for presenting the lesson to relieve the burden off the lesson designer. The default configuration is a variation of the depth-first strategy. The first reduction will be presented with all its sub-problems or solutions, and then the reduction of the left-most child, and so on, as shown in Figure 45.

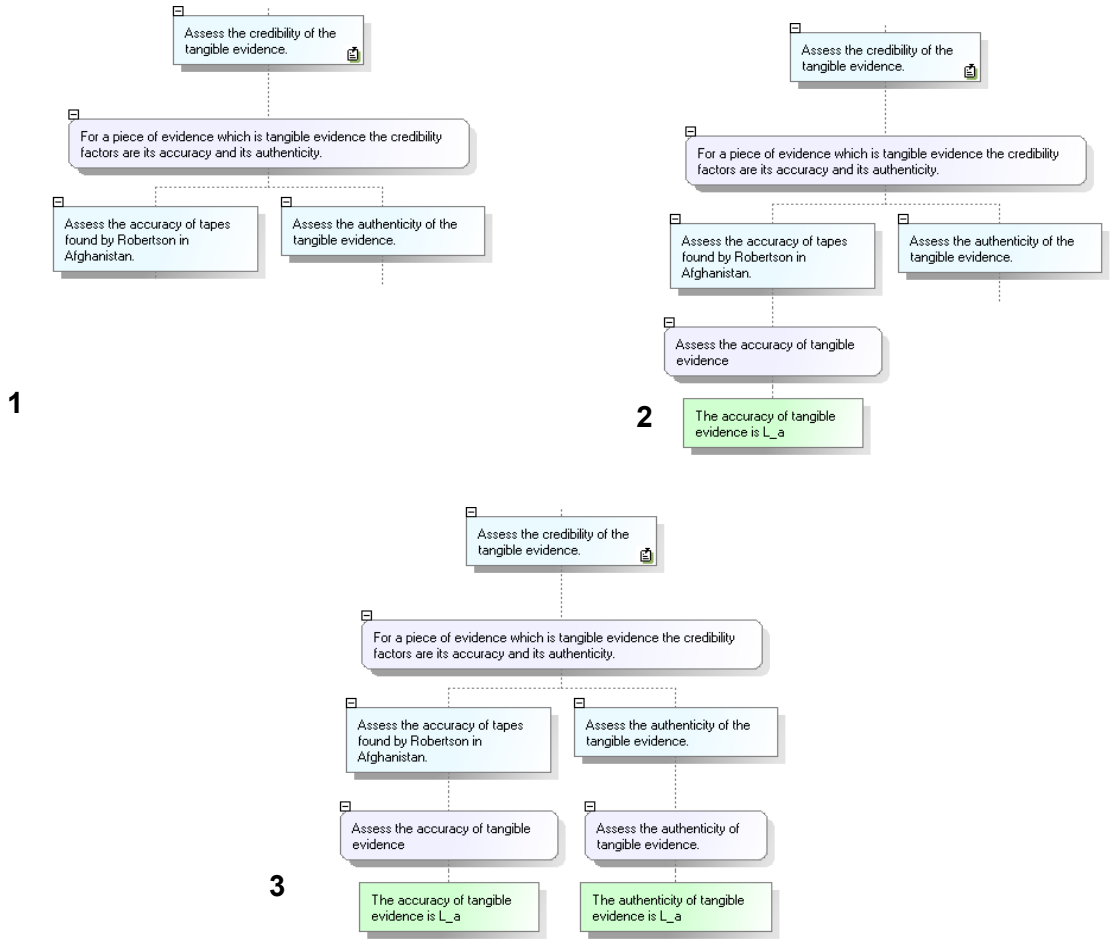


Figure 45: Variation of the Depth-First Strategy

The tutoring strategy also involves the ordering of the abstract problem solving strategies and their examples. By default, the abstract strategies are presented, and are then illustrated with concrete examples. This strategy reinforces the learning by using the examples as the illustrations of what has been taught. The order however can be changed to reflect the reverse order, i.e., the examples displayed first and the abstract strategies next. This approach presents first the examples and then the abstract problem solving strategy illustrated by them. Or the tutoring designer can mix abstract fragments with

examples. In essence, the order and mix of the abstract fragments and their examples can be modified by using the *LifeCycle* feature mentioned above.

5.1.3 Lesson Generation

The lesson generation process starts by invoking the script loader to load the XML files of the lesson scripts into the memory, in the order indicated by their indices. The curriculum is then created by executing the scripts in the corresponding order. The sequence of generated lessons is held together by the lessons' pre-requisites and post-requisites which are built based on the indexed lesson scripts. The lesson scripts are themselves linked to each other via the abstract problem references. The starting abstract problem reference of one script must be the ending abstract problem reference of another script. This makes the latter the pre-requisite of the former, and the former the post-requisite of the latter. These links are loaded to become the pre-requisites and post-requisites of the lessons.

Once the sequence of the lessons is laid out, the system splits the set of lessons into 3 groups:

- the previous lessons group, which have already been presented to the students;
- the proposed lesson, which is the lesson to be delivered next, and
- the next lessons group.

This classification is based on the information from the student's model which holds information about the student progress, as will be described in the Student Module section.

The organization of the lessons in the curriculum reflect the *chaining and logical sequencing of content* strategy (Dabbagh, 2007), where the lessons and their contents follow the hierarchical problem reduction/solution synthesis paradigm.

5.1.3.1 Table of Contents Generation

Each lesson teaches a strategy to reduce a problem to simpler sub-problems and to synthesize the solutions of simple sub-problems into the solution of the problem. Accordingly, the table of contents of each lesson has two main sections: the reduction section and the synthesis section. Figure 46 shows a typical table of contents. It was generated for the lesson addressing the "*Believability of the reporter of a piece of evidence"* and has entries for the individual sections (e.g. "*Components of believability*") and illustrations (e.g. "*Reduction examples*").

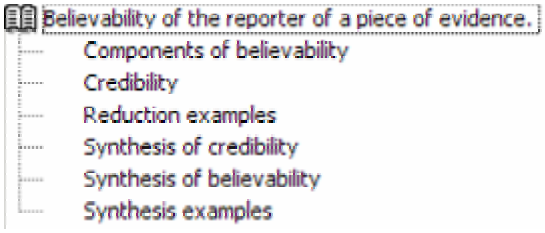


Figure 46: Lesson TOC

5.1.3.2 Lesson Content Generation

There are three types of lesson components: lesson decorative components, lesson header components and lesson section components.

The *lesson decorative components* are classified into two types: lesson annotations (to annotate a lesson components) and lesson definitions (to define the definitions for some terms).

The *lesson annotation* clarifies a lesson component with more explanations. Table 16 in Appendix B shows an XML script of *Annotation.* The annotation life cycle is by default very short - one step. The life cycle however can be expanded to serve some purposes. This component can be attached to any type of nodes except the decorative nodes themselves, i.e., there is not annotation of an annotation. The lesson designer is responsible for defining the content. Figure 47 shows an annotation that introduces a problem solving task, and some popup options.

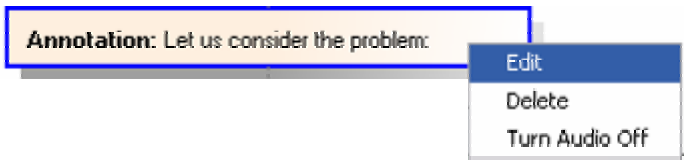


Figure 47: An Example of Annotation

The *lesson definitions* are another type of lesson decorative. There are two types of definitions that are built into the system, brief definitions and detailed definitions. The brief definitions are used as tool tips for lesson components and for quick access. The longer definitions define the terms in details and with examples for illustration. The lesson designer does not have to specify the terms to be described or does not have to think about the descriptions. The terms are the concepts and instances which come with the ontology. All the definitions are loaded from the ontology as well. This feature

relieves the lesson designer of the burden to provide the necessary definitions of the new concepts introduced in various lessons. Not only the system inserts the definitions automatically, it also allows the designer to customize the lesson definitions by selecting some of the terms to be inserted. Table 17 shows an XML script of *Definition.* The lesson definitions are displayed in two phases. The first phase displays the brief descriptions of the terms. The second phase shows the full descriptions if the "click here" hyperlink is invoked (see Figure 48). The full descriptions can be very large to cover a full-blown lesson about the term. In Figure 48, the full description is a lesson about evidence with supporting stories.

Figure 48: Lesson Definitions

There are two types of *lesson header components*: lesson title and lesson objective. They are special components because there is only one lesson title and at most one lesson objective in each lesson.

The *lesson title* is the start of a lesson which summaries the lesson content (see Figure 49). Table 18 shows an example of lesson title script.

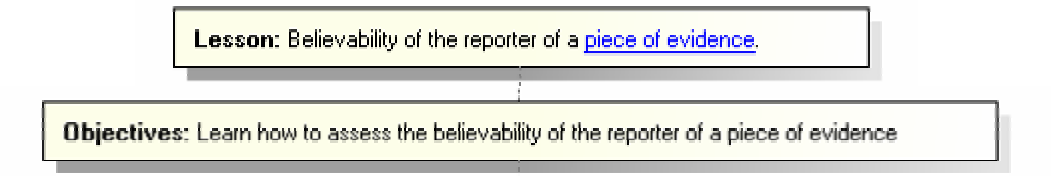


Figure 49: Lesson Title and Lesson Objective

The *lesson objective* is an optional complement of the lesson title. Its function is to emphasize the purpose of the lesson (see Figure 49). Table 19 shows an example of the lesson objective script.

Figure 38 shows a hierarchical set of *lesson section components*. There are lesson problems and lesson reductions. In that figure, the lesson section components contain information about problems and reductions respectively.

The *lesson problem* is the lesson component that links to the abstract problem in the abstract reasoning tree. The lesson problem covers the problem that will be reduced to simpler sub-problems in the reduction process. Table 20 shows a sample of lesson problem script in the XML format. When the lesson problem is constructed, its examples are also formed, by reference to the abstract reasoning tree.

The *lesson reduction* is the lesson component that links to the abstract reduction in the abstract reasoning tree. The lesson reduction teaches the problem solving strategy that reduces a particular problem to some simpler sub-problems or results. Table 21 shows a

sample of lesson reduction script XML. The link to an abstract reduction in the abstract reasoning tree serves as a bridge to load the concrete reasoning from the concrete reasoning tree to become the reduction examples.

After each lesson section, there usually are some examples that illustrate the lesson learned. For the reduction process there are reduction examples, and for the synthesis process there are synthesis examples. The examples are generated automatically by the system based on the abstract reasoning tree.

The process of generating the reduction examples is described as following. The abstract reasoning tree is built from the concrete reasoning tree. Each abstract node in the abstract reasoning tree is the abstraction of one or several concrete nodes in the concrete reasoning tree. As detailed in Chapter 3, there are three types of abstract reduction nodes: abstract problem nodes, abstract reduction nodes and abstract solution nodes. An abstract problem node is the abstraction of a set of problem nodes in the concrete reasoning tree. An abstract solution node is the abstraction of a set of elementary solution nodes in the concrete reasoning tree. An abstract reduction node is more complex being an abstraction of both problems nodes and reduction nodes in the concrete reasoning tree.

A reduction process in the abstract reasoning tree is captured in an abstract sub-tree that contains an abstract problem node, an abstract reduction node and a set of either abstract solution nodes or abstract problem nodes. Therefore that abstract sub-tree is, in fact, the abstraction of a sub-tree of a concrete reasoning tree. A lesson that is based on an abstract sub-tree is going to use the concretion of the abstract sub-tree as an example. Figure 40 shows two generated reduction examples for the lesson in Figure 38.

We use Figure 31 and Figure 50 to show how the lesson section is built from the abstract tree. The right hand side of Figure 50 shows a lesson section which was constructed from the abstract reasoning tree in Figure 31. The dimmed nodes are not included in the lesson section. The lesson section thus contains the lesson's problem LP_1, the lesson's reduction *LR1*, and the lesson's sub-problems LP_2, LP_3, LP_4, and LP_5. The abstract nodes in the lesson sections are the abstraction of the two sub-trees that are bordered by the broken blue lines on the left hand side of Figure 50. These two sub-trees are retrieved automatically during the lesson generation to be used as examples for the lesson section.

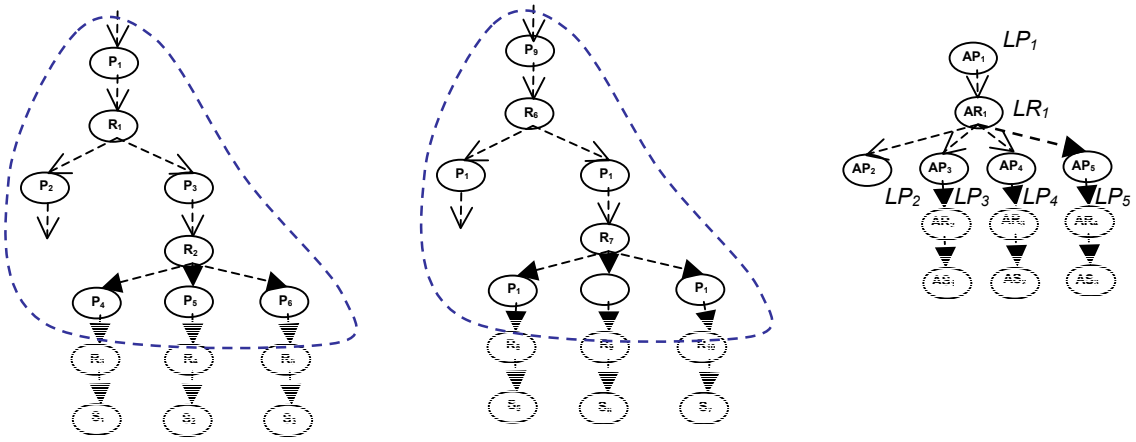


Figure 50: Lesson's Examples Generated for a Lesson's Section

Lesson solution is the lesson component that links to the abstract solution in the abstract reasoning tree. The lesson solution teaches how a solution is obtained. Table 22 presents a sample of the lesson solution script in the XML format. Figure 42 shows a sub-tree with a set of lesson solutions.

Lesson synthesis is a component of the synthesis process. The synthesis process is guided by the lesson synthesis which instructs the students how to compose the available solutions into the solution of a more complex problem. An example of lesson synthesis script in the XML format is shown in Table 23. An example of generated lesson syntheses is shown in the top part of Figure 42. In this figure, the lesson syntheses are differentiated from the lesson solutions by lighter green.

Figure 42 shows a snapshot of the synthesis process and the *synthesis examples*. The synthesis examples are generated automatically as their counterparts, the reduction examples. In the concrete reasoning tree, each problem node is associated with a solution node; each reduction node is associated with a synthesis node. The synthesis examples are then presented correspondingly to the reduction examples.

5.1.3.3 Lesson Text Generation

The lesson's text is, in essence, the text version of the lesson's content and it is generated from the content of the lesson's components. Figure 51 shows part of the lesson text in a text panel. This is the text which is spoken when the voice is enabled.

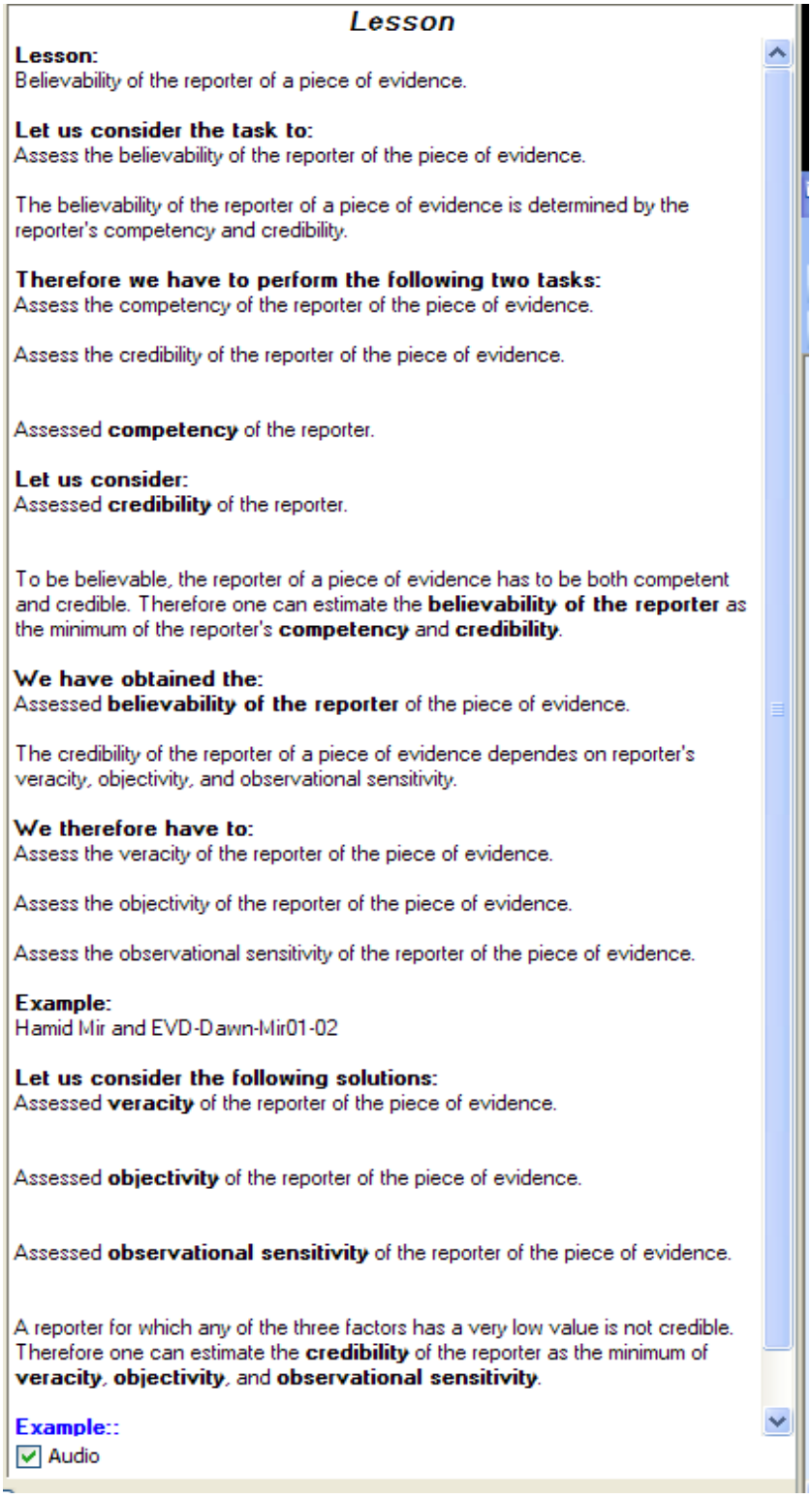


Figure 51: Lesson Text Panel

5.1.4 Lesson Generation Algorithm

There are two phases in lesson generation: abstract problem solving strategy generation and example generation.

5.1.4.1 Abstract Problem Solving Strategy Generation

The abstract problem solving strategy generation algorithm is described as follows:

Table 5: Abstract Problem Solving Strategy Generation Algorithm

Given:

- LSL - *list of lesson scripts*

Return:

- GLL - *list of generated lessons*

AbstractProblemSolvingStrategyGeneration(LSL **)**

```
1.  GLL ← ∅
2.  for each lesson script LS ∈ LSL do
3.      create lesson title
4.      create lesson objective (if any)
5.      create lesson definition for lesson title (if any)
6.      create lesson annotation for lesson title (if any)
7.      for each lesson section LSec ∈ LSecL ∈ LS do
8.          if lesson problem LP is not created then
9.              create lesson problem LP
10.             create lesson annotation for LP (if any)
11.             create lesson definition for LP (if any)
12.             create lesson solution LS for LP
13.         end if
14.         create lesson reduction LR
15.         create lesson annotation for LR (if any)
16.         create lesson definition for LR (if any)
17.         create lesson synthesis LS for LR
18.         for each sub-problem LP_i ∈ LSec do
19.             create lesson problem LP_i
20.             create lesson annotation for LP_i (if any)
21.             create lesson definition for LP_i (if any)
22.             create lesson solution LS_i for LP_i
23.         end for
```

24. **end for**
25. build lesson GL from lesson components above
26. add GL to GLL
27. **end for**
28. **return** GLL

end AbstractProblemSolvingStrategyGeneration

The algorithm shows how the abstract problem solving strategies are generated from the lesson scripts. Each lesson script generates a corresponding lesson which tutors the abstract problem solving strategy.

5.1.4.2 Example Generation

Note: according to Cormen (1997), the complexity of breadth-first traversing of a tree $RT = (V, \delta_t)$ is the same with complexity of depth-first one, which is $O(N_V + N_\delta)$ = $O(N_v + N_v - 1) = O(N_v)$ where N_V is number of vertices and N_δ is number of edges. The semantics of breadth-first search however is more meaningful in the problem reduction paradigm where a node is broken down into sub-nodes. Traversing the tree using the bread-first strategy makes more sense than using the depth-first strategy.

Table 6 describes the process of retrieving the concrete components from an abstract component. This process is frequently used in lesson example generation.

Table 6: Concrete Component Retrieval

Given:

- ARL - *a set of abstraction rules*
- AbstC - *an abstract component which is abstract problem class, abstract reduction class, abstract solution class or abstract synthesis class*.

Return:

- CCs - *a set of concrete components which are problem classes if abstract component is abstract problem class, reduction rules if abstract component is*

abstract reduction class, solution classes if abstract component is abstract solution class, synthesis rules if abstract component is abstract synthesis class.

RetrieveConcreteComponents(ARL, AbstC **)**

1. CCs – *list of concrete components*
2. CCs ← ∅
3. **for** each abstraction rule AR = (CC, AC) ∈ ARL **do**
4. **if** AbstC = AC **then**
5. add CC to CCS
6. **end if**
7. **end for**
8. **return** CCs

end RetrieveConcreteComponents

Table 7 describes the process of searching for the instantiations of a problem class, a solution class or a reduction rule. The instantiation of a problem class (or instantiated problem) is represented by a problem node in concrete reasoning tree. Similarly for the other types of statements, the instantiation of a solution class (instantiated solution) is represented by a solution node in concrete reasoning tree; the instantiation of a reduction rule (instantiated reduction rule) is represented by a reduction node in concrete reasoning tree.

Table 7: Search Instantiations

Given:

- C – *a class, which is a problem class or a solution class or a reduction rule*
- RT - *the reasoning tree*

Return:

- ICs - *list of instantiated classes which are problem nodes or solution nodes or reduction nodes*

SearchInstantiation(RT, C**)**

1. ICs ← ∅
2. Queue ← ∅

```
3.  add root of RT to Queue
4.  while Queue is not empty do
5.      Node ← pop a node from Queue
6.      Children ← get children of Node
7.      add Children to queue
8.      retrieve a class C' from Node
9.      if C' = C then
10.         add Node to ICs
11.     end if
12. end while
13. return ICs
return SearchInstantiation
```

The search of instantiations of a class, i.e., problem class, solution class or reduction rule starts from the root of a concrete reasoning tree (line 3). The algorithm uses breadth-first search (lines 4 to 7). For each node, a class is extracted from the node (line 8). To be specific, the problem class is retrieved from the problem node, the solution class is from the solution node, and the reduction rule is from the reduction node. Each of the classes C' is compared against the class C as argument (line 9). If they are the same, then add that node into the returned list (line 9 to line 11). The algorithm searches the entire tree, because there is no guarantee that the target node is not near the bottom of the tree.

Table 8 shows the lesson example generation algorithm.

Table 8: Lesson Example Generation Algorithm

Given:

- RT - *a reasoning tree*
- ARs - *a set of abstraction rules*
- GLs - *a set of generated lessons*

Return:

- GEs - *a set of generated lesson examples*

```
LessonExampleGeneration(RT, ARs, GEs)
1.  IPs ←∅   - IPs is a set of  instantiated problem nodes
2.  ISPs ← ∅ - ISPs is a set of instantiated sub-problem nodes
3.  IRs ← ∅ - IRs is a set of instantiated reduction nodes
4.
5.  for each generated lesson GL ∈ GLs do
6.      for each lesson section LSec ∈ GL do
7.          extract the abstract problem class AP from lesson problem LP in LSec
8.          set of problem classes PCs ← RetrieveConcreteComponents(ARs, AP)
9.          for each problem class PC ∈ PCs do
10.             IPs ← SearchInstantiation(RT, PC)
11.         end for
12.         extract the abstract reduction class AR  from lesson reduction LR in LSec
13.         set of reduction rules RdRs ← RetrieveConcreteComponents(ARs, AR)
14.         for each reduction rule RdR ∈ RdRs do
15.             IRs ← SearchInstantiation(RT, RdR)
16.         end for
17.         for each lesson sub-problem LSP ∈ LPs in LSec
18.             extract the abstract problem class AP' from lesson sub-problem LSP
19.             set of problem classes PC's ← RetrieveConcreteComponents(ARs, AP')
20.             for each problem class PC'∈ PC's do
21.                 temp ← SearchInstantiation(RT, PC')
22.                 add temp to ISPs
23.             end for
24.         end for
25.         Connect each IP in IPs to its child IR in IRs which in turn connects to its
                children in ISPs.
26.     end for
27.     add the examples to GEs
28. end for
29. return GEs
end LessonExampleGeneration
```

Lesson example generation is based on the generated lessons. The FOR loop on line 5 enumerates all generated lessons. For each lesson, the lesson sections are examined (line 6). For each lesson section, the abstract problem class is retrieved based on the reference to it from the lesson problem (line 7). From the abstract problem class and abstraction rules, a list of concrete problem classes is retrieved (line 8). The problem classes are then

used to retrieve instantiated problem nodes from the concrete reasoning tree (line 9, line 10). Similarly the abstract reduction class is obtained from the lesson reduction (lines 12). And the concrete reduction rules are obtained from the reduction abstraction rules (line 13). Line 14 and line 15 shows how the instantiated reduction rules which are reduction nodes are retrieved from the concrete reasoning tree. From the lesson sub-problems, the sub-problem nodes are also obtained. Three sets of problem nodes, reduction nodes and sub-problem nodes are linked together to become the examples for the lesson section. An enumeration of all lesson sections in one generated lesson also links all the examples for lesson sections together to become larger examples to illustrate the generated lesson.

5.1.5 Complexity Analysis of the Lesson Generation Process

The complexity of lesson generation is computed based on the two algorithms, described in Table 5 and Table 8, abstract problem solving strategy generation and lesson example generation.

5.1.5.1 Complexity of Abstract Problem Solving Strategy Generation

The algorithm of abstract problem solving strategy generation in Table 5 depends only on the lesson scripts. Let N_s be the number of lesson scripts; each with maximum N_{st} lesson sections. Each lesson section has one lesson problem, one lesson reduction and at most N_{sub} lesson sub-problems. For each computation of a reduction process, there must be at most one counterpart of the synthesis process, i.e., the lesson problem versus lesson solution, lesson reduction versus lesson synthesis.

First, we compute the second FOR loop of generating the lesson sections (line 7 of Table 5). Each lesson section has:

- One lesson problem plus at most one lesson annotation and one lesson definition. The lesson problem can be generated before if this section is not the first section. It means that the lesson problem of this section can be the lesson sub-problem of the previous lesson section. Each computation for generating a lesson problem is a constant *O(1),* similar to that of lesson annotation and lesson definition. Each lesson problem has at most one lesson solution which also costs a constant *O(1)*. In other words, each lesson problem plus its lesson decorations and its synthesis counterpart cost a constant *O(1)*.
- One lesson reduction plus at most one lesson annotation and one lesson definition and one synthesis counterpart – lesson synthesis. Similar to the lesson problem they also cost a constant *O(1)*.
- A third FOR loop (innermost FOR loop at line 17) for generating the lesson sub-problems plus at most one lesson annotation and one lesson definition for each lesson sub-problem and their lesson solutions of synthesis process. Similar to the lesson problem, each lesson sub-problem and its lesson decorations plus its lesson solution cost *O(1)*. Therefore the third FOR loop costs $O(N_{sub})$.

Thus the second FOR loop costs $O(N_{st})(O(1) + O(1) + O(N_{sub})) = O(N_{st}N_{sub})$ including the third FOR loop.

The first FOR loop (outermost for loop at line 2) is of the loop of N_s lesson scripts. Each lesson script consist of one lesson title, at most one lesson objective, at most one lesson annotation and at most one lesson definition plus the second for loop. As we discussed above, all the lesson header components are similar to the lesson decorations,

they cost a constant O(1). Hence, the first FOR loop costs $O(N_s)(O(1) + O(N_{st}N_{sub})) = O(N_sN_{st}N_{sub})$ including the second and third FOR loops. From the abstract tree point of view, $N_sN_{st}N_{sub}$ is linear with number of abstract nodes of the reasoning tree N_{an}. Therefore, the algorithm performs in $\boldsymbol{O(N_sN_{st}N_{sub}) = O(N_{an})}$.

5.1.5.2 Complexity of Examples Generation

The algorithm of lesson example generation in Table 8 depends on the algorithm in Table 7 for retrieving the instantiations of a knowledge component from the reasoning tree and the algorithm in Table 6 for retrieving the concrete components corresponding to an abstract component.

As shown by Cormen (1997), the cost of traversing a tree $t = (V_t, \delta_t)$ using either breadth-first or depth-first strategy is $O(N_V + N_\delta)$ where N_V is number of nodes in the tree and N_δ is number of edges in the tree. Therefore the cost of searching for the instantiations of a knowledge component in SearchInstantiation algorithm is $O(N_V + N_\delta)$. Once the traverse of the tree is finished, the map between a class and its instantiated classes are established to reduce the time for later searches. In other words, searching for the instantiations of all necessary classes cost only $O(N_V + N_\delta) = O(N_v)$ because $N_v = N_\delta + 1$, no matter how many times the search is called.

The algorithm of retrieving the concrete components from an abstract component (RetrieveConcreteComponents algorithm) consists of a loop of abstraction rules, each of which compare its abstract component against the searched one. If they are equal, the list of concrete components of that abstract rule is returned. The comparison operation costs a

constant O(1). Worst case scenario enumerates all the abstraction rules which costs $O(N_{ar})*O(1) = O(N_{ar})$ where N_{ar} is the number of abstraction rules. As similar to the SearchInstantiation algorithm, a map between the concrete classes and their abstract classes are established to reduce the time for later searches. In other words, retrieving concrete components from an abstract component takes only $O(N_{ar})$, no matter how many times the method is invoked.

In Table 8, the second FOR loop (line 6) is the loop of lesson sections. Each lesson section contains one lesson problem, one lesson reduction and a loop of lesson sub-problems. Each lesson problem, lesson sub-problem and lesson reduction retrieves a set of concrete components via RetrieveConcreteComponents which cost $O(N_{ar})$. As stated above, no matter how many times the method is invoked, the cost is only $O(N_{ar})$. Each concrete component (a problem class, a solution class or a reduction rule) retrieves a set of its instantiations (a set of problem nodes, a set of solution nodes, or a set of reduction nodes in concrete reasoning tree, respectively) that costs $O(N_v)$. As stated above, no matter how many times the method is invoked, the cost is only $O(N_v)$. In other words, the operation of a lesson component (problem or reduction or sub-problem) retrieving its own instantiations costs $O(N_{ar}) + O(N_v)$. Because the lesson section has one lesson problem, one lesson reduction and a loop of lesson sub-problems, the complexity of the whole lesson section is $O(N_{ar}) + O(N_{sub}) + O(N_v) = O(N_{sub} + N_{ar} + N_v)$ where N_{sub} is maximum number of lesson sub-problems per lesson section.

Let N_{gl} be the number of generated lessons, and N_{st} be the maximum number of sections in each lesson, the algorithm in Table 8 costs $O(Ngl \times Nst \times N_{sub}) + O(N_{ar}) +$

$O(N_v) = O(N_{gl}N_{st}\ N_{sub} + N_{ar} + N_v)$. From the abstract reasoning tree point of view, $N_{gl}N_{st}N_{sub}$ is linear with number of abstract nodes of the abstract tree N_{an}. In other words, the algorithm in Table 8 costs $O(N_{ar} + N_{an} + N_v)$.

5.1.5.3 Complexity of Lesson Generation

The complexity of lesson generation equals the complexity of abstract problem solving strategy generation plus the complexity of lesson examples generation. The complexity of the former costs $O(N_{an})$. The complexity of lesson examples generation is $O(N_{ar} + N_{an} + N_v)$. Over all, the complexity of lesson generation is:

$$O(N_{an}) + O(N_{ar} + N_{an} + N_v) = \boldsymbol{O(N_{ar} + N_{an} + N_v).}$$

5.1.6 Generality of Abstraction-Based Lesson Generation

The abstraction-based lesson generation is based on the abstract reasoning tree. As discussed above, the lesson section components are linked to the abstract nodes of the tree. Each abstract node is the abstraction of a number of reasoning nodes in concrete reasoning trees. An abstract problem node is an abstraction of concrete problem nodes. An abstract solution node is an abstraction of concrete elementary solution nodes. An abstract reduction node is an abstraction of concrete sub-tree consisting of problem nodes and reduction nodes. The concrete reasoning trees are generated by the problem solving engine which applies general reduction and synthesis rules to solve a given problem in the context of a given scenario.

Figure 52 shows an example of an IF-THEN reduction rule which was learned from a subject matter expert. This reduction rule can be instantiated in different scenarios of the same domain, as illustrated in the following. One such scenario is Intelligence Analysis

where intelligent analysts assess pieces of evidence that favors or disfavors the hypotheses under study. A similar scenario is Crime Scene Investigation where police officers investigate various crimes.

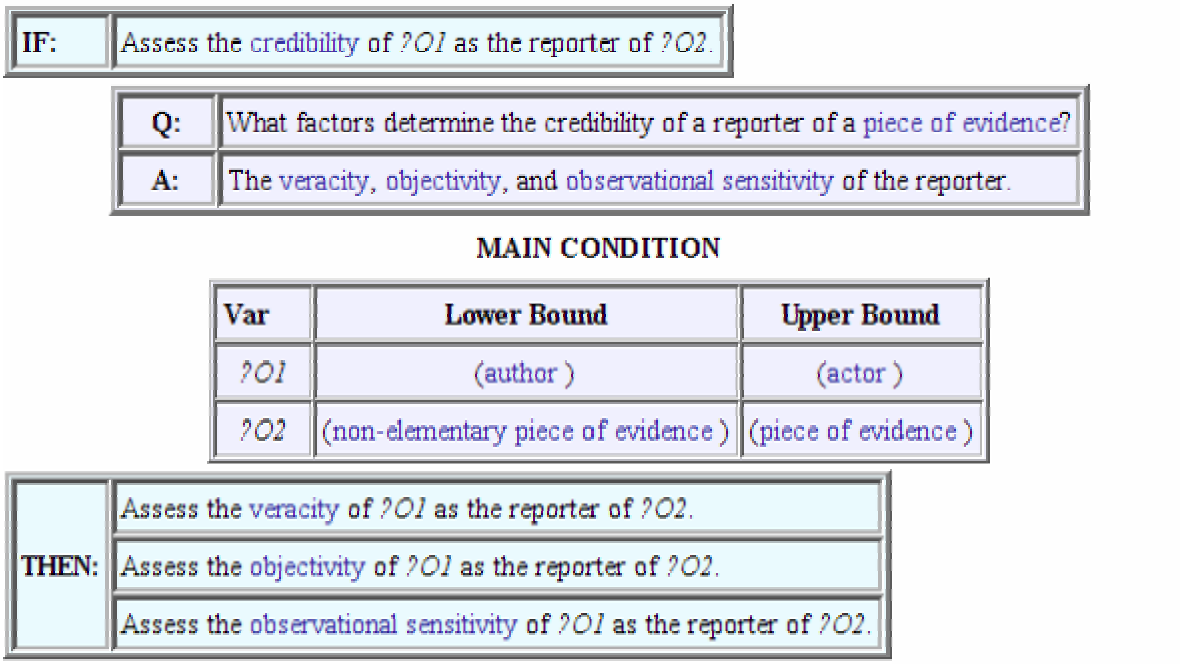


Figure 52: Reduction Rule

In the first scenario, the reduction rule can be instantiated as shown in Table 9. This rule questions the credibility of Hamid Mir, a reporter of Dawn Magazine who wrote an article about Bin Laden who was quoted as saying "*We have chemical and nuclear weapons as a deterrent and if America used them against us we reserve the right to use them.*"

Table 9: Instantiated Reduction Rule in Intelligence Analysis Scenario

INSTANTIATED REDUCTION RULE
***IF**: Assess the credibility of Hamid Mir as the reporter of EVD-Dawn-Mir01-02.* ***Q:** What factors determine the credibility of a reporter of a piece of evidence?* ***A:** The veracity, objectivity, and observational sensitivity of the reporter.* ***THEN**:* *Assess the veracity of Hamid Mir as the reporter of EVD-Dawn-Mir01-02.* *Assess the objectivity of Hamid Mir as the reporter of EVD-Dawn-Mir01-02.* *Assess the observational sensitivity of Hamid Mir as the reporter of EVD-Dawn-Mir01-02.*

In the second scenario the rule can be instantiated as shown in Table 10. In this scenario, the police officer Connolly reported that Sacco committed the robbery and shooting in South Braintree on April 15, 1920 [Schum, 1994].

Table 10: Instantiated Reduction Rule in Crime Scene Investigation Scenario

INSTANTIATED REDUCTION RULE
***IF**: Assess the credibility of Connolly as the reporter of a testimony under oath.* ***Q:** What factors determine the credibility of a reporter of a piece of evidence?* ***A:** The competency, veracity, objectivity, and observational sensitivity of the reporter.* ***THEN**:* *Assess the veracity of Connolly as the reporter of a testimony under oath.* *Assess the objectivity of Connolly as the reporter of a testimony under oath.*

Assess the observational sensitivity of Connolly as the reporter of a testimony under oath.

Let us now assume that an abstract reduction rule and the corresponding abstract problems are built from the instantiated reduction rule in the first scenario, as shown in Table 11. The lesson that is built from the abstract rule/reasoning in Table 11 can be used both in the Intelligence Analysis scenario and in the Crime Scene Investigation scenario, with examples generated automatically in each scenario.

Table 11: Abstract Rule Corresponding to the Rule Instance in Table 2

ABSTRACT RULE
***IF**: Assess the credibility of a reporter of a piece of evidence.*
***Q:** What factors determine the credibility of a reporter of a piece of evidence?*
***A:** The veracity, objectivity, and observational sensitivity of the reporter.*
***THEN**:*
Assess the veracity of a reporter of a piece of evidence.
Assess the objectivity of a reporter of a piece of evidence.
Assess the observational sensitivity of a reporter of a piece of evidence

There are two dimensions of generality of our approach to lesson design and generation. The first regards the automatic generation of lesson examples for different scenarios in the same domain, with no authoring or customization needed from the instructor. The second regards the ability to apply the same abstract lesson to different

knowledge bases. The first dimension expresses the capability to capture the essence of reasoning behind the problem solving approaches and to apply that knowledge into different problems of different scenarios in the same domain. The second dimension emphasizes the reusability of the abstraction-based lesson. The other side effect of this capability is the automatic lesson generation from a knowledge base. *If we already have a lesson built for one knowledge base then the system can automatically generate other lessons for other knowledge bases as long as they all rely on the same abstract problem solving strategies*.

5.1.7 User Interface

The lesson construction process has two phases: lesson design and lesson generation. The lesson design targets the instructor who designs the lesson. The lesson generation is mostly for the students who learn the problem solving expertise from the tutoring system. Each of them has its own user interface.

5.1.7.1 Lesson Design User Interface

The lesson designer uses the lesson editor to design the lesson. The lesson editor has two panels, as illustrated in Figure 53. The left-hand side panel displays a part of the abstract reasoning tree whose root is the abstract problem associated with the lesson to be designed. The right-hand side panel is the panel where the designer places the lesson components and manipulates them. On its right margin is the widget toolbar with several widgets to build the lesson components.

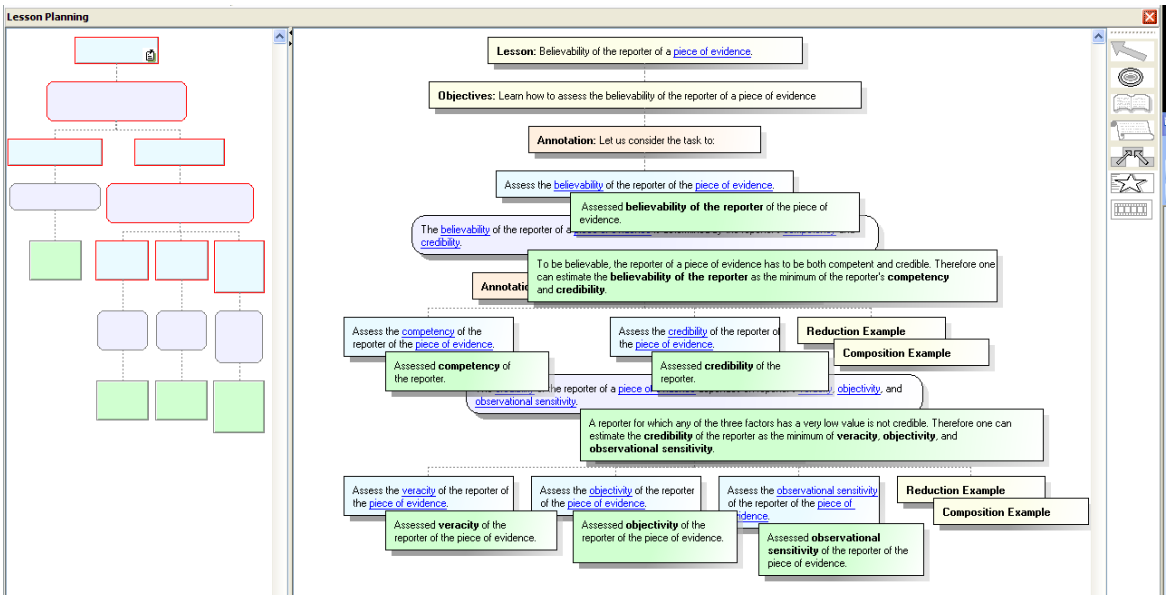


Figure 53: The Interface of the Lesson Editor

When the lesson editor is invoked, the right hand-side panel always has the lesson title, the lesson objective and the lesson problem to be presented in the lesson. The left-hand side panel contains the tree whose root already has a lesson problem created by default. The nodes which are used in the lesson components are highlighted in red, as seen in Figure 53. The designer can drag an abstract reduction to create a lesson section. Each lesson section contains a reduction example node and a synthesis example node. These two example nodes are just placeholders. They will be automatically generated later when the lesson is generated.

Figure 77 shows the widget toolbar. The objective button creates a lesson objective. This type of node has a constraint: there is only one lesson objective in a lesson. Therefore if the objective exists then selecting that button will not yield another lesson

objective. The definition button creates a definition for a specific lesson component (each lesson component which is not a decorative may include one definition). The definition token is editable. It allows the lesson designer to select one or several terms to be defined. The terms to be defined are generated based on the content of the lesson component which the definition is for.

Figure 54 shows the interface of the definition editor. In this editor there are two terms to be defined, piece of evidence and credibility. By default, all are selected, but the designer can change this by un-checking some terms and saving the change. Only the checked terms are presented in the lesson.

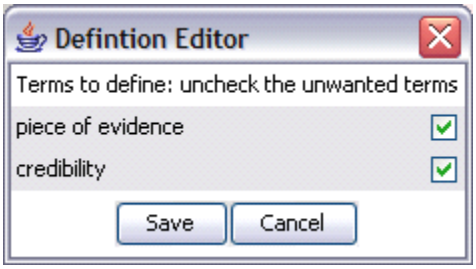


Figure 54: The Interface of the Definition Editor

The next button is the annotation button, a decorative component for the creation of the lesson's annotations. This type of component is to clarify or introduce some phrases before another component. All the decorative components have an option to turn off the voice when being generated. The synthesis button generates the lesson's synthesis component for any lesson section components. Each lesson component in the lesson editor is generated by default. One may right-click on each component to modify the content and the text to be displayed in the table of contents of the tutoring system.

The order setting button allows the designer to specify when components should be displayed and for how long, as order-duration pairs associated with each lesson component, as illustrated in Figure 55. In this figure, the lesson's title and objective are displayed at the same time and at first. The lesson title lasts until the end of the lesson, whereas the lesson objective stays only one step due to its duration value being 1. The next component to be displayed is the lesson's problem which lasts until the end of the lesson because its duration is -1. The lesson's reduction appears next, then the lesson's sub-problem. The synthesis example is displayed last.

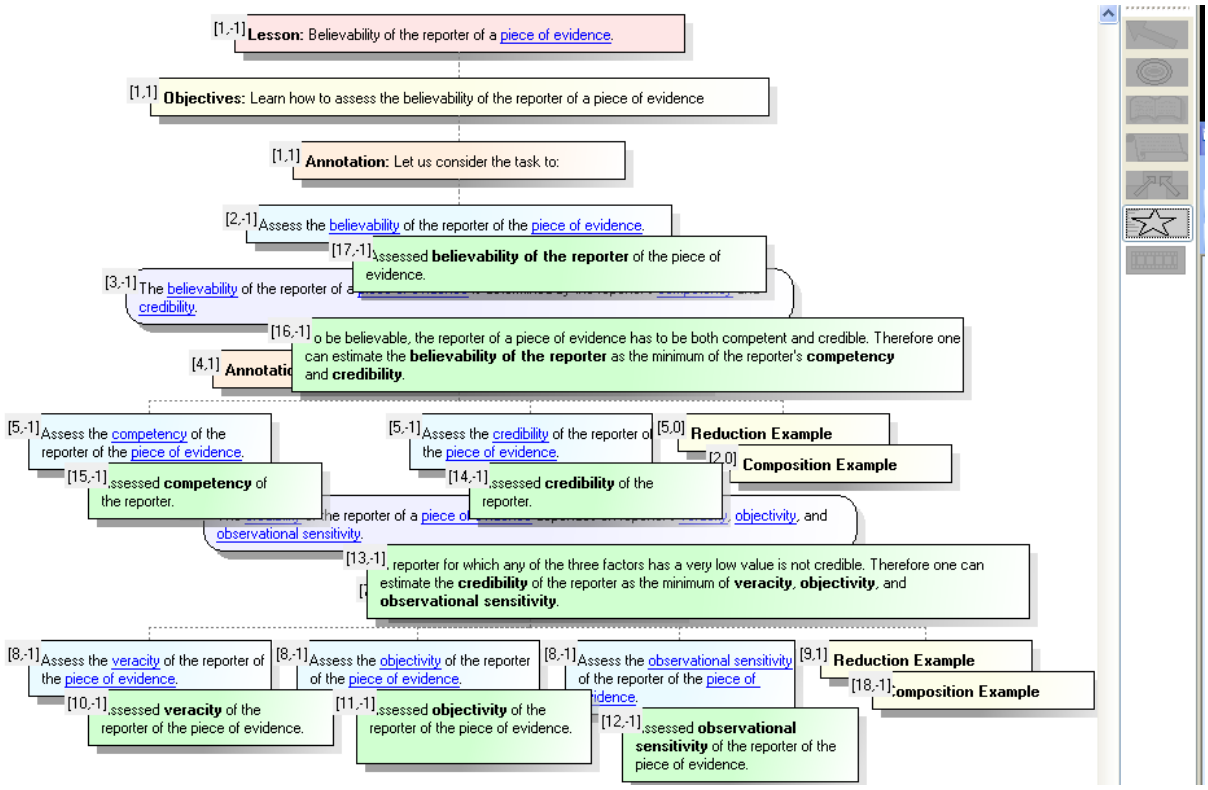


Figure 55: The Interface of the Order Setting Module

This feature is very important in the lesson design process, allowing the customization of the lessons. Each tutoring strategy is different based on the student's knowledge, the domain, the content of the lesson and the designer's teaching style.

The last button in the tool bar is the preview button which displays the lesson's components based on their order and duration. The lesson designer can stop the automatic display to navigate back and forth at his/her own pace. This preview panel can visualize the lesson in the tutoring system, allowing the lesson designer to see the current status of the lesson based on its settings. The designer can go back to the setting order mode to modify the configuration and the order and then preview again to view the effect of the new changes. Figure 56 shows the preview panel.

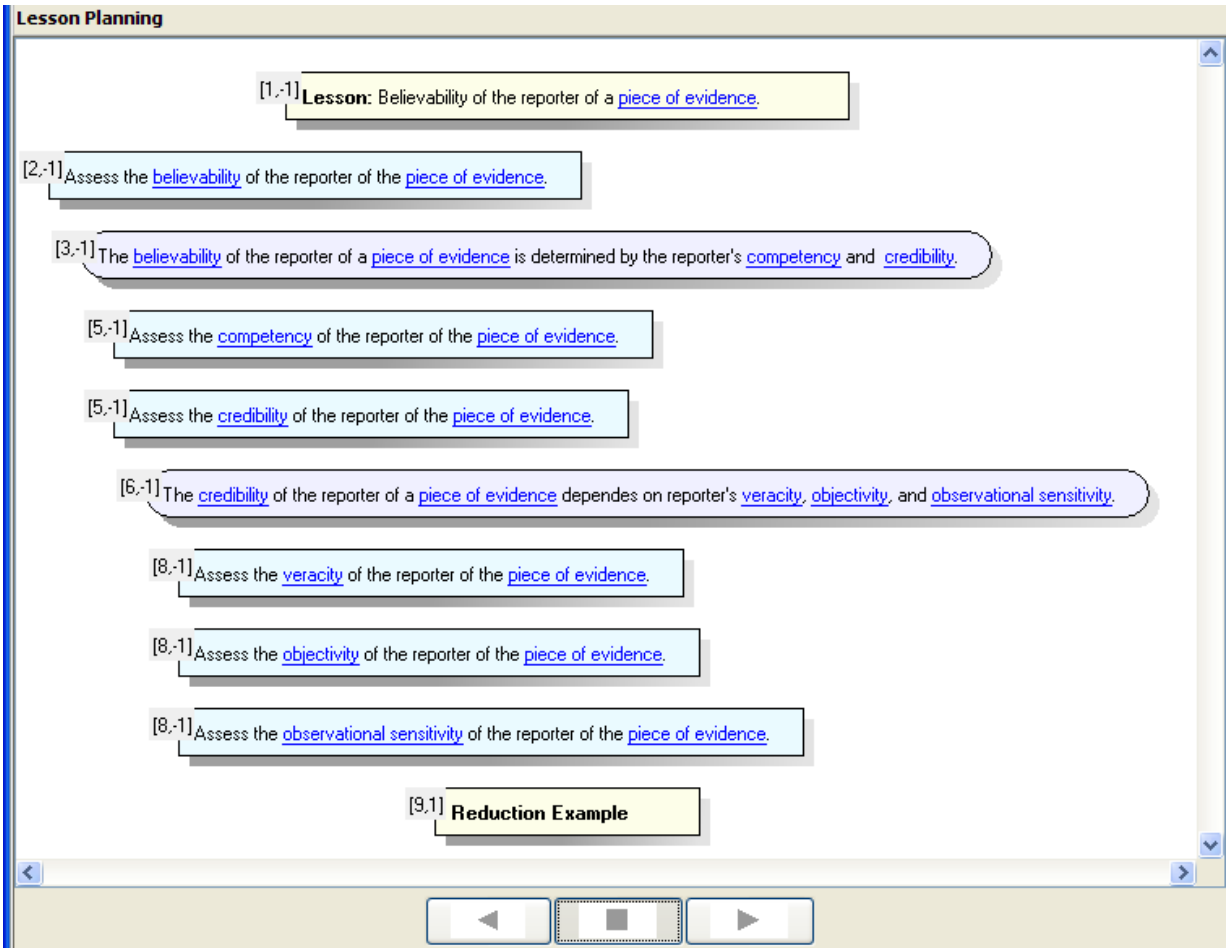


Figure 56: Preview of a Designed Lesson

5.1.7.2 Lesson Generation User Interface

The lesson generation user interface is for the students who take the lessons. The lesson has three components, table of contents, lesson content and lesson text, each with its panel. The table of contents panel contains three sub-panels: previous lessons, current lesson and next lessons (see Figure 57). The previous lessons panel displays all the previously presented lessons. Their tables of contents are accessible for a quick review. The table of contents of the current lesson is displayed fully. The next lessons panel does

not allow the view of the tables of contents. Once the current lesson is finished, it is moved up to the previous lessons panel and the next lesson in the next lessons panel is moved to the current lesson panel (if there is a next lesson). The next and previous lessons link to the current lesson via the post-requisites and pre-requisites of the lessons.

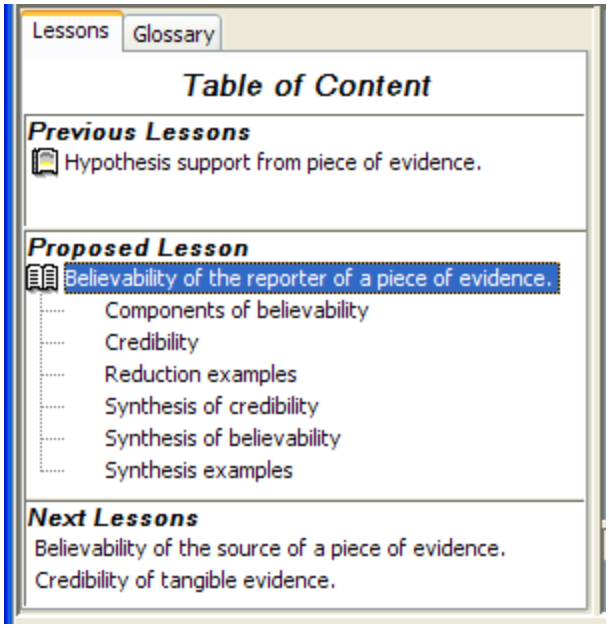


Figure 57: Lesson's Table of Contents Panel

The lesson content panel contains two sub-panels, the abstract panel and the example panel, as illustrated in Figure 58. The lesson example panel is minimized during the lesson display until there is an example to show. The abstract panel displays the abstract problem solving strategy being taught. It follows the order setting to present the lesson components. The lesson's example component is displayed in the lower panel. The student can browse the available generated examples by clicking on the navigational

labels “Next” and ”Previous,” or by selecting a certain example from the “Select Example” combo box. The student controls the display of the lesson with the navigation buttons at the bottom panel: the next button will advance one step, the previous button will go back one step and the stop button will stop whatever is currently displayed.

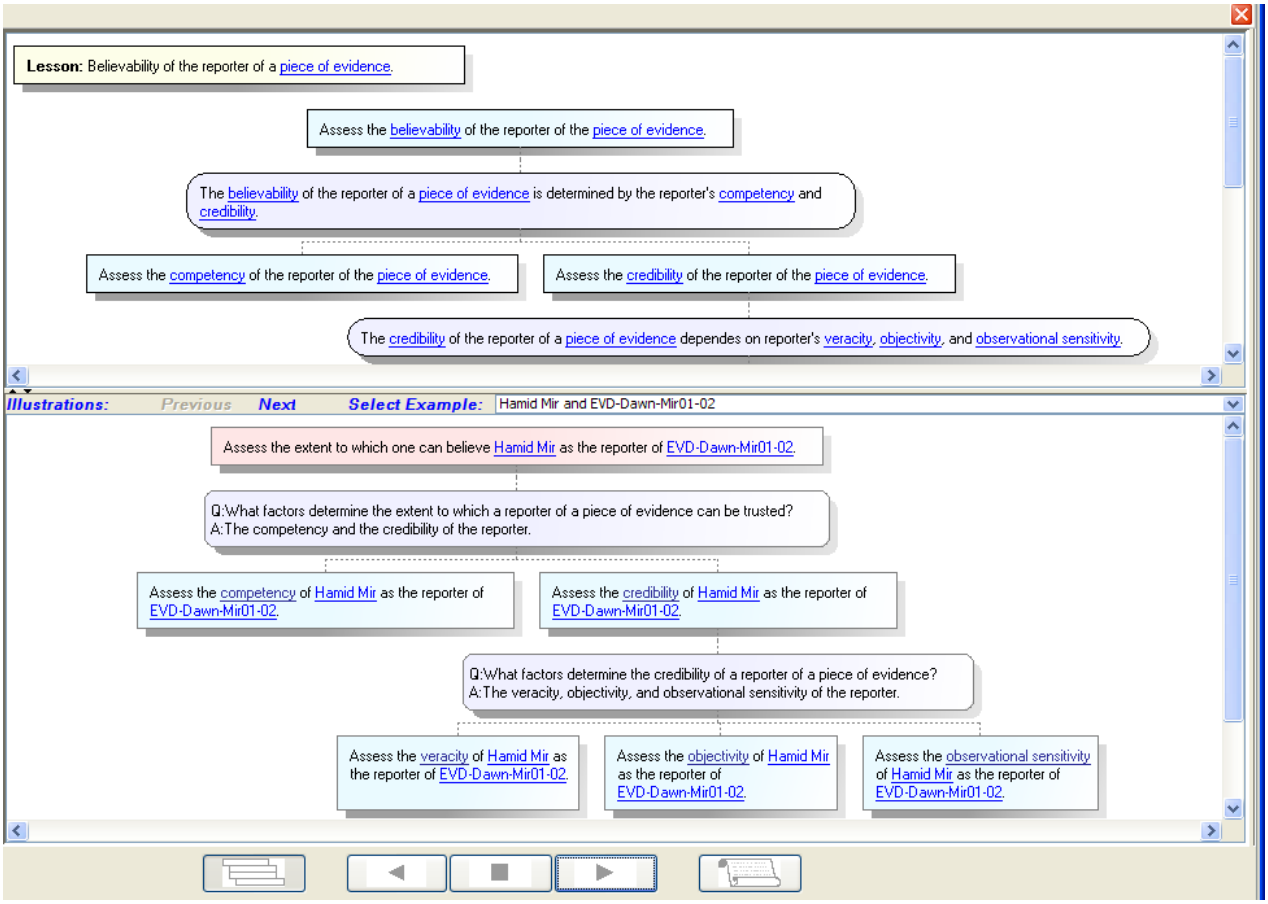


Figure 58: Sample Lesson Content

Lesson’s text is generated automatically based on the content of the lesson. Each lesson component will produce a text version of its content. The collection of all lesson component texts forms the text version of the current lesson. The text of the current

component is highlighted blue. By default the audio is turned on but the student can turn off that option. Figure 51 shows a part of the text of the current lesson.

5.1.8 Evaluation of Lesson Generation

The research is implemented as an extension of the Disciple agent development environment. The Disciple learning agent shell uses a multi-strategy approach for developing intelligent agents where an expert can teach the agent how to solve domain-specific problems. Disciple has proved to be successful in developing learning agents that can learn as apprentices. Such agents can use their learning capability to learn how to generate lessons and exercises.

Disciple provides the basic framework to develop the tutoring systems. Disciple has a workspace manager who manages and provides the public interfaces to integrate its components altogether. The abstraction-based tutoring systems which are built with the Disciple learning agent shell take advantage of that facility to ease the process of developing their necessary components which work together with the Disciple components. Disciple also provides the infrastructure for the tutoring systems, such as the knowledge base module and the learning module.

As mentioned earlier, the new approach speeds up the process of building the tutoring systems partly due to the rapid knowledge acquisition capability that Disciple has. This capability not only reduces the time it takes to acquire the domain knowledge, but it is also used by the tutoring system to simplify the acquisition of pedagogical knowledge. Therefore Disciple is an essential component in achieving rapid development of a

tutoring system. The domain that was used in the experimentation for our work is *Intelligence Analysis*.

In Spring 2006 we had an opportunity to evaluate the tutoring system with the students of the course "*Military Application of Artificial Intelligence"* (MAAI-2006) at US Army War College (USAWC). The students were either experienced intelligence analysts or users of intelligence. We have repeated this evaluation with the students in the GMU course "CS 681 Designing Expert Systems." As opposed to the Army War College students, none of the GMU students had significant prior knowledge of intelligence analysis.

After using the tutoring system, the students evaluated various aspects of it by expressing their disagreement or agreement with certain statements, on a five point scale (strongly disagree, disagree, neutral, agree, and strongly agree). Figure shows a sample of these subjective evaluation results. 7 of the 12 USAWC students agreed that the tutoring system helped them to learn the addressed topic and 11 of them agreed that the examples facilitate the understanding of the presented topic, as shown in the left-hand side of Figure 59. The right-hand side of Figure 59 shows the evaluation of the same aspects by the GMU students. All 15 students agreed that the tutoring system helps to learn the addressed topic. Also, 14 of the 15 GMU students agreed or strongly agreed that the examples facilitate the understanding of the presented topic.

In this evaluation, we can see that background knowledge plays an important role in the perceived usefulness of a tutoring system. The GMU students were not familiar with the domain at all, while the USAWC students were very familiar. Therefore, the tutoring

system therefore seemed more valuable to the GMU students than to the USAWC students. It is however very encouraging that even the USAWC students considered the tutoring system useful.

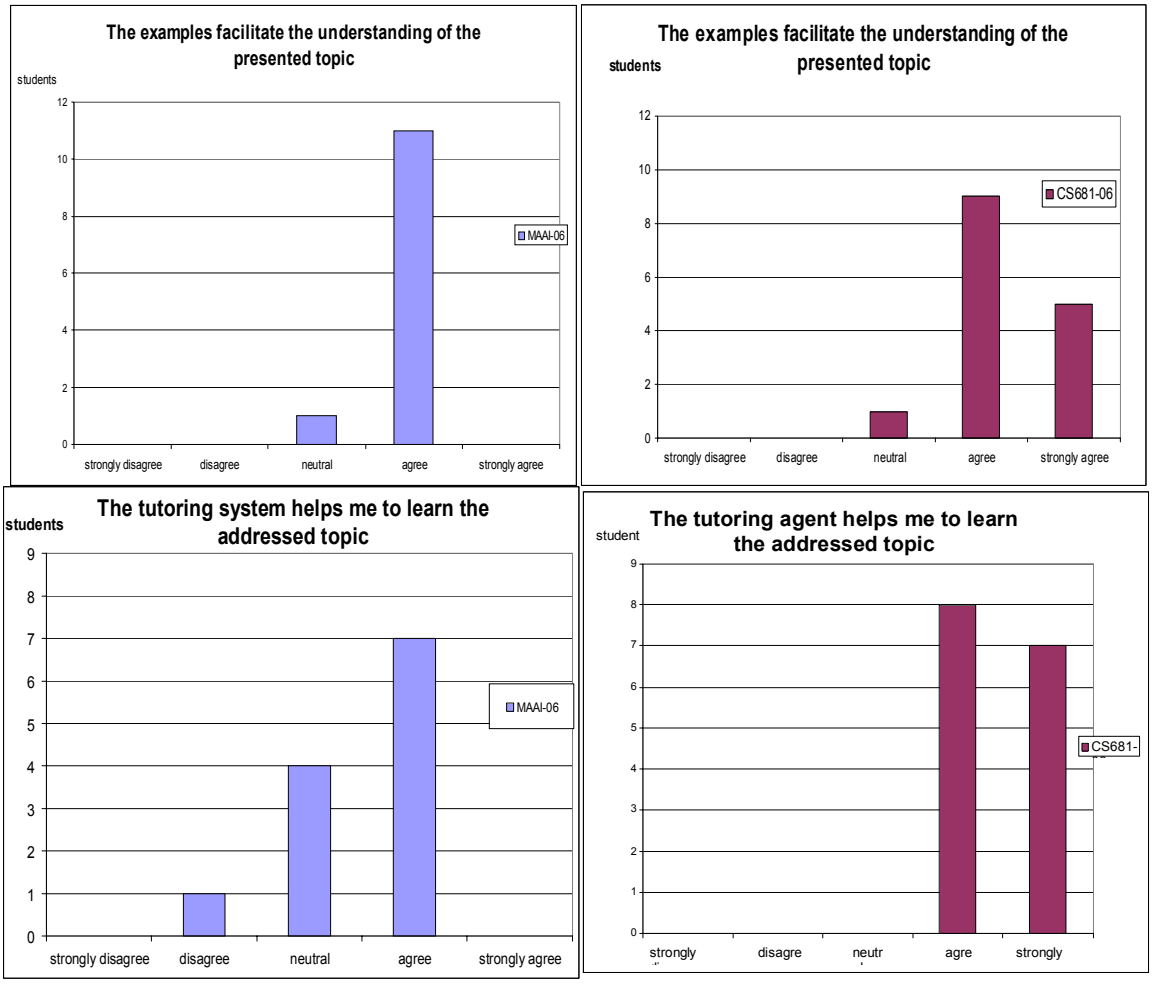


Figure 59: Evaluation of Generated Lessons

Figure 60 presents a different type of evaluation performed with the GMU students, which is based on the Kirkpatrick test model (Kirkpatrick, 1998). We have surveyed the

students, both before and after they have used the tutoring system, on how much knowledge they thought they had about specific intelligence analysis topics tutored by the system. In addition, at the end of the class, the students were tested to objectively evaluate their learned knowledge. The first five charts of Figure 60 compares the students' perception of their intelligence analysis knowledge (on several basic topics) before using the tutoring system (in blue), and after using the system (in red). The charts show clearly a very significant improvement in the tutored topics: hypothesis assessment, information content and credibility, credibility of reported evidence, credibility of the reporter, and credibility of tangible evidence.

The last figure of Figure 60 presents the objective evaluation of CS 681 students. At the end of the class, the students took the tests generated by the test agent (see Section 5.2). The tests focus on the understanding of the Intelligence Analysis domain. The agent graded the students based on the correct answers. The lowest score was 71, and the highest was 100. Out of 15 students, six scored from 70 to 79, three scored from 80 to 89 and six scored over 90. According to top charts of Figure 60, there were some students who did not know any thing about this domain, and some how could score at least 70 points. Therefore this evaluation suggests that our experimental tutoring system is a valuable tool to enhance a student's knowledge.

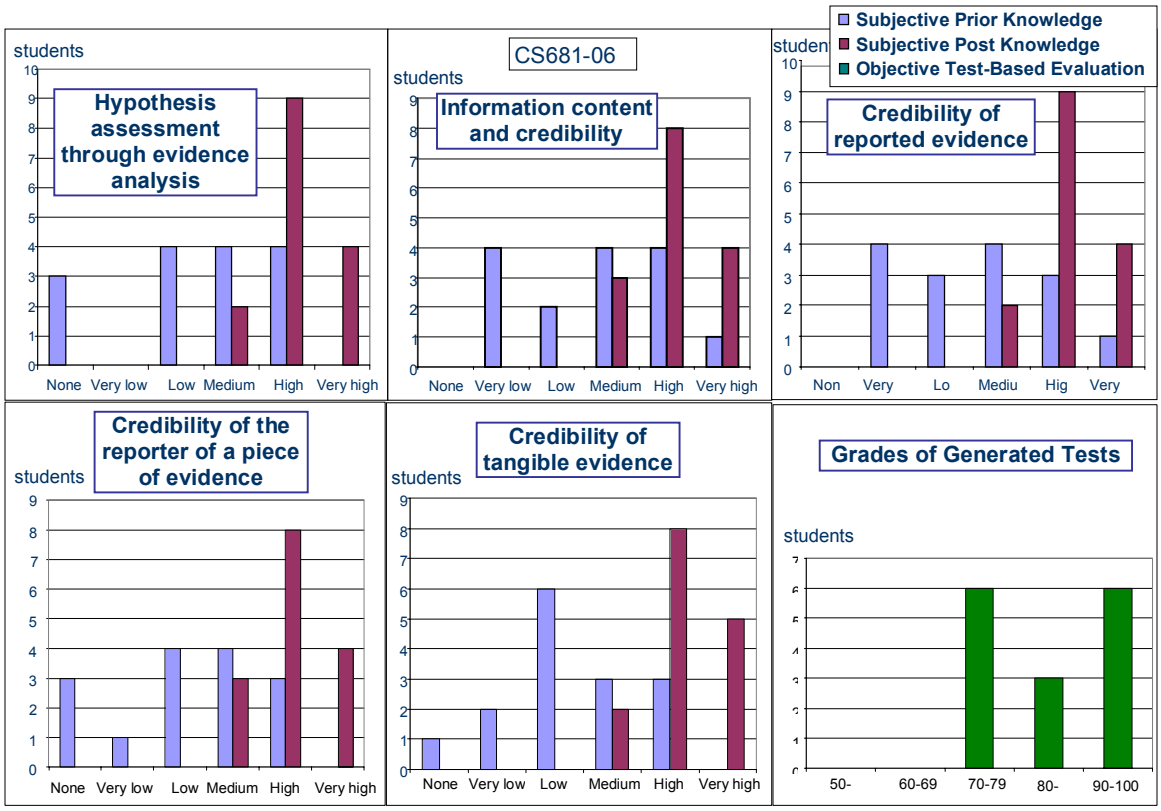


Figure 60: Evaluation of Tutoring

5.2. Learning and Generation of Test Questions

In general, the test questions are categorized into six levels of cognition, known as the Bloom's Taxonomy (Bloom, 1956). They correspond to different levels of understanding, as explained below with examples from the problem reduction/solution synthesis paradigm.

- *Knowledge level*: the ability to recall data or information such as a problem reduction rule.
- *Comprehension level*: the ability to understand the meaning of instructions or problems, for instance, to recognize an error in the reduction of a problem.

- *Application level*: the ability to apply a concept to a new situation, for example, to apply a learned reduction strategy to solve a new problem.
- *Analysis level*: the ability to distinguish between facts and inferences and to decompose the material into components, such as being able to reconstruct the reduction step that is applicable to a certain problem.
- *Synthesis level*: the ability to combine components into a whole, for example, to synthesize a final solution of a problem from elementary solutions.
- *Evaluation level*: the ability to make judgments about the values of ideas or materials, such as being able to judge if some new reduction steps are logically sound.

The tests can be developed to measure the level of a student's understanding, based on the Bloom Taxonomy. In this dissertation we focus only on some of the levels, such as, knowledge, comprehension and analysis.

5.2.1 Learning of Test Questions

We have developed learning methods that allow an instructional designer to teach an agent how to construct test questions. Our methods are based on the problem reduction rules that have been previously learned by the agent. They consist in extending these rules with additional components, to transform them into test questions rules. The rules are then applied in appropriate settings to generate specific test questions. The designer selects an example of a problem reduction rule and transforms it into a test for the knowledge, comprehension or analysis level, as discussed below.

To test the knowledge level, the designer drops one or several sub-problems in a reasoning step to produce a deliberately wrong reasoning step. Figure 61 shows one of the examples for such *omission test*. In this example, the first sub-problem of assessing the degree to which a piece of evidence favors a hypothesis was dropped. The reasoning step becomes incomplete and that would alert the student who learned it by heart and encounters it during the testing period. During the testing period, the question and answer is not shown, to make the test more difficult.

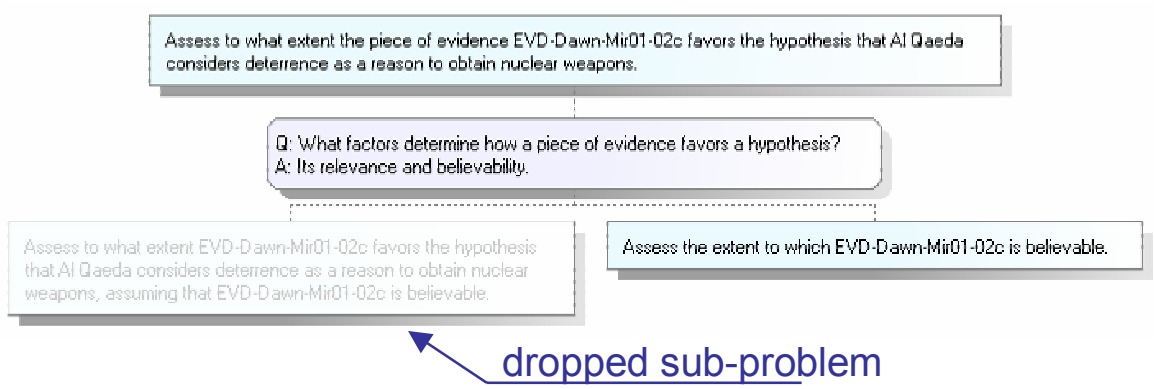


Figure 61: Test Example for Knowledge Level

Figure 62 shows a modified reasoning step where the instructional designer deliberately altered the meaning of one of the sub-problems. In particular, the assessment of the believability was replaced with the assessment of the authenticity. This type of test question which is named *modification test* requires the students to have deeper knowledge about the subject compared to the knowledge level tests.

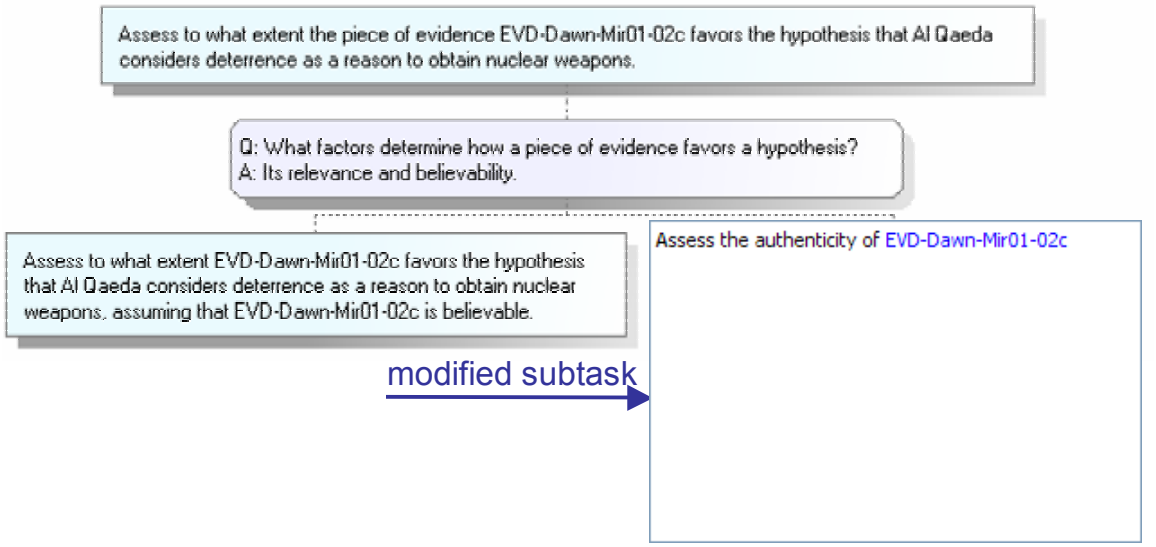


Figure 62: Test Example for Comprehension Level

Another type of test question that is more challenging than the above two is the *construction test*. The designer defines several sub-problems which may be unrelated or incorrectly related to the correct sub-problems of a problem. The test question will present a problem and a list of potential sub-problems, including the correct and the incorrect ones. The student must select the correct sub-problems. This type of test requires the student to analyze the sub-problems to build up a correct reasoning step. Figure 63 illustrates the design of a construction test. It shows the extra deliberately "wrong" sub-problems: assessing the availability, the accuracy and the relevancy of a piece of evidence. Those three together with the original two sub-problems will make a pool of sub-problems to select from.

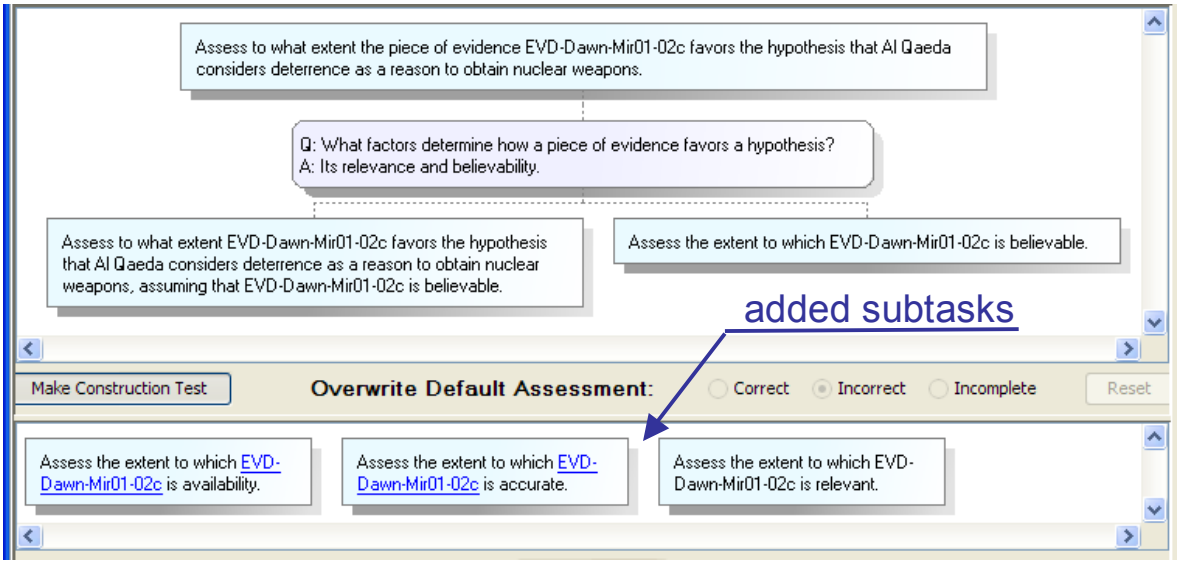


Figure 63: Test Example for Analysis Level

No matter what type of test question the designer plans to build, a set of explanations and a hint must be constructed in parallel with the content of the test. Figure 64 shows a panel where the explanations are created by the designer. There are three explanations for three types of the answers: correct, incorrect, and incomplete. The explanations are displayed once the answer is given. The hint, on the contrary, is given before answering the test question and by request only. Notice that the explanations and the hint correspond to the particular test example being built. That is, they are very specific, containing the instances (such as EVD-Dawn01-02c) from the example. The example, however, corresponds itself to a previously learned rule. This rule will be extended with generalizations of the explanations and the hint, obtained by replacing the contained instances with the corresponding rule variables, as discussed in the following.

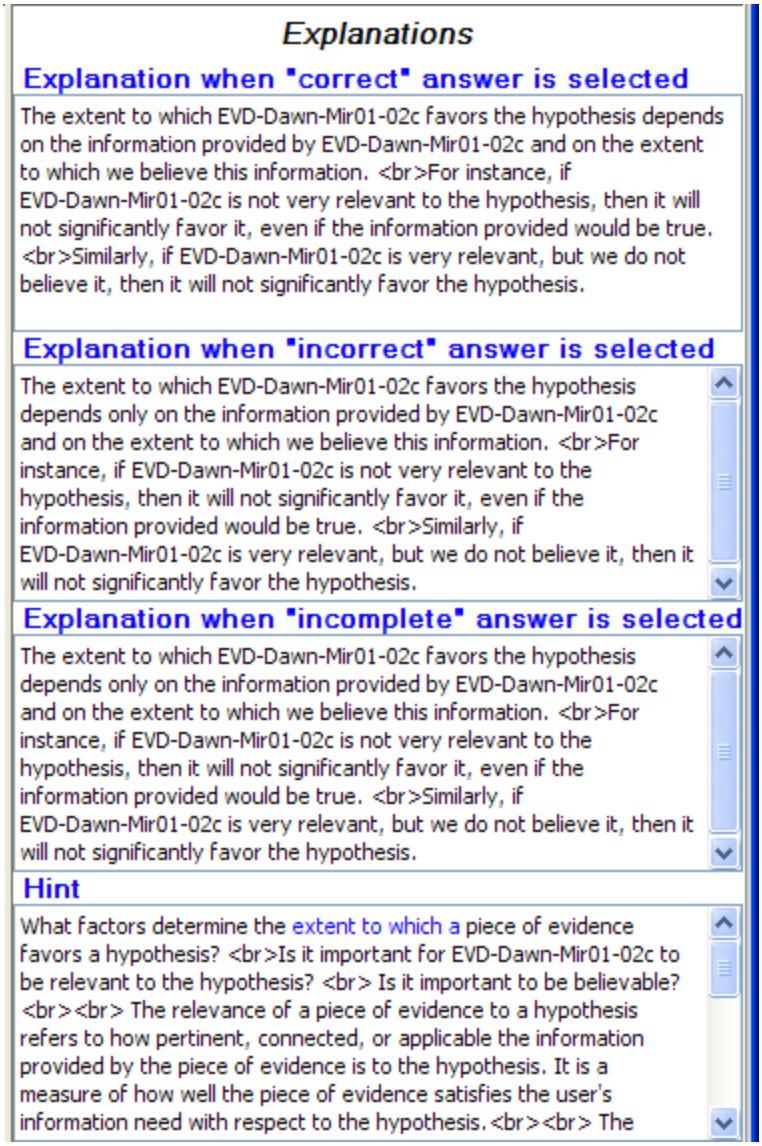


Figure 64: Explanations Construction

Once an example of the test question is provided, the task now is to learn how to generate similar tests in future. Learning by test examples is processed in a sequence of steps:

- Receive a reduction rule to learn a test rule based on it. When the instructional designer plans to create a test example, s/he usually goes through a list of available reduction rules and picks out the desired one. The reduction rule corresponding to the above examples is shown in Figure 65.
-

DECOMPOSITION RULE DDR.00013 FORMAL DESCRIPTION

IF:	Assess to what extent the piece of evidence *?O1* favors the hypothesis that *?O2* considers deterrence as a reason to obtain nuclear weapons.

Q:	What factors determine how a piece of evidence favors a hypothesis?
A:	Its relevance and believability.

MAIN CONDITION

Var	Lower Bound	Upper Bound
?O1	(elementary piece of evidence)	(piece of evidence)
?O2	(terrorist group)	(actor)

THEN:	Assess to what extent *?O1* favors the hypothesis that *?O2* considers deterrence as a reason to obtain nuclear weapons, assuming that *?O1* is believable.
	Assess the extent to which *?O1* is believable.

Figure 65: A Reduction Rule

- Construct a test rule based on the reduction rule and the modifications and extensions of one of its examples. A test rule basically contains a reference to the reduction rule and a list of extensions. The extensions include the test category (i.e. omission, modification, or construction), the category-related information and the generalizations of the explanations and hint. For the omission test, the

related information is the reference to the dropped sub-problems. For the modification test, it is the old and the new contents of the modified sub-problems. For the construction test, it is the extra sub-problems that were entered during the construction of the test question example. The explanations and hint are the same for all types of test. They are generalized to be applicable to different scenarios.

5.2.2 Generation of Test Questions

With a set of test rules available in the tutoring knowledge base, the agent can generate numerous test questions to present to the students who already took the related lessons. Indeed, each test question is based on a reduction rule, and for each instance of the rule in a knowledge base, there is a corresponding test question. Consequently, a lot of different test questions can be generated from a single test rule if the domain knowledge is rich. The Table 12 presents the algorithm for generating test questions.

Table 12: Algorithm of Test Question Generation

Given:

- TRs - *set of test rules*
- RT - *a reasoning tree*

Return:

- GTQs - *list of generated test questions*

TestQuestionGeneration (TRs, RT**)**

```
1.  for each test rule TR ∈ TRs do
2.      RdR ← retrieve reduction rule from TR
3.      List of instantiations of reduction rule IRdRs ← SearchInstantiation(RT, RdR)
4.      for each IRdR ∈ IRdRs do
5.          if TR is omission test then
6.              GTQ ← drop the sub-problem node(s) of IRdR specified in TR
7.          else if TR is modification test then
8.              GTQ ← modify the content of the sub-problem node(s) of IRdR specified
                  in TR
```

```
9.          else if TR  is construction test then
10.             GTQ ← create a pool of sub-problem nodes from the sub-problem node(s)
                    of IRdR plus added sub-problem node(s) specified in TR
11.         end if
12.         add GTQ to GTQs
13.     end for
14. end for
15. return GTQs
end TestQuestionGeneration
```

From a list of generated test questions, a sort procedure is initiated based on sorting criteria: random distribution or an ordering of the test questions in context. The random distribution generates the test questions in the random order each time the test agent starts. That makes the tests more versatile and interesting: the student cannot tell what test will be next. No test session will be the same for all students, even for the same student. For the ordering of the test questions in context, the tests are arranged in such a way that they are presented from the top down to the bottom of the reasoning tree. This type of distribution helps student to recall the learned knowledge by following the context.

The two types of distribution are suitable for two types of test mode: self-test and assessment test. In the self-test mode, the students are tested to reinforce their learned knowledge rather than to assess of their knowledge. The students are able to go back to the lesson corresponding to the test, via the "*Go To Lesson"* option, to review the lesson. In the assessment mode, the students do not have access to the lessons. In other words, they do not have the "cheat sheets" with them.

Figure 66 shows a generated test question which is based on the test rule learned from the test example in Figure 62. The test question displays an incorrect reasoning in which one sub-problem is modified. Note that the question and answer are omitted. The student will have to indicate whether the reasoning step displayed is correct, incomplete or incorrect. Each selection is followed by a context-sensitive explanation. A hint is always available to help the student in case s/he needs. Glossary is also provided for clarification of various terms. Once the answer is chosen, the system will grade it and report back both this grade and the cumulative grade (which corresponds to all the test questions answered).

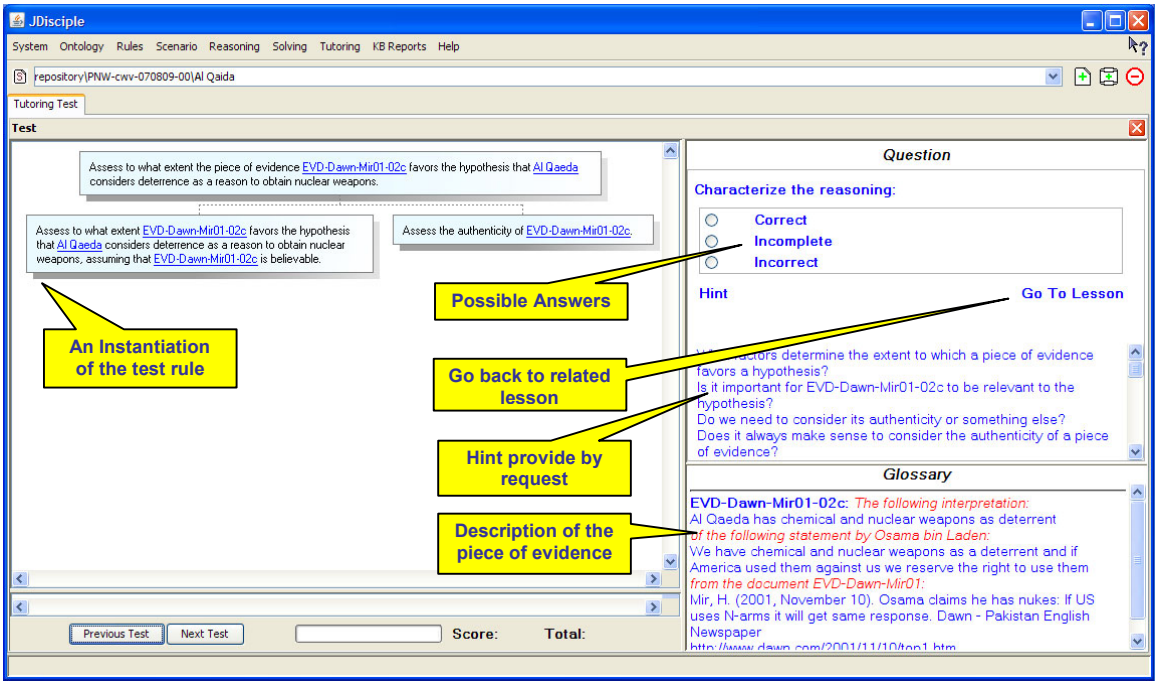


Figure 66: A Generated Test Question

Figure 67 illustrates a construction test question where the student has to select the correct sub-problems (shown at the bottom left of Figure 67) of a given problem (shown at the top left of Figure 67). Such selection can evaluate the student's understanding of the subject. Therefore the grading for this type of test is strict: only selecting all the correct sub-problems is considered as correct answer, otherwise it is either incorrect if one or more incorrect sub-problems are chosen, or incomplete if not enough correct ones are selected.

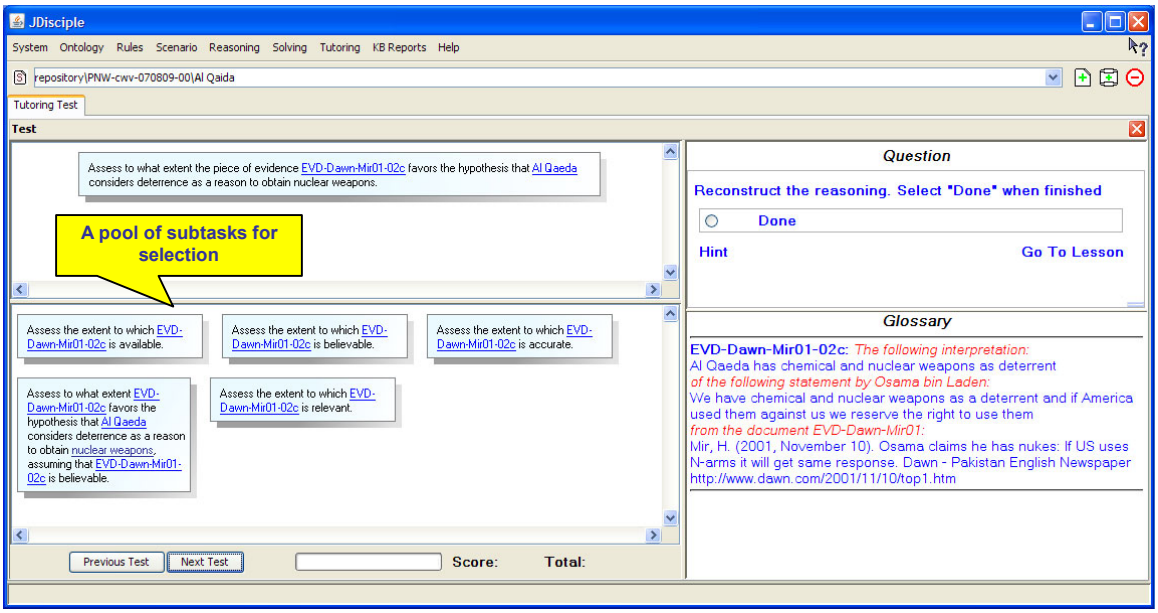


Figure 67: A Generated Construction Test Question

5.2.3 Complexity Analysis

The test generation algorithm is in the one from Table 12. In this algorithm, each operation on the instantiated sub-problems to create a GTQ costs a constant *O(1)* (line 6, line 8 and line 10). Therefore the inner FOR loop on line 4 which enumerates a list of N_{ir}

reduction instantiation IRdR $\in$ IRdRs is $N_{ir}O(1) = O(N_{ir})$. The outer loop (line 1) depends on N_{tr} the number of test rules TRs. In other words, the entire outer loop costs $O(N_{tr} \times N_{ir}) = O(N_{gt})$ where N_{gt} is number of generated test questions. Besides, the search of all instantiations of a reduction R_i in reasoning tree $RT = (V, \delta_t)$ costs $O(N_v + N_\delta) = O(N_v + N_v - 1) = O(N_v)$ where N_v is number of nodes in reasoning tree and N_δ number of edges that connect all the nodes together. After all, the complexity of the test question generation based on the algorithm presented in Table 12 is

$$\boldsymbol{O(N_{gt} + N_v).}$$

5.2.4 Evaluation of Test Generation

Two versions of the test generation agent were tested by students at the US Army War College in Spring 2006, and students at George Mason University in Fall 2006. Figure 68 shows a sample of the subjective evaluations by these students. The assessment of "*The exercises are challenging"* is important because it suggests the value of the test questions. In Spring 2006, only 7 of the 12 students agreed that the exercises were challenging. The evaluation result was better in Fall 2007 where 1 out of 12 students strongly agreed and 10 students agreed that the test questions were challenging.

The agreement or disagreement with the statement "*The exercises improve the understanding of the presented topics"* assessed the overall usefulness of the tests. In Spring 2006, 7 out of the 12 students agreed and 5 were neutral. The result was better in Fall 2006 where 2 out of 15 students strongly agreed and 8 agreed with the above statement. Overall, the novice analysts gave better assessment than the expert analysts did. That was expected because the experts were very familiar with the domain.

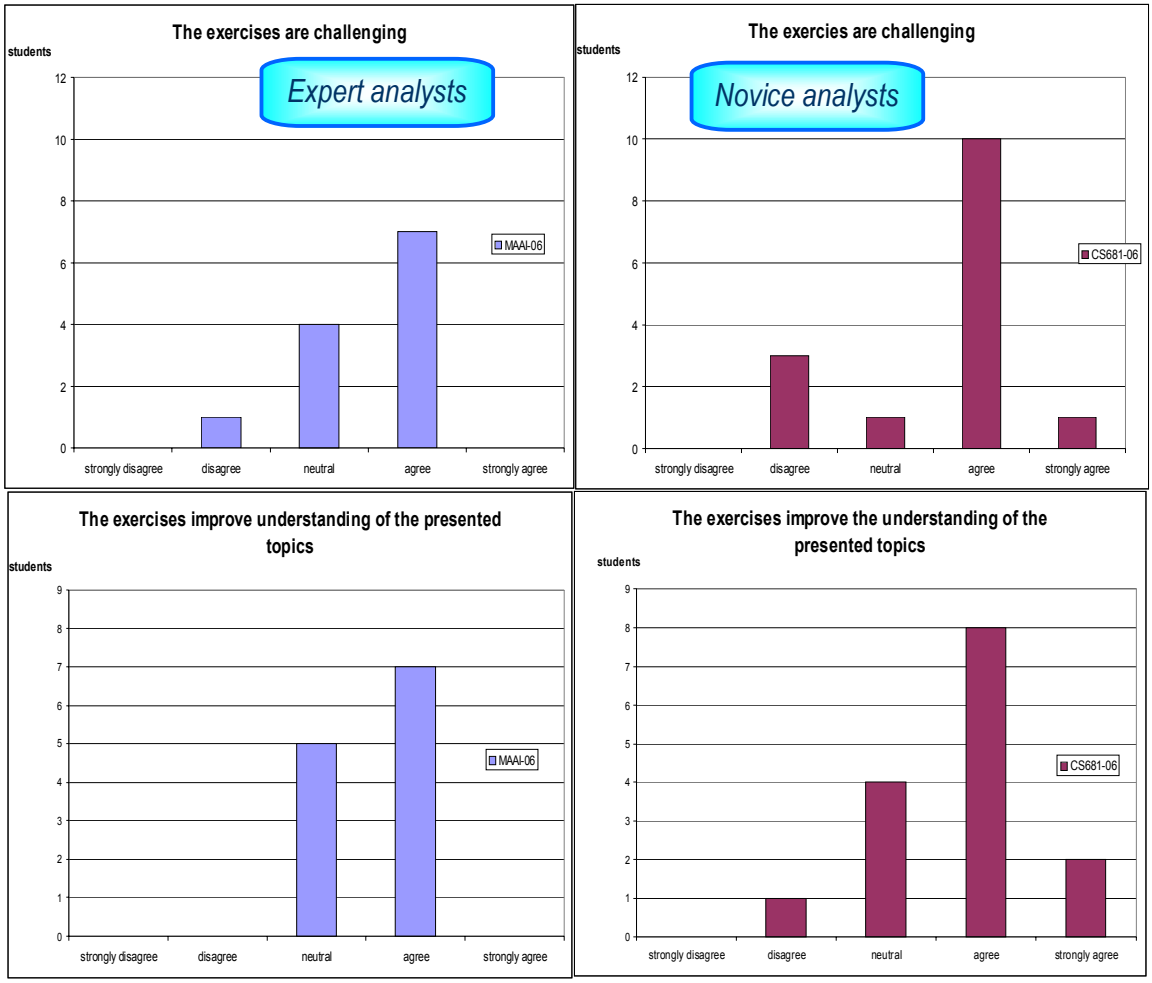


Figure 68: Evaluation of the Test Agent

6. Learning and Tutoring Agent Shell (LTAS)

6.1. From Expert System Shells to Learning and Tutoring Agent Shells

Since the first expert systems were developed (during 1970s) and commercially used (during 1980s), the idea of constructing a generic shell that can facilitate the process of building expert systems came up as a natural way of evolving the methodology of developing these systems. This is because the cost of building an expert system is very high and often unaffordable. Moreover, the time it takes to build a useful expert system is very long and the dynamics of some domains will require frequent knowledge maintenance. As discussed in Section 1.2, the expert system shell simplifies the process of constructing an expert system. The main principle of the shell is re-usability of the inference engine and the associated tools such as editors, knowledge base checkers, etc (Whitley, 1990). An expert system shell may be regarded as an expert system with an empty domain knowledge base that has a pre-defined knowledge representation. Now the problem of building an expert system reduces to building a knowledge base that can be plugged into the shell. The knowledge base must be built following the required syntax and other constraints. The expert system shell thus alleviates some burdens from the task of building an expert system and shortens the construction time (Whitley, 1990).

However, even with the help of an expert system shell, the task of building an expert system remains a very difficult one. The difficult task that still remains is building the

knowledge base. The knowledge base needs to represent the expertise of a subject matter expert which has to be encoded in such a way that a computer can understand and process. The procedure, described in Figure 2, of acquiring the knowledge from the expert and encoding it into the knowledge base is time consuming and error prone. The expert usually does not have enough computer science background to encode his/her knowledge, so the need of involving a knowledge engineer to transform the raw expert knowledge into a formal representation is necessary. However, the knowledge that is elicited from the expert is not always clear and straightforward because of the use of commonsense in communication. Unfortunately, commonsense knowledge is very hard to encode and is easily mistreated. A back-and-forth communication between the knowledge engineer and the subject matter expert needs to frequently occur to avoid mistakes. This is the well-known *knowledge acquisition bottleneck* problem as mentioned in Section 1.2 (Buchanan and Wilkins, 1993).

To alleviate the knowledge acquisition bottleneck, a learning component is integrated into an expert system shell (Tecuci, 1998). Such a system (shown in Figure 3) is called a Learning Agent Shell (LAS) and is implemented in a family of Disciple shells (Boicu, M. et al., 2002). In Disciple, the process of building the knowledge base is a mixed-initiative one between the expert and the learning agent, with limited assistance from a knowledge engineer. The top part of Figure 69 shows the traditional way to build a knowledge base in which the subject matter expert works closely with the knowledge engineer throughout the whole process. The knowledge engineer has to model the reasoning process of the subject matter expert, making explicit the way the subject matter expert solves problems.

Then the knowledge engineer develops the object ontology. He or she also needs to define general problem solving rules and to debug them (Tecuci, 1998).

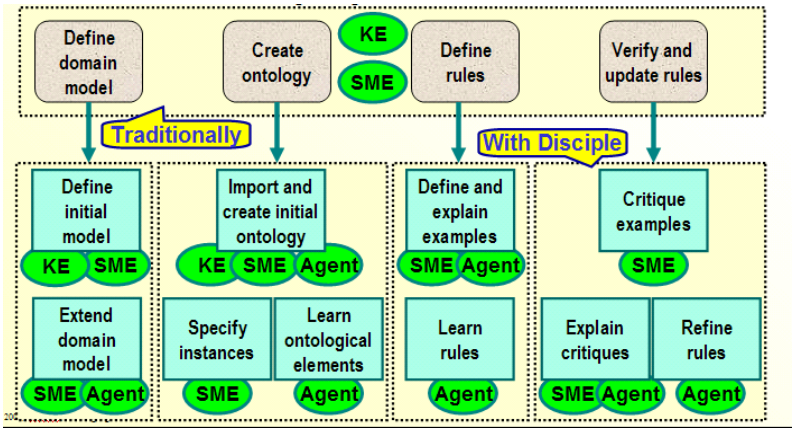


Figure 69: Knowledge Engineering with Disciple Learning Agent - from (Boicu, 2002)

With the introduction of the learning agent, the expert now works mostly with the agent and that reduces a lot of errors, uncertainties and processing time. As shown in Figure 69, each activity from the top part is replaced with an equivalent activity that is either entirely performed by the subject matter expert (SME) and the agent (Agent), or requires some assistance from the knowledge engineer (KE). The knowledge engineer needs to model the reasoning process of the subject matter expert and to instruct the expert how to make explicit his/her reasoning. The knowledge engineer also needs to develop an initial object ontology. After that, however, the subject matter expert can collaborate with the agent to develop problem solving examples and their explanations, to extend the ontology, to learn problem solving rules, and to refine the rules (Tecuci, 1998).

Maintenance of the knowledge base traditionally involves the communication between the expert and the knowledge engineer to ensure the stored knowledge is always consistent and up to date. That process is changed with the introduction of a learning agent, as shown in Figure 68. The agent is now the only partner that works closely with the expert to maintain the integrity of the whole knowledge base (Boicu, C. et al., 2005).

With the evolution from the expert system shell to the learning agent shell, it seems natural to have it evolved further to broaden its applicability. One such development is adding the capability to tutor the expert knowledge which is already acquired when building the knowledge base. Being able to rapidly acquire expertise in a certain domain and to rapidly construct a curriculum to teach this knowledge pedagogically is the main goal of the Learning and Tutoring Agent Shell (LTAS) concept.

LTAS can alleviate some of the difficult problems that are encountered when building intelligent tutoring systems. They include the difficult and time-consuming acquisition of the expert's knowledge, the complexity of building a curriculum to teach the expertise pedagogically, and the challenges of customizing the lessons for different student skills in various circumstances. If the tutoring system would be easier to build, there would be available for a wider set of domains at different levels. As a consequence, such systems would have a significant positive impact on the education in schools, as well as in the continuous education of the professionals.

6.2. Architecture of the Learning and Tutoring Agent Shell

An LTAS is an extension of a Learning Agent Shell (LAS) with tutoring related capabilities, as shown in Figure 70. These additional modules include the pedagogical

knowledge base, the knowledge management module, the tutoring module, the authoring module and the student module. They are tightly integrated with the existing modules. For example, the pedagogical knowledge base couples with the learning engine to learn the teaching knowledge from the teacher. The domain knowledge base is used with the tutoring engine to provide rich and dynamic examples and exercises.

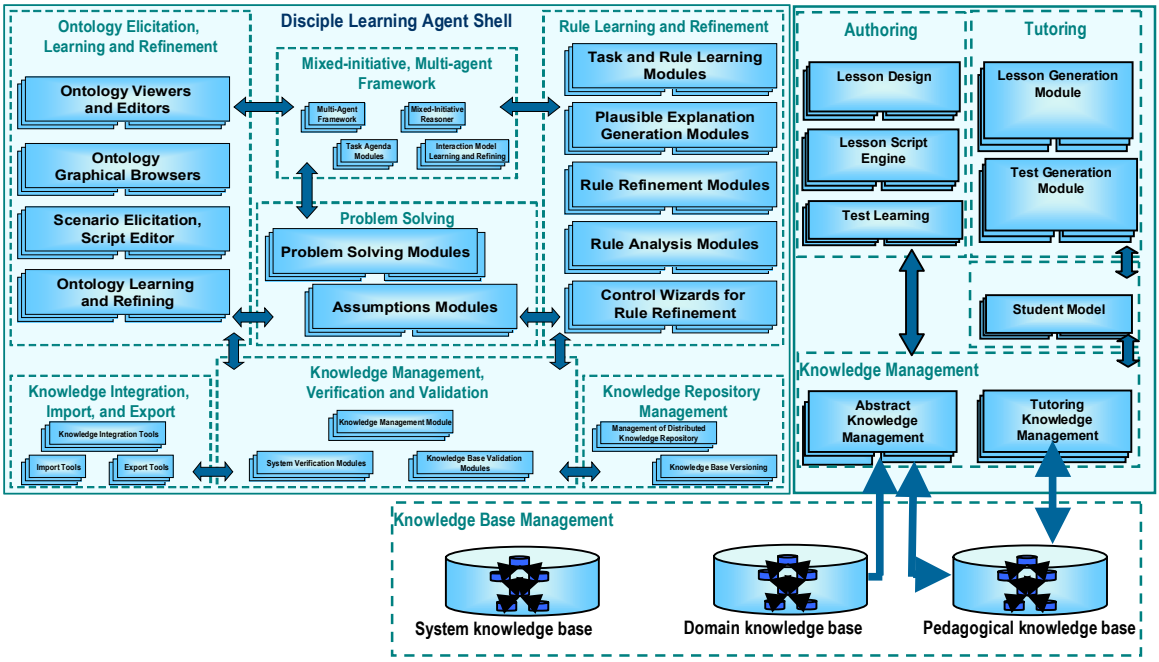


Figure 70: Architecture of the Disciple Learning and Tutoring Agent Shell

6.2.1 Pedagogical Knowledge

The pedagogical knowledge includes two types of knowledge: pre-defined knowledge (which is stored in the pedagogical knowledge base) and generated knowledge. The pre-defined knowledge is the knowledge that is created by the instructor and the system, and is used to generate the generated knowledge. The pre-defined knowledge consists of

abstraction rules, lesson scripts, and test rules. The generated knowledge include: the abstract reasoning tree, the generated lessons, table of contents, glossary, specific test questions together with explanations and hints.

6.2.1.1 Pre-defined Knowledge

Abstract Knowledge

As presented in Chapter 3, the abstraction of reasoning is constructed for several purposes, one of them being tutoring. The abstract knowledge that is preserved for tutoring purpose consists of the abstract problem solving strategies employed by the subject matter expert. The abstract problem solving strategies have several components:

- *abstract problems* that describe the kinds of problems to be solved;
- *abstract reductions* that reduce the abstract problems to one or several simpler abstract sub-problems;
- *abstract solutions* which are the solutions of the abstract problems;
- *abstract syntheses* that compose the abstract solutions of the simpler abstract sub-problems at one level into the abstract solutions of the abstract problems at the next higher level;
- *abstraction rules* that govern the abstraction operations.

Lesson Script

The lesson scripts are created by lesson script engine during the lesson design (see Section 6.2.3). The lesson scripts are represented in the ABLE scripting language (see Appendix A). A lesson script consists of a lesson header and several lesson sections. The lesson header includes the lesson title, optional lesson objectives, lesson annotation and

lesson definitions. Each lesson section presents an abstract problem solving strategy which, in essence, reduces an abstract problem to several simpler abstract sub-problems. Each lesson component has a pair of numbers that indicate the order when it will be displayed and for how long it will last during the tutoring session. More details on the lesson scripts are provided in Section 5.1.2.

Test Question Rule

A test question rule includes a reduction rule from the domain knowledge base and a list of generalized components. The components include the test type - omission, modified or construction, the type-related information, explanations and hint (see Section 5.2.1). Depending on the type of test question, the type-related information differs. For an omission test, they are the sub-problems that were dropped. For a modified test, they are the old and new contents of the modified sub-problems. And for a construction test, they are the extra sub-problems that were entered during the test question learning.

6.2.1.2 Generated Knowledge

Abstract Reasoning Tree

The abstract reasoning tree is constructed from the abstract knowledge in the pedagogical knowledge base. The abstraction rules govern how an abstract reasoning tree is built from a concrete reasoning tree. An abstract reasoning tree is a representation of the abstract problem solving strategies that are used to solve a problem. The abstract reasoning tree is described in detail in Chapter 3.

Lessons

The lessons contain two parts, the abstract problem solving strategies to be taught, and the examples that illustrate these strategies. They are illustrated in Figure 71. The top part of the figure shows the strategy to *assess a piece of evidence that favors the hypothesis*. The bottom part shows an example of *assessing the evidence EVD-Dawn-Mir01-02c that favors the hypothesis that Al Qaeda considers the deterrence as a reason to obtain the nuclear weapons*. The abstract strategies are constructed from the abstract components of an abstract reasoning tree by the instructor, as discussed in Section 5.1.1. The examples are generated from the knowledge base by using the abstraction rules that link the abstract components in an abstract strategy section to their concrete components in the concrete reasoning tree. These links allow the tutoring module to retrieve the examples corresponding to the abstract strategy which is being taught. More details of example generation are presented in Section 5.1.3. The process of lesson generation highlights the interaction between the pedagogical knowledge (the abstract problem solving strategies and their examples) and the domain knowledge (the concrete reasoning tree). Together they can produce many lessons with various examples, provided that the domain knowledge base is rich enough. Moreover, the same abstract reasoning strategies might be exemplified with scenarios in different application domain, e.g. assessing tangible evidence may be useful in counter-terrorism, law enforcement, practice of law, and even in scientific discovery. The lessons are generated automatically from the lesson scripts, as described in Section 5.1.2.

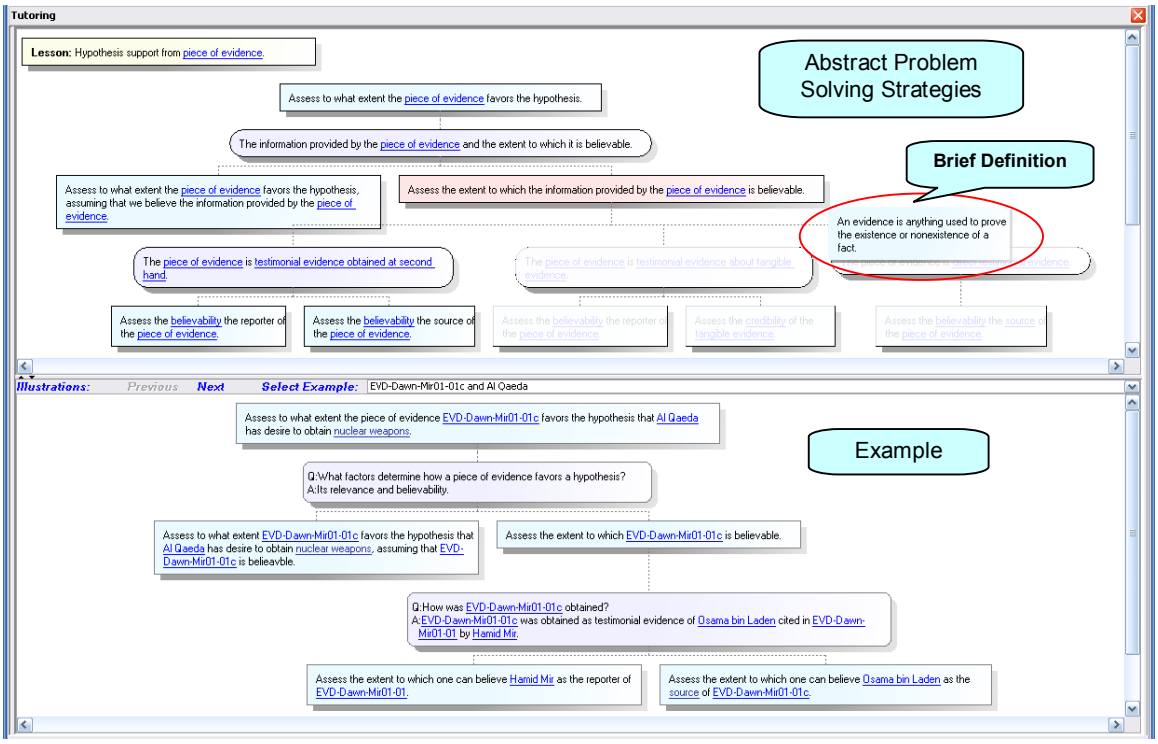


Figure 71: Lesson Interface

Table of Contents

The table of contents helps navigating the organized set of lessons. This type of knowledge is almost automatically generated by the system based on the content of lessons and the connections between them. Figure 72 illustrates a table of contents which contains three parts: the current lesson, the learned lessons and the next available lessons. Each of the lessons teaches two types of processes: problem reduction and solution synthesis, for a certain type of problem. The abstract part of the lesson is structured into several sections and is illustrated by examples at the end. The table of contents captures the structure of the lesson. For example, in Figure 72, the problem reduction process has two sections "*Components of believability*" and "*Credibility*". The reduction process is

then illustrated by "*Reduction examples*". When a lesson is in the design mode, the instructor does not have to specify the order of the lessons. The system sorts this order out based on the links between the abstract components that the lessons contains. The instructor however must explicitly define the structures of each process as described above during the design process.

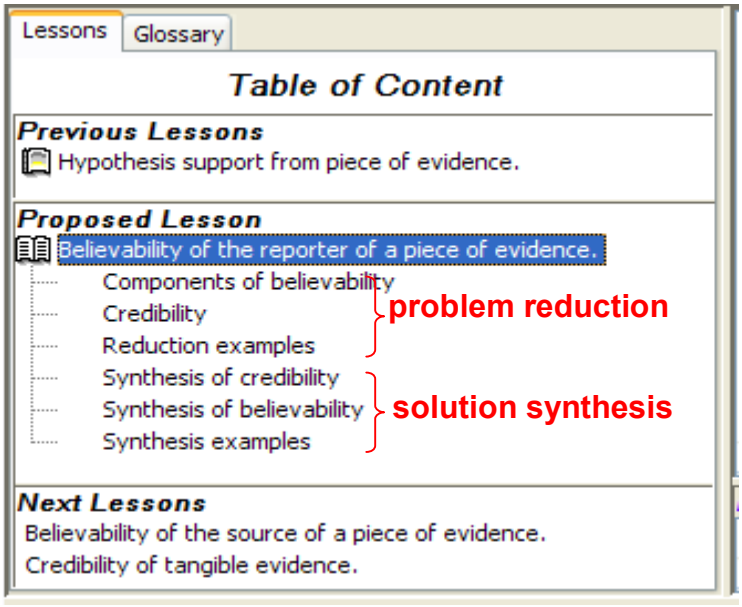


Figure 72: Table of Contents

Glossary

The glossary is generated automatically from the ontology of the system which is part of the domain knowledge base (Barbulescu et al., 2003). The glossary is displayed in alphabetical order. It provides brief definitions of the domain concepts (see Figure 71), more complete definitions, or even detailed lessons (as illustrated in Figure 73 and Figure

74). In essence, the glossary supplies a means of enhancing the understanding of the lessons.

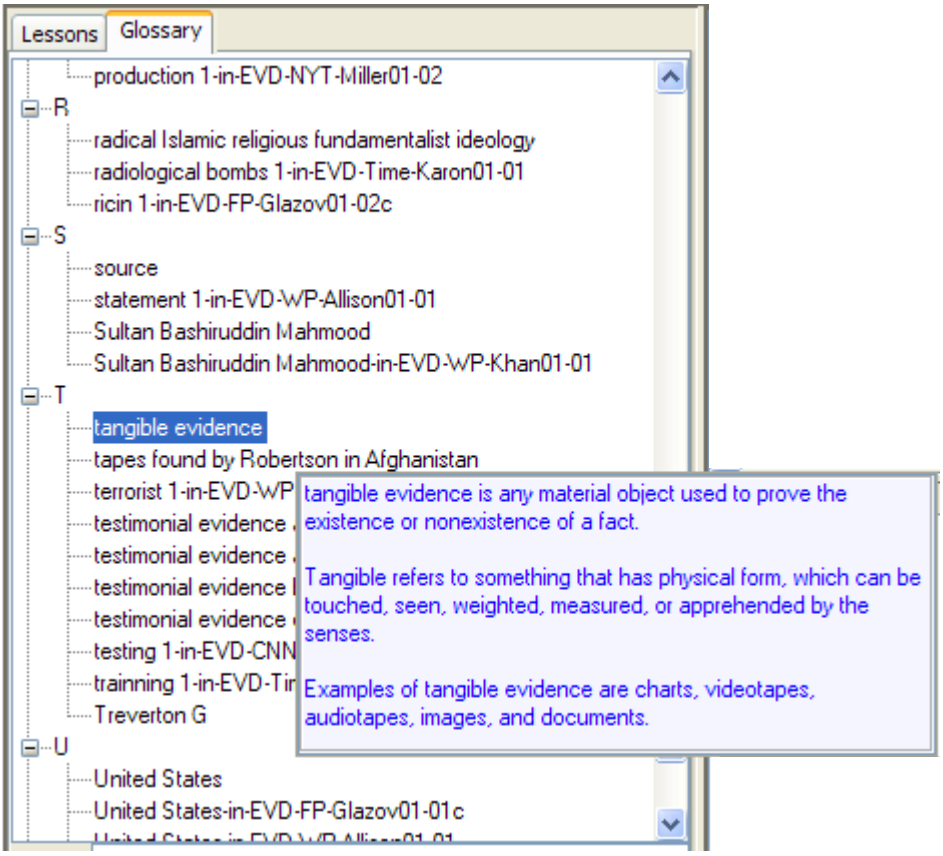


Figure 73: Sample Glossary

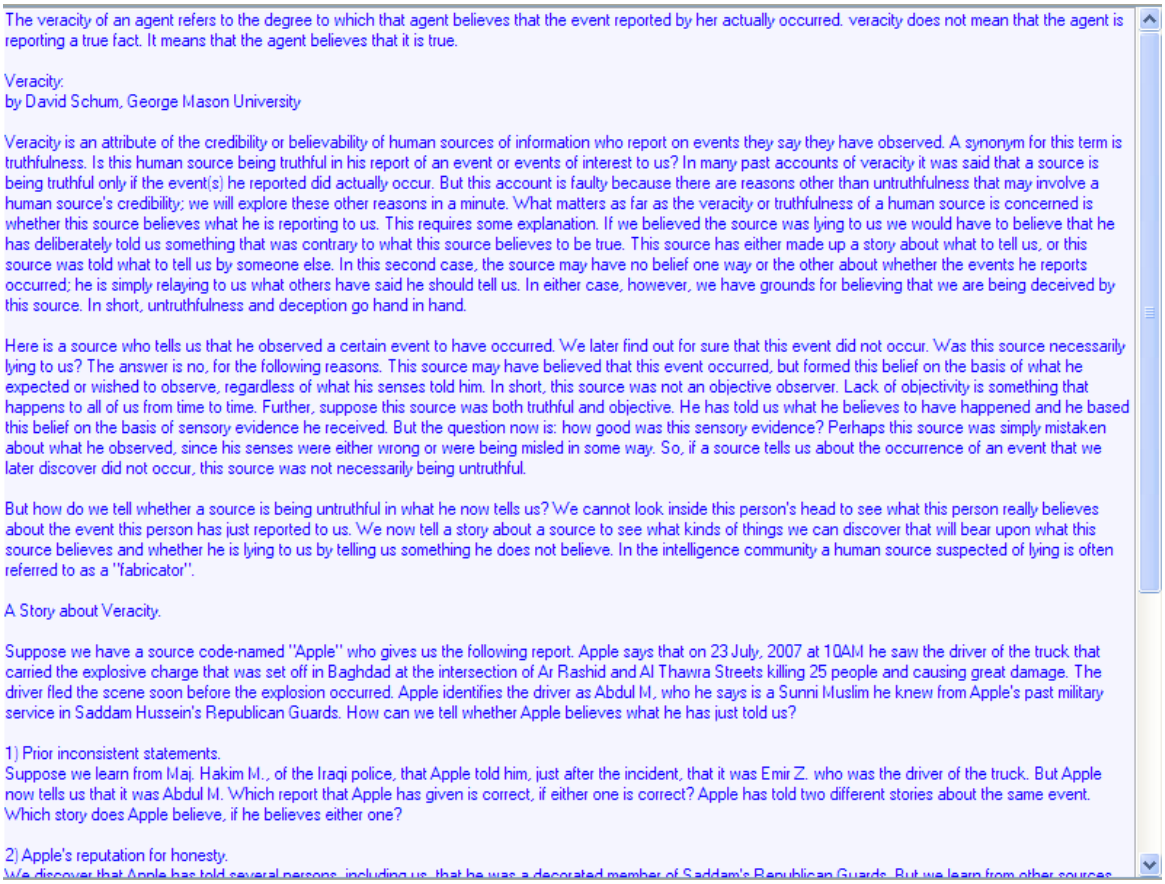


Figure 74: Presentation of the Veracity Concept

Tutoring Strategies

During the design process, the instructor can also provide the tutoring strategies to teach the lessons in different ways, in order to increase the effectiveness of tutoring (Kukla et al., 2002). To be specific, the instructor can design the lesson to display the abstract problem solving strategies in different orders, either bottom up or top down or any other way considered most appropriate by the instructor. The tutoring strategies are represented by the pair of numbers associated with each lesson component in the lesson script. More details on the tutoring strategies are provided in Section 5.1.2.

What we have discussed so far in the generated knowledge section of the pedagogical knowledge is the lesson module of tutoring module. The next topic is the pedagogical knowledge for its test module. The test module is provided to measure the student's understanding of the learned subject. The pedagogical knowledge for the test module consists of test questions, including explanations and hints.

Test Questions

A test question is generated from a test rule and is presented in the context of where the reduction should have been in the reasoning tree. Figure 75 shows a reasoning step from a test question in which one of the sub-problems was dropped. The reasoning step is bordered red and located in a sub-tree as its context. The student must judge if the presented reasoning step is correct, incorrect or incomplete. In this test question, the student is asked if it is correct to assess the believability of the report fragment *EVD-TRC-Najm01-01c,* where Najm S. cites Osama Bin Laden, by only assessing the believability of Najm S. The correct answer must be "Incomplete" due to the fact that the reasoning is missing a sub-problem which is the assessment of the believability of Osama Bin Laden as the source of the information.

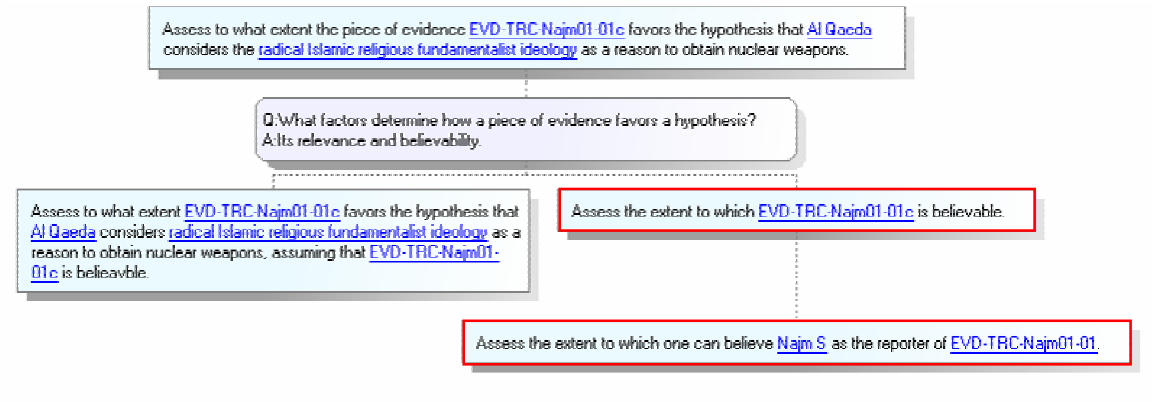


Figure 75: Reasoning Step from a Test Question

Hint

During the test, the student is provided with the relevant glossary (as discussed earlier and) and with hints (they are associated with penalties in student's assessment). The hints are learned from the instructor during the test learning process. An example of specific hint is:

"*EVD-TRC-Najm01-01c was obtained as testimonial evidence of Osama bin Laden cited in EVD-TRC-Najm01-01 by Najm S. Let us assume that Osama bin Laden is not believable. Does this affect the believability of EVD-TRC-Najm01-01c?*

The believability of some information refers to the degree to which that information is considered to be true. Similarly, the believability of an agent refers to the degree to which the information provided by that agent is considered to be true.

Belief is:

1: a state or habit of mind in which trust or confidence is placed in some person or thing

2: something believed; especially: a tenet or body of tenets held by a group

3: conviction of the truth of some statement or the reality of some being or phenomenon especially when based on examination of evidence.

Merriam-Webster's Online Dictionary, http://www.m-w.com/dictionary/belief. "

Explanation

The other type of information given when the student answers a test question is the explanations. This type of knowledge is similar to the hints with a minor difference: the explanations are given based on the student's answer which may be *correct, incorrect* or *incomplete.* Both the hints and the explanations are structured similarly. For instance, if the student answers the reduction from Figure 75 is *incorrect*, then the received explanation is:

"*EVD-TRC-Najm01-01c was obtained as testimonial evidence of Osama bin Laden cited in EVD-TRC-Najm01-01 by Najm S. Therefore its believability depends both on the believability of the reporter (Najm S) and the believability of the source (Osama bin Laden). For instance, if either Osama bin Laden is lying, or Najm S is distorting Osama bin Laden's testimony, then the information provided by EVD-TRC-Najm01-01c is not true.*"

What is important for hints and explanations is that they are not defined for each generated test, but they are learned from specific examples, and generated automatically, as described in Section 5.2.

The system follows a *scaffolding* approach, where the test questions are presented in a context from simple to complex (Dabbagh, 2007). This is achieved by following a concrete reasoning tree from top to bottom.

6.2.2 Knowledge Management

The knowledge management is performed by two modules: Management of Abstract Knowledge (MAK) and Management of Tutoring Knowledge (MTK).

Management of Abstract Knowledge

MAK is a module that handles the abstraction process of problem solving knowledge in a particular domain. The product of that process is the abstract knowledge that is stored in the pedagogical knowledge base, as described in Section 6.2.1. This module includes a tool named *Abstraction Editor* (see Figure 76) to abstract the knowledge in the domain knowledge base. The knowledge in the domain knowledge base generates a concrete reasoning tree and the abstraction of that tree results in the abstract reasoning tree. The Abstraction Editor allows the user to abstract a concrete reasoning tree into an abstract tree through a drop-and-drag operator. The user selects one or several nodes in the concrete tree (shown in the left part of Figure 75) to be abstracted into an abstract node in the right panel. The editor is able to recognize the parent of the newly created abstract node in the abstract tree (if this exists) to properly integrate the new abstract node into the abstract tree. More details on the reasoning tree abstraction process are provided in Section 3.3. As the abstract reasoning tree is built by using the editor, the MAK module learns the corresponding abstraction rules which govern how reasoning trees are to be

abstracted in general. The abstraction rules are then stored in the pedagogical knowledge base.

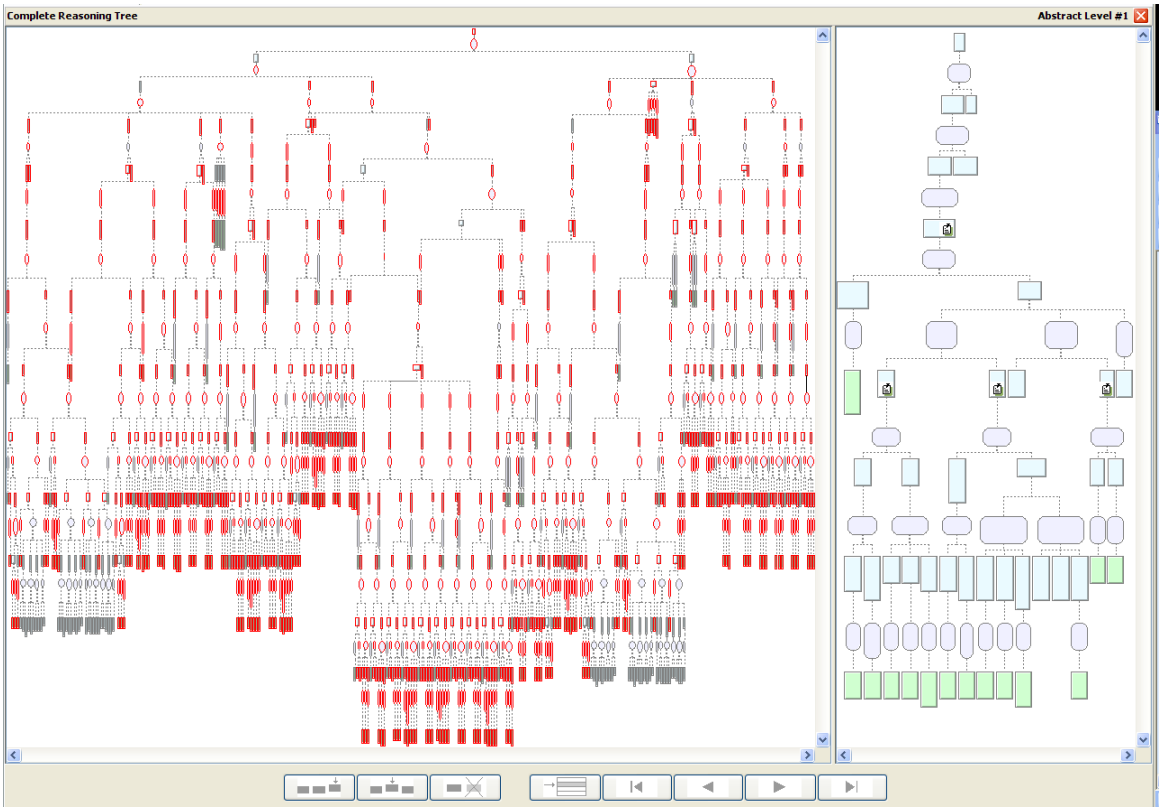


Figure 76: Interface of the Abstraction Editor

Management of Tutoring Knowledge

Tutoring knowledge includes all the pedagogical knowledge except the abstract knowledge. The tutoring knowledge is developed by the instructor, with the help of the Authoring module which will be described in next sub-sections. The knowledge then is stored in the pedagogical knowledge base. This knowledge is retrieved either by the

Authoring module for update or by the Tutoring module for lessons and tests generation. The TKM module is responsible for managing the storing, updating and retrieval of the tutoring knowledge.

6.2.3 Authoring Module

The Authoring module consists of three main sub-modules: lesson design, test learning and lesson script engine. The lesson design module and test learning module are used by the instructor. The lesson script engine converts the lesson components designed by the instructor into corresponding lesson scripts.

Lesson Design Module

The lesson design module is used by the instructor to builds lessons from the abstract tree, as illustrated in Figure 53. The Lesson Editor has two panels: the one on the left is for the abstract tree and the other is for designing lessons. The process of designing lessons is discussed at length in Section 5.1.1. In this section, we focus mostly on the authoring part of the lesson design process. The instructor uses the available toolbox to create the lesson components such as title, objectives, header, annotations, definitions, examples, and so on. The tool also lets the instructor review how the lessons are going to be displayed during the tutoring session. The outcome of the lesson design is the lesson script which is saved in the pedagogical knowledge base. The tutoring module groups all the lessons together to form the curriculum, based on the pre-requisites and post-requisites automatically inferred from the abstract tree.

Figure 77 shows the widget toolbar which is a part of the Lesson Editor. The toolbar has multiple widget buttons which simplify the task of creating lesson components.

Section 5.1.7.1 presents in detail the functionality of this widget toolbar. In short, the instructor can design the lesson, configure the display order of each component and preview the design. Once the instructor is satisfied with the lesson, the lesson script engine generates the lesson scripts based on the current setting of the lessons and saves it in the pedagogical knowledge base.

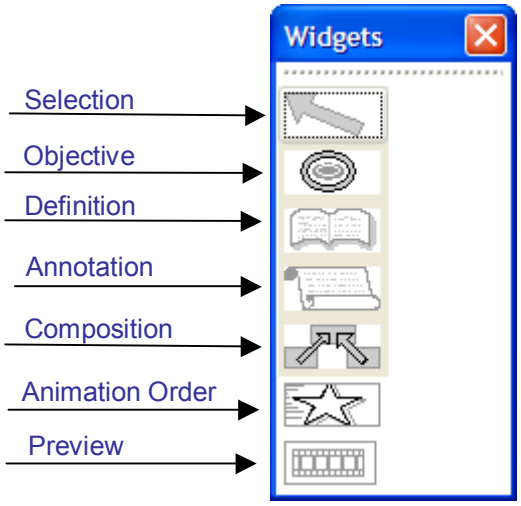


Figure 77: Widget Toolbar for Lesson Design

Lesson Script Engine

During the lesson design, the instructor builds the lessons by dragging and dropping some lesson components from the widget toolbar shown in Figure 77 and configuring them to achieve some particular tutoring effect. Once the lesson design is finished, the lesson scripts are generated by the script engine. The engine scans all the lesson components from the top down. For each of the lesson components, the engine captures its properties, such as the order and duration values, its relationships with other lesson

components, its description, and characteristics (see Appendix B for detailed information on the lesson script description). The lesson script engine then formats the obtained information in the ABLE language (see Appendix A) to create the lesson scripts. The scripts then are saved in the pedagogical knowledge base.

Test Learning Module

As presented in Section 5.2, the instructor teaches the system how to generate test questions. The system learns test rules by generalizing the test examples designed by the instructor. The instructor designs specific test questions by using the *Test Editor* shown in Figure 78. The editor has a main panel that displays a reasoning step from the concrete reasoning tree. This reasoning step serves as a test example. The instructor can manipulate the sub-problems of the example in three different ways: modification of sub-problems, dropping one or several of them, or adding deliberately wrong sub-problems. Each of the modification creates a different type of test example. They are modification test, omission test and construction test. In the right panel, the instructor defines the explanations and the hint. By default, the right answer for an omission test is "*incomplete"* and for a modification test is *"incorrect"*. However, the instructor can overwrite that default value by making a different selection in the "*Overwrite default assessment"* radio box. For example, in a modification test, the modified sub-problem may be equivalent to the original one and test's answer should be "*correct"* instead of "*incorrect"*.

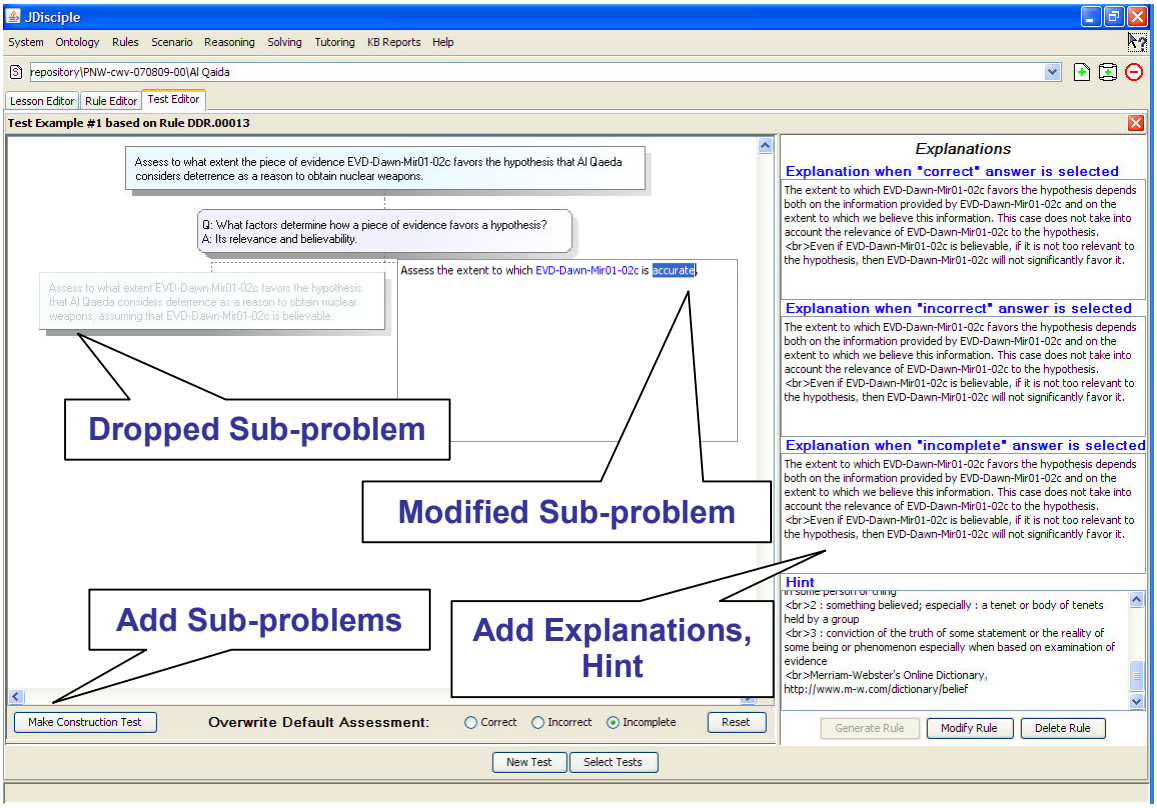


Figure 78: Interface of the Test Editor

The procedure of generating the test rules is detailed in Section 5.2.1. In this section, we just briefly summarize this process. The basic idea of learning the test rules is to extend a previously learned domain rules with test-related components that are appropriately generalization from specific examples provided by the instructor.

6.2.4 Tutoring Module

The tutoring module is responsible for generating lessons from lesson scripts and exercises from the test rules. The generated lessons and exercises are then presented to the students under the control of the Student Model.

Lesson Generation Module

Generating lessons based on existing domain knowledge base is done automatically, as described in Section 5.1.3. All the generated lessons contain two main parts: the abstract problem solving strategies and their examples.

The abstract strategies are generated from the lesson scripts (see Section 5.1.2). The examples are generated based on the abstract reasoning tree and a concrete reasoning tree. The example set is then displayed heuristically based on their relative similarity and complexity. The examples to be displayed can also be selected by the user.

The generated lessons have two auxiliary components, table of contents and glossary, both generated automatically.

The lesson window (Figure 79) provides several functions that can be used by a student to follow a lesson, either by reading and/or by listening. There are three panels, the one on the left is the table of content and glossary, the middle panel presents the lesson's content, and the right panel contains the automatically generated lesson's text. The middle panel also contains two sub-panels. The one on top is for the abstract problem solving strategies and the bottom one for the examples.

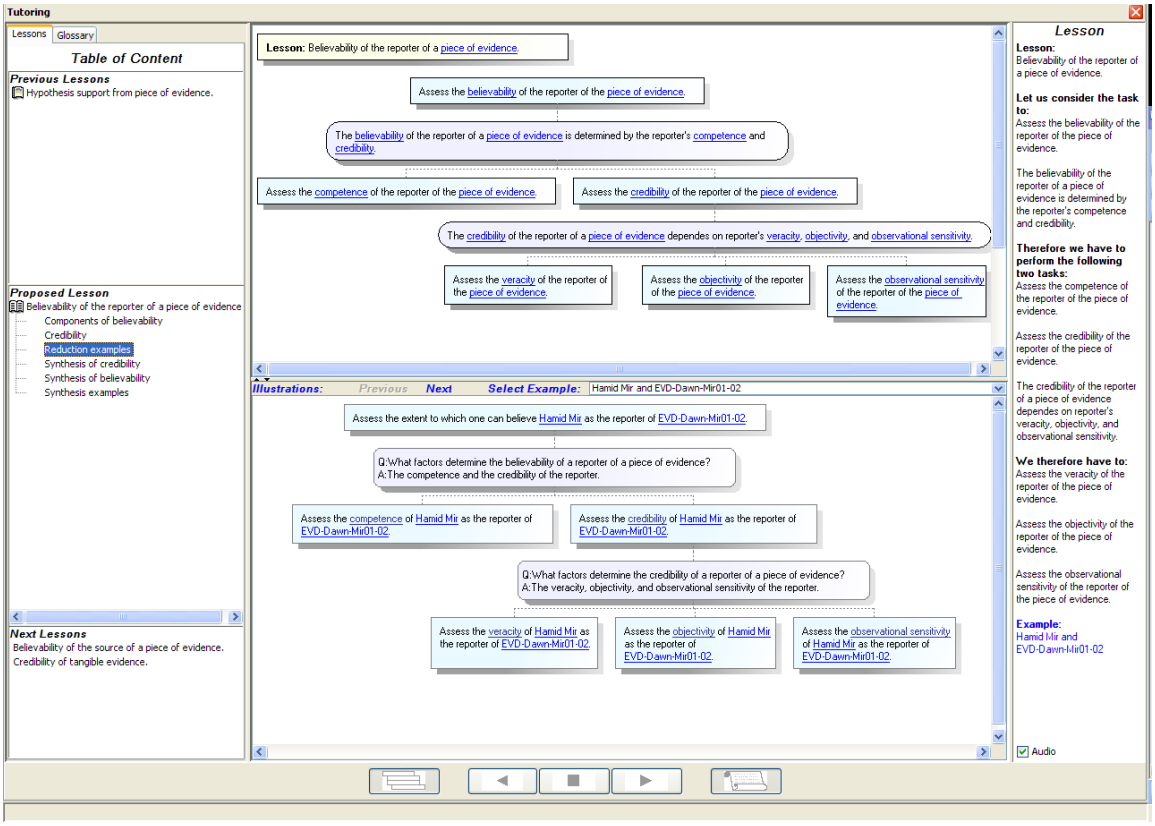


Figure 79: Lesson Interface

The table of contents is hidden by default but available on request. It has three sub-panels: the top one is for the previously learned lessons, the middle one is for the current lesson and the bottom panel is for the next available lessons. Each lesson has several components, presented in two groups: reduction and synthesis. Each group has a set of examples represented by "*Reduction examples*" or "*Synthesis examples*". Once the current lesson reaches the end, the next available lesson will replace the current lesson which moves to the Previous Lessons panel. The glossary panel is also available when the student needs it. Clicking on a hyperlink in the lesson will set the focus on the glossary panel and display the full description of the selected term, as illustrated in Figure

73. Similarly, the lesson's text is available on request only. The student can either read it or listen to it. The text reflects what has been shown visually in the lesson panel. The current line is colored blue and spoken. The text panel helps the students to follow the lessons in a traditional way.

The lesson panel is the main focus of the lesson window. The abstract lesson panel teaches the problem solving strategy that the teacher constructed in the design phase. The lower panel illustrates it with a set of examples. The strategy is taught step by step. The student has to click on *next* or *previous* buttons to move forward or backward. Each step is spoken by default and can be turned off as an option.

Test Generation Module

The test generation module generates specific test questions by applying the learned test rules. Figure 78 illustrates the test editor where the instructor defines a test question. LTAS then learns the test question rule from this example, as described in Section 5.1.3.3. These rules are saved in pedagogical knowledge base.

Test-taking is illustrated in Figure 80. It has five sub-panes. The top left panel is to display the test question and its context, the middle left is the pool of available sub-problems for construction test. The top right panel is the location for answer, explanations and hint. The bottom right panel is the glossary of terms used in the test question. The bottom left is the navigational and assessment panel where the grade is posted. There are two test-taking modes: self-test and assessment. The self-test mode lets the student go back to the appropriate lesson for review. The latter mode does not allow

this. All the tests are generated automatically and are dynamically changed each time the test starts. More details on this process are provided in Section 5.2.2.

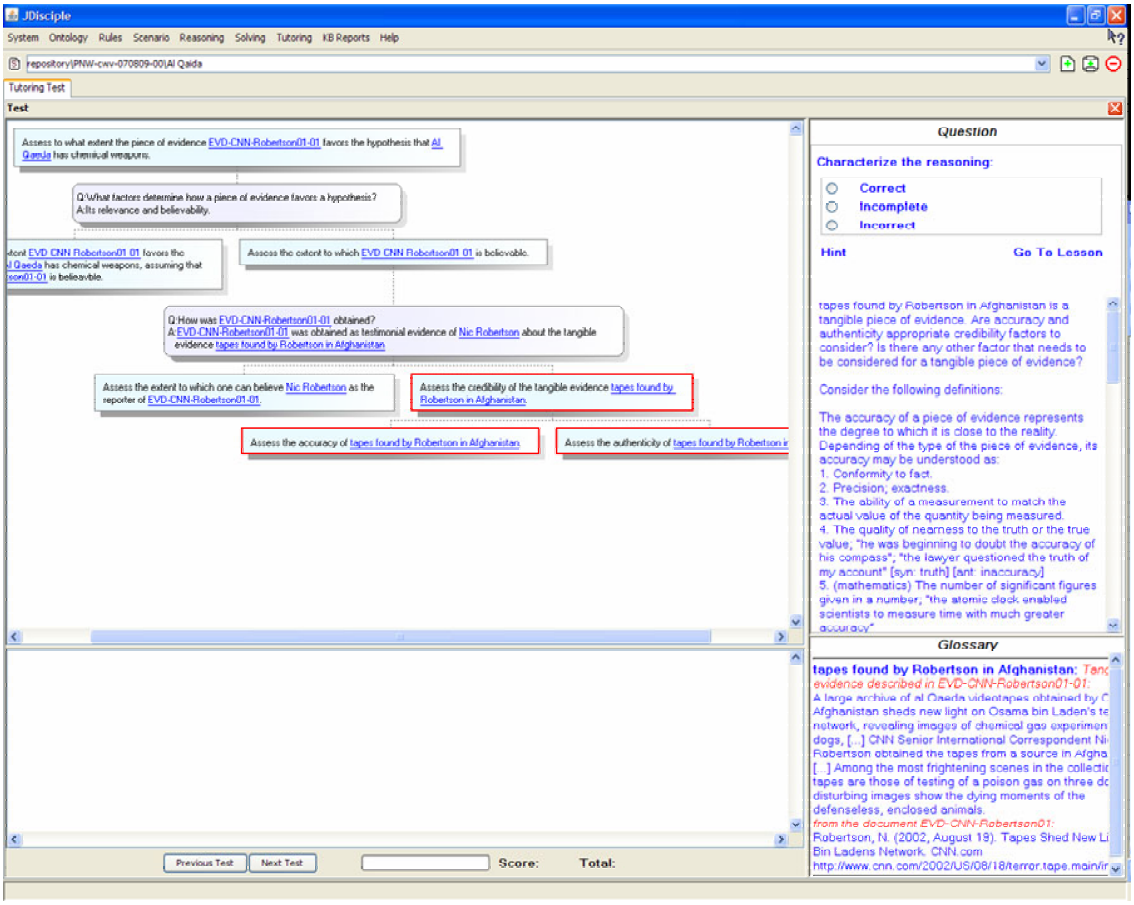


Figure 80: Test Generation Interface

6.2.5 Student Module

The student module contains information about the student, such as the lessons taken and the failed tests. It determines which lessons belong to the list of previous lessons, and the list of the next available lessons. It also controls the test generation process, by only allowing the tests that are included in the presented lessons. One important aspect of

this module in monitoring the tests is providing remedial test questions that are similar to the failed tests.

We have built a simplified student model in order to offer minimal support for the other developed functionality. Further research is needed to develop and integrate a more complex student model and to adapt the tutoring to it.

6.3. Methodology for Building Tutoring Systems

A learning and Tutoring Agent Shell (LTAS) allows the instructor to quickly develop a tutoring system that can tutor expert problem solving knowledge in a particular domain. Because the LTAS is build on top of a Learning Agent Shell (LAS), there are several assumptions regarding the LAS modules:

- The domain knowledge base is already developed. The expert knowledge has been acquired by LAS.
- The concrete reasoning trees are generated by the LAS for specific problems.
- The abstract reasoning tree is possibly partially constructed during the modeling of the knowledge of the subject matter expert. In fact, part of the abstract reasoning tree is constructed for human-agent collaboration in problem solving.

There are several steps that are required to be done in sequence (see Figure 81). First of all, the instructor needs to construct the abstract reasoning tree, if one was not already developed. The abstraction of a concrete reasoning tree requires the instructor to have deep knowledge of the application domain. Therefore, the instructor is usually also the subject matter expert. The purpose of the abstraction for tutoring is to uncover the

problem solving strategies used in solving problems, and to develop a hierarchical structure of the abstract problem solving strategies.

In the second step the instructor designs the lesson by using the abstract reasoning tree. The instructor can create a lesson based on any abstract problem. The lesson content is automatically built based on the content of the selected abstract components that the lesson is built upon. The instructor however can modify it, for instance, by selecting the order in which its parts are presented to the students. When the lessons are saved, the lesson scripts are generated accordingly and saved into the pedagogical knowledge base.

The last step is authoring the test questions by the instructor.

When a student uses the system to learn expert problem solving knowledge, the lessons are generated dynamically, based on the current scenarios included in the domain knowledge base. The student model captures the student progress to provide the appropriate the lessons and test questions.

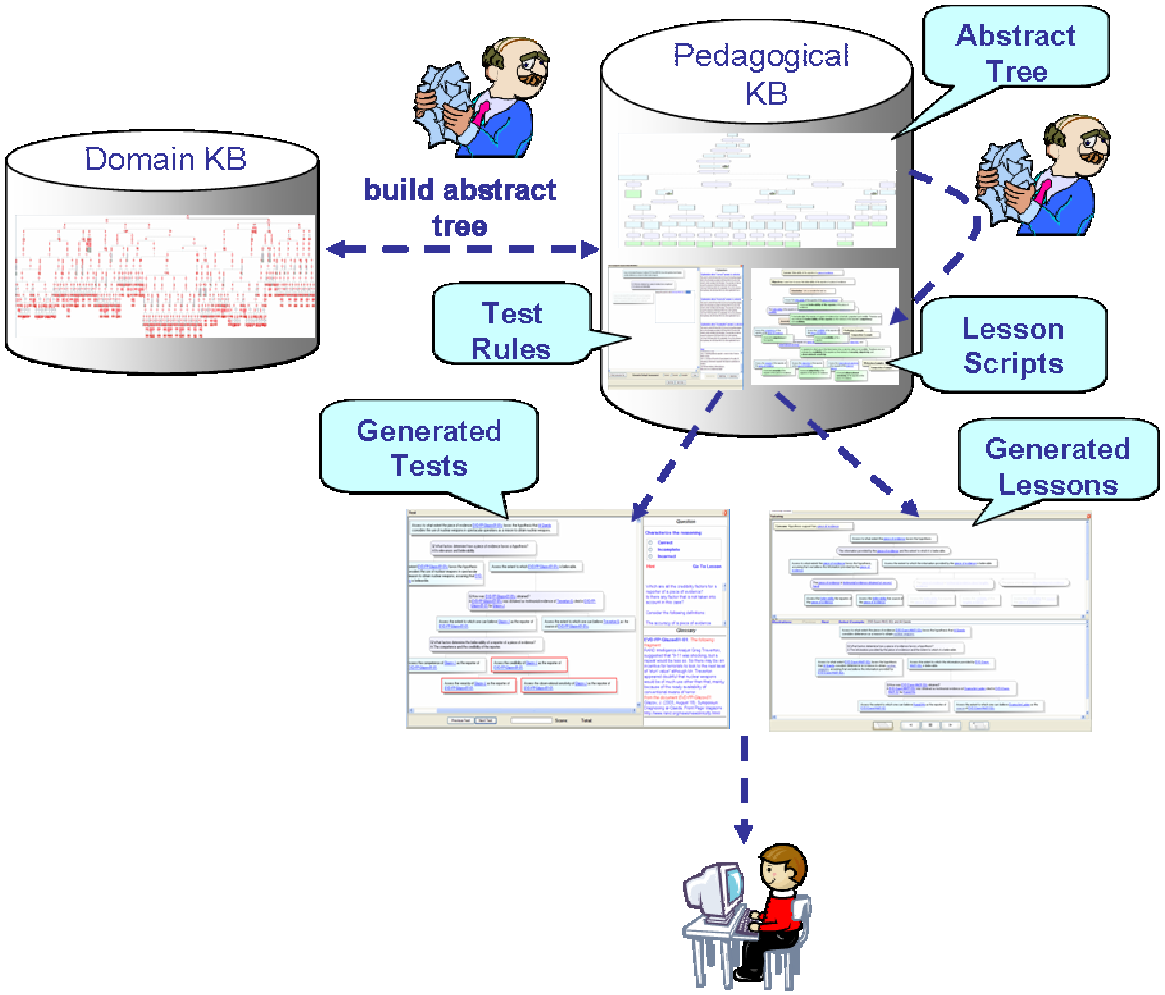


Figure 81: Methodology for Building a Tutoring System

7. Contributions and Future Research

This chapter concludes the dissertation with the summary of my contributions and the most promising directions for future research.

7.1. Summary of Contributions

This dissertation research has advanced the state of the art in the area of knowledge-based agents for expert problem solving.

The main contribution of my dissertation is the development of a theory for the abstraction of reasoning that facilitates:

- human-agent collaboration in complex problem solving and decision-making;
- rapid development of intelligent tutoring systems for complex problem solving;
- teaching complex problem-solving to non-experts.

Abstraction has been previously used in different areas of Artificial Intelligence, such as, Planning, Problem Solving, Constraint Satisfaction, Reasoning about Physical Systems, to facilitate the search for solutions in large spaces. The general idea is to first find an approximate solution in a reduced, abstract space, and then use it to guide the search for the actual solution in the large concrete space. In our research we have not investigated how to use abstraction to develop a reasoning tree that solves a problem. Instead, we have investigated how to abstract a complex reasoning tree to facilitate its understanding.

Effective human-agent collaboration in complex problem-solving and decision-making requires an ability of the user to easily browse, understand, and modify complex reasoning, with many thousands of reasoning steps. Our theory of abstraction of reasoning for collaborative problem solving allows:

- the partition of a complex tree into meaningful and manageable sub-trees;
- the abstraction of individual sub-trees;
- the automatic generation of an abstract tree that plays the role of a table of contents for the display, understanding and navigation of the concrete tree.

Abstraction of reasoning is also very important for teaching complex problem-solving to non-experts. Although based on the same general theory, we have found that the abstraction for tutoring is different from the abstraction for collaborative problem solving. In the abstraction for problem solving, the emphasis is on easily identifying the main sub-problems of a given problem, and their solutions. In the abstraction for tutoring, however, the emphasis is on how to abstract the problem reduction and solution synthesis processes, in order to identify the abstract strategies to be taught. Our theory of abstraction of reasoning for tutoring allows:

- the definition of abstract problem solving strategies for tutoring;
- the rapid development of lesson scripts for teaching these strategies;
- the automatic generation of specific lessons corresponding to a particular expertise domain.

Another major contribution of my dissertation is the development of methods deriving from our theory of abstraction, as indicated in the following.

We have developed a method for rapid authoring of lessons for tutoring problem solving in a complex domain. The lessons are organized around the abstract problem solving strategies to be taught. They present these strategies under the control of the student who may request definitions or detailed descriptions of the used concepts, as well as concrete examples of the application of these strategies. An important characteristic of these lessons is that they are automatically customized based on the content of the domain knowledge of the tutoring agent. In particular, a lesson will automatically teach only those cases of an abstract problem solving strategy that can be with the current domain knowledge base. Also, changing the domain knowledge base will automatically change the generated examples, without any change in the design of the lesson. The automation of example creation is a main factor in cutting down the time to build the lessons. Examples are essential parts of a lesson and their availability, number and diversity play an important role in making a lesson more interesting and understandable.

We have developed methods for:

- Learning different types of test questions by modifying and enhancing examples of problem reduction rules from the domain knowledge base.
- Automatic generation of test questions in the context of a reasoning tree, together with hints and explanations.
- Dynamic adaptation of the generated test questions to the lessons taken by a student, and an ability to invoke the lesson corresponding to a given test question.

The types of test questions learned are:

- Omission test question (knowledge level questions where a student is asked to judge the completeness and correctness of a problem reduction that may omit some sub-problems).
- Modification test question (comprehension level questions where a student is asked to judge the completeness and correctness of a problem reduction that may have some sub-problems modified).
- Construction test question (analysis level questions where a student is asked to define the reduction of a given problem by selecting sub-problems from a given list.

Finally, another major contribution is the development of:

- The concept of "learning and tutoring agent shell" and the associated methodology for rapid development of an intelligent tutoring system.
- An experimental learning and tutoring agent shell.
- An experimental tutoring system for the domain of intelligence analysis which has been used by military officers at the Army War College and by students at George Mason University.

7.2. Future Research Directions

There are also various limitations of the obtained results that point to future research directions.

The current methods for defining abstractions (both those for collaborative problem solving and those for tutoring) are to be considered methods for a knowledge engineer. They need to be further simplified to be used by a subject matter expert.

The lesson design methods could be extended to allow additional customization by an instructor. For instance, the current lessons have to first introduce an abstract strategy and then can illustrate it with examples. The instructor may wish to define lessons which first introduce examples of reasoning and then present their abstraction.

Also the generated lessons should be made more interactive and engaging. In general, our research has focused on the artificial intelligence aspects of tutoring rather than the instructional design ones. Therefore, there are good opportunities for advancing this research by emphasizing more the instructional design and educational aspects which have only been developed to a limited extend. They include *building fluency, drill and practice, and repetition* where the same type of test questions are presented repeatedly to help student acquire fluency; *chaining and logical sequencing of content* where the lesson contents are presented in a hierarchical way of problem reduction/solution synthesis paradigm; and *scaffolding* where test questions are presented in a context from simple to complex (Dabbagh, 2007). A tutoring system is more effective if it includes instructional strategies for developing student's creativity such as, *self-directed learning, learning by discovery, hypothesis generation.*

The research on the abstraction of reasoning trees described in this dissertation can naturally be extended to enhance other capabilities of a knowledge-based agent. For example, the abstract reasoning patterns used in tutoring may guide the acquisition of related problem solving strategies from a subject matter expert. Also, one could investigate the generation of solutions and justifications at different levels of abstraction.

Appendix A: Abstraction-Based Lesson Emulation (ABLE)

Table 13: The ABLE Scripting Language

```
//Tokens
Problem          ;The abstract problem
Reduction        ;The abstract reduction (reduction process)
Solution         ;The abstract solution
Synthesis        ;The abstract synthesis (composition process)
Title            ;The lesson title
Objectives       ;The lesson objectives
Annotation       ;The annotation
Description      ;The definitions of the new terms
//Examples
ReductionExample ;The reduction example
SynthesisExample ;The synthesis example

//Decorative Tokens
Decorative := Annotation | Description | <empty>     ;Decorative tokens
Decoratives := Decorative | Decoratives              ;List of decorative tokens
DécorProblem := Problem Decoratives                  ;Problem with decorative tokens
DécorReduction := Reduction Decoratives              ;Reduction with decorative tokens
DécorSynthesis := Synthesis Decoratives              ;Synthesis with decorative tokens
DécorSolution := Solution Decoratives                ;Solution with decorative tokens
DécorTitle := Title Decoratives                      ;Title with decorative tokens
DécorObjectives := Objectives Decoratives            ;Objectives with decorative tokens

//Reduction and Synthesis Process
Sub-problem := DécorProblem | DécorSolution | <empty> ;A sub-problem
Sub-problems := Sub-problem | Sub-problems    ;Set of sub-problems
Solutions := DécorSolution Solutions | <empty> ;Set of solutions
ReductionSet := DécorReduction Sub-problems ;A reduction set
SynthesisSet := Solutions DécorSynthesis          ;A synthesis set
ReductionProcess := DécorProblem ReductonSet          ;A reduction process
SynthesisProcess := SynthesisSet DécorSolution;A synthesis process

//Reduction and Synthesis Examples
ReductionExamples := ReductionExample ReductionExamples | <empty>
CompositionExamples := CompositionExample CompositionExamples | <empty>
//Lesson Section
Section := ReductionProcess ReductionExamples |
   SynthesisProcess CompositionExamples
LessonSection := Section | <empty>
LessonSections := LessonSection LessonSections
```

```
//Lesson Header
LessonHeader := DécorTitle DécorObjectives
//Lesson
Lesson := LessonHeader LessonSections
```

Table 14: LifeCycle Feature

```
Token := Problem | Reduction | Solution | Synthesis |
      Title | Objective | Annotation | Description | Test
Order := 1 | after (Token)                ;When the component is displayed
Duration := -1 | 0 | 1 | before (Token)    ;How long the component is displayed
LifeCycle := Order Duration | <empty>
LifeCycles := LifeCycle LifeCycles
TimingToken := Token LifeCycles
```

Table 15: Order and Duration Computation

$Order_{Token\ i}$ = 1: Token i is the first one to be displayed.
$Order_{Token\ j}$ = after($Token_k$) = $Order_{Token\ k}$ + 1.
Example:
if $Order_{Title}$ = 1 and $Order_{Objective}$ = after(Title)
then $Order_{Objective}$ = $Order_{Title}$ + 1 = 2.

$Duration_{Token\ i}$ = -1: Token i always appears on the screen
$Duration_{Token\ i}$ = 0: Token i never appears on the screen
$Duration_{Token\ i}$ = 1: Token i appears on the screen for one step.
$Duration_{Token\ i}$ = before($Token_j$) = $Order_{Token\ j}$ – $Order_{Token\ i}$
Example:
if $Order_{Problem}$ =10, $Order_{Objective}$ =5 and $Duration_{Objective}$ = before(Problem)
then $Duration_{Objective}$ = $Order_{Problem}$ - $Order_{Objective}$ = 10 – 5 = 5.

Appendix B: Lesson Scripts in XML

Table 16: Lesson Annotation Script in XML

```
<LessonAnnotation id="annotation_1" parent="title">
   <LifeCycles>
      <LifeCycle>
         <Order value=after(“title:0”) />
         <Longevity value="1" />
      </LifeCycle>
   </LifeCycles>
   <Descriptions>
      “Let us consider the problem:”
   </Description>
</LessonAnnotation>
```

Table 17: Lesson Definition Script in XML

```
<LessonDefinition id="definition_0" parent="problem_0">
   < LifeCycles>
      < LifeCycle>
         <Order value="problem_0:0" />
         <Duration value="1" />
      </ LifeCycle>
   </ LifeCycles>
   <Terms>
      <Term name="piece of evidence" />
   </Terms>
</LessonDefinition>
```

Table 18: Lesson Title Script in XML

```
<LessonTitle id="title">
   <LifeCycles>
      <LifeCycle>
         <Order value="1" />
         <Duration value="-1" />
      </LifeCycle>
   </LifeCycles>
   <Description>
      Assess to what extent the piece of evidence supports the hypothesis
   </Description>
</LessonTitle>
```

Table 19: Lesson Objective Script in XML

```
<LessonObjective id="objectives" parent="title">
   <LifeCycles>
      <LifeCycle>
         <Order value="1" />
         <Duration value="1" />
      </ LifeCycle>
   </ LifeCycles>
   <Description>
      There are 2 objectives: &lt;p> Learn how to handle the piece of evidence. &lt;p>
Learn how to assess the piece of evidence to support a hypothesis
   </Description>
</LessonObjective>
```

Table 20: Lesson Problem Component Scrip in XML

```
<LessonProblem id="problem_1" parent="annotation_2">
   <AbstractProblemReference kbPartName="LTA final" index="3" />
   <LifeCycles>
      < LifeCycle>
         <Order value=after("annotation_2:0") />
         <Duration value="-1" />
      </ LifeCycle>
   </ LifeCycles>
   <Description>
      Assess to what extent piece of evidence supports the hypothesis, assuming that we
believe the information provided by the piece of evidence
   </Description>
</LessonProblem>
```

Table 21: Lesson Reduction Script in XML

```
<LessonReduction id="reduction_0" parent="problem_0">
   <AbstractReductionReference kbPartName="LTA final" index="3" />
   <LifeCycles>
      <LifeCycle>
         <Order value="problem_0:0" />
         <Duration value=before("problem") />
      </LifeCycle>
   </LifeCycles>
   <Description>
      The information provided by the piece of evidence and the extent to which it is
```

```
believable.
   </Description>
</LessonReduction>
```

Table 22: The Lesson Solution Script in XML

```
<LessonSolution id="solution_0" parent/host="reduction_0">
   <AbstractSolutionReference kbPartName="LTA final" index="1" />
   <LifeCycles>
      < LifeCycle>
         <Order value=after("reduction_0:0") />
         <Duration value="-1" />
      </ LifeCycle>
   </ LifeCycles>
   <Description>
      Assessed believability of the reporter of the piece of evidence
   </Description>
</LessonSolution>
```

Table 23: The Lesson Synthesis Script in XML

```
<LessonSynthesis id="synthesis_2" host="reduction_0">
   <LifeCycles>
      <LifeCycle>
         <Order value="composition_1:0" />
         <Duration value="-1" />
      </LifeCycle>
   </LifeCycles>
   <Description>
      Determine the likelihood of the hypothesis given the likelihood of the credibility of
the piece of evidence.
   </Description>
</LessonSynthesis>
```

REFERENCES

REFERENCES

Aleven, V., McLaren, B., Sewall, J., and Koedinger, K. The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains, in *the Proceedings of the 8th International Conference on Intelligent Tutoring Systems,* Jhongli, Taiwan, June 26-30, 2006.

Aleven, V., and Rose, C. P. Towards Easier Creation of Tutorial Dialogue Systems: Integration of Authoring Environments for Tutoring and Dialogue Systems, in *Proceedings of the ITS Workshop on Tutorial Dialogue Systems*, Alagoas, Brazil, 2004, Springer.

Anderson, J.R. Intelligent Tutoring and High School Mathematics. in *The second International Conference on Intelligent Tutoring System*, (Berlin, Germany, 1992), Spring–Verlag.

Anderson, J.R. Rules of the Mind. Lawrence Erlbaum. 1993.

Anderson, J.R. The Expert Module. Martha C. Polson, J.J.R. ed. *Foundations of Intelligent Tutoring System*, 1998, 21-53.

Anderson, J.R., Boyle, C. F., and Yost, G. The Geometry tutor. *The Journal of Mathematical Behavior*. 5-20, 1986.

Anderson, J.R., and Reiser, B. J. The LISP tutor *Byte*, 1985, 159-175.

Barbulescu M., Balan G., Boicu M., and Tecuci G. Rapid Development of Large Knowledge Bases in *Proceedings of the 2003 IEEE International Conference on Systems, Man & Cybernetics*, Volume: 3, pp. 2169 - 2174, Washington D.C., October 5-8, 2003.

Barr, A., Cohen, P. R., and Feigenbaum, E. A. The Handbook of Artificial Intelligence, Volume 1, 1998.

Blessing, S.B. A programming by demonstration authoring tool for model tracing tutors. in *Artificial Intelligence in Education*, (1997), 233-261.

Bloom, B.S. Taxonomy of Educational Objectives, *Handbook I: The Cognitive Domain*. David McKay Co Inc, New York, 1956.

Boicu, C., Tecuci, G., and Boicu, M. A Mixed-Initiative Approach to Rule Refinement for Knowledge-Based Agents. in *the AAAI-05 Fall Symposium on Mixed-Initiative Problem-Solving Assistants*, Arlington, VA, 2005, AAAI.

Boicu, M. Modeling and Learning with Incomplete Knowledge, *PhD Thesis in Information Technology*, *Learning Agents Laboratory, School of Information Technology and Engineering, George Mason University*, 2002.

Boicu, M., Tecuci, G., Marcu, D., Stanescu, B., Boicu, C., Balan, C., Barbulescu, M., and Hao, X. Disciple-RKF/COG: Agent Teaching by Subject Matter Experts. in *AAAI-IS02*, (2002), AAAI.

Boicu, M., Tecuci, G., Stanescu, B., Marcu, D., Barbulescu, M., and Boicu, C. Design Principle for Learning Agents in *Proceedings of AAAI-2004 Workshop on Intelligent Agent Architectures: Combining the Strengths of Software Engineering and Cognitive Systems,* July 26, San Jose, AAAI Press, Menlo Park, CA, 2004.

Bowman, M., Tecuci, G., and Ceruti, M. Application of Disciple to Decision Making in Complex and Constrained Environments, in *Proceedings of the 2001 IEEE Systems, Man and Cybernetics Conference, October 2001.*

Brown, J., Burton, R.R., and deKleer, J. Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II, and III. Sleeman, D., Brown, J.S. ed. *Intelligent Tutoring Systems*, Academic Press, New York, 1982, 227-282.

Buchanan, B. and Wilkins, D. (editors). *Readings in Knowledge Acquisition and Learning: Automating the Construction and Improvement of Expert Systems.* Morgan Kaufmann, San Mateo, CA, 1993.

Carbonell, J. AI in CAI: An artificial intelligence approach to computer aided instruction in *IEEE Transactions on Man-Machine systems*, (1970), 190-202.

Clancey, W.J. Dialog Management for Rule-Based Tutorials. in *International Joint Conference on Artificial Intelligence 6*, (Tokyo, Japan, 1979), William Kurfmann, Inc.

Clancey, W.J. The Handbook of Artificial Intelligence, William Kaufmann, Inc, Los Altos, CA, 1982.

Clancey, W.J. from GUIDON to NEOMYCIN and HERACLES in Twenty Short Lessons: ORN Final Report 1979-1985, *AI Magazine,* August 1986, 40-60.

Clancey, W.J. *Knowledge-Based Tutoring*. MIT Press, Massachusetts, 1987.

Cormen, T., Leiserson, C., and Rivest, R. Introduction to Algorithms. McGraw Hill, 1997.

Dabbagh, N. The Instructional Design Knowledge Base. Retrieved 09-29-2007 from Nada Dabbagh's Homepage, George Mason University, Instructional Technology Program. Website: http://classweb.gmu.edu/ndabbagh/Resources/IDKB/index.htm

Davis, R. Expert Systems: where Are We? And Where Do We Go From Here? *AI Magazine*, 1982, 3-22.

DeJong, K. Evolutionary Computation: A Unified Approach. MIT Press, Cambridge, MA, 2006.

Digangi, A. S., Jannasch-Pennell, A., Yu, H., C., and Mudiam, V. S. Curriculum-based Measurement and Computer Based Assessment: Constructing an intelligent, web-based evaluation tool. http://www.creative-wisdom.com/pub/scip_cbm.html. November, 1999.

Durham S. 2000. Product-Centered Approach to Information Fusion, *AFOSR Forum on Information Fusion,* Arlington, VA, 18-20 October, 2000.

Eugenio, B., Fossati, D., Yu, D., Haller, S., and Glass, M., Natural language generation for intelligent tutoring systems: a case study in *AIED 2005*, July 2005.

Eugenio, B., Glass, M., and Trolio, M., The DIAG experiments: Natural Language Generation for Intelligent Tutoring Systems. In *INLG02, The Third International Natural Language Generation Conference, 2002*, pages 120--127.

Even, M., Brandle, S., Chang, R., Freedman, R., Glass, M., Lee, Y., Shim, L., Woo, C., Zhang, Y., Zhou., Y., Michael, J., and Rovick., A., CIRCSIM-Tutor: An Intelligent Tutoring System using Natural Language Dialogue, *12th Midwest AI and Cognitive Science Conference,* Oxford OH, 2001, 16-23.

Feigenbaum, E.A. Knowledge Engineering in the 1980's, Dept. of Computer Science, Stanford University, Stanford, CA, 1982.

Feigenbaum, E.A. Tiger in a Cage: The Applications of Knowledge-based Systems. *The Fifth Annual Conference on Innovative Applications of Artificial Intelligence*. AAAI, 1993.

Fisher, D.H. Knowledge acquisition via incremental conceptual clustering. *Machine Learning* 2: 139–172. 1987.

Furnkranz, J. The Role of Qualitative Knowledge in Machine Learning. http://citeseer.ist.psu.edu/116655.html, 1992.

Giunchiglia, F. and Walsh, T. A Theory of Abstraction, *Artificial Intelligence* 56(2-3) pp 323-390. 1992.

Halff, H.M. Curriculum and Instruction in Automated Tutors. Martha Polson, J.J.R. ed. *Foundations of Intelligent Tutoring System*, 1988, 79-108.

Jarvis, M.P. Applying Machine Learning Techniques to Rule Generation in Intelligent Tutoring Systems. *Master thesis, Worcester Polytechnic Institute*, 2004.

Josephson, J., Chandrasekaran, B., Smith, J.W., and Tanner, M.C., A Mechanism for Forming Composite Explanatory Hypotheses. in *IEEE Trans. on Systems, Man and Cybernetics*, (1987), 445-454.

Kaschek, H. R., Intelligent Assistant Systems: Concepts, Techniques and Technologies. Idea Group Publishing, 2006.

Kirkpatrick D., Evaluating Training Programs: The Four Levels. Second edition, Berrett-Koehler Publishers, Inc. San Fransico, 1998.

Kodratoff Y., Tecuci, G., Learning Based on Conceptual Distance. in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1988, 897-909.

Koedinger, K. R., Aleven, V., and Heffernan, N. T. (2003). Toward a Rapid Development Environment for Cognitive Tutors. In U. Hoppe, F. Verdejo, & J. Kay (Eds.), *Proceedings of the 11th International Conference on Artificial Intelligence in Education,* AI-ED 2003 (pp. 455-457). Amsterdam: IOS Press.

Koedinger, K. R., Aleven, V., Heffernan, N., McLaren, B. and Hockenberry, M. Opening the Door to Non-Programmers: Authoring Intelligent Tutor Behavior by Demonstration. in *Intelligent Tutoring Systems 2004*: 162-174.

Kukla, E., Nguyen, N., and Sobecki, J., The consensus-based tutoring strategy selection in CAL systems. in *World Transactions on Engineering and Technology Education*, Vol.1, No.1, 2002.

Langley, P., Simon, H.A., Bradshow, G.L., and Zytkow, J.M. *Scientific Discovery: Computational Explorations of the Creative Processes*. MIT Press, Cambridge, MA, 1987.

Maiden, N.A.M., and Sutcliffe, A.G., A computational mechanism for parallel problem decomposition during requirements engineering. in *8th International Workshop on Software Specification and Design,* (Schloss Velen, Germany), Pages 159-163, 1996.

Matsuda, N., Cohen, W. W., and Koedinger, K. R., An Intelligent Authoring System with Programming by Demonstration. in *Japan National Conference on Information and Systems in Education.*, (2005).

Matsuda, N., Cohen, W. W., and Koedinger, K. R., Building Cognitive Tutors with Programming by Demonstration. in *International Conference on Inductive Logic Programming*, (2005), 41-46.

Matsuda, N., Cohen, W.W., Sewall, J., and Koedinger K.R. Applying Machine Learning to Cognitive Modeling for Cognitive Tutors. *Technical Report CMU-ML-06-105* July 2006.

Matsuda, N., Cohen, W.W., Sewall, J., Lacerda, G., and Koedinger, K.R. Evaluating a Simulated Student using Real Students Data for Training and Testing. in *Proceedings of the International Conference on User Modeling* (Berlin, Germany), pp. 107-116. 2007.

Meyers, W., Linear Representation of Tree Structure - a Mathematical Theory of Parenthesis-Free Notations. In *Proceedings of the third annual ACM symposium on Theory of computing*, Shaker Heights, Ohio, pp: 50-62. 1971.

Michalski, R., and Tecuci, G. *Machine Learning: A Multistrategy Approach*. Morgan Kaufmann, 1994.

Mitchell, T. Version Spaces: An Approach to Concept Learning, Stanford University, 1978.

Mitchell, T.M. Machine Learning. McGraw-Hill, 1997.

Murray, T., Authoring Intelligent Tutoring Systems: An Analysis of the State of The Art. in *International Journal of Artificial Intelligence in Education*, (1999), 98-129.

Murray, T., Blessing, S., and Ainsworth, S., Authoring Tools for Advanced Technology Learning Environments: Toward Cost-Effective Adaptive, Interactive and Intelligent Educational Software, Kluwer Academic Publishers, Netherlands, 2003 493-546.

Mustière, S., Zucker, J. D., and Saitta, L. An abstraction-based Machine Learning Approach to Cartographic Generalization. In *Proceedings of the 9th International Symposium on Spatial Data Handling*, Beijing, 2000, pp 50-63.

Nayak, P. P., and Levy, A. Y. A semantic theory of abstraction. In *Procedure of Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, 20–25 August 1995 (ed. A. Toshi), pp. 196–202.

Nguyen, D.T., Ho, B. T., and Shimodaira, H., A Visualization Tool for Interactive Learning of Large Decision Trees. in *12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, (Vancouver, BC, Canada, 2000), IEEE.

Ong, J., and Ramachandran, S. Intelligent Tutoring Systems: The What and the How, 2000. http://www.learningcircuits.org/2000/feb2000/ong.htm

Plaisted, D. 1981 Theorem proving with abstraction. *Artificial Intelligence*. 16, 47–108

Polson, M.C., and Richardson, J. J. *Foundation of Intelligent Tutoring System*. Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, 1988.

Powel G.M. and Schmidt C.F. 1988. A First-order Computational Model of Human Operational Planning, *CECOM-TR-01-8, US Army CECOM*, Fort Monmouth, New Jersey.

Roschelle, J. Learning in Interactive Environments: Prior Knowledge and New Experience, 1995.

Rumelhart, D., and McClelland, J.L. Parallel Distributed Processing. MIT Press, Cambridge, MA, 1986.

Russell, S., and Norvig, P. Artificial Intelligence A Modern Approach. Prentice Hall, Upper Saddly River, NJ, 1995.

Saitta, L. and Zucker, J.D.. Semantic Abstraction for Concept Representation and Learning. *Symposium on Abstraction, Reformulation and Approximation (SARA98)*, Asilomar Conference Center, Pacific Grove, California. 1998.

Sebastia, L., Onaindia, E., and Marzal, E., Decomposition of planning problems. in *AI Communications*, Volume 19, Issue 1, Pages: 49-81. 2006

Smith, S. Tutorial on Intelligent Tutoring System, 1998. http://www.cs.mdx.ac.uk/staffpages/serengul/Intelligent.Tutoring.System.Architectures.htm

Sutton, R. Learning to predict by the methods of temporal differences *Machine Learning*, 1988, 9-44.

Tecuci, G., Building Intelligent Agents: An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies. London, England: Academic Press, 1998.

Tecuci, G., Lecture Notes on "Knowledge Acquisition and Problem Solving", CS 785, George Mason University, Fall 2001.

Tecuci, G., Boicu, M., Ayers, C., and Cammons, D. Personal cognitive assistants for Military Intelligent Analysis: Mixed-Initiative Learning, Tutoring, and Problem Solving. in *First International Conference on Intelligence Analysis*, (McLean, VA, 2005).

Tecuci, G., Boicu, M., Bowman, M., and Marcu, D. An Innovative Application from the DARPA Knowledge Bases Programs: Rapid Development of a Course of Action Critiquer *AI Magazine*, AAAI Press, 2001, 43-61.

Tecuci, G., Boicu, M., Bowman, M., Marcu, D., Shyr, P., and Cascaval, C, An Experiment in Agent Teaching by Subject Matter Experts, in *International Journal for Human-Computer Studies*, pp. 583-610, 2000.

Tecuci, G.; Boicu, M.; and Marcu, D., Learning Agents Teachable by Typical Computer Users. In *Procedure of the AAAI-2000 Workshop on New Research Problems for Machine Learning*, Austin, Texas, 2000.

Tecuci, G., Boicu, M., Marcu, D., Stanescu, B., Boicu, C., and Barbulescu, M. A Learning Agent Shell for Building Knowledge-Based Agents. In *the Technology Demonstration Session of the 14th International Conference on Knowledge Engineering and Knowledge Management, EKAW 2004*, (Northamptonshire, UK, 2004), Whittlebury Hall.

Tecuci, G. Boicu, M., Marcu, D., Stanescu, B., Boicu, C. and Comello, J. Training and using Disciple agents: A case study in Military Center of Gravity Analysis Domain. *AI Magazine,* 24.4, 2002, pp.51 - 68. AAAI Press, Menlo Park, California, 2002.

Tecuci, G., Boicu, M., Wright, K., Lee, S. W., Marcu, D., and Bowman, M., An Integrated Shell and Methodology for Rapid Development of Knowledge-Based Agents. in *The Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, (Orlando, Florida, 1999), AAAI Press, Menlo Park, CA.

Tecuci, G., Wright, K., Lee, S.W., Boicu, M., Bowman, M., and Webster, D., A Learning Agent Shell and Methodology for Developing Intelligent Agents. in *The AAAI-98 Workshop on Software Tools for Developing Agents*, (Madison, Wisconsin, 1998), AAAI Press, 37-46.

Tenenberg, J. 1987 Preserving consistency across abstraction mappings. In *Procedure of IJCAI-87*, Milan, Italy, 1987 (ed. J. McDermott), pp. 1011–1014.

Tsinakos, A.A., and Margaritis, G. K., Student Models: The transit to Distance Education, http://www.eurodl.org/materials/contrib/2000/tsinakos.html, 2000.

Towne, D., Approximate reasoning techniques for intelligent diagnostic instruction, in *International Journal of Artificial Intelligence in Education*, 1997

Tucker, A., Applied Combinatorics, Third Edition. John Wiley & Sons, Inc., 1995.

Turner, T.E. The Assistment Builder: A tool for rapid tutor development *Computer Science*, WORCESTER POLYTECHNIC INSTITUTE, 2005.

Turner, T.E., Lourence, A., Heffernan, N., Macasek, M., Nuzzo-Jones, G., and Koedinger, K. The Assistment Builder: An Analysis of ITS Content Creation Lifecycle. *The 19th International FLAIRS Conference,* Melbourne Beach, Florida, 2006.

Turner, T.E., Macasek, M. A., Nuzzo-Zones, G. , Heffernan, N. T., and Koedinger, K. The Assistment Builder: A Rapid Development Tool for ITS. in *the 12th Artificial Intelligence In Education,* (Amsterdam, 2005), 929-931.

Waltz, D. Artificial Intelligence: An assessment of the State of the art and Recommendation for Future Direction *AI Magazine*, 1983, 55-67.

Whitley, E.A. Embedding expert systems in semi-formal domains: Examining the boundaries of the knowledge base. *School of Economics and Political Science*-290, 1990.

Winston, P. Learning and Reasoning by Analogy. *Communication of the ACM*, *23* (12). 689-703, 1980.

Zucker, and Jean-Daniel, A grounded theory of abstraction in artificial intelligence. *Philosophical Transactions of the Royal Society B: Biological Sciences,* 2003 July 29; 358(1435): 1293–1309.

CURRICULUM VITAE

Vu Le received his Bachelor of Science in Computer Science from George Mason University in 1997. He worked for Boeing Company as software engineer after graduation. He continued to attend George Mason University graduate program in Computer Science. He received his Master of Science in 1999. Vu Le was employed as software engineer in Science Applications International Corporation (SAIC) in 1999. He worked for Alphatech, Inc (now is BAE Systems) in 2004 as senior software engineer and for Learning Agent Center at George Mason University in 2005 as research instructor.