Widening the Knowledge Acquisition Bottleneck for Intelligent Tutoring Systems

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Doctor of Philosophy
in the
University of Canterbury
by
Pramuditha Suraweera

University of Canterbury
2006

This thesis is dedicated to my dear wife

Abstract

Empirical studies have shown that Intelligent Tutoring Systems (ITS) are effective tools for education. However, developing an ITS is a labour-intensive and time-consuming process. A major share of the development effort is devoted to acquiring the domain knowledge that accounts for the intelligence of the system. The goal of this research is to reduce the knowledge acquisition bottleneck and enable domain experts to build the domain model required for an ITS. In pursuit of this goal an authoring system capable of producing a domain model with the assistance of a domain expert was developed. Unlike previous authoring systems, this system (named CAS) has the ability to acquire knowledge for non-procedural as well as procedural tasks.

CAS was developed to generate the knowledge required for constraint-based tutoring systems, reducing the effort as well as the amount of expertise in knowledge engineering and programming required. Constraint-based modelling is a student modelling technique that assists in somewhat easing the knowledge acquisition bottleneck due to the abstract representation. CAS expects the domain expert to provide an ontology of the domain, example problems and their solutions. It uses machine learning techniques to reason with the information provided by the domain expert for generating a domain model.

A series of evaluation studies of this research produced promising results. The initial evaluation revealed that the task of composing an ontology of the domain assisted with the manual composition of a domain model. The second study showed that CAS was effective in generating constraints for the three vastly different domains of database modelling, data normalisation and fraction addition. The final study demonstrated that CAS was also effective in generating constraints when assisted by novice ITS authors, producing constraint sets that were over 90% complete.

Table of Contents

List of Tables

List of Figures

Acknowledgments

Thank you to my supervisor Associate Professor Tanja Mitrovic, for all her invaluable guidance and making this period of study enjoyable. Thank you to my associate supervisor Dr. Brent Martin, for all your helpful advice and valuable feedback on my thesis.

I am grateful for the fruitful discussions with the members of the Intelligent Computer Tutoring Group, especially Jay, Nancy, Konstantin, Moffat, Amali and Nilufar.

Thank you also to all the participants involved in evaluating the different versions of the authoring system.

Finally, I thank my wife Natashka for her support and encouragement during this period. I also thank my parents and my brother for all their support.

This research was supported by a University of Canterbury Doctoral Scholarship.

Chapter I

Introduction

Intelligent Tutoring Systems (ITS) are effective educational programs that assist students in their learning. A number of empirical evaluation studies [Koedinger, Anderson, Hadley & Mark 1997, Mitrovic & Ohlsson 1999] have demonstrated that ITSs are effective tools for teaching students with possible learning gains between 1 and 2 standard deviations in comparison to traditional classroom-based teaching. The main reason for the success of ITSs is their ability to customise pedagogical support to each individual student, similar to a human one-on-one tutor. ITSs keep an up-to-date model of the student's knowledge and provide pedagogical support based on its domain model, which is a formal representation of the domain. The task of building a domain model is a difficult process and it requires much time and effort [Murray 1997]. This difficulty has imposed a major bottleneck in producing ITSs.

Constraint based modelling (CBM) [Ohlsson 1994] is a student modelling approach that eases the knowledge acquisition bottleneck to some extent by using a more abstract representation of the domain compared to other commonly used approaches [Mitrovic, Koedinger & Martin 2003]. However, building a knowledge base for CBM still remains a major challenge. This process requires multi-faceted expertise, such as knowledge engineering, programming and the domain itself. It also consumes a major portion of the time required to build a constraint-based tutoring system. The goal of this research is to reduce the knowledge acquisition bottleneck for CBM tutors while opening the door for domain experts with little or no programming expertise to produce ITSs. In this research, we investigated authoring support for composing a domain model for constraint-based tutoring systems. We present an authoring system, named CAS, that reduces the workload for composing domain models. It also enables domain experts with little or no programming and knowledge engineering expertise to produce constraint bases.

This introductory chapter presents a high-level overview of the thesis. Intelligent Tutoring Systems are introduced in Section 1.1. The central component of ITSs, the domain model is introduced in Section 1.2. The knowledge acquisition bottleneck, which is the central problem faced by ITS authors is discussed in Section 1.3. Section 1.4 proposes a solution: an authoring system for producing domain models with the assistance of a domain expert. Finally, a guide to the rest of the thesis is outlined in Section 1.5.

1.1 Intelligent Tutoring Systems

Educational software systems came to light over three decades ago. These initial systems [Last 1979, Ayscough 1977], called Computer-Aided Instruction (CAI) systems, presented the same educational material to all users. Users of these systems had some control over the navigation within the curriculum. Later systems organised feedback into blocks called frames. The feedback presented in these systems was dependant on the solution to a multiple choice question chosen by the student. These systems achieved only modest gains over classroom-based teaching.

Students learn by applying skills or "learning by doing". One-on-one human tutoring is extremely effective because the human tutor is able to scrutinise the student's solution, identify their misconceptions and provide explanations [Bloom 1984]. While providing explanations, human tutors also take into account the student's strengths and weaknesses. They customise their explanations to the student's knowledge level. CAI systems, on the other hand, present the same feedback to all students regardless of their capabilities. They do not take into account the student's emerging knowledge.

CAI systems are also unable to target the learning material to a particular audience. This may lead to frustration among the students as they may either be presented with content they already know. The worse case scenario is when the system incorrectly assumes that the students already know content. The systems may present problems at the end of a topic to ascertain the competency level of the student. However, the system may make invalid assumptions about the competency of the student and offer problems that the student may not be able to solve. Furthermore, as the system is unaware of how far the student has misunderstood the concepts, it cannot provide supplementary exercises.

Researchers have been exploring ways of simulating human tutors, to achieve the same level of effectiveness as in human one-on-one tutoring. The resultant software systems are called Intelligent Tutoring Systems (ITS). ITSs customise their feedback to the individual by dynamically reasoning about the student. They possess the flexibility to identify an array of correct solutions rather than a single ideal solution. This enables the system to respond to the unique behaviour of individual students.

ITSs have been developed for a variety of domains including mathematics, programming and vocabulary. There have been many successful ITSs developed, such as PACT Algebra tutor [Corbett, Trask, Scarpinatto & Hadley 1998], Andes Physics Tutor [VanLehn, Lynch, Schulze, Shapiro, Shelby, L., Treacy, Weinstein & Wintersgill 2005], SQL-Tutor [Mitrovic 1998*b*] etc. ITSs have also been developed for teaching design tasks such as database modelling [Suraweera & Mitrovic 2002] and object-oriented design [Baghaei & Mitrovic 2006*a*].

1.2 Domain Knowledge

The adaptive nature of ITSs enable them to be far more effective in student learning than CAI systems. This is made possible by their deep model of the domain and a model of the student. The model of the domain (i.e. domain model) represents the subject being taught in a way that can be used for reasoning. The model of the student (i.e. student model) is used to keep track of what parts of the domain model the student does and does not know.

The domain model represents the subject matter of an ITS as a dynamic model that governs the system's reasoning process. It has the ability to identify a range of correct solutions rather than a single idealised solution, and is able to infer the student's domain knowledge. This enables an ITS to dynamically generate a path through the domain in order to respond to the unique behaviour of each individual student. The domain model also supports important pedagogical actions such as generating feedback and selecting problems.

The representation used for modelling a domain depends on the student-modelling technique used. Constraint-based modelling [1994] (CBM) is a student-modelling technique introduced by Ohlsson from his theory of "learning from performance errors" [Ohlsson 1996]. He proposes that we often make mistakes when performing a task, even when we have been taught the correct way to carry it out. He asserts that this is because the declarative knowledge that we have learnt is not internalised in our procedural knowledge. We make mistakes due to the large number of decisions that have to be made. However, by practising the task and identifying mistakes by ourselves (or someone else), our procedural knowledge is modified to incorporate the rules that have been violated. The number of mistakes decline over practice as the declarative knowledge gets internalised.

Unlike in other popular student-modelling techniques, such as model tracing [Anderson, Corbett, Koedinger & Pelletier 1996], that compare the set of actions carried out by the student against the system's correct set of actions, CBM is not interested in the path followed by the student. CBM is only interested in the state that the student is currently in. This is supported by the fact that correct solutions cannot be arrived at traversing a problem state that violates the principles of the domain.

A number of tutoring systems have been developed using constraint-based modelling by the Intelligent Computer Tutoring Group at the University of Canterbury. They cover a wide range of domains, demonstrating the expressibility of constraints. Constraint-based tutors have been implemented for programming-type domains such as the database query language SQL [Mitrovic 1998*b*], design tasks such as database modelling [Suraweera & Mitrovic 2002], and procedural tasks such as database normalisation [Mitrovic 2002]. All constraint-based tutoring systems have been implemented as problem-solving environments. The system evaluates solutions composed by students and provides feedback.

1.3 The Problem: Authoring Domain Knowledge

The task of building an Intelligent Tutoring System is difficult and requires much time and effort. The majority of the effort is consumed in encoding the domain knowledge in the chosen student-modelling representation [Murray 1997]. Constraint-based modelling somewhat eases the knowledge acquisition bottleneck by using a more abstract representation of the domain, compared to other commonly used student modelling approaches [Mitrovic et al. 2003]. However, building a set of constraints still remains a major challenge. For example, SQL-Tutor contains over 700 constraints, each taking over an hour to develop [Mitrovic 1998*a*]. Therefore, the task of composing the knowledge base of SQL-Tutor would have taken over four months to complete.

Typically, a domain model is a result of a collaborative effort between a knowledge engineer, an Artificial Intelligence (AI) programmer, and a domain expert. The knowledge engineer has to interact with the domain expert to extract the domain knowledge and encode it in some formal notation. The knowledge engineer and the programmer collaborate to produce the final domain model, encoded in the format required for an ITS. Although domain experts are fairly common, knowledge engineers and AI programmers are a rare commodity, which also adds to the difficulties in producing domain models. It is very rare to find individuals with all three facets of expertise.

The logical solution to the knowledge acquisition bottleneck is to provide domain experts with authoring systems that enable them to produce domain models with minimal training. Research attempts at developing such authoring systems that automatically produce knowledge for ITSs have met with limited success. Although several authoring systems have been developed[1], they have only focussed on acquiring procedural knowledge and fail to acquire knowledge required for non-procedural tasks. Consequently, these systems cannot be used for design-type domains that do not have a strict problem-solving procedure.

The majority of existing authoring systems have focused on certain types

[1] Examples of authoring systems include CTAT [Koedinger, Aleven, Heffernan, McLaren & Hockenberry 2004, Jarvis, Nuzzo-Jones & Heffernan 2004], Disciple [Tecuci & Keeling 1999] and Demonstr8 [Blessing 1997]

of domains. For example, Demonstr8 is effective in acquiring knowledge for algebraic domains, but is restricted to only such domains. Even authoring tools developed for acquiring knowledge required for simulated domains, such as KnoMic [van Lent & Laird 2001], are restricted to the environment it was developed for.

1.4 A Solution: An Authoring System that Generates Constraint Bases with the Assistance of a Domain Expert

The aim of this research is to reduce the time and effort required to produce domain models for constraint-based ITSs and empower domain experts to build such domain models. In pursuit of this goal, an authoring system capable of producing a domain model with the assistance of a domain expert (called CAS) was developed. Unlike previous authoring systems, CAS has the ability to acquire domain models for non-procedural as well as procedural tasks. CAS was developed to generate knowledge required for constraint-based tutoring systems, due to the abstract representation of constraints that assists in easing the knowledge acquisition bottleneck to some extent.

CAS enables domain experts to produce complete web-based ITSs in conjunction with WETAS [Martin 2002, Martin & Mitrovic 2002*a*], an authoring shell that provides all the domain independent functionality required by an ITS. As WETAS provides all the domain-independent functionality, CAS can be used to generate the required domain-dependent components. This would result in a significant reduction in the time and effort required for producing ITSs.

CAS has been developed with the goal of minimising the expertise in knowledge engineering and programming required for creating an ITS. It is designed to hide the details of constraint implementation such as the constraint language and constraint structures. The user is required to model an ontology of the domain and provide example problems and their solutions. CAS uses machine learning techniques to reason with the information provided for producing constraints. Although implementation is hidden from novices, experts can directly modify the generated constraints using the provided tools.

Authoring knowledge using CAS is a semi-automated process that requires the assistance of an expert of the domain. The domain expert initiates the process of producing a domain model by composing a model of the domain as an ontology. Then the domain expert has to model the structure of solutions for problems in the domain. CAS's constraint generators use the ontology and the solution structure to produce constraints that ensure syntactic validity of solutions. The domain expert is also required to provide sample problems and their solutions, which are used by the constraint generators to produce constraints. These constraints are used to verify that a solution for a problem composed by a student is semantically equivalent to its correct solution. Finally, the author has to be involved in validating the generated set of constraints.

The effectiveness of CAS was evaluated in three studies, that produced promising results. The first evaluation revealed that the task of creating a model of the domain as an ontology assists even in the process of manually composing constraints. A second study evaluated the system's effectiveness in generating constraints, where CAS was used to generate constraints for the three vastly different domains, namely database modelling, data normalisation and fraction addition. Analysis of the generated constraints revealed the generated constraint sets were over 90% complete. The final study demonstrated that CAS was also effective in generating constraints when assisted by novice ITS authors, producing constraint sets that were over 90% complete. It also confirmed that CAS dramatically reduced the total effort required to produce constraints.

1.5 Guide to the Thesis

Chapter 2 provides the context of this research by introducing Intelligent Tutoring Systems. In particular, it outlines the typical architecture of ITSs and describes the two popular domain modelling techniques of model tracing and constraint-based modelling. This chapter also includes a section on comparing the strengths and weaknesses of the two modelling techniques. Details of a survey of currently available domain knowledge authoring tools is provided in Chapter 3, including their strengths and weaknesses. As the task of modelling concepts of the domain is an integral part of authoring domain models, a section outlining a collection of tools available for modelling domain concepts is included in this chapter. Chapter 4 describes CAS, the authoring system developed as part of this research. It outlines the authoring process and the system's architecture. It includes examples to illustrate the outcomes of the different phases involved in authoring. The extensive evaluations conducted to evaluate the effectiveness of CAS and their results are presented in Chapter 5. Finally, Chapter 6 outlines the conclusions and presents future directions for research in this area.

Chapter II

Intelligent Tutoring Systems

Researchers have been exploring ways of utilising computers as a means of enhancing the efficiency of learning for over three decades. The first Computer Aided Instruction (CAI) systems [Last 1979, O'Shea & Self 1983, Ayscough 1977] presented the educational material to students in a static form, where every student received the same material but had some control over the navigation through the curriculum. Later systems included "branching" where their responses dependent on student's answers. Their feedback was organised into blocks of information called frames, which define both the topic and the feedback to be presented. For example, if the student provides correct answers for a set of questions, the next frame in the sequence is presented; if the student could not answer correctly then an alternative screen is presented. These sequences are static and predefined. These systems could only provide questions that required a limited set of solutions, such as 'yes'/'no', multiple choice or numerical solutions.

Although CAI systems managed to achieve modest gains over traditional classroom based teaching [Kulik, Kulik & Cohen 1980], they failed to match the effectiveness of human one-on-one tutoring[1]. With the view of achieving the success of human one-on-one tutoring, researchers have been exploring ways of simulating human tutors. These systems, called Intelligent Tutoring Systems (ITS), customise their feedback to the individual student by dynamically reasoning about the student's knowledge.

The subject matter of ITSs is represented as a dynamic model with a set of rules which governs the way the system reasons. They have the ability to identify an array of correct solutions rather than a single idealised solution.

[1] Human one-on-one tutoring can enhance students learning up to two standard deviations [Bloom 1984]

They are also capable of approximating the student's pedagogical knowledge. This enables ITSs to dynamically generate their own path through the domain knowledge in order to respond to the unique behaviour of individual students.

ITSs have been developed for numerous domains using a variety of approaches. Some tutors have been developed for individual learning, whereas others have focused on collaborative learning. Model-tracing tutors [Anderson et al. 1996] provide problem-solving environments with rich feedback to the learner. Simulation-based tutors [Alexe & Gescei 1996, Munro, Johnson, Pizzini, Surmon, Towne & Wogulis 1997] guide learners in simulation environments to perform tasks of the domain. Collaborative tutors [Dillenbourg & Self 1992] on the other hand, try to facilitate positive interaction between learners by encouraging participation, supporting collaboratively solving problems and encouraging tutoring between peers.

In this research we mainly focus on tutoring systems similar to model tracing tutors that support learning by solving problems. These systems consist of a problem-solving environment where the student solves a given problem and receives rich feedback as they progress. These systems are designed to support individual students learning at their own pace while receiving customised feedback.

The following section outlines the typical architecture of an Intelligent Tutoring System. It details the general features of the four main components of an ITS: domain module, student modeller, pedagogical module and interface. Since the focus of this research is on domain models, the remaining sections concentrate on the two popular domain modelling techniques: model tracing and constraint-based modelling. Both sections include details of the psychological theories that form the basis for the domain modelling techniques and provides a selected set of example tutoring systems. A brief comparison of the two modelling techniques is also included.

2.1 ITS Architecture

The architecture of a typical ITS consists of a domain module, student modeller, pedagogical module and an interface (see shown in Figure 2.1). Typ-

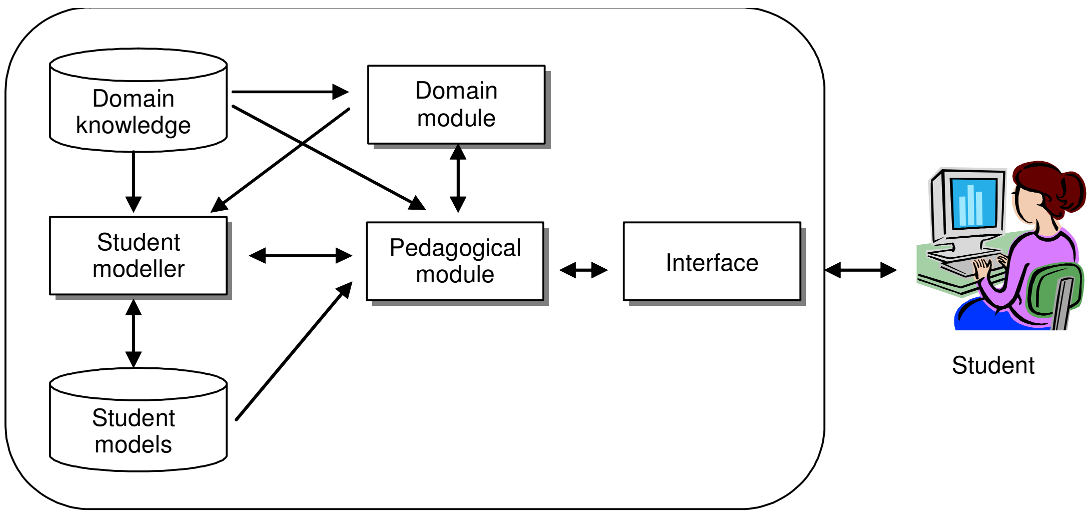


Figure 2.1: ITS Architecture

ically, these four components combine to provide a customised learning experience for the student.

As shown in the diagram, the student interacts with the system through the interface. The interface is typically a problem-solving environment. The system evaluates the student's attempt and provides customised feedback. This evaluation is triggered either by the system or the student depending on the chosen pedagogical approach. The domain module contains the domain knowledge required to compare a solution composed by the student against the correct solution contained in the system. The correct solution is compared against the student's solution by the student modeller using the domain knowledge. In turn the student model is updated by the student modeller to reflect the student's new state of knowledge. Finally, the pedagogical module either selects feedback to be presented to the student or, in the event of the problem being solved correctly, a new problem that best suits the student's knowledge level is selected. A detailed account of each component of a typical ITS is presented in the next four sections.

2.1.1 Domain Module

The domain module contains all the required domain knowledge (facts and rules of the domain) to validate a student's attempt. The nature of the domain knowledge may vary. Some domain models encode expert knowledge required for solving problems. The extent of encoded knowledge could range from expert knowledge required to solve the problems to a subset of expert knowledge required for the purpose of teaching. In certain cases, the domain module is capable of generating the correct solution according to the problem solving path chosen by the student. The domain model of PACT Algebra tutor [Corbett et al. 1998] is an example of a system that generates the solution path based on the student's actions. Conversely, the domain module may contain the domain knowledge necessary to validate a student's attempt by comparing it to the correct solution stored in the system. The domain model of SQL-Tutor [Mitrovic 2003*a*] is capable of validating and providing feedback on a solution composed by a student by comparing it to the system's stored ideal solution.

There are also domain models used for teaching that contain knowledge capable of solving problems using methodologies different from experts. The Guidon system [Clancey 1982] encoded knowledge in a form that was needed for learning medical diagnosis. In addition to expert problem solving rules, it included knowledge such as procedural knowledge about how to use the rules of problem solving. It also took into account the rules that helped students remember a particular rule.

Domain knowledge can be represented in various ways such as frames, production rules, and constraints. Since domain knowledge is used to solve problems as well as to explain solutions, the mode of representation should be powerful enough for solving problems and be comprehensive enough for providing pedagogical explanations.

Knowledge in ITSs have been represented in either black-box form or glass-box form. In earlier systems knowledge was represented in the black-box form where the problem-solving methodology was undisclosed to the student. Although these systems were able to determine the solution for a particular problem, they were unable to provide the rationale behind it. The system by the name SOPHIE I [Brown, Burton & Bell 1975] is a classic example, developed for the domain of electric circuits. SOPHIE I could calculate an electrical measurement at any point in a complex circuit, but is unable to explain the reasoning behind the reported values. As such explanations are extremely important for learning domain concepts, this type of system is more likely to encourage shallow learning.

More recent systems have adopted the glass-box form of knowledge representation. Here the knowledge is represented in such a way that it closely resembles the human capability. These systems offer richer explanations to the student. For example, the model-tracing tutors developed by the ITS researchers at the Carnegie Mellon University trace the students' actions within a tree of correct and typical incorrect solutions. This tree of possible actions is composed by observing typical students solving problems in the domain. The model-tracing tutors force students to follow the correct solution path by providing hint messages when they divert from the correct solution path. Constraint-based tutors developed by the University of Canterbury researchers use a set of constraints to describe the underlying concepts of the domain. The constraints are used to identify domain concepts that are violated by the student's solution and provide feedback messages.

2.1.2 Student Modeller

The student modeller performs two functions. It evaluates the student's solution and dynamically maintains a model of the state of the student's knowledge and skill level. The representation called the student model is developed by deducing the student's knowledge level through their interactions with the system and the quality of their solutions. The student model contains an estimation of the student's long-term knowledge (e.g. estimation on student's domain mastery) and short-term knowledge (e.g. the mistakes made by the student during their last attempt).

The student model should also include the system's opinion of strengths and weaknesses of the student, as well as their misconceptions. Maintaining a good approximation of the student's knowledge is essential as all pedagogical decisions depend on the information found in the student model. The pedagogical decisions taken by the system is a direct reflection of the quality of the student model.

ITS researchers have developed a variety of student-modelling techniques for modelling both the short-term and long-term knowledge of the student.

Model tracing (MT) [Anderson et al. 1996] and constraint-based modelling (CBM) [Ohlsson 1994] are two important short-term student modelling techniques. MT focuses on representing procedural knowledge of the domain, whereas CBM deals with declarative knowledge.

The well grounded methods for modelling long-term student knowledge include overlays [Holt, Dubs, Jones & Greer 1994] and stereotypes [Rich 1989]. Overlay models represent the knowledge of the student as a subset of the domain expert's knowledge. The initial state of an overlay model assumes that the student has no knowledge. The model is populated as the student interacts with the system. Stereotype modelling also models the student's domain knowledge with respect to the desired knowledge. It overcomes the problem of starting from an empty model by classifying the student into levels of expertise, usually based on the pre-test score.

2.1.3 Pedagogical Module

The pedagogical module acts as the driving engine of the tutoring system. It decides what to present to the student. The decisions are derived using information available in the student model and the domain model. The pedagogical actions, initiated by the pedagogical module, range from low-level decisions (such as deducing the difficulty of a problem) to high-level decisions (such as selecting the next topic for the student).

Pedagogical strategies vary from traditional didactic methods to more informal discovery-oriented learning. The didactic approach involves instructing the learner, in which the tasks are strongly goal-oriented [Anderson 1993]. Tutoring systems that adopt the didactic approach initiate and control student activity. Novices find this method of teaching effective as they require close guidance. However, more knowledgeable students find such systems restrictive and not sufficiently challenging. In contrast, discovery-oriented learning involves learning from experience [Lesgold 1987, Shute, Glaser & Raghavan 1989]. It promotes new knowledge to be constructed through self-explanation, induction and ontology based on concrete experiences with the concepts and capabilities possessed by the learner. Although this approach is effective for knowledgeable students, novices may take a long time to make the discoveries that attribute to achieving the pedagogical objectives. Researchers have been exploring ways of guiding the discovery process with the view of increasing the efficiency of the learning process.

2.1.4 Interface

The interface is the mediator between the student and tutoring system. Typically, the students use the interface to solve problems presented to them and the system presents feedback to students through the interface.

There has been considerable research effort towards developing efficient interfaces. In this respect a number of characteristics that would increase the student's perception of the system and enhance learning efficiency have been identified. For example, reducing the student's working memory load is an important characteristic that leads to improved learning efficiency. All parts of the problem which are not part of the teaching focus should be made available to the student through the interface. The interface should also be able to visualise the goal structure for solving the problem in order to assist the student towards task completion. Since the student motivation is an essential ingredient of the success of the system, the interface should also be able to motivate students.

2.2 Model Tracing

In this research we are primarily interested in domain models. Model Tracing is a well grounded domain modelling technique based on Anderson's ACT-R [Anderson et al. 1996] theory. Tutoring systems that are implemented based on this theory are called model-tracing tutors or cognitive tutors [Corbett & Anderson 1995]. Their domain model is runnable and is capable of mapping out all the possible paths a student may follow in order to solve a problem.

2.2.1 ACT-R Theory

The ACT-R cognition theory claims that there are two long-term memory stores: declarative memory and procedural memory. Anderson contends that a person needs the relevant procedural knowledge in order to perform a task, and he/she needs to know the underlying declarative knowledge before being able to perform the task. According to this theory, learning is a three step process: initially students must acquire declarative knowledge, then they must transform it into procedural knowledge and finally, in order to internalise the knowledge they must practice the task several times. Although we can perform the tasks of which we have forgotten the declarative knowledge that led to acquiring the skill, Anderson asserts that we must have had the exposure to the declarative knowledge at some stage.

According to the ACT-R theory, the declarative knowledge (which includes factual knowledge that the student uses such as theorems in the mathematical domain) is represented as 'chunks'. Procedural knowledge (which includes goal-oriented knowledge such as how to apply a mathematical theorem) is represented as 'production rules'. Incorrect knowledge or misconceptions are represented as 'buggy rules'.

2.2.2 Production Rules

The principal claim of the ACT-R theory is that cognitive skills are realised by production rules. Accordingly, tutoring becomes the process of transferring production rules from the system to the student, so the students are tutored specifically on productions.

Production rules are modelled as goal-oriented if-then rules. As an example, consider the set of production rules for computing the third angle of a triangle when two of the angles are known. The theorem governing the three angles of a triangle asserts that the sum of the three angles is 180^o. The set of production rules, as shown in Figure 2.2, consists of two if-then rules. The first rule checks whether the current goal is to ascertain all three angles of a triangle in which two of the angles are already known. If the first condition is satisfied, the rule sets a sub-goal to calculate the unknown angle of the triangle. Since production rules are procedural in nature, the second production rule triggers only when the goal is to calculate the third angle of the triangle. In other words, the second rule will only be activated after the first rule has been completed. The second rule specifies the action of determining the value of the third angle, calculated as $(180^o - \theta_1 - \theta_2)$.

```
IF the goal is to find the sizes of all three angles ($\theta_1$, $\theta_2$
    and $\theta_3$) of a triangle and two ($\theta_1$ and $\theta_2$) are known
THEN set a sub-goal to calculate $\theta_3$ of triangle

IF the goal is to calculate $\theta_3$ of triangle
THEN set a sub-goal to write out $\theta_3$, where $\theta_3$ is $180^o -$
    $\theta_1 - \theta_2$
```

Figure 2.2: Set of Production Rules for Calculating the Third Angle of a Triangle

The set of if-then rules used to calculate angles of a triangle (itemised in Figure 2.2) should also include buggy rules to detect incorrect solutions. A simple buggy rule which identifies such an erroneous solution, as illustrated in Figure 2.3, can be added. The rule catches the popular misconception where students assume that the two base angles of a triangle are always equal. As a consequence, they determine θ_3 to be of the same value as θ_2.

```
IF the goal is to calculate $\theta_3$ of a triangle
  AND NOT $\theta_3 = \theta_2$
THEN set a sub-goal to write out $\theta_3$, where the value of
    $\theta_3$ is $\theta_2$
```

Figure 2.3: Buggy Rule for Calculating the Third Angle of a Triangle

Composing the set of production rules that make up the domain model of a cognitive tutors is a labour-intensive and time-consuming process. Anderson and co-workers reported the estimated time to produce a single production rule was ten hours or more [Anderson et al. 1996]. Furthermore, Koedinger and co-workers estimated that an average of 200 hours of development time was required to produce one hour of educational content [Koedinger et al. 2004]. Domain models for complex domains may be composed of hundreds or perhaps thousands of production rules. Using the ten-hour estimate, building a domain model for a domain such as SQL would take years. The effort required for building the domain model for cognitive tutors posess a serious bottleneck to building tutors for complex domains, where the need for ITSs is at its greatest.

The need to compose a library of typical misconceptions as a collection of buggy rules is also a major limiting factor in the development of cognitive tutors [Soloway, Guzdial & Hay 1994]. Furthermore, a bug library composed for a particular group of students may not be appropriate for another group [Mitrovic et al. 2003], as different groups of students tend to have varied misconceptions and make different mistakes. Thus, composing a robust library of bugs is an extremely difficult task, if not impossible. The task of identifying typical misconceptions is tedious and laborious, since the space of incorrect knowledge is limitless.

2.2.3 Model Tracing Tutors

Model-tracing tutors are able to solve problems in the domain and trace the solution path of a student through a complex problem-solving space. The tutors provide immediate feedback on the student's problem-solving performance, in addition to assisting the student to follow the correct problem-solving path. Additionally, they have an explicit goal-structure of the problem, in order for the students to master the abstract as well as the concrete skills of the domain.

Model-tracing tutors have been developed for a number of domains. For example, the PACT Algebra tutor [Corbett et al. 1998] and PACT Geometry tutor [Aleven & Koedinger 2000] are designed to teach high school students to solve algebraic and geometric problems respectively. They are able to trace the student's solution path and offer feedback on their performance. These tutoring systems have been coupled with knowledge tracing to model the student's long-term knowledge in the domain. Knowledge tracing is a long-term student modelling technique.

2.3 Constraint-based Modelling

Constraint-based modelling (CBM) [Ohlsson 1994] is a method for domain and student modelling proposed by Ohlsson, based on his psychological learning theory. Both the learning theory and domain modelling method vastly differ from Anderson's ACT-R theory. This section outlines the theory behind constraint-based modelling and introduces its constraints. A brief description of selected constraint-based tutoring systems and particular examples of their constraints are also presented in here.

2.3.1 Learning from Performance Errors

CBM is based on Ohlsson's theory called "learning from performance errors" [Ohlsson 1996]. This asserts that we learn when we catch ourselves (or are caught by a third party) making mistakes. Moreover, the theory says that we often make mistakes even though we know what to do, as a result of having too many things to consider and that we are unable to make the correct decision because we are overloaded. In other words, though we may posess the required declarative knowledge, there may be too many possibilities to consider in a given situation. Hence, in order to be able to make the correct choice, we have to learn how to apply the declarative knowledge in addition to learning it.

Ohlsson uses constraints to represent the application of declarative knowledge items to a particular situation. Each constraint consists of two clauses: a relevance condition and a satisfaction condition. The relevance condition identifies the appropriate problem states and the satisfaction condition identifies states where this piece of knowledge has been correctly applied. A generic constraint can be represented as:

IF <relevance constraint> is true
THEN <satisfaction condition> has to be also true

Consider a person starting to drive a car in a new country. Among a number of factors, he/she has to consider whether the road system is designed for left-hand side driving or right-hand side driving. For example, consider a person from New Zealand (left-hand side driving) starting to drive in the United States (right-hand side driving). When the driver first enters the road, he has to make a decision whether to steer the car on to the left-hand side of the road or the right-hand side. A constraint for this situation can be encoded as follows:

IF driving in the United States
You better be on the right-hand side of the road

The constraint is specified as a predicate. It is relevant for all occurrences of driving in the United States. The analogous production rule would contain a goal as the 'if' part and an action for the 'then' part of the rule. A production rule may read, "IF entering a road in the United States THEN steer to the right-hand side".

According to Ohlsson's theory, a driver who is new to the conditions, but is knowledgeable about the right-hand driving rule, may still steer the car to the left. They will internalise the constraint only after catching themselves violating it or by being reminded by someone else. However, a driver who is accustomed to right-hand driving, may "intuitively" steer the car to the right hand side of the road as a result of repeated application of the constraint. The constraint has been internalised as procedural knowledge in the case of an expert driver.

2.3.2 CBM in ITS

The domain model of constraint-based tutors is represented by a set of constraints, where each constraint represents a pedagogically significant set of problem states. If a constraint is relevant to the student's solution and its satisfaction condition is violated, the principle represented by the constraint needs to be taught to the student. Violation of a constraint by a student means he/she has a misunderstanding about a principle of the domain that needs to be corrected. Misconceptions of the student are corrected with feedback actions. Consequently, a basic constraint includes the following components:

- Constraint identifier
- Relevance condition
- Satisfaction condition
- Feedback

Since the space of incorrect knowledge is much greater than correct knowledge, a domain is represented in CBM as a set of constraints on correct solutions. The set of constraints have the ability to identify correct solutions from the space of all possible solutions. Constraints represent only declarative knowledge of the domain, such as theorems. Incorrect solutions can be identified as solutions that contravene the semantic and syntax rules of the domain.

Consider the same example presented in Section 2.2.3, the theorem governing the sum of the three angles of a triangle in the domain of geometry. Suppose the applicability condition (C_r) of the hypothetical constraint is the condition where two (θ_1 and θ_2) of the three angles (θ_1, θ_2 and θ_3) are known. The C_r would match any triangle with two known angles. The C_s that defines the correct solution is of the form $\theta_3 = 180^o - (\theta_1 + \theta_2)$. In other words, C_s defines that the unknown angle (θ_3) should be the result obtained by subtracting the sum of the two known angles from 180^o. The relevance condition (C_r) and the satisfaction condition (C_s) of the constraint are illustrated in Figure 2.4.

C_r: The student computes the third angle (θ_1) of a triangle where two angles (θ_1 and θ_2) are known
C_s: The third angle (θ_3) should be equal to $180^o - \theta_1 - \theta_2$

Figure 2.4: A Constraint Based on the Third Angle of a Triangle

A constraint can be represented as a pair of patterns, where each pattern is a list of elementary propositions combined with negation, conjunction and/or disjunction. Alternatively, constraints can be implemented as pairs of functional predicates. For example, consider the example illustrated in Figure 2.4. The C_r may be implemented as $knownAngles(triangleIdentifier) = 2$, where $knownAngles$ is a function that takes the identifier of a specific triangle as its argument and returns the number of known angles of the triangle.

A complete constraint-base of a tutoring system accounts for all significant pedagogical states of the domain. The important principles of composing a solution for problems presented by the system are encoded as syntax constraints. They ensure that the solutions composed by students are syntactically valid. The correctness of a solution to the given problem is verified by semantic constraints. They ensure that the solution composed by the student contains all the required items and nothing extra. The semantic constraints achieve this by comparing the student's solution (SS) against the system's solution to the problem (referred to as the ideal solution). The ideal solution (IS) can be either generated by a problem solver or be stored along with the problems. Typically only a single ideal solution is stored. In cases where the ideal solution is stored, the constraints have to be sufficiently flexible to permit correct solutions arrived at using alternate solution paths.

2.3.3 Constraint-based Modelling and Model Tracing: A Comparison

A key assumption in CBM is that diagnostic information is in the problem state rather than the path taken to arrive at the problem state. This assumption is supported by the fact that a correct solution cannot be arrived at by traversing a problem state that violates the principles of the domain. Unlike in MT, where all possible correct paths of solving a problem should be outlined, the domain model in CBM is only interested in pedagogically significant problem states. Consequently, the domain model of a constraint-based tutor is far simpler than that of a cognitive tutor.

The domain model in model tracing describes the student's knowledge. It includes correct knowledge as well as common misconceptions in the form of buggy rules. In contrast, CBM only focuses on correct knowledge. This ensures that the space of knowledge modelled in CBM is considerably smaller than in MT. As a result, the size of domain models in CBM tutors is significantly smaller than that of cognitive tutors.

It is crucial when using model tracing that the domain model be comprehensive or the tutor will not be able to trace the actions of the student. With CBM, the effect of a missing constraint is highly restricted. A missing constraint simply results in failing to identify a particular error. As the constraints are modular in nature, the solution can be analysed with the remaining constraints. This reduces the need for large-scale evaluations about the correctness of the domain model involving domain experts and allows the domain to be incrementally developed by incorporating it in a tutoring system.

The difference between the effort required to compose domain models using the two student modelling techniques is evident in the domain of database modelling. In order to assess a simple database modelling problem, KERMIT [Suraweera & Mitrovic 2004], the constraint-based tutor for database modelling, requires 23 constraints and ideal solutions to the problems. In contrast, a cognitive model requires 25 production rules, 10 general chunks and 30 problem-specific chunks, or a total of 65 elements [Mitrovic et al. 2003]. Moreover, the 30 problem-specific chunks are specific to the problem and cannot be used outside that particular problem. On the other hand, constraints are not problem-specific and they cover a wider range of domains.

It has also been documented that the time required to acquire constraints is significantly less than that of identifying production rules. Mitrovic [1999] reported that the time required to identify a constraint was only 1.1 hours. This is a significant saving in effort compared to the ten or more hours spent on identifying a single production rule [Koedinger et al. 1997].

2.3.4 Constraint-based Tutors

A number of tutoring systems based on constraint-based modelling have been developed by the Intelligent Computer Tutoring Group (ICTG) of the University of Canterbury. They cover a wide range of domains demonstrating the expressibility of constraints. The variety of domains modelled using constraints include domains that contain programming tasks, design tasks and procedural tasks. All tutoring systems have been implemented as practice environments, where students are presented with a problem to be solved. The student is able request assistance from the system at any time during the problem-solving process. Once a problem is complete, the system chooses a problem that best suits the student's competency level.

Constraint-based systems have also been developed to cater for students of a variety of age groups. While the majority of the tutoring systems have been developed for university-level students, there are also constraint-based tutors for younger students. The punctuation tutor (CAPIT) [Mayo, Mitrovic & McKenzie 2000] was developed with the goal of improving the capitalisation and punctuation skills of 10-11 year old school children. LBITS [Martin & Mitrovic 2002*a*] is another constraint-based ITS developed to teach basic English language skills to elementary and secondary school students. It uses a series of "puzzles" such as crosswords, synonyms, and plurals for pedagogy.

A suite of tutoring systems to cover introductory database concepts such as database modelling, database normalisation and database querying have been developed by Mitrovic and co-workers [Mitrovic, Mayo, Suraweera & Martin 2001]. The database suite includes KERMIT [Suraweera & Mitrovic 2002] for learning database modelling concepts, ERM-Tutor [Milik, Marshall & Mitrovic 2006] for practising the process of converting an ER model to a relational schema, NORMIT [Mitrovic 2002] for practising database normalisation and SQL-Tutor [Mitrovic 1998*b*] for learning the SQL database query language. An outline of each of these tutoring systems is presented next.

SQL-Tutor

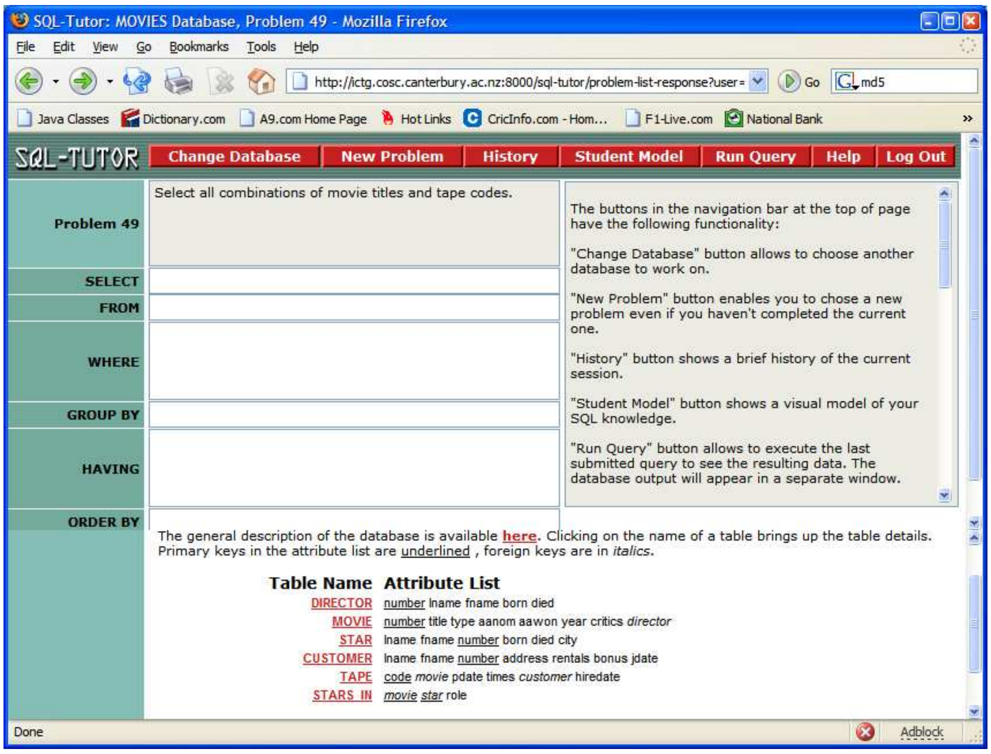
SQL-Tutor [Mitrovic 2003*a*] is the inaugural constraint-based intelligent tutoring system, constructed as a practice environment for students learning SQL, the database query language. Here the students are required to compose an SQL statement that satisfies all the given requirements. It was developed with the goal of assisting university-level students practice composing SQL queries at their own pace with the assistance of the system.

Problems in SQL-Tutor are presented in the form of a set of requirements for which the student has to compose SQL queries. The answer must be provided by populating the clauses of a 'select' statement and the student may request feedback at any time by submitting the solution to the system. Once the solution is submitted, the system evaluates it and provides feedback regarding the state of the solution.

The interface of SQL-Tutor is shown in Figure 2.5. The centre of the interface displays the structure of an SQL statement with six areas for composing the six clauses of an SQL statement: *select*, *from*, *where*, *group-by*, *order-by* and *having*. Students compose an answer by populating these fields. When the student completes a solution or requires assistance from the system, they can click on the "submit answer" button and the system will evaluate the solution composed by the student and provide hints.

The system contains six levels of feedback messages ranging a from simple correct/incorrect message to more detailed hint messages to a complete solution. The level of feedback is automatically increased by the system after each submission. The student also can request more assistance from the system by directly selecting a specific level of feedback.

The system attempts to ease cognitive load on the student by providing scaffolding. The bottom area of the interface outlines the schema of the database the student is working on. This eliminates the need for students to remember the details of the database tables and their attributes.



Select all combinations of movie titles and tape codes.

The buttons in the navigation bar at the top of page have the following functionality:

"Change Database" button allows to choose another database to work on.

"New Problem" button enables you to chose a new problem even if you haven't completed the current one.

"History" button shows a brief history of the current session.

"Student Model" button shows a visual model of your SQL knowledge.

"Run Query" button allows to execute the last submitted query to see the resulting data. The database output will appear in a separate window.

The general description of the database is available here. Clicking on the name of a table brings up the table details. Primary keys in the attribute list are underlined , foreign keys are in *italics*.

Table Name	Attribute List
DIRECTOR	number lname fname born died
MOVIE	number title type aanom aawon year critics *director*
STAR	lname fname number born died city
CUSTOMER	lname fname number address rentals bonus jdate
TAPE	code *movie* pdate times *customer* hiredate
STARS_IN	*movie* *star* role

Figure 2.5: SQL-Tutor Interface (web version)

The knowledge base of SQL-Tutor contains 700 constraints, covering both the syntax and the semantics of SQL. The constraints are problem-independent; they only describe the principles of the domain, and do not directly refer to any particular problem. They are also modular, and are not dependant to each other.

Figure 2.6 illustrates three constraints from SQL-Tutor. Each constraint consists of five components: a number as the identifier, hint message, relevance condition, a satisfaction condition and the SQL clause in focus. The first constraint (constraint 2) ensures that the the *select* clause is not empty. Its relevance condition specifies that the constraint is always relevant and the satisfaction condition checks that the *select* clause is not null. The last component specifies that the constraint focuses on the *select* clause.

Constraint 38 is an example of a simple constraint that checks for the semantic validity of a student's solution. It verifies that, for a problem where the *select* clause of its ideal solution is the symbol '*', the student solution's *select* clause should also contain a '*' or a list of all attributes found in tables named in the *from* clause. By checking for either '*' or list of attributes in the student's solution, the constraint is able to correctly identify equivalent correct solutions by using only a single ideal solution.

The final example in Figure 2.6 (constraint 99), is a relatively complicated constraint which ensures that the SQL keyword 'IS' is used according to the correct syntax. In SQL, the keyword 'IS' can only be used in conjunction with the 'NULL' predicate. The constraint is relevant for solutions which contain a where clause that includes a 'NULL' predicate. The satisfaction condition checks that the 'IS' predicate is followed by either a 'NULL' or a 'NOT'.

KERMIT

KERMIT [Suraweera & Mitrovic 2002] is an intelligent tutoring system for teaching database design using the Entity-Relationship (ER) data model [Chen 1976]. It consists of a problem-solving environment in which students compose ER diagrams that satisfy the given set of requirements. The system is designed to complement classroom teaching, given that the students are familiar with the fundamentals of database theory. The system assists the student by evaluating his/her database schema and providing feedback.

During the problem-solving stage, the student is given a textual description of the requirements of the database that should be modelled using KER-

```
(p 2
   "The SELECT clause is a mandatory one.
     Specify the attributes/expressions to retrieve
     from the database."
   t
   (not (null (select-clause ss)))
   "SELECT")

(p 38
   "The SELECT clause should contain either a *, or the
     names of all attributes of the tables appearing in
     the FROM clause."
   (equalp '("*") (select-clause is))
   (or  (equalp '("*") (select-clause ss))
        (null (set-difference (all-att
                   (find-names is 'from-clause) '())
            (find-names ss 'select-clause)
             :test 'equalp)))
   "SELECT")

(p 99
   "Check the search condition you have specified with
     the IS NULL predicate. Make sure you use the right
     attribute in it. The IS keyword must be preceded
     by an attribute name and may only be followed by
     NOT or NULL. Remember that IS cannot be used in
     the place of the equality operator."
   (and (not (null (where ss)))
        (bind-all ?n (names (where ss)) bindings)
        (match '(?*d1 ?n "IS" ?*d2)
         (where ss) bindings))
   (and (attribute-in-from ss ?n)
        (match '(??p "NULL" ?*d3) ?d2 bindings)
        (member ?p '(nil "NOT") :test 'equalp))
   "WHERE")
```

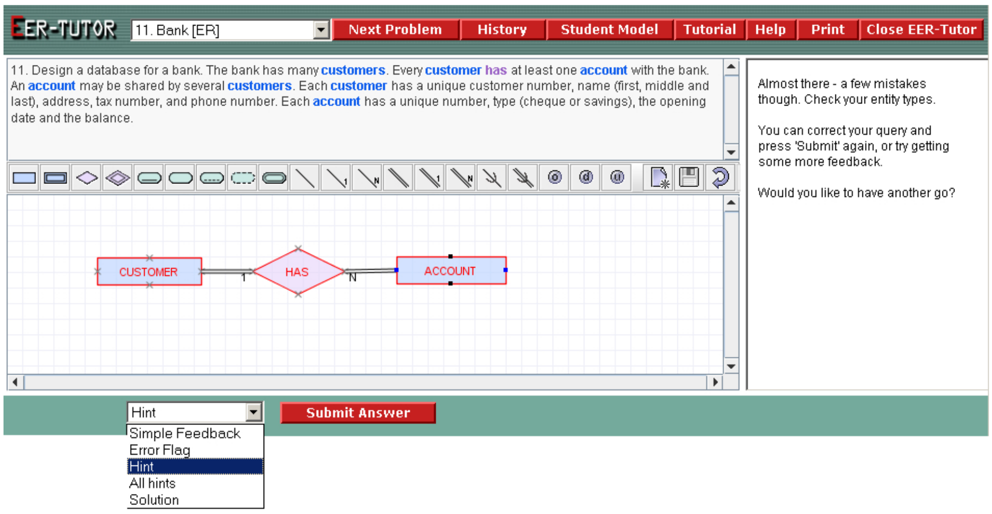
Figure 2.6: A Set of Sample Constraints from SQL-Tutor

MIT's interface, as illustrated in the example given in Figure 2.7. The description of the scenario is available at the top section of the interface and the ER modelling workspace is available below. During the problem-solving process, the student composes an ER diagram that satisfies the requirements using the ER modelling workspace. It provides a drawing tool like interface for composing database models using the ER modelling constructs and a set of connectors. In order to eliminate the need for natural language processing, the student is forced to highlight a word or a phrase that describes each construct in their diagram. This enables the system to reliably determine the semantics of each construct.

Assistance from the tutoring system has to be requested by the student. The student may either request the system to evaluate whether their solution is correct or ask for a hint to proceed further by pressing the 'submit' button. The system then evaluates the student's submission and presents appropriate feedback messages. Since the domain model contains constraints for checking for both the completeness and the correctness of a solution, both incomplete and complete solutions are evaluated using the same procedure. The solutions that do not violate any constraints are identified as correct and complete. On the other hand, solutions that violate constraints can be incorrect, incomplete or both.

KERMIT displays hint messages from either the first violated constraint or hints from all violated constraints depending on the level of feedback requested by the student. It contains five levels of feedback with increasing degree of detail. The initial feedback level simply indicates whether the attempt is correct or not. The *error flag* level indicates the type of construct (entity, relationship or attribute) that contains errors. The *hint* level provides a single hint message from the first constraint that has been violated. The level of feedback is automatically incremented with each submission up to the *hint* level. Hints on all violated constraints can be viewed by selecting the *all hints* feedback level. The student may also request the full solution by selecting the *complete solution* feedback level.

The task of database design is an open ended process. Its outcome is only defined in abstract terms and there is no procedure to be used to find the outcome. A good solution in this domain is identified as an ER schema



11. Design a database for a bank. The bank has many **customers**. Every **customer** **has** at least one **account** with the bank. An **account** may be shared by several **customers**. Each **customer** has a unique customer number, name (first, middle and last), address, tax number, and phone number. Each **account** has a unique number, type (cheque or savings), the opening date and the balance.

Almost there - a few mistakes though. Check your entity types.

You can correct your query and press 'Submit' again, or try getting some more feedback.

Would you like to have another go?

Figure 2.7: Interface of the Web Version of KERMIT

that satisfies all the given requirements and integrity rules of the chosen data model. In order to identify integrities, the student needs to use their own world knowledge to make valid assumptions. There is no single right solution to a particular problem, and often there are several good solutions which satisfy the same requirements.

The constraint base of KERMIT consists of over 200 constraints that ensure that the student's solution is syntactically and semantically correct. It consists of 69 syntax constraints, varying from simple constraints such as "an entity name should be in upper case", to more complex constraints such as "the participation of a weak entity in the identifying relationship should be total". For example, constraint 25 (Figure 2.8) ensures that each weak entity in an ER diagram participates in an identifying relationship. Its relevance condition binds all the weak entities from the student's diagram. The satisfaction condition verifies that the weak entity is connected to an identifying relationship. As each construct in KERMIT is identified by a unique tag, it is used to ensure that a connector (attached a particular weak entity) is also attached to an identifying relationship.

The constraint base contains 138 semantic constraints which compare the student's solution against the system's ideal solution. Constraint 15, outlined in Figure 2.8, is an example of a simple semantic constraint. It ensures that the student's solution contains all the regular entities that exist in the system's ideal solution. The relevance condition binds all the regular entities from the ideal solution and the satisfaction condition checks whether each of them have a corresponding regular entity in the student's solution, using their tags.

Constraint 64 is a relatively complicated semantic constraint that enables KERMIT to identify multiple correct solutions even though it contains only a single ideal solution. It models the fact that multi-valued composite attributes can also be represented as weak entities. Constraint 64 allows the student to model a multi-valued composite attribute found in the ideal solution as either an identical multi-valued composite attribute or a weak entity. The relevance condition contains three tests. The first and second tests ensure that the constraint only binds entities of the ideal solution that contains a corresponding entity in the student's solution. The third test ensures that the binding list only contains entities that contain a multi-valued composite attribute. The satisfaction condition asserts that either the student's solution should also contain a matching multi-valued composite attribute or that it should contain a corresponding entity. The check for whether the corresponding entity is weak is performed in another constraint. This enables the system to provide more focused feedback.

The system was evaluated in an empirical evaluation conducted at the University of Canterbury involving 62 volunteers from students of the introductory databases course. The study compared the learning gains achieved with KERMIT (experimental group) against its cut-down version, which was a tool for composing ER diagrams. The cut-down version did not present any feedback to the student and only allowed students to view the complete solution. The students interacted with the system during a two hour period.

The study showed that the group of students who used KERMIT scored significantly higher in the post-test [Suraweera & Mitrovic 2002]. Conversely there was no statical significance between the pre- and post-test performances of the control group. Furthermore, the difference in pre-test scores between the two groups were insignificant, confirming that both groups initially posessed similar levels of knowledge in ER modelling.

The students using KERMIT spent longer with the system. They also

```
(25
   "Each weak entity type should participate in an
     identifying relationship."
   (match SS ENTITIES (?* "@" ?tag ?label1 "weak" ?*))
   (and (match SS CONNECTIONS
        (?* "@" ?part ?card ?r_tag ?tag ?*))
        (match SS RELATIONSHIPS
            (?* "@" ?r_tag ?label3 "ident" ?*)))
   "weak entity types"
   (?tag))

(15
   "Make sure that you have all required entity types.
     Some regular entity types are missing."
   (match IS ENTITIES (?* "@" ?tag ?l1 "regular" ?*))
   (match SS ENTITIES (?* "@" ?tag ?l2 "regular" ?*))
   "entity types"
   nil)

(64
   "Check whether you have all the multivalued
     composite attributes that belong to entities
     as specified by the problem."
   (and (match IS ENTITIES (?* "@" ?ent_tag ?*))
        (match SS ENTITIES (?* "@" ?ent_tag ?*))
        (match IS ATTRIBUTES (?* "@" ?att_tag
          ?label1 "multi" "composite" ?ent_tag ?*)))
   (or-p (match SS ATTRIBUTES (?* "@" ?att_tag
           ?label2 "multi" "composite" ?ent_tag ?*))
         (match SS ENTITIES (?* "@" ?att_tag ?*)))
   "attributes"
   (?ent_tag))
```

Figure 2.8: A Set of Sample Constraints from KERMIT

rated the usefulness of feedback at a significantly higher level than the control group. Students using the complete system also indicated that they would recommend the system to others. Analysis of student logs revealed that the probability of violating a constraint decreased steadily, closely approximating a power curve.

NORMIT

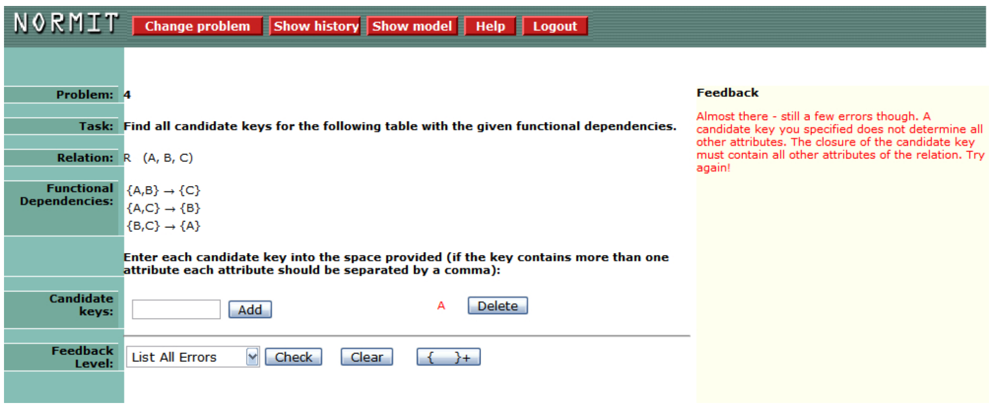
Mitrovic and co-workers developed NORMIT [Mitrovic 2002, Mitrovic 2005*a*] for students to practice solving problems in the domain of database normalisation. Database normalisation is the process of refining relational database schema in order to ensure that all tables are of high quality for the domain of database normalisation [Elmasri & Navathe 2003]. The system was developed to complement classroom teaching where students would learn the necessary conceptual knowledge from lectures and practice problems using the tutoring system.

Database normalisation is a procedural task. Unlike SQL and ER-modelling, problems in this domain have to be solved by following a strict sequence of steps. The sequence of steps to be followed in NORMIT, outlined in Figure 2.9, is fixed by the interface. The student will only see the web page to solve the current step. The student may submit the solution at any time for the system to evaluate it and provide feedback.

1. Determine candidate keys
2. Determine the closure of a set of attributes and prime attributes
3. Simplify functional dependencies
4. Determine normal form
5. If necessary, decompose the table

Figure 2.9: Problem-solving Procedure Adopted by NORMIT

After the student submits a solution, the system evaluates it and provides feedback depending on the chosen level of feedback. The first level of feedback only specifies whether or not the solution is correct and highlights errors in red. A description of the error is provided for *hint* level feedback as illustrated in Figure 2.10. In the instance depicted in Figure 2.10, the student has specified an incorrect candidate key (A) and the system provides a message that contains a general description of the error specifying what general concept of the domain that has been violated. The erroneous part of solution (the candidate key A) is also highlighted. The next level of feedback provides the student with a more detailed message with a hint for how the student could change their solution to satisfy the violated principle. The student may also request the complete correct solution.



Problem: 4

Task: Find all candidate keys for the following table with the given functional dependencies.

Relation: R (A, B, C)

Functional Dependencies:
{A,B} → {C}
{A,C} → {B}
{B,C} → {A}

Enter each candidate key into the space provided (if the key contains more than one attribute each attribute should be separated by a comma):

Candidate keys: A

Feedback Level: List All Errors

Feedback

Almost there - still a few errors though. A candidate key you specified does not determine all other attributes. The closure of the candidate key must contain all other attributes of the relation. Try again!

Figure 2.10: Interface of the Web Version of NORMIT

The domain model of NORMIT contains a total of 82 constraints that check the syntax and semantics of the student's attempt. NORMIT analyses the semantics of a solution by comparing the student's submission to the ideal solution generated by NORMIT's problem solver. As normalisation is a procedural task, not all constraints are relevant for a given step. Each constraint is relevant for only a single step. This is achieved in NORMIT with a variable named *current-task* that records the process of the student. All constraints use this variable to ensure that the constraint is only fired for a particular step in the problem-solving process.

Constraint 8 shown in Figure 2.11 is an example of a simple syntax constraint that ensures that the student has at least made an attempt at iden-

```
(8
   "You need to specify the candidate key(s) for
     the given relation!"
   (equalp (current-task sol) 'candkeys)
   (not (null (candkeys sol)))
   "CANDIDATE KEYS"
   "A candidate key is an attribute or a set of
     attributes such that all other attributes
     depend on it."
   nil)

(15
   "Check the prime attributes!"
   (and (equalp (current-task sol) 'prime)
       (not (null (prime-att sol)))
       (bind-all ?a (prime-att sol) bindings))
   (member ?a (remove-duplicates
             (flatten (candkeys sol)) :test 'equalp)
         :test 'equalp)
   "PRIME ATTRIBUTES"
   "Some of the attributes you specified as prime
     attributes do not appear in any candidate keys."
   (?a "prime-att"))


(12
   "You have not specified all the candidate keys
     for the given relation!"
   (and (equalp (current-task sol) 'candkeys)
       (not (null (candkeys sol))))
   (and (equalp (length (candkeys sol))
               (length (candkeys IS)))
        (null (list-set-difference (candkeys sol)
                  (candkeys IS))))
   "CANDIDATE KEYS"
   "Identify other candidate key(s) for this table."
   nil)
```

Figure 2.11: A Set of Sample Constraints from NORMIT

tifying candidate keys during the appropriate step. The relevance condition (third component of the constraint) checks whether the student is working on the candidate key identification step (encoded as *candkeys*). The satisfaction condition asserts that the candidate keys component (*candkeys*) of SS (represented using the variable *sol*) should not be empty.

Each constraint in NORMIT contains two hint messages: a general hint message (second component) and a more detailed hint message (sixth component). The general message is presented to the student during the first time the constraint is violated by the student's solution. Subsequent violations of the constraint prompt the system to present the detailed hint message.

Constraint 15, which ensures that all prime attributes appear in some candidate key, is an example of a constraint that is only relevant during the prime attribute identification step (*prime*). Its relevance condition checks that the value of *current-task* is *prime* and the *prime-att* component of SS is not empty. It also binds all the prime attributes specified by the student. The satisfaction condition ensures that they are members of candidate keys.

Constraint 12 is an example of a semantic constraint in NORMIT. It ensures that the student has specified all the candidate keys. The relevance condition of the constraint checks whether the student is working on the task of identifying candidate keys and whether the candidate keys component is populated. The satisfaction condition compares the number of candidate keys of the ideal solution to the number of candidate keys specified by the student and makes sure that the elements in both lists are the same.

NORMIT was evaluated in an evaluation study conducted in 2004 involving 29 volunteers who were enrolled in an introductory database course at the University of Canterbury [Mitrovic 2005*a*]. The study evaluated a basic version of NORMIT against a version that supports self explanation (NORMIT-SE). NORMIT-SE requires an explanation from the student for each action that is performed for the first time. Explanation is sought for subsequent actions of the same type in situations where the action is incorrectly performed.

Students were free to use the system when ever they wanted and for as long they wanted. NORMIT was used by 27 students and NORMIT-SE was used by 22 students. The sizes of the groups were different since not all students who volunteered to participate in the study had not used the system for problem-solving. The pre-test and post-test scores were similar for both groups, with no significant difference. However, analysing the constraint learning curves, which shows the probability of violating a constraint for each occasion it was relevant, revealed that students who who used NORMIT-SE had learnt faster than the control group.

2.3.5 WETAS: An Authoring Shell

Web-Enabled Tutor Authoring System (WETAS) [Martin 2002, Martin & Mitrovic 2002*a*] was developed as a tutoring engine that performs all common functions of a tutoring system. It is implemented as a web server and supports students learning multiple subjects simultaneously. Students using the system interact through a standard web browser such as Mozilla or Internet Explorer. WETAS provides the implementation for all the generic functions of constraint-based tutoring systems. In particular it provides generic implementations for answer evaluation, student modelling, problem selection, feedback selection and text-based interface.

Constraints that form the domain model for a tutoring system in WETAS have to be composed using a pattern matching language specific to WETAS. The language consists of three main functions: match, test and test_symbol. Constraints written using this language are interpreted by WETAS and used for evaluating student's solutions. Evaluation results are used to update the student model.

Students' knowledge and ability are modelled in WETAS in the form of an overlay model. It contains counts of how many times each constraint was relevant and violated. It also contains a trace of the behaviour of each constraint since the initiation of the model, which can be used to determine whether a constraint is learned or not.

The pedagogical module of WETAS selects a problem that best suits a student's current state and level of ability according to the structural and conceptual difficulty of each candidate problem. The problem selection algorithm calculates the difficulty each constraint adds to a problem and finds a problem that matches the level of the student's knowledge of each constraint calculated from the student model.

Feedback is selected by the pedagogical module by analysing the list of constraints violated by the student's submission. The system automatically increases the feedback level after each submission. The student is also given the ability to directly choose the desired level of feedback. The student may wish to view a single hint or a list of hints on all committed errors. In the event of the student choosing to view a single hint message, the system chooses the feedback of the first violated constraint. On the other hand, feedback messages of all violated constraints are displayed when *all errors* is chosen as the feedback level.

The interface module of WETAS automatically generates a text-based interface for the student with a fixed layout. The generic interface consists of four panels: problem statement, solution workspace, scaffolding and feedback. The problem panel presents the text of the problem. The solution workspace accepts student input for composing a solution. General help about the domain is provided in the scaffolding panel. Feedback messages to the student are displayed in the feedback window. The interface of a tutoring system running on WETAS can be made more sophisticated by replacing the default text-based interface by a set of HTML pages or a Java applet. WETAS also supports typical application based interfaces where the client and the server communicate via RPC (Remote Procedure Call).

Building an ITS

A new tutoring system can be authored in WETAS by providing the necessary data files. In particular, it requires two sets of domain-dependent data: problem set and domain knowledge base. The problem set contains a list of problems and their ideal solutions, and the domain knowledge base contains domain constraints (both syntax and semantic) and any macros.

Problems are described by their problem statement. The solutions on the other hand can be divided into components, where each component may be a single textual element or a list of sub-components. For example, the solutions in the SQL domain have six components: select, from, where, group by, having and order by. A sample problem and solution from SQL-Tutor is given in Figure 2.12.

```
(2
   "Retrieve the birth date and address of the
   employee whose name is John Smith."
     ;id       answer
 (("SELECT" "BDATE, ADDRESS")
  ("FROM" "EMPLOYEE")
  ("WHERE" "LNAME='Smith' AND FNAME='John'")
  ("GROUP BY" "")
  ("HAVING" "")
  ("ORDER BY" "")))
```

Figure 2.12: A Sample Problem and Ideal Solution from SQL-Tutor

The knowledge base consists of syntax and semantic constraints and macros for the domain. Constraints should account for all significant pedagogical states of the domain. Syntax constraints cover all the important principles of composing any solution for the domain. On the other hand, semantic constraints relate the student's solution to the ideal solution in order to verify that the requirements of the problem are satisfied. They must be sufficiently flexible in order to permit correct solutions that differ from the ideal solution.

Constraints are specified using a purely pattern-matching language developed for WETAS [Martin & Mitrovic 2002*b*]. Each pattern may be compared against the ideal solution (using the *MATCH* function) or against a variable (using *TEST* or *TEST_SYMBOL*) whose value has already been determined. Patterns which support wild cards and variables are combined with the use of logical connectives (*AND*, *OR*, *NOT*).

Figure 2.13 gives examples of a syntax and a semantic constraint from SQL-Tutor implemented in WETAS. Constraint 61 has a relevance condition with a pattern that tests the *HAVING* component of the student's solution for the existence of the key word *SELECT*. Its satisfaction condition tests for existence of a opening parenthesis before *SELECT* and a closing parenthesis.

The semantic constraint (constraint 55) contains a relatively complicated relevance condition that ensures the student has valid table names used in the *FROM* clause. The condition uses the ^*name* macro, written specifically for this domain, that checks whether a table name is valid. The satisfaction condition tests whether the table names used by the student are used in *WHERE* or *FROM* of the ideal solution.

```
; syntactic constraint from SQL-Tutor
(61
   "A subquery in the HAVING clause must be enclosed
     within brackets."
   (match SS HAVING (?* "SELECT" ?*))
   (match SS HAVING (?* "(" "SELECT" ?* ")" ?*))
   "HAVING")

; semantic constraint from SQL-Tutor
(55
   "You do not need all the tables you used in FROM."
   (and (not-p (match SS WHERE (?* "SELECT" ?*)))
      (or-p (match SS FROM (?* (^name ?t) ?* "ON" ?*))
         (and
            (not-p (match SS FROM (?* "ON" ?*)))
            (match SS FROM (?* (^name ?t) ?*)))))
   (or-p
      (match IS WHERE (?* "FROM" ?* ?t ?*))
      (match IS FROM (?* ?t ?*)))
   "FROM")
```

Figure 2.13: Sample Constraints from SQL-Tutor Written in WETAS Language

Example Tutoring Systems Developed in WETAS

WETAS has been used to implement intelligent tutors for a variety of domains including SQL, English language skills, database modelling and object oriented design. SQL-Tutor was reimplemented in WETAS in order to explore the capabilities of WETAS. The system for English language skills, called LBITS [Martin & Mitrovic 2003], was a paper-based teaching aid that was converted to a tutoring system. It teaches basic language skills with the use of a collection of *puzzles* such as crosswords, synonyms etc. ER-Tutor [Zakharov, Ohlsson & Mitrovic 2005], a database modelling tutor built initially as a Windows application, was reimplemented as a web-based system using WETAS. UML-Tutor [Baghaei & Mitrovic 2006*b*, Baghaei, Mitrovic & Irwin 2006] was implemented using WETAS for teaching object oriented design using unified modelling language (UML). WETAS has also been used to create an adaptive hypermedia system for paediatric radiology.

WETAS has been used as a teaching aid for a graduate course on Intelligent Tutoring Systems at the University of Canterbury. Students use the system as part of their course work for building a working prototype of a tutoring system for an allocated domain. They are introduced to WETAS during lectures and given a access to the domain dependent components of LBITS. Although students are novices to the task of constructing a tutoring system, the majority of students produce satisfactory tutoring systems [Martin & Mitrovic 2003].

2.4 Summary

The chapter is an introduction to Intelligent Tutoring Systems focusing on systems that support learning by problem-solving. It contained an outline of the typical architecture of an ITS including the functionality of main components. The chapter focused on two popular domain modelling techniques: model tracing and constraint-based modelling. The two modelling techniques were introduced, including their psychological theories. Descriptions of each modelling technique was also accompanied by details of tutoring systems developed using each method.

Building Intelligent Tutoring Systems require a lot of time and effort. The chapter included details of WETAS, an authoring shell that reduces the workload required for producing constraint-based tutors. However, WETAS does not provide any assistance for composing domain knowledge. The following Chapter (Chapter 3) discusses authoring support for composing domain knowledge and introduces a few currently available authoring systems.

Chapter III

Domain Knowledge Authoring Systems

Intelligent Tutoring Systems are computer-based instructional systems that contain models which specify how and what to teach [Ohlsson 1987]. These models, consisting of instructional content and teaching strategies, make inferences about a student's mastery of topics in order to dynamically adapt the content and style of instruction. The domain model that exists in an ITS allows learning content to be dynamically generated while evaluating attempts made by a student. As a consequence ITSs provide students with realistic and meaningful environments where they can "learn by doing".

The development of these domain models is a time- and labour-intensive task, consuming a major portion of the development effort spent in building an ITS system. Typically domain models consist of hundreds of elements that collectively cover all pedagogically significant states. In the case of model tracing, where knowledge elements are production rules, Anderson and co-workers estimated that ten hours or more were required to produce a single production rule [Anderson et al. 1996]. Mitrovic reported that she required approximately an hour to produce a single constraint for building the domain model for the constraint-based tutoring system, SQL-Tutor [Mitrovic 1998*a*]. A major factor that increases the time required for composing domain models is the necessity of ensuring that a production rule or a constraint fires appropriately. Furthermore, the task of debugging knowledge elements is also hard.

Building a domain model requires multi-faceted expertise, such as knowledge engineering, programming and the domain itself. Typically a domain model is constructed collaboratively by a domain expert, a programmer and a knowledge engineer. While the domain expert may outline the necessary domain knowledge at a high level, the knowledge engineer would specify the knowledge elements of the domain. The programmer would collaborate closely with the knowledge engineer to program the knowledge elements to build the domain model.

In order to overcome the bottleneck of building domain models, researchers have been investigating ITS authoring tools ever since the inception of ITSs. These domain-knowledge authoring systems reduce the time and effort required for building domain models by generating the domain model with the assistance of a domain expert. As a consequence, the need for programming and knowledge engineering expertise also diminishes as the domain expert directly authors domain model content. These systems also have tools for validating the generated domain model and ensuring that the complete domain is covered, reducing the effort required for debugging the domain model.

The chapter commences with an introduction to research attempts at alleviating the knowledge acquisition bottleneck. The following section outlines the state of the art in domain-knowledge authoring tools for ITS. While introducing the systems and their capabilities, their strengths and weaknesses are discussed. As composing an ontology of the domain is an integral part of the typical domain authoring process, Section 3.2 introduces a selection of popular tools for modelling domain ontologies. It includes descriptions of a collection of both commercial and research tools, along with discussions of their strengths and weaknesses.

3.1 Domain Knowledge Authoring Tools

Researchers have been looking for solutions for reducing the knowledge acquisition bottleneck since the inception of Intelligent Tutoring Systems. Extensive research aiming to reduce the time and effort required for composing domain models has resulted in domain-knowledge authoring systems. Authoring systems ranging from basic systems with form-based interfaces that ease the task of inputting the required components to more sophisticated systems using machine learning techniques for acquiring the knowledge have been developed.

Murray [1999, 2003] classified ITS authoring tools into two main groups: pedagogy-oriented and performance-oriented (see Figure 3.1). Pedagogy-oriented systems focus on sequencing and teaching relatively fixed content. They include systems for sequencing and planning curriculum, authoring tutorial strategies, composing multiple knowledge types (e.g. facts, concepts and procedures) and adaptive hypermedia. On the other hand, performance-oriented systems focus on providing rich learning environments where students can learn by solving problems and receiving dynamic feedback. Authoring systems for domain expert systems, simulation-based learning and some special purpose authoring systems focus on performance.

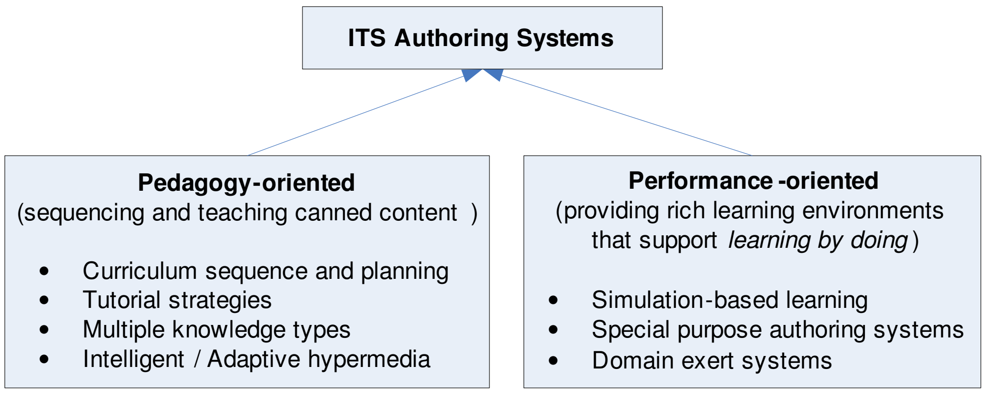

ITS Authoring Systems
Pedagogy-oriented
(sequencing and teaching canned content)
Curriculum sequence and planning
Tutorial strategies
Multiple knowledge types
Intelligent / Adaptive hypermedia
Performance -oriented
(providing rich learning environments that support learning by doing)
Simulation-based learning
Special purpose authoring systems
Domain exert systems

Figure 3.1: Classification of Authoring Systems

Domain expert systems are the most sophisticated type of tutoring systems. We are mainly interested in the performance-oriented authoring systems, in particular authoring systems for domain expert systems. Typically students use these systems to solve problems and receive customised feedback depending on their attempt. These systems have a deep model of expertise, which enables the tutor to correct the student as well as provide assistance on solving a problem. Authoring systems for these systems focus on generating rules that form the expert model or the domain model. They typically use sophisticated machine learning techniques for acquiring rules of the domain with the assistance of a domain expert (e.g. Disciple [Tecuci 1998], Demonstr8 [Blessing 1997]).

Authoring systems for simulated domains have been popular. These systems record the domain expert performing a task within the simulated domain and generalise the recorded procedure using machine learning algorithms. The student experiments within the simulated domain in order to discover ways of generalising the expert's problem-solving path. The "what-if" activities for the simulated environment are generated by these systems automatically.

The following subsections describe a few significant authoring tools that have been published in the literature. They contain details of two authoring systems that focus on acquiring rules from simulated environments; KnoMic and Dilligent. Details of Disciple, which is a learning agent shell for developing learning agents follows. The section also contains details of Demonstr8 and CTAT, two authoring systems developed for assisting the process of building model-tracing tutors.

3.1.1 KnoMic

Knowledge Mimic (KnoMic) [van Lent & Laird 2001] is a learning-by-observation system based in a framework for learning procedural knowledge by observing an expert. In addition to demonstrating a task, the expert is also required to annotate goal transitions. The system uses the observation traces to learn production rules for the Soar architecture. Ultimately, the production rule system can be used to replace the expert by automatically performing the task interacting with the simulated environment.

The first step of the knowledge acquisition process results in the generation of a number of observation traces. They are generated by observing the expert performing tasks within the simulation environment using an interface that sits between them. The environment interface sends the expert's commands to the environment and returns the sensor information to the expert. In order to generate a complete observation trace, the system also requires the expert to annotate any changes in a goal as a result of it being achieved or being abandoned. The system allows the expert to model a goal hierarchy for tasks in the environment prior to demonstration in order to aid in specifying goals. The observation trace generator uses the sensor inputs from the environment, expert's commands and the expert's goal change annotations to compose observation traces.

The next step, after observation traces are available, is to learn operator conditions using a condition learning algorithm. The system, typically requiring 4-8 observation traces, incrementally learns conditions for operators, actions and goals. It uses a specific-to-general induction algorithm named "Find-S" to learn the conditions. Each step in the observation trace that contains an operator change annotation is used as a positive instance of that operator's pre-conditions and a positive instance of the previous operator's goal conditions. The algorithm treats the first positive instance as an initial, most specific hypothesis and generalises it to cover the subsequent positive instances. If generalising to cover a new instance results in an empty set of pre-conditions (no pre conditions) a disjunctive second set of pre-conditions is created. Each new instance is then applied to the set of conditions that require the least generalisation to cover it. Once all the pre-conditions for operators have been learnt, the goal conditions are learnt in a similar manner. Finally, action conditions are learnt by treating each step in which the expert performs an action as a positive instance.

The third step involves classifying each operator as homeostatic, one-time or repeatable. Operators are classified by examining situations where an operator is reselected as a result of its goal conditions changing from achieved to unachieved. This classification determines whether each operator has a constant feature that indicates whether the goal is achieved. If an operator's goal achievement feature is constant, the system can keep track whether the particular goal has been achieved and that it should not be pursued again, even if the goal conditions are no longer satisfied. On the other hand, in the case of operators that do not have a constant achievement goal, their goal features should be achieved and maintained.

The system was evaluated for its accuracy for learning to control a military aircraft in the ModSAF battlefield simulator. The task involved taking off, flying to a specific point and flying a patrol pattern. If an enemy plane is detected and a set of criteria is satisfied, it should be intercepted and shot down. The environment interface provides the user a with 54 sensor inputs, 23 task parameters and 22 output commands. The knowledge required for the domain consists of 31 operators, including initialisation operators, take-off operators and mission operators etc. As part of the evaluation, two tests were conducted in the the air combat domain. The first involved using the knowledge learnt by the system to perform the task in the simulated domain and the second involved comparing the generated rules against a set of rules composed by a human programmer. The two tests were used to classify each rule in to three categories: fully correct, functionally correct or incorrect. The rules that fall into the functionally correct category are rules that did not cause any errors in the first test (performing the task) but do not match the the hand-coded rules. These rules may cause errors in novel situations.

The system was evaluated in two experiments. The first experiment evaluated the correctness of the knowledge generated from a set of error-free observation traces generated by observing a hand-coded expert system performing tasks in the domain. The expert system, capable of generating traces with precision timing, content and operator annotations provided KnoMic with four traces. The different traces provided to the learning system included differences in attributes, speeds etc due to different starting conditions and non-deterministic environmental conditions. KnoMic learnt the 31 domain operators from the observation traces producing a total of 140 productions. From the total set of generated productions, 101 were fully functionally correct, 29 were functionally correct and 10 were incorrect.

The second experiment involved evaluating the correctness of knowledge produced by a set of observation traces generated from observing a human expert. The observation traces were limited only to initialisation, take-off and racetrack parts of the task as the intercept portion of the task was difficult to be completed consistently. The traces generated by the human expert included more variability in task performance in comparison to the traces generated from expert system. KnoMic generated a total of 45 productions by analysing two observation traces. From the generated productions, 29 were correct, 13 were functionally correct and 3 were incorrect.

Although the published preliminary results look promising, the true effectiveness of the system has to be studied further. In order to evaluate the true potential of the system, it has to be evaluated in a variety of simulated domains. However, evaluations published in [van Lent & Laird 2001] have been conducted in the same domain. Furthermore, as the main aim of KnoMic is to ease the process of generating knowledge by observing an expert of the domain, the limited experiment conducted with the human expert does not provide any insights into the general effectiveness of the system. From the results, KnoMic seems to do well, with highly accurate observation traces. However, good authoring systems should also be able to handle noisy traces as well.

3.1.2 Diligent

Diligent [Angros, Johnson, Rickel & Scholer 2002] is an authoring system that acquires the knowledge required for a pedagogical agent in an interactive simulation-based learning environment. It learns the required rules for the agent by observing a domain expert demonstrating the skill to be taught in the simulated environment. Once the skill has been demonstrated, the system automatically experiments with the recorded traces in order to understand the role of each step in the procedure. The expert can also directly modify the learnt procedures providing clarifications at the completion of the experimentation stage. The authoring process also involves acquiring the linguistic knowledge required to explain procedural tasks to students.

The demonstration phase is initiated by the domain expert, who issues a command for Diligent to observe his/her actions. It records each step of the task, noting the state of the environment before and after each action. During the observation, each action results in an operator that models the effects of the action. At the completion of the demonstration, the system learns a sequence of steps in one possible ordering to perform the particular task. Diligent hypothesises that the likely goals of the task are the final values of the state variables that changed during the demonstration. The domain expert has to review the points classified by Diligent as goals to remove the points that are merely side effects of the task.

The system uses experimentation to generalise the pre-conditions of the actions to achieve the task demonstrated by the expert. The experimentation involves repeating the task for each step and omitting a step from the original sequence during each repetition. The system uses a machine learning version space algorithm named INBF to generalise the set of pre-conditions for each effect of the operator. It maintains two sets of pre-conditions for each effect, namely the most specific and the most generic. The most specific set is initialised to match the state before the action during the demonstration and the generic set is initialised to match any state. During the experimentation process, the specific set is generalised and the generic set is specialised by the learning algorithm to produce the most probable set.

After the experimentation stage, the agent uses the original demonstration, the end goals of the task and its refined operators to generate a representation of the learned task. The agent uses this representation to identify the causal links and ordering constraints of the task. The domain expert can review the procedural rules learnt by the system by examining a graph of the task model or by allowing the agent to demonstrate its acquired knowledge by teaching it back to the domain expert. During the review process, the expert can directly modify the task model by adding or removing steps, ordering constraints etc.

Diligent learns the required linguistic knowledge to communicate with the students from the text fragments specified by the domain expert for tasks, steps and goals. The agent uses domain-independent text templates to support natural language generation during tutoring.

The system was evaluated on a series of tasks for operating gas turbine engines on board a simulated US naval ship. The study revealed that demonstrating and later modifying the task model produced a significantly better task model (with fewer errors) than directly manually specifying it. It also revealed that Diligent's demonstration, experimentation in conjunction with direct specification produced a higher quality task model in comparison to a task model produced by only demonstration and direct specification. This suggests that demonstration and experimentation both played a role in improving the quality of the task model. The results also showed that Diligent's assistance is most beneficial with complex procedures that have higher risk for errors.

Although the evaluation has produced encouraging results, Diligent is bound by its limitations. It is best suited for authoring procedural knowledge where the focus is to achieve some desired effect in a virtual world and where the consequences are readily observed. It is not effective in environments such as in medicine, where the steps are determined by the patient's condition rather than the effects they have on the patient.

The nature of domain

3.1.3 Disciple

Disciple [Tecuci 1998, Tecuci, Wright, Lee, Boicu & Bowman 1998, Tecuci & Keeling 1999] is a learning agent shell for developing intelligent educational agents. A domain expert teaches the agent to perform domain-specific tasks similar to an expert teaching an apprentice, by providing examples and explanations. The expert is also required to supervise and correct the behaviour of the agent. Disciple acquires knowledge using a collection of complementary learning methods including inductive learning from examples, explanation-based learning, learning by analogy and learning by experimentation. A completed Disciple agent can be used to interact with students and guide them while performing tasks of the domain.

Initially the Disciple shell has to be customised by building two domain-specific interfaces and a problem solver for the domain. One domain-specific interface provides the expert with a natural means of expressing their knowledge, and the other is built to facilitate interaction between the agent and the student. The extent and the nature of the problem solver depend on the type and the purpose of the learning agent to be developed. The tasks of developing the interfaces and the problem solver requires expertise of a software engineer and a knowledge engineer, where the developer has to collaboratively work with a domain expert. Once the customisation is complete, the domain expert can interact with the agent to demonstrate how to perform domain-specific tasks.

The customised Disciple agent acquires knowledge using a four stage process;

1. Knowledge elicitation

2. Rule learning

3. Rule refinement

4. Exception handling

The goal of the initial knowledge elicitation phase is to construct an initial knowledge base to be further extended and improved during the latter stages of the knowledge acquisition process. The knowledge base consists of a semantic network and a rule base. The semantic network outlines the concepts and instances of the domain, described by their properties, including relationships between them. It can be either composed manually using Disciple's interface or imported from a repository, if available. The rule base, characterised by tasks, operations and examples, is generally generated during the following phases.

The expert can demonstrate how typical domain-specific problems are solved during the rule learning phase when the semantic network is sufficiently complete. A rule learning episode is initiated by the author providing a correct example from the domain. The agent then attempts to explain the reasoning behind why the example is correct using the semantic network with the guidance from the domain expert. An explanation may contain several paths in the semantic network between an object in the example and another object. The agent then generalizes the example and the discovered explanation paths and produces plausible version space (PVS) rules (with a specific and a general condition) using analogical reasoning. The explanation generated from the previous phase is set as the lower bound of the PVS rule and a generalised version according to the semantic network is set as the upper bound of the rule.

The goal of the rule refinement phase is to improve the PVS rules in the knowledge base while extending and correcting the semantic network until all instances of the rules in the semantic network are correct. A rule is either generalised or specialised using a positive or negative example, either generated by the the agent itself through active experimentation or provided by the expert or obtained during problem-solving. A positive example results in generalising the lower bound of the rule to cover the positive example. On the other hand, if an explanation, according to the semantic network, can be found to distinguish the negative example from a positive example, both bounds of the rule are specialised. For instances where such an explanation cannot be found, the lower bound of the rule is specialised so as not to cover the example or it is added as a negative exception to the rule.

The final phase of handing exceptions aims to reduce the number of exceptions of a rule by either refining the semantic network or replacing it with a set of rules with fewer exceptions. During this phase the agent hypothesises additional knowledge and guides the expert to define the missing knowledge in the semantic network. First, the agent focuses on a rule's positive exceptions that do not have the required features of the rule's conditions. It attempts to cover them by the rule by engaging the expert in a dialogue to extract any missing features. Next the agent attempts to resolve the negative exceptions, that are not covered by the corresponding concepts in the rule's condition, by discovering new concepts with the assistance of the expert.

The Disciple agent has been applied to a number of domains including history, statistical analysis, engineering design and military combat. The history agent uses the rules of the knowledge base to generate new problems. Students interacting with the agent are presented with multiple-choice questions and the agent evaluates their solutions and provides hints to guide the students towards the correct solution. The statistics analysis agent was also used to generate new problems. It assists students in performing routine tasks as they worked through their assignments. The engineering design agent also learns to perform routine, labour-intensive tasks to assist designers. The agent is capable of producing new designs using its reasoning capabilities that can be later verified and corrected by the designer. Another agent was trained to behave as a military commander in a virtual combat environment, where it was trained to carry out defensive missions.

An evaluation study conducted to evaluate the effectiveness of the history agent has reported encouraging results [Tecuci & Keeling 1999, Tecuci, Boicu, Marcu, Stanescu, Boicu & Comello 2002]. The rule base developed by the agent was evaluated by experts who were involved in training the agent, as well as independent domain experts. Both groups of experts rated the accuracy of the questions generated by the agent as approximately 96% correct. A class-room-based experiment revealed that most students rated the questions as helpful and understandable. A group of teachers also rated the agent as a helpful tool for learning history.

The task of customising the Disciple agent requires extensive programming skills and considerable effort. The programmer and the domain expert have to collaborate with each other in order to identify the requirements and design an interface that is a natural problem-solving environment for the domain. The task of building a problem solver for the domain also requires extensive collaboration between the domain expert and a programmer. Furthermore, building a problem solver in some domains may be extremely hard, if not impossible. For example, for domains that are open-ended, such as database modelling, a problem solver with the ability to identify a correct solution needs to be able to perform multiple complex tasks. It should be able to process natural language as the problem is given in natural language, to generate a database model that satisfies the given requirements and identify semantically equivalent database models to ensure that equivalent solutions are identified as correct. In the case of database modelling, building a problem solver is probably impossible.

The semantic network, which is modelled during the knowledge elicitation phase, contains information about concepts of the domain as well as instances. The need to add all elements participating in an example solution prior to demonstrating how it is solved adds a high degree of repetitiveness to the knowledge authoring process. Furthermore, in some domains, the instances may only be used in a single solution, which forces the expert to add large numbers of instances to the semantic network. As the semantic network is highly descriptive, showing all relationships between elements in the network, it tends to be very complex even for small domains. Larger domains would tend to have far more complex semantic networks, where locating nodes of the network becomes very difficult and domain experts may even get disoriented.

Although Tecuci and co-workers haver reported encouraging results, there has been no evaluation of the effort required to build an ITS using the Disciple method. In the event of not being able to find a semantic network from a repository, manually building a semantic network from scratch is a time-consuming process. Furthermore, demonstrating problem-solving steps, providing explanations as well as refining the generated rules would also require a considerable amount of effort.

3.1.4 Demonstr8

Blessing [1997] developed Demonstr8, an authoring system that assists in the development of model-tracing tutors, within the domain of arithmetic. The system aims to reduce the time and outside knowledge required for building model-tracing tutors. It infers production rules using programming by demonstration techniques, coupled with methods for abstracting the underlying productions. The author is required to model an interface and solve problems using it to demonstrate the method of solving a problem.

The process of building a tutoring system using Demonstr8 is initiated by creating a student interface using a drawing tool like interface. It consists of a cell widget tool, which can be used to add place holders for numbers and a line tool, which allows lines to be drawn on the interface. All cells placed on the interface automatically become available as working memory elements (WMEs).

After the interface has been composed, the higher-order WMEs, which are collections of atomic WMEs that have a meaning, have to be identified. They can be specified by simply highlighting them from the student interface as a group and specifying that they are a WME. During this stage, classes for WMEs also have to be defined. To assist in the process of defining classes, the system automatically adds the cells within the WME that belongs to the class as properties of the class. Higher-order WMEs can also be specified by selecting the WMEs from the list of existing WMEs.

The system relies on *knowledge functions* for declarative knowledge not depicted directly in the interface. These functions are implemented as a two-dimensional table of values that contain the result for applying the function to two inputs. The result of two inputs can be found by locating the cell with the first input as the row and the second as the column. The system contains built-in *knowledge functions* for standard arithmetic operators, but can be extended by adding a new table of results.

After creating all the WMEs and adding any new *knowledge functions*, the author has to demonstrate the domain problem-solving procedure. Initially the author provides an example problem for the domain and notifies the system to record the problem-solving process by clicking a *Record* button. The author can either directly select the *knowledge function* used for a problem-solving step, or let the system automatically choose the appropriate function from the available list. The system selects a function by exhaustively going through the list of available *knowledge functions* to locate the one which produces the demonstrated result. In the event of discovering multiple *knowledge functions* with the same outcome, the system gets the author to select the correct one from the group of candidates.

The system generates a production rule and displays it to the user immediately after each action performed by the author during the problem-solving demonstration. The displayed production rule can be modified by the author by selecting a more general of specific condition or action from the available list. The author is also required to specify a goal and skill covered by the generated production rule. The author also has to specify four help messages, with increasing level of detail, specific to the generated rule.

The process of demonstrating an action and fine-tuning the generated production rule is continued until the problem-solving task is complete. The system contains an implicit 'Done' production which assumes that the problem is solved when a particular cell, pointed out by the author, is filled. The system however, lacks a utility for specifying buggy rules, that are essential in model tracing for identifying common misconceptions between students.

The production rule generation process is highly dependent on the WMEs created by the author. In order for correct productions to be generated by the system, the author should use the right representation for WMEs. Although the methodology for selecting higher-order WMEs is relatively straight forward, the task itself requires knowledge in cognitive science and model tracing. It is unreasonable to assume typical educators such as teachers would be knowledgeable in the model tracing approach. Furthermore, a considerable amount of practice is required in order to master the task of identifying higher order WMEs.

Each production rule generated by the system by analysing an author's action is displayed to the author for fine-tuning. Understanding and being able to identify deficiencies in a production rule also requires expertise in model tracing. Furthermore, if the production rule requires subgoaling, it also has to be directly specified by the author. Consequently, the system's target of paving the way for typical educators to build tutoring systems leaves a lot to be desired.

Although the author argues that methodology used in Demonstr8 can be adapted for other domains, the system described in [Blessing 1997] is limited to arithmetic domains only. He also admits that creating a tutoring system for geometry is difficult as the knowledge required to progress from one step to the other is not directly observable from the interface. Furthermore, developing a tutoring system for a non-procedural, open-ended domain such as database modelling would be extremely difficult, if not impossible.

3.1.5 CTAT

The Cognitive Tutor Authoring Tools (CTAT) [Koedinger et al. 2004, Jarvis et al. 2004, CTAT 2005], developed at Carnegie Mellon University assist the creation and delivery of ITSs based on model tracing. The main goal of these tools is to reduce the amount of Artificial Intelligence (AI) programming expertise required. The system allows authors to create two types of tutors: 'Cognitive tutors' and 'Pseudo tutors'. 'Cognitive tutors' contain a cognitive model that simulates the student's thinking to monitor and provide pedagogical assistance while solving problems. In contrast, 'Pseudo tutors' mimic the behaviour of a tutor without a cognitive model. Although 'Pseudo tutors' do not require AI programming, they are specific to the set of problems that were modelled.

The process of authoring a Pseudo tutor using the CTAT tools involves four steps. The authoring task is initiated by creating a graphical user interface (GUI) to be used by the prospective students of the final tutoring system. The second step requires the author to demonstrate alternative correct and incorrect solutions for a problem. At the completion of demonstrating solutions, the author is required to annotate solution steps with hint messages, feedback messages and labels for associated skills. Finally the skill matrix has to be inspected and revised.

CTAT contains a GUI builder tool that makes it possible to create interfaces by directly manipulating interface widgets. Authors can simply add interface widgets in a manner similar to using a drawing program with out the need for any programming expertise. The GUI builder tool, implemented using Java NetBeans consists of a collection of widgets ranging from text boxes, combo-boxes to more complex widgets such as “Chooser” and “Composer”. The “Choose” widget allows students to enter hypotheses and “Composer” widget allows students to compose sentences by combining phrases from a series of menus.

After the completion of the interface, the author can use the behaviour recorder to author problems and demonstrate their solutions using the interface created during the interface building phase. Once the interface is populated with the state of a new problem, the author issues a command to the behaviour recorder to create a start state. As the author demonstrates problem-solving procedures, the behaviour recorder produces a behaviour graph that contains arrows, representing the actions performed by the author, and labels, representing the resultant states of the interface. In order for the resulting tutoring system to be robust, the author needs to demonstrate all possible correct solutions of a problem as well as typical incorrect solutions.

After demonstrating solutions for a problem, the author has to annotate the behaviour graph by adding hint messages to correct links and buggy messages to incorrect links. The author can enter up to three messages, with increasing amounts of detail, to be presented to the student when they request for assistance. The author is also required to add knowledge labels to links in the behaviour graph to represent the knowledge behind problem-solving steps. Adding knowledge labels to a step allows the author to easily copy hint messages from one problem to another. As the hint messages are specific to the problem it belongs to, it needs to be modified to suit the new problem.

The knowledge labels added during the third phase of building a ‘Pseudo tutor’ is used for generating a skill matrix. The skill matrix outlines the knowledge elements required to solve each problem supplied by the author. The author can inspect the skill matrix and either modify previously added problems or add new problems that cover the skills which are not covered by the existing problems. The author can also reflect on the plausibility of transfer predictions made by the skill matrix or even use a tutor to collect performance data to test the predictions.

The time required for building instructional content for 'Pseudo tutors' using CTAT tools has been informally evaluated. The tools were evaluated in four projects for the domains of mathematics, economics, law (LSAT) and languages skills. The evaluations revealed that the ratio of design and development times to instructional time is approximately 23:1 on average. This ratio compares favourably to the corresponding estimate of 200:1 for manually constructed fully functional cognitive tutors [Koedinger et al. 2004]. The effectiveness of the pseudo tutor was evaluated for LSAT involving 30 pre-law students. The experimental group of 15 students used the tutor for a period of one hour, whereas the control group worked through a selection of sample problems on paper. The control group were provided the correct solutions after 40 minutes of solving problems. The results showed that students using the tutor had performed significantly better in the post test.

Although, in theory, both pseudo tutors and cognitive tutors exhibit identical interaction with the student, pseudo tutors are extremely specific to problems. While new problems to a full cognitive tutor can be added with little effort, pseudo tutors require all possible problem-solving paths to be demonstrated for each new problem. The task of demonstrating solution paths becomes increasingly tedious as the number of similar problems increases and the complexity of alternate paths increases.

Jarvis and co-workers have implemented an automatic rule authoring system for CTAT tools [Jarvis et al. 2004], generating JESS (Java Expert System Shell) rules. Their goal is to automatically generate the required production rules, through programming by demonstration techniques, given the background knowledge of the domain and examples of problem-solving steps. The rule generation system attempts to generalise the set of behaviour graphs generated during the process of building Pseudo tutor and produce a Cognitive tutor. The correct steps demonstrated by the expert are used as positive examples and incorrect steps are used as negative examples for the machine learning system.

The rule generation process requires a domain expert to list the set of skills required to solve a problem in the domain. Each skill describes one of the actions that is required for solving a problem. After the skills are outlined, the author has to select inputs and outputs for each application of a skill from the demonstrated problem-solving paths by highlighting widgets in the interface. The rule generation algorithm generates a rule for each outlined skill. The right hand side and left hand side of the rule is generated separately.

The right hand side of the rule is generated by searching through the space of all possible combinations of available functions for the domain and all possible permutations of variables. The search is repeated until a depth limit is reached, calculated as a probability of occurrence based on a default value, user preference and historical usage. The result is a rule with a number of functions in a particular order with a set of variable bindings. This rule is tested against all the positive and negative examples and if the rule does not cover a positive example or incorrectly predicts a negative example, the search is restarted by removing the last function from the rule.

The left hand side, which consists of a set of conditionals that must be satisfied for the right side to execute, is generated by generalising a sample behaviour graph. The process of generating the left-hand side only uses positive examples and assumes that facts in WMEs are somehow connected and that the connections do not form any loops. Each positive example is represented as a tree with inputs as the root and outputs as leaves. The trees of each positive example is merged into one general tree using wild cards to represent a collection of WMEs.

The rule generation system was tested in the domains of multi-column multiplication, fraction addition and tic-tac-toe. The rules for all three domains were learnt in a reasonable amount of time. However, it was reported that the rule generation algorithm generated over-generalised left-hand sides for the domains of multiplication and tic-tac-toe. Moreover, the tic-tac-toe domain required the creation of higher order WMEs to represent three consecutive cells.

In order to use the automatic rule generation system, the author has to possess a thorough knowledge of model tracing. Listing the complete set of skills required for the domain is a task that requires practice as well as understanding that they have to be outlined in great detail. As the rule generation engine is fully dependent upon the list of skills, an incomplete list by the author would result in an incomplete set of generated rules. Furthermore, the specification of higher order WMEs is also an important task that has a direct impact on the final set of generated rules. Typical educators with no or little background knowledge of model tracing would find the task complicated. The task of producing the required information in addition to the behaviour graph would be tedious for the domain expert. It may be easier for the author to produce the required input during the problem-solving demonstration phase itself.

3.2 Producing a Domain Ontology

Composing a domain ontology (semantic network, concept map) that explicitly outlines the domain's concepts and structure is an essential task of a typical domain authoring process. Authoring systems that provide assistance in composing domain models for ITSs, require a model of the domain in terms of its concepts to understand the boundaries of the domain. Typically authoring systems that produce a runnable model of domain expertise use some form of a conceptual model of the domain as a starting point in the knowledge acquisition process.

Model-tracing tutoring systems focus on operational knowledge, implemented using production rules. The operational knowledge for these systems can be represented in an abstract sense using graphs which outline all user interactions and their resulting states. CTAT tools implemented to assist the production of model tracing tutors use a graphical model called *behavioural graphs* to outline the problem-solving process. Behavioural graphs allow the problem-solving process to be outlined in fine detail displaying alternative steps as well as incorrect steps.

In contrast, constraint-based tutoring systems focus on declarative knowledge. In other words, the domain model of a constraint-based tutoring system outlines the principles of the domain. In order for a computer-based authoring system to provide assistance in authoring the domain model, it needs to understand the vocabulary of the domain. While a natural language description of the domain is almost impossible for computers to understand, an ontology of the domain provides a machine interpretable definition of the domain. Such an explicit formal outline of the domain may also assist domain authors during the process of manually authoring a domain model.

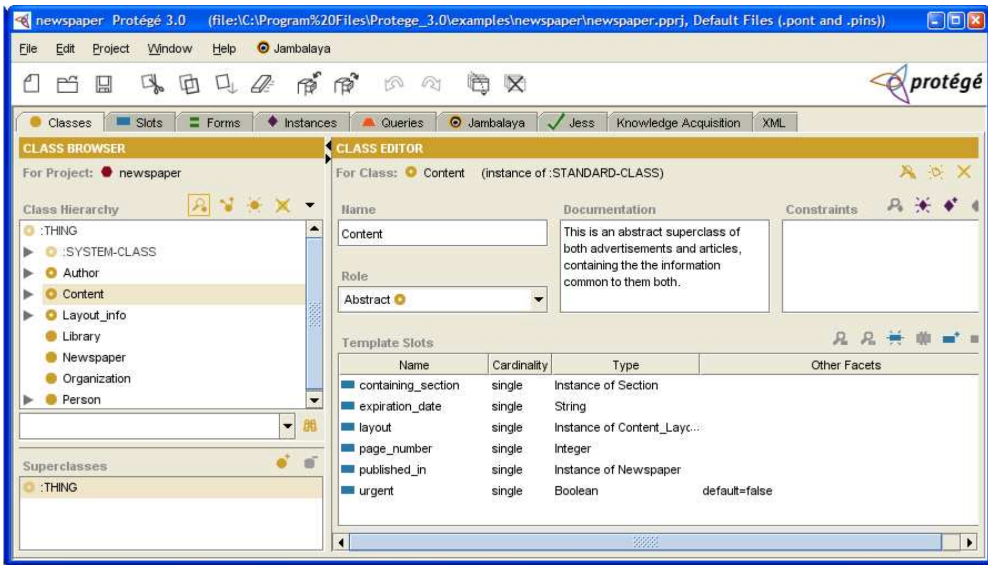
Gruber and Studer have described an ontology as an "explicit and formal specification of a conceptualization" [Gruber 1993]. In other words, it can be described as an exhaustive and rigorous conceptual schema of a domain [Wikipedia 2005]. It outlines all the concepts of the domain and relationships between them. The concepts within an ontology are described by their attributes. The concepts typically form a hierarchy with super- and sub-concepts.

There have been various, academic and commercial, tools developed for composing conceptual models of a domain. They support a variety of ontology languages such as RDF [RDF 2006], DAML [DAML 2006], OWL [OWL 2004] etc. The following sub sections outline three popular domain modelling tools: Prot*ég*é, OilEd and CMapTools.

3.2.1 Protégé

Prot*ég*é [Knublauch 2003, Noy, Sintek, Decker, Crubézy, Fergerson & Musen 2001] is a successful open source knowledge modelling platform. It can be used to build conceptual models and knowledge bases. Conceptual models can be composed using the Prot*ég*é's form-based interface for modelling concepts, their attributes and relationships between them. Instances can be created to populate a knowledge base using interactive forms for entering instance data generated by the system. Once the model is complete, it can be queried using plug-ins provided by the system. The domain models can also be loaded and saved in various formats, including XML, RDF, DAML etc.

The interface of Prot*ég*é, as shown in Figure 3.2, has several tabs containing the interfaces for achieving specific tasks. The figure illustrates the left most tab, which allows the creation of concepts (i.e. classes) along with their attributes and relationships. The 'Classes' interface is divided into the 'Class browser' and the 'Class editor'. The 'Class browser' allows the creation of new classes and arranging them in an inheritance hierarchy. The inheritance hierarchy of classes is visually depicted as a tree. As Prot*ég*é supports multi-



:THING
:SYSTEM-CLASS
Author
Content
Layout_info
Library
Newspaper
Organization
Person

Name: Content

Role: Abstract

Documentation: This is an abstract superclass of both advertisements and articles, containing the the information common to them both.

Name	Cardinality	Type	Other Facets
containing_section	single	Instance of Section	
expiration_date	single	String	
layout	single	Instance of Content_Layo...	
page_number	single	Integer	
published_in	single	Instance of Newspaper	
urgent	single	Boolean	default=false

Figure 3.2: Interface of Protégé 3.0

ple inheritance, classes with more than one super class are duplicated in the inheritance tree. The details of the class selected from the 'Class browser' are shown in the 'Class editor'. It allows specification of class details such as attributes, relationships and any restrictions.

Protégé uses the term 'slots' for both attributes and relationships of a class. A slot has a name and a type of values it may hold. The type of values a slot may hold can be either a primitive type such as boolean, integer, float and string or instances of another class. Protégé also offers a primitive type called 'symbol', which represents an enumeration of textual values. Relationships between instances of classes are modelled using slots that hold instances of another class.

Slots are implemented in Protégé as global objects. They do not need to be attached to a class. They may even be attached to multiple classes. Protégé contains a tab named 'Slots' for viewing all slots, independent of their classes. Slots may also contain restrictions on the range of values they may hold. One of the restrictions is the cardinality of the slot, which determines the maximum and minimum number of values a slot may hold. Further restrictions include maximum and minimum values, default values etc. The system also provides Protégé Axiom Language (PAL) for specifying more complex restrictions.

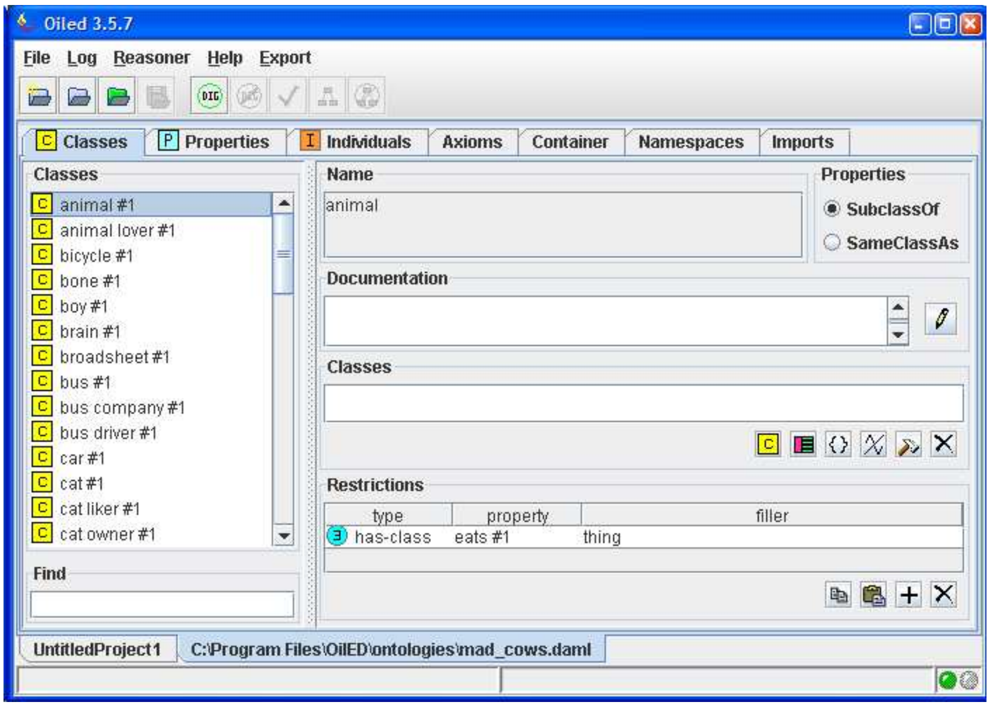
After modelling classes and slots, their instances can be created using the automatically generated graphical forms. The interface of the generated forms consist of text fields, radio buttons, check boxes, combo boxes, lists and other widgets reflecting the structure of the underlying class. The automatic layout generated by the system can also be modified using the 'Form' tab, where form widgets can be moved by dragging them. Any change to the underlying class prompts the system to regenerate the corresponding instance entry form.

The system can be enhanced by plugging in additional modules. Prot*égé* supports three types of plug-ins: storage plug-ins, slot widgets and tabs. Storage plug-ins enable saving and loading ontologies in different formats such as XML, RDF, DAML, OWL etc. The slot widget plug-ins are graphical components such as text fields and combo boxes for viewing and editing instances in the instance forms. The interface and the functionality of Prot*égé* can be enhanced by adding Tab plug-ins. Such plug-ins include tools for visualisation (e.g. Jambalaya), intelligent reasoning (e.g. Jess) etc.

Prot*égé* was designed for knowledge engineers to create an ontology of a domain, add instances of concepts and reason about the domain. Consequently, users of the tool must posses extensive knowledge engineering expertise in order to be able to produce ontologies. Novice users may find it difficult to use features such as specifying axioms using the PAL language. They may simply be overwhelmed by the tool, as a consequence of all the extra functionality provided such as queries, intelligent reasoning using Jess etc.

Although Prot*égé* includes a tool for visualising the ontology, the graphical tool only allows editing existing classes. It does not let new classes to be added [Storey, Musen, Silva, Best, Ernst & Noy 2001]. A modifiable graphical view of the ontology may assist users in understanding the conceptual domain. The 'class' interface of Prot*égé* provides a basic hierarchical structure of the domain, however, some users, particularly novices, may prefer a graphical view of the ontology, where concepts are arranged and grouped in areas of the canvas for better understanding.

3.2.2 *OilEd*



animal #1
animal lover #1
bicycle #1
bone #1
boy #1
brain #1
broadsheet #1
bus #1
bus company #1
bus driver #1
car #1
cat #1
cat liker #1
cat owner #1

Name: animal

Properties: [✓ SubclassOf] SameClassAs

type	property	filler
has-class	eats #1	thing

C:\Program Files\OilED\ontologies\mad_cows.daml

Figure 3.3: Interface of OilEd

OilEd [Bechhofer, Horrocks, Goble & Stevens 2001*a*, Bechhofer, Horrocks, Goble & Stevens 2001*b*] is an ontology editor for the semantic language OIL, produced by the University of Manchester. The interface is similar to Prot*é*g*é*, consisting of a set of tabs, with each tab offering a particular feature. It contains a reasoning engine which facilitates the development of detailed and consistent ontologies. A completed ontology can be exported into various forms such as RDFS, DAML+OIL and OWL.

The top most tab, named 'Classes', provides an interface for creating classes and describing their details (Figure 3.3). A class is described by its name and super or similar classes. A super or similar class can be specified by selecting the desired class from the list of all the created classes. Unlike Prot*é*g*é*, the class browser does not show the hierarchical class structure. As all classes are shown in a linear list, the hierarchical structure can be viewed by selecting the 'view hierarchy' feature. The interface also allows the specification of restrictions on the class in terms of the types of values a class's property may hold.

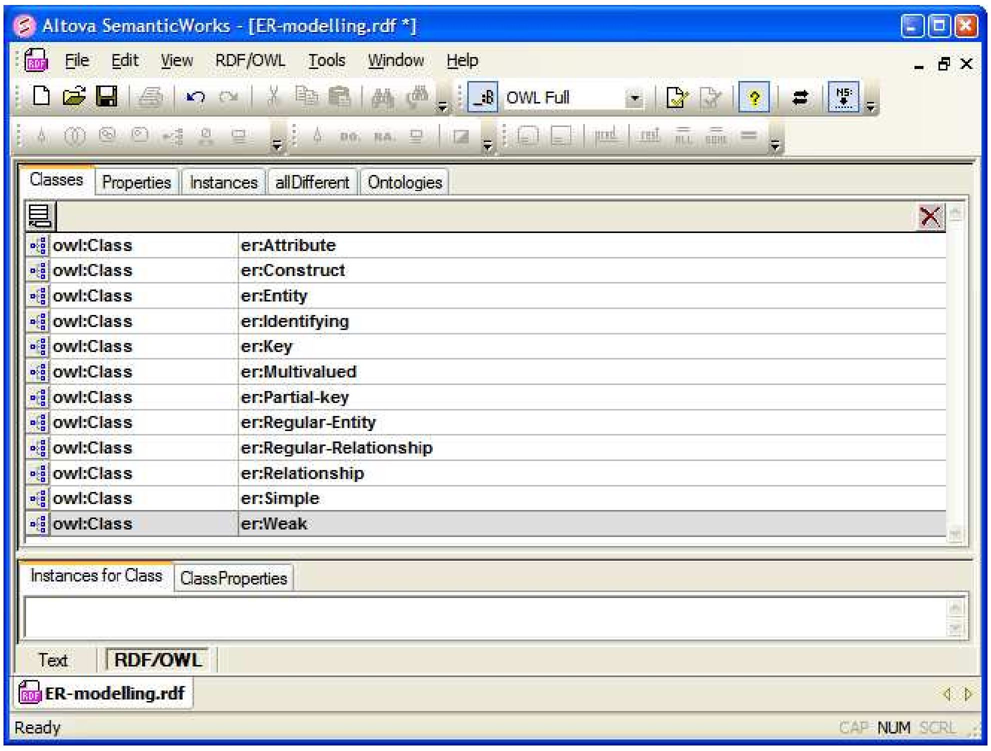
Both attributes and relationships between classes are specified as properties using the 'Properties' tab, described by their names, domain and range. By default, all new properties are created to hold instances of one or more concepts. The type of the property has to be explicitly changed by right clicking on the property and selecting 'set datatype' in order for it to hold a primitive type value such as integer, real, boolean, string etc. The domain of the property can be specified as a class, a restriction, a collection of instances or an expression. Properties may also support an inheritance structure similar to concepts where each of them can be assigned 'super properties'. However, OilEd does not contain a feature that allows the graphical hierarchical structure of properties to be viewed.

Instances of concepts can be created using the 'Individuals' tab. In order to create an instance, the class it belongs to should be specified. As properties are not directly associated with a class, the list of properties of the instance have to be selected. The properties of the instance have to be added from the list of all available properties, created using the 'Properties' tab. After adding a property, a pop-up window is displayed for entering its value. Each time a new instance is created, all its properties and their values have to be added.

The main drawback of OilEd is that both the 'Classes' tab and 'Properties' tab do not display the hierarchical structure of concepts and properties. Although the hierarchy of concepts can be viewed, it would be more helpful for the user if the hierarchical view is displayed instead of the flat list view in the main interface. The tool has no facility to even view the hierarchical structure of properties. Even though the hierarchical view for properties is not as essential as the hierarchical view for concepts, it would be useful for the user.

As properties are completely independent of concepts and because they cannot be not associated with concepts, composing a comprehensive list of properties becomes a hard task. The user has to manually go through each individual concept and add their properties to a linear list. Furthermore, as properties and their values have to be added each time an instance is created, it is extremely likely for users to miss properties. The task of adding properties each time an instance is created may also become very tedious.

3.2.3 SemanticWorks 2006


Altova SemanticWorks - [ER-modelling.rdf *]
File Edit View RDF/OWL Tools Window Help
OWL Full
Classes Properties Instances allDifferent Ontologies
Instances for Class ClassProperties
Text RDF/OWL
ER-modelling.rdf
Ready
CAP NUM SCRL

owl:Class	er:Attribute
owl:Class	er:Construct
owl:Class	er:Entity
owl:Class	er:Identifying
owl:Class	er:Key
owl:Class	er:Multivalued
owl:Class	er:Partial-key
owl:Class	er:Regular-Entity
owl:Class	er:Regular-Relationship
owl:Class	er:Relationship
owl:Class	er:Simple
owl:Class	er:Weak

Figure 3.4a: Interface of SemanticWorks 2006

SemanticWorks 2006 is a commercial tool developed by Altova [*Altova XML, Data Management, UML, and Web Services Tools* 2005] for composing RDF and OWL ontologies. It is a tool for visually editing ontologies. Its main interface is similar to both Prot*ég*é and OilEd with a set of tabs that contain interfaces for adding classes, properties, instances etc. The tool possesses the ability to perform syntax checks to ensure RDF schemas conform to the RDF specification. It also contains editors capable of syntax highlighting for manually editing the RDF and OWL source files.

The main interface of SemanticWorks, as shown in Figure 3.4a, consists of tabs named, 'Classes', 'Propertes', 'Instances' and 'allDifferent' and 'Ontologies'. The 'Classes' tab outlines the list of classes of an ontology in tabular

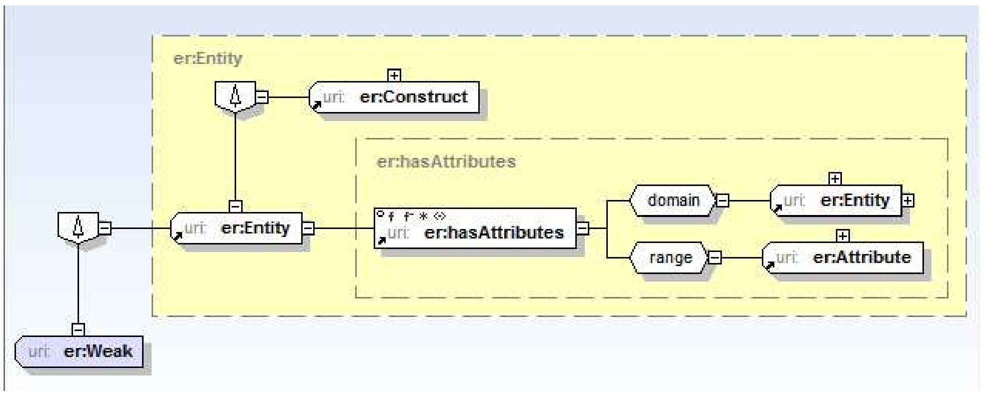


Figure 3.4b: Concept Visualisation of SemanticWorks 2006

form. A name space has to be declared prior to adding classes. A class which is described by its name has the form of <name-space>:<class-name>. The tabular interface only allows the creation of a class by specifying its name. The class hierarchy has to be modelled by individually specifying the super concepts of each defined class in the graphical view, shown in Figure 3.4b, accessed by clicking the 'detail view' button on the row representing the class. Within the graphical view, super classes relationship can be modelled creating a 'subClassOf' link and selecting the appropriate super class from the list of all available classes.

Both properties and relationships of concepts are modelled using the 'Properties' tab. Similar to the 'Classes' tab, properties are initially added by simply specifying their name. Other details of the property such as its domain and range have to be specified in the graphical view accessed via the 'detail view' button. The details of each property have to be set individually by clicking on their respective 'detail view' buttons. Within the graphical view, a 'domain' link can be created to specify the classes that the property belongs to. Similarly a 'range' link can be created to specify the type of values the property may hold. In the case of a property, which holds primitive values, its type is specified. Relationships between two or more classes are specified by selecting a class as the range. Properties added in the 'Properties' tab are reflected in the 'Class' tab, where both the tabular view and the graphical view show all properties that belong to each class.

The task of composing instances can be performed using the ‘Instances’ tab. First, an empty instance has to be created in the tabular view by specifying a name. Details of the instance such as its type and properties have to be specified in the graphical view. The type of the instance can be specified by selecting a class from the list of available classes. The properties of the instance can be specified by creating a ‘Predicate’ link and selecting the required properties from all available properties. The interface places no restrictions on what properties can be added to an instance. Even properties that do not belong to the class of the instance can be added. The value of a property can be populated by either selecting another instance, in the case of a relationship, or typing in a value, in the case of a primitive value. All newly created instances that belong to each class are outlined in the ‘Class’ view under the respective class.

Although SemanticWorks claim to provide support for visual creation of ontologies, the graphical view is limited to the scope of a single ontology element (concept, property, instance). The user has to first create an ontology element in the tabular view and switch to the graphical view in order to model the details of the particular element. Although this focusses attention to details of an individual element, users may loose sight of the big picture. As the tool does not contain a feature for displaying the entire concept hierarchy, the user’s cognitive load is increased.

The need to switch to the graphical view each time an ontology element is created can get tedious. Furthermore, modelling the details of each element in the graphical view is also tedious, as the creation of links is based on menu driven system. At each step of the link creation process, the selection has to be made by either right clicking or double clicking and selecting from a drop down menu.

The task of creating instances can also get very tedious as each property of an instance needs to be added manually even after selecting the type of the instance. The need to explicitly add a value object after adding a property to an instance further increases the burden. The need to manually add properties to instances also increases the risk of users missing properties of an instance. Furthermore, the need to explicitly add a value object also encourages the creation of incomplete instances.

3.3 Summary

This chapter provided details of domain knowledge authoring tools developed with the goal of alleviating the knowledge acquisition bottleneck. All the discussed authoring systems focused on producing knowledge for procedural tasks. Two of them, KnoMic and Diligent, were developed for acquiring knowledge for simulated environments. Their applicability is restricted to their respective environments. The other tools expect users to be adept in knowledge engineering (model-tracing in particular), alienating the average domain expert. Furthermore, the overall effort required to produce a domain model using these tools still remains high.

Conceptualising the domain or producing an ontology is an integral part of the process of authoring a domain model. This chapter included details of a selection of popular, state-of-the-art tools for producing domain ontologies. As these tools are designed for experts and as they contain a lot of features, novices are likely to find them difficult to use. Furthermore, although some tools make it possible to view a created ontology graphically, none of them allow the composition of an ontology graphically.

In order to fill the void of an authoring system that can acquire knowledge for non-procedural tasks, we designed and developed Constraint Acquisition System (CAS). CAS is capable of generating a domain model for both procedural and non-procedural tasks with the assistance of the domain expert. It was designed with the aim of overcoming the deficiencies of the currently available authoring systems. As domain ontologies play a central role in the authoring process of CAS, we also developed a restricted ontology workspace that allows ontologies to be created graphically. A detailed description of CAS is given in the following chapter (Chapter 4).

Chapter IV

A Constraint-based Domain Model Authoring System

Manually composing a constraint base for an intelligent tutoring system is a labour-intensive process that requires extensive expertise in constraint-based modelling and programming. Mitrovic [1998*a*] reported that she required approximately an hour to compose a constraint when SQL-Tutor was developed. She also mentioned that she may have required less time as she was the domain expert, knowledge engineer and the programmer. According to the approximation of one hour per constraint, composing the constraint base for SQL-Tutor, which contains over 700 constraints would take a period of approximately three months.

We have developed Constraint Acquisition System (CAS), an authoring system that generates the domain model required for constraint-based tutoring systems with the assistance of the domain expert. The goal of the system is to significantly reduce the time and effort required for composing constraint bases. It also fills in a research void by supporting knowledge acquisition for both procedural and non-procedural tasks.

We envisage that CAS would enable domain experts with minimal expertise in constraint-based modelling to produce constraint bases. CAS was designed to hide details of constraint implementation, such as the constraint language and components of constraints. The user is only required to model the domain in terms of an ontology and provide example problems and their solutions. The system assists the users in the tasks by guiding them to provide the required information. CAS analyses the information provided to generate constraints.

Although the system is designed to support users with minimal knowledge engineering expertise, it also offers utilities for experts in CBM. The system allows experts to directly modify constraints during the process of validating the system-generated constraints. Users are also provided with editors for directly adding new constraints to the domain model.

The remainder of the chapter is organised into seven sections. The following section outlines the process proposed for authoring domain knowledge. Section 4.2 includes a description of the architecture of CAS. Each phase of the domain authoring process is detailed in the remaining sections. The phase of modelling ontologies is detailed in section 4.3. Modelling a structure for solutions is detailed in section 4.4. Section 4.5 describes the syntax constraint generation algorithm. Section 4.6 includes details of adding sample problems and solutions. The algorithm for generating semantic constraints is described in Section 4.7. The final section includes details on validating the generated constraints.

4.1 Domain Authoring Process

Authoring knowledge using CAS is a semi-automated process with the assistance of an expert of the particular domain. The domain expert is entrusted to carry out a number of tasks, including modelling the domain as an ontology, providing problems and solutions for the domain and validating the generated constraints. Once the ontology is complete, CAS is capable of automatically generating the syntax constraints. The semantic constraints are generated by CAS by analysing the problems and solutions provided by the author.

The process of generating a domain model for a constraint-based tutoring system using CAS consists of six phases:

1. Modelling the domain as an ontology

2. Modelling the structure of solutions

3. Automatically generating syntax constraints

4. Providing sample problems and their solutions

5. Automatically generating semantic constraints

6. Validating the generated constraints

Initially the domain expert models the domain as an ontology (a specification of domain concepts) using CAS's ontology workspace. During this phase, the domain expert models concepts of the domain and specifies how they are related to other concepts (e.g. sub-, super-class relationships, part-of relationships etc). The task of modelling an ontology also includes specifying details of properties that belong to a concept.

Solutions should be decomposed into meaningful components during the second phase. This assists students using the final tutoring system by enabling them to focus only on a part of a solution while solving problems. Furthermore, the feedback provided by the tutoring system can also be made more specific by focusing on the component that contains errors. In the case of procedural tasks, the system also needs to know how many steps are involved in a problem-solving procedure and their details.

CAS analyses the modelled ontology for generating syntax constraints during the third phase. As ontologies contain a lot of information about the syntax of the domain, this phase involves translating the syntactical information embedded in ontologies into constraints. The system searches for particular features within the ontology and generates syntax constraints that describe them. The problem-solving procedure for procedural tasks also results in the generation of a set of syntax constraints that ensure the student solves the problem by following the correct sequence of steps.

The domain expert is requested to provide sample problems and their solutions, during phase four. In this phase, problems and solutions (based on the structure modelled in step 2) are added using the problem/solution interface. While providing solutions to a problem, the expert is encouraged to provide a collection of solutions for each problem illustrating different ways of solving the problem. We expect the domain expert to provide a set of problems that are significantly diverse, covering all parts of the domain.

The system uses machine learning techniques to reason with the provided problems and their solutions to generate semantic constraints. The constraint generation algorithm generates semantic constraints by comparing and contrasting two solutions for the same problem. The generated constraints are generalised or specialised during consequent analysis of other solutions.

The final step involves validating the constraints generated by the system. This phase achieves both validation of the constraints and adding meaningful feedback messages to constraints. The system presents a high-level description of each constraint to the domain expert, who can label the constraint as either valid or invalid. The domain expert is expected to add a meaningful feedback message to the constraint if it is correct. The invalid constraints can be corrected by the author by directly modifying the constraints in question. The author can also provide more example problems and solutions and instruct the system to regenerate the whole constraint base.

4.2 Architecture

CAS is implemented according to the client-server architecture depicted in Figure 4.1. The server is implemented in Allegro Common Lisp using the AServe light weight web server. It is responsible for storing and retrieving all components of the domain model such as the ontology, problems, solutions and constraints. The client, implemented as a Java application, provides the interfaces for the domain expert to compose the required domain-dependent components. The Java application is also capable of generating syntax and semantic constraints, enabling it to function as a stand alone application if a connection to the server is not available. The client stores and retrieves the domain model components from the server through HTTP requests.

The server is implemented as a simple web server that listens to HTTP requests and returns the appropriate response. Users have to be authenticated prior to being granted permission to access any data. Authenticated users can request for the ontology that they modelled, any problems and solutions that have been added and constraints (syntax and semantic) generated during a prior session. Authenticated users can also store domain model components composed using the client.

The client, implemented in Java, consists of an interface, the authoring controller, constraint generators and a set of managers responsible for persisting components of the domain model. The interface is a collection of three components that are used during different stages of the authoring

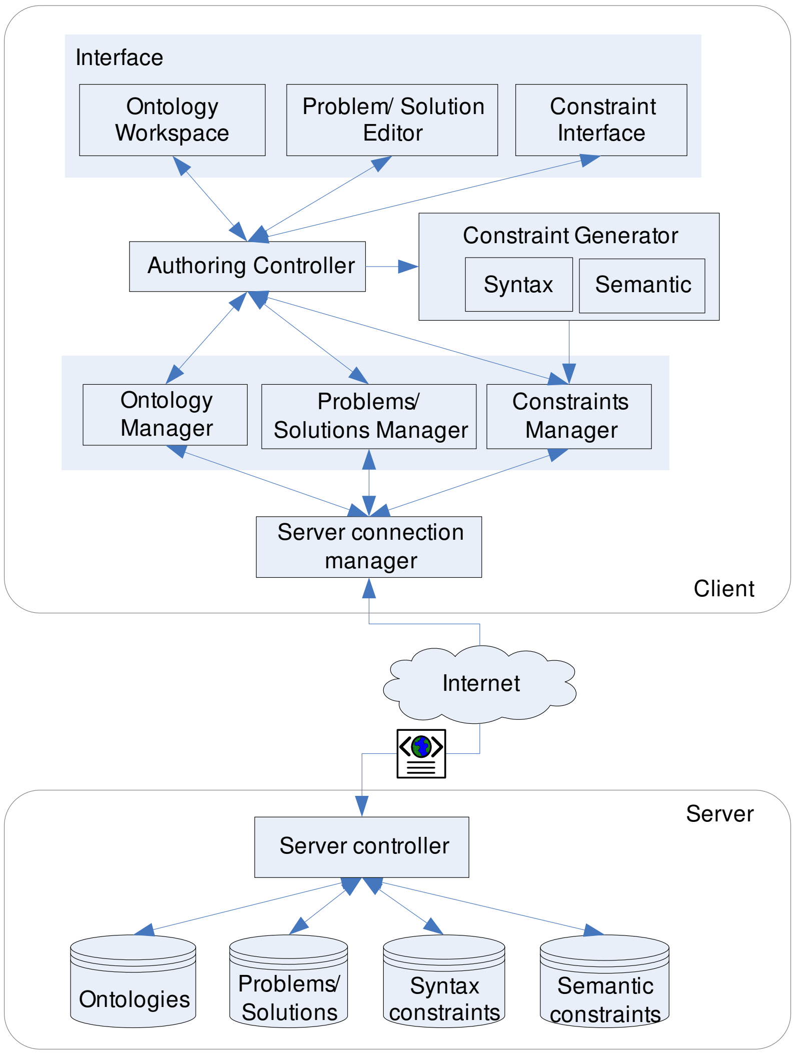


Figure 4.1: Architecture of the Constraint Acquisition System

process. It consists of an *ontology workspace*, *problem/solution interface* and a *constraint interface*. The *ontology workspace* is provided for modelling an ontology of the domain. Problems and their solutions can be added using the *problem/solution interface*. It assists the authors by providing a form with input boxes based on the properties of the element's type (i.e. the concept). The constraint validation phase uses the *constraint interface* for validating constraints and adding meaningful feedback messages.

The *authoring controller* is the driving engine of the system. It ensures that the proper authoring process is followed by guiding the user to complete the required tasks. It directs the user to the appropriate interface for the particular task. The *authoring controller* communicates with the corresponding manager to retrieve the required domain model component from the server and automatically loads it in the corresponding interface module.

CAS includes two constraint generators: *syntax constraint generator* and *semantic constraint generator*. The *syntax constraint generator* is responsible for generating syntax constraints by analysing the ontology. On the other hand, the *semantic constraint generator* analyses the problems and their solutions along with the ontology for generating semantic constraints. The generated constraints are passed on to the *constraint manager* for persistence.

The system contains an *ontology manager*, *problem/solution manager* and *constraint manager* for persisting components of domain models. The domain model components are stored in memory as Java objects and are later converted by their respective managers to an XML representation for transmission over the network. The XML representation of the domain model is transferred to the server via the server connection manager, where it is persisted in a data store. Retrieving data from the server also involves the domain model managers where the XML representations received are used to rebuild the object structure.

4.3 Modelling Domain's Ontology

Domain ontologies play a central role in the knowledge acquisition process. They are used in every phase of the domain knowledge acquisition process. The structure of solutions (modelled during the second phase) is based on the concepts of the ontology. During the third phase, the syntax constraints are generated directly from the ontology. The fourth phase, which involves providing problems and solutions, also depends on the ontology, as solutions are composed by adding instances of ontological concepts. Finally, the last phase where constraints are presented to the domain expert for validation and populating feedback messages, also uses the ontology to categorise the set of generated constraints into meaningful sets.

4.3.1 Domain Ontology

An ontology is an "explicit specification of a conceptualisation" [Gruber 1993]. In other words an ontology is a description of concepts of a domain. It also contains information about properties that describe concepts and describes inter-relationships between concepts. Typically an ontology takes a hierarchical structure with super- and sub-concepts.

An example ontology that models the domain of ER modelling is depicted in Figure 4.2. It has *Construct* as the top level concept and the three main types of constructs in ER modelling, *Relationship*[1], *Entity* and *Attribute* as its sub-concepts. *Relationship* is specialised into *Regular* and *Identifying*, the two relationship types in ER modelling. Similarly *Entity* is specialised into *Regular* and *Weak*, while *Attribute* is specialised to *Simple* and *Composite*. The sub-concepts of *Simple attribute* include *Key*, *Partial key*, *Simple*, *Derived* and *Multi-valued*.

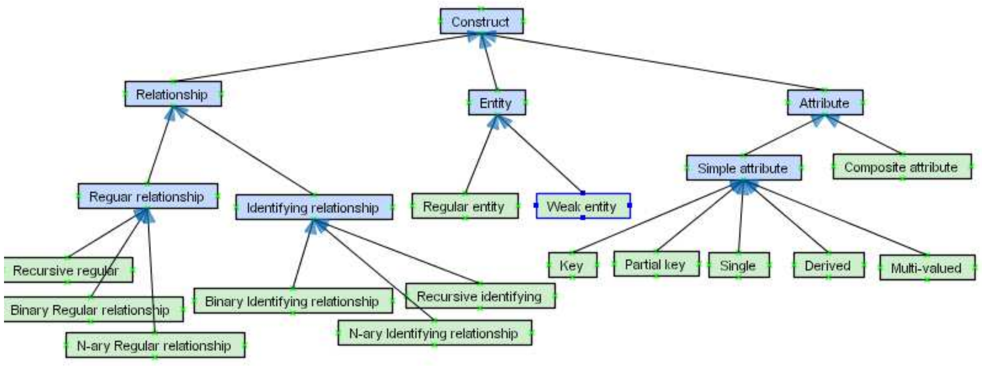


Figure 4.2: Ontology for Entity Relationship Modelling

Each concept is described by a set of properties, which is inherited by its sub-concepts. In the example ER modelling ontology (Figure 4.2), the *Construct* concept has a property named *Tag*, which holds a unique identifier for the construct and a *Name* property that holds the name of the construct. As all other concepts in the ontology are sub-concepts of *Construct*, the two properties, *Tag* and *Name* are inherited by all other concepts

[1] The relationship construct in ER modelling is different from relationships between concepts of an ontology

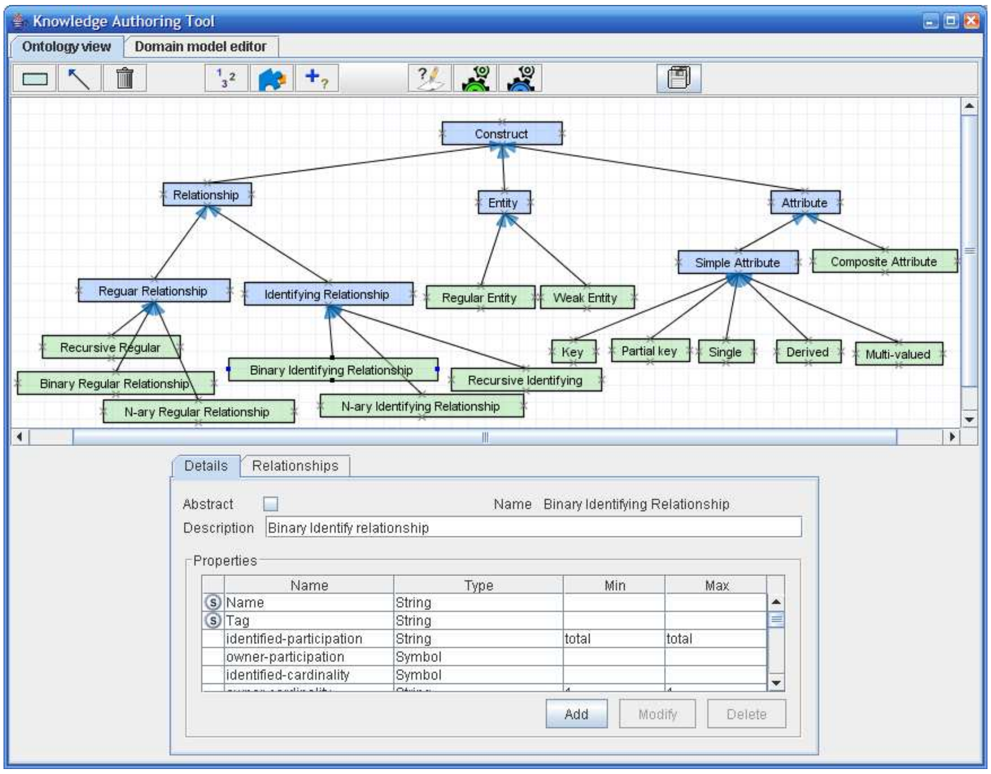
of the ontology. Concepts can also have their own properties in addition to the inherited properties. For example, the *Binary Identifying Relationship* concept requires properties for recording participation and cardinality of the 'owner' and the 'identified' entity. Consequently, the *Binary Identifying Relationship* concept was modelled to contain four properties: *Identified participation*, *Owner participation*, *Identified cardinality* and *Owner cardinality.*

An ontology contains relationships between two or more concepts. Figure 4.2 shows all generalisation relationships ("is-a"). The generalisation relationships are depicted using arrows, where each concept's super-concept is pointed using the arrowhead. Ontologies may also contain association relationships ("has") between concepts. An association relationship indicates that the related concept makes a reference to the other. The ER ontology in Figure 4.2, although not explicitly shown, contains a number of association relationships. For example, it contains a relationship between *Entity* and *Attribute*, denoting that Entities have Attributes.

4.3.2 Ontology Workspace

CAS contains an ontology workspace for composing domain ontologies. The interface, as depicted in Figure 4.3, consists of two panels: top and bottom. The top panel contains the ontology workspace, which allows users to compose a graphical representation of the ontology. The ontology workspace represents concepts using rectangles and generalisation relationship using arrows. As it has no restrictions in placing concepts within the workspace, the user can position concepts to resemble a hierarchical structure. The bottom panel shows the properties and relationships of the currently selected concept. It also allows users to add new properties and relationships.

Figure 4.3 shows the example ER modelling ontology developed using the ontology workspace (discussed in Section 4.3.1). The *Binary Identifying Relationship* concept is selected in the ontology workspace and its properties are shown in the bottom panel. The top two properties, *Tag* and *Name* are properties inherited from the *Construct* concept. The inherited properties are distinguished by the "S" icon at the start of the row. The other four properties of *Identified participation*, *Owner participation*, *Identified cardinality* and *Owner cardinality* are properties of the *Binary Identifying Relationship* concept. Most of the shown properties are of type 'String', except *Owner participation* and *Owner cardinality*, which are of type 'Symbol'. The 'Symbol' type specifies that the value of a property is one of the pre-defined set of values. The set of values that a 'Symbol' type property may hold has to be specified during the definition of the property. For example, the *Identified cardinality* property has only two choices: '1' or 'n'. Properties can also have default values. Both *Identified cardinality* property and *Identified participation* property have default values of '1' and 'total' respectively.



Abstract ☐ Name Binary Identifying Relationship

Description Binary Identify relationship

Properties

	Name	Type	Min	Max
Ⓢ	Name	String		
Ⓢ	Tag	String		
	identified-participation	String	total	total
	owner-participation	Symbol		
	identified-cardinality	Symbol		

Figure 4.3: Interface of Ontology Workspace

The properties of a concept are added by using the property input interface shown in Figure 4.4. A property is described by its name and the type of values it may hold. Properties can be of type 'String', 'Integer', 'Float', 'Symbol' or 'Boolean'. The interface allows the specification of a default value for 'String' and 'Boolean' properties. It allows the specification of a range of values, in terms of a minimum and maximum, for 'Integer' and 'Float' properties. When creating a property of type 'Symbol', its domain has to be specified as a collection of distinct values. A 'Symbol' property can can only hold one of the values specified in its domain.

Other restrictions on properties include specifying that the value of a property is unique, optional or can contain multiple values. In situations where properties can hold multiple values, the exact number of values that it may hold can be specified using the 'at least' and 'at most' fields of the property interface. The 'at least' field specified the minimum number of values a property may hold and the 'at most' field specifies the maximum number of values a property may hold.

Figure 4.4 depicts the *identified-participation* property of the *Binary identifying relationship* concept. The property is of type *string* and has a default value of 'total'. Furthermore as the 'at least' and 'at most' fields are both set to '1' the *identified participation* property can only hold one value.

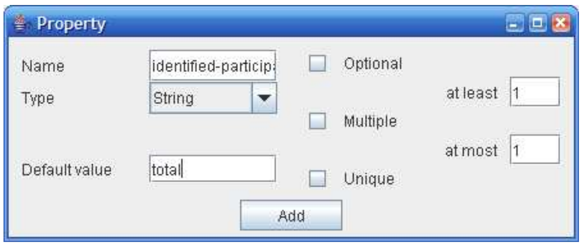
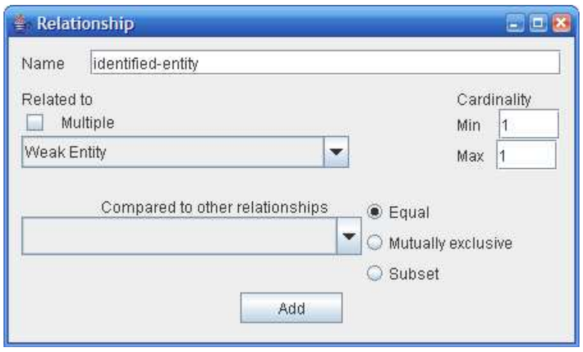

Name: identified-particip
Type: String
Default value: total
at least: 1
at most: 1

Figure 4.4: Property Interface

Relationships between concepts are specified using the relationship composing interface shown in Figure 4.5. Adding a relationship involves specifying its name, selecting the concepts involved and specifying any restrictions for elements participating in the relationship. All relationships are between the concept selected in the graphical representation of the ontology and the concept chosen from the relationship composing interface. A collection of related concepts can be chosen from the interface by ticking the 'Multiple' option. This specifies the types of elements (in terms of concepts) that can be involved in the relationship.

The interface can be used to specify the minimum and maximum number of elements that are required to participate in the relationship. The 'min' and 'max' input boxes can be populated to specify the minimum and maximum cardinality of the relationship. The interface also allows the restriction of elements participating in two relationships, such as whether an element must participate in two relationships. This can be achieved by selecting a previously defined relationship under the 'Compared to other relationships' selection box and selecting either 'equal', 'mutually exclusive' or 'subset'. This is useful for denoting restrictions such as "an attribute can belong to either an entity or a relationship, but not both".


Relationship

Name: identified-entity

Related to

☐ Multiple

Weak Entity

Cardinality

Min: 1

Max: 1

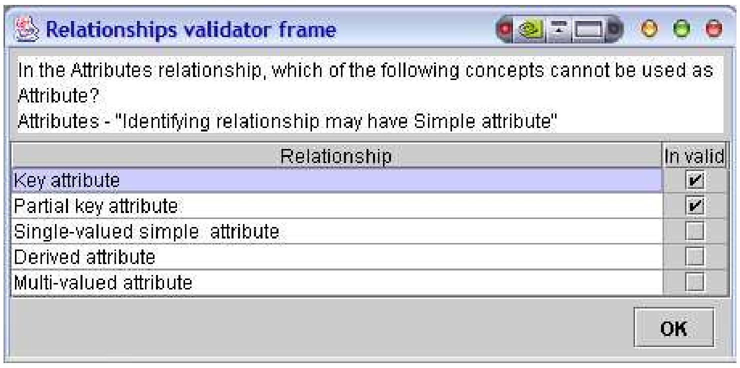
Compared to other relationships

[✓ Equal] Mutually exclusive Subset

Add

Figure 4.5: Relationship Interface

As ontologies play a central role in the knowledge acquisition process, it is imperative that the relationships are correctly defined. In order to ensure that the relationships are correct and are not too general, the system engages the author in a dialogue by using a pop-up window. During this dialogue the author is presented with specialisations of concepts involved in the relationship and is asked to label the specialisations that violate the principles of the domain. As an example, consider the relationship between *Identifying Relationship* and *Attribute*. As shown in Figure 4.6, the initial question asks whether each specialisation of *Attribute* (*Key*, *Partial key*, *Single-valued* etc) is allowed to participate in the *Attributes* relationship. As *Key* or *Partial key* attributes cannot participate as attributes of any *Identifying Relationship* according to the principles of ER modelling, the author would indicate that the two specialisations are invalid. The ontology workspace uses the outcome of this dialogue to automatically replace the original relationship with a more specific one.


Relationships validator frame

In the Attributes relationship, which of the following concepts cannot be used as Attribute?
Attributes - "Identifying relationship may have Simple attribute"

Relationship	In valid
Key attribute	✓
Partial key attribute	✓
Single-valued simple attribute	☐
Derived attribute	☐
Multi-valued attribute	☐

Figure 4.6: Relationship Validation Interface

Ultimately, *Binary Identifying Relationship* concept is involved in three association relationships: *Attributes*, *Owner* and *Identified entity*. The *Attributes* relationship exists between *Binary Identifying Relationship* and three types of attributes (*Single-valued Simple*, *Derived* and *Multi-value*) with no restrictions on the cardinality. The *Owner* relationship with *Regular Entity* has a minimum cardinality of 1. The *Identified entity* relationship, as shown in Figure 4.5, between *Binary Identifying Relationship* and *Weak Entity* has a minimum and maximum cardinality of 1.

The decision to design and implement an ontology editor specifically for CAS was taken after evaluating a variety of commercial and research tools developed for composing ontologies (see Section 3.2 for details). Although the discussed tools were sophisticated and possessed the ability to represent complicated syntactical domain restrictions in the form of axioms, this made them less intuitive to use. They were designed for knowledge engineering experts. Consequently, novices in knowledge engineering would struggle with the steep learning curve.

One of the goals of this research is to enable domain experts with little knowledge engineering background to produce tutoring systems for their domains. In order to achieve this goal, the system should be easy and intuitive to use. We decided that an ontology editor had to be built specifically for this project to reduce the required training for users. The editor was designed to compose ontologies in a manner analogous to using a drawing program.

The ease of use of the ontology editor was achieved by restricting its expressive power. In comparison to Protégé [Protege 2006], where axioms can be specified as logical expressions, the ontology editor of CAS only has a set of syntactic restrictions that can be specified through the interface. We believe the limited set of restrictions is sufficient for the purpose of generating syntax constraints for most domains.

4.3.3 Internal Representation

The domain model is internally represented in memory as a set of objects, using the class structure outlined in the UML diagram in Figure 4.7. The domain model consists of three main components: ontology, problems and constraints. Ontology is represented as a set of *Concept* objects, where each *Concept* contains a list of *PropertyType* objects and a list of *RelationshipType* objects. The *RelationshipType* objects keeps track of both super and sub-concepts, as well as relationships between other concepts. The *PropertyType* stores all information about a property.

The domain model contains a list of *Problem* objects and each problem may contain several *Solution* objects. Each solution consists of a set of *Components*, where each component relates to a specific *ComponentType* in the solution structure. *Components* of a solution contain a set of *elements*, which are instances of concepts, described by its *Properties* and *Relationships*.

CAS generates a set of semantic and syntax constraints by analysing the ontology and the provided problems. The satisfaction and relevance condition of *Constraints* for the domain are generated as a set of *Tests* that are combined using conjunctions and disjunctions. A test may look for a particular error in the student's solution or compare a section of the student solution against a section of the ideal solution.

The internal representation of the ontology is stored in the central server as XML. All three components of the domain model (ontology, problems and constraints) are converted from objects to their respective XML representations by their managers and transferred to the server via the server connection manager. The server stores the received XML files in the appropriate location under the particular domain. In order to retrieve domain model components,

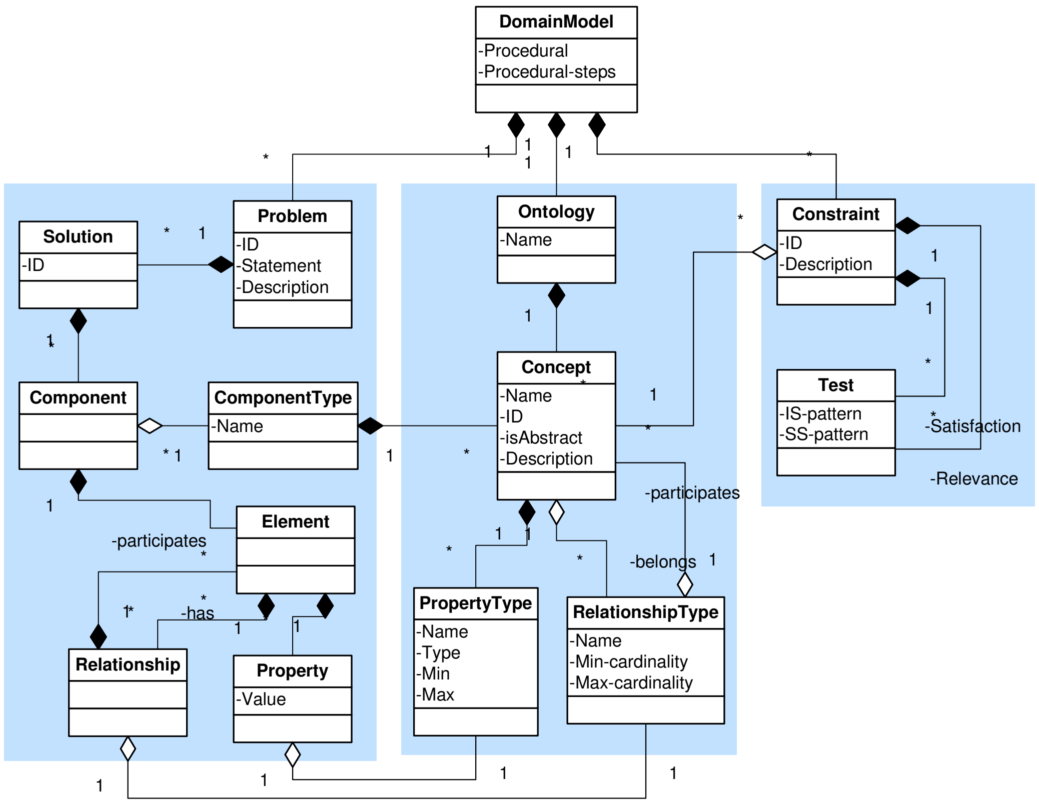


Figure 4.7: UML Class Diagram of Internal Representation

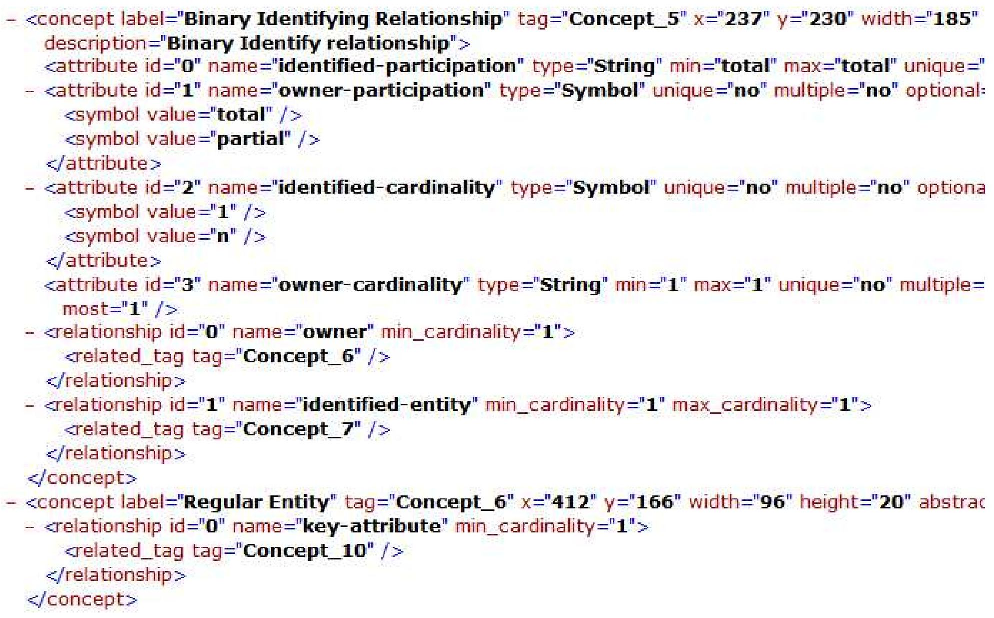
the client initiates a request to the server, that returns the XML representation of the required domain model component. The manager responsible for the particular domain component, uses the XML representation to re-build the object structure in memory.

An extract of the XML representation of the ontology for ER modelling (Figure 4.2) shown in Figure 4.8, illustrates that the ontology is represented using XML tags defined specifically for this project. It uses *concept*, *attribute* and *relationship* tags to record details of concepts, properties and relationships respectively. Each concept is assigned an automatically generated tag to uniquely identify it, which is used to specify the concepts involved in a relationship.

The XML representation of an ontology also includes positional and dimensional details of each concept. This information is recorded for the purpose of regenerating the same layout of concepts as produced by the author.

Although it is possible to use an algorithm to automatically layout the concepts to form a hierarchical structure, we have not implemented this feature, since ontology modelling was not the focus of this research. Furthermore, to ensure that the author retrieves their exact ontology, it is essential to record the geometrical details.

Although the ontology is stored using custom XML tags, it can be transformed to a standard ontology representation form such as DAML [DAML 2006] via an XSLT transformation. CAS was developed to save ontologies using its own XML representation as it directly reflected its internal representation of ontologies. This sped up the progress of the research project, avoiding the need for extensive research into ontology standards.



```
- <concept label="Binary Identifying Relationship" tag="Concept_5" x="237" y="230" width="185"
    description="Binary Identify relationship">
    <attribute id="0" name="identified-participation" type="String" min="total" max="total" unique="
  - <attribute id="1" name="owner-participation" type="Symbol" unique="no" multiple="no" optional=
      <symbol value="total" />
      <symbol value="partial" />
    </attribute>
  - <attribute id="2" name="identified-cardinality" type="Symbol" unique="no" multiple="no" optiona
      <symbol value="1" />
      <symbol value="n" />
    </attribute>
    <attribute id="3" name="owner-cardinality" type="String" min="1" max="1" unique="no" multiple=
      most="1" />
  - <relationship id="0" name="owner" min_cardinality="1">
      <related_tag tag="Concept_6" />
    </relationship>
  - <relationship id="1" name="identified-entity" min_cardinality="1" max_cardinality="1">
      <related_tag tag="Concept_7" />
    </relationship>
  </concept>
- <concept label="Regular Entity" tag="Concept_6" x="412" y="166" width="96" height="20" abstrac
  - <relationship id="0" name="key-attribute" min_cardinality="1">
      <related_tag tag="Concept_10" />
    </relationship>
  </concept>
```

Figure 4.8: XML Representation of ER Modelling Ontology

The internal representations of problems and constraints are also converted to XML to be transferred to the server. Both are represented in XML using custom tags defined for this project. They use the unique identifiers assigned to concepts of the ontology for referencing the ontology XML. For example, the concept id is used to specify the concept that a constraint belongs to.

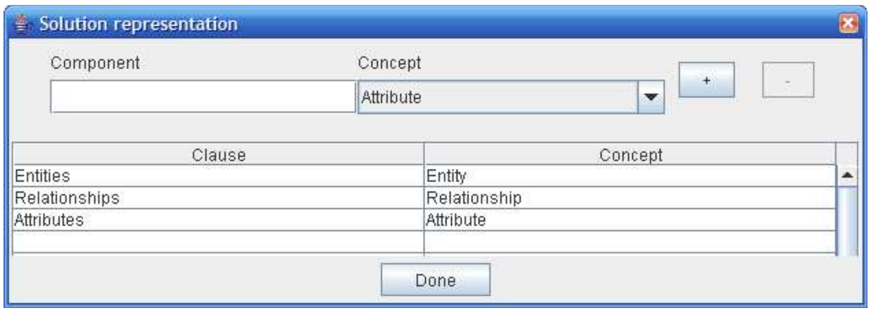
4.4 Modelling the Structure of Solutions

The structure of a solution is the decomposition of the solution into meaningful components. The objectives of decomposing a solution are two fold: reducing the student's cognitive load and providing more focused feedback. Presenting the student with a goal structure with components that need to be populated to compile a full solution reduces the cognitive load on the student. On the other hand, having the solution decomposed into components enable the tutoring system to analyse components of the solution independently and provide feedback specific to the particular component. Although this does not eliminate the need for analysing the solution as a whole, it enables the system to provide feedback focussed at the component level.

The number of components of a solution varies from domain to domain and depends on the form of the expected solution. For example, if the final tutoring system is to present multiple choice questions with a single solution to the student, the solution structure would only have a single component that contains the student's choice. On the other hand, a tutoring system for the domain of algebraic equations that expects an algebraic equation as the solution might consist of solutions with two components: left-hand-side expression and right-hand-side expression.

As the ontology describes domain concepts, the types of elements that can be held by each component can be specified in terms of concepts. Consequently, the task of modelling the solution structure involves decomposing a solution into components and identifying the type of elements (in terms of concepts) that each component may hold. The solution structure composition interface, as shown in Figure 4.9, allows authors to model the solution structure as a table. The figure shows that ER models consists of three components: entities, relationships and attributes.

CAS was developed to be able to author domain knowledge for both procedural as well as non-procedural tasks. Procedural tasks such as adding two fractions, solving a mathematical equation require a strict procedure to be followed to accomplish their goals. On the other hand, non-procedural tasks such as designing a database, writing a computer program do not have a rigid procedure that has to be followed to achieve their goals.



Clause	Concept
Entities	Entity
Relationships	Relationship
Attributes	Attribute

Figure 4.9: ER Modelling Solution Structure

Typically, procedural tasks have a well defined set of steps that need to be completed in order to arrive at the solution. CAS requires details of the problem-solving procedure for generating constraints for such domains. The author is required to outline the complete list of problem-solving steps as well as the solution components relevant for each step.

CAS's interface initially prompts the user to specify the total number of steps involved in solving a problem. Subsequently, the interface provides a table for adding the name of each step and the corresponding component of the solution that is affected as part of the step. For example, consider a tutoring system that provides a practice environment for applying physics equations, where students are given a textual description of a scenario and they have to calculate a specific value. The procedure for producing the final solution would involve:

1. Identifying the given variables and their values
2. Identifying the correct equation
3. Substituting the values of known variables in the equation
4. Solving the equation

The structure of the expected solution has to be modelled to reflect the the steps involved. The solutions would also consist of four components: known variables, chosen equation, equation with substituted values, final solution. The complete structure of solutions for the domain is given in Table 4.1.

Component	Concept
Known Variables	*Variable*
Chosen equation	*Physics equation*
Equation with substituted values	*Substituted equation*
Final solution	*Numeric value*

Table 4.1: Solution Structure for Applying Physics Equations

4.5 Syntax Constraints Generation

An ontology contains a lot of information about the syntax of the domain. Composing a domain ontology is much easier and quicker than composing syntax constraints that are responsible for ensuing that the student has used the correct syntax. The goal of the syntax constraint generation algorithm is to extract all useful syntactic information from the ontology and to translate it into syntax constraints for the domain model.

Syntax constraints are generated by analysing relationships between concepts and properties of concepts specified in the ontology. The algorithm generates constraints based on restrictions specified for relationships and properties [Suraweera, Mitrovic & Martin 2004*b*], which are applicable to both procedural and non-procedural tasks. A set of extra constraints for procedural tasks to ensure that the student adheres to the correct problem-solving procedure is also generated as part of the syntax constraint generation process.

The syntax constraint generation algorithm is outlined in Figure 4.10. During step 1, this algorithm produces constraints by analysing relationships between concepts. The minimum and maximum cardinalities specified in relationships result in syntax constraints that ensure the correct number of elements participate in relationships within a student solution. The ontology editor allows specifying a restriction on the elements participating in a relationship by specifying that they should also participate in another relationship. The syntax constraint generator also produces constraints to ensure such restrictions are verified in student solutions.

The constraint generation algorithm makes use of a set of templates for

1. For each relationship between concepts
    - Generate constraints that ensure the correct number of elements participate in relationship (cardinality)
    - If a restriction on elements participating in another relationship is specified, generate a constraint that ensures the restriction is upheld
2. For each specialised relationship marked as invalid during interactions with the relationship validation dialogue
    - Generate a constraint that ensures elements in a solution do not participate in the relationship
3. For each concept property
    - If min and max are set, generate constraints that ensure value of property is within the specified range
    - If unique restriction is set, generate constraint that ensures uniqueness
    - Generate a constraint that ensures the type of value for each property is correct (i.e. integer/float)
    - Generate a constraint that ensures that a valid value is chosen for a property of type symbol
    - If property can hold multiple values, generate constraints that ensure that correct number of values are supplied
4. If domain is procedural, for each problem solving step
    - Generate a constraint to ensure that the correct solution components are populated for the step

Figure 4.10: Syntax Constraint Generation Algorithm

generating syntax constraints. The current set of templates produce constraints in a high-level representation. The collection of templates used for generating constraints by analysing relationships is given in Figure 4.11. The syntax constraint generator produces constraints by replacing these tags within the templates, such as <concept-name>.

As an example, consider the 'Identified-entity' relationship in Figure 4.5. A constraint can be generated from this relationship, which says "each *Binary Identifying Relationship* contained in the student solution must have one and only one *Weak Entity* as its 'Identified entity"' (Figure 4.12). The relevance condition focuses on instances of *Binary Identifying Relationships* in the student's solution and the satisfaction condition asserts that each one of them has to have exactly one *Weak Entity* as its 'identified-entity'.

The results of the relationship validation dialogue that engages the user after the creation of a relationship, also contributes towards the generation of syntax constraints. The algorithm produces a constraint for each specialised relationship labelled as invalid by the author, during step 2. They are generated using the template shown in Figure 4.13.

As an example, consider the scenario of adding a relationship called 'has-attribute' between *Regular Entity* and *Attribute* in the ontology shown in Figure 4.2. During the relationship validation dialogue, the domain expert would realise and indicate that *Regular Entities* cannot contain *Partial-key Attributes*. The syntax constraint generation process produces a constraint which ensures that a regular entity can not contain any *Partial-key Attributes* (Figure 4.14). Its relevance condition checks for the existence of a *Regular Entity* in the entities component of the student solution, and its satisfaction condition enforces that the *Regular Entity* should not contain any *Partial-key Attributes.*

After generating constraints from relationships, the algorithm analyses all the properties defined in the ontology for generating constraints (step 3). During this phase, the constraint generator creates a constraint for each restriction on the domain and range of a property. Such restrictions include the minimum and maximum values allowed, whether the property is multi-valued or unique. The list of templates used for generating constraints from property restrictions are given in Figure 4.15.

For example, the 'Identified-participation' property (shown in Figure 4.4) of a *Binary Identifying Relationship*, is restricted to have only 'total' as its value. This restriction on the property results in the constraint shown in Figure 4.16.

- *Max and min cardinalities are both equal*
  Relevance: SS <component-name> has a <concept-name> element
  Satisfaction: It has to have exactly <cardinality> element of <related-concept(s)>

- *Max and min cardinalities set, but unequal*
  Relevance: SS <component-name> has a <concept-name> element
  Satisfaction: It has to have at least <cardinality> element of <related-concept(s)>

  Relevance: SS <component-name> has a <concept-name> element
  Satisfaction: It can not have more than <cardinality> element of <related-concept(s)>

- *Neither max or min cardinalities set*
  Relevance: SS <component-name> has a <concept-name> element
  Satisfaction: It can only have <related-concept(s)> elements participating in <relationship-name>

- *Participants restricted against participating in another relationship*
  Relevance: <concept-name> element in SS <component-name> participates in <relationship-name> relationship
  Satisfaction: It has to also participate in <restricted-relationship-name>

  Relevance: <concept-name> element in SS <component-name> participates in <relationship-name> relationship
  Satisfaction: At least one such element has to participate in <restricted-relationship-name>

  Relevance: <concept-name> element in SS <component-name> participates in <relationship-name> relationship
  Satisfaction: It cannot participate in <restricted-relationship-name>

Figure 4.11: Syntax Constraint Templates for Relationship Restrictions

Relevance: SS *Relationships* component has a *Binary Identifying Relationship* element
Satisfaction: It has to have exactly one element of *Weak Entity*

Figure 4.12: Example Syntax Constraint from a Relationship Restriction

Relevance: SS <component-name> has a <concept-name> element
Satisfaction: It can not contain a <related-concept> element that participates in <relationship-name> relationship

Figure 4.13: Syntax Constraint Templates for Relationships Labelled as Invalid

Relevance: *Entities* component of SS has a *Regular Entity*
Satisfaction: It can not contain a *Partial-key Attribute* element that participates in 'has-attributes' relationship

Figure 4.14: Example Syntax Constraint from Relationship Validation Dialogue

The sample constraint verifies that the student has not violated the rule in ER modelling which states that the participation of the identified entity should be always 'total'. Its relevance condition checks for the existence of a *Binary Identifying Relationship* in the student solution, and satisfaction condition ensures that for each such relationship, the value of the 'Identified-participation' is 'total'.

All constraints generated by analysing properties and relationships are applicable to both procedural and non-procedural tasks. In procedural tasks, the tutoring system needs to ensure that the student follows the correct sequence of steps. In order to achieve this, the domain model should contain a set of syntax constraints that verify that the student has at least made an attempt at composing the part of the solution relevant to the current step. The syntax constraint generator produces a constraint for each problem

- *Min and max are set and are equal*
  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> must be <min-value>

- *Min and max are set and not equal*
  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> must be greater than <min-value>

  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> has to be less than <max-value>

- *Unique restriction is set*
  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> must be unique

- *Ensuring numbers are specified for properties of type integer/float*
  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> must be a number

- *Ensuring that value of symbol type property is valid*
  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> must be <value1> OR value of <property-name> must be <value2> ...

- *Property can have multiple values*
  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> must have at least <at-least-count> values

  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> can not have more than <at-most-count> values

Figure 4.15: Syntax Constraint Templates for Property Restrictions

Relevance: *Relationships* component of SS has a *Binary Identifying Relationship*
Satisfaction: Its 'Identified-participation' property must be set to 'total'

Figure 4.16: Example Syntax Constraint from Property Restriction

solving step using the template shown in Figure 4.17, during step 4.

Relevance: current-step is <step-identifier>
AND <solution-component-associated-with-step> component of IS is not empty
Satisfaction: <solution-component-associated-with-step> component of SS should not be empty

Figure 4.17: Syntax Constraint Template for Procedural Task

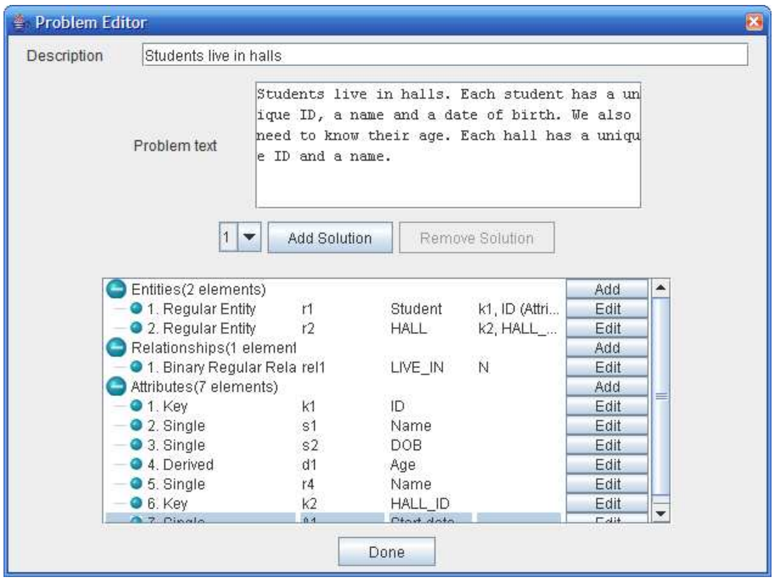
The constraint in Figure 4.18 is an example of a constraint for the procedural task of applying physics equations. The constraint becomes relevant if the step that the student is working on is 'identifying known variables' (the *current_step* variable keeps track of the progress made by the student within the problem-solving procedure). The satisfaction condition verifies that the student has made an attempt to populate the *variables* component (i.e. *variables* component is not empty).

Relevance: current_step is 'identify-known-variables'
AND *variables* component of IS is not empty
Satisfaction: *variables* component of SS should not be empty

Figure 4.18: Example Constraint for Procedural Task

4.6 Adding Problems and Solutions

The author is required to provide sample problems along with sets of solutions depicting alternative ways of solving them using the solution composition interface. Problems are described by their problem statement, whereas the solutions are specified as sets of elements that relate to each other. The solution structure has to be modelled in phase 2 of the authoring process prior to entering problems and solutions.



Description: Students live in halls

Problem text:

```
Students live in halls. Each student has a un
ique ID, a name and a date of birth. We also
need to know their age. Each hall has a uniqu
e ID and a name.
```

1

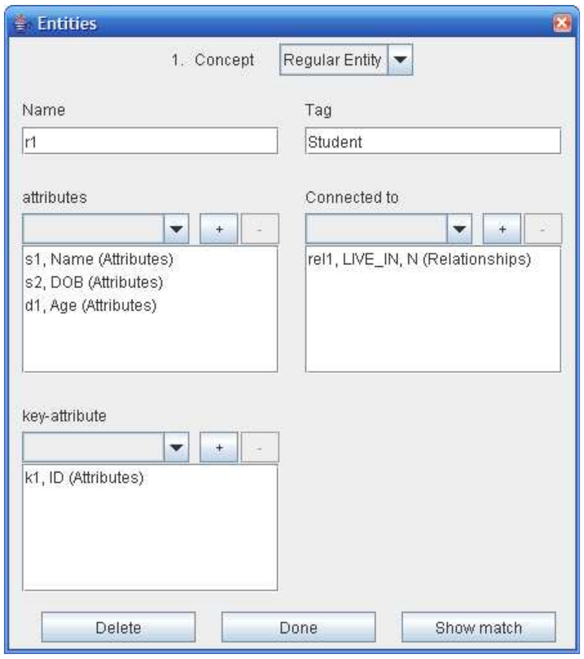
Entities(2 elements)			
1. Regular Entity	r1	Student	k1, ID (Attri...
2. Regular Entity	r2	HALL	k2, HALL_...
Relationships(1 element			
1. Binary Regular Rela	rel1	LIVE_IN	N
Attributes(7 elements)			
1. Key	k1	ID	
2. Single	s1	Name	
3. Single	s2	DOB	
4. Derived	d1	Age	
5. Single	r4	Name	
6. Key	k2	HALL_ID	

Figure 4.19: Solution Composing Interface

Solutions are composed by populating the components of the solution structure using the interface illustrated in Figure 4.19. The interface was designed to simplify the task of composing solutions by outlining the solution structure to the user. Users can create and add elements of the concept type or its sub-concepts specified for each component during the solution modelling phase.

The solution composition interface is designed to compose solutions that strictly adhere to the ontology. Elements can be created using a form-based interface, as shown in Figure 4.20, generated to reflect the concept definition. The forms contain input boxes for specifying values for concept's properties and drop-down-lists for selecting elements involved in relationships. Relationships between elements are specified by selecting the desired element and adding it to the respective container. In order for a relationship to be specified, the element has to have been created previously.

Figure 4.20 shows the interface for creating an instance of *Regular Entity* concept. It contains two text boxes for populating the *name* and the *tag* properties. It also contains three drop-down-lists for selecting elements that participate in each relationship. Elements participating in a relationship have to be selected from the drop-down-list and added to the container under it by clicking the '+' button. Elements in the container can be removed by selecting it and clicking the '-' button. The Figure shows participating elements after being added to containers. For example, the entity in Figure 4.20 has three *attribute* elements participating in the *attributes* relationship: 'Name', 'DOB' and 'Age'.



1. Concept: Regular Entity

Name: r1

Tag: Student

attributes:
s1, Name (Attributes)
s2, DOB (Attributes)
d1, Age (Attributes)

Connected to:
rel1, LIVE_IN, N (Relationships)

key-attribute:
k1, ID (Attributes)

Figure 4.20: Element Creation Interface

The system expects a series of solutions depicting alternate ways of solving the same problem. The interface reduces the amount of effort required for composing alternate solutions by allowing the author to transform a copy of the first solution into the desired solution. The user can modify the previously specified solution by adding, modifying or dropping elements. As most alternative solutions have a high degree of similarity, this feature significantly reduces the workload of the author.

As the constraints are generated by analysing the similarities and differences between two solutions to the same problem, it is imperative for the system to be able to accurately identify similar elements between two solutions to a problem. The system achieves this by maintaining a map of matching elements between the initial solution and alternative solutions. Although forcing the author to compose alternate solutions by modifying the initial solution simplifies the task of mapping matching elements between the two different solutions, the map needs to be updated when new elements are added. The system initially attempts to automatically find matches for new elements and if a successful match is found, the author is consulted to validate it. If the system fails to find a matching element, the author is requested to select a matching element, if one exists.

4.7 Semantic Constraints Generation

Semantic constraints are generated by an algorithm that learns from examples [Suraweera, Mitrovic & Martin 2005] using the problems and solutions provided by the author during phase four of the authoring process. The domain expert is encouraged to provide a collection of solutions for each problem to illustrate different ways of solving the same problem. The list of solutions assists the algorithm to generate constraints that are capable of correctly validating a student's attempt by comparing it to a single correct solution regardless of the methodology used by the student.

The algorithm generates semantic constraints by analysing pairs of solutions and identifying similarities and differences between them. The constraints generated from a pair of solutions contribute towards either generalising or specialising constraints in the main constraint base. The resulting set of constraints is capable of identifying different correct solutions produced using alternative problem-solving strategies, by comparing them to a single ideal solution.

The semantic constraints generation algorithm, outlined in Figure 4.21, generates new semantic constraints by analysing a pair of solutions. The constraints are generated by identifying similarities and differences between the solutions. Constraints generated from a single pair of solutions are generalised or specialised as other pairs of solutions are analysed.

1. For each problem P_i
2. For each pair of solutions S_i & S_j
3. Generate a set of new constraints N
4. Evaluate each constraint N_i in set N against each previously analysed solution,
    - If N_i is violated, generalise or specialise N_i until it is satisfied
5. Evaluate each constraint CB_i in main constraint base, CB, against solutions S_i & S_j used for generating N,
    - If CB_i is violated, generalise or specialise CB_i to satisfy S_i & S_j
6. For each constraint N_i in set N
    - If main constraint base has a constraint C_i that has the same relevance condition but a different satisfaction condition to N_i, add the satisfaction condition of N_i as a disjunctive test to the satisfaction condition of C_i
    - else if N_i does not exist in main constraint base, add it to main constraint base
7. Generate set of constraints that check for extra elements

Figure 4.21: Semantic Constraint Generation Algorithm

The constraint generation algorithm analyses all the problems and their solutions provided by the domain expert focusing on a single problem at a time. The algorithm generates a new set of constraints (represented as N in the algorithm shown in Figure 4.21) for each pair of solutions for a problem. It makes use of all possible permutations of solution pairs, including solutions compared to themselves.

A new set of constraints from a pair of solutions is generated using the algorithm outlined in Figure 4.22. It generates semantic constraints to compare each element of the ideal solution against its matching element in the student solution. It also generates constraints that ensure the relationships between elements of the ideal solution also exists between the corresponding elements of the student solution.

The set of constraints N, generated from a pair of solutions, is evaluated against all solutions previously used for generating constraints. This step (step 4) ensures that the newly generated set of constraints are consistent with all solutions analysed so far. During the evaluations, if a constraint in constraint set N is found to be violated by a solution, it is either generalised or specialised according to the algorithm outlined in Figure 4.28 to satisfy that solution.

The next step (step 5) ensures that constraints in the main constraint base are consistent with the pair of solutions used to generate the new constraint set N. Each constraint in the main constraint base is evaluated against the solution pair, and the violated constraints are either specialised or generalised using the same algorithm used for specialising or generalising new constraints (Figure 4.28).

After constraints in both constraint set N and the main constraint base are generalised, the main constraint base is updated with constraints in set N. During this step, if the main constraint base contains a constraint C_i that has the same relevance condition as a constraint N_i (from constraint set N) with a different satisfaction condition, the satisfaction condition of N_i is added to the satisfaction condition of C_i as a disjunctive test. The main constraint base is expanded by adding the constraints in constraint set N that do not exist in the main constraint base. This step ensures that the constraint base grows with each iteration.

Finally, at the end of analysing all problems and their solutions, a constraint base containing sufficiently generalised constraints is produced. These constraints are used for generating a set of constraints that check the student solution for extra elements. This is achieved by reversing the constraints that exist in the constraint base. For example, the constraint base for ER modelling would contain a constraint which ensures that "the student solution contains a matching regular entity for each regular entity in the ideal solution". This constraint can be reversed as "the ideal solution should contain a regular entity for each regular entity in the student solution", which ensures that the student solution does not contain any extra regular entities.

Generating Constraints from a Pair of Solutions

New semantic constraints are generated by analysing similarities and differences between elements of two solutions, using the algorithm outlined in Figure 4.22. The algorithm treats the first solutions as the ideal solution and the other as the student solution. Consequently, the set of constraints generated is dependent upon the order in which the two solutions are supplied. The algorithm generates constraints for each element and relationship occurrence in the ideal solution. These generated constraints are generalised by introducing variables and wild cards.

```
1. Treat $S_i$ as the ideal solution (IS) and $S_j$ as the student solution
   (SS)

2. For each element $A$ in the IS

     a. Generate a constraint that asserts that if IS contains the
        element $A$, SS should contain its matching element
     b. For each relationship that element is involved with, generate
        constraints that ensures that the relationship holds
        between the corresponding elements of the SS

3. Generalise the properties of similar constraints by introducing
   variables and wild cards
```

Figure 4.22: Algorithm for Generating Constraints from a Pair of Solutions

A constraint is generated for each pair of matching elements in the ideal solution and the student solution. In the case of non-procedural tasks, a constraint generated as the result of a pair of elements asserts that if the ideal solution contains the particular element, the student solution should also contain its matching element. As the system keeps a map of similar elements during the phase of entering problems and solutions, determining matching elements between the two solutions is straightforward. The matches automatically determined by the system are also accurate as they are validated by the author.

Figure 4.23 contains an example problem for the domain of ER modelling and two alternative correct solutions. Both solutions are structurally equivalent, and the only difference is with the names used for constructs. As the tags (used to uniquely identify constructs) are consistent between the two solutions, they can be used to identify matching elements between the two ER diagrams.

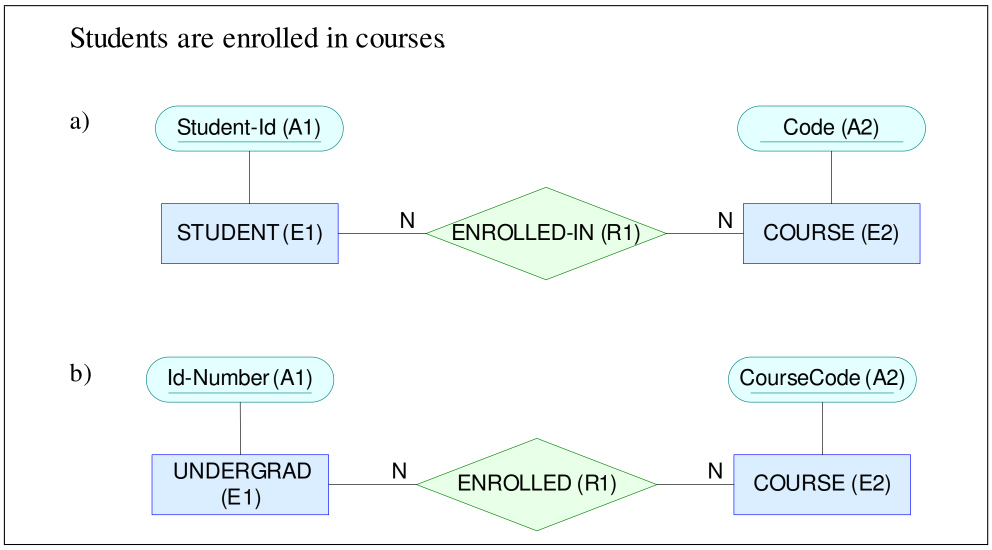


Figure 4.23: A Sample ER Modelling Problem and Two Correct Solutions

The constraint generation algorithm (Figure 4.22) generates a constraint for each pair of matching elements in step 2a. Considering that solution *a* in Figure 4.23 was treated as the ideal solution and solution *b* was treated as the student solution, the constraint generator produces two constraints from the two *Regular Entity* elements, ‘STUDENT’ and ‘COURSE’ (see Figure 4.24). The first of the two constraints ensures that the student solution contains a

‘STUDENT’ *Regular Entity* element for each equivalent element (‘UNDERGRAD’) found in the ideal solution. The second constraint ensures that the student solution contains a ‘COURSE’ *Regular Entity* element for each equal element found in the ideal solution.

(*a*)
Relevance: *Entities* component of IS has a (STUDENT, E1) *Regular Entity*
Satisfaction: *Entities* component of SS has a (UNDERGRAD, E1) *Regular Entity*
(*b*)
Relevance: *Entities* component of IS has a (COURSE, E2) *Regular Entity*
Satisfaction: *Entities* component of SS has a (COURSE, E2) *Regular Entity*

Figure 4.24: Semantic Constraints from Matching Pairs of Elements

The constraints generated from a matching pair of elements are extremely specific. They are only applicable to the problem that was used to generate them. Furthermore, the constraints can only validate the student solution chosen to generate them. Constraint *a* in Figure 4.24 only checks for the existence of a ‘UNDERGRAD’ entity element in the student solution and Constraint *b* only checks for the existence of a ‘COURSE’ entity. During step 3 of the constraint generation algorithm (Figure 4.22) the two constraints are replaced by a more general constraint.

Step 3 of the constraint generation algorithm replaces similar constraints with a single general constraint. It produces a general constraint from the two constraints in Figure 4.24 by introducing variables and wild cards in place of property values. A variable is introduced to a property if its value is constant during every occasion it appears within the constraint and different between constraints. In the case of the first constraint in Figure 4.24, the second property (tag property) of regular entity elements always has a value of ‘E1’, and its value in the second constraint is ‘E2’. Consequently, it can be replaced by a variable, represented as ‘?var’. Although the value of the first property is constant within the second constraint, it has two values within the first constraint. So, the value of the property is generalised to a wild card, represented as '?*'. This denotes that the property's value of the ideal solution element and the same property's value of its corresponding element in the student solution can be different. The generalised constraint that replaces the two constraints in Figure 4.24 is shown in Figure 4.25. It denotes that the student solution must contain a *Regular Entity* element for each such element and that both their tags should be the same.

Relevance: *Entities* component of IS has a (?*, ?var1) *Regular Entity*
Satisfaction: *Entities* component of SS has a (?*, ?var1) *Regular Entity*

Figure 4.25: Sample Generalised Semantic Constraint

Step 2 of the constraint generation algorithm also produces a set of constraints for each relationship that the focussed element is involved with. As an example, consider the 'STUDENT' element in the first solution of Figure 4.23. It participates in two ontological relationships with 'Student-id' *Key Attribute* and 'ENROLLED-IN' *Regular Relationship*.

The constraints generated based on a relationship is only relevant for problems where the particular ontological relationship exists in the ideal solution. In order for a ontological relationship to exist, the solution must also contain the two elements that participate in the relationship. As a result, the constraint generation algorithm produces three constraints for each occurrence of a relationship. Two of them ensure that the student solution contains the two elements that are supposed to participate in the relationship. The other ensures that the two elements in the student solution actually participates in the relationship.

The constraints generated from the relationship between 'STUDENT' *Regular Entity* and 'Student-Id' *Key Attribute* in the first solution of Figure 4.23 are given in Figure 4.26. The relevance conditions of all three constraint contain three tests to ensure that the ideal solution contains the two elements ('STUDENT' and 'Student-Id') and that they participate in the relationship, named 'has-key-attribute'. The first constraint is relevant when

1. **Relevance:** *Entities* component of IS has a (STUDENT, E1) *Regular Entity*
   AND *Attributes* component of IS has a (Student-Id, A1) *Key Attribute*
   AND IS (STUDENT, E1) *Regular Entity* participates in 'has-key-attribute' relationship with (Student-Id, A1) *Key Attribute*
   AND *Entities* component of SS has a (UNDERGRAD, E1) *Regular Entity*
   Satisfaction: *Attributes* component of SS has a (Id-Number, A1) *Key Attribute*

2. **Relevance:** *Entities* component of IS has a (STUDENT, E1) *Regular Entity*
   AND *Attributes* component of IS has a (Student-Id, A1) *Key Attribute*
   AND IS (STUDENT, E1) *Regular Entity* participates in 'has-key-attribute' relationship with (Student-Id, A1) *Key Attribute*
   AND Attributes component of SS has a (Id-Number, A1) *Key Attribute*
   Satisfaction: *Entities* component of SS has a (UNDERGRAD, E1) *Regular Entity*

3. **Relevance:** *Entities* component of IS has a (STUDENT, E1) *Regular Entity*
   AND *Attributes* component of IS has a (Student-Id, A1) *Key Attribute*
   AND IS (STUDENT, E1) *Regular Entity* participates in 'has-key-attribute' relationship with (Student-Id, A1) *Key Attribute*
   AND *Entities* component of SS has a (UNDERGRAD, E1) *Regular Entity*
   AND *Attributes* component of SS has a (Id-Number, A1) *Key Attribute*
   Satisfaction: SS (UNDERGRAD, E1) *Regular Entity* participates in 'has-key-attribute' relationship with (Id-Number, A1) *Key Attribute*

Figure 4.26: Example Semantic Constraints from a Relationship

the student solution contains a matching element for the first element ('UNDERGRAD'). It has a satisfaction condition which ensures that the student solution contains a matching element for the second element that participates in the relationship. Similarly, the second constraint ensures that the student solution contains the first element that participates in the relationship. The third constraint is only relevant when both matching elements exist in the student solution. It is satisfied when both the elements of the student solution participates in the specific relationship. These constraints are also generalised by introducing variables and wild cards in a manner similar to generalising constraints that ensure matching elements between the ideal and student solutions.

The constraints generated for procedural tasks include two extra relevance tests, to ensure only the constraints that are applicable for a particular step become relevant. The relevance condition of a procedural constraint test whether the student is currently working on a particular step and that he/she has at least made an attempt to solve the task involved for the step. Consider a constraint for the task of solving physics equations, that ensures that the student solution should contain a variable v for each variable v that appears in the ideal solution (Figure 4.27). It contains two extra tests in the relevance condition specific to procedural constraints: test for whether the *current_step* is of a particular value and a test to check that the solution component relevant for the particular step is not empty.

Relevance: current_step = 'identify-known-variables'
    AND NOT *variables* component of SS component = nil
    AND *variables* component of IS has a *variable* **v**
Satisfaction: *variables* component of SS has a *variable* **v**

Figure 4.27: Example Semantic Constraint for Procedural Task

Generalisation

Step 3 of the semantic constraint generation algorithm (Figure 4.21) produces a set of constraints (denoted as constraint set N) from a pair of solutions.

These constraints are evaluated against the previously analysed solutions in order to ensure that they are consistent with the solutions (step 4). Although these solutions are correct, this process may result in violated constraints, as a result of constraints being too specific or general. Such constraints have to be either generalised or specialised in order to correctly validate all correct solutions and identify errors in incorrect solutions.

1. If constraint set (*C-set*), which does not contain the violated constraint V, has a similar but a more restrictive constraint C,
   - replace V with C and exit.
2. Else if *C-set* has a constraint C that has the same relevance condition but a different satisfaction condition to V,
   - add the satisfaction condition of C as a disjunctive test to the satisfaction condition of V,
   - replace C with V and exit
3. Else attempt to restrict the relevance condition of constraint V to be irrelevant for solution pair S_i & S_j
   - Find solution pair S_x & S_y that satisfies constraint V
   - Look for extra constructs in S_i not found in S_x
   - Use extra construct to form a test that can be added to relevance condition of V
   - Add new test to relevance condition of V using a conjunction
4. If restriction failed, constraint must be incorrect, label it as invalid and drop it

Figure 4.28: Constraint Generalisation/Specialisation Algorithm

The conflicting constraints in constraint set N are generalised or specialised using the algorithm outlined Figure 4.28. It is applied by the constraint generation process to remedy each violated constraint individually.

The violated constraint is generalised or specialised until it satisfies the violated pair of solutions. However, if the algorithm fails to sufficiently generalise or specialise the violated constraint, it is labelled as invalid and removed from the constraint set. The system keeps track of incorrect constraints to ensure that similar newly generated constraints do not get added to the main constraint base.

Initially, the algorithm attempts to restrict the violated constraint by replacing it with a similar constraint that has already been restricted. It searches the main constraint base (denoted as *C-set*) for a constraint that is similar to but more restrictive than the violated constraint. Such a constraint would contain one or more extra relevance tests than the violated constraint. It must also contain the same set of satisfaction tests, but it may contain extra disjunctive tests. If such a constraint is found, the algorithm simply uses it to replace the violated constraint and completes the specialisation/generalisation process.

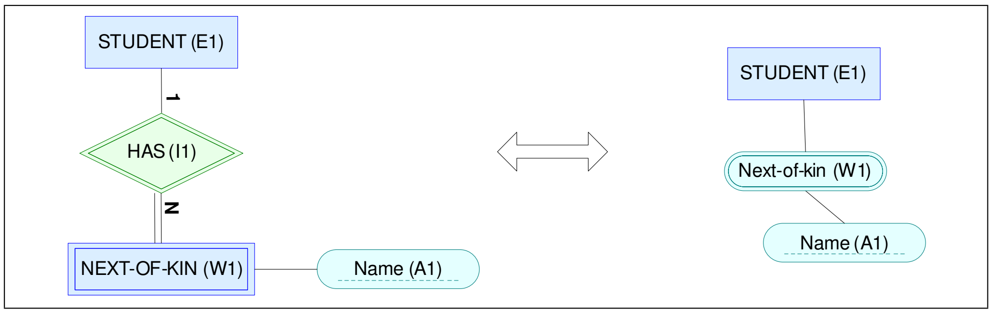


Figure 4.29: Equivalent ER models: A *Weak Entity* Represented as a *Composite Multi-valued Attribute*

Failure in attempting to find a similar constraint that has already been specialised, the algorithm attempts to generalise the violated constraint. It first searches for a constraint in the main constraint base (*C-set*) with the same relevance as the violated constraint but a different satisfaction condition. If the search is successful, both constraints are replaced by a new constraint that has the same relevance and a disjunctively combined satisfaction.

Consider the example given in Figure 4.30. The violated constraint V ensures that the student solution contains a matching *Weak Entity* for each such entity in the ideal solution. Since in ER modelling, *Weak Entities* can also be represented as *Composite Multi-valued Attributes* (Figure 4.29) constraint V gets violated if a student has opted to use a *Composite Multi-valued Attribute* instead of a *Weak Entity.* In the example, the main constraint-set (*C-set*) contains a constraint that allows this equivalent situation. As both constraints have the same relevance condition, they can be replaced by a single constraint that has the same relevance condition and the two satisfaction conditions joined using a disjunction (OR).

Violated constraint, V:

Relevance: *Entities* component of IS has a *Weak Entity*
Satisfaction: *Entities* component of SS has a *Weak Entity*

Constraint from *C-set*:

Relevance: *Entities* component of IS has a *Weak Entity*
Satisfaction: *Attributes* component of SS has a *Composite Multi-valued Attribute*

Combined constraint:

Relevance: *Entities* component of IS has a *Weak Entity*
Satisfaction: *Entities* component of SS has a *Weak Entity*
OR *Attributes* component of SS has a *Composite Multi-valued Attribute*

Figure 4.30: Generalising Violated Constraint by Adding a Disjunctive Satisfaction Test

During step 3 of the algorithm in Figure 4.28, the relevance condition of constraint V is restricted to make it irrelevant for the solution which violated it. This is achieved by first searching for a pair of solutions that satisfies the constraint. Although a constraint is violated by a new solution pair, a previously analysed pair of solutions is guaranteed to satisfy the same constraint. This is because the algorithm generates a constraint from a pair of solutions and it satisfies the pair of solutions used to generate it. Any significant differences between the satisfied solutions pair and the violated solutions pair is used to restrict the relevance condition of V.

Consider the example of the violated constraint given in Figure 4.31. The constraint ensures that the student solution contains a *Partial-key Attribute* for each such attribute found in the ideal solution. However, this constraint gets violated if a *Weak Entity* is represented as a *Composite Multi-valued Attribute* as shown in Figure 4.29. Even though a *Weak Entity* has a *Partial-key Attribute*, A1, when it is represented as a *Composite Multi-valued Attribute*, A1 is represented as a *Single Attribute*. The ER model (a) in Figure 4.29, which satisfies the constraint contains a *Weak Entity* which is not found in the ER model (b) in Figure 4.29. This difference is added as a new conjunctive test to the relevance condition of the violated constraint as shown in Figure 4.31.

Violated constraint, V:

> **Relevance:** *Attributes* component of IS has a *Partial-key Attribute*
> **Satisfaction:** *Attributes* component of SS has a *Partial-key Attribute*

Modified constraint, V:

> **Relevance:** *Attributes* component of IS has a *Partial-key Attribute*
> AND *Entities* component of SS has a *Weak entity*
> **Satisfaction:** *Attributes* component of SS has a *Partial-key Attribute*

Figure 4.31: Violated Semantic Constraint Resolved with New Relevance Test

The constraint specialisation/generalisation algorithm (Figure 4.28) is also used to resolve conflicting constraints from the main constraint base found during step 5 of the semantic constraint generation process (Figure 4.21). When the algorithm is applied for resolving constraints from the main constraint base, the constraint set N is searched for similar constraints. In other words, constraint set N is treated as the *C-set* in the constraint specialisation/generalisation algorithm.

4.8 Constraint Validation

The final phase of the constraint generation process involves validating the constraints generated by the system with the assistance of the domain expert. This phase ensures that the generated constraints are accurate and that the resulting domain model can be used in a tutoring system. In addition to validating the constraints, this phase also forces the domain expert to add meaningful feedback messages to constraints.

During the constraint validation process, the author is presented with a system-generated, high-level hint for each constraint (Figure 4.32). These hints along with the constraints' high-level relevance and satisfaction conditions describe the meaning of each constraint. The hints are generated by analysing each constraint and determining whether they check for missing or extra elements/relationships. The hints are generated with the use of a set of templates. For example, the hints for constraints that deal with missing elements are produced using the template of "Some required <concept-name> elements are missing". During hint generation, the <concept-name> tag is replaced with the appropriate name(s) of the concept (See Figure 4.32).

Relevance: *Entities* component of IS has a **?var1** *Regular Entity*
Satisfaction: *Entities* component of SS has a **?var1** *Regular Entity*
Generated description: "Some required *Regular Entity* elements are missing"

Figure 4.32: Example Description of Constraint

The author is required to go through each constraint's description and verify that they are valid. If the constraint is valid, the author has to provide a meaningful hint message to be presented to the student as part of their feedback. For example, the author may provide "Make sure you have created all the required regular entities. Your solution is missing some entities" as a hint for the constraint in Figure 4.32.

During the process of inspecting constraints, the author may come across invalid constraints. These constraints can be labelled as invalid and would be removed from the constraint base. The author can also revisit step four of the authoring process and provide more example problems and solutions demonstrating why the constraint is invalid. The constraint base can be regenerated by executing the semantic constraint generator with the updated problem set.

Regenerating the constraint base results in a new set of constraints. Consequently, any modifications to feedback messages would also be lost. The author is warned of these consequences prior to regeneration. The authors are encouraged to only modify feedback messages of constraints after they are satisfied that the constraint base accounts for all pedagogically significant problem states.

Users with some expertise in composing constraints can directly modify the constraints using the set of editors provided by CAS. The constraints would have to modified in the final low-level Lisp representation. CAS is currently not capable of translating the high-level constraints into the low-level representation. However, both the constraint generators can be enhanced to automatically produce constraints in the executable Lisp form (See Annex A for a detailed outline of an example translation). In the case of the syntax constraint generator the current set of templates have to be replaced with templates that produce Lisp code. As the constraint generation algorithm generalises and specialises constraints using the high-level representation, the final high-level constraints have to be transformed into the Lisp code. This can be achieved by identifying a mapping between the two representations and producing a set of templates. As properties have been generalised by introducing variables and wild cards, they can be used for pattern matching within the student solution.

4.9 Summary

The chapter provided a detailed description of CAS, the authoring system that was developed with the goal of reducing the time and effort required for producing constraint bases for ITSs. It included details of CAS's architecture as well as the authoring process for composing a domain model. Each phase of the authoring process was explained with examples from the domain of ER modelling.

The true effectiveness of CAS can only be determined through evaluation. We conducted a series of evaluation studies that evaluated the effectiveness and the usefulness of CAS. The following chapter (Chapter 5) presents details on the three evaluation studies conducted.

Chapter V

Evaluation

Evaluation is an integral part of research. We strongly believe that true effectiveness of any research system can only be determined with empirical evaluations. The usefulness and effectiveness of CAS was evaluated in a series of studies that focussed on different aspects of the system.

Domain ontologies play a central role in CAS. We conducted an experiment (study 1) to validate our hypothesis that domain ontologies are useful even in the process of manually authoring a domain model. The experiment was carried out with novice domain model authors with a tool designed to encourage the use of ontologies. It revealed that domain ontologies do indeed facilitate the creation of complete constraint bases.

Upon the completion of CAS, its effectiveness was evaluated (study 2) by using it to produce constraint-bases for the three domains of ER modelling, Fraction addition and Data normalisation. The constraint-bases were developed by the same author to ensure consistency between the domains. The analysis of the generated constraints revealed that CAS was extremely effective in producing constraints for the evaluated domains.

Finally, a comprehensive evaluation (study 3) was carried out with a group of novice domain model authors to evaluate the effectiveness of the system as a whole. They were assigned the task of producing a complete domain model runnable in WETAS for the domain of adding fractions using CAS. The evaluation showed that CAS was capable of generating highly accurate constraint bases even with the assistance of novices. It also showed that CAS reduced the overall effort required to produce domain models.

The remainder of the chapter contains details of the three evaluation studies carried out. Section 5.1 contains details of the experiment conducted to evaluate the usefulness of domain ontologies. It includes details of the process followed, the results observed and the conclusions of the study. The following section describes the three domains that were used to evaluate the effectiveness of CAS and the generated sets of constraints. It also includes details of the analyses performed to evaluate the quality of the generated constraint bases. Details of the evaluation of CAS's effectiveness with novice domain model authors is available in Section 5.3. It includes details of the task allocated to domain authors, observed results and conclusions of the evaluation. Finally, the last section presents a discussion of all the evaluation studies.

5.1 *Usefulness of Ontologies for Manually Composing Domain Models (Study 1)*

It is widely accepted that the quality of the knowledge base is one of the main the determining factor for the quality of diagnosis and instruction in ITSs. The knowledge base of a constraint-based tutor consists of a collection of constraints which formalises the syntactic and semantic rules of the domain. A knowledge base is incomplete if it is missing one or more constraints that account for significant problem states. Incomplete knowledge bases result in student solutions falsely diagnosed as correct. So careful engineering and iterative testing has to be performed in order to avoid producing incomplete constraint bases [Mitrovic et al. 2003].

We believe that it is highly beneficial for the author to develop a domain ontology even when the constraint set is developed manually, because this helps the author to reflect on the domain. Such an activity would enhance the author's understanding of the domain and therefore be a helpful tool when identifying constraints. We also believe that categorising constraints according to the ontology would assist the authoring process. As a consequence, the author would only be required to focus on constraints related to a single concept, reducing their working memory load. The use of ontologies may encourage authors to produce more complete constraint bases.

We hypothesized that the task of composing an ontology and organising the constraints according to its concepts would assist in manually composing constraints; it would help the author to produce a more complete constraint base, as the task of building an ontology requires the author to reflect on the entire domain and explicitly outline its concepts.

To evaluate our hypothesis, we conducted an empirical study [Suraweera, Mitrovic & Martin 2004*a*]. The study involved authors producing constraints using a tool that supported organising constraints according to concepts of an ontology (called the domain model composition tool).

5.1.1 Process

The evaluation involved 18 students enrolled in the 2003 graduate course on Intelligent Tutoring Systems at the University of Canterbury. They were assigned the task of building an Intelligent Tutor using WETAS for teaching adjectives in the English language. They were asked to compose constraints using the domain model composition tool, which supported modelling an ontology and categorising constraints according to the concepts of the ontology. The participants were allocated a total time period of three weeks to complete the task.

The participants had attended 13 lectures on ITSs, including five on CBM, prior to assigning the task. They also had a 50 minute presentation on WETAS, and were given a description of the task, instructions on how to write constraints, and the section on adjectives from the Clutterbuck English text book [Clutterbuck 1990] for vocabulary. They were free to explore LBITS [Martin & Mitrovic 2002*a*], a tutor developed in WETAS that teaches simple vocabulary skills. The participants had access to the "last two letters" puzzle of LBITS. The puzzle involved determining a set of words that satisfied the given clues, with the first two letters of each word being the same as the last two letters of the previous one. All domain specific components, including its ontology, the constraints and problems, were available.

The participants were given a set of example problems that the adjectives tutor was supposed to present to its students (referred to as ITS users in the rest of the Section). The adjectives tutor should present sentences to be completed by ITS users by providing the correct form of a given adjective. "My sister is much ________ than me (wise)" is an example problem.

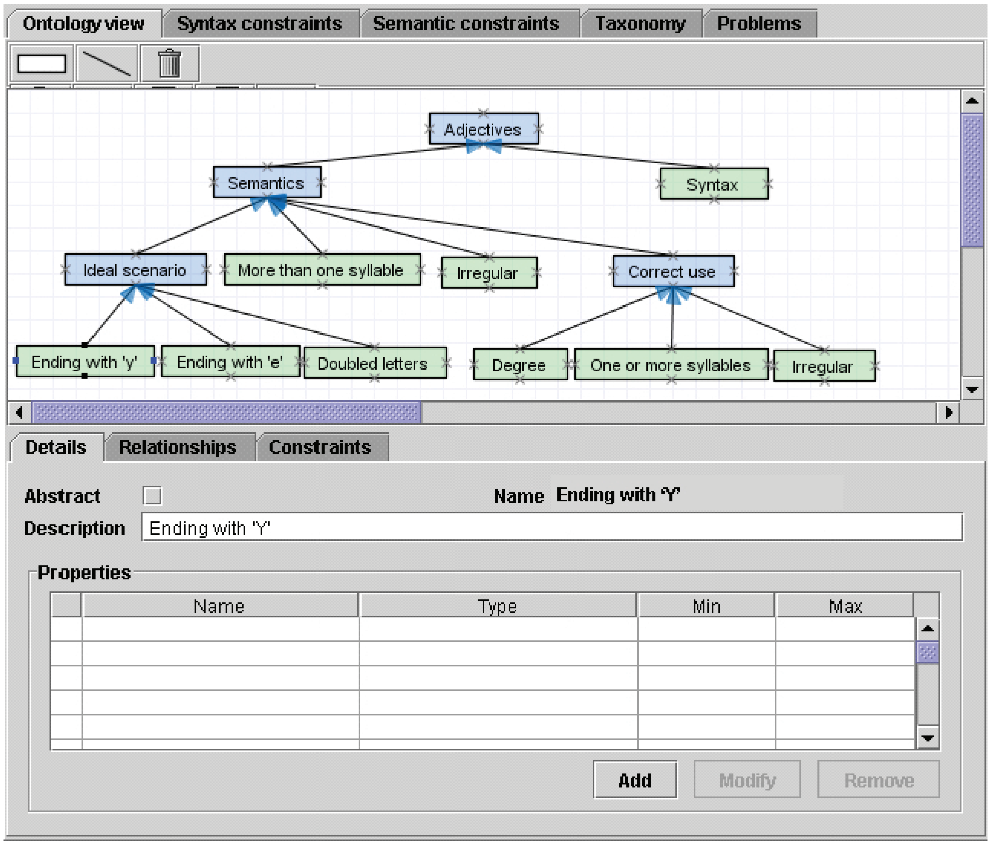
The participants were required to produce all the domain-dependent components required for the adjectives tutor. All domain-dependent components required by WETAS, including syntax and semantic constraints, problems and solutions, were to be composed using the domain model composition tool. The participants were also required to compose an ontology of the domain.

The domain model composition tool for WETAS was designed to encourage the use of a domain ontology as a means of visualising the domain and organising the knowledge base. The tool supported modelling an ontology graphically and composing constraints using a text editor that divides the available text area according to concepts of the ontology. It contained two constraint editors for syntax and semantic constraints. The editors also provided syntax highlighting facilities, by automatically colouring keywords in a different colour, to assist in constraint composition. The users were able to compose all the required domain model components using the authoring tool and deploy the produced domain model on WETAS to instantiate a tutoring system.

The domain model composition tool for WETAS was developed as a Java applet. Participants were able to access the tool through their web browser, to compose the domain model components. Completed domain models could be deployed as a tutoring system on WETAS, by using the "reload domain" feature. This results in starting an intelligent tutor instance on WETAS, provided that the domain model components are syntactically valid. Once deployed, the new ITS can be accessed by pointing a web browser to the appropriate URL.

The interface (Figure 5.1a) consists of a workspace for developing a domain ontology (ontology view) and textual editors for composing syntax constraints, semantic constraints, macros and problems. Similar to the ontology workspace of CAS (see Chapter 4), concepts are represented as rectangles, and their generalisations are indicated by arrows. The details of each concept, such as its attributes and relationships with other concepts, can be specified in the bottom section of the ontology view.

Constraints can be composed using the "Constraints" tab of the ontology view (Figure 5.1b) or using the syntax/semantic constraints text editors. The "Constraints" tab of the ontology view outlines syntactic and semantic



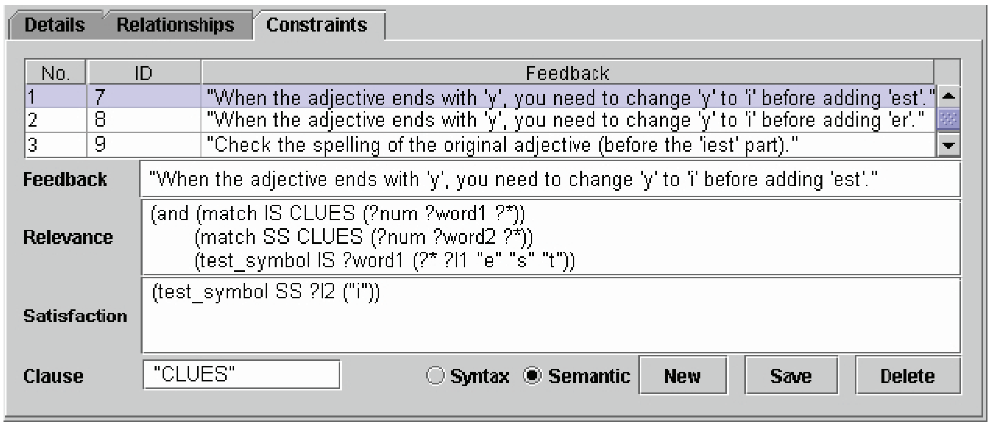
Abstract ☐ Name Ending with 'Y'

Description Ending with 'Y'

Properties

	Name	Type	Min	Max

Figure 5.1a: Interface of Domain Model Composition Tool



No.	ID	Feedback
1	7	"When the adjective ends with 'y', you need to change 'y' to 'i' before adding 'est'."
2	8	"When the adjective ends with 'y', you need to change 'y' to 'i' before adding 'er'."
3	9	"Check the spelling of the original adjective (before the 'iest' part)."

Feedback "When the adjective ends with 'y', you need to change 'y' to 'i' before adding 'est'."

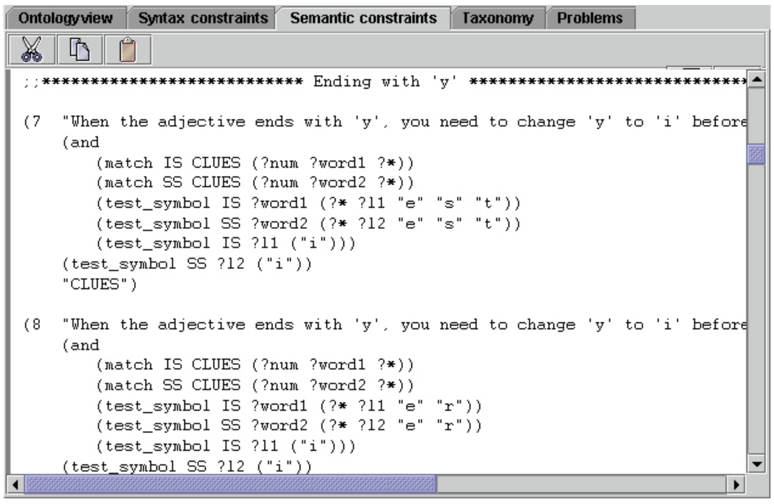
Relevance
```
(and (match IS CLUES (?num ?word1 ?*))
     (match SS CLUES (?num ?word2 ?*))
     (test_symbol IS ?word1 (?* ?l1 "e" "s" "t"))
```

Satisfaction
```
(test_symbol SS ?l2 ("i"))
```

Clause "CLUES" ○ Syntax ● Semantic

Figure 5.1b: Constraints List for *Ending with 'Y'* Concept

constraints related to the selected concept in the ontology. The constraints editor, on the other hand, lists all syntactic or semantic constraints categorised according to the corresponding concepts. Constraints in constraint editors are separated into groups by using comment lines as shown in Figure 5.2 (pre-pended by ';;' symbols). The figure shows two semantic constraints (Constraint 7 and 8) that relate to the *Ending with 'Y'* concept. Constraints composed using the constraints tab of the ontology view and the constraints editors are automatically synchronised. Constraints developed in the "Constraints" tab of the ontology view are added to the constraints editors, and constraints composed using the constraint editors are added to the "Constraints" tab of the appropriate concept.



```
;;*************************** Ending with 'y' *****************************

(7  "When the adjective ends with 'y', you need to change 'y' to 'i' before
    (and
        (match IS CLUES (?num ?word1 ?*))
        (match SS CLUES (?num ?word2 ?*))
        (test_symbol IS ?word1 (?* ?l1 "e" "s" "t"))
        (test_symbol SS ?word2 (?* ?l2 "e" "s" "t"))
        (test_symbol IS ?l1 ("i")))
    (test_symbol SS ?l2 ("i"))
    "CLUES")

(8  "When the adjective ends with 'y', you need to change 'y' to 'i' before
    (and
        (match IS CLUES (?num ?word1 ?*))
        (match SS CLUES (?num ?word2 ?*))
        (test_symbol IS ?word1 (?* ?l1 "e" "r"))
        (test_symbol SS ?word2 (?* ?l2 "e" "r"))
        (test_symbol IS ?l1 ("i")))
    (test_symbol SS ?l2 ("i"))
```

Figure 5.2: Semantic Constraints Editor of Domain Model Composition Tool

5.1.2 Results and Analysis

Seventeen (out of eighteen) participants completed the task satisfactorily. They produced working tutoring systems for the domain of adjectives. Although the participants were allocated three weeks, the majority started working on the domain model only after about a week into the study. Unfortunately, one participant who started early, lost all his work during the early stages of his project due to a glitch in the server. As he was significantly disadvantaged, his results were not included in the analysis. The glitch did not affect any others as it was rectified promptly, before affecting others.

	Ontology view	Total	%
S1	4.57	38.16	12%
S2	7.01	51.55	14%
S3	1.20	10.22	12%
S4	2.54	45.25	6%
S5	4.91	48.96	10%
S6	4.66	44.89	10%
S7	2.87	18.97	15%
S8	4.99	22.94	22%
S9	4.30	34.29	13%
S10	7.23	33.90	21%
S11	3.28	55.76	6%
S12	2.84	30.46	9%
S13	3.47	60.94	6%
S14	1.96	32.42	6%
S15	4.04	33.35	12%
S16	6.24	29.60	21%
Mean	4.13	36.98	12%
S.D.	1.72	13.66	5%

Table 5.1: Interaction Times with WETAS Front End

The domain model composition tool recorded interaction details of each participant in a log file. A summary of the total interaction times for the sixteen participants that were analysed are given in Table 5.1. The participants took 37 hours on average to complete the task, spending 4 hours in the ontology view (12% of the total time). The time in the ontology view varied widely, with a minimum of 1.2 and maximum of 7.2 hours. This can be attributed to different styles of developing the ontology. Some students may have initially developed the ontology on paper before using the system, whereas others may have developed the whole ontology online. Furthermore, some students also used the ontology view to add constraints. However, the logs showed that this was not a popular option, as most students composed constraints using the constraint editors. One factor that contributed to this behaviour may be the restrictiveness of the constraint interface in the ontology view, which only displays the details of a single constraint at a time.

The participants produced constraint bases that ranged from 13 con-

	Syntax	Semantic	Total	Syntax %	Semantic %
S1	3	27	30	10%	90%
S2	10	3	13	77%	23%
S3	1	14	15	7%	93%
S4	4	30	34	12%	88%
S5	5	11	16	31%	69%
S6	1	24	25	4%	96%
S7	15	1	16	94%	6%
S8	18	3	21	86%	14%
S9	4	11	15	27%	73%
S10	14	0	14	100%	0%
S11	1	16	17	6%	94%
S12	16	0	16	100%	0%
S13	15	1	16	94%	6%
S14	17	1	18	94%	6%
S15	14	1	15	93%	7%
S16	30	0	30	100%	0%
Mean	10.50	8.94	19.44	54%	46%
S.D.	8.23	10.47	6.60		

Table 5.2: Numbers of Constraints Composed by Students

straints to 34 constraints (summaries of each constraint base are listed in Table 5.2). On average they produced 19.4 constraints, of which 10.5 were syntax and 8.9 were semantic. The average proportions of syntax and semantic constraints were similar (54% and 46% respectively). Both the numbers of syntax constraints and semantic constraints had very high variability, with some participants opting to label the majority of constraints as syntax and others vice versa. As WETAS treats the two types of constraints with equal priority during the process of evaluating a student solution, assigning a constraint to either group does not have any noticable consequences. Furthermore, in the domain of adjectives, it is not clear as to which category the constraints belong. For example, in order to determine whether a solution is correct, it is necessary to check whether the correct rule has been applied (semantics) and whether the resulting word is spelt correctly (syntax).

We evaluated the correctness and completeness of each domain model component (ontology, syntax and semantic constraint sets) produced by the

	Ontology	%	Constraints	%	Overall	%
S1	5	100%	20	100%	25	100%
S2	4	80%	19	95%	23	92%
S3	4	80%	17	85%	21	84%
S4	5	100%	18	90%	23	92%
S5	4	80%	20	100%	24	96%
S6	5	100%	18	90%	23	92%
S7	4	80%	17	85%	21	84%
S8	3	60%	15	75%	18	72%
S9	5	100%	18	90%	23	92%
S10	3	60%	18	90%	21	84%
S11	5	100%	17	85%	22	88%
S12	3	60%	10	50%	13	52%
S13	3	60%	13	65%	16	64%
S14	3	60%	12	60%	15	60%
S15	3	60%	11	55%	14	56%
S16	5	100%	4	20%	9	36%
Mean	4.00	80%	15.44	77%	19.44	78%
S.D.	0.89	18%	4.37	22%	4.69	19%

Table 5.3: Accuracies of Constraints Produced by Students

participants by comparing them to an accurate and complete domain model produced by an expert, with about three years of experience in composing constraint bases. The ontologies produced by participants were compared against the desirable ontology and given a mark out of 5. As the complete constraint base consisted of 20 constraints, the constraint bases produced by the participants were given a mark out of 20. Consequently, the maximum total score for both the ontology and constraint base was 25.

The mark for ontology and constraints for each participant is given in Table 5.3. All participants scored high for their ontology, with an average score of 4.0. This was expected because the ontology was straightforward. Almost every participant had specified a separate concept for each group of adjectives according to the rules specified in [Clutterbuck 1990]. However, some students constructed a flat ontology, which contained only the six groupings corresponding to the rules (see Figure 5.3a). Five students scored full marks for the ontology by including the *adjective degree* concept (comparative or superlative) and concepts that dealt with syntax, such as spelling (see Figure 5.3b).

Even though the participants were only given a brief introduction to ontologies and the example ontology of LBITS, they created ontologies of a high standard. However, we cannot make any general assumptions on the difficulty of constructing ontologies since the domain of adjectives is very simple. Furthermore, the six rules for determining the comparative and superlative degree of an adjective gave strong hints on what concepts should be modelled.

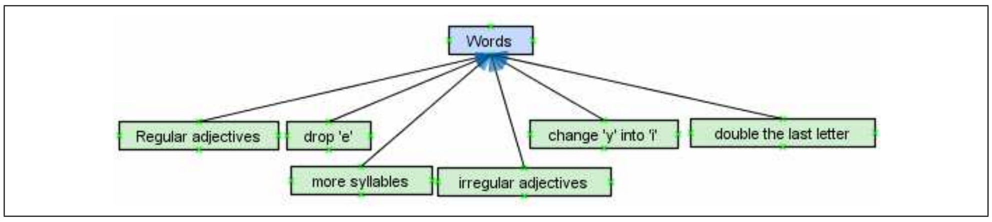


Figure 5.3a: Flat Ontology Composed by Participant S10

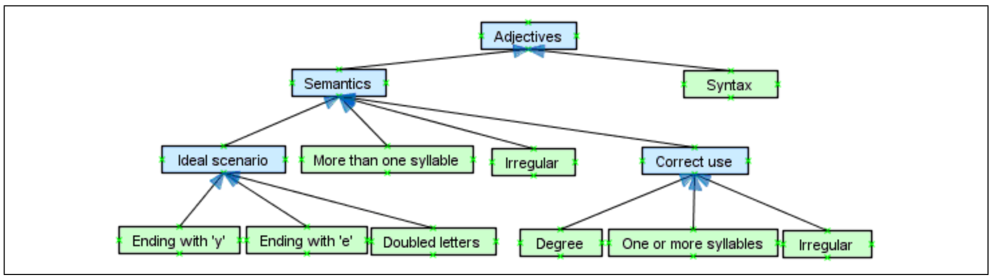


Figure 5.3b: Complete Ontologies Composed by Participant S4

The 'Constraints' column of Table 5.3 denotes the number of constraints accounted for, from the complete set of 20, by each participant. Note that the mapping between the complete constraint base and those produced by the participants is not necessarily 1:1, as a result of their differing granularity. Some participants produced a collection of more specific constraints that accounted for the same set of problem states as a single constraint produced by the expert. By contast, others produced a single constraint that accounted for multiple constraints produced by the expert. The constraints that are not at the correct granularity level would produce either more specific feedback and or more general feedback resulting in reduced learning gains. However, even constraints that are not at the appropriate granularity level can identify errors in student solutions appropriately. Therefore, we were only interested in whether the participants had accounted for all the problem states identified by the expert.

The constraint bases produced by the participants covered 15 (77%) of the constraints found in the ideal constraint base, on average. Two participants accounted for all 20 constraints. Almost all managed to account for at least 10 constraints. One participant struggled in composing constraints, managing to account for only four constraints. In general, the quality of constraints was high.

All but two participants (S13 and S16) categorised their constraints according to the concepts of the ontology. For these participants, there was a significant correlation between the ontology score and the constraints score (0.679, $p<0.01$). However, there was no significant correlation between the ontology score and the constraints score when all participants were considered. This strongly suggests that the participants who made use of the ontology to categorise constraints while composing them, developed more complete constraint bases.

An obvious reason for this finding may be that more able students produced better ontologies and also produced a complete set of constraints. To test this hypothesis, we determined the correlation between the participant's final grade for the course (which included two other assignments and a final examination) and the ontology/constraint scores. There was indeed a strong correlation (0.840, $p<0.01$) between the grade and the constraint score. However, there was no significant correlation between the grade and the ontology score. This lack of a correlation can be due to a number of factors. Since the task of building ontologies was novel for the participants, they may have found it interesting and performed well regardless of their ability. Another factor is that the participants had more practise at writing constraints (in another assignment for the same course) than on modelling ontologies. Finally, the simplicity of the domain could also be a contributing factor.

The participants also felt that building an ontology assisted in the task of identifying constraints. Their comments on the usefulness of ontologies are listed in Figure 5.4. The comments suggested that the participants effectively used concepts of the ontology to decompose the constraint base into meaningful categories. Furthermore, the ontology has provided an initial framework for composing constraints.

- "Ontology helped me organise my thinking" - S2
- "The ontology made me easily define the basic structure of this tutor" - S10
- "The constraints were constructed based on the ontology design" - S13
- "Ontology was designed first so that it provides a guideline for the tasks ahead" - S9

Figure 5.4: Comments by Participants on the Usefulness of Ontologies

The participants spent 2 hours per constraint (calculated as the total interaction time/total number of constraints, SD=1 hour). This is twice the time reported in [Mitrovic & Ohlsson 1999], but the participants are neither knowledge engineers nor domain experts, so the difference is understandable. The language used to compose constraints by Mitrovic is slightly different from the language used by participants of the study, which may also be a factor for the difference. It is interesting that this time of two hours is still much shorter than 10 hours reportedly necessary for acquiring a single production rule for model-tracing tutors [Anderson et al. 1996], even when constraints are generated by non-experts.

5.2 Effectiveness of the Constraint Generation Algorithms (Study 2)

We evaluated the effectiveness of CAS's constraint generation algorithms for three vastly different domains: Database modelling, Data normalisation and Fraction addition. The domains of Database modelling and Data normalisation were specifically chosen because we had previously developed two successful constraint based tutors for the two domains; KERMIT [Suraweera & Mitrovic 2004] and NORMIT [Mitrovic 2005*a*]. The constraint bases of these tutors were used as benchmarks to evaluate the correctness and completeness of the constraint bases generated by CAS.

The choice of domains were also influenced by the desire to evaluate CAS for procedural as well as non-procedural tasks. The domains of Data normalisation and Fraction addition can be categorised as procedural, where a strict set of steps have to be followed to arrive at the solution. On the other hand, the task of modelling a database is not a well-formed process and there is no strict procedure to be followed to produce an ER model.

CAS relies on the domain expert to provide a correct and complete ontology of the domain, problems and solutions. It requires a collection of solutions for each problem outlining different ways of solving the same problem. In order to ensure that CAS is supplied with all the correct and complete information, the domain-dependent information for all three domains were supplied by the developer of CAS. Constraints for each domain were generated by following the six steps in the constraint generation process outlined in Section 4.1.

5.2.1 Entity Relationship Modelling

Entity Relationship (ER) modelling is a popular database modelling technique, used frequently for the conceptual design of databases. ER modelling, originally proposed by P. Chen [1976], views the world as consisting of entities and relationships between them. An entity is the basic item represented in the ER model, which is an object in the real world that exists independently. The properties of each entity that describe it are called attributes. For example, a student entity may be described by his/her name, date of birth, address etc. A relationship is an association between two or more entities.

Typical problems in the domain of database modelling involve composing an ER diagram that satisfies a given set of requirements. Problems available in KERMIT also followed this pattern, where students are given the requirements in the form of a textual description. They are required to compose an ER diagram (using the ER modelling workspace) that satisfies the given requirements.

In order to compose an ER diagram, students must understand the data model used, the basic building blocks available and the integrity constraints specified on them. In real situations the description of the scenario is long and often ambiguous and incomplete. To identify integrities, the student must be able to reason about the requirements and use his/her own world knowledge to make valid assumptions. ER modelling is not a well-defined process, and the task is open-ended. There is no algorithm to use to derive the ER schema for a given set of requirements. There is no single, best solution for a problem, and often there are several correct solutions for the same requirements.

Composing an Ontology of the Domain

The ontology composed for the domain of ER modelling, as shown in Figure 5.5, consists of *Construct* as the top level concept. The three types of constructs in ER modelling, *Relationship*, *Entity* and *Attribute* are sub-concepts of *Construct*. *Relationship* is specialised to *Regular* and *Identifying* with each of them being specialised to the three types of relationships; *Binary*, *N-ary* and *Recursive*. The *Entity* concept is specialised to *Regular* and *Weak*, which are the types of entities. *Attributes* can be either *Simple* or *Composite*. *Simple* attributes are specialised further to according to their types of *Key*, *Partial-key*, *Single*, *Derived* and *Multi-valued*.

Modelling the Structure of Solutions

The structure modelled for solutions for the domain of ER modelling is outlined in Figure 5.6. It consists of “Entities”, “Relationships” and “Attributes” components, which are assigned to the concepts of *Entity*, *Relationship* and *Attribute* respectively. As solutions can only contain instances of concrete (non-abstract) concepts, the “Entities” component can only hold instances of *Regular entity* and *Weak entity*, which are the concrete sub-concepts of *Entity*. Similarly, the “Relationships” list can contain instances

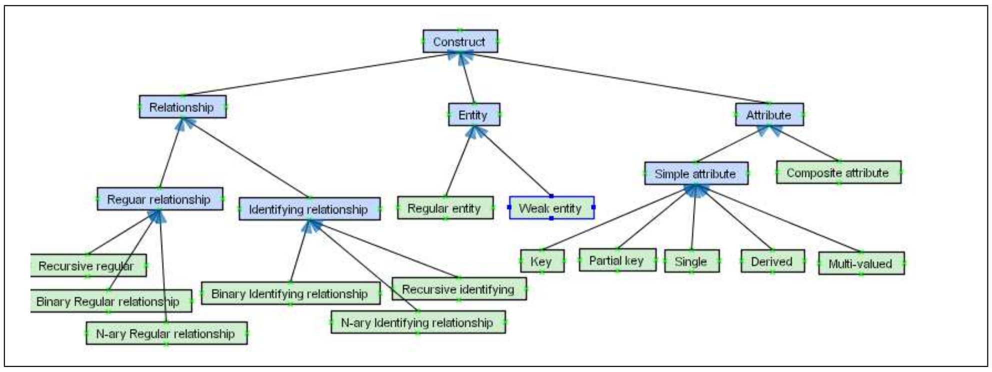


Figure 5.5: Ontology for Domain of ER Modelling

of the six concrete types of relationships such as *Binary regular*, *N-ary regular* etc. Finally, the "Attributes" component contains instances of the concrete types of *Attribute* concept.

The ER ontology defines a relationship between the *Relationship* concept and the *Entity* concept. Creating an instance of a *Relationship* requires the specification of the instances of *Regular* or *Weak* entities that participate in that relationship. Similarly, when an instance of an *Attribute* is created, the *Entity* or *Relationship* which it belongs to has to be selected.

The solution structure modelled for ER modelling is similar to KERMIT's internal solution structure. The ER workspace applet of KERMIT converts the graphical representation of ER models into a textual format that consists of four lists: Entities, Relationships, Attributes and Connectors. The Entities, Relationships and Attributes lists are similar to three components of the ER solution structure modelled in CAS. Connectors are used in KERMIT to keep track of entities that participate in relationships and attributes that belong to entities. However, as these associations are modelled in the ontology as relationships, the connectors become redundant when adding solutions in CAS. As discussed in Section 4.6, the solution composition interface of CAS allows direct specification of elements that participate in association relationships.

Solution component	Concept
Entities	Entity
Relationships	Relationship
Attributes	Attribute

Figure 5.6: Solution Structure for ER Diagrams

Adding Problems and their Solutions

The problem and solutions editor was used to add seven ER modelling problems and their solutions. They were chosen from problems found in KERMIT. Most problems in KERMIT require solutions that contain a lot of repetition. The problems for which their solutions did not contain many repetitive elements (less than two elements of the same type) were directly copied from KERMIT. The complexity of other problems were reduced by removing the repeating elements in order to reduce the author's workload.

Each problem was accompanied by at least two alternative correct solutions using the solutions composer. Although the solutions were chosen to maximise the difference between them, solutions in ER modelling tend to only have subtle differences. As the solution composer enabled the composition of secondary solutions by modifying the first solution, the task of adding alternative solutions required very little effort.

Generating Syntax Constraints

The syntax constraints generator produced a total of 49 syntax constraints from the ER modelling ontology shown in Figure 5.5 [Suraweera et al. 2005]. They varied from simple constraints such as '*Entities* should have a name' to more complex constraints such as '*Regular Entity* must have at least one *Key Attribute*'.

For example, consider the *Binary Identifying Relationship* concept. It is described by three properties; 'name', 'identified cardinality' and 'identified participation'. It also participates in a relationship with *Weak Entity* named 'identified entity' and *Regular Entity* named 'owner'. The restrictions of each of the properties and relationships specified using the ontology workspace, generated a total of six constraints, which are outlined in Figure 5.7.

- Relationship type must have exactly one name
- The name property of *Relationship* has to be unique
- The identified cardinality of *Binary identifying relationship* must be “1”
- The identified participation of *Binary identifying relationship* must be “total”
- Exactly one *Weak entity* must participate in each *Binary identifying relationship* as the identified entity
- At least one *Regular entity* must participate in each *Binary identifying relationship* as owners

Figure 5.7: Syntax Constraints Produced from Properties and Relationships of *Binary Identifying Relationship* Concept

Generating Semantic Constraints

CAS produced a total of 135 semantic constraints by analysing the seven problems and their solutions provided during step four of the authoring process [Suraweera et al. 2005]. These constraints were able to identify errors in an ER model composed by a student by comparing it to the system's corresponding ideal solution. They ensure that the student's ER diagram consists of all the required constructs and that they correctly participate in relationships.

A sample list of semantic constraints generated by CAS is given in Figure 5.8. Constraint 1 ensures the student's solution contains a corresponding regular entity for each one found in the ideal solution. Constraint 2 is an example of a constraint that enables the tutoring system to identify solutions equivalent to the system's ideal solution. It allows for the alternative method for modelling a weak entity. It verifies that the student solution contains either a weak entity or a multi-valued attribute for each weak entity found in the ideal solution.

Constraints 3, 4 and 5 ensure that the student has correctly connected the

1. **Relevance:** IS.Entities component has a Regular entity (?id, ?*)
   Satisfaction: SS.Entities component has a Regular entity (?id, ?*)

2. **Relevance:** IS.Entities component has a Weak entity (?id, ?*)
   Satisfaction: SS.Entities component has a Weak entity (?id, ?*)
   OR SS.Attributes component has a multi-valued attribute (?id, ?*)

3. **Relevance:** IS.Entities has a Regular entity (?id1, ?*)
   AND IS.Attributes has a Key (?id2, ?*)
   AND SS.Entities has a Regular entity (?id1, ?*)
   AND IS Regular entity (?id1, ?*) has Key (?id2, ?*) as key-attribute
   Satisfaction: SS.Attributes has a Key (?id2, ?*)

4. **Relevance:** IS.Entities has a Regular entity (?id1, ?*)
   AND IS.Attributes has a Key (?id2, ?*)
   AND SS.Attributes has a Key (?id2, ?*)
   AND IS Regular entity (?id1, ?*) has Key (?id2, ?*) as key-attribute
   Satisfaction: SS.Entities has a Regular entity (?id1, ?*)

5. **Relevance:** IS.Entities has a Regular entity (?id1, ?*)
   AND IS.Attributes has a Key (?id2, ?*)
   AND SS.Entities has a Regular entity (?id1, ?*)
   AND SS.Attributes has a Key (?id2, ?*)
   AND IS Regular entity (?id1, ?*) has Key (?id2, ?*) as key-attribute
   Satisfaction: SS Regular entity (?id1, ?*) has Key (?id2, ?*) as key-attribute

Figure 5.8: Sample Semantic Constraints Produced by CAS

all the key attributes with the respective regular entities. In order to achieve this, constraints 3 and 4 check whether the student's solution contains the necessary constructs (the matching regular entity and key attribute). Constraint 5 ensures that the correct relationship (ie. *key-attribute* relationship) holds between the two constructs.

Analysis and Discussion

The constraint base of KERMIT, which was developed entirely manually, was used to evaluate the correctness and completeness of the generated set of constraints. The effectiveness of KERMIT has been evaluated in a series of empirical evaluations [Suraweera & Mitrovic 2004, Mitrovic, Suraweera, Martin & Weerasinghe 2004, Weerasinghe & Mitrovic 2003]. All evaluations revealed that its constraints were effective for learning ER modelling concepts. The constraint base has also been enhanced over a period of four years.

KERMIT contains 35 syntax and 138 semantic constraints. The graphical representation of ER models composed by students is converted into a textual representation to be evaluated by the constraints. The syntax constraints ensure the syntactic validity of an ER diagram by searching for particular patterns within the textual representation. The semantic constraints on the other hand compare significant features of KERMIT's ideal solution against corresponding features of the solution composed by the student. The semantic constraints of KERMIT included 13 constraints that are relevant for enhanced ER modelling. As the ER modelling ontology developed in CAS did not contain concepts related to enhanced ER modelling, only the 125 semantic constraints in KERMIT relevant to ER modelling were used for comparison. All syntax constraints of KERMIT were used for comparison as it did not contain any constraints specific to enhanced ER modelling.

The correctness and completeness of constraints generated by CAS were evaluated manually comparing them against the constraints found in KERMIT. As the generated constraints and the constraints found in CAS were composed in two different languages, manual comparison was essential. The constraints in KERMIT are composed in the WETAS language using pattern matching functions with conjunctions and disjunctions. The generated constraints, on the other hand, were composed in a higher level language as shown in Figure 5.7 and Figure 5.8.

The process of evaluating the generated constraints involved searching for an equivalent constraint for each constraint found in KERMIT. The generated set of syntax constraints contained equivalent constraints for all syntax constraints found in KERMIT. In other words, the generated syntax constraints were 100% complete compared to the constraints of KERMIT.

Further analysis of the generated syntax constraints revealed that some generated constraints were more specific than the constraints found in KERMIT. In other words, the two sets were inconsistent with respect to the constraint granularity. In some cases, several constraints generated by CAS were required to identify the set of problem states identified by a single constraint in KERMIT. This accounted for the difference in the numbers of constraints in the generated set (49) and KERMIT's syntax constraints (35).

The generated semantic constraints managed to account for 90% of the 125 constraints found in KERMIT's constraint-base. Simillar to syntax constraints, the counts in the two semantic constraint sets had disparities because the generated constraints were more specific. For example, KERMIT contains a constraint that verfies that the student solution contains a matching a key attribute attached to an entity. The domain model generated by CAS contains two constraints to account for the same problem state. One checks for the existence of the required key attribute and the other checks whether the key attribute and the entity are connected.

The semantic constraints generator failed to produce 12 of the constraints found in KERMIT's constraint base. Some of these constraints ensured that constructs were not misrepresented using different types of constructs. For example, they ensured that entities in the ideal solution were not represented as attributes or relationships in the student solution. This is a common misconception among students. Currently, CAS is not capable of generating such constraints. However, this problem state is covered by other constraints that ensure that there can be no extra constructs of each type. These constraints would provide a more general feedback message that would treat the misrepresented construct as an extra construct.

CAS also failed to generate the set of constraints that ensured that the students can produce equivalent solutions according to the equivalence shown in Figure 5.9. According to the equivalence, an attribute that belongs to a binary relationship can be assigned to an entity that participates in the relationship, as long as it has a cardinality of 1. CAS is not be able to generate these constraints as they deal with two equivalent relationship instances (ontological relationships). Currently, CAS is only able to identify equivalent elements in two solutions.

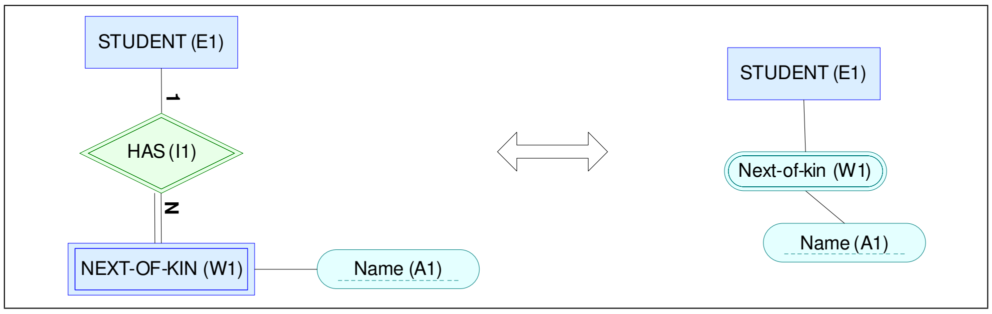


Figure 5.9: Equivalent ER Models: An Attribute Belonging to a Relationship can also be Assigned to a Participating *Regular Entity* with Cardinality 1

CAS also produced some constraints that suggested modifications to existing constraints in KERMIT, improving KERMIT's ability to handle alternative solutions. For example, although the constraints in KERMIT allowed a weak entity to be modelled as a composite multi-valued attribute, the student was required to have all the attributes of the weak entity with their types identical to their corresponding attributes in the ideal solution. However CAS correctly identified that when a weak entity is represented as a composite multi-valued attribute, the partial key of the weak entity has to be modelled as a simple component of the composite attribute (illustrated in Figure 4.29). Furthermore, the identifying relationship essential for the weak entity becomes obsolete. These two examples illustrate how CAS improved upon the original domain model of KERMIT.

5.2.2 Fraction Addition

Adding two fractions is an important part of the mathematics curriculum. In order to calculate the sum of two fractions correctly, a series of steps have to be followed sequentially. The process consists of four steps as outlined in Figure 5.10. Initially, the lowest common denominator (LCD) of the two fractions has to be found. Then the two fractions have to be converted to have LCD as their denominator. The sum of the numerators of the converted fractions become the numerator of the resulting fraction, with LCD as its denominator. Finally, the resulting fraction has to be simplified, if possible, to produce the final result.

1. Find the lowest common denominator (LCD)
2. Convert fractions to LCD as denominator
3. Add the resulting fractions
4. Simplify the final result

Figure 5.10: Problem Solving Procedure for Fraction Addition

Composing an Ontology of the Domain

The domain of fractions is simple. The ontology for the domain, as shown in Figure 5.11, contains *Number* as the most generic concept and *Whole number* and *Fraction* as its sub concepts. The *Whole number* concept is specialised to *LCD*, whereas *Fraction* is specialised to *Improper fraction* and *Reduced fraction.*

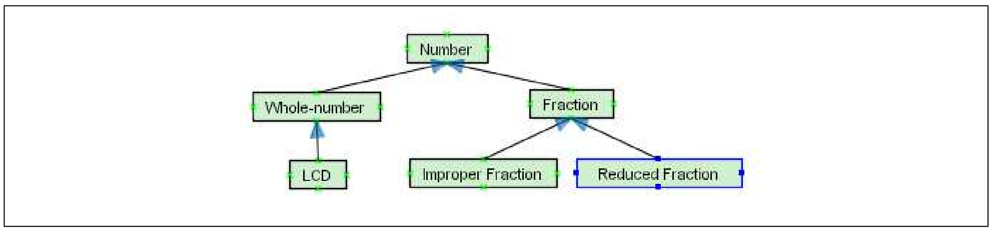


Figure 5.11: Ontology for the Domain of Adding Two Fractions

The *Fraction* concept participates in two relationships; ‘numerator’ with

Whole number and ‘denominator’ with *LCD*. The *Reduced fraction* concept participates in an extra relationship with *Whole number* for the quotient.

Modelling the Structure of Solutions

Problems in the domain of adding fractions have to be solved in a series of four steps. As each step has its own solution(s), their structure has to be modelled seperately. The complete structure of solutions for adding fractions is outlined in Figure 5.12. The solution for the first step of finding the LCD requires an instance of a *LCD* concept. The solution for the second step, in contrast, consists of two components with both requiring an instance of *Improper fraction.* Steps three and four both consist of a single component each and require instances of *Improper fraction* and *Reduced fraction* respectively.

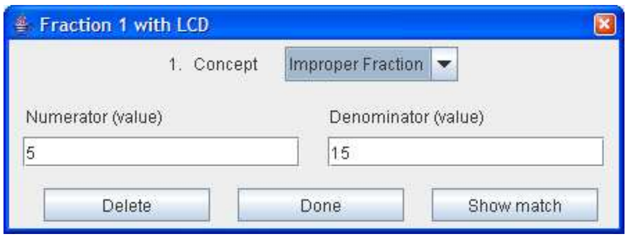
Problem-solving step	Solution component	Concept
1. Find LCD	LCD	LCD
2. Convert fractions to LCD	Fraction 1	Improper fraction
	Fraction 2	Improper fraction
3. Sum of fractions	Improper sum	Improper fraction
4. Final reduced sum	Final sum	Reduced fraction

Figure 5.12: Solution Structure for Adding Two Fractions

Providing Problems and their Solutions

The problem and solution editor was used to add three problems and their solutions. Problems were represented in the textual form of “1/2 + 3/4”. Solutions were composed using the solution editor, creating instances of *LCD*, *Fraction* and other concepts. As problems of adding two fractions have only a single correct solution, no alternate solutions were provided.

The solution editor reduces the effort required to compose solutions by reasoning about the ontology during the process of rendering an input form for instantiating concepts. This was exemplified during the process of creating solutions for fraction addition problems. Let us consider the *Improper fraction* concept. It does not contain any properties, but participates in two relationships for numerator and denominator. Both *Whole number* and *LCD* concepts, which are involved in these relationships possess a single property each. As a consequence, the solution editor creates an interface with two input boxes for the properties *Whole number* and *LCD*, as shown in Figure 5.13, easing the workload on the domain expert. If the solution editor did not perform any reasoning, the domain expert would have to create three instances; *Whole number*, *LCD* and *Improper fraction*.


Fraction 1 with LCD

1. Concept	Improper Fraction
Numerator (value)	Denominator (value)
5	15

Figure 5.13: Input Form for Creating an Instance of *Fraction*

Generating Syntax Constraints

CAS produced ten constraints for the domain of fraction addition from the ontology in Figure 5.11 and the solution structure in Figure 5.12. The majority of constraints were generated to ensure that the student followed the correct problem-solving procedure. For example, constraint 1, shown in Figure 5.14, verifies that the student has populated the LCD input box during the first step of finding the LCD. Due to the simplistic nature of the domain, there were very few other syntax constraints generated. Constraint 2 is an example of a syntax constraint, which ensures the LCD is always a positive number.

The constraint generator also produced constraints that were trivially satisfied due to the restrictive nature of the interface shown in Figure 5.13. For example, constraint 3 ensures that all instances of *Improper fractions* participate with a single instance of *LCD* as its denominator. Although this constraint is trivially satisfied as the interface does not allow specifying more than a single value as a fraction's denominator, the constraint is still valid. We believe these constraints are necessary as the domain expert may design a less restrictive interface. The constraint would be useful for an interface that allows free form text.

1. **Relevance:** Current_step = ‘Find LCD’
   Satisfaction: SS ‘Lowest Common Denominator’ component should not be empty

2. **Relevance:** SS ‘Lowest Common Denominator’ component is not be empty
   Satisfaction: SS ‘Lowest Common Denominator’ > 0

3. **Relevance:** SS has an Improper Fraction
   Satisfaction: SS Improper Fraction has to participate with exactly 1 LCD as its denominator

Figure 5.14: Example Syntax Constraints Generated for the Domain of Adding Two Fractions

Generating Semantic Constraints

A total of ten semantic constraints were generated for fraction addition, from only two example problems. Each problem in this domain has only a single valid solution. The semantic constraints check that the different components of the student’s solution match the ideal solution.

As the domain is procedural, constraints are only relevant for a particular set of problem-solving steps. For example constraint 1, outlined in Figure 5.15, is only relevant if the student is attempting the first problem-solving step (find LCD), the LCD component of their solution is not empty (i.e., the student has specified the LCD) and the ideal solution contains an LCD (i.e. it is necessary to find the LCD for the current problem). If the constraint is relevant, then the student’s answer needs to be the same as the one specified in the ideal solution. Constraint 2 is a similar constraint, which ensures that the student has supplied the correct fraction as the solution during the second step of converting fractions to LCD.

1. **Relevance:** Current_step = ‘Find LCD’
   AND SS ‘Lowest Common Denominator’ component is not empty
   AND IS.Lowest Common Denominator has a LCD(?var1)
   Satisfaction: SS.Lowest Common Denominator has a LCD(?var1)

2. **Relevance:** Current_step = ‘Convert to LCD’
   AND SS ‘Fraction 1 with LCD’ component is not empty
   AND IS.Fraction 1 with LCD has a Improper Fraction(?var1, ?var2)
   Satisfaction: SS.Fraction 1 with LCD has a Improper Fraction(?var1, ?var2)

Figure 5.15: Example Semantic Constraints Generated for the Domain of Adding Two Fractions

Analysis and Discussion

Unlike for the domains of ER modelling and Normalisation, we did not have a benchmark to evaluate the completeness of constraints generated for the domain of adding fractions. However, we manually compiled a list of significant problem states for the task to evaluate the completeness of the generated constraint set. The list contained three syntactically significant problem states that collectively ensured that the correct problem solving procedure is followed. It also contained 13 semantically significant problem states that ensured each significant component of the solution such as numerator and denominator of each fraction is correct.

The three syntax constraints generated by CAS accounted for all the syntactically significant problem states. The constraint generator also produced a few extra constraints such as “a fraction should contain a numerator and a denominator”, that were trivially satisfied due to the restrictive nature of CAS’s form-based solution composition interface. The interface contained two text boxes for each fraction ensuring that each fraction can only contain two integers; numerator and denominator. The extra constraints, however, are useful for a less restrictive interface.

The ten semantic constraints generated by CAS accounted for all but one (92%) of the semantically significant problem states. CAS was not able to generate a constraint that checked whether the student had entered a higher common multiple instead of the least common multiple. This was expected as CAS does not possess the ability to generate constraints that require algebraic functionality.

CAS required only two problems and their ideal solutions for composing sufficiently generalised semantic constraints for the domain of adding fractions. This was due to the fact that problems in the domain model have a single correct solution. Furthermore, unlike ER modelling, solutions for problems required all of their components to be populated. Each input box of the solution input form required an integer as input.

5.2.3 Normalisation

Database normalisation is the process of refining a relational database schema in order to ensure that all tables are of high quality [Elmasri & Navathe 2003]. The process proceeds in a top-down fashion by evaluating each table against a criteria for each normal form and decomposing tables as necessary. There are four widely used normal forms: first normal form (1NF), second normal form (2NF), third normal form (3NF) and Boyce-Codd normal form (BCNF).

Typical problems in the domain of data normalisation focus on finding the normal form of a table given its functional dependencies. The table then has to be decomposed to satisfy BCNF normal form (the highest level of normalisation). A total of 11 steps, as outlined in Figure 5.16, have to be followed sequentially in order to normalise a table into BCNF normal form.

The process of normalising a table is initiated by determining the candidate keys of the table. In order to verify that one or more attributes is the candidate key of a table, their closure has to be determined. Once the candidate keys are found, the prime attributes of the table have to be pointed out. Then the functional dependencies (FD) have to be replaced by an equivalent set of FDs with simplified right-hand sides. In order to determine the normal form of the table, any partial FDs and other FDs that violate third normal form and BCNF have to be determined. Finally, the table has to be

1. Determine candidate keys of the table
2. Determine closure of an attribute set
3. Specify prime attributes
4. Simplify to include a single attribute on the right-hand sides of functional dependencies
5. Specify normal form of the table
6. Identify partial functional dependencies
7. Specify FDs that violate third normal form (3NF)
8. Specify FDs that violate Boyce Codd normal form (BCNF)
9. Reduce left hand sides of F.D.s
10. Find minimal cover
11. Decompose table into BCNF normal form

Figure 5.16: Problem Solving Procedure for Normalising Database Tables

decomposed so as to satisfy BCNF after reducing the left hand sides of the FDs, and finding the minimal cover.

Composing an Ontology of the Domain

The ontology for the domain of Normalisation (see Figure 5.17) contains four top level concepts; *Attribute type*, *Relation*, *Normal form*, *Functional dependency. Attribute type* is specialised to *Attribute* and *Candidate key. Attribute* is further specialised as *Prime attribute.* The *Functional dependency* concept is specialised according to the five types of functional dependencies; *Partial FD*, *3NF-violate FD*, *BCNF-violate FD*, *Right reduced FD*, *Left reduced FD.*

A functional dependency is a dependency between two sets of attributes. Consequently, the *Functional dependency* concept participates in two rela-

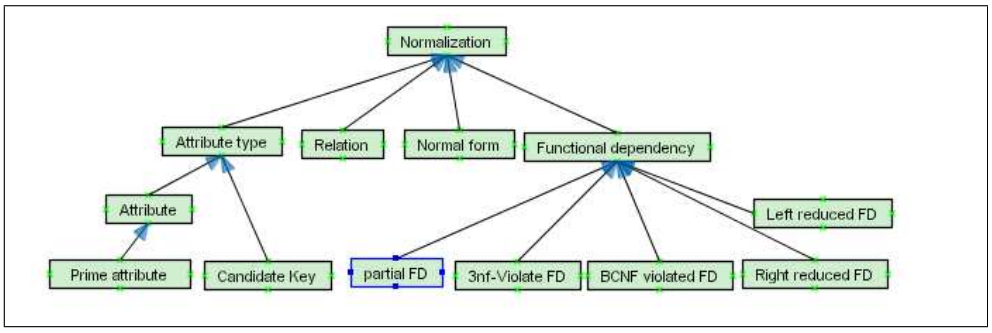


Figure 5.17: Ontology for the Domain of Normalisation

tionships with *Attribute*. Each kind of functional dependencies has different restrictions on the cardinality of the relationships. For example, a *Right-reduced FD* can only have a single *Attribute* in its right-hand side.

Modelling the Structure of Solutions

Similar to the domain of adding fractions, the structure of solutions for each step has to be modelled. Solutions for problems in Normalisation require a collection of elements of a particular type for each step. The structure of solutions, decomposed step-wise is given in Figure 5.18.

The student is required to provide a set of instances of attributes as the solution for the first step of determining the closure. The following step requires the student to outline candidate keys of the given table. Then instances of *Prime attributes* have to specified. Steps four to seven require instances of the different types of functional dependencies. Step eight requires an instance of the *Normal form* concept. Steps nine and ten both require instances of *Left reduced FD* as their solutions. Finally, an instance of *Relation* concept is required for decomposing the table to BCNF.

Providing Problems and their Solutions

Three problems and their solutions were added using the problem editor of CAS. They were chosen from NORMIT to ensure that CAS was exposed to at least one example of concepts that are expected to be taught. The solutions were also obtained by NORMIT, using the available problem solver. As

Problem-solving step	Solution component	Concept
1. Find candidate keys	Candidate keys	Candidate Key
2. Determine closure	Attributes	Attribute
3. Specify prime attributes	Prime attributes	Prime attribute
4. Simplify right hand of FDs	Functional dependencies	Right reduced FD
5. Specify Normal form of table	Normal form	Normal form
6. Outline partial FDs	Partial FDs	Partial FD
7. Specify FDs that violate 3NF	NF Violated FDs	3NF-Violate FD
8. Specify FDs that violate BCNF	BCNF violated FDs	BCNF-Violate FD
9. Reduce left hand side of FDs	Left reduced FDs	Left reduced FD
10. Find minimal cover	Min cover	Left reduced FD
11. Decompose table to BCNF	Decomposition	Relation

Figure 5.18: Solution Structure for Normalising Database Tables

the chosen problems only had a single correct solution, each problem was accompanied with its correct solution.

Although some problems may have more that one correct path for solving a problem correctly, the problem solver of NORMIT would produce a single correct solution relevant to the current step of the student. The solution produced by the problem solver would depend on the problem solving path taken by the student. As the solution to each step composed by a student is always compared to a single correct solution, the constraints generated by analysing a single solution would still be applicable for problems with multiple solutions.

Generating Syntax Constraints

The syntax constraints generator produced a total of 26 constraints from the data normalisation ontology. Similar to the the domain of adding fractions, the constraint generator produced a set of syntax constraints to ensure that the student followed the correct problem-solving path. Constraint 1 of Figure 5.19 is an example of a constraint that forces the student to be on the correct problem-solving path.

The generated set also contained constraints that ensured the functional dependencies produced by students adhered to the correct syntax. For example, constraint 2 ensures that every right-reduced functional dependency produced by students contains only a single attribute in its right-hand side.

1. **Relevance:** Current_step = 'candkeys'
   Satisfaction: SS 'Candidate key' component should not be empty

2. **Relevance:** SS has a Right reduced FD, FD1
   Satisfaction: FD1 can only have one Attribute in its right hand side

Figure 5.19: Example Syntax Constraints Generated for the Domain of Data Normalisation

Generating Semantic Constraints

The semantic constraints generator produced a total of 45 constraints by analysing three problems and their solutions chosen from NORMIT's list of problems. As the domain is procedural, the generated constraints were only relevant for a particular set of steps. For example, Constraint 1 shown in Figure 5.20 is only relevant when the student is working on the second step of identifying candidate keys of the table. It ensures that each candidate key specified by the student has a corresponding equal candidate key in the ideal solution. Similarly, constraint 2 ensures that the normal form specified by the student is equal to the normal form denoted in the system's ideal solution.

Analysis and Discussion

The completeness and correctness of constraints generated for the domain of data normalisation was evaluated by manually comparing the generated constraints against the constraints found in NORMIT. The constraint base of NORMIT was developed manually by Antonija Mitrovic and has been evaluated with students in a series of empirical evaluations [Mitrovic 2005*a*, Mitrovic 2005*b*, Mitrovic 2003*b*].

The generated constraints covered all but two of the 21 syntax constraints that existed in NORMIT. The generated constraints accounted for 90% of

1. **Relevance:** Current_step = 'candkeys'
   AND SS 'Candidate key' component is not empty
   AND IS.Candidate keys has a Candidate Key (?var1)
   Satisfaction: SS.Candidate key has a Candidate Key (?var1)

2. **Relevance:** Current_step = 'nf'
   AND SS 'Normal form' component is not empty
   AND IS.Normal form has a Normal form(?var1, ?var2, ?var3, ?var4)
   Satisfaction: SS.Normal form has a Normal form(?var1, ?var2, ?var3, ?var4)

Figure 5.20: Example Semantic Constraints Generated for the Domain of Data Normalisation

the syntax constraints found in NORMIT. Similar to the experience with the constraints generated for the domain of ER modelling, the constraints generated for normalisation were more specific than the constraints that were found in NORMIT.

Both constraints that were not generated by CAS contained tests on more than one component of the solution. For example, the constraint that ensures the correctness of the specified partial FDs contains a check to ensure that the student has specified the table not to be in 2NF. It also contains a check that verifies the specified partial FDs have a candidate key in their left-hand and non-prime attributes in their right-hand. This constraint is a result of combining problem solving knowledge within constraints. A problem solver for normalisation, which is independent of constraints, would produce the correct solution for each step based on the problem solving path taken by the student. Consequently, this constraint can be replaced by a simple constraint that compares each student specified partial FDs against the ideal solution's partial FDs.

The 47 generated semantic constraints accounted for all the 45 semantic constraints found in NORMIT. The result of 100% accuracy was excellent considering that only three sample problems and their solutions were provided.

Although CAS was provided with problems with single correct solutions, the generated constraints are capable of identifying correct solutions arrived at using alternative problem solving paths. This is made possible by NORMIT's problem solver, which produce a single correct solution to the student's current step based on the problem solving path taken. As the solution to each step composed by a student is always compared to a single correct solution, the generated constraints, with the solution generator, are capable of identifying all available correct solutions to a problem.

5.2.4 Discussion

The author has to provide example problems and solutions according to the solution structure. Therefore, the final structure of solutions would have to be very accurate or the domain expert would struggle to add the complete solution. As the solution structure portrays the most accurate structure of the task, the syntax constraint generation algorithm generates constraints only on the concepts that are relevant for the solution structure. This ensures that the generated constraints are useful and are unlikely to be trivially satisfied. However, the syntax constraints generation algorithm is limited by its dependence on the domain ontology and the solution structure modelled by the expert. It has no ability to handle incomplete ontologies or incomplete solution structures.

The semantic constraints generator is also highly dependent on the domain ontology. It requires all relationships between instances to be explicitly stated in the domain ontology. If a relationship is not modelled, then the relevant constraints would not be generated.

The effort required to produce semantic constraints involves providing the system with sample problems and their solutions. WETAS also requires problems and their solutions to be supplied even if constraints are composed manually. CAS can be enhanced to automatically translate the problems and solutions added using the problem editor to the format required by WETAS. The only extra effort required would be in composing alternative solutions. However, as alternative solutions are composed by modifying a copy of the primary solution, the extra effort is minimal.

Although CAS produced constraint sets that were over 90% complete, it failed to produce a few constraints for ER modelling (semantic), fraction addition (semantic) and data normalisation (syntax). In the case of ER modelling, the constraint generator failed to produce constraints that dealt with common misconceptions such as misrepresented constructs. It also failed to generate constraints that are based on matching instances of ontological relationships. For the domain of adding fractions, the semantic constraints generator failed to produce a constraint, which required algebraic functionality. The syntax constraints generator failed to produce a few constraints for normalisation that had problem-solving logic embedded in them. However, since constraints operate individually and do not depend on each other, the generated constraints would be sufficient for the domain model of a beta version of a tutoring system. Testing the tutoring system may reveal the missing constraints.

Some constraints generated by CAS for the evaluated domains were too specific. The correct granularity of constraints is required to provide helpful feedback messages without enabling students to guess solutions. Constraints that are too specific may encourage shallow learning whereas constraints that are too general may frustrate the students, due to their vague feedback messages. Since experts may also have misconceptions on the granularity of significant problem states, the correct granularity can only be determined by an empirical evaluation involving students [Martin & Mitrovic 2006, Martin & Mitrovic 2005]. Once an appropriate level of granularity is determined, the constraint algorithms have to be tweaked to produce constraints with the desirable level of granularity.

The process of producing a domain model using CAS is outlined as consisting of six distinct steps. However, the experience in producing domain models for the evaluated domains confirmed that it is an iterative process, where certain steps may have to be revisited to modify the previously composed domain components. For example, the domain expert may identify inadequacies in the domain ontology while modelling the solution structure (step two) or composing solutions (step four). In such situations, the domain ontology has to be modified by revisiting step one of the authoring process. CAS has been implemented to accommodate such behaviour, providing the author with freedom to modify previously composed domain model components.

5.3 *Effectiveness of CAS with Novice ITS Authors (Study 3)*

During the second evaluation study the role of the domain expert was played by the developer of CAS. As the developer of CAS has an in-depth knowledge of the authoring system implementation including the constraint generation algorithms, the results may have been biased. In real world scenarios, we envisage domain experts to have little or no prior experience in producing ITSs, let alone being knowledgeable of CAS's implementation. To simulate this scenario, we conducted an evaluation of CAS with novice ITS authors. We envisaged that this kind of evaluation would provide insights as to how potential users of the system would cope with it.

We carried out an evaluation study involving students enrolled in a graduate course of ITSs, where they were allocated the task of developing a domain model for fraction addition using CAS. The goal of the evaluation study was to validate four hypotheses;

1. *CAS is effective even when the required domain-dependant components are composed by a novice ITS author*

   CAS has been evaluated in a variety of domains, including database modelling, data normalisation and fraction addition in study 2. The results showed that CAS is effective in producing domain models for the evaluated domains (see Section 5.2 for details). However, the role of the domain author was played by the developer of CAS, who produced all the required domain-dependant components including the ontology, problems and solutions etc. Evaluation study 3 was conducted to verify that CAS can be equally effective even when the domain-dependant components required for the authoring process are composed by a novice to the authoring system as well as a novice to composing ITSs.

2. *The constraint generation algorithm depends on the ontology and an incomplete ontology would result in an incomplete constraint set*

Modelling an ontology of a domain is a design task, which depends on the creator's perceptions of the domain. The ontologies developed by two users, especially if the domain is complicated, are very likely to be different. Similarly, ontologies developed using CAS's ontology workspace, by the participants of the study would also be unique to each participant. The evaluation study would provide insights into the dependence of CAS's constraint generation algorithm on the ontologies composed by typical CAS users.

The study would also enable the evaluation of the system's ability to handle incomplete and partially correct domain ontologies. All previous evaluations were conducted with correct and complete ontologies produced by the developer of CAS. As participants are novices at modelling domain ontologies, incomplete ontologies can be expected. The experiment paves the way for observing the consequences of providing an incomplete ontology during the authoring process.

3. *The order of problems provided during the authoring process does not matter for constraint generation*

   The algorithm learns constraints by analysing problems iteratively comparing one solution to another for the same problem. During the second study where constraint generation algorithms were evaluated in a number of domains, the author introduced simple problems first and gradually introduced problems that dealt with more complicated concepts. Although the constraint generation is unlikely to be affected by the order of problems for most simple domains, it may be important in more complicated domains. This experiment provides us with an opportunity to obtain experimental evidence to support our hypothesis.

4. *The process of authoring constraints using CAS requires less effort than composing constraints manually*

   At the completion of the experiment, the amount of time required for composing constraints for each domain model produced by the participants would be calculated by analysing their logs. This would give an approximate indication of the time each participant spent interacting with CAS. The average time spent on composing a constraint can be compared to the average time per constraint (of 1.1 hours) reported by Mitrovic [Mitrovic & Ohlsson 1999].

The average time required for composing a constraint can also be compared to the average time taken to manually compose a constraint during study 1. Although constraints were composed for a different domain during that study, comparing the times would give a rough approximation of the difference in effort.

5.3.1 Procedure

The evaluation was carried out with students enrolled for the 2006 graduate course of Intelligent Tutoring Systems at the University of Canterbury. The 13 students enrolled for the course were assigned the task of building an ITS for the domain of adding two fractions. They were asked to compose the necessary domain model components to run a fractions addition tutor in WETAS using the complete version of CAS.

The fraction tutors that were to be produced by the participants were required to present problems where the sum of two fractions had to be calculated. The students using the fraction tutor were supposed to calculate the sum of the two fractions by initially calculating the lowest common denominator (LCD) of the two fractions. The two given fractions have to be converted to have LCD as their denominators (if needed). Then the sum of the converted fractions have to be calculated and finally the sum is simplified (if possible). The participants were given an outline (shown in Figure 5.21) of the interface for their fraction-tutors.

The students were required to develop the domain models for their tutoring systems using the full version of CAS. The full version of CAS facilitates the six step authoring process described in Section 4.1. It contains tools such as an ontology workspace and problem solution interface for composing domain dependent components required by the constraint generators. It also contains two textual editors for displaying and modifying constraints generated by CAS. The domain model can be tested by loading it in to WETAS,

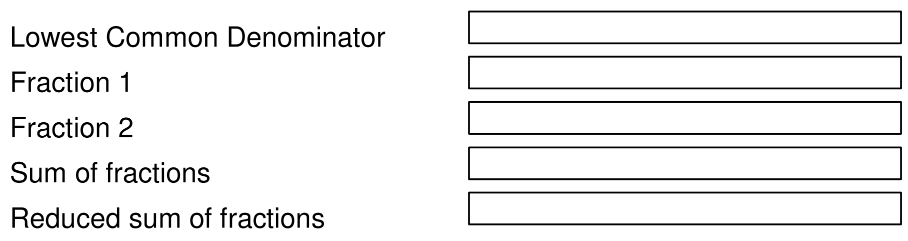


Figure 5.21: Fraction Addition Tutor Interface

which instantiates a web-based ITS.

The constraint generation process has to be initiated by the user, which results in the production of syntax and semantic constraints based on the domain-dependant component produced by the user. The constraints are added to the syntax and semantic text editors. However, as the constraints generated by CAS are in a high-level language, they are not directly runnable in WETAS. Participants of the study were required to produce a constraint base runnable in WETAS for fraction addition. They were free to use the generated constraints for guidance or assistance (i.e. to translate them into the WETAS constraint language).

The participants had attended 13 lectures on ITSs, including five on constraint based modelling. They were briefly introduced to CAS and WETAS. We provided them with a task description document that outlined the process of authoring a domain model using CAS and described the WETAS constraint language (See Appendix B for task description document). We recommended that the participants follow a six-step authoring process:

1. Compose an ontology of the domain,

2. Model the problem solving procedure for the domain,

3. Model the structure of solutions for problems of the domain,

4. Add problems and their solutions,

5. Automatically generate syntax and semantic constraints,

6. Translate the automatically generated syntax and semantic constraints to WETAS language or define new constraints.

In addition to the task description, participants were also given access to all the domain model components of LBITS [Martin & Mitrovic 2002*a*], a tutoring system for English language skills. The were also provided with an ontology for database modelling (Figure 5.5), as an example.

The participants were allocated a period of six weeks to complete the task. Although the students were allowed to work on the task for a total of six weeks, most students only started working on it at the end of week 3.

5.3.2 Interaction Times

The CAS environment logged all the significant actions performed by the participants during the process of composing domain models. The logs were analysed to produce summaries of participant interactions with CAS. Although all thirteen students in the course were novices in composing constraint bases, the task was satisfactorily completed by almost all the participants. One student failed to complete the final step of the authoring process where the system-generated constraints have to be translated to the WETAS language. His results are not included in the analysis.

All but one participant submitted their work before the dead-line. One participant opted to submit their work a week after the dead-line, which incurred a penalty of 15% from their final mark. As this study focuses on the final domain model produced by the participants, even the participant who completed the task late was included in the analysis without imposing any penalties.

The interface of CAS consists of two main tabs: the "ontology view" and the "domain model editor". The "ontology view" contains the ontology workspace and other tools necessary for composing the domain-dependant components necessary for each step in the authoring process. It contains a domain-structure composer for specifying domain characteristics, including the problem solving procedure, and a solution-structure composer for modelling the structure of solutions. It also contains a problem solution interface for adding problems and their solutions that are used for constraint

	Ontology workspace	Domain structure	Solution structure	Adding problems	Total
S1	2.57	0.08	0.10	0.82	3.57
S2	1.22	0.10	0.12	0.32	1.75
S3	4.98	0.10	0.23	1.68	7.00
S4	3.50	0.17	0.72	1.30	5.68
S5	7.48	0.08	0.78	4.70	13.05
S6	6.34	1.29	1.15	1.02	9.80
S7	2.67	0.43	0.38	0.50	3.98
S8	3.35	0.33	0.17	0.32	4.17
S9	2.28	0.42	0.88	3.47	7.05
S10	2.33	0.27	0.27	1.23	4.10
S11	1.62	0.13	0.18	0.38	2.32
S12	10.80	0.28	1.50	3.20	15.78
Mean	4.10	0.31	0.54	1.58	6.52
S.D.	2.83	0.33	0.46	1.44	4.34

Table 5.4: Interaction Times (hours) with CAS's Ontology View
s

generation.

A summary of the total amount of time spent interacting with each tool within the "ontology view" of CAS is given in Table 5.4. The participants spent the majority of the time interacting with ontology workspace. On average, they spent a total of 4.1 hours within the ontology workspace, ranging from 1.22 hours to 10.80 hours. The participant who submitted late (S12), accounted for the maximum interaction time. The large variation in the interaction times can be attributed to the different design methodologies used by each participant. Some may prefer to compose an ontology on paper and produce the final version using the ontology workspace, while others may use the ontology workspace to iteratively improve the quality of an ontology.

The participants spent 18 minutes (0.31 hours) on average interacting with the domain-structure composer tool. One participant (S6) spent almost four times the average time (1.29 hours). The time of 1.29 hours seem too high for a task that requires identifying the problem solving steps for adding two fractions. Further inspection of the logs revealed that the participant had been idle for long periods after activating the domain structure composing tool. The idle periods can be due to either the participant thinking about their actions or working on something else. However, the logs do not reveal the exact cause of the idle periods.

The average interaction time with the domain-structure composition tool, disregarding S6's extremely high interaction time of 1.29, is 13 minutes (0.22 hours). This time can also be considered too high, since the task only required making the decision of whether fraction addition problems needed to be solved in a strict procedural manner and if so outlining the set of steps. We believe that the interface may have confused the participants. It offers a choice between "non-procedural" and "procedural". The participants, who were novice ITS authors, may not have been comfortable with the terminology. A more intuitive question such as "When solving a problem, is there a particular order that needs to be followed?" may be easier for novices to comprehend.

The domain authors are expected to model the structure of solutions for the domain's problems during the second phase of the authoring process. The participants spent approximately half an hour on average (0.54 hours) composing the solution structure. The interaction times varied from a maximum of one and a half hours to six minutes (0.1 hours). The majority of the participants (7 out of 12) found the task relatively straight-forward and required under half an hour in total. As the solution structure depends on the concepts of the ontology, participants who had incomplete ontologies may have struggled with the task.

After completing the solution structure for the domain, problems and their solutions can be added using CAS's problem solution interface within the "ontology view". It contains a form-based interface, dependant on the solution structure, for composing solutions. The added problems and solutions are stored within CAS as objects. They are automatically converted to the Lisp format required by WETAS and added to the problems editor. The participants were also free to add problems and their solutions directly through the problem editor.

Most participants added at least one problem through CAS's "ontology view". Analysis of logs revealed that some chose to add all problems through this interface, while others only added the initial problems using the interface and later added problems directly in the Lisp format using the text editor. This behaviour was reflected in the times spent on adding problems through the "ontology view". The interaction times varied from 19 minutes (0.32 hours) to over four and a half hours with a mean interaction time of one and a half hours. We believe that the problem solution interface is useful for novices at the start. As they gained experience with the Lisp representation, they found the textual editor to be more efficient.

The participants spent a total of six an half hours on average interacting with the ontology workspace and the suite of tools offered by CAS. The majority of the time was spent on interacting with the ontology workspace. There was a very high variance in the total interaction time, which can be attributed to each individual's ability and the high variance in interacting with the ontology workspace.

The participants used the textual editors that were available under the "domain model editor" tab to modify/add domain model components required for WETAS, including problems and their solutions, syntax and semantic constraints and macros. The total times spent interacting with each of the editors are given in Table 5.5. These total times are only an approximation of the time that the participants actually spent working on the domain model, as the study was not completely controlled. Some participants had left the system logged on, while they were away from their computer. While analysing the logs, we came across a few instances where some had left the system logged on during the course of a whole weekend. The times given in Table 5.5 do not include inactive periods longer than an hour.

The participants used the problem editor for a total of approximately three hours on average. The interaction times vary from half an hour to over ten hours. Typically, participants who chose to add problems and their solutions through the "ontology view", spent less time with the problem editor, while others spent more time with the editor. However, there was no significant inverse correlation between the times spent adding problems through the "ontology view" and times spent interacting with the problem editor.

CAS generates syntax and semantic constraints in a high-level language

	Problems	Syntax	Semantic	Taxonomy	Total
S1	0.57	2.80	8.10	0.33	11.80
S2	1.32	19.82	5.72	0.68	27.53
S3	1.07	8.82	14.33	3.78	28.00
S4	0.40	6.57	13.85	0.82	21.63
S5	2.47	24.13	6.35	2.03	34.98
S6	4.92	19.00	7.80	3.08	34.80
S7	4.13	1.77	7.03	1.60	14.53
S8	3.10	8.67	4.48	0.23	16.48
S9	1.08	12.08	9.07	0.05	22.28
S10	3.40	10.27	3.20	0.08	16.95
S11	4.15	14.32	5.03	0.28	23.78
S12	10.42	15.60	17.70	0.50	44.22
Mean	3.08	11.99	8.56	1.12	24.75
S.D.	2.77	6.87	4.46	1.24	9.63

Table 5.5: Interaction Times with CAS's Text Editors

and adds these constraints to the respective constraint editors. Although the automatically generated constraints were not in the correct format for WETAS, their purpose was to assist the participants in composing a domain model runnable in WETAS. The participants were free to use the generated constraints or ignore them.

The participants spent a total of 12 hours on average composing syntax constraints in the syntax editor. The times ranged from 1.77 hours to 24.13 hours. The participants spent less time (on average) composing constraints within the semantic editor (eight and a half hours). The times ranged from 3.2 hours to a maximum of 17.7 hours. The variations can be attributed to a variety of factors. One of the common factors is the students' ability; some are better and faster in programming constraints than others. In some cases, the participants' familiarity with Lisp would assist them in composing constraints. Another factor could be the limitation of debugging facilities available for WETAS. It provides very basic syntax checking and does not contain a feature to investigate the variable bindings after the evaluation of a constraint. So, participants who encountered bugs in their constraint bases spent a considerable amount of time debugging the constraints.

The WETAS constraint language allows macros to be defined to perform common functions. These macros can be defined in the taxonomy editor. Macros are not essential for the domain of adding fractions due to its simple nature. Only a few participants opted to use macros, and this is reflected in the total interaction times. While the mean interaction time in the taxonomy editor is just over one hour, half the participants spent less than half an hour interacting with it.

The participants spent a total of 24.75 hours on average interacting with the textual editors. The total interaction times varied from a minimum of 11.8 hours to 44.22 hours. The maximum interaction with the text editors was reported by the participant who completed the task a week later than others. Due to the lack of debugging facilities in WETAS, the interaction times with the editors depend a lot on whether the constraints composed by the participants contained bugs or not.

5.3.3 Analysis of Produced Constraint Sets

The constraint bases produced by participants were analysed and evaluated for completeness. In order to evaluate the completeness of constraint bases, we manually compiled a list of pedagogically significant problem states for the domain of adding two fractions. The list contained eight syntactically significant problem states for the domain. They ensure that each component of a solution, such as the LCD and converted fractions are in the correct format and the final sum is arrived at by following the correct problem solving procedure. The list also contained thirteen semantically significant problem states, to ensure that each significant component of a solution, such as a numerator and denominator of each fraction is correct.

The number of problem states covered by each participant's constraint base was calculated through manual investigation and is given in Table 5.6. It lists the total number of constraints (syntax and semantic) composed by the participants under the "Constraints" column. The total number of problem states that each constraint base covers is listed under the "Problem states" column. The completeness of the constraint-base, calculated as the percentage of problem states accounted for by each constraint-base, is given under the "Completeness" column.

	Constraints		Problem states		Completeness	
	Syntax	Semantic	Syntax (out of 8)	Semantic (out of 13)	Syntax	Semantic
S1	5	12	5	7	63%	54%
S2	5	13	5	12	63%	92%
S3	4	12	4	12	50%	92%
S4	16	16	5	12	63%	92%
S5	14	18	8	13	100%	100%
S6	15	11	5	12	63%	92%
S7	2	5	3	3	38%	23%
S8	8	13	7	4	88%	31%
S9	5	8	4	4	50%	31%
S10	7	11	5	12	63%	92%
S11	4	18	5	12	63%	92%
S12	9	16	6	1	75%	8%
Mean	7.83	12.75	5.17	8.67	64.58%	66.67%
S.D.	4.73	3.89	1.34	4.50	16.71%	34.61%

Table 5.6: Total Numbers of Constraints Composed by Participants

The participants produced approximately eight syntax constraints and thirteen semantic constraints on average. The total number of syntax constraints composed by students ranged from a minimum of two constraints to a maximum of 16 constraints. In the case of semantic constraints, the range of total constraints was between five and 18. The participants had accounted for five out of the eight significant syntactic problem states on average, which amounted for 64%. The participants also accounted for 66% of the significant semantic problem states on average (8.6 out of 13). From the total group of participants, only one participant (S5) had accounted for all the significant problem states for the domain. The majority of the remaining participants accounted for over half of the problem states. Almost every participant had accounted for over 30% of the significant semantic problem states. However, one participants (S12), had struggled with composing semantic constraints.

The participants were informed about the authoring process using CAS via the task description document. It contained a description of the six step process along with examples. At the end of the authoring process, CAS generates both syntax and semantic constraints, that are available via the respective constraint editors. However, the constraints generated by CAS are not in the final Lisp-based form, and are not runnable. The task of composing constraints in the final Lisp-based form was assigned to the participants. They were free to use the generated constraints as suggestions or to ignore them.

In order to evaluate the effectiveness of CAS and its sensitivity towards the quality of the domain dependant components, we used the domain dependant components composed by each participant to automatically generate a set of constraints. A summary of the results are presented in Table 5.7. The first two columns contains total numbers of syntax and semantic constraints generated by CAS. The generator produced an average of nine syntax constraints and 15 semantic constraints. The constraint generator did not produce constraints due to a bug in the system for two participants: S6 (syntax constraints) and S3 (semantic constraints). The semantic constraints generator relies on problems and their solutions added through CAS's problem-editor to produce constraints. As participant S8 had not added any solutions using CAS's problem solution interface, the semantic constraints generator failed to generate any semantic constraints for the participant.

The syntactically significant problem states for the domain of adding two fractions in WETAS includes five states for verifying the syntax of the inputs, such as whether the LCD is an integer and whether entered fractions are syntactically valid. They need to be verified due to the generic nature of the interface that can be produced in WETAS (see Figure 5.21). As students are not restricted in what they can add in the input boxes, constraints are needed to ensure that students have specified syntactically correct terms. For example, constraints are required to verify that fractions are of the format "numerator / denominator". However, these constraints are redundant for a more restrictive solution composition interface that would be produced by CAS from a complete ontology (see Figure 5.22). It contains two text boxes for inputting a fraction, ensuring that each fraction has the two required components. Furthermore, as text boxes only accept the specified type (integers in this case), constraints such as the one to verify that the LCD is an integer is also redundant.

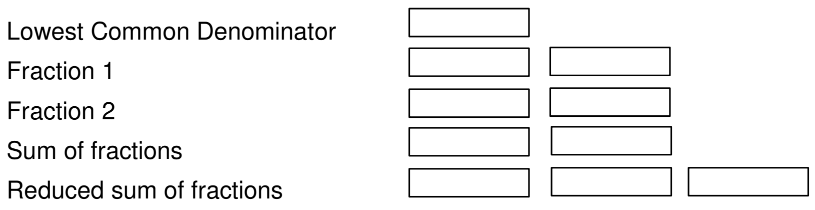


Figure 5.22: Solution Addition Interface Generated by CAS from a Complete Ontology

We calculated the total number of significant problem states covered by the generated constraints (given in columns four and five of Table 5.7) to evaluate the completeness of each generated constraint-set. Due to the simple nature of the domain, only three syntactical constraints are required to verify that a student follows the correct problem solving path for the type of problem solving interface shown in Figure 5.22. CAS generated all three syntax constraints for all but one of the participants. As shown in the "Completeness" column (Table 5.7), the CAS-generated constraints were complete in almost all cases. There was only one situation where only two of the required set of constraints were generated (S12). This was a result of an incorrectly specified solution structure.

The syntax constraints generator also produced extra constraints that are trivially satisfied with a restricted interface as shown in Figure 5.22. They verify that the inputs are of the correct syntax. For example, two constraints were generated to ensure that each fraction composed in the student solution contains a numerator and a denominator. Although these constraints are trivially satisfied, these constraints are necessary for interfaces with more freedom such as the interface depicted in Figure 5.21.

For the task of adding two fractions, we identified 13 semantically significant problem states. CAS only has the ability to generate constraints that accounted for 12 out of the 13 problem states, as it is unable to generate constraints that require algebraic functionality. Consequently, CAS cannot generate a constraint that verifies that a student has entered a common multiple of the two denominators larger than the lowest common multiple. So

	Numbers		Problem states		Completeness	
	Syntax	Semantic	Syntax	Semantic	Syntax	Semantic
S1	7	15	3	12	100%	92%
S2	8	12	3	12	100%	92%
S3	18	0*	3	0	100%	
S4	6	13	3	0	100%	0%
S5	9	8	3	12	100%	92%
S6	0*	23	0	1		8%
S7	13	26	3	1	100%	8%
S8	6	0*	3	0	100%	
S9	11	23	3	1	100%	8%
S10	9	12	3	12	100%	92%
S11	7	12	3	12	100%	92%
S12	17	42	2	1	67%	8%
Mean	9.25	15.50	2.67	5.33	96.97%	49.00%
S.D.	4.97	11.70	0.89	5.90	10.05%	45.46%

(* - no constraints generated due to bugs in CAS)

Table 5.7: Number of Constraints Generated by CAS

the maximum degree of completeness that can be expected is 92%.

CAS generated semantic constraints that covered a maximum possible 12 problem states from domain-dependant components supplied by five participants. The semantic constraints generated for the remaining participants covered either one or no problem states. Further analysis revealed that there were two main reasons for CAS not generating useful constraints. One of the reasons was that two of the participants (S4 and S9) added empty duplicate solutions for problems. Currently, the constraint generation algorithm does not check for empty duplicate solutions. So it would assume that even an empty solution can be accepted as a correct solution for the problem. The generated constraints were incorrect as they allowed for empty solutions as well. This situation can be avoided easily by restricting the solution interface from saving empty solutions.

Another common mistake that the participant had made is modelling an incomplete ontology. Four participants (S4, S6, S7 and S12) modelled the "fraction" concept with only a single property of type *String*. Modelling the fraction concept with a single property would result in a set of constraints that compare each component of the student solution against the respective ideal solution component as a whole. These constraints are not of the correct level of granularity and can only perform basic checks to ensure that entered solution components are correct or not. Consequently, the resulting feedback is very limited in pedagogical significance. For example, the constraints would have the ability to denote that the student has made an error in converting the first fraction to LCD, however, they would not be able to pinpoint whether the student had made a mistake in the numerator or the denominator. We believe that the decision to model the fraction concept with a single property may have been influenced by the student interface and solution representation of WETAS. Since WETAS only allows a single input for each solution component, the participants may have attempted to produce an ontology and solution structure in CAS that mimics the student interface of WETAS.

The constraint generator failed to produce any semantic constraints for two participants, S3 and S8. It failed to generate constraints for S3 due to a bug in the system. The other participant, S8, did not add any solutions through CAS's solution interface. The semantic constraint generator only has access to problems and solutions added via the solution interface. As problems and solutions are essential for semantic constraint generation, it would fail to produce any constraints without sample problems and solutions.

We hoped that the participants would use the constraints generated by CAS as a starting point and translate them into the WETAS language. However, as the generated constraints were appended to the free form constraint editor, we could not obtain an accurate measure of whether the high-level constraints were used to produce constraints runnable in WETAS. Assuming all participants read and understood the constraints suggested by CAS, the completeness of the manually composed constraint set can be compared to the completeness of the generated constraint set. Four out of the five participants, whose generated sets of constraints were 92% complete, also had manually composed constraint sets that were over 92% complete. The fifth participant's manually composed constraint set was only 53% complete. This observation suggests that the participants were also able to produce a highly accurate constraint set runnable in WETAS in situations where CAS generated highly accurate constraint sets. However, the completeness of the generated constraint set is not consistent with the completeness of the manually generated constraint set. Although the generator did not produce any constraints for participants S3 due to a bug, he managed to manually compose a near complete constraint set. Furthermore, generated constraints were too general for S4 and incorrect for S6, but they managed to produce near complete constraint sets. Conversly, both the generated sets of constraints and the manually composed sets of constraints were equally poor for participants S7, S8, S9 and S12. The observation is also confirmed by the fact that there is no significant correlation between the accuracies of the generated set and manually composed set.

There can be numerous reasons for the observation of no direct correlation between the completeness of the generated set and the manually composed set. One explanation is that the participants for whom very poor constraint sets were generated had disregarded the generated constraint sets and composed constraints manually using their individual abilities. One of the participants encountered a bug and no constraints were generated. Other participants may have had misconceptions about the authoring process and/or misconceptions about using the tool. The flexibility of the tool allowed these participants to manually add constraints that were not generated by the system.

Although we assumed that the generated high-level constraints assisted the participants, there was little evidence in their reports that supported this assumption. Only one participant indicated that the generated constraints assisted him. Since no explanation of the high-level constraint representation was provided to the participants, they may have struggled to understand the notation (see Figure 5.15 for examples) and find commonalities between the two representations.

5.3.4 Discussion

The version of CAS that was provided to the participants of the experiment only generated constraints in an English-like high-level language. However, as there is a direct mapping between the high-level representation and the Lisp representation it can be performed automatically (See Appendix A for a detailed outline of an example translation). At the completion of generating the syntax and semantic constraints, they can be translated directly into the required format. This would save the effort required for manual translation and provide ITS authors with the opportunity to concentrate on modifying the generated constraints and composing any missing constraints.

Analysing the results from evaluation study 3 confirmed all our hypotheses;

1. *CAS is effective even when the required domain-dependant components are composed by a novice ITS author*

   The results of the evaluation study showed that CAS was effective in generating constraints for the task of adding fractions with the assistance of a novice ITS author. It revealed that CAS was able to generate all the required syntax constraints for the domain for all but one participants. Furthermore, CAS generated over 90% of the semantic constraints for half of the participants. Considering that the participants were given very little or no training in using the authoring system, the results are very encouraging. Providing the users with more training and improving CAS to be fully integrated with a tutoring server (similar to WETAS) would increase its effectiveness.

2. *The constraint generation algorithm depends on the ontology and an incomplete ontology would result in an incomplete constraint set*

   Ontologies created by different people are most likely to be different as they are based on the creator's perceptions. During this study, the constraint learning algorithm generated constraints using ontologies developed by the 12 participants. Although the ontologies had differences, the syntax constraint generation algorithm managed to produce fully complete constraint sets for almost all participants. However, the semantic constraint generation was more sensitive to the ontology. In particular, it was reliant on defining the "fraction" concept correctly.

The semantic constraint generator managed to produce an almost complete constraint set with the correct granularity for ontologies with a correctly defined "fraction" concept. By contrast, the constraints generated for ontologies with a partially defined "fraction" concept were too general. They compared each fraction composed by students as a whole against its corresponding fraction in the ideal solution. These constraints result in feedback with limited in pedagogical significance.

3. *The order of problems provided during the authoring process does not matter for constraint generation*

   The participants of this experiment composed problems and their solutions by themselves and ordered them according to their preference. As problems for adding fractions only have one correct solution, all participants provided a single solution for each problem. The results indicated that the order of the problems or solutions had no effect in the process of generating constraints. The constraint generator produced 92% of the semantic constraints required for the domain for all participants who had composed the ontology appropriately and provide at least two problems and their solutions.

4. *The process of authoring constraints using CAS requires less effort that composing constraints manually*

   Currently, CAS is only capable of generating constraints in an English-like high-level language. The generated constraints have to be translated manually to the correct Lisp-based form. In order to obtain a measure for the effort required for producing constraints using CAS, we calculated the average total time required for producing a single constraint. Only the participants, whose domain model components resulted in generating near complete constraint bases were used for calculating the average effort, to ensure that incorrectly generated constraints were not accounted. The five participants (S1, S2, S5, S10, S11), for whom almost complete constraint sets were generated, spent a total of 24.8 hours interacting with the "ontology view" of CAS. They spent a total of 115.04 hours interacting with the textual editors producing a total of 107 constraints. Considering times spent interacting with the "ontology view" and the textual editors, the participants spent a total of 1.3 hours on average to produce one constraint.

The average time of 1.3 hours per constraint is very close to the 1.1 hours reportedly required by Mitrovic [Mitrovic & Ohlsson 1999] to compose a single constraint while building SQL-Tutor. The time estimated by Mitrovic can be considered as biased since she is an expert in SQL and knowledge engineering in addition to being an expert in composing constraints. Therefore, the achievement by novice ITS authors producing constraints in a time similar to the time reported by Mitrovic is significant. Furthermore, the time of 1.3 hours is a significant improvement from the two hours required by the participants of study 1. The participants of study 1 and 3 are at comparable levels and are both novices to composing constraints.

Although CAS currently generates constraints in a high-level language, it can be improved to generate constraints directly in the language required for execution. Assuming the generated constraints were produced in the required runnable form, the total of 99 syntax and semantic constraints were produced from 24.8 hours of interaction with the "ontology view". Consequently, the participants only required an average of 15 minutes (0.25 hours) on average to generate one constraint.

The average time of 15 minutes per constraint is a significant improvement from 1.1 hours reported by Mitrovic [Mitrovic & Ohlsson 1999]. The time is more significant as the authoring process was driven by novice ITS developers. The average time of 15 minutes per constraint is an improvement of eight fold over the time required by ITS authors of a similar level during study 1. However, this does not take into account the effort required for validating the generated constraints. The effort required for validating constraints can be minimised by using CAS's constraint validation tool, that enables a constraint-set to be evaluated on a set of student and ideal solutions. It produces a report outlining the constraints that were violated and satisfied. As the constraint generation algorithm may not produce all the required constraints, the domain author may also be required to modify the generated constraints or add new constraints manually.

The evaluation study conducted with novice ITS authors produced very encouraging results. The syntax constraint generator was extremely effective, producing complete constraint sets for almost every participant. The semantic constraints generator produced constraint sets that were over 90% accurate for half of the participants. Although the generator produced overly general semantic constraints for the remaining participants, ITS authors can modify them with little effort to produce constraints of correct granularity. We believe that this would contribute towards the reduction of the author's total workload.

Currently constraints produced by CAS are not runnable. However, it can be enhanced to directly produce constraints that are runnable. This would dramatically reduce the effort required for composing constraints. The evaluation revealed that the total workload required to produce a single constraint reduces by almost eight fold, if CAS produced constraints in the desired representation.

5.4 Summary

We conducted three evaluation studies to evaluate the effectiveness of CAS. The initial evaluation study was conducted to verify our hypothesis that composing a domain ontology assisting the process of manually composing constraints. The study involved novice ITS authors manually producing constraint bases using a tool that encouraged the use of ontologies. The results of the study confirmed our hypothesis; the process of composing a domain ontology indeed facilitates the creation of complete constraint bases.

The second evaluation study was conducted to evaluate the effectiveness of CAS in generating constraint bases for different domains. We used CAS to generate constraints for the three significantly different domains of ER modelling, fraction addition and data normalisation. CAS was provided with complete domain ontologies and collections of problems and their solutions composed by a domain modelling expert. This study demonstrated that

CAS was extremely effective in producing constraint bases for the evaluated domains, resulting in constraint bases that were over 90% complete.

The final study evaluated the effectiveness of CAS in producing domain models with the assistance of novice domain authors. A group of fourth year computer science students who had no prior experience in composing constraint bases produced constraint bases using CAS for the domain of adding two fractions. CAS was able to produce the complete set of essential syntax constraints for all but one of the participants. Furthermore, CAS was capable of generating over 90% of the required semantic constraints for half the participants. The experiment also demonstrated that CAS has the potential to dramatically reduce the overall effort required to produce domain models, if the constraints are produced in the desired Lisp representation.

Chapter VI

Conclusions

Producing a domain model that enables the adaptiveness of ITSs is a complicated process. It requires much time and effort. For example, Mitrovic [Mitrovic & Ohlsson 1999] reported that she spent over an hour on average to produce a single constraint for SQL-Tutor, consisting of over 700 constraints. In addition to requiring extensive effort to complete the task, it requires multiple facets of expertise. Typically, this is a collaborative effort between a knowledge engineer, AI programmer and a domain expert.

This research explored ways of providing support for domain experts to produce domain models with little training. In doing so, we have made several contributions to the field of ITS. The main contribution of this research is an authoring system for producing domain models for CBM tutors, named CAS. A summary of the main contribution is given in Section 6.1. This section also includes a summary of the series of evaluations that produced very promising results. An account of the other significant contributions are outlined in Section 6.2. Future research directions that build on the outcomes of this research are described in Section 6.3 and some closing remarks are given in Section 6.4.

6.1 Main Contribution

WETAS [Martin 2002, Martin & Mitrovic 2002*a*], an authoring shell for constraint-based modelling, has reduced the amount of work required to produce constraint-based tutors. It provides general implementations of all the functions essential for a constraint-based ITS, such as evaluating answers, modelling students, selecting problems, selecting feedback. In order to start a tutor in WETAS, the author needs to provide a model of the domain, which includes syntax and semantic constraints, problems and solutions. The main limitation of WETAS is its lack of support for composing the required domain model. In order to fill this gap this research developed CAS, an authoring system for Constraint-based modelling.

CAS was designed with the goal of opening the door for domain experts with little or no programming knowledge to produce a domain model required for a constraint-based ITS. Domain authors using CAS are required follow a six step process to develop domain models. Firstly, they are required to model an ontology of the domain using the ontology workspace provided by the system. Secondly, the composition of solutions to problems of the domain need to be outlined. During the third step, CAS uses the ontology and the solution structure to generate syntax constraints for the domain. The author provides sample problems and their solutions during step four. In order for CAS to be able to generate semantic constraints that identify correct solutions arrived at using different methodologies, the author needs to provide multiple solutions to a problem, outlining different methods of solving it. The system then (in step five) generates semantic constraints by analysing the provided problems and their solutions. Finally in step six, the domain expert validates the system-generated constraints by perusing the high-level descriptions of each constraint.

The syntax constraints are generated by translating the syntactical information embedded in the ontology into constraints. The syntax constraint generator identifies the concepts relevant to solutions by going through the solution structure and searching for restrictions specified in those concepts. The identified restrictions are used to generate syntax constraints. It also generates extra constraints to ensure that students follow the appropriate problem solving path by analysing the solution structure for procedural tasks.

Semantic constraints are generated by CAS using a machine learning algorithm that analyses the problems and their solutions provided by the domain expert. The algorithm generates constraints by comparing and contrasting two solutions to the same problem. Constraints are generated iteratively, and they are generalised or specialised during subsequent analysis of others pairs of solutions.

Unlike previous authoring systems that focused only on procedural tasks, CAS is capable of acquiring domain knowledge for both procedural and non-procedural tasks. The process of authoring a domain model for both types of tasks are similar with the exception of having to outline the problem solving steps for procedural tasks. The constraints produced for procedural tasks are relevant for only a single problem-solving step, whereas constraints generated for non-procedural tasks are always relevant.

A series of evaluation studies were conducted to assess the effectiveness of CAS. The first study was conducted to verify whether the creation of an ontology and organising constraints according to its concepts would assist in the process of manually composing constraints. The results revealed that the task of composing ontologies had indeed assisted in the process of composing constraint bases.

The effectiveness of CAS's constraint generation was evaluated in a study that generated domain models for three vastly different domains. The selected domains included a non-procedural tasks (Database modelling) and two procedural tasks (Data normalisation and Fraction addition). The domain models of KERMIT (a database modelling constraint-based tutor) and NORMIT (a data normalisation constraint-based tutor) were compared against the domain models generated by CAS. Analysis of the constraints revealed that the generated constraints accounted for over 90% of the constraints in the domain model of the respective constraint-based tutors.

CAS was used to generate constraint-bases for the domains of Database modelling, Data normalisation and Fraction addition. The domains of Database modelling and Data normalisation were specifically chosen as we had already developed ITSs for the domains. Their domain models were used as bench marks to gauge the completeness of the constraint bases generated by CAS. The domain of Fraction addition was chosen as it is a simple domain, which almost every one is comfortable with. The constraint base required to model all significant problem states of the domain is also small. Moreover, the domain is of the sufficient complexity to be set as a task for novices to produce a constraint base.

The final evaluation study conducted to evaluate CAS's effectiveness in generating constraints with the assistance of novice ITS authors, produced promising results. The study included twelve fourth year computer science students (with no prior experiencing in composing constraints), who were assigned the task of producing a domain model for adding two fractions using CAS. The syntax constraints generator of CAS was able to produce all the required constraints for all but one of the participants. CAS also generated over 90% of the required semantic constraints for half the participants, who had modelled complete ontologies. Incomplete ontologies resulted in an overly general set of semantic constraints, that would produce less significant pedagogical feedback.

Furthermore, CAS failed to produce constraints containing domain dependant functions. Domain dependant functions encapsulate declarative knowledge specific to the domain. For example, the domain of Fraction addition requires a domain function capable of identifying higher order common multiples of two denominators. Currently, CAS is not capable of allowing the author to provide the required domain dependant functions.

6.2 Other Significant Contributions

In addition to the Constraint Authoring System (CAS), this research makes several original contributions to the field of ITS research.

6.2.1 Ontology Workspace

Domain ontologies play a central role in the knowledge authoring process of CAS. Syntax constraints are generated by directly translating the restrictions specified in the ontology. The semantic constraints generator uses example solutions provided by the domain expert added via the problem/solution interface. As both syntax and semantic constraint generation algorithms depend heavily on the domain ontology, it is imperative that the domain expert develops a comprehensive ontology for the domain.

One of the goals of CAS was to enable domain experts with little or no programming and knowledge engineering experience to produce a domain model required for an ITS. In order to achieve this goal, CAS should contain a tool that enables users to quickly and easily model a domain ontology. The tool should be intuitive to use, minimising the need for training. It should also encourage the users to compose complete and accurate ontologies. In order to achieve these requirements, an ontology workspace that supports the composition of ontologies graphically was created.

The ontology workspace of CAS, described in Section 4.3.2, was designed to be similar to a drawing tool. Users can compose ontologies by "drawing" concepts as rectangles on the canvas and "draw" specialisations using arrows. The workspace allows users to compose ontologies with little training, and also organise concepts to resemble a hierarchical structure. The tool also contains input forms for specifying restrictions on properties and relationships. After a relationship is composed, the ontology workspace engages the user in a dialogue to ensure that the relationship is between the correct sets of concepts.

The ontology workspace is capable of converting the graphical model of the ontology to an XML representation and vice versa. Although the ontology workspace uses a proprietary XML format, it can be converted to a standard representation such as DAML or OIL by performing an XSLT transformation. This can also be employed to load existing ontologies from ontology repositories.

The decision to implement a customised ontology workspace for CAS was taken after evaluating a set of state-of-the-art ontology composition tools. All the tools evaluated were designed for experts with features needed for knowledge engineers. The ontology workspace of CAS was designed to have a restrictive interface that was sufficient for extracting only the essential information about domain concepts from a domain expert.

The ability of domain experts with little or no programming and knowledge engineering experience to develop domain ontologies using CAS's ontology workspace is yet to be evaluated. Although the fourth year Computer Science students who participated in the evaluations have little or no knowledge engineering experience, they possess other skills that assist in the task of composing an ontology. It is unreasonable to assume typical domain experts would perform in a similar manner in composing ontologies. Further evaluations are needed to gauge whether domain experts are capable of composing complete domain ontologies expected by CAS.

6.2.2 Problem/Solution Interface

The semantic constraints generator produces constraints by analysing example problems and their solutions. It requires solutions for a problem to be composed as a set of instantiations of concepts in the ontology. This enables the solution to be decomposed into meaningful elements to be compared with another solution. In order to minimise the effort required for composing a solution, we developed a solution composition interface that was dependant on the ontology and the solution structure.

The solutions composition interface (see Section 4.6 for details) was designed to ensure that users composed solutions that strictly adhered to the ontology and the solution structure. It reduced the effort in composing instances of concepts by providing form-based input screens that enforced populating each property of the concept. Choosing elements that participate in a relationship was made easy by providing drop-down-lists for selecting elements involved.

CAS's solution editor also allows the user to modify a copy of the primary solution. This feature dramatically reduces the effort required to construct alternate solutions, since alternative solutions in most domains have many similarities. The solution editor also ensured that the semantic constraint generation algorithm has an accurate map of matching elements between solutions by verifying each new match with the user.

6.2.3 Domain Model Authoring Tool for WETAS

WETAS is an authoring shell for constraint-based Intelligent Tutoring Systems. It provides all the domain independent components of a web-based ITS including the user interface, pedagogical module and student modeler. In order to start a tutoring system in WETAS, it should be provided with a domain model. However, WETAS does not provide any support for composing a domain model. It expects the components of the domain model (syntax constraints, semantic constraints, problems and solutions) as a collection of files in the appropriate directory.

Typically constraints are composed using a text editor to modify the text file containing either syntax or semantic constraints. Composing constraints is not a well-defined procedure. The author is responsible for composing constraints that cover all the significant problem states of the domain. As constraints are modular in nature, authors are very likely to miss constraints, resulting in incomplete constraint bases. We believe that it is highly beneficial for the author to develop a domain ontology even when the constraint sets is developed manually, because building an ontology helps the author to reflect on the domain. Such an activity would enhance the authors understanding of the domain and therefore be a helpful when identifying constraints. Furthermore, we also believe that categorising constraints according to concepts of the ontology would assist in producing complete constraint bases.

In order to assist authors in manually composing constraints, this research developed a tool that supported modelling an ontology and encouraged the grouping of constraints according to concepts of the ontology (discussed in detailed in Section 5.1.1). It was integrated with WETAS to function as a front end for domain authors. Its effectiveness was evaluated with a group of novices in authoring ITS domain models. Analysis of results strongly suggested that the participants used the ontology's concepts to group constraints developed more complete constraint bases. Their subjective comments also confirmed this observation.

6.3 Future Directions

A number of research avenues are opened as a result of this research. Firstly, the authoring system developed in this research can be enhanced by providing more support for domain ontologies. A major limitation of the current version of CAS is its inability to import ontologies composed in standard ontology languages such as DAML and OWL. As a consequence, CAS is unable to make use of ontologies available in ontology repositories that store ontologies in standard representations. The domain author's workload can be significantly reduced by enhancing CAS to be able to import ontologies from other repositories. Even though these ontologies may not be complete, the domain author can improve them after importing.

Secondly, the constraint generation can be improved to generate constraints directly in the runnable Lisp form. Currently, as the constraints are generated in a high-level language, they have to be manually translated to the Lisp form. This requires much effort and is a task that requires programming expertise. The third evaluation study (see Section 5.3 for details) showed that enhancing the constraint generators to produce constraints in the final form would dramatically reduce the time and effort required for composing constraint bases. The syntax constraints can be produced directly in the Lisp form by replacing the set of templates that are used to generate them. The semantic constraints on the other hand have to be converted to the Lisp form by introducing a new set of templates that map the high-level constraints to the desired Lisp form.

Thirdly, CAS can be improved by providing guidance for domain authors to assist them in following the authoring process. This would minimise the load on novices to the authoring system being overwhelmed by having to follow a six step authoring process. The guidance can be in the form of an interactive tutorial or even an animated agent that guides the domain author in the authoring process.

Fourthly, CAS's semantic constraint generation algorithm can be improved to generate constraints that contain domain dependant functions. This would enhance the effectiveness and robustness of the constraint generation algorithm. It can be achieved by requesting the domain expert to specify the domain functions prior to generating constraints. The provided domain functions can be used by CAS's semantic constraint generation algorithm while determining matches between elements of two correct solutions to a problem.

The effectiveness of the constraint generation algorithms can also be enhanced by improving the ontology representation in order to model declarative knowledge specific to domains. The ontology workspace can be enhanced to model such domain semantics graphically or to allow authors to directly specify domain dependant functions. For example, consider the task of adding two fractions. The ontology representation can be enhanced to represent the semantics of common multiples of two denominators. Using such a representation, a 'Common multiple' concept that represents the semantic relationship between two denominators can be added to the ontology. The 'LCD' concept can be a sub concept of 'Common multiple' with an the added condition of being the minimum 'Common multiple'.

Another area of future research is to assess CAS's effectiveness in various domains to further determine its strengths and weaknesses. These evaluations can be used to identify the characteristics of domains for which CAS can be used to successfully generate constraints. This would also identify the type of domains for which CAS would be unsuccessful at generating constraints. Evaluation studies can also be used to examine whether teachers with no prior experiences in producing ITSs, in particular school teachers or university lecturers, would be able to produce an ITS for their students.

Currently, CAS only uses correct solutions (to problems of the domain) provided by the domain expert for generating constraints. As domain experts have knowledge of students' common misconceptions, they would be able to provide incorrect solutions highlighting these misconceptions. The constraint generation algorithm can be made more robust by enhancing it to make use of both correct and incorrect solutions.

The main focus of CAS was producing constraint bases with the assistance of domain experts. CAS can be enhanced by tightly integrating it with WETAS to provide customisations to tutors running in WETAS based on classrooms. These customisations, such as the problem selection strategy, the look and feel of the interface, the greeting message etc. should be easily modified by classroom teachers.

6.4 Concluding Remarks

Intelligent Tutoring Systems are being increasingly used in real classroom settings producing significant learning gains. Ideally, the teachers of the classroom should be able to produce ITSs according to their needs. However, building an ITS requires extensive effort and multi-faceted expertise. In particular, the domain model, which is a formal representation of the domain, requires months of work that can only be carried out by experts in knowledge engineering and AI programming. The contribution of this research enables domain models to be generated automatically with the assistance of an expert of the domain, such a teacher or a lecturer. With ITSs being able to be produced by domain experts, they are poised to have a much significant and wider role in on education in the near future.

Appendix A

Translating Pseudo-code Constraints into Lisp-code: an Example

The constraint generators of CAS produces constraints in a high-level pseudo-code representation. This pseudo-code representation can be transformed automatically to a runnable Lisp-code representation by enhancing the constraint generators. The following sections outline a methodology in which the pseudo-code representation can be mapped to the Lisp-code representation found in SQL-Tutor [Mitrovic 2003*a*]. The representation of solutions expected by the Lisp-code constraints is described in section A.1. Section two outlines the conversion of syntax constraints into the Lisp form and section three outlines a methodology for converting semantic constraints into the Lisp form.

A.1 Solution Representation

Solutions in CAS are represented as objects in memory. They consist of a collection of components and each component contains a list of elements. The constraint language used in SQL-Tutor expects solutions to be represented in a textual format as lists. As a result, the object representation of solutions has to be converted to the expected list format.

An element of a solution can be represented by a list of property values. The name of the concept should also be included in the list of values as it identifies the type of the element. For example a 'Student' entity with a tag of 'E1' (see Figure 4.23) can be represented as '('Regular entity' E1 Student)'.

Figure A.1 contains a general template for representing solutions in the list format. It contains a list for each solution component. These lists contain list representations of elements delimited by the '@' symbol.

- <component-name1> - (@ <concept-name> <property1-value> <property2-value> ... @ <concept-name> <property1-value> <property2-value> ...)
- <component-name2> - (@ <concept-name> <property1-value> <property2-value> ... @ <concept-name> <property1-value> <property2-value> ...)
- <component-name3> - (@ <concept-name> <property1-value> <property2-value> ... @ <concept-name> <property1-value> <property2-value> ...)

Figure A.1: Solution Representation

A.2 Syntax Constraints

The syntax constraint generator can be enhanced to produce constraints in the executable Lisp form by replacing the current set of templates with templates that produce Lisp code. An example template for a syntax constraint that ensures the value of a particular property is always greater than a specified minimum value is given in Figure A.2. Its relevance condition would have to perform a pattern match to identify an element of the correct type within the student solution. The regular expression of '(?* @<concept-name> ?var1 ?var2 ... ?*)' can be used to match against all elements of the particular <concept-name> type. The satisfaction condition simple verifies the syntactical condition on the property.

A.3 Semantic Constraints

The semantic constraint generation algorithm generalises and specialises constraints using the high-level representation. At the conclusion of constraint generation, the high-level constraints in the main constraint base can be transformed into the Lisp code representation by identifying mappings between the two representations. For example, consider the high-level constraint that ensures the student solution contains all the required *Regular Entity* elements (See Figure A.3). It can be transformed to the Lisp form

- High-level representation
  Relevance: SS has a <concept-name> element
  Satisfaction: Its <property-name> must be greater than <min-value>

- Lisp code representation
  Relevance: (match '(?* @ <concept-name> <property1> <property2> ... ?*) (<component-name> SS) bindings)
  Satisfaction: (>= <property1> <min-value>)

Figure A.2: Example Lisp-code Template for a Syntax Constraint

shown in the figure. It uses the properties generalised by introducing variables and wild cards during the constraint generation process to identify matching elements between the ideal solution and the student solution.

- High-level representation
  Relevance: *Entities* component of IS has a (?*, ?var2) *Regular Entity*
  Satisfaction: *Entities* component of SS has a (?*, ?var2) *Regular Entity*

- Lisp code representation
  Relevance: (match '(?* @ 'Regular Entity' ?IS_var1 ?var2 ?*) (Entities IS) bindings)
  Satisfaction: (match '(?* @ 'Regular Entity' ?SS_var1 ?var2 ?*) (Entities SS) bindings)

Figure A.3: Semantic Constraint that Ensures the Student Solution Contains all the Required *Regular Entity* Elements

The constraint language used in SQL-Tutor uses '?*' to represent wild cards that can be matched against zero or more values. Consequently, the wild cards produced by the high-level representation (which match against only a single value) have to replaced by two (or more) distinct variables stating that their values do not have to be equal (See Lisp code constraint in Figure A.3). The '?*' symbols are required within the pattern to match against all elements in the solution.

The translation from the high-level representation to the Lisp-level representation can be automated by producing a set of general templates that outline the mappings between the two representations. Figure A.4 contains an example of a template that can be used to transform a high-level semantic constraint that ensures the existence of the required elements in the student solution (e.g. constraint in Figure A.3). As properties have been generalised by introducing variables and wild cards, they can be used for pattern matching within the student solution.

Semantic constraint for ensuring the existence of a matching element in the student solution

- High-level representation
  Relevance: <component-name> component of IS has a (<pattern>) <concept-name>
  Satisfaction: <component-name> component of SS has a (<pattern>) <concept-name>

- Lisp code representation
  Relevance: (match '(?* @ <concept-name> <pattern> ?*) (<component-name> IS) bindings)
  Satisfaction: (match '(?* @ <concept-name> <pattern> ?*) (<component-name> SS) bindings)

Figure A.4: Example Semantic Constraint Template for Lisp-code Mapping

Appendix B

Study 3 Task Outline

COSC420 Intelligent Tutoring Systems

Homework 3: Implementing a Constraint-based Tutor

Due date: 11 May 2005, 5pm **Weight:** 25% of the final

1. The task

In this assignment you will use WETAS to implement a simple constraint-based ITS. Important features of WETAS have been discussed in lectures and are also described in this handout. WETAS (Web-Enabled Tutor Authoring System) is an ITS authoring shell that allows the rapid development of new constraint-based tutoring systems. Four systems have been implemented using WETAS: SQL-Tutor, Language Builder ITS (LBITS), EER-Tutor and COLLECT-UML. LBITS teaches school children about English grammar and vocabulary, by presenting them with word puzzles that they have to solve. You will have full access to one activity, called "Last two letters". LBITS is available at http://ictg.cosc.canterbury.ac.nz:8004/wetas/. To try LBITS, follow the link for Language Builder, use any name to log on, and leave the password blank.

For this assignment you will build a new tutor, which teaches fraction addition. You will develop the ontology, the constraint set (including any macros) and the problem set.

2. The instructional domain

Your tutor should support children learning how to add fractions, with an interface similar to Figure 1. You should define at least 10 problems for students to solve.

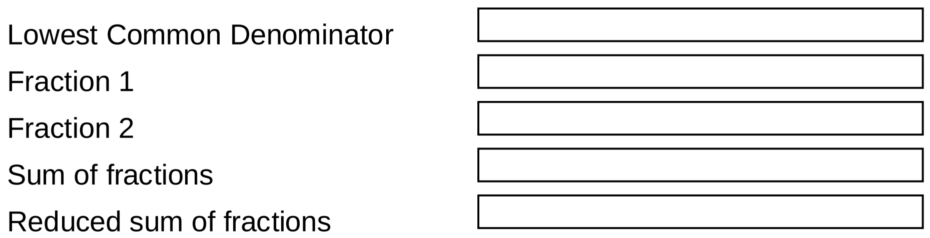


Figure 1: Fraction Addition Tutor Interface

3. Building your tutor

To implement a tutor in WETAS, you need to provide only domain specific information, such as the problems and the knowledge base. WETAS automates all tutoring functions. The process of developing a new tutor consists of the following phases:

1. Develop the ontology
2. Define the features of the domain

3. Define the solution structure
4. Add problems and solutions
5. Develop constraints

You need to download the front-end of the authoring tool and follow the instruction given at http://ictg.cosc.canterbury.ac.nz:8080/. The domain model components of LBITS are also linked on the same page. An example ontology can be viewed by logging into the front WETAS front-end using “er” as the as the user name, and no password.

To develop your own tutor, use your COSC username to log on, and the password that you will receive in an email. You are not permitted to look at each others’ domain models, or to run each other’s tutors.

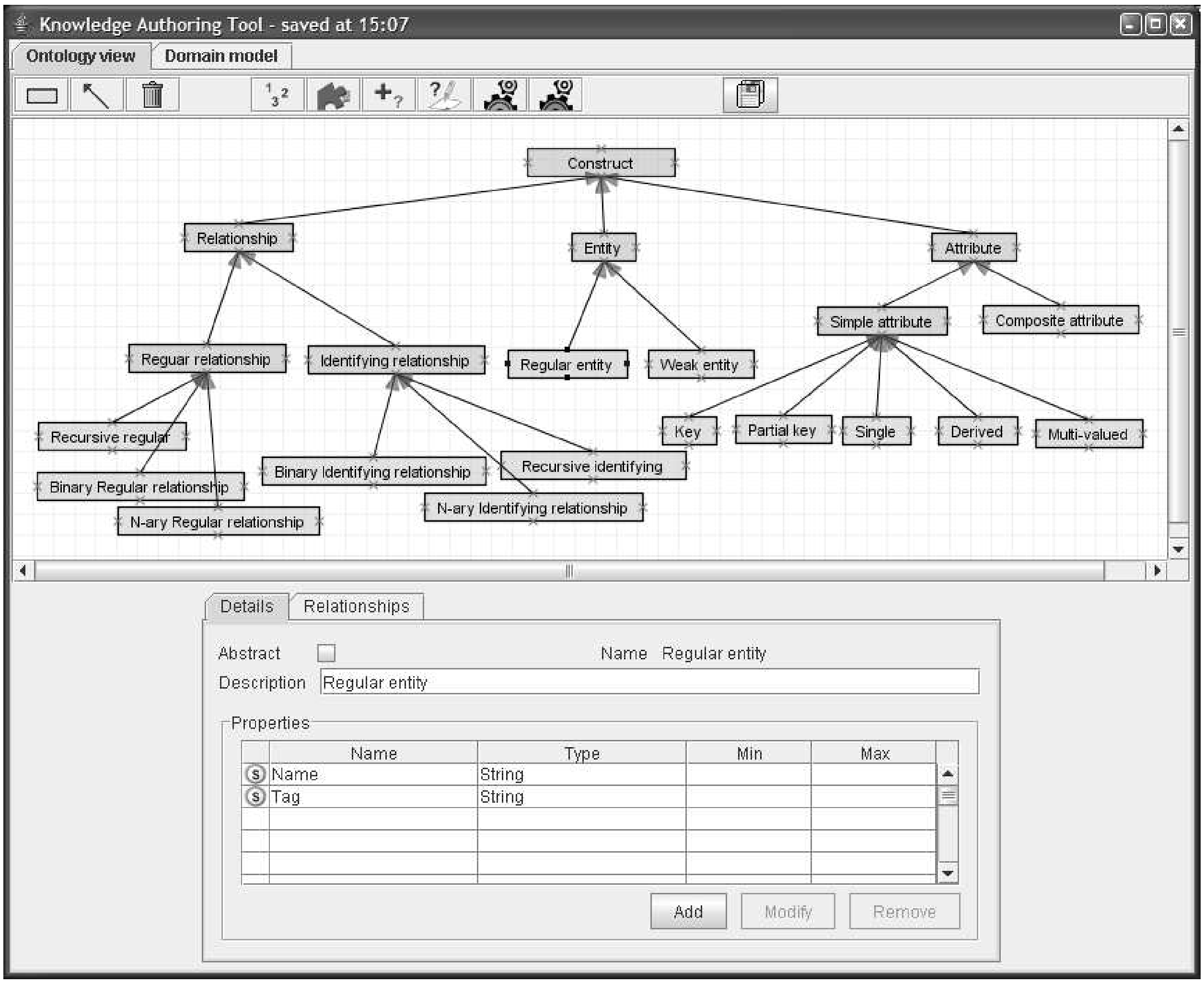
4. Developing the ontology

1. An ontology describes the structure of the domain by showing the basic domain concepts, their properties and the relationships between concepts. A widely accepted definition is that an ontology is a specification of a formalisation (Gruber, 1993); in other words, it is an explicit, formal specification of the domain vocabulary which presents a common understanding of topics that can be communicated between users and applications. An ontology thus enables all the people involved to speak the same language, supporting knowledge sharing by applications and reuse. An ontology makes domain assumptions explicit, so that it is easier to change the domain description, as well as to understand and update existing data. An important feature of ontologies is that they separate domain knowledge from operational knowledge, in the same way in which a database schema is separated from the actual data stored in a database.
2. There is no silver bullet when it comes to ontology development; similar to other design tasks, ontology development is under-specified and ambiguous. Therefore, there is neither one correct approach to ontology development, nor a single best ontology for a particular domain (Noy and McGuinness, 2001). In order to specify a domain ontology, the author needs to specify domain concepts, their properties (attributes) and relationships between concepts. Each ontology will reflect the author’s subjective view of the domain, and of the importance of domain concepts. The process is always iterative. Initially, it is necessary to decide on the scope of the ontology - how much of the domain will it cover?
3. When developing an ontology, it is necessary to identify the important concepts. Roughly speaking, these concepts will include all types of entities appearing in the domain that students need to know about. Once concepts are known, they need to be arranged into a taxonomy, using the specialisation/generalisation relationship. This relationship is also commonly referred to as the 'is-a' relationship, or the 'a-kind-of' relationship. Taxonomy can be specified using a top-down or a bottom-up approach, or a combination of the two, which is probably most common. When using the top-down approach, the ontology is developed starting from the most general concepts, which are then refined into subclasses. The bottom-up approach, on the other hand, starts from specific concepts which are generalised into superclasses. Every concept in the ontology is important because of its properties and/or relationships to other concepts; therefore, properties and relationships need to be defined. The properties of a concept will be inherited by all of its subconcepts.

The WETAS front-end, as shown in Figure 2, consists of two main tabs. The “ontology view” provides a workspace for composing ontologies, and the “domain model” tab contains a set of text editors for defining domain components such as constraints, problems etc. The application automatically saves the all domain model components every 10 minutes. The last time the domain model was saved is displayed in title bar of the application.

The ontology displayed in Figure 2 represents the concepts of ER modelling, which describes data as entities, attributes and relationships. The ontology contains *Construct* as the most general concept. *Relationship*, *Entity*, *Attribute* are sub-concepts of *Construct*. *Relationship* is specialised into *Regular* and *Identifying*, which are the two types of relationships and *Entity* is specialised, according to its types, as *Regular* and *Weak*. Subclasses of *Attribute* are *Simple* or *Composite* attributes and *Simple* attributes are further specialised into five categories: *Key*, *Partial key*, *Single*, *Derived* and *Multi-valued*.

When you select a concept from the ontology, you will be able to see all the attributes and relationships for that concept at the bottom of the page. To define an attribute, use the “Add” button. You will get a pop-up window, asking for the name and the type of attribute. If you specify a numerical type, you can also add the minimal and/or maximal value for the attribute. Closing the application using the close window button also saves and logs you out.



Abstract ☐ Name Regular entity

Description Regular entity

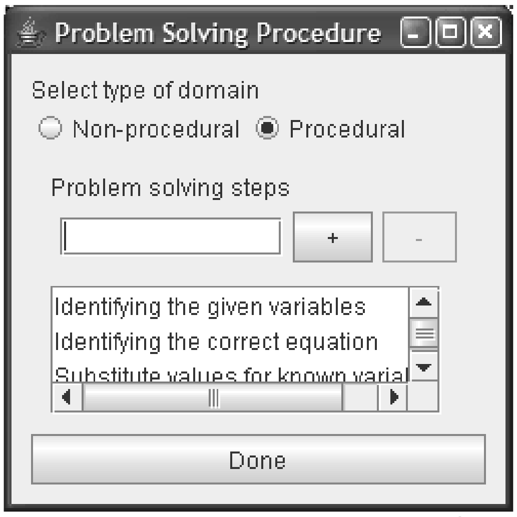
Properties

Name	Type	Min	Max
Name	String		
Tag	String		

Figure 2: Ontology Editor Interface

5. Defining the characteristics of the domain

The front-end of WETAS characterises domains as declarative by default. For procedural domains, the set of problem solving steps that have to be followed needs to be enumerated. For example, consider a tutoring system that provides a practice environment for applying physics equations where students are given a textual description to calculate a specific value. The procedure for producing the final solution (as shown in Figure 3) involves four steps; identifying the given variables and their values, identifying the correct equation, substituting the values of known variables in equation and finally solving equation.

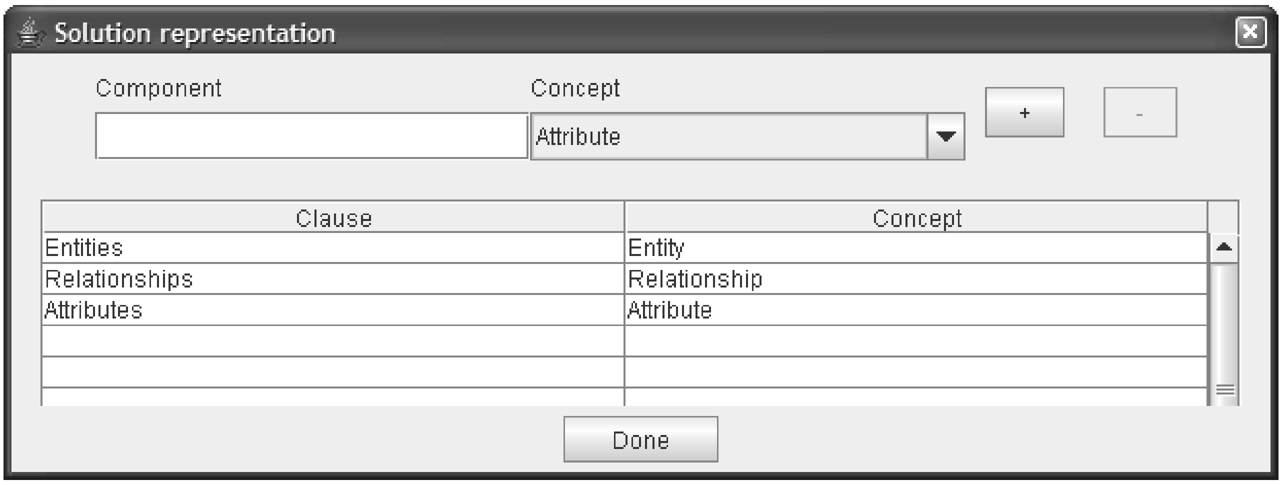


Identifying the given variables
Identifying the correct equation
Substitute values for known varial

Figure 3: Domain Characteristics Interface

6. Defining the solution structure

Problems in WETAS are represented as textual statements. Solutions on the other hand can consist of a number of components representing meaningful components. You will need to decide upon how the composition of a solution. For example, the structure of a solutions for problems in ER modelling (see Figure 4) would consist of three components; Entities, Relationships and Attributes.

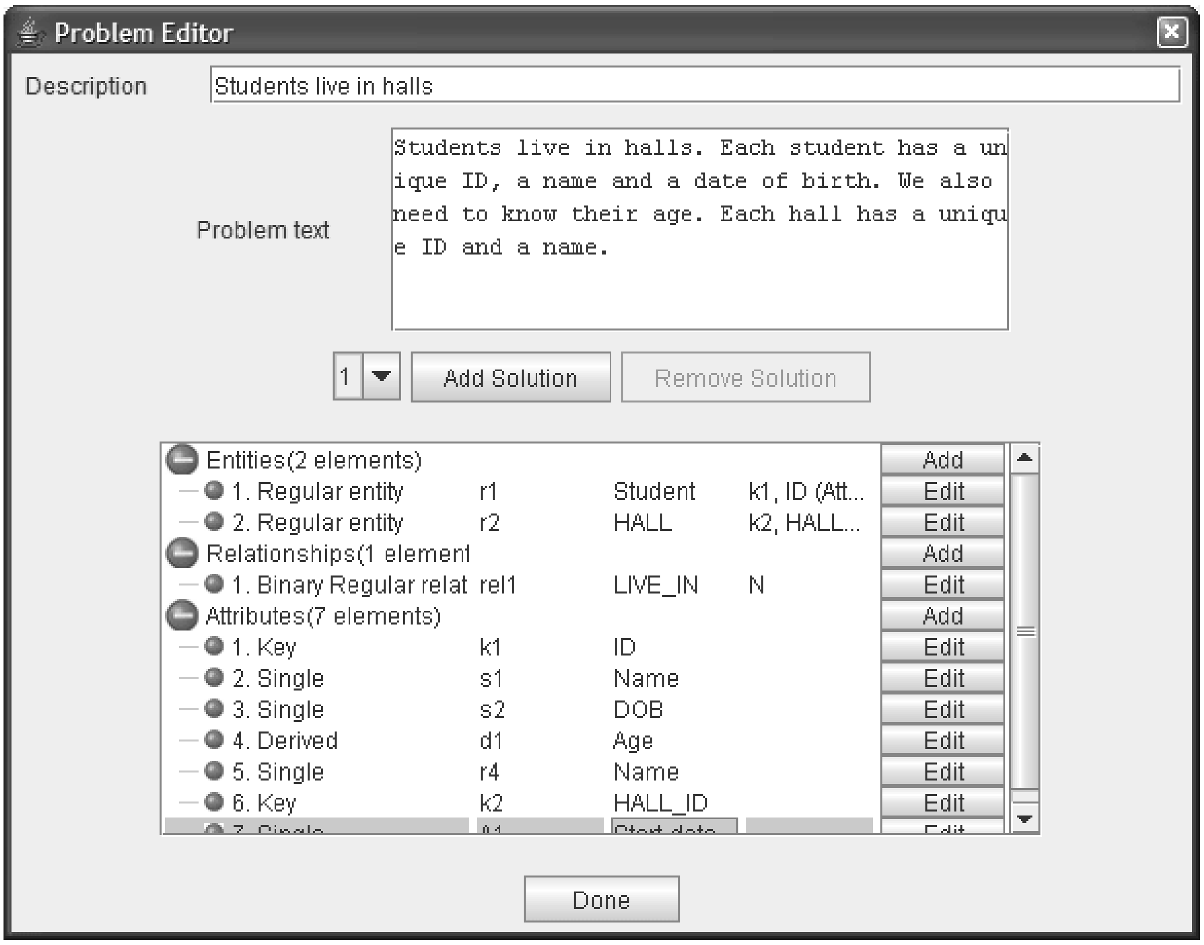


Clause	Concept
Entities	Entity
Relationships	Relationship
Attributes	Attribute

Figure 4: Solution Structure Interface

7. Adding problems and solutions

The front end for WETAS provides input forms for adding problems and their solutions. The input form for composing a solution depends on the solution structure described in step 6 (Figure 5). The input forms are automatically generated to create instances of concepts modelled in the ontology. Each property of the instance would result in an input box where as each relationships would result in a drop down list for selecting an already created instance.



Problem Editor

Description: Students live in halls

Problem text:
```
Students live in halls. Each student has a un
ique ID, a name and a date of birth. We also
need to know their age. Each hall has a uniqu
e ID and a name.
```

Entities(2 elements)				Add
1. Regular entity	r1	Student	k1, ID (Att...	Edit
2. Regular entity	r2	HALL	k2, HALL...	Edit
Relationships(1 element				Add
1. Binary Regular relat	rel1	LIVE_IN	N	Edit
Attributes(7 elements)				Add
1. Key	k1	ID		Edit
2. Single	s1	Name		Edit
3. Single	s2	DOB		Edit
4. Derived	d1	Age		Edit
5. Single	r4	Name		Edit
6. Key	k2	HALL_ID		Edit
7. Single	t1	Start date		Edit

Figure 5: Problem Editor Interface

In order to generate constraints the system requires multiple correct solutions for each problem outlining different ways of solving the same problem. Once all problems and their solutions are added, they can be converted to the lisp representation, stored internally within WETAS by clicking on the "write problems" button.

WETAS will store the problems and solutions internally in the following format:

```
(<problem-num>
 <problem text>
 <difficulty>-      not used, specify NIL
      ((<clause name1>
             (<subclause-name1-1> <answer1-1> <clue1-1> <def-input1-1>)
```

```
          (<subclause-name1-2> <answer1-2> <clue1-2> <def-input1-2>)
          …
          (<subclause-name1-m> <answer1-m> <clue1-n> <def-input1-m>))
          …
     (<clause namen>
          (<subclause-namen-1> <answern-1> <cluen-1> <def-inputn-1>)
          (<subclause-namen-2> <answern-2> <cluen-2> <def-inputn-2>)
          …
          (<subclause-namen-m> <answern-m> <cluen-m> <def-inputn-m>))))
```

Note that LBITS uses this structure, but has only one clause ("CLUES"). Note also that the number of subclauses does *not* need to be consistent between problems, e.g. one problem may have four words to solve, and another may have six.

The problem statement is used to paint the screen (with the clause name, subclause names, and clues), and as the ideal solution for evaluating the student's answer. For the latter, only the clause/subclause names and the answers are used. For example:

```
(1
   NIL
   NIL
; IS - (#   answer      clue                    default-input)
   (("CLUES"
            ("1" "road"       "long street"           "ro")
            ("2" "adventure"  "exciting journey"      "")
            ("3" "rest"       "stop for a while"      "re")
            ("4" "stone"      "small rock"            "")
            ("5" "nest"       "home for a bird"       "") )))
```

is presented to the constraint evaluator as:

("CLUES" ("1" "road") ("2" "adventure") ("3" "rest") ("4" "stone") ("5" "nest"))

Hence, `(MATCH IS CLUES (?n ?word))` will match to each of the subclauses in turn. It is *not* possible for the constraints to access the *question* parts of the problem statement.

8. Develop constraints

The WETAS front-end generates both syntax and semantic constraints by analysing the ontology of the domain and the supplied problems and solutions. Once the ontology is complete, the syntax constraints can be generated by clicking on the green "constraint generation" button. Semantic constraints can be generated by clicking on the blue "constraint generation" button after problems and their solutions have been added. The constraint generator only generates a high-level description of the constraint. You are supposed to convert the high-level description to the WETAS constraint language.

9. Constraint Language

Both the constraints and the problem definitions are written in special representation languages, that are based on the LISP programing language. The general form of a function call in LISP is:

`(<function> <param1> <param2>…)`

See the files in LBITS for examples. Constraints all have the following format:

(<constraint identifier>
 <feedback message>
 <relevance condition>
 <satisfaction condition>
 <clause name>)

The constraint identifier need not be numeric, but it *must* be unique within your domain. The relevance and satisfaction conditions consist of calls to the functions MATCH (to pick words out of a string), TEST (to test the value of a variable), and TEST_SYMBOL (to test for letters in a word). These may be joined together using AND, OR-P and NOT-P. The latter are functionally identical to OR and NOT, but *must* be used in their place, or strange behaviour may result.

In WETAS, constraints are encoded purely as pattern matches. Each pattern may be compared either against the ideal or student solutions (via a MATCH function) or against a variable (via the TEST and TEST_SYMBOL functions) whose value has been determined in a prior match. An example of a constraint in SQL-Tutor using this representation is:

```
  (34
  "If there is an ANY or ALL predicate in the WHERE clause, then the
  attribute in question must be of the same type as the only expression of
  the SELECT clause of the subquery."
  ; relevance condition
  (match SS WHERE '(?* ?a1 ("<" ">" "=" "!=" "<>" "<=" ">=")
     ("ANY" "ALL") "(" "SELECT" ?a2 "FROM" ?* ")" ?*))
; satisfaction condition
  (and (test SS (^type (?a1 ?type))
       (test SS (^type (?a2 ?type))))
  "WHERE")
```

This constraint tests that if an attribute is compared to the result of a nested SELECT, the attribute being compared and that which the SELECT returns have the same type. *^type* is a macro, and is explained later in this document. The constraint representation consists of logical connectives (*and*, *or-p* and *not-p*) and three functions: *match*, *test*, and *test_symbol*. These are now described.

MATCH

This function is used to match an arbitrary number of terms to a clause in the student or ideal solutions. The syntax is:

(MATCH <solution name> <clause name> (pattern list))

where <*solution name*> is either *SS* (student solution) or *IS* (ideal solution) and <*clause name*> is the name of the problem/solution clause to which the pattern applies. However, the notion of clauses is not domain-dependent; it simply allows the solution to be broken into subsets of the whole solution. The <*pattern list*> is a set of terms, which match to individual elements in the solution being tested. The following constructs are supported:

?*	**wildcard**: matches zero or more terms that we are not interested in. For example, `(MATCH SS WHERE (?* ?a ?*)` matches to any term in the WHERE clause of the student solution, because the two wildcards can map to any number of terms before and after ?a, so all possible bindings of this match gives ?a bound to each of the terms in the input.
?*var	**named wildcard**: a wildcard that appears more than once, hence is assigned a variable name to ensure consistency. For example: `(AND (MATCH SS SELECT (?*w1 "AS" ?*w2)      (1)` `(MATCH IS SELECT (?*w1 ?N ?*))     (2)` First, (1) tests that the SELECT clause in the student solution contains the term "AS". Then, ?*w1 in (2) tests that the ideal solution also contains all the terms that preceded the "AS", and then maps the variable ?N to whatever comes next. The unnamed wildcard at the end of (2) discards whatever comes after ?N.
`?var`	**variable**: matches a single term. For example, (MATCH IS SELECT (?what)) matches `?what` to one and only one item in the SELECT clause of the ideal solution;
`"str"`	**literal string**: matches a single term to a literal value. For example, in (MATCH SS WHERE (?* ">=" ?*)) ">=" must appear in the WHERE clause of the student solution.
`(lit1, lit2, ...)`	**literal list:** a list of possible allowed values for a single term. For example: `(MATCH SS WHERE (?* ?n1 (">=" "<=") ?n2 ?*))` assigns the variable ?n1 to any term preceding either a ">=" or a "<=", and ?n2 to the term following it. Note that because ?n1 and ?n2 are not wildcards, they must map to a single term each, hence if the ">=" or "<=" is either at the start or the end of the clause this match will fail, because one (or both) of ?n1 and ?n2 will fail to match.

Variables and literals (or lists of literals) may be combined to give a variable whose allowed value is restricted. For example:

```
  (MATCH IS ORDER_BY (?* (("ANY" "ALL") ?what) ?*))
```

means that the term that the variable *?what* matches to must have a value of "ANY" or "ALL". There is no limit to the number of terms that may appear in a literal list, or in a MATCH in general.

TEST

Having performed a MATCH to determine the existence of some sequence of terms, we often wish to further test the value of one or more variables that were bound. This is carried out using the TEST function, which is a special form of MATCH that accepts a single pattern term and one or more variables. For example (the following constraint is simplified):

```
(2726
"Check you have used the correct logical connective in WHERE to represent a
range of numbers."
(and
 (match SS WHERE (?* ?n1 ?op1 ?what1 (("and" "or") ?lc) ?n1 ?op2 ?what2) ?*)
 (match IS WHERE (?* ?n1 "between" ?*)))
(test SS ("and" ?lc))
"WHERE")
```

This constraint first tests for an attribute (*?n1*) in the WHERE condition of the student solution that is being compared to two different values (*?what1* and *?what2*). Then, the second match looks for the same attribute being used in a BETWEEN construct in the ideal solution. If this is the case, the two tests in the student solution must be ANDed together. The TEST function call in the satisfaction condition does this. The syntax of the TEST function is:

```
(TEST <SOLUTION NAME> <TEST-TERM>)
```

where SOLUTION NAME is again IS or SS, and <TEST-TERM> is a single value test, such as a test against a literal or list of literals. In the previous example, a single value test is made for the value "and". In effect, TEST performs a specified pattern match, where the pattern contains just a single match term, on a list that contains just the value of the variable in question. Examples:

▪ (<value> <var>)	("and" ?word1)
▪ (<values> <var>)	(("and" "or") ?v1)
▪ (<var1> <var2>)	(?t1 ?t2)
▪ (<macro>)	(^valid-table ?t)

TEST may also used for variable assignment, as follows:

```
(TEST SS (?word1 ?word2))
```

This statement tests that the value of ?word2 is equal to the value of ?word1. If ?word2 is currently not bound, it will be set to the value of ?word1. Note that it only works this way round: ?word1 will never be modified by this statement.

TEST_SYMBOL

Often, we also need to be able to test characters within the value of a term. For example, a valid SQL string is defined as a single quote, followed by any characters, and closed with another single quote. To test this, we add the function TEST_SYMBOL, which acts exactly like the MATCH function, except it accepts a variable name instead of a clause name, and further parses the value of the variable binding into individual characters, before applying the match pattern. For example, to test for a valid SQL string in the variable ?str:

```
(TEST_SYMBOL SS ?str ("'" ?* "'"))
```

This test would succeed for values of *`?str`* such as `"'Kubrick'"` for example, but fail for `"'Smith"` because of the missing closing quote. The general syntax is:

```
(TEST_SYMBOL <SOLUTION> <VAR> <PATTERN>)
```

Note that in both TEST and TEST_SYMBOL, the solution name is passed as a parameter, even though it doesn't appear to be necessary since these tests are on already bound variables, not an input string. However, this is required because the test may be a *macro*, which may perform further pattern matches on the input, so it needs to know which solution to match. Macros are now described.

Macros

The original version of SQL-Tutor uses domain-specific functions to extract features of the solutions and to make special comparisons between them. In the new representation, this is forbidden, because it hides the logic of the test. In SQL-Tutor, almost all domain-specific functions test for a valid value, or pair of values. For example, in:

```
  (valid-table (find-schema (current-database *student*)) '?t1)
```

"Valid-table" tests that *?t1* is a valid table name in the student's current database. Similarly:

```
(attribute-of (find-table ?t1 (current-database *student*)) '?a1))
```

tests that *?a1* is a valid attribute in the table *?t1*. Routines such as "find-table" and "current-database" are simply data accessors. In both "valid-table" and "attribute-of", the function might alternatively be represented as a membership test on an enumerated list: for "valid-table", the list will contain the set of table names for a given database, while for "attribute-of", each member of the list will be a tuple of type (<attribute> <table>). Since our language already supports testing against lists of literals, these can be encoded using the pattern matching language, i.e.

```
  (TEST SS (("MOVIE" "DIRECTOR"...) ?t1))
```

which tests that *?t1* is a valid table, and

```
  (TEST SS (("TITLE" "MOVIE") ("YEAR" "MOVIE") ("LNAME"
  "DIRECTOR")...) (?a1 ?t1))
```

which tests that *?a1* and *?t1* form a valid attribute/table combination, i.e. that *?a1* is an attribute of table *?t1*.

We have now replaced function calls with pattern matching, however it would be cumbersome to have to enumerate all attributes of all tables every time we wish to perform such a test. To overcome this, we use macros to represent partial pattern matches that are used often. For example, the macro for *^attribute-of* used previously is:

```
  (^attribute-of (??a ??t)
      (TEST SS ((("TITLE" "MOVIE") ("LNAME" "DIRECTOR")...)
                (??a ??t))))
```

The syntax of a macro definition is:

```
  (<MACRO NAME> (<PARAMETERS>) <BODY>)
```

The name must always begin with a "^" so that macros can be easily identified by the constraint compiler. Similarly, the parameter names are preceded by "??" so that they can be distinguished from local variables in the macro body. In MATCH and TEST statements, the solution name should always be "??" (this gets substituted for the caller's solution name at compile time). Local variables must begin with "?_", e.g. `?_word1`. The body may be any valid condition, including logical connectives, MATCH functions, and other macro calls. Consider the following example from SQL:

```
(^attribute-alias (??name ??attr ??table)
  (and (test ?? (^name ??name))
       (or-p (test ?? (^attr-name (??name ??attr ??table)))   (1)
             (match ?? SELECT (?* (^attr-name (?_a1 ??attr
                                    ??table)) "AS" ??name))))) (2)
```

This macro accepts an attribute name as input, and returns the physical attribute and table names. In SQL, attributes can be aliassed, i.e. they can be assigned another name. For example:

```
  SELECT movie.number AS num
  FROM movie
  ORDER-BY num
```

In this example, *num* is defined as an alias for *movie.number* in the SELECT clause, and is used again in ORDER-BY. To test that *num* in ORDER-BY is a valid attribute, we need to know what it maps to, which is achieved by the *^attribute-alias* macro: If *??name* fails the test in (1), i.e. it is not a valid attribute name, (2) tries to match it to an alias definition in the SELECT clause. Hence, the macro needs to know which solution it is testing. The constraint that tests for a valid attribute in ORDER-BY is:

```
(149
"You have used some names in the ORDER BY clause that are not from
this database."
(match SS ORDER_BY (?* (^name ?n) ?*))
(test SS (^attribute-alias (?n ?a ?t)))
"ORDER BY")
```

When the constraint set is loaded, the macro names are expanded into their corresponding pattern matches. The parameter names in the macro definition are substituted for those passed in, and the "??" solution name placeholders are replaced with the solution name from the caller. Hence, all routines that can call a macro (i.e., MATCH, TEST and TEST_SYMBOL) must specify a solution name. Note that macros may also be embedded in pattern matches, and that the macro being called may have more than one parameter. For example:

```
(match SS SELECT (?* (^attr-name (?n ?a ?t)) ?*))
```

In this case, the *first* parameter (*?n*) is matched to a term in the input string, with *?a* and *?t* being either tested or instantiated by the macro, depending on whether or not they are already bound.

10. Running your tutor

Once the necessary domain model components have been completed (syntax constraints, semantic constraints, problems and solutions and macros), you need to load your files using:

http://ictg.cosc.canterbury.ac.nz:8004/wetas/reload?name=NAME,

where NAME is your COSC username (in uppercase). Be patient, this may take some time, depending on the size of your constraint and problem files. If there are mistakes in your files, you will receive an error message. Your tutor will be available at:

http://ictg.cosc.canterbury.ac.nz/wetas/NAME

To log in, you may enter *any* username. However, you *must* use the password you were provided with, otherwise WETAS will not let you use the system (note that LBITS does not require a password). You can now try out your system. Note that if there are errors in your code (e.g. in the constraints), the system will present you with an error message in the "feedback" window at the bottom of the screen.

11. What to hand in

In you report, please include a page or less describing what you did, and also a page or less describing any problems you found, or any suggestions you might have about improving WETAS. Please describe your experiences with using the ontologies, using the problem editor, using WETAS and the constraints generated by the system.

References

1. Gruber, T.R. (1993) A Translation Approach to Portable Ontology Specification. Knowledge Acquisition, 5, 199-200.
2. Noy, N., McGuinness, D. (2001) Ontology Development 101: a Guide to Creating your First Ontology. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880.

Appendix C

ITS 2004 papers

The role of domain ontology in knowledge acquisition for ITSs

Pramuditha Suraweera, Antonija Mitrovic and Brent Martin

Intelligent Computer Tutoring Group
Department of Computer Science, University of Canterbury
Private Bag 4800, Christchurch, New Zealand
{psu16, tanja, brent}@cosc.canterbury.ac.nz

Abstract. There have been several attempts to automate knowledge acquisition for ITSs that teach procedural tasks. The goal of our project is to automate the acquisition of domain models for constraint-based tutors for both procedural and non-procedural tasks. We propose a three-phase approach: building a domain ontology, acquiring syntactic constraints directly from the ontology, and engaging the author in a dialog, in order to induce semantic constraints using machine learning techniques. An ontology is arguably easier to create than the domain model. Our hypothesis is that the domain ontology is also useful for reflecting on the domain, so would be of great importance for building constraints manually. This paper reports on an experiment performed in order to test this hypothesis. The results show that constraints sets built using a domain ontology are superior, and the authors who developed the ontology before constraints acknowledge the usefulness of an ontology in the knowledge acquisition process.

1 Introduction

Intelligent Tutoring Systems (ITS) are educational programs that assist students in their learning by adaptively providing pedagogical support. Although highly regarded in the research community as effective teaching tools, developing an ITS is a labour intensive and time consuming process. The main cause behind the extreme time and effort requirements is the knowledge acquisition bottleneck [9].

Constraint based modelling (CBM) [10] is a student modelling approach that somewhat eases the knowledge acquisition bottleneck by using a more abstract representation of the domain compared to other common approaches [7]. However, building constraint sets still remains a major challenge. In this paper, we propose an approach to automatic acquisition of domain models for constraint-based tutors. We believe that the domain ontology can be used as a starting point for automatic acquisition of constraints. Furthermore, building an ontology is a reflective task that focuses the author on the important concepts of the domain. Therefore, our hypothesis is that ontologies are also important for developing constraints manually.

To test this hypothesis we conducted an experiment with graduate students enrolled in an ITS course. They were given the task of composing the knowledge base for an ITS for adjectives in the English language. We present an overview of our goals and the results of our evaluation in this paper.

The remainder of the paper is arranged into five sections. The next section presents related work on automatic knowledge acquisition for ITSs, while Section 3 gives an overview of the proposed project. Details of enhancing the authoring shell WETAS are given in Section 4. Section 5 presents the experiment and its results. Conclusions and future work are presented in the final section.

2 Related Work

Research attempts at automatically acquiring knowledge for ITSs have met with limited success. Several authoring systems have been developed so far, such as KnoMic (Knowledge Mimic)[15], Disciple [13, 14] and Demonstr8 [1]. These have focussed on acquiring procedural knowledge only.

KnoMic is a learning-by-observation system for acquiring procedural knowledge in a simulated environment. The system represents domain knowledge as a generic hierarchy, which can be formatted into a number of specific representations, including production rules and decision trees. KnoMic observes the domain expert carrying out tasks within the simulated environment, resulting in a set of observation traces. The expert annotates the points where he/she changed a goal because it was either achieved or abandoned. The system then uses a generalization algorithm to learn the conditions of actions, goals and operators. An evaluation conducted to test the accuracy of the procedural knowledge learnt by KnoMic in an air combat simulator revealed that out of the 140 productions that were created, 101 were fully correct and 29 of the remainder were functionally correct [15]. Although the results are encouraging, KnoMic's applicability is restricted to simulated environments.

Disciple is a shell for developing personal agents. It relies on a semantic network that describes the domain, which can be created by the author or imported from a repository. Initially the shell has to be customised by building a domain-specific interface, which gives the domain expert a natural way of solving problems. Disciple also requires a problem solver to be developed. The knowledge elicitation process is initiated by a proble-solving example provided by the expert. The agent generalises the given example with the assistance of the expert and refines it by learning from experimentation and examples. The learned rules are added to the knowledge base.

Disciple falls short of providing the ability for teachers to build ITSs. The customisation of Disciple requires multiple facets of expertise including knowledge engineering and programming that cannot be expected from a typical domain expert. Furthermore, as Disciple depends on the problem solving instances provided by the domain expert, they should be selected carefully to reflect significant problem states.

Demonstr8 is an authoring tool for building model-tracing tutors for arithmetic. It uses programming by demonstration to reduce the authoring effort. The system provides a drawing tool like interface for building the student interface of the ITS. The system automatically defines each GUI element as a working memory element (WME), while WMEs involving more than a single GUI element must be defined manually. The system generates production rules by observing problems being solved by an expert. Demonstr8 performs an exhaustive search in order to determine the problem-solving procedure used to obtain the solution. If more than one such procedure exists, then the user would have to select the correct one. Domain experts must have significant knowledge of cognitive science and production systems in order to be able to specify higher order WMEs and validate production rules.

3 Automatic constraint acquisition

Existing approaches to knowledge acquisition for ITSs acquire procedural knowledge by recording the expert's actions and generalising recorded traces using machine learning algorithms. Even though these systems are well suited to simulated environments where goals are achieved by performing a set of steps in a specific order, they fail to acquire knowledge for non-procedural domains. Our goal is to develop an authoring system that can acquire procedural as well as declarative knowledge.

The authoring system will be an extension of WETAS [4], a web-based tutoring shell. WETAS provides all the domain-independent components for a text-based ITS, including the user interface, pedagogical module and student modeller. The pedagogical module makes decisions based on the student model regarding problem/feedback generation, whereas the student modeller evaluates student solutions by comparing them to the domain model and updates the student model. The main limitation of WETAS is its lack of support for authoring the domain model.

WETAS is based on Constraint based modelling (CBM), proposed by Ohlsson [10] which is a student modelling approach based on his theory of learning from performance errors [11]. CBM uses constraints to represent the knowledge of the tutoring system [6, 12], which are used to identify errors in the student solution. CBM focuses on correct knowledge rather than describing the student's problem solving procedure as in model tracing [7]. As the space of false knowledge is much grater than correct knowledge, in CBM knowledge is modelled by a set of constraints that identify the set of correct solutions from the set of all possible student inputs. CBM represents knowledge as a set of ordered pairs of relevance and satisfaction conditions. The relevance condition identifies the states in which the constraint is relevant, while the satisfaction condition identifies the subset of the relevant states in which the constraint is satisfied.

Manually composing a constraint set is a labour intensive and time-consuming task. For example, SQL-Tutor contains over 600 constraints, each taking over an hour to produce [5]. Therefore, the task of composing the knowledge base of SQL-Tutor would have taken over 4 months to complete. Since WETAS does not provide any assistance for developing the knowledge base, typically a knowledge base is composed using a text editor. Although the flexibility of a text editor may be powerful for knowledge engineers, novices tend to be overwhelmed by the task.

Our goal is to significantly reduce the time and effort required to generate a set of constraints. We see the process of authoring a knowledge base as consisting of three phases. In the first phase, the author composes the domain ontology. This is an interactive process where the system evaluates certain aspects of the ontology. The expert may choose to update the ontology according to the feedback given by the system.

Once the ontology is complete, the system extracts certain constraints directly from it, such as cardinality restrictions for relationships or domains for attributes. The second stage involves learning from examples. The system learns constraints by generalising the examples provided by the domain expert. If the system finds an anomaly between the ontology and the examples, it alerts the user, who corrects the problem. The final stage involves validating the generated constraints. The system generates examples to be labelled as correct or incorrect by the domain expert. It may also present the constraints in a human readable form, for the domain expert to validate.

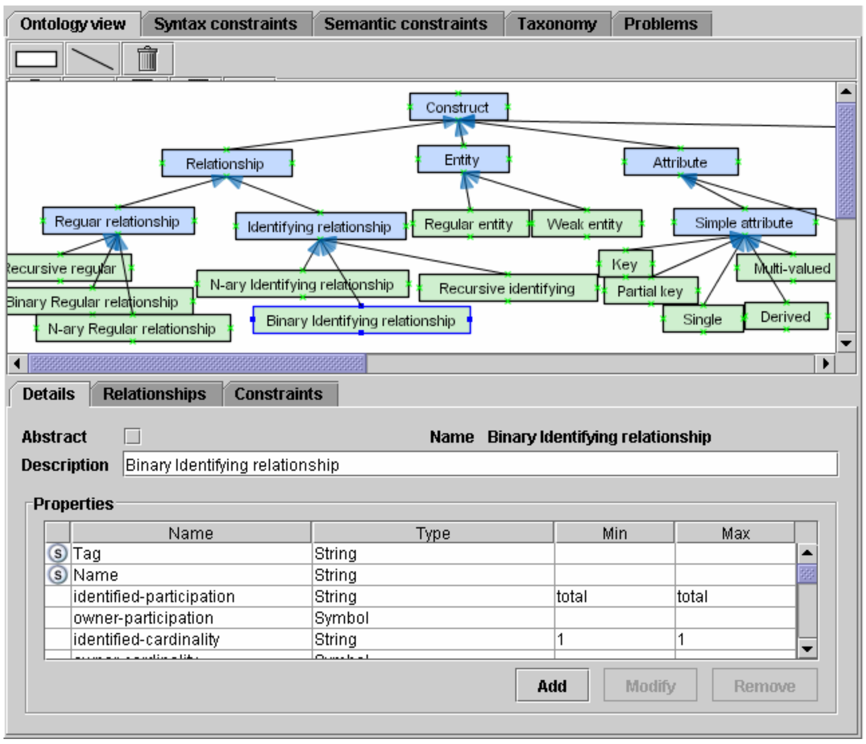
4 Enhancing WETAS: Knowledge Base Generation via Ontologies

We propose that the initial authoring step be the development of a domain ontology, which will later be used to generate constraints automatically. An ontology describes the domain, by identifying all domain concepts and relationships between them. We believe that it is highly beneficial for the author to develop a domain ontology even when the constraint sets is developed manually, because this helps the author to reflect on the domain. Such an activity would enhance the author's understanding of the domain and therefore be a helpful tool when identifying constraints. We also believe that categorising constraints according to the ontology would assist the authoring process.

To test our hypothesis, we built a tool as a front-end for WETAS. Its main purpose is to encourage the use of domain ontology as a means of visualising the domain and organising the knowledge base. The tool supports drawing the ontology, and composing constraints and problems. The ontology front end for WETAS was developed as a Java applet. The interface (**Fig. 1**a) consists of a workspace for developing a domain ontology (*ontology view*) and editors for syntax constraints, semantic constraints, macros and problems. As shown in **Fig. 1**a, concepts are represented as rectangles, and sub-concepts are related to concepts by arrows. The concept details such as attributes and relationships can be specified in the bottom section of the ontology view. The interface also allows the user to view the constraints related to a concept.

The ontology shown in **Fig. 1**a conceptualises the Entity Relationship (ER) data model. *Construct* is the most general concept, which includes *Relationship*, *Entity, Attribute* and *Connector* as sub-concepts. *Relationship* is specialized into *Regular* and *Identifying* ones. *Entity* is also specialized, according to its types, into *Regular* and *Weak* entities. *Attribute* is divided in to two sub-concepts of *Simple* and *Composite* attributes. The details of the *Binary Identifying relationship* concept are depicted in **Fig. 1**. It has several attributes (such as *Nam*e and *Identified-participation*), and three relationships (**Fig. 1**b): *Attributes* (which is inherited from *Relationship*), *Owner*, and *Identified-entity*. The interface allows the specification of restrictions of these relationships in the form of cardinalities. The relationship between *Identifying relationship* and *Regular entity* named *Owner* has a minimum cardinality of 1. The interface also allows the author to display the constraints for each concept (**Fig. 1**c). The constraints can be either directly entered in the ontology view interface or in the syntax/semantic constraints editor.

a



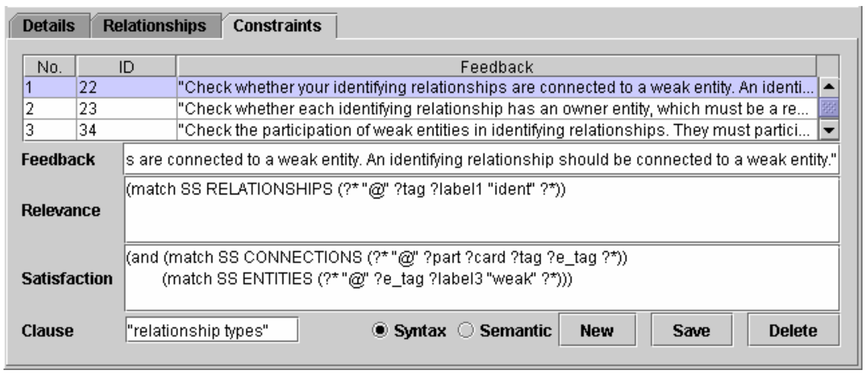
Abstract ☐ **Name** Binary Identifying relationship

Description Binary Identifying relationship

Properties

	Name	Type	Min	Max
S	Tag	String		
S	Name	String		
	identified-participation	String	total	total
	owner-participation	Symbol		
	identified-cardinality	String	1	1

b



No.	ID	Feedback
1	22	"Check whether your identifying relationships are connected to a weak entity. An identi...
2	23	"Check whether each identifying relationship has an owner entity, which must be a re...
3	34	"Check the participation of weak entities in identifying relationships. They must partici...

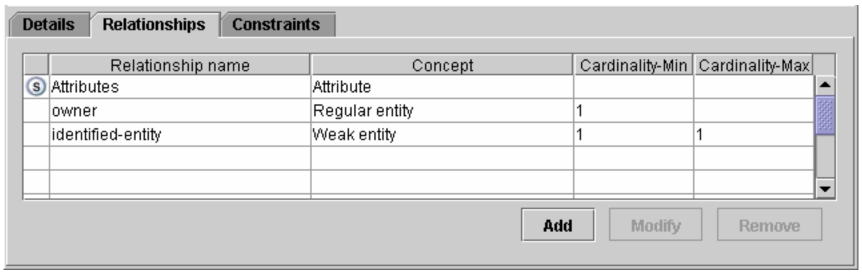
Feedback s are connected to a weak entity. An identifying relationship should be connected to a weak entity."

Relevance (match SS RELATIONSHIPS (?* "@" ?tag ?label1 "ident" ?*))

Satisfaction (and (match SS CONNECTIONS (?* "@" ?part ?card ?tag ?e_tag ?*))
(match SS ENTITIES (?* "@" ?e_tag ?label3 "weak" ?*)))

Clause "relationship types" [✓ Syntax] Semantic

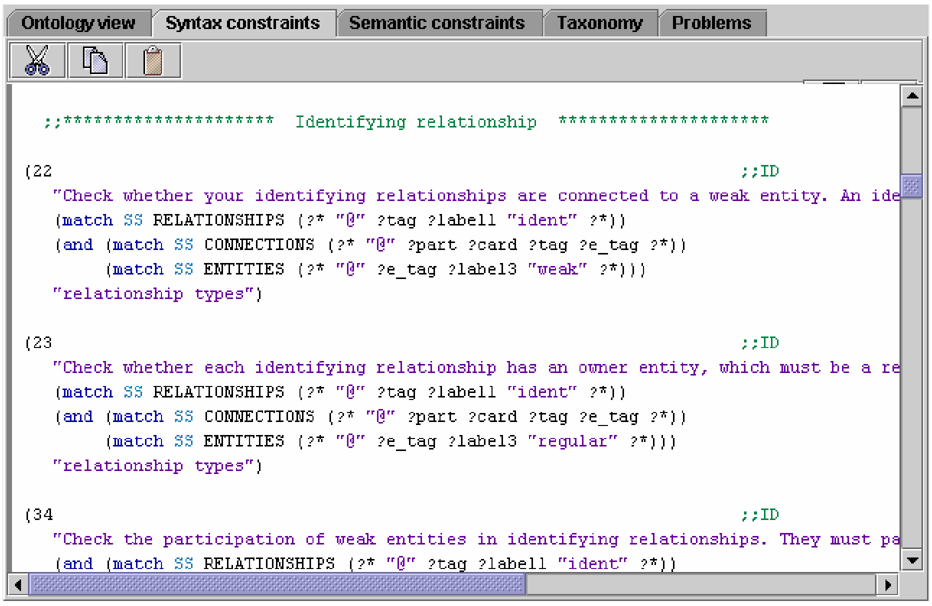
c



	Relationship name	Concept	Cardinality-Min	Cardinality-Max
S	Attributes	Attribute		
	owner	Regular entity	1	
	identified-entity	Weak entity	1	1

Fig. 1. Ontology for ER data model

The constraint editors allow authors to view and edit the entire list of constraints and problems. As shown in **Fig. 2**, the constraints are categorised according to the concepts that they are related to by the use of comments. The Ontology view extracts constraints from the constraint editors and displays them under the categorised concept. **Fig. 2** shows two constraints (Constraint 22 and 23) that belong to *Identifying relationship* concept.



```
  ;;*********************  Identifying relationship  *********************

(22                                                                    ;;ID
   "Check whether your identifying relationships are connected to a weak entity. An ide
   (match SS RELATIONSHIPS (?* "@" ?tag ?label1 "ident" ?*))
   (and (match SS CONNECTIONS (?* "@" ?part ?card ?tag ?e_tag ?*))
        (match SS ENTITIES (?* "@" ?e_tag ?label3 "weak" ?*)))
   "relationship types")

(23                                                                    ;;ID
   "Check whether each identifying relationship has an owner entity, which must be a re
   (match SS RELATIONSHIPS (?* "@" ?tag ?label1 "ident" ?*))
   (and (match SS CONNECTIONS (?* "@" ?part ?card ?tag ?e_tag ?*))
        (match SS ENTITIES (?* "@" ?e_tag ?label3 "regular" ?*)))
   "relationship types")

(34                                                                    ;;ID
   "Check the participation of weak entities in identifying relationships. They must pa
   (and (match SS RELATIONSHIPS (?* "@" ?tag ?label1 "ident" ?*))
```

Fig. 2. Syntax constraints editor

All domain related information is saved on the server as required by WETAS. The applet monitors all significant events in the ontology view and logs them with their time stamps. The logged events include log in/out, adding/deleting concepts etc.

5 Experiment

We hypothesized that composing the ontology and organising the constraints according to its concepts would assist in the task of building a constraint set manually. To evaluate our hypothesis, we set 18 students enrolled in the 2003 graduate course on Intelligent Tutoring Systems at the University of Canterbury the task of building a tutor using WETAS for adjectives in the English language.

The students had attended 13 lectures on ITS, including five on CBM, before the experiment. They also had a 50 minute presentation on WETAS, and were given a description of the task, instructions on how to write constraints, and the section on adjectives from a text book for English vocabulary [2]. The students had three weeks to implement the tutor. A typical problem is to complete a sentence by providing the correct form of a given adjective. An example sentence the students were given was: "My sister is much ________ than me (wise)."

The students were also free to explore LBITS [3], a tutor developed in WETAS that teaches simple vocabulary skills. The students were allowed to access the "last two letters" puzzles, where the task involved determining a set of words that satisfied the clues, with the first two letters of each word being the same as the last two letters of the previous one. All domain specific components, including its ontology, the constraints and problems, were available.

Seventeen students completed the task satisfactorily. One student lost his entire work due to a system bug, and this student's data was not included in the analysis. The same bug did not affect other students, since it was eliminated before others experienced it. Table 1 gives some statistics about the remaining students, including their interaction times, numbers of constraints and the marks for constraints and ontology.

The participants took 37 hours to complete the task, spending 12% of the time in the ontology view. The time in the ontology view varied widely, with a minimum of 1.2 and maximum of 7.2 hours. This can be attributed to different styles of developing the ontology. Some students may have developed the ontology on paper before using the system, whereas others developed the whole ontology online. Furthermore, some students also used the ontology view to add constraints. However, the logs showed that this was not a popular option, as most students composed constraints in the constraint editors. One factor that contributed to this behaviour may be the restrictiveness of the constraint interface, which displays only a single constraint at a time.

WETAS distinguishes between semantic and syntactic constraints. In the domain of adjectives, it is not clear as to which category the constraints belong. For example, in order to determine whether a solution is correct, it is necessary to check whether the correct rule has been applied (semantics) and whether the resulting word is spelt correctly (syntax). This is evident in the results for the total number of constraints for each category. The averages of both categories are similar (9 semantic constraints and 11 syntax constraints). Some participants have included most of their constraints as semantic and others vice versa. Students on average composed 20 constraints in total.

We compared the participants' solution to the "ideal" solution. The marks for these two aspects are given under *Coverage* (the last two columns in Table 1). The ideal knowledge base consists of 20 constraints. The *Constraints* column gives the number of the ideal constraints that are accounted for in the participants' constraint sets. Note that the mapping between the ideal and participants' constraints is not necessarily 1:1. Two participants accounted for all 20 constraints. On average, the participants covered 15 constraints. The quality of constraints was high generally.

The ontologies produced by the participants were given a mark out of five (the *Ontology* column in Table 1). All students scored high, as expected because the ontology was straightforward. Almost every participant specified a separate concept for each group of adjectives according to the given rules [2]. However, some students constructed a flat ontology, which contained only the six groupings corresponding to the rules (see Fig. 3a). Five students scored full marks for the ontology by including the degree (comparative or superlative) and syntax such as spelling (see Fig. 3b).

Even though the participants were only given a brief description of ontologies and the example ontology of LBITS, they created ontologies of a reasonable standard. However, we cannot make any general assumptions on the difficulty of constructing ontologies since the domain of adjectives is very simple. Furthermore, the six rules for determining the comparative and superlative degree of an adjective gave strong hints on what concepts should be modelled.

	Time (hours)		Number of constraints			Coverage	
	Total	Ontology view	Se-mantic	Syntax	Total	Con-straints	Ontology
S1	38.16	4.57	27	3	30	20	5
S2	51.55	7.01	3	10	13	19	4
S3	10.22	1.20	14	1	15	17	4
S4	45.25	2.54	30	4	34	18	5
S5	48.96	4.91	11	5	16	20	4
S6	44.89	4.66	24	1	25	18	5
S7	18.97	2.87	1	15	16	17	4
S8	22.94	4.99	3	18	21	15	3
S9	34.29	4.30	11	4	15	18	5
S10	33.90	7.23	0	14	14	18	3
S11	55.76	3.28	16	1	17	17	5
S12	30.46	2.84	0	16	16	10	3
S13	60.94	3.47	1	15	16	13	3
S14	32.42	1.96	1	17	18	12	3
S15	33.35	4.04	1	14	15	11	3
S16	29.60	6.24	0	30	30	4	5
Mean	36.98	4.13	8.94	10.50	19.44	15.44	4.00
S.D.	13.66	1.72	10.47	8.23	6.60	4.37	0.89

Table 1. Results

Fourteen participants categorised their constraints according to the concepts of the ontology as shown in **Fig. 2**. For these participants, there was a significant correlation between the ontology score and the constraints score (0.679, $p<0.01$). However, there was no significant correlation between the ontology score and the constraints score when all participants were considered. This strongly suggests that the participants used the ontology to write constraints developed better constraints.

An obvious reason for this finding may be that more able students produced better ontologies and also produced a complete set of constraints. To test this hypothesis, we determined the correlation between the participant's final grade for the course (which included other assignments) and the ontology/constraint scores. There was indeed a strong correlation (0.840, $p<0.01$) between the grade and the constraint score. However, there was no significant correlation between the grade and the ontology score. This lack of a relationship can be due to a number of factors. Since the task of building ontologies was novel for the participants, they may have found it interesting and performed well regardless of their ability. Another factor is that the participants had more practise at writing constraints (in other assignments for the same course) than on ontologies. Finally, the simplicity of the domain could also be a contributing factor.

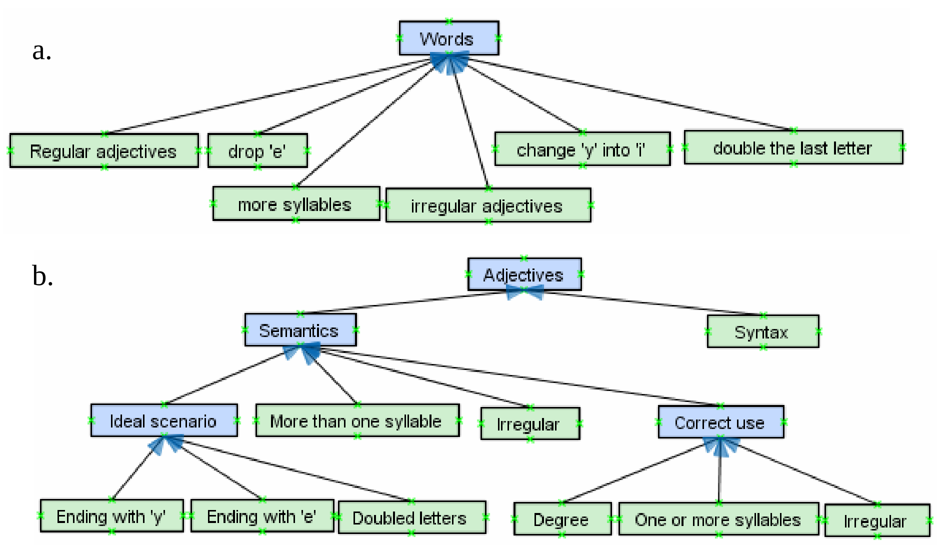


Fig. 3. Ontologies constructed by students

The participants spent 2 hours per constraint (sd=1 hour). This is twice the time reported in [8], but the participants are neither knowledge engineers nor domain experts, so the difference is understandable. The participants felt that building an ontology made constraint identification easier. The following comments were extracted from their reports: "*Ontology helped me organise my thinking;*" "*The ontology made me easily define the basic structure of this tutor;*" "*The constraints were constructed based on the ontology design;*" "*Ontology was designed first so that it provides a guideline for the tasks ahead.*"

The results indicate that ontologies do assist constraint acquisition: there is a strong correlation between the ontology score and the constraints score for the participants who organised the constraints according to the ontology. Subjective reports confirmed that the ontology was used as a starting point when writing constraints. As expected, more able students produced better constraints. In contrast, most participants composed good ontologies, regardless of their ability.

6 Conclusions

We performed an experiment to determine whether the use of domain ontologies would assist manual composition of constraints for constraint-based ITSs. The

WETAS authoring shell was enhanced with a tool that allowed users to define a domain ontology and use it as the basis for organizing constraints. We showed that constructing a domain ontology indeed assisted the creation of constraints. Ontologies enable authors to visualise the constraint set and to reflect on the domain, assisting them to create more complete constraint bases.

We intend to enhance WETAS further by automating constraint acquisition. Preliminary results show that many constraints can be induced directly from the domain ontology. We will also be exploring ways of using machine learning algorithms to automate constraint acquisition from dialogs with domain experts.

Acknowledgements The work reported here has been supported by the University of Canterbury Grant U6532.

References

1. Blessing, S.B.: A Programming by Demonstration Authoring Tool for Model-Tracing Tutors. Artificial Intelligence in Education, 8 (1997) 233-261
2. Clutterbuck, P.M.: The art of teaching spelling: a ready reference and classroom active resource for Australian primary schools. Longman Australia Pty Ltd, Melbourne, 1990.
3. Martin, B., Mitrovic, A.: Authoring Web-Based Tutoring Systems with WETAS. In: Kinshuk, Lewis, R., Akahori, K., Kemp, R., Okamoto, T., Henderson, L. and Lee, C.-H. (eds.) Proc. ICCE 2002 (2002) 183-187
4. Martin, B., Mitrovic, A.: WETAS: a Web-Based Authoring System for Constraint-Based ITS. Proc. 2nd Int. Conf on Adaptive Hypermedia and Adaptive Web-based Systems AH 2002, Springer-Verlag, Berlin Heidelberg New York, pp. 543-546, 2002.
5. Mitrovic, A.: Experiences in Implementing Constraint-Based Modelling in SQL-Tutor. In: Goettl, B.P., Halff, H.M., Redfield, C.L. and Shute, V.J. (eds.) Proc. 4th Int. Conf. on Intelligent Tutoring Systems, San Antonio, (1998) 414-423
6. Mitrovic, A.: An intelligent SQL tutor on the Web. Artificial Intelligence in Education, 13, (2003) 171-195
7. Mitrovic, A., Koedinger, K. Martin, B.: A comparative analysis of cognitive tutoring and constraint-based modeling. In: Brusilovsky, P., Corbett, A. and Rosis, F.d. (eds.) Proc. UM2003, Pittsburgh, USA, Springer-Verlag, Berlin Heidelberg New York (2003) 313-322
8. Mitrovic, A., Ohlsson, S.: Evaluation of a Constraint-based Tutor for a Database Language. Artificial Intelligence in Education , 10(3-4) (1999) 238-256
9. Murray, T.: Expanding the Knowledge Acquisition Bottleneck for Intelligent Tutoring Systems. Artificial Intelligence in Education, 8 (1997) 222-232
10. Ohlsson, S.: Constraint-based Student Modelling. Proc. Student Modelling: the Key to Individualized Knowledge-based Instruction, Springer-Verlag (1994) 167-189
11. Ohlsson, S.: Learning from Performance Errors. Psychological Review, 103 (1996) 241-262
12. Suraweera, P., Mitrovic, A.: KERMIT: a Constraint-based Tutor for Database Modeling. In: Cerri, S., Gouarderes, G. and Paraguacu, F. (eds.) Proc. 6th Int. Conf on Intelligent Tutoring Systems ITS 2002, Biarritz, France, LCNS 2363 (2002) 377-387
13. Tecuci, G.: Building Intelligent Agents: An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies. Academic press, 1998.
14. Tecuci, G., Keeling, H.: Developing an Intelligent Educational Agent with Disciple. Artificial Intelligence in Education, 10 (1999) 221-237
15. van Lent, M., Laird, J.E.: Learning Procedural Knowledge through Observation. Proc. Int. Conf. on Knowledge Capture, (2001) 179-186

The use of ontologies in ITS domain knowledge authoring

Pramuditha Suraweera, Antonija Mitrovic and Brent Martin

Intelligent Computer Tutoring Group
Department of Computer Science, University of Canterbury
Private Bag 4800, Christchurch, New Zealand
{psu16,tanja,brent}@cosc.canterbury.ac.nz

Abstract. Acquiring the domain knowledge is a task that requires a major portion of the time and effort when building an ITS. Researchers have been exploring ways of automating the knowledge acquisition process since the inception of ITSs with limited success. All past research attempts have focussed on acquiring knowledge for procedural domains. Our goal is to develop an authoring system that acquires knowledge for procedural as well as nonprocedural domains. We propose a four phase approach: composing an ontology of the domain, extracting syntax constraint from it, learning semantic constraints from the examples provided by the domain expert and finally verifying the generated constraints. This paper presents an overview of the knowledge acquisition system for acquiring knowledge for constraint-based tutors. It mainly focuses on composing the ontology and acquiring syntax constraints from it. Further work on this project will focus on learning from examples and validating the generated constraints.

Keywords: intelligent tutoring systems, authoring systems, constraint-based modelling, domain model authoring

1 Introduction

Acquiring domain knowledge is a major hurdle in building Intelligent Tutoring Systems (ITS) [1]. Although there have been several attempts to ease the burden on ITS developers by automating the process, they have met with limited success. All previous attempts have focussed on acquiring knowledge required for teaching procedural tasks. Our goal is to drastically reduce the time and effort required for acquiring domain knowledge by automating knowledge acquisition for intelligent tutors for both procedural and nonprocedural domains.

Constraint based modelling (CBM) [2] is a student modelling approach that somewhat eases the knowledge acquisition bottleneck by using a more abstract representation of the domain compared to other commonly used approaches [3]. CBM is based on Ohlsson's theory of learning from performance errors [4]. It focuses on correct knowledge rather than describing the student exactly as with model tracing. However, building a complete constraint base still remains a major challenge. Mitrovic reported that she took just over an hour to produce a constraint for SQL-Tutor [5, 13], which currently contains more than 650 constraints. Therefore, the task of composing the knowledge base of SQL-Tutor would have taken over 4 months to complete. Our goal is to dramatically reduce the time and effort required for composing the knowledge base required for constraint-based tutors by automating the knowledge acquisition process.

We envisage ontologies to play a central role in the whole knowledge acquisition process. A preliminary study conducted to evaluate the role of ontologies in manually composing a constraint base showed that constructing a domain ontology indeed assisted the composition of constraints [6]. The study showed that ontologies can be used to organise the constraint base into meaningful categories. This enabled the author to visualise the constraint set and to reflect on the domain assisting them to create more complete constraint bases.

The remainder of the paper is organised into five sections. The next section presents a brief description of automatic knowledge acquisition systems. Section 3 gives an overview of our project. Details on developing the ontology are given in Section 4. Section 5 discusses the process of acquiring syntax constraints from the ontology. Conclusions and future work are presented in the final section.

2 Related Work

Past research on acquiring knowledge for ITSs have solely focused on acquiring knowledge for teaching procedural tasks such as tasks in simulated environments and solving mathematical algebraic problems. The knowledge acquisition systems that acquire domain knowledge as a runnable model for evaluating student solutions include KnoMic [7], Disciple [8, 9] and Demonstr8 [10]. All these systems acquire knowledge by observing the domain expert performing a task and generalising it to be applicable for other problems.

KnoMic is a learning-by-observation system for acquiring procedural knowledge in a simulated environment. The system observes and records the procedure taken by the domain expert in performing a task within the simulated environment. While performing the task the expert has to annotate the points where he/she had changed goals because it was either achieved or abandoned. The resulting set of observation traces are generalised by the system to learn the conditions of actions, goals and operators. During an evaluation to test the accuracy of the procedural knowledge learnt in an air combat simulator, KnoMic acquired 140 productions. Out of the total 140 created, 101 were fully correct and 29 of the remainder were functionally correct [7]. Although the results are encouraging KnoMic's applicability is limited to only simulated environments.

Disciple is a shell for developing personal agents. It relies on a semantic network of the domain that describes the domain, which can be either composed by the author or imported from a repository. Initially the shell has to be customised to the domain by building a domain-specific interface, which gives a natural way of solving problems for the domain expert. Disciple also requires a problem solver for the domain. The domain expert has to initiate the knowledge elicitation process by providing problem-solving examples. The agent generalises the provided example using a generalisation algorithm with the assistance of the domain expert. The generalised example is refined by requesting the expert to validate the examples generated by the system. As Disciple depends on problem solving instances provided by the domain expert, they should be carefully selected to reflect significant problem states. The task of selecting significant problem states requires expertise in knowledge engineering which is scarce. Furthermore, building a problem solver for some domains is extremely difficult, if not impossible.

Demonstr8 is an authoring tool for building model-tracing tutors for arithmetic. It relies on the domain expert to specify all the algebraic functions that can be used and their outcomes in the form of a table. It uses programming by demonstration to reduce the authoring effort. The system provides a drawing tool like interface for building the student interface of the ITS. The system automatically defines each GUI element as a working memory element (WME), while WMEs involving more than a single GUI element must be defined manually. The system generates production rules by observing problems being solved by an expert. Demonstr8 performs an exhaustive search in order to determine the problem-solving procedure used to obtain the solution. If more than one such procedure exists, then the user would have to select the correct one.

3 Automatic Constraint Acquisition

Existing approaches to knowledge acquisition for ITSs acquire procedural knowledge by recording the domain expert's actions and generalising recorded traces using machine learning algorithms. Even though these systems are well suited to simulated environments where goals are achieved by performing a set of steps in a specific order, they fail to acquire knowledge for non-procedural domains. Our goal is to develop an authoring system that can acquire procedural as well as declarative knowledge.

The authoring system will be an extension of WETAS [11], a web-based tutoring shell that facilitates building constraint-based tutors. WETAS provides all the domain-independent components for a text-based ITS, including the user interface, pedagogical module and student modeller. The pedagogical module makes decisions based on the student model regarding problem/feedback generation and the student modeller evaluates student solutions by comparing them to the domain model and updates the student model. The main limitation of WETAS is its lack of support for authoring the domain model.

The domain model for CBM tutors [14, 15] consists of a set of constraints, which are used to identify errors in student solutions. As the space of false knowledge is much grater than correct knowledge, in CBM knowledge is modelled by a set of constraints that identify the set of correct solutions from the set of all possible student inputs. CBM represents knowledge as a set of ordered pairs of relevance and satisfaction conditions. The relevance condition identifies the states in which the constraint is relevant, while the satisfaction condition identifies the subset of the relevant states in which the constraint is satisfied.

As WETAS does not provide any assistance for developing the knowledge base, typically a knowledge base is composed using a text editor. Although the flexibility of a text editor may be adequate for knowledge engineers, novices tend to be overwhelmed by the task. Our goal is to reduce the time and effort required for building a constraint base by adding support for automatic constraint acquisition to WETAS. We propose a four-stage process initiated by modelling the domain as an ontology. The ontology would be composed by a domain expert using the ontology modelling tool. Once the ontology is completed, the system would analyse the ontology and extract syntax constraints directly from the completed ontology. During the third phase, the system would acquire constraints by analysing sample solutions provided by the expert. Finally the constraint set is validated with the assistance of the domain expert, where the expert would label the system generated examples as correct or incorrect.

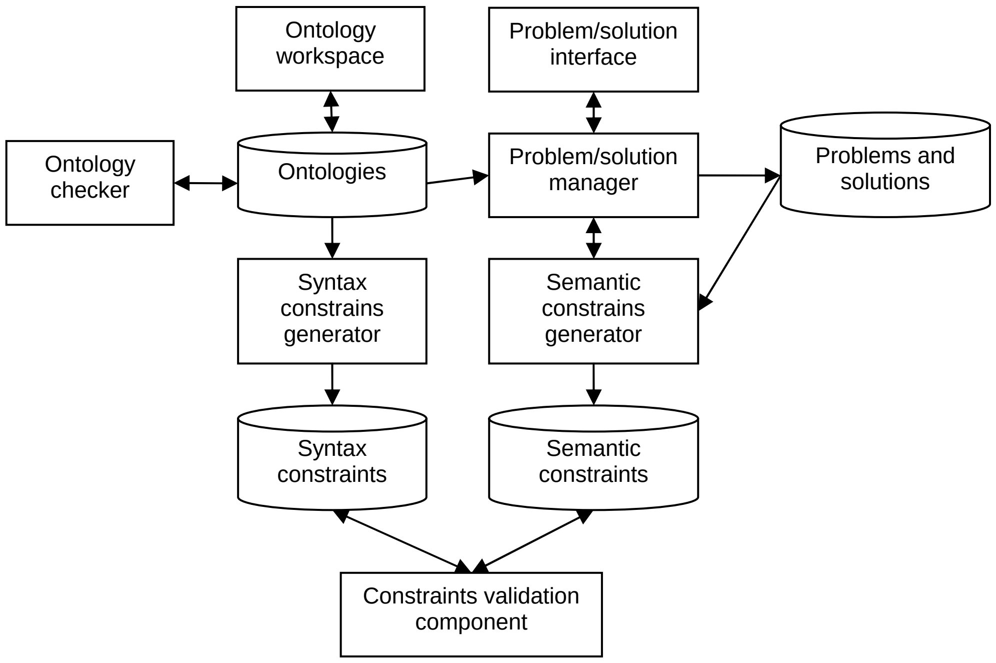


Figure 1 Architecture of the constraint-acquisition system

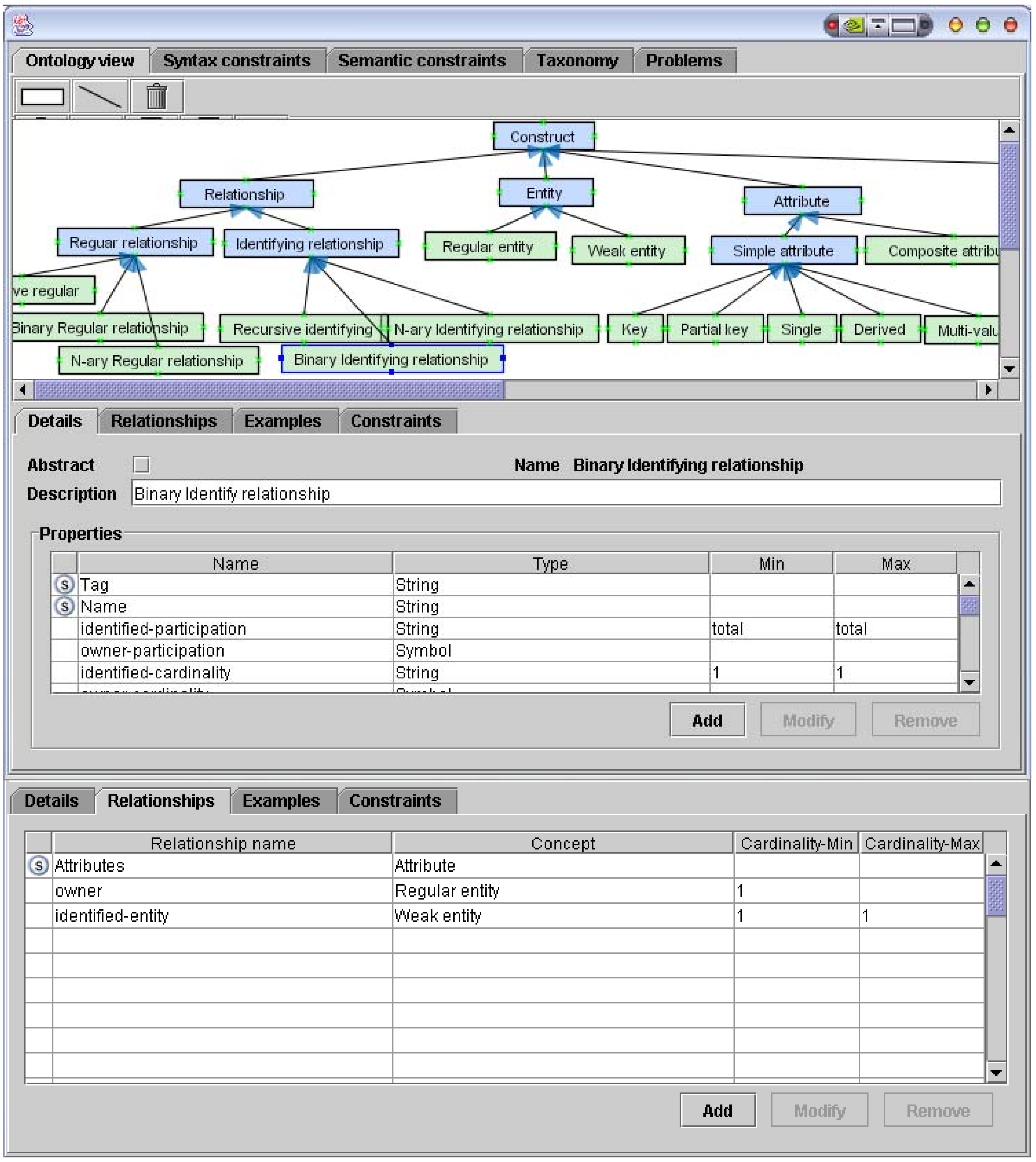
The architecture of the constraint acquisition system consists of an ontology workspace, ontology checker, problem/solution manager and syntax and semantic constraint generators, as depicted in Figure 1. During the initial phase, the domain expert models an ontology of the domain in the ontology workspace. The ontology checker validates the ontology during the ontology composition state. The completed ontology is stored in the ontology repository. The syntax constraints generator analyses the completed ontology and generates syntax constraints from it. The generated syntax constraints are stored in the syntax constraints repository. The generation of constraints from a domain ontology is discussed further in Section 5.

The domain expert has to specify the representation for solutions prior to entering problems and sample solutions. The solution representation is a decomposition of the solution into components consisting of a list of instances of concepts. For example, a sentence in English consists of a list of words and a list of punctuation marks.

The domain expert has to enter sample problems and their solutions during the third phase of knowledge acquisition. The problems/solution interface assists the user by providing a dynamic form that consists of input boxes for populating each property of the concept instance. The expert is requested to provide different solutions that depict different ways of solving the same problem. While the expert enters in an alternative correct solution, the system attempts to match each component of the solution to components of the initial solution. These matches are later used to compose a set of semantic constraints that compare the student's solution against the system's ideal solution. The expert is also encouraged to supply solutions containing typical errors made by students. The system would use these erroneous solutions to provide more detailed assistance. The system also verifies the solutions provided by the expert using the generated syntax constraints. If a discrepancy is identified, the user is alerted and the solution may be modified to comply with the ontology or vice versa.

The final phase involves ensuring the validity of constraints. During this phase the system would generate examples for the domain to be validated by the author. In situations where the author's validation conflicts with the system's evaluation according to the domain model, the system would request the author to provide further examples to illustrate the rationale behind the conflict. The system would use the new examples to resolve the conflicts and may also generate new constraints. The author may also wish to examine the English description of the generated constraints and dispute them by providing counter examples.

We started developing the constraint acquisition system in 2003. The ontology workspace, ontology checker, problem/solution interface and the syntax constraint generator are completed. We are currently working on the semantic constraints generator.



Abstract ☐ Name Binary Identifying relationship

Description Binary Identify relationship

Properties

	Name	Type	Min	Max
S	Tag	String		
S	Name	String		
	identified-participation	String	total	total
	owner-participation	Symbol		
	identified-cardinality	String	1	1

	Relationship name	Concept	Cardinality-Min	Cardinality-Max
S	Attributes	Attribute		
	owner	Regular entity	1	
	identified-entity	Weak entity	1	1

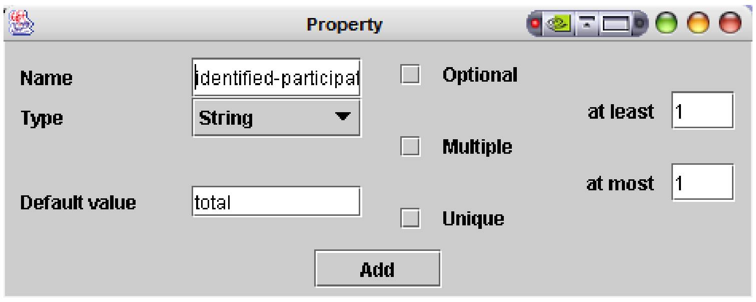
Figure 2 Ontology workspace interface

4 Developing the ontology

As discussed earlier, the first phase in authoring a constraint base is developing the domain ontology. The ontology will be later used to generate constraints automatically. An ontology describes the domain, by identifying all the important domain concepts and various relationships between them. The ontology workspace provides an environment for composing the domain ontology in terms of concepts and their sub concepts as shown in Figure 2. All concepts are represented using rectangles and they are related to their sub concepts using arrows. The interface has no restrictions in placing concepts within the workspace. The user can position the concepts to display a hierarchical structure. The completed ontology is saved on a central server in the XML format.

The ontology displayed in Figure 2 represents the concepts of ER modelling, a popular database modelling technique. The ER model describes data as entities, attributes and relationships. An entity is the basic object represented in the ER model, which is a 'thing' in the real world with an independent existence. Each entity has particular properties, called *attributes*, that describe it. A relationship is an association between two or more entities.

The ER ontology depicted in Figure 2 contains *Construct* as the most general concept. *Relationship*, *Entity*, *Attribute* are sub-concepts of *Construct*. *Relationship* is specialised into *Regular* and *Identifying*, which are the two types of relationships and *Entity* is specialised, according to its types, as *Regular* and *Weak*. Subclasses of *Attribute* are *Simple* or *Composite* attributes and *Simple* attributes are further specialised into five categories: *Key*, *Partial key*, *Single*, *Derived* and *Multi-valued*.



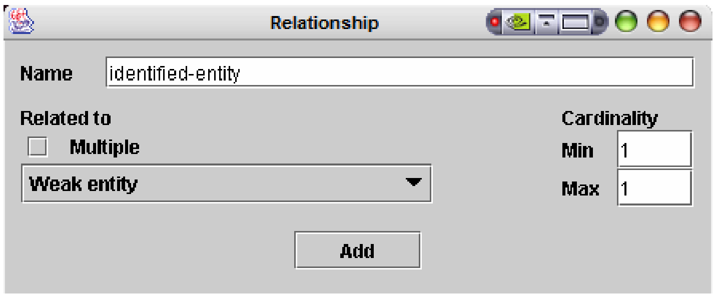
Name	identified-participat	☐ Optional			
Type	String			at least	1
		☐ Multiple			
				at most	1
Default value	total	☐ Unique			

Figure 3 Details of *identified-participation* property

Each concept has a set of properties that describes it. To define properties for a concept, the author uses the property addition interface shown in Figure 3. The range of values that the property may hold can be specified in terms of minimum and maximum values or as a set of distinct values. Other restrictions include specifying that the value of a property is unique, optional or can contain multiple values. Figure 3 depicts the *identified participation* property of the *Binary identifying relationship* concept. The property has a default value of 'total'. Furthermore as the 'at least' and 'at most' fields are both set to 1 the *identified participation* property has to have a single value.

The remainder of the properties of *Binary identifying relationship* concept, as shown in Figure 2, include *name, owner participation* and *identified cardinality*. Most properties are of type 'string' except *owner participation* and *owner cardinality* are of type 'symbol'. Both *identified cardinality* property and *identified participation* property have default values: 1 and 'total' respectively.

The relationships that are involved with *Binary identifying relationship* concept are detailed in Figure 2. They include *attributes*, *owner* and *identified entity*. The *attributes* relationship is a relationship between *Binary identifying relationship* and *Attribute* with no restrictions on the cardinality. The *owner* relationship with *Regular entity* has a minimum cardinality of 1. The *identified entity* relationship, as detailed in Figure 4, between *Binary identifying relationship* and *Weak entity* has a minimum and maximum cardinality of 1.

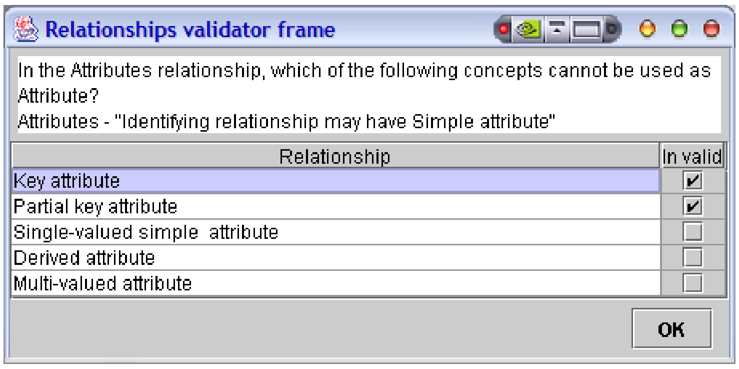


Relationship

Name	identified-entity
Related to	
Multiple	☐
	Weak entity
Cardinality	
Min	1
Max	1

Figure 4 Details of *identified-entity* relationship

During the task of composing an ontology, the domain expert may add relationships that are too general. Since constraints are composed directly from the relationships found in the ontology, it is imperative that the relationships are valid. In order to ensure that all added relationships are completely accurate, the system engages the author in a dialog. During this dialog the author is presented with lists of specialisations of concepts involved in the relationship and is asked to label the specialisations that violate the principles of the domain. As an example, consider the relationship between *Binary identifying relationship* and *Attribute*. As shown in Figure 5, the initial question posed asks whether each of the specialisations of *attribute* (*key*, *partial key*, *single-valued* etc) are applicable to the *attributes* relationship. The user would indicate that *key* or *partial key* attributes cannot be used in the *attributes* relationship. The system replaces the original relationship with a more specific one at the completion of the dialog.



Relationships validator frame

In the Attributes relationship, which of the following concepts cannot be used as Attribute?
Attributes - "Identifying relationship may have Simple attribute"

Relationship	In valid
Key attribute	✓
Partial key attribute	✓
Single-valued simple attribute	☐
Derived attribute	☐
Multi-valued attribute	☐

Figure 5 Relationship validate dialog for ‘Entity has attribute’ relationship

The ontology is represented in memory as a list of objects that keeps track of all the details about the concepts. The concept’s properties and relationships are contained as lists within each concept object. The concept object keeps a record of its super concepts and sub concepts. The generated constraints that belong to each concept are also stored in a list with the concept object.

The internal representation of the ontology gets converted to XML for storing in the central server. The XML representation uses a set of XML tags defined specifically for this project. Details of each concept along with a unique id that identifies the concept are enlisted using the appropriate XML tags. The concept ids are used to specify the relationships between concepts. The position and sizes of the graphical representations for the concepts are also recorded in XML in order to recreate an identical ontology in the ontology workspace. During the restoration of the ontology new objects for representing concepts are created with the relevant information extracted from the XML representation.

Although the ontology is stored using proprietary XML tags, it can be easily be transformed to a standard ontology representation form such as DAML [16]. The system was developed to save ontologies using its own XML representation in order to speed up the progress of the research project, avoiding the need for extensive research into DAML implementation.

5 Acquiring Syntax Constraints from the Domain Ontology

An ontology contains a lot of information about the domain and is much easier to create than the final domain model. Our goal is to extract the useful syntactic information from an ontology to generate syntax constraints for the domain model. This process involves analysing the relationships between concepts and the properties of concepts that exist in the ontology.

Initially the constraint generator extracts all the relationships between concepts. Each relationship with restrictions on the cardinality such as a minimum or maximum yields a syntax constraint that restricts the instances participating in the particular relationship. As an example, consider the *identified-entity* relationship found in Figure 4. It generates a constraint which says *Binary identifying relationship* must have exactly 1 *Weak entity* as the *identified entity*. The relevance condition of the constraint focuses on identifying instances of *Binary identifying relationships*, whereas the satisfaction condition specifies that each of them has to have exactly one weak entity as the *identified-entity*.

The domain and range of each concept's properties are also analysed for generating constraints. The constraint generator creates a constraint for each restriction on the domain and range of a property. Such restrictions involve minimum and maximum values allowed, whether the property is required, multivalued or unique. The generated constraints are similar to the constraints generated from relationships, having a check for identifying each property as a relevance condition and a satisfaction condition that ensures that the specified condition is met.

For example, when the processing of the *Binary identifying relationship* concept illustrated in Figure 2, 4 the system generates six constraints:

- *Binary identifying relationship* must have at least 1 *Regular entity* as the owner
- *Binary identifying relationship* must have exactly 1 *Weak entity* as the identified entity
- The *identified participation* property of *Binary identifying relationship* must be total
- The *identified cardinality* property of *Binary identifying relationship* must be 1
- The *name* property of *Relationship* type has to be unique
- *Relationship* type must have exactly 1 name

The dialog sessions for validating relationships during the ontology composing phase also contribute towards generating syntax constraints. The specialisations of concepts involved in the relationship that violate the principles of the domain are used to generate constraints. They ensure that elements of a solution does not participate in such relationships that violate the domain principles. The specialisations marked as violations by the author in Figure 5 would be used to generate two constraints: *Binary identifying relationship* cannot have a *key* attribute and *Binary identifying relationship* cannot have a *partial key* attribute.

The syntax constraints generator produced a total of 48 syntax constraints from the ER ontology depicted in Figure 2. The generated set of constraints covered all syntax constraints that existed in KERMIT [12], a constraint-based tutor for ER modelling. Although the initial results are derived from only a single domain, we believe that the system would be able to successfully handle most non-procedural domains. The ontology workspace would be enhanced to handle procedural domains by adding further constructs.

6 Conclusions and Future Work

We provided a brief overview of our main research objective: automatically acquire domain knowledge required for constraint-based tutors. We propose a four phase process, initiated by modelling a domain ontology. The system then analyses the completed ontology and extracts syntax constraints from it. During the third phase, the author provides example problems and their solutions and the system generates semantic constraints by analysing the solutions. Finally, the induced constraint set is validated with the assistance of the author.

The paper included a detailed description of the first two phases: modelling the ontology and extracting constraints from it. The initial tests conducted on acquiring constraints from an ontology composed for ER modelling produced encouraging results. The system generated the complete set of syntax constraints found in KERMIT, a constraint based tutor developed for the same domain.

Currently we are working on acquiring semantic constraints from examples provided by the domain expert. We will be exploring machine learning algorithms such as learning from examples and learning from analogy for automatically acquiring semantic constraints. The ontology workspace will also be enhanced to handle procedural domains.

Finally the system will be thoroughly evaluated to test its effectiveness. Most importantly, the quality and the correctness of the knowledge base generated by the system have to be evaluated. Since this research aims to produce a system that is capable of acquiring knowledge for a vast range of tasks, it will be tested in different domains. The usability of the system will also be tested.

Acknowledgements

The work reported here has been supported by the University of Canterbury Grant U6532.

References

1. Murray, T.: Expanding the knowledge acquisition bottleneck for intelligent tutoring systems. Int. J. Artificial Intelligence in Education 8 (1997) 222–232
2. Ohlsson, S.: Constraint-based student modelling. In: Student Modelling: the Key to Individualized Knowledge-based Instruction, Berlin, Springer-Verlag (1994) 167– 189
3. Mitrovic, A., Koedinger, K., Martin, B.: A comparative analysis of cognitive tutoring and constraint-based modelling. In Brusilovsky, P., Corbett, A., Rosis, F.d., eds.: UM2003, Pittsburgh, USA, Springer-Verlag (2003) 313–322
4. Ohlsson, S.: Learning from performance errors. Psychological Review 103 (1996) 241–262
5. Mitrovic, A.: Experiences in implementing constraint-based modelling in SQL-tutor. In Goettl, B.P., Halff, H.M., Redfield, C.L., Shute, V.J., eds.: 4th International Conference on Intelligent Tutoring Systems, ITS 98, San Antonio (1998) 414–423
6. Suraweera, P., Mitrovic, A., Martin, B.: The role of domain ontology in knowledge acquisition for ITSs. In: 7th International Conference on Intelligent Tutoring Systems, ITS 2004. (2004) to appear
7. van Lent, M., Laird, J.E.: Learning procedural knowledge through observation. In: International conference on Knowledge capture, Victoria, British Columbia, Canada, ACM Press (2001) 179–186
8. Tecuci, G.: Building Intelligent Agents: An Apprenticeship Multi-strategy Learning Theory, Methodology, Tool and Case Studies. Academic press (1998) 9. Tecuci, G., Keeling, H.: Developing an intelligent educational agent with disciple. International Journal of Artificial Intelligence in Education 10 (1999) 221–237
10. Blessing, S.B.: A programming by demonstration authoring tool for model-tracing tutors. Int. J. Artificial Intelligence in Education 8 (1997) 233–261
11. Martin, B., Mitrovic, A.: Domain modelling: Art or science? In U. Hoppe, F.V..J.K., ed.: Artificial Intelligence in Education, AIED 2003, IOS Press (2003) 183–190
12. Suraweera, P., Mitrovic, A.: Kermit: a constraint-based tutor for database modelling. In Cerri, S., Gouarderes, G., Paraguacu, F., eds.: 6th International Conference on Intelligent Tutoring Systems, ITS 2002, Biarritz, France (2002) 377–387
13. Mitrovic, A. An Intelligent SQL Tutor on the Web. Int. J. Artificial Intelligence in Education, v13no2-4, 2003: 173-197.
14. Mitrovic, A., Mayo, M., Suraweera, P and Martin, B. Constraint-based tutors: a success story. Proc. 14th Int. Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE-2001, Budapest, June 2001, L. Monostori, J. Vancza and M. Ali (eds), Springer-Verlag Berlin Heidelberg LNAI 2070, 2001: 931-940.
15. Mitrovic, A. Supporting Self-Explanation in a Data Normalization Tutor. In: V. Aleven, U. Hoppe, J. Kay, R. Mizoguchi, H. Pain, F. Verdejo, K. Yacef (eds) Supplementary proceedings, Artificial Intelligence in Education, AIED 2003, pp. 565-577, 2003
16. DARPA Agent Markup Language, http://www.daml.org/

Automatic Acquisition of Knowledge for Constraint-based Tutors

Pramuditha Suraweera

Intelligent Computer Tutoring Group
Department of Computer Science, University of Canterbury
Private Bag 4800, Christchurch, New Zealand
psu16@cosc.canterbury.ac.nz

Intelligent Tutoring Systems (ITS) assist students in learning by adaptively providing pedagogical assistance. Numerous empirical studies have shown that students learn more effectively by interacting with ITSs in comparison to traditional classroom based teaching [1, 2]. Although ITSs are highly regarded as effective tools for education, developing an ITS is a time and labour intensive task requiring programming skill as well as knowledge engineering skills. A major proportion of the time and effort for building an ITS is spent on acquiring the domain knowledge required for providing adaptive assistance. Anderson and co-workers estimated that ten hours or more were required to produce a single production rule [3]. Our main goal is to automate the knowledge acquisition process to drastically reduce the time and effort require for building an ITS.

Researchers have been exploring ways of conquering the knowledge acquisition bottleneck ever since the inception of ITSs. Previous research including KnoMic (Knowledge Mimic) [4], Disciple [5] and Demonstr8 [6] have focussed on acquiring procedural knowledge by recording the actions of a domain expert and generalising the recorded trace using machine learning algorithms. Although these systems are well suited for inherently procedural domains like simulated environments, they fail to acquire declarative knowledge required for non-procedural domains. Our goal is to develop an authoring system that is capable of acquiring knowledge for procedural as well as non-procedural domains.

Constraint based modelling (CBM) [7] is a student modelling technique that somewhat eases the knowledge acquisition bottleneck by using a more abstract representation of the domain compared to other popular domain modelling techniques [8]. However, building a constraint base still remains a major challenge. Mitrovic reported that, she took just over an hour to produce a constraint for SQL-Tutor, which currently contains more than 650 constraints [9]. Our research is focussed on automating the process of acquiring knowledge for constraint-based tutors.

The authoring system will be an extension to the web-based tutoring shell, named WETAS [10], that facilitates building constraint-based tutors. WETAS provides all the domain-independent components for a text-based ITS, including the user interface, pedagogical module and student modeller. The main limitation of WETAS is its lack of support for authoring the domain model.

We propose a four-stage process to infer constraints automatically. During the first phase the domain expert composes an ontology of the instructional domain. At the completion of the ontology, the system analyses the ontology and extracts syntax constraints directly from it. The third phase involves learning from examples. During this phase the system generates constraints by identifying commonalities between solutions provided by the domain expert. Finally the constraint set is validated with the assistance of the domain expert. The system would generate examples to be labelled as correct or incorrect by the expert.

An ontology contains a lot of information about the domain and it is much easier to create than the final domain model. Ontology defines the concepts of the domain and the relationships between them. Each concept has a set of attributes that describes itself. The range of the attributes specified in terms of minimum and maximum values or a set of distinct values can be directly translated to constraints. Furthermore, the minimum and maximum number of instances that can participate in a relationship (cardinality) can also be translated directly to constraints.

The authoring system analyses the ontology composed by the domain expert and generates constraints during the second phase. Since all the restrictions specified in the ontology deal with the syntax of the domain, the generated constraints are also syntactic in nature. As an example consider an ontology for the domain of punctuation in the English language. The ontology would contain a 'sentence' concept and a 'period' concept. The 'sentence' concept would be involved in a relationship with the 'period' concept with a minimum and maximum cardinality of 1. This translates directly to a constraint that specifies that a 'sentence' must have exactly one 'period'.

The domain expert has to specify the representation for solutions prior to entering problems and sample solutions. The solution representation is a decomposition of the solution into components consisting of a list of instances of concepts. For example, a sentence in English consists of a list of words and a list of punctuation marks.

The domain expert enters problems and solutions in the next phase which uses learning from examples techniques to induce semantic constraints. After a solution is specified by the author, the system evaluates it against its collection of automatically generated syntax constraints. When a discrepancy is identified the expert is alerted and they may chose to alter the solution or alter the ontology. The expert is encouraged to enumerate all correct solutions to demonstrate different ways of solving a problem. While the expert enters in alternative correct solution, the system attempts to match each component of the solution to components of the initial solution. These matches are later used to compose a set of semantic constraints that compare the student's solution against the system's ideal solution. The expert is also called upon to provide typical erroneous solutions for the system to generate semantic constraints that identify typical student errors.

The validation phase involves ensuring the correctness of the generated constraints. The expert can go through the generated constraint set and ensure that it does not contain an redundant or erroneous ones. The expert may either directly modify erroneous constraints or provide new examples to illustrate the reasoning for disputing the constraint. The expert may also opt to validate the constraints by labelling system generated examples as correct or incorrect.

A brief description on a system for acquiring the domain knowledge required for constraint-based tutors was provided. The envisaged interactions with the domain expert were outlined.

Currently, the implementation of the ontology composer and the syntax constraints generator are complete. The functionality of the two modules were tested by modelling an ontology for ER modelling, a popular database modelling technique. The syntax constraints generated by the system covered all the syntax constraints found in KERMIT [11], a CBM tutor for the same domain. Future work involves completing the learning from examples module and constraint validation module. We indent to conduct a comprehensive evaluation study at the completion of the authoring system.

References

1. Koedinger, K.R., Anderson, J., Hadley, W., Mark, M.A.: Intelligent tutoring goes to school in the big city. International Journal of Artificial Intelligence in Education **8** (1997) 30–43
2. Mitrovic, A., Ohlsson, S.: Evaluation of a constraint-based tutor for a database language. International Journal on AIED **10** (1999) 238–256
3. Anderson, J.R., Corbett, A., Koedinger, K., Pelletier, R.: Cognitive tutors: Lessons learned. Journal of the Learning Sciences **4** (1996) 167–207
4. Lent, M.v., Laird, J.E.: Learning procedural knowledge through observation. In: International conference on Knowledge capture, Victoria, British Columbia, Canada, ACM Press (2001) 179–186
5. Tecuci, G., Keeling, H.: Developing an intelligent educational agent with disciple. International Journal of Artificial Intelligence in Education **10** (1999) 221–237
6. Blessing, S.B.: A programming by demonstration authoring tool for model-tracing tutors. International Journal of Artificial Intelligence in Education **8** (1997) 233–261
7. Ohlsson, S.: Constraint-based student modelling. In: Student Modelling: the Key to Individualized Knowledge-based Instruction, Berlin, Springer-Verlag (1994) 167–189
8. Mitrovic, A., Koedinger, K., Martin, B.: A comparative analysis of cognitive tutoring and constraint-based modeling. In Brusilovsky, P., Corbett, A., Rosis, F.d., eds.: 9th International conference on User Modelling UM2003. Volume LNAI 2702., Pittsburgh, USA, Springer-Verlag (2003) 313–322
9. Mitrovic, A.: Experiences in implementing constraint-based modelling in sql-tutor. In Goettl, B.P., Halff, H.M., Redfield, C.L., Shute, V.J., eds.: 4th International Conference on Intelligent Tutoring Systems, San Antonio (1998) 414–423
10. Martin, B., Mitrovic, A.: Domain modeling: Art or science? In U. Hoppe, F.V..J.K., ed.: 11th Int. Conference on Artificial Intelligence in Education AIED 2003, IOS Press (2003) 183–190
11. Suraweera, P., Mitrovic, A.: Kermit: a constraint-based tutor for database modeling. In Cerri, S., Gouarderes, G., Paraguacu, F., eds.: 6th Int. Conf on Intelligent Tutoring Systems ITS 2002, Biarritz, France, LCNS 2363 (2002) 377–387

Appendix D

AIED 2005 paper

A Knowledge Acquisition System for Constraint-based Intelligent Tutoring Systems

Pramuditha Suraweera, Antonija Mitrovic and Brent Martin
Intelligent Computer Tutoring Group
Department of Computer Science, University of Canterbury
Private Bag 4800, Christchurch, New Zealand
{psu16, tanja, brent}@cosc.canterbury.ac.nz

Abstract. Building a domain model consumes a major portion of the time and effort required for building an Intelligent Tutoring System. Past attempts at reducing the knowledge acquisition bottleneck by automating the knowledge acquisition process have focused on procedural tasks. We present CAS (Constraint Acquisition System), an authoring system for automatically acquiring the domain model for non-procedural as well as procedural constraint-based tutoring systems. CAS follows a four-phase approach: building a domain ontology, acquiring syntax constraint directly from it, generating semantic constraints by learning from examples and validating the generated constraints. This paper describes the knowledge acquisition process and reports on results of a preliminary evaluation. The results have been encouraging and further evaluations are planned.

1 Introduction

Numerous empirical studies have shown that Intelligent Tutoring Systems (ITS) are effective tools for education. However, developing an ITS is a labour intensive and time consuming process. A major portion of the development effort is spent on acquiring the domain knowledge that accounts for the intelligence of the system. Our goal is to significantly reduce the time and effort required for building a knowledge base by automating the process.

This paper details the Constraint Acquisition System (CAS), which automatically acquires the required knowledge for ITSs by learning from examples. The knowledge acquisition process consists of four phases, initiated by an expert of the domain describing the domain in terms of an ontology. Secondly, syntax constraints are automatically generated by analysing the ontology. Semantic constraints are generated in the third phase from problems and solutions provided by the author. Finally, the generated constraints are validated with the assistance of the author.

The remainder of the paper is initiated by a brief introduction to Constraint-based modelling, the student modelling technique focused in this research, and a brief overview of related research. We then present a detailed description of CAS, including its architecture and a description of the knowledge acquisition process. Finally, conclusions and future work is outlined.

2 Related work

Constraint based modelling (CBM) [6] is a student modelling approach that somewhat eases the knowledge acquisition bottleneck by using a more abstract representation of the domain compared to other commonly used approaches [5]. However, building constraint sets still remains a major challenge. Our goal is to significantly reduce the time and effort required for acquiring the domain knowledge for CBM tutors by automating the knowledge acquisition process. Unlike other automated knowledge acquisition systems, we aim to produce a system that has the ability to acquire knowledge for non-procedural, as well as procedural, domains.

Existing systems for automated knowledge acquisition have focused on acquiring procedural knowledge in simulated environments or highly restrictive environments. KnoMic [10] is a learning-by-observation system for acquiring procedural knowledge in a simulated environment. It generates the domain model by generalising recorded domain experts' traces. Koedinger et al have constructed a set of authoring tools that enable non AI experts to develop cognitive tutors. They allow domain experts to create "Pseudo tutors" which contain a hard coded domain model specific to the problems demonstrated by the expert [3]. Research has also been conducted to generalise the domain model of "Pseudo tutors" by using machine learning techniques [2].

Most existing systems focus on acquiring procedural knowledge by recording the domain expert's actions and generalising recorded traces using machine learning algorithms. Although these systems appear well suited to tasks where goals are achieved by performing a set of steps in a specific order, they fail to acquire knowledge for non-procedural domains, i.e. where problem-solving requires complex, non-deterministic actions in no particular order. Our goal is to develop an authoring system that can acquire procedural as well as declarative knowledge.

The domain model for CBM tutors [7] consists of a set of constraints, which are used to identify errors in student solutions. In CBM knowledge is modelled by a set of constraints that identify the set of correct solutions from the set of all possible student inputs. CBM represents knowledge as a set of ordered pairs of relevance and satisfaction conditions. The relevance condition identifies the states in which the represented concept is relevant, while the satisfaction condition identifies the subset of the relevant states in which the concept has been successfully applied.

3 Constraint Authoring System

The proposed system is an extension of WETAS [4], a web-based tutoring shell that facilitates building constraint-based tutors. WETAS provides all the domain-independent components for a text-based ITS, including the user interface, pedagogical module and student modeller. The pedagogical module makes decisions based on the student model regarding problem/feedback generation, and the student modeller evaluates student solutions by comparing them to the domain model and updates the student model. The main limitation of WETAS is its lack of support for authoring the domain model.

As WETAS does not provide any assistance for developing the knowledge base, typically a knowledge base is composed using a text editor. Although the flexibility of a text editor may be adequate for knowledge engineers, novices tend to be overwhelmed by the task. The goal of CAS (Constraint Authoring System) is to reduce the complexity of the task by automating the constraint acquisition process. As a consequence the time and effort required for building constraint bases should reduce dramatically.

CAS consists of an ontology workspace, ontology checker, problem/solution manager, syntax and semantic constraint generators, and constraint validation as depicted in Figure 1. During the initial phase, the domain expert develops an ontology of the domain in the ontology workspace. This is then evaluated by the ontology checker, and the result is stored in the ontology repository.

The syntax constraints generator analyses the completed ontology and generates syntax constraints directly from it. These constraints are generated from the restrictions on attributes and relationships specified in the ontology. The resulting constraints are stored in the syntax constraints repository.

CAS induces semantic constraints during the third phase by learning from sample problems and their solutions. Prior to entering problems and sample solutions, the domain expert specifies the representation for solutions. This is a decomposition of the solution into components consisting of a list of instances of concepts. For example, an algebraic equation consists of a list of terms in the left hand and a list of terms in the right hand side.

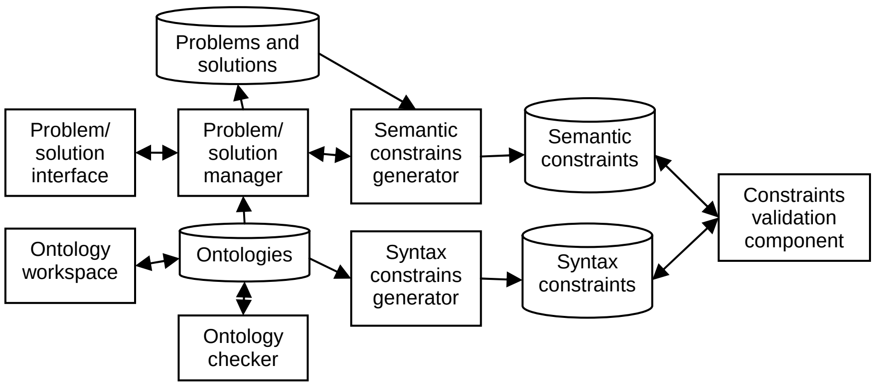

Problems and solutions
Problem/ solution interface
Problem/ solution manager
Semantic constrains generator
Semantic constraints
Constraints validation component
Ontology workspace
Ontologies
Syntax constrains generator
Syntax constraints
Ontology checker

Figure 1: Architecture of the constraint-acquisition system

The final phase involves ensuring the validity of the generated constraints. During this phase the system generates examples to be validated by the author. In situations where the author's validation conflicts with the system's evaluation according to the domain model, the author is requested to provide further examples to illustrate the rationale behind the conflict. The new examples are then used to resolve the conflicts, and may also lead to the generation of new constraints.

3.1 Modelling the domain's ontology

Domain ontologies play a central role in the knowledge acquisition process of the constraint authoring system [9]. A preliminary study conducted to evaluate the role of ontologies in manually composing a constraint base showed that constructing a domain ontology assisted the composition of the constraints [8]. The study showed that ontologies help organise constraints into meaningful categories. This enables the author to visualise the constraint set and to reflect on the domain, assisting them to create more complete constraint bases.

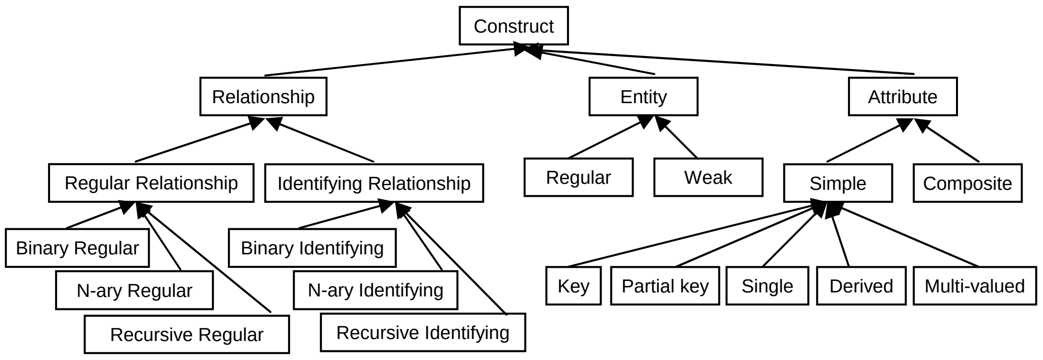


Figure 2: Ontology for ER modelling domain

An ontology describes the domain by identifying important concepts and relationships between them. It outlines the hierarchical structure of the domain in terms of sub- and super-concepts. CAS contains an ontology workspace for modelling an ontology of the domain. An example ontology for Entity Relationship Modelling is depicted in Figure 2. The root node, *Construct,* is the most general concept, of which *Relationship*, *Entity* and *Attribute* are sub-concepts. *Relationship* is further specialised into *Regular* and *Identifying*, which are the two possible types of relationships, and so on.

As syntax constraints are generated directly from the ontology, it is imperative that all relationships are correct. The ontology checker verifies that the relationships between concepts are correct by engaging the user in a dialog. The author is presented with lists of specialisations of concepts involved in a relationship and is asked to label the specialisations that are incorrect. For example, consider a relationship between *Binary identifying relationship* and *Attribute*. CAS asks whether all of the specialisations of *attribute* (*key, partial key, single-valued* etc) can participate in this relationship. The user indicates that *key* and *partial key* attributes cannot be used in this relationship. CAS therefore replaces the original relationship with specialised relationships between *Binary identifying relationship* and the nodes *single-valued, multi-valued* and *derived*.

Ontologies are internally represented in XML. We have defined set of XML tags specifically for this project, which can be easily be transformed to a standard ontology representation form such as DAML [1]. The XML representation also includes positional and dimensional details of each concept for regenerating the layout of concepts in the ontology.

3.2 Syntax Constraint Generation

An ontology contains much of information about the syntax of the domain: information about domain concepts; the domains (i.e. possible values) of their properties; restrictions on how concepts participate in relationships. Restrictions on a property can be specified in terms of whether its value has to be unique or whether it has to contain a certain value. Similarly, restrictions on the participation in relationships can also be specified in terms of minimum and maximum cardinality.

The syntax constraints generator analyses the ontology and generates constraints from all the restrictions specified on properties and relationships. For example, consider the *owner* relationship between *Binary identifying relationship* and *Regular entity* from the ontology in Figure 2, which has a minimum cardinality of 1. This restriction specifies that each *Binary identifying relationship* has to have at least one *Regular entity* participating as the *owner*, and can be translated to a constraint that asserts that each *Identifying relationship* found in a solution has to have at least one *Regular entity* as its *owner*.

To evaluate the syntax constraints generator, we ran it over the ER ontology in Figure 2. It produced a total of 49 syntax constraints, covering all the syntax constraints that were manually developed for KERMIT [7], an existing constraint-based tutor for ER modelling. The generated constraint set was more specific than the constraints found in KERMIT, i.e. in some cases several constraints generated by CAS would be required to identify the problem states identified by a single constraint in KERMIT. This may mean that the set of generated constraints would be more effective for an ITS, since they would provide feedback that is more specific to a single problem state. However, it is also possible that they would be overly specific.

We also experimented with basic algebraic equations, a domain significantly different to ER modelling. The ontology for algebraic equations included only four basic operations: addition, subtraction, multiplication and division. The syntax constraints generator produced three constraints from an ontology composed for this domain, including constraints that ensure whenever an opening parenthesis is used there should be a corresponding closing parenthesis, a constant should contain a plus or minus symbol as its sign, and a constant's value should be greater than or equal to 0. Because basic algebraic expressions have very little syntax restrictions, three constraints are sufficient to impose the basic syntax rules.

3.3 Semantic Constraint Generation

Semantic constraints are generated by a machine learning algorithm that learns from examples. The author is required to provide several problems, with a set of correct solutions for each depicting different ways of solving it. A solution is composed by populating each of its components by adding instances of concepts, which ensures that a solution strictly adheres to the domain ontology. Alternate solutions, which depict alternate ways of solving the problem, are composed by modifying the first solution. The author can transform the first solution into the desired alternative by adding, editing or dropping elements. This reduces the amount of effort required for composing alternate solutions, as most alternatives are similar. It also enables the system to correctly identify matching elements in two alternate solutions.

The algorithm generates semantic constraints by analysing pairs of solutions to identify similarities and differences between them. The constraints generated from a pair of solutions contribute towards either generalising or specialising constraints in the main constraint base. The detailed algorithm is given in Figure 3.

```
a.  For each problem $P_i$
b.  For each pair of solutions $S_i$ & $S_j$
        a.  Generate a set of new constraints N
        b.  Evaluate each constraint $CB_i$ in main constraint base, CB, against $S_i$ & $S_j$,
                If $CB_i$ is violated, generalise or specialise $CB_i$ to satisfy $S_i$ & $S_j$
        c.  Evaluate each constraint $N_i$ in set N against each previously analysed pair of solu-
            tions $S_x$ & $S_y$ for each previously analysed problem $P_z$,
                If $N_i$ is violated, generalise or specialise $CB_i$ to satisfy $S_x$ & $S_y$
        d.  Add constraints in N that were not involved in generalisation or specialisation to CB
```

Figure 3: Semantic constraint generation algorithm

The constraint learning algorithm focuses on a single problem at a time. Constraints are generated by comparing one solution to another of the same problem, where all permutations of solution pairs, including solutions compared to themselves, are analysed. Each solution pair is evaluated against all constraints in the main constraint base. Any that are violated are either specialised to be irrelevant for the particular pair of solutions, or generalised to satisfy that pair of solutions. Once no constraint in the main constraint base is violated by the solution pair, the newly generated set of constraints is evaluated against all previously analysed pairs of solutions. The violated constraints from this new set are also either specialised or generalised in order to be satisfied. Finally, constraints in the new set that are not found in the main constraint base are added to the constraint base.

```
1.  Treat $S_i$ as the ideal solution (IS) and $S_j$ as the student solution (SS)
2.  For each element A in the IS
        a.  Generate a constraint that asserts that if IS contains the element A, SS should con-
            tain a matching element
        b.  For each relationship that element is involved with,
                Generate constraints that ensures that the relationship holds between the corre-
                sponding elements of the SS
3.  Generalise the properties of similar constraints by introducing variables or wild cards
```

Figure 4: Algorithm for generating constraints from a pair of solutions

New constraints are generated from a pair of solutions following the algorithm outlined in Figure 4. It treats one solution as the ideal solution and the other as the student solution. A constraint is generated for each element in the ideal solution, asserting that if the ideal solution contains the particular element, the student solution should also contain the matching element.

E.g. Relevance: IS.Entities has a Regular entity
Satisfaction: SS.Entities has a Regular entity

In addition, three constraints are generated for each relationship that an element participates with. Two constraints ensure that a matching element exists in SS for each of the two elements of IS participating in the relationship. The third constraint ensures that the relationship holds between the two corresponding elements of SS.

```
E.g. 1. Relevance: IS.Entities has a Regular entity
          AND IS.Attributes has a Key
          AND SS.Entities has a Regular entity
          AND IS Regular entity is in key-attribute with Key
          AND IS Key is in belong to with Regular entity
        Satisfaction: SS.Attributes has a Key

     2. Relevance: IS.Entities has a Regular entity
          AND IS.Attributes has a Key
          AND SS.Attributes has a Key
          AND IS Regular entity is in key-attribute with Key
          AND IS Key is in belong to with Regular entity
        Satisfaction: SS.Entities has a Regular entity

     3. Relevance: IS.Entities has a Regular entity
          AND IS.Attributes has a Key
          AND SS.Entities has a Regular entity
          AND SS.Attributes has a Key
          AND IS Regular entity is in key-attribute with Key
          AND IS Key is in belong to with Regular entity
        Satisfaction: SS Regular entity is in key-attribute with Key
          AND SS Key is in belong to with Regular entity
```

```
a. If constraint set, C-set that does not contain violated constraint V, has a similar but a more restrictive constraint C then replace V with C and exit.
b. If C-set has a constraint C that has the same relevance condition but different satisfaction condition to V,
     Add the satisfaction condition of C as a disjunctive test to the satisfaction of V, remove C from C-set and exit
c. Find a solution $S_k$ that satisfies constraint V
d. If a matching element can be found in $S_j$ for each element in $S_k$ that appears in the satisfaction condition,
     Generalise satisfaction of V to include the matching elements as a new test with a disjunction and exit
e. Restrict the relevance condition of V to be irrelevant for solution pair $S_i$ & $S_j$, by adding a new test to the relevance signifying the difference and exit
f. Drop constraint
```

Figure 5: Algorithm for generalising or specialising violated constraints

The constraints that get violated during the evaluation stage are either specialised or generalised according to the algorithm outlined in Figure 5. It deals with two sets of constraints (C-set): the new set of constraints generated by a pair of solutions and the main constraint base. The algorithm remedies each violated constraint individually by either specialising it or generalising it. If the constraint cannot be resolved, it is labelled as an incorrect constraint and the system ensures that it does not get generated in the future.

The semantic constraints generator of CAS produced a total of 135 constraints for the domain of ER modelling using the ontology in Figure 2 and six problems. The problems supplied to the system were simple and similar to the basic problems offered by KERMIT. Each problem focused on a set of ER modelling constructs and contained at least two solutions that exemplified alternate ways of solving the problem. The solutions were selected that maximised the differences between them. The differences between most solutions were small because ER modelling is a domain that does not have vastly different solutions. However, problems that can be solved in different ways consisted of significantly different solutions.

The generated constraints covered 85% of the 125 constraints found in KERMIT's constraint-base, which was built entirely manually and has proven to be effective. After further analysing the generated constraints, it was evident that the reason for not generating most of the missing constraints was due to a lack of examples. 85% coverage is very encouraging, considering the small set of sample problems and solutions. It is likely that providing further sample problems and solutions to CAS would increase the completeness of the generated domain model. Although the problems and solutions were specifically chosen to improve the system's effectiveness in producing semantic constraints, we assume that a domain expert would also have the ability to select good problems and provide solutions that show different ways of solving a problem. Moreover, the validation phase, which is yet to be completed, would also produce constraints with the assistance of the domain expert.

CAS also produced some modifications to existing constraints found in KERMIT, which improved the system's ability to handle alternate solutions. For example, although the constraints in KERMIT allowed weak entities to be modelled as composite multivalued attributes, in KERMIT the attributes of weak entities were required to be of the same type as the ideal solutions. However CAS correctly identified that when a weak entity is represented as a composite multivalued attribute, the partial key of the weak entity has to be modelled as simple attributes of the composite attribute. Furthermore, the identifying relationship essential for the weak entity becomes obsolete. These two examples illustrate how CAS improved upon the original domain model of KERMIT.

We also evaluated the algorithm in the domain of algebraic equations. The task involved specifying an equation for the given textual description. As an example, consider the problem "Tom went to the shop to buy two loafs of bread, he gave the shopkeeper a \$5 note and was given \$1 as change. Write an expression to find the price of a loaf of bread using x to represent the price". It can be represented as $2x + 1 = 5$ or $2x = 5 – 1$. In order to avoid the need for a problem solver, the answers were restricted to not include any simplified equations. For example the solution "$x = 2$" would not be accepted because it is simplified.

a) Relevance: IS LHS has a Constant (?Var1)
Satisfaction: SS LHS has a Constant (?Var1)
or SS RHS has a Constant (?Var1)

b) Relevance: IS RHS has a +
Satisfaction: SS LHS has a –
or SS RHS has a +

c) Relevance: IS RHS has a Constant(?Var1)
and IS RHS has a –
and SS LHS has a Constant(?Var1)
and SS LHS has a +
and IS Constant (?Var1) is in Associated-operator with –
Satisfaction: SS Constant (?Var1) is in Associated-operator with +

Figure 6: Sample constraints generated for Algebra

The system was given five problems and their solutions involving addition, subtraction, division and multiplication for learning semantic constraints. Each problem contained three or four alternate solutions. CAS produced a total of 80 constraints. Although the completeness of the generated constraints is yet to be formally evaluated, a preliminary assessment revealed that the generated constraints are able to identify correct solutions and point out many errors. Some generated constraints are shown in Figure 6. An algebraic equation consists of two parts: a left hand side (LHS) and a right hand side (RHS). Constraint *a* in Figure 6 specifies that for each constant found in the LHS of the Ideal solution (IS), there has to be an equal constant in either the LHS or the student solution (SS) or the RHS. Similarly, constraint *b* specifies that an addition symbol found in the RHS of the IS should exist in the SS as either an addition symbol in the same side or a subtraction in the opposite side. Constraint *c* ensures the existence of the relationship between the operators and the constants. Thus, a constant in the RHS of the IS with a subtraction attached to it, can appear as a constant with addition attached to it in the LHS of the SS.

4 Conclusions and Future work

We provided an overview of CAS, an authoring system that automatically acquires the constraints required for building constraint-based Intelligent Tutoring Systems. It follows a four-stage process: modelling a domain ontology, extracting syntax constraints from the ontology, generating semantic constraints and finally validating the generated constraints.

We undertook a preliminary evaluation in two domains: ER modelling and algebra word problems. The domain model generated by CAS for ER modelling covered all syntax constraints and 85% of the semantic constraints found in KERMIT [7] and unearthed some discrepancies in KERMIT's constraint base. The results are encouraging, since the constraints were produced by analysing only 6 problems. CAS was also used to produce constraints for the domain of algebraic word problems. Although the generated constraints have not been formally analysed for their completeness, it is encouraging that CAS is able to handle two vastly different domains.

Currently the first three phases of the constraints acquisition process have been completed. We are currently developing the constraint validation component, which would also contribute towards increasing the quality of the generated constraint base. We also will be enhancing the ontology workspace of CAS to handle procedural domains. Finally, the effectiveness of CAS and its ability to scale to domains with large constraint bases has to be empirically evaluated in a wide range of domains.

References

[1] DAML. DARPA Agent Markup Language, http://www.daml.org.

[2] Jarvis, M., Nuzzo-Jones, G. and Heffernan, N., *Applying Machine Learning Techniques to Rule Generation in Intelligent Tutoring Systems*. In: Lester, J., et al. (eds.) Proc. ITS 2004, Maceio, Brazil, Springer, pp. 541-553, 2004.

[3] Koedinger, K., et al., *Openning the Door to Non-programmers: Authoring Intelligent Tutor Behavior by Demonstration*. In: Lester, J., et al. (eds.) Proc. ITS 2004, Maceio, Brazil, Springer, pp. 162-174, 2004.

[4] Martin, B. and Mitrovic, A., *WETAS: a Web-Based Authoring System for Constraint-Based ITS*. Proc. 2nd Int. Conf on Adaptive Hypermedia and Adaptive Web-based Systems AH 2002, Malaga, Spain, LCNS, pp. 543-546, 2002.

[5] Mitrovic, A., Koedinger, K. and Martin, B., *A comparative analysis of cognitive tutoring and constraint-based modeling*. In: Brusilovsky, P., et al. (eds.) Proc. 9th International conference on User Modelling UM2003, Pittsburgh, USA, Springer-Verlag, pp. 313-322, 2003.

[6] Ohlsson, S., *Constraint-based Student Modelling*. Proc. Student Modelling: the Key to Individualized Knowledge-based Instruction, Berlin, Springer-Verlag, pp. 167-189, 1994.

[7] Suraweera, P. and Mitrovic, A. *An Intelligent Tutoring System for Entity Relationship Modelling*. Int. J. Artificial Intelligence in Education, vol 14 (3,4), 2004, pp. 375-417.

[8] Suraweera, P., Mitrovic, A. and Martin, B., *The role of domain ontology in knowledge acquisition for ITSs*. In: Lester, J., et al. (eds.) Proc. Intelligent Tutoring Systems 2004, Maceio, Brazil, Springer, pp. 207-216, 2004.

[9] Suraweera, P., Mitrovic, A. and Martin, B., *The use of ontologies in ITS domain knowledge authoring*. In: Mostow, J. and Tedesco, P. (eds.) Proc. 2nd Int. 2nd International Workshop on Applications of Semantic Web for E-learning SWEL'04, ITS2004, Maceio, Brazil, pp. 41-49, 2004.

[10] van Lent, M. and Laird, J.E., *Learning Procedural Knowledge through Observation*. Proc. International conference on Knowledge capture, pp. 179-186, 2001.

Appendix E

ITS 2006 paper

Authoring Constraint-based Tutors in ASPIRE

Antonija Mitrovic, Pramuditha Suraweera, Brent Martin,
Konstantin Zakharov, Nancy Milik and Jay Holland

Intelligent Computer Tutoring Group
University of Canterbury, Christchurch, New Zealand
{tanja, psu16, brent, kza10, nmi14, jah130}@cosc.canterbury.ac.nz

Abstract: This paper presents a project the goal of which is to develop ASPIRE, a complete authoring and deployment environment for constraint-based intelligent tutoring systems (ITSs). ASPIRE is based on our previous work on constraint-based tutors and WETAS, the tutoring shell. ASPIRE consists of the authoring server (ASPIRE-Author), which enables domain experts to easily develop new constraint-base tutors, and a tutoring server (ASPIRE-Tutor), which deploys the developed systems. Preliminary evaluation shows that ASPIRE is successful in producing domain models, but more thorough evaluation is planned.

1 Introduction

Building a constraint-based tutor, like any other ITS, is a labour-intensive process that requires expertise in constraint-based modelling (CBM) and programming. While ITSs contain a few modules that are domain-independent, their domain model, which consumes the majority of the development effort, is unique. Our goal is to reduce the time and effort required for producing ITSs by building an authoring system that can generate the domain model with the assistance of a domain expert and produce a fully functional system. We also envisage that the authoring system would enable teachers, with little or no expertise in CBM, to build their own ITSs.

This paper presents ASPIRE, an authoring system that assists in the process of composing domain models for constraint-based tutors and automatically serves tutoring systems on the web. The proposed system is an enhancement of WETAS [4, 5], a web-based tutoring shell that facilitates building constraint-based tutors. WETAS is a prototype system that provides all the domain-independent components for text-based ITSs. The main limitation of WETAS is its lack of support for authoring domain models. ASPIRE guides the author through a semi-automated process for building the domain model and seamlessly deploys the resulting domain model to produce a fully functional web-based tutoring system.

The paper commences with a brief introduction to related authoring systems for building ITSs. Section 3 details the ASPIRE authoring system, including an outline of the domain authoring process and the architecture of the system. We also include an overview the constraint generation algorithms, the central component of the authoring process. Finally, Section 4 presents conclusions and the directions of future work.

2 Related Work

Murray [10] classified ITS authoring tools into two main groups: pedagogy-oriented and performance-oriented. Pedagogy-oriented systems focus on instructional sequencing and teach relatively fixed content. On the other hand, performance-oriented systems focus on providing rich learning environments, where students learn by solving problems while receiving dynamic feedback on their progress. These systems have a deep model of expertise, which enables the tutor to correct the student as well as provide assistance on problem solving. Authoring systems thus need to support the acquisition of domain models. Typically, sophisticated machine learning techniques are used for acquiring domain rules with the assistance of a domain expert.

Only a few authoring systems are capable of generating domain models. Disciple, developed by Tecuci and co-workers [15, 16], is an example of a learning agent shell for developing intelligent educational agents. A domain expert teaches the agent to perform domain-specific tasks, similar to a manner of an expert teaching an apprentice, by providing examples and explanations. The expert is also required to supervise and correct the behaviour of the agent. Disciple acquires knowledge using a collection of complementary learning methods including inductive learning from examples, explanation-based learning, learning by analogy and learning by experimentation. A completed Disciple agent can be used to interact and guide students in performing tasks of the domain.

The Cognitive Tutor Authoring Tools (CTAT) [1, 2] assist in the creation and delivery of ITSs based on model tracing. The main goal of these tools is to reduce the amount of artificial intelligence (AI) programming expertise required. The system allows authors to create two types of tutors: ‘Cognitive tutors’ and ‘Pseudo tutors’. ‘Cognitive tutors’ contain a cognitive model that simulates the student's thinking to monitor and provide pedagogical assistance during problem solving. In contrast, ‘Pseudo tutors’ do not contain a cognitive model: to develop a tutor of this kind, the author needs to specify a recording of possible student actions and corresponding feedback messages. Although ‘Pseudo tutors’ do not require AI programming, they are specific to the demonstrated set of problems, and cannot deal with student actions’ which are not pre-specified by the author.

3 ASPIRE

ASPIRE assists with the creation and delivery of constraint-based tutoring systems. It generates constraints that make up the domain model with the assistance of the domain expert, minimising the programming expertise required for developing a new constraint-based tutor. The system also provides all the domain-independent functionality of constraint-based ITSs.

3.1 Authoring Process

Authoring a constraint-based tutor in ASPIRE is a semi-automated process, carried out with the assistance of the domain expert. The authoring process, summarised in Figure 1, consists of nine distinct phases. Initially, the author specifies general features of the chosen instructional domain, such as whether the domain consists of a sub-domains focusing on specific areas, and whether the domain is procedural or not. In the case of procedural domains, the author is required to enumerate the problem-solving steps. As an example, let us consider the procedural domain of adding fractions. The problem-solving procedure can be broken down into four steps, as outlined in Figure 2. Initially, it is necessary to check whether the two fractions have the same denominator; if that is not the case, the lowest common denominator must be found. Step two involves modifying the two fractions to have the lowest common denominator (when needed). After that, the two fractions are added, which may result in an improper fraction. Finally, the result is to be simplified, if appropriate.

1. Specifying the domain characteristics
2. Composing the domain ontology
3. Modelling the problem and solution structures
4. Designing the student interface
5. Adding problems and solutions
6. Generating syntax constraints
7. Generating semantic constraints
8. Validating the generated constraints
9. Deploying the tutoring system

Figure 1. The phases of the authoring process

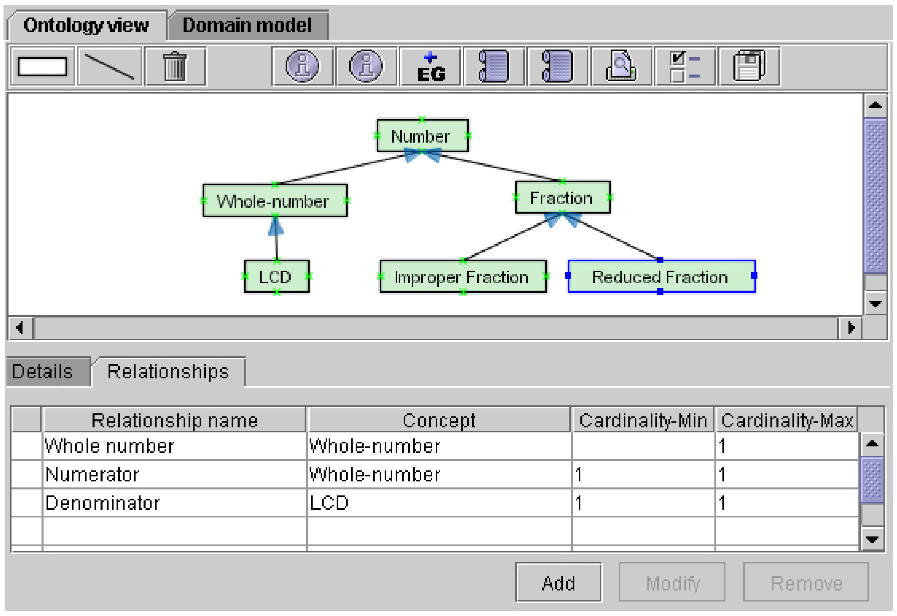
In the second phase, the author develops an ontology of the chosen instructional domain, which plays a central role in the authoring process. ASPIRE-Author provides an ontology workspace for visually modelling ontologies (Figure 3). A domain ontology describes the domain by identifying important concepts and relationships between them. The ontology outlines the hierarchical structure of the domain in terms of sub- and super-concepts. Each concept might have a number of properties, and may be related to many other domain concepts. A preliminary study conducted to evaluate the role of ontologies in manually composing a constraint base showed that constructing a domain ontology assisted the composition of constraints [13]. The study showed that ontologies support authors to reflect on the domain, organise constraints into meaningful categories and produce more complete constraint bases.

1. Find the lowest common denominator (LCD)
2. Convert fractions to LCD as denominator
3. Add the resulting fractions
4. Simplify the final result

Figure 2. Problem-solving procedure for fraction addition

An ontology for the domain of adding fractions is illustrated in Figure 3. It contains *Number* as the most generic concept, which has two specialisations, *Whole-*

number and *Fraction*. *Whole-number* is further specialised into lowest common denominator (*LCD*), while *Fraction* is specialised into *Improper* and *Reduced*. The specialization/generalization relationships between domain concepts are visually represented as arrows between concepts. Figure 3 shows three additional relationships defined for the *Reduced Fraction* concept: *whole number*, *numerator* and *denominator*. While *numerator* and *denominator* and mandatory relationships, *whole number* may only occur if the resulting fraction needs to be simplified.



Relationship name	Concept	Cardinality-Min	Cardinality-Max
Whole number	Whole-number		1
Numerator	Whole-number	1	1
Denominator	LCD	1	1

Figure 3. Ontology for adding fractions

In the third phase, the author specifies the problem/solution structures. Problems can consist of components (textual or graphical) and a problem statement. In our example domain, problems contain a common statement (“Add these two fractions”), and the problem to be solved (e.g. “1/3 + 1/5”). Student solutions may also consist of several components. The overall structure of solutions depends on whether the domain is procedural or declarative. A declarative task requires a single solution that may consist of a number of components, whereas a procedural task requires a solution for each step of the procedure. As the result, the structure of solutions for each step has to be modelled. The solution structure for fraction addition is outlined in Figure 4, showing also the corresponding domain concepts.

The student interface needs to be designed next. The final outcome of this phase is a form-based interface that can be used by students to compose their solutions. The system initially generates a default interface, placing an input area for each component defined in the solution structure [9]. The domain expert can rearrange the interface components in order to provide a more intuitive interface for students. An example of an interface for adding fractions is shown in Figure 5.

After designing the student interface, the author enters example problems and their solutions. For each problem, the author enters a problem statement, and one or more correct solutions. In order for the authoring system to learn about different ways of solving a problem, the expert is required to provide multiple solutions to a problem depicting different ways of solving it. These solutions are used by the authoring system for generating semantic constraints.

Problem solving step	Solution component	Concept
1. Find LCD	LCD	LCD
2. Convert fractions to LCD	Fraction 1 numerator Fraction 1 denominator Fraction 2 numerator Fraction 2 denominator	Improper fraction
3. Sum of improper fractions	Improper sum numerator Improper sum denominator	Improper fraction
4. Final reduced sum	Final sum whole number Final sum numerator Final sum denominator	Reduced fraction

Figure 4. Solution structure for adding two fractions

Once example problems and their solutions are available, ASPIRE-Author generates the domain model. The syntax constraint generator analyses the domain ontology and generates syntax constraints directly from it. These constraints are generated by translating the restrictions on the properties and relationships of concepts specified in the ontology, as detailed in Section 3.3. The constraint generator produces an extra set of syntax constraints for procedural domains that ensure that the student progresses correctly in the problem solving process.

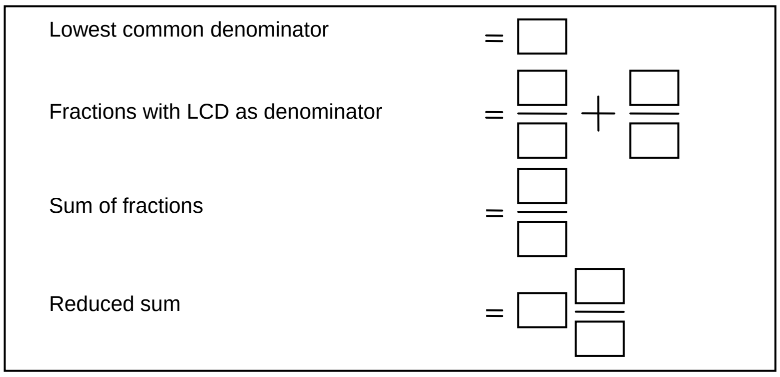


Figure 5. Student interface for adding two fractions

Semantic constraints are generated using a machine learning algorithm that learns from the solutions provided for each problem. It analysing pairs of solutions to identify similarities and differences between them. Section 3.4 provides more details on the semantic constraint generation algorithm.

The generated domain model is validated during the penultimate phase of authoring the domain model. The author requests the system to identify errors in an incorrect solution. If errors are identified incorrectly, further example problems and solutions have to be provided by the domain expert. The author may also examine a high-level description of each generated constraint and dispute them by providing counter examples.

Finally, the domain model is deployed as a tutoring system during the final phase of the authoring process. A new instance of a tutoring system is started in ASPIRE-Tutor, which can be tested by the domain expert and made available to students. The domain expert can evaluate the effectiveness of the domain model by analysing the learning curves for constraints produced by ASPIRE-Tutor.

3.2 Architecture

ASPIRE consists of an authoring server (ASPIRE-Author) for assisting with the development of new systems, and a tutoring server (ASPIRE-Tutor) for delivering tutors. Both servers are implemented in Allegro Common Lisp [3] as web servers for users to interact through a standard web browser. All required domain-dependent information, such as the domain model and other configuration details produced by ASPIRE-Author, are transferred to ASPIRE-Tutor as an XML database.

3.2.1 Authoring Server

The authoring server consists of a set of modules, where each module is assigned a specific set of responsibilities in generating constraint-based tutors. The basic architecture of the ASPIRE-Author, as depicted in Figure 6, consists of a web interface, authoring controller, constraint generator, constraint validator and the domain model manager [8]. The domain expert interacts with each component of the web interface to generate the domain model.

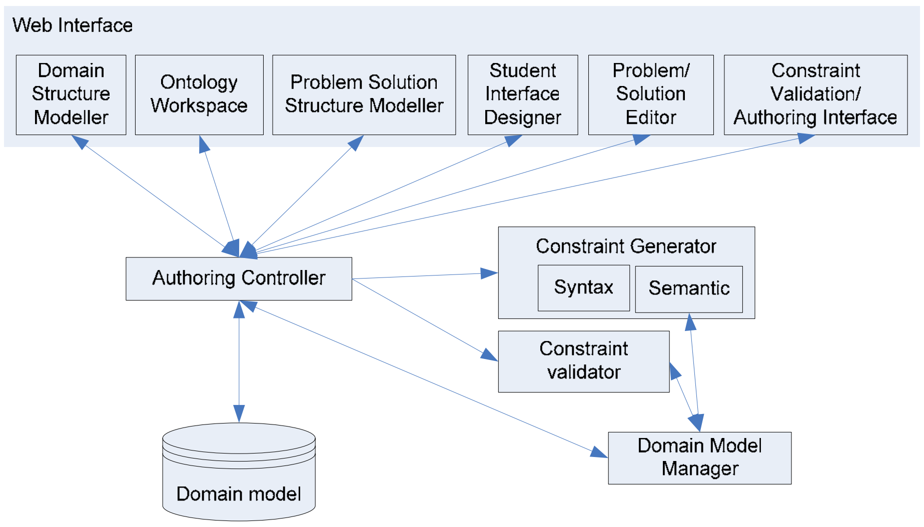


Figure 6. The architecture of ASPIRE-Author

The Authoring Controller manages the process and guides the author. This module receives all requests from the interface layer, initiates processes within other modules and returns the results to the relevant interface component.

The Syntax Constraint Generator is responsible for generating syntax constraints by analysing the domain ontology. Semantic constraints are generated by the Semantic Constraint Generator using a machine learning algorithm that learns from problems and their solutions. The Constraint Validator is responsible for carrying out all the necessary operations required for validating the constraints generated by the constraint generators.

The Domain Model Manager contains the necessary classes for storing the components of domain models. It is responsible for creating and updating domain model components such as ontology, problem solution structure, problems, solutions etc. The Domain Model Manager is also capable of producing XML representations of all domain model components for data transfer.

3.2.2 Tutoring Server

ASPIRE-Tutor (Figure 7) is also designed as a collection of modules, based on the typical ITS architecture. ASPIRE-Tutor is capable of serving a collection of tutoring systems in parallel. Each tutoring system served by ASPIRE-Tutor would have its own unique URL. Students can access the tutoring system relevant to them by pointing their browser to the appropriate URL.

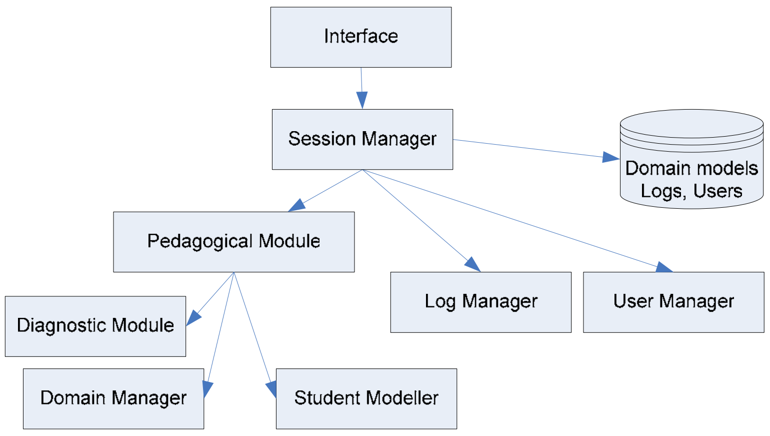


Figure 7. The architecture of ASPIRE-Tutor

The interface module is responsible for producing an interface for each tutoring system deployed on the server. The interface provides features such as login/logout, select/change problem, submit solution for evaluation etc.

The session manager is responsible for maintaining the state of each student during their interaction. The current state of a student is described by information such as the selected domain, sub-domain and problem number. The session manager also acts as the main entry point to the system, invoking the relevant modules to carry out necessary tasks. For example, when a student submits a solution to be validated, the session manager passes on all information to the pedagogical module, which returns the feedback to be presented to the student.

The Pedagogical Module (PM) decides how to respond to each student request. It is responsible for handing all pedagogy-related requests including selecting a new problem, evaluating a student's submission and viewing the student model. In the event of evaluating a student's submission and providing feedback, the PM delegates the task of evaluating the solution to the diagnostic module and decides on the appropriate feedback by consulting the student model. The student modeler maintains a long term model of the student's knowledge.

3.3 Syntax Constraints Generation

An ontology contains a lot of information about the syntax of the domain. Composing a domain ontology is a much easier task for the author than composing constraints that check whether the student has used correct syntax. The goal of syntax constraint generator is to extract all useful syntactic information from the ontology and translate them into syntax constraints for the domain model.

Syntax constraints are generated by analysing relationships between concepts and properties of concepts specified in the ontology. The algorithm extracts the restrictions specified for relationships and properties and generates syntax constraints by translating them into constraints. These constraints are applicable to both procedural and non-procedural domains. An extra set of constraints are generated for procedural domains to ensure that the student adheres to the correct problem-solving procedure. These constraints are generated by analysing the solution structure modelled during stage three of the authoring process. The syntax constraints generation algorithm is detailed in further in [12, 14].

ASPIRE-Author produced 11 constraints for fraction addition from the ontology in Figure 3 and the solution structure in Figure 4. For example, constraint 7 is relevant while the student is carrying out the first problem solving step ('Find LCD') and its satisfaction condition ensures that the student has entered the answer. As the domain does not contain any complicated syntax restrictions, and inputs are restricted by the student interface, the generated constraints are sufficient to ensure that students use the correct syntax and the correct problem-solving procedure.

The syntax constraint generation algorithm has been evaluated in a number of domains. The evaluations carried out for the domains of ER modelling and database normalisation produced promising results. All syntax constraints that were hand-crafted in KERMIT [7, 11], a successful constraint-based tutor for ER modelling were generated by ASPIRE. Furthermore, the algorithm produced all but two syntax constraints that existed in NORMIT [6, 7], an effective tutoring system for database normalisation.

3.4 Semantic Constraints Generation

Semantic constraints ensure that a student's solution satisfies all semantic requirements of a problem, by comparing the student's and ideal solution. They are generated by a machine learning algorithm. Problems and solutions provided by the author are used as examples for semantic constraint generation. Multiple solutions for a problem depict different ways of solving it, and enable the algorithm to generate constraints that can identify all correct solutions, regardless of the student's approach.

The algorithm generates new semantic constraints by analysing a pair of correct solutions for the same problem. Constraints are generated by identifying similarities and differences between two solutions. The process of generating constraints is iterated until all pairs of solutions are analysed. Each new pair of solutions can lead to either generalising or specialising previously generated constraints. If a newly analysed pair of solutions violate a previously generated constraint, its satisfaction condition is generalised in order to satisfy the solutions, or the constraint's relevance condition is specialised for the constraint to be irrelevant for the solutions. This algorithm is discussed in [12]. Evaluations performed show that the semantic constraints generator produced 85% of the semantic constraints found in KERMIT. Moreover, the generated constraints for the domain of database normalisation covered all the semantic constraints that exist in NORMIT.

39 semantic constraints were generated for fraction addition, from only two example problems. As each problem in this domain has only a single valid solution, semantic constraints check that the student's solution matches the ideal solution. For example, constraint 1 ensures that if the student is currently doing the first problem solving step ('Find LCD'), the LCD component of their solution is not empty (i.e., the student has specified the LCD) and the ideal solution contains an LCD (i.e. it is necessary to find the LCD for the current problem), then the student's answer needs to be equal to the one specified in the ideal solution.

The majority of generated semantic constraints ensure that relationships, such as fractions having a numerator and a denominator, exist in student solutions. As the interface implicitly forces these relationships, some semantic constraints are trivially satisfied. However, we believe that it is still necessary for the domain model to contain such constraints, because the author may design a less restrictive interface. Only two example problems were needed to generate semantic constraints for fraction addition, as the domain is very simple.

4 Conclusions

We provided an overview of ASPIRE, an authoring system that assists domain experts in building constraint-based ITSs and serves the developed tutoring systems over the web. ASPIRE follows a semi-automated process for generating domain models, and produces a fully functional web-based ITS, which can be used by students. We also outlined the constraint generation algorithms, which produced promising results during preliminary evaluations. ASPIRE-Author produced a satisfactory domain model for fraction addition, consisting of 11 syntax and 39 semantic constraints. The generated domain model can be used to power a tutoring system for students with minor modifications.

ASPIRE will be completed in July 2006, and then we will conduct a thorough evaluation of the system's effectiveness. We also intend to develop a tutorial outlining the authoring process to assist novices in building constraint-based tutoring systems using ASPIRE, especially modelling domain ontologies.

Acknowledgements: The ASPIRE project is supported by the eCDF grant from the Tertiary Education Commission of New Zealand. We thank all members of ICTG for their support.

References

1. Jarvis, M., Nuzzo-Jones, G., Heffernan, N., Applying Machine Learning Techniques to Rule Generation in Intelligent Tutoring Systems. In *ITS 2004*, (Maceio, Brazil, 2004), Springer, 541-553.
2. Koedinger, K., Aleven, V., Heffernan, N., McLaren, B. and Hockenberry, M., Openning the Door to Non-programmers: Authoring Intelligent Tutor Behavior by Demonstration. In *ITS 2004*, (Maceio, Brazil, 2004), Springer, 162-174.
3. Allegro Common Lisp (www.franz.com)
4. Martin, B., Mitrovic, A. Authoring Web-Based Tutoring Systems with WETAS. Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson, C-H Lee (eds) Proc. ICCE 2002 (Auckland, 2002), 183-187.
5. Martin, B., Mitrovic, A. Domain Modelling: Art or Science? In: U. Hoppe, F. Verdejo & J. Kay (ed) Artificial Intelligence in Education 2003, 183-190.
6. Mitrovic, A. The Effect of Explaining on Learning: a Case Study with a Data Normalization Tutor. In: C-K Looi, G. McCalla, B. Bredeweg, J. Breuker (eds) Proc. Artificial Intelligence in Education, 2005, IOS Press, 499-506.
7. Mitrovic, A., Suraweera, P., Martin, B., Weerasinghe, A. DB-suite: Experiences with Three Intelligent, Web-based Database Tutors. *Journal of Interactive Learning Research*, 15, 2004, 409-432.
8. Mitrovic, A., Martin, B., Suraweera, P., Zakharov, K., Milik, N., Holland, J. ASPIRE: Functional Specification and Architectural Design**.** Tech. Report TR-COSC 05/05, University of Canterbury, 2005.
9. Mitrovic, A., Martin, B., Suraweera, P., Zakharov, K., Milik, N., Holland, J. ASPIRE: Student Modelling and Domain Specification. Tech. Report TR-COSC 08/05, University of Canterbury, 2005.
10. Murray, T. An Overview of Intelligent Tutoring System Authoring Tools: Updated analysis of the state of the art. *Authoring tools for advanced technology learning environments*. 2003, 491-545.
11. Suraweera, P., Mitrovic, A., An Intelligent Tutoring System for Entity Relationship Modelling. *Artificial Intelligence in Education*, 14, (2004), 375-417.
12. Suraweera, P., Mitrovic, A., Martin, B., A Knowledge Acquisition System for Constraint-based Intelligent Tutoring Systems. In: C-K Looi, G. McCalla, B. Bredeweg, J. Breuker (eds) Artificial Intelligence in Education, 2005, IOS Press, 638-645.
13. Suraweera, P., Mitrovic, A., Martin, B., The role of domain ontology in knowledge acquisition for ITSs. In *Intelligent Tutoring Systems 2004*, (Maceio, Brazil, 2004), Springer, 207-216.
14. Suraweera, P., Mitrovic, A., Martin, B., The use of ontologies in ITS domain knowledge authoring. in *2nd Int. Workshop on Applications of Semantic Web for E-learning SWEL'04, ITS2004*, (Maceio, Brazil, 2004), 41-49.
15. Tecuci, G. *Building Intelligent Agents: An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies*. Academic press, 1998.
16. Tecuci, G., Keeling, H. Developing an Intelligent Educational Agent with Disciple. *Artificial Intelligence in Education*, 10, 1999, 221-237.

Appendix F

AIED 2007 paper

Constraint Authoring System: An Empirical Evaluation

Pramuditha SURAWEERA, Antonija MITROVIC and Brent MARTIN

Intelligent Computer Tutoring Group
Department of Computer Science, University of Canterbury
Private Bag 4800, Christchurch, New Zealand
{pramudi, tanja, brent}@cosc.canterbury.ac.nz

Abstract. Evaluation is an integral part of research that provides a true measure of effectiveness. This paper presents a study conducted to evaluate the effectiveness of CAS, a knowledge acquisition authoring system developed for generating the domain knowledge required for constraint-based tutoring systems with the assistance of a domain expert. The study involved a group of novice ITS authors composing domain models for adding two fractions. The results of the study showed that CAS was capable of generating highly accurate knowledge bases with considerably less effort than the effort required in manual composition.

Introduction

Composing the domain knowledge required for Intelligent Tutoring Systems (ITS) consumes the majority of the total development time [6]. The task requires a multi-faceted set of expertise, including knowledge engineering, AI programming and the domain itself. Our goal is to widen the knowledge acquisition bottleneck by empowering domain experts with little or no programming and knowledge engineering expertise to produce domain models necessary for ITSs.

Researchers have been exploring ways of automating the knowledge acquisition process since the inception of ITSs. Disciple, developed by Tecuci and co-workers[10], is an example of a learning agent shell for developing intelligent educational agents. A domain expert teaches the agent to perform domain-specific tasks by providing examples and explanations (similar to an expert teaching an apprentice), and Disciple uses machine learning techniques to infer the necessary domain knowledge. The Cognitive Tutor Authoring Tools (CTAT) [1] assist in the creation and delivery of model-tracing ITSs. The main goal of these tools is to reduce the amount of artificial intelligence (AI) programming expertise required. CTAT allows authors to create two types of tutors: 'Cognitive tutors' and 'Pseudo tutors'. 'Cognitive tutors' contain a cognitive model that simulates the student's thinking to monitor and provide pedagogical assistance during problem solving. In contrast, 'Pseudo tutors' do not contain a cognitive model: to develop a tutor of this kind, the author needs to record possible student actions and provide corresponding feedback messages.

We have developed an authoring system, named Constraint Authoring System (CAS) that generates a domain model with the assistance of a domain expert. The author is required to describe a domain in terms of an ontology and provide problems and their solutions. CAS analyses the provided information using machine learning techniques to generate a domain model.

This paper outlines a study conducted with a group of novice authors to evaluate the effectiveness of CAS. They were given the task of composing a domain model for a fractions addition ITS using CAS. The results showed that CAS was capable of generating highly accurate constraint bases even with the assistance of novices. It also showed that CAS reduced the overall effort required to produce domain models.

The remainder of the paper is organised into three sections. The next section gives a brief introduction to CAS. The results and analysis of the experiment are given in section three. The final section presents conclusions and future work.

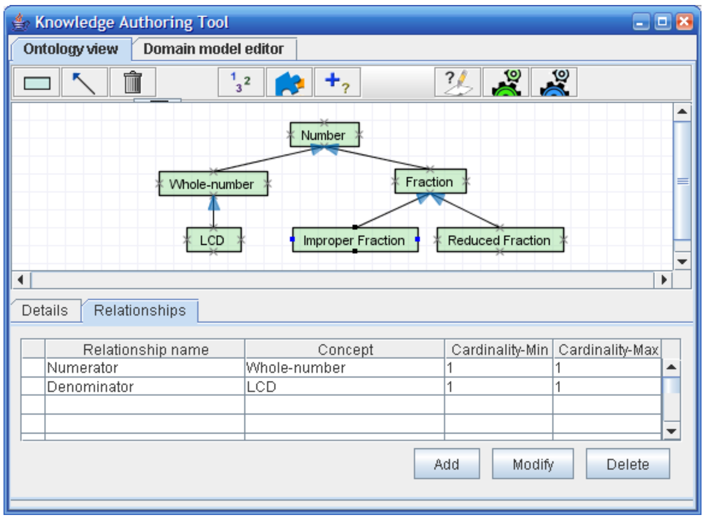
1. Constraint Authoring System

Constraint Acquisition System is an authoring system that generates the domain model required for constraint-based tutoring systems [5] with the assistance of a domain expert/ teacher. The goal of the system is to significantly reduce the time and effort required for composing constraint bases. We envisage that CAS would enable domain experts with minimal expertise in constraint-based modelling to produce new tutoring systems. The user is only required to model the domain in terms of an ontology and provide example problems and their solutions. Once the required domain-dependant information is provided, CAS generates constraints by analysing the provided information. Although the system is designed to support authors with minimal knowledge engineering expertise, it also offers utilities for experts in composing constraint bases. The system allows experts to modify constraints during the stage of validating the system-generated constraints. Users are also provided with editors for directly adding new constraints to the domain model.

A detailed discussion of CAS is beyond the scope of this paper and has been presented in [7]. Here we only give a short overview of its features. CAS is developed as an extension of WETAS [3], a web-based tutoring shell that facilitates building constraint-based tutors. WETAS provides all the domain-independent components for a text based ITS, including the user interface, pedagogical module and student modeller. It does not provide support for authoring domain models; however, authoring tools may be added to WETAS to provide this need. CAS was used in this manner.

Authoring knowledge using CAS is a semi-automated process with the assistance of a domain expert. The author carries out a number of tasks, including modelling the domain as an ontology, specifying the general structure of solutions and providing example problems and solutions. The constraint generators of CAS use this information to generate both syntax and semantic constraints. The syntax constraints are generated by analysing the ontology and translating each specified restriction into a constraint [9]. The semantic constraints generator uses machine learning techniques to generate constraints by analysing the problems and solutions provided by the domain expert [7]. It generates constraints by comparing and contrasting two alternative solutions to the same problem. Constraints are generated iteratively, and they are generalised or specialised during subsequent analysis of other solutions.

The interface of CAS consists of two main tabs: the “ontology view” and the “domain model editor”. The “ontology view” contains the ontology workspace and other tools necessary for composing the domain-dependant components necessary for each step of the authoring process. Figure 1 illustrates the ontology for the fraction addition domain, and the relationships defined for the *Improper Fraction* concept. The "ontology view" also contains an interface for adding problems and their solutions that are used for constraint generation. The "domain model editor" tab consists of a set of textual editors for viewing and modifying constraints generated by CAS. Constraints in these editors can be directly loaded into WETAS for testing in an ITS environment. The "domain model editor" tab also contains an editor for modifying and composing problems and solutions in the Lisp representation expected by WETAS.



Relationship name	Concept	Cardinality-Min	Cardinality-Max
Numerator	Whole-number	1	1
Denominator	LCD	1	1

Figure 1: CAS Interface, illustrating the ontology workspace

2. Evaluation Study

In previous work we evaluated CAS by comparing its generated domain models to the domain models developed manually [7, 9]. The domain models were generated with the assistance of expert authors. However, the goal of CAS is to support novice authors to develop ITSs. For that reason, we performed a study with a group of 13 novice authors to evaluate the effectiveness of the system as a whole. The participants were students enrolled in a 2006 graduate course about ITSs at the University of Canterbury. They were assigned the task of producing a complete ITS in WETAS for the domain of adding fractions, using CAS to author the domain model. The goal of the evaluation study was to validate three hypotheses: CAS makes it possible for novices to produce complete constraint bases; the process of authoring constraints using CAS requires less effort than composing constraints manually; the constraint generation algorithm depends on the ontology (i.e. an incomplete ontology will result in an incomplete constraint set).

The participants were required to compose all the domain-dependant components necessary for CAS to generate constraints, including a domain ontology, problems and solutions. After composing the required components, CAS can be used to generate syntax and semantic constraints in a high-level language (pseudo-code). At the time of the study, CAS was not able to convert these into the WETAS constraint language, so the participants were also required to perform this step manually. They were also free to modify/delete constraints.

The participants had attended 13 lectures on ITSs, including five on constraint-based modelling before the study. They were introduced to CAS and WETAS, and were given a task description document that outlined the process of authoring a domain model using CAS and described the WETAS constraint language. The participants were encouraged to follow the suggested authoring process. In addition to the task description, participants were also given access to all the domain model components of LBITS [2], a tutoring system for English language skills. They were also provided with an ontology for database modelling as an example. Finally, the students were instructed to use a particular interface style (one free-form text box per fraction) when building their tutor. The participants were allocated a period of six weeks to complete the task, however, most students only started working on it at the end of week 3.

Twelve out of the 13 participants completed the task satisfactorily, i.e. produced working tutoring systems. One participant failed to complete the final step of converting the pseudo-code constraints to the target constraint language. We only present results of the twelve participants in the rest of the paper.

Analysis of CAS's logs revealed that the participants spent a total of 31.3 (13.4) hours on average interacting with the system. Six and a half hours (4.34) of that was spent interacting with the "ontology view". The majority of the time in the "ontology view" was spent developing ontologies. There was a very high variance in the total interaction time, which can be attributed to each individual's ability.

The participants used the textual editors that were available under the "domain model editor" tab to modify/add domain model components required for WETAS, including problems and their solutions, syntax and semantic constraints and macros. The participants spent a mean total of 24.7 (9.6) hours interacting with the textual editors.

	Constraints		*Coverage*		*Completeness*	
	Syntax	Semantic	Syntax (8)	Semantic (13)	Syntax	Semantic
S1	5	12	5	7	63%	54%
S2	5	13	5	12	63%	92%
S3	4	12	4	12	50%	92%
S4	16	16	5	12	63%	92%
S5	14	18	8	13	100%	100%
S6	15	11	5	12	63%	92%
S7	2	5	3	3	38%	23%
S8	8	13	7	4	88%	31%
S9	5	8	4	4	50%	31%
S10	7	11	5	12	63%	92%
S11	4	18	5	12	63%	92%
S12	9	16	6	1	75%	8%
Mean	7.83	12.75	5.17	8.67	64.58%	66.67%
S.D.	4.73	3.89	1.34	4.5	16.71%	34.61%

Table 1: Total Numbers of Constraints Composed by Participants

In order to evaluate the completeness of participants' constraint bases, we manually compiled an "ideal" set of constraints for the domain, containing eight syntax constraints and 13 semantic constraints. The syntax constraints ensure that each component of a solution (such as the LCD and converted fractions) is in the correct format, and the correct problem-solving procedure is followed. The semantic constraints ensure that each component of a solution is correctly defined.

Table 1 lists the total numbers of constraints (syntax and semantic) composed by the participants under the "Constraints" column. The total numbers of "ideal" constraints covered by each constraint base are listed under the “Coverage” column. The completeness of each constraint base, calculated as the percentage of constraints accounted for by each constraint base, is given under the “Completeness” column. The participants accounted for five syntax constraints in the “ideal” set (65%) and nine semantic constraints (66%). Only one participant (S5) produced all the necessary constraints. The majority of the others had accounted for over half of the desired constraints. One participant (S12) struggled with composing semantic constraints and only managed to account for one desired constraint.

The “ideal” set of constraints contains five syntax constraints for verifying the syntactic validity of inputs, such as whether the LCD is an integer and whether the entered fractions are syntactically valid. They need to be verified due to the generic nature of the interface the students were told to use, which consisted of a single input box to input a fraction. For example, constraints are required to verify that fractions are of the ``numerator / denominator" form. However, these constraints are redundant for the solution-composition interface produced by CAS from a complete ontology. It contains two text boxes for inputting a fraction (one for its numerator and the other for its denominator), ensuring that a fraction is of the correct format. Furthermore, these input boxes only accept values of the type specified for the relevant property in the ontology (numerator and denominator would both be defined as integer in this case). As a consequence, constraints such as the ones to verify whether the specified numerator is an integer are also redundant.

	Constraints		Coverage		Completeness	
	Syntax	Semantic	Syntax (3)	Semantic (13)	Syntax	Semantic
S1	7	15	3	12	100%	92%
S2	8	12	3	12	100%	92%
S3	18	0	3	0	100%	
S4	6	23	3	1	100%	8%
S5	9	8	3	12	100%	92%
S6	0	23	0	1		8%
S7	13	26	3	1	100%	8%
S8	6	0	3	0	100%	
S9	11	23	3	1	100%	8%
S10	9	12	3	12	100%	92%
S11	7	12	3	12	100%	92%
S12	17	42	2	1	67%	8%
Mean	9.25	16.33	2.67	5.42	97.00%	49.97%
S.D.	4.97	11.86	0.89	5.82	9.95%	44.30%

Table 2: Total Numbers of Constraints Generated by CAS

The participants were free to make any modifications to the initial set of constraints produced by CAS, including deleting all of them and composing a new set manually. For that reason, we also analysed the constraints produced by CAS automatically, from the domain information supplied by the author. The generated constraint sets were analysed to calculate their completeness (Table 2). CAS generated the three syntax constraints necessary for CAS’s solution interface for 10 participants. There was one situation where just two required constraints were generated (S12), as the result of an incorrectly specified solution structure. The syntax constraints generator failed produce any constraints for participant S6 due to a bug.

CAS only has the ability to generate 12 out of the 13 required semantic constraints, as it is unable to generate constraints that require algebraic functionality. CAS cannot generate a constraint that accounts for common multiples of the two denominators larger than the lowest common multiple. So the maximum degree of completeness that can be expected is 92%.

CAS generated the maximum possible 12 semantic constraints from domain-dependant components supplied by five participants. However, it was not successful in generating constraints for the remaining participants. Further analysis revealed that there were two main reasons. One of the reasons was that two of the participants (S4 and S9) had unknowingly added empty duplicate solutions for problems, which resulted in constraints that allowed empty solutions. This situation can be avoided easily by restricting the solution interface not to save empty solutions.

Another common cause for not generating useful semantic constraints was an incomplete ontology. Four participants (S4, S6, S7 and S12) modelled the *Fraction* concept with only a single property of type *String*. This results in a set of constraints that compare each component of the student solution against the respective ideal solution component as a whole. These constraints are not of the correct level of granularity. Consequently, the resulting feedback is limited in pedagogical significance. For example, the constraints would have the ability to denote that the student has made an error in the sum, but not able to pinpoint whether the mistake is in the numerator or the denominator. We believe that the decision to model the *Fraction* concept with a single property may have been influenced by the student interface that we required them to use. The participants may have attempted to produce an ontology and solution structure that is consistent with this particular student interface.

The constraint generator failed to produce any semantic constraints for two participants, S3 and S8. It failed to generate constraints for S3 due to a bug in the system. The other participant (S8) did not add any solutions, and therefore semantic constraints could not be generated.

```
a. Relevance:    Fraction-1 component of IS has a (?var1, ?*) ‘Improper fraction’
   Satisfaction: Fraction-2 component of SS has a (?var1, ?*) ‘Improper fraction’
b. Relevance:    (match IS Fraction-1 ("2." ?var1 ?IS-var2 ?*))
   Satisfaction: (match SS Fraction-1 ("2." ?var1 ?SS-var2 ?*))
```

Figure 2: An example of translating a pseudo-code constraint into WETAS language

Although we assumed that the generated high-level constraints assisted the participants, there was little evidence in their reports that supported this assumption. Only one participant indicated that the generated constraints assisted him. Since no explanation on the high-level constraint representation was provided to the participants, they may have struggled to understand the notation and to find commonalities between the two representations. For example, Figure 2a shows the pseudo-code representation of a constraint that ensures that the numerator of first fraction specified by the student (SS) is the same as the corresponding value in the ideal solution (IS). The equivalent constraint in the WETAS language is given in Figure 2b.

3. Discussion

Analysing the results from the evaluation study confirmed all our hypotheses. Our first hypothesis about CAS being effective has been confirmed in previous work [7, 9]. The 2006 study revealed that CAS was able to generate all the required syntax constraints for 10 of the 12 participants. Furthermore, CAS generated over 90% of the semantic constraints for half of the participants. Considering that the participants were given very little training in using the authoring system, the results are very encouraging. Providing the users with more training and improving CAS to be fully integrated with a tutoring server (similar to WETAS) would further increase its effectiveness.

The second hypothesis claims that CAS requires less effort than composing constraints manually. In order to obtain a measure for the effort required for producing constraints using CAS, we calculated the average time required for producing a single constraint. Only the participants whose domain model components resulted in generating near complete constraint bases were used for calculating the average effort, to ensure that incorrectly generated constraints were not accounted. Five participants (S1, S2, S5, S10, S11) spent a total of 24.8 hours composing the required domain-dependant information. They also spent a total of 115.04 hours interacting with the textual editors to produce a total of 107 constraints. Consequently, the participants spent a total of 1.3 hours on average to produce one constraint.

The average time of 1.3 hours per constraint is very close to the 1.1 hours per constraint reported by Mitrovic [4] for composing constraints for SQL-Tutor. The time estimated by Mitrovic can be considered as biased since she is an expert of SQL and knowledge engineering, in addition to being an expert in composing constraints. Therefore, the achievement by novice ITS authors producing constraints in a time similar to the time reported by Mitrovic is significant. Furthermore, the time of 1.3 hours is a significant improvement from the two hours required by a similar study reported in [8]. Although that study involved composing constraints for the domain of adjectives in the English language, the overall complexities of the tasks are similar. Further, after the experiment was completed, it was discovered that the setup of WETAS was not optimal, which led the participants to perform additional effort when writing the final constraints. The figure of 1.3 hours per constraint is therefore probably pessimistic.

Although CAS currently generates constraints in a high-level language, it can be extended to generate constraints directly in the language required for execution. Assuming the generated constraints were produced in the required runnable form, the total of 99 syntax and semantic constraints were produced from 24.8 hours spent on composing the required domain information. Consequently, the participants would only require an average of 15 minutes (0.25 hours) to generate one constraint.

An average of 15 minutes to produce a constraint is a significant improvement from 1.1 hours reported by Mitrovic [4]. The time is more significant as the authoring process was driven by novice ITS authors. However, this does not take into account validating the generated constraints. As the constraint generation algorithm may not produce all the required constraints, the domain author may also be required to modify the generated constraints or add new constraints manually.

Finally, the third hypothesis concentrates on the sensitivity of the constraint generation on the quality of the ontology. Developing a domain ontology is a design task, which depends on the creator's perceptions of the domain. The ontologies developed by two users, especially if the domain is complicated, are very likely to be different. The constraint learning algorithm generated constraints using ontologies developed by the 12 participants. Although the ontologies were different, the syntax constraint generation algorithm managed to produce full constraint sets for almost all participants. However, the semantic constraint generation was more sensitive to the ontology. In particular, it was reliant on defining the *Fraction* concept with enough details. The semantic constraint generator managed to produce 92% complete constraint sets of the correct granularity for ontologies with a correctly defined

Fraction concept. On the contrary, the constraints generated for ontologies with a partially defined *Fraction* concept were too general. They compared each fraction composed by students as a whole against its corresponding fraction in the ideal solution. These constraints result in feedback limited in pedagogical significance.

4. Conclusions

This paper reports on an evaluation study of CAS, an authoring system for constraint-based tutors. The evaluation study conducted with novice authors produced very encouraging results. The syntax constraint generator was extremely effective, producing complete constraint sets for almost every participant. The semantic constraints generator produced constraint sets that were over 90% accurate for half of the participants. Although the constraints produced by the generator for the remaining participants were too general, ITS authors can modify them with little effort to produce constraints of correct granularity. We believe that this would contribute towards the reduction of the author's total workload.

At the time of the evaluation, constraints produced by CAS were not runnable, but the system can be easily extended to produce constraints in the target language. This would dramatically reduce the effort required for composing constraints. The evaluation revealed that the total workload required by a novice to generate a single constraint using CAS would be even less than the time required by experts to write them by hand.

We intend to enhance CAS further to generate constraints directly in the runnable form. CAS should also be fully integrated with WETAS and made consistent with its internal representation. We also plan to conduct further evaluations on the effectiveness of CAS's constraint generation.

References

1. Aleven, V., McLaren, B., Sewall, J. and Koedinger, K., The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. in *ITS 2006*, (Taiwan, 2006), Springer-Verlag, 61-70.
2. Martin, B. and Mitrovic, A., Domain Modeling: Art or Science? in *AIED 2003*, (Sydney, Australia, 2003), IOS Press, 183-190.
3. Martin, B. and Mitrovic, A., WETAS: a Web-Based Authoring System for Constraint-Based ITS. in *AH 2002*, (Malaga, Spain, 2002), LCNS, 543-546.
4. Mitrovic, A., Koedinger, K. and Martin, B., A comparative analysis of cognitive tutoring and constraint-based modeling. in *UM 2003*, (Pittsburgh, USA, 2003), Springer-Verlag, 313-322.
5. Mitrovic, A., Mayo, M., Suraweera, P. and Martin, B., Constraint-based Tutors: a Success Story. in *14th Int. Conf on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, (Budapest, 2001), Springer-Verlag, 931-940.
6. Murray, T. Expanding the Knowledge Acquisition Bottleneck for Intelligent Tutoring Systems. *International Journal of Artificial Intelligence in Education*, *8*. 222-232.
7. Suraweera, P., Mitrovic, A. and Martin, B., A Knowledge Acquisition System for Constraint-based Intelligent Tutoring Systems. in *AIED 2005*, (Amsterdam, Netherlands, 2005), IOS Press, 638-645.
8. Suraweera, P., Mitrovic, A. and Martin, B., The role of domain ontology in knowledge acquisition for ITSs. in *ITS 2004*, (Maceio, Brazil, 2004), Springer, 207-216.
9. Suraweera, P., Mitrovic, A. and Martin, B., The use of ontologies in ITS domain knowledge authoring. in *SWEL '04, ITS 2004*, (Maceio, Brazil, 2004), 41-49.
10. Tecuci, G. *Building Intelligent Agents: An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies*. Academic press, 1998.

Appendix G

Relative Contributions to Published Papers

Regulation 8(c) of the Degree of Doctor of Philosophy section in the 2000 University Calender states that "*where the published work has more than one author it shall be accompanied by a statement signed by the candidate identifying the candidates own contribution.*" The contributions are as follows.

1. The use of ontologies in ITS domain knowledge authoring [Suraweera et al. 2004*b*]

   This was my own research. The authoring tool that supports ontologies was designed and implemented by myself. I carried out the evaluation and analysed the results. Dr. Mitrovic provided advice. The paper was reviewed by Dr. Mitrovic and Dr. Martin.

2. The role of domain ontology in knowledge acquisition for ITSs [Suraweera et al. 2004*a*]

   This was my own research. The algorithm for generating constraints from the information available in an ontology was designed and implemented by myself with advice from Dr. Mitrovic. The paper was reviewed by Dr. Mitrovic and Dr. Martin.

3. Automatic Acquisition of Knowledge for Constraint-based Tutors [Suraweera 2004]

   This was my own research. Dr. Mitrovic reviewed the paper.

4. A Knowledge Acquisition System for Constraint-based Intelligent Tutoring Systems [Suraweera et al. 2005]

This was my own research. The algorithms for generating syntax and semantic constraints were designed and implemented by myself with the advice from Dr. Mitrovic. The paper was reviewed by Dr. Mitrovic and Dr. Martin.

5. Authoring constraint-based tutors in ASPIRE [Mitrovic, Suraweera, Martin, Zakharov, Milik & Holland 2006]

   The constraint generation algorithms of ASPIRE was based on my doctoral research. I was a major co-author of this paper.

6. Constraint Authoring System: An Empirical Evaluation [Suraweera, Mitrovic & Martin 2007]

   This was my own research. The Constraint Authoring System was designed and implemented by myself. The reported evaluation study was also conducted by myself under the guidance of Dr. Mitrovic. The paper was reviewed by Dr. Mitrovic and Dr. Martin.

Signed __________________ Date __________________

References

Aleven, V. & Koedinger, K. [2000], Limitations of student control: Do students know when they need help, *in* G. Gauthier, C. Frasson & K. VanLehn, eds, '5th International Conference on Intelligent Tutoring Systems', Springer, Montreal, pp. 292–303.

Alexe, C. & Gescei, J. [1996], A learning environment for the surgical intensive care unit, *in* C. Frasson, G. Gauthier & A. Lesgold, eds, 'Third International Conference on Intelligent Tutoring Systems', Montreal, pp. 439–447.

Altova XML, Data Management, UML, and Web Services Tools [2005], http://www.altova.com.

Anderson, J. [1993], *Rules of the Mind*, Erlbaum, Hillsdale, NJ.

Anderson, J. R., Corbett, A., Koedinger, K. & Pelletier, R. [1996], 'Cognitive tutors: Lessons learned', *Journal of the Learning Sciences* **4**(2), 167–207.

Angros, R., Johnson, W. L., Rickel, J. & Scholer, A. [2002], Learning domain knowledge for teaching procedural skills, *in* 'First international joint conference on Autonomous agents and multiagent systems', ACM Press, Bologna, Italy, pp. 1372–1378.

Ayscough, P. B. [1977], 'CALCHEMistry', *British Journal of Education Technology* **8**(3), 201–213.

Baghaei, N. & Mitrovic, A. [2006*a*], A constraint-based collaborative environment for learning uml class diagrams, *in* M. Ikeda, K. Ashley & T.-W. Chan, eds, 'Intelligent Tutoring Systems 2006', Springer, Jhongli, Taiwan, pp. 176–186.

Baghaei, N. & Mitrovic, A. [2006*b*], A constraint-based collaborative environment for learning UML class diagrams, *in* M. Ikeda, K. Ashley & T.-W. Chan, eds, 'Intelligent Tutoring Systems 2006', Jhongli, Taiwan, pp. 176–186.

Baghaei, N., Mitrovic, A. & Irwin, W. [2006], 'Problem-solving support in a constraint-based intelligent system for unified modelling language', *Technology, Instruction, Cognition and Learning Journal* **4**(1-2), To appear.

Bechhofer, S., Horrocks, I., Goble, C. & Stevens, R. [2001*a*], OilEd: a reasonable ontology editor for the semantic web, *in* '14th International Workshop on Description Logics, DL2001', Stanford, USA, p. 396408.

Bechhofer, S., Horrocks, I., Goble, C. & Stevens, R. [2001*b*], OilEd: a reason-able ontology editor for the semantic web, *in* 'KI2001, Joint German/Austrian conference on Artificial Intelligence', Springer-Verlag, Vienna, pp. 396–408.

Blessing, S. B. [1997], 'A programming by demonstration authoring tool for model-tracing tutors', *International Journal of Artificial Intelligence in Education* **8**, 233–261.

Bloom, B. S. [1984], 'The 2-sigma problem: The search for methods of group instruction as effective as one-to-one tutoring', *Educational Researcher* **13**, 4–16.

Brown, J. S., Burton, R. R. & Bell, A. G. [1975], 'SOPHIE: A step toward creating a reactive learning environment', *International Journal of Man-Machine Studies* **7**(5), 675–696.

Chen, P. [1976], 'The entity relationship model - toward a unified view of data', *ACM Transactions Database Systems* **1**(1), 9–36.

Clancey, W. [1982], Tutoring Rules for Guiding a Case Method Dialogue, Academic Press, Cambridge, Mass.

Clutterbuck, P. [1990], *The art of teaching spelling: a ready reference and classroom active resource for Australian primary schools*, Longman Australia Pty Ltd, Melbourne.

Corbett, A. & Anderson, J. [1995], 'Model tracing: Modeling the acquisition of procedural knowledge', *User Modeling and User-Adapted Interaction* **4**, 253–278.

Corbett, A. T., Trask, H. J., Scarpinatto, K. C. & Hadley, W. S. [1998], A formative evaluation of the PACT Algebra II tutor: Support for simple hierarchical reasoning, *in* B. P. Goettl, H. M. Halff, C. L. Redfield & V. J. Shute, eds, '4th International Conference on Intelligent Tutoring Systems', San Antonio, Texas, pp. 374–383.

CTAT [2005], 'Cognitive tutor authoring tools', http://ctat.pact.cs.cmu.edu/tiki-index.php.

DAML [2006], 'DARPA agent markup language', http://www.daml.org.

Dillenbourg, P. & Self, J. A. [1992], People power: a human-computer collaborative learning system, *in* C. Frasson, G. Gauthier & G. McCalla, eds, 'Second International Conference on Intelligent Tutoring Systems', Springer-Verlag, Montreal, pp. 651–660.

Elmasri, R. & Navathe, S. B. [2003], *Fundamentals of Database Systems, Fourth Edition*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gruber, T. R. [1993], 'A translation approach to portable ontologies', *Knowledge Acquisition* **5**(2), 199–220.

Holt, P., Dubs, S., Jones, M. & Greer, J. [1994], 'The state of student modelling', pp. 3–35.

Jarvis, M. P., Nuzzo-Jones, G. & Heffernan, N. T. [2004], Applying machine learning techniques to rule generation in intelligent tutoring systems., *in* 'Intelligent Tutoring Systems 2004', pp. 541–553.

Knublauch, H. [2003], 'An AI tool for the real world: Knowledge modeling with Protégé', http://www.javaworld.com/javaworld/jw-06-2003/jw-0620-protege.html.

Koedinger, K., Aleven, V., Heffernan, N., McLaren, B. & Hockenberry, M. [2004], Openning the door to non-programmers: Authoring intelligent tutor behavior by demonstration, *in* J. Lester, R. Vicari & F. Paraguacu, eds, 'Intelligent Tutoring Systems 2004', Springer, Maceio, Brazil, pp. 162–174.

Koedinger, K. R., Anderson, J., Hadley, W. & Mark, M. A. [1997], 'Intelligent tutoring goes to school in the big city', *International Journal of Artificial Intelligence in Education* **8**(1), 30–43.

Kulik, J. A., Kulik, C.-L. C. & Cohen, P. A. [1980], 'Effectiveness of computer-based college teaching: a meta-analysis of findings', *Rev. Educ. Research* **50**, 524–44.

Last, R. W. [1979], 'The role of computer-assisted learning in modern language teaching', *Assoc. for Literary and Linguistic Computing* **7**, 165–171.

Lesgold, A. [1987], Toward a theory of curriculum for use in designing intelligent instructional systems, *in* H. Mandl & A. M. Lesgold, eds, 'Learning Issues for Intelligent Tutoring Systems', Springer-Verlag, New York, pp. 114–137.

Martin, B. [2002], Intelligent Tutoring Systems: The Practical Implementation of Constraint-based Modelling, Phd thesis, University of Canterbury.

Martin, B. & Mitrovic, A. [2002*a*], Authoring web-based tutoring systems with WETAS, *in* Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson & C.-H. Lee, eds, 'International Conference on Computers in Education 2002', Auckland, NZ, pp. 183–187.

Martin, B. & Mitrovic, A. [2002*b*], WETAS: a web-based authoring system for constraint-based ITS, *in* '2nd Int. Conf on Adaptive Hypermedia and Adaptive Web-based Systems AH 2002', Vol. 2347, LCNS, Malaga, Spain, pp. 543–546.

Martin, B. & Mitrovic, A. [2003], Domain modeling: Art or science?, *in* U. Hoppe, F. Verdejo & J. Kay, eds, '11th Int. Conference on Artificial Intelligence in Education AIED 2003', IOS Press, Sydney, Australia, pp. 183–190.

Martin, B. & Mitrovic, A. [2005], Using learning curves to mine student models, *in* '10th international conference on User Modelling, UM05', Edinburgh, pp. 79–88.

Martin, B. & Mitrovic, A. [2006], The effect of adapting feedback granularity in ITS, *in* V. Wade, H. Ashman & B. Smyth, eds, '4th International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems, AH2006', Springer, Dublin Ireland, pp. 192–202.

Mayo, M., Mitrovic, A. & McKenzie, J. [2000], CAPIT: An intelligent tutoring system for capitalisation and punctuation, *in* Kinshuk, C. Jesshope & T. Okamoto, eds, 'Advanced Learning Technology: Design and Development Issues', IEEE Computer Society, Los Alamitos, CA, pp. 151–154.

Milik, N., Marshall, M. & Mitrovic, A. [2006], Teaching logical database design in ERM-Tutor, *in* M. Ikeda, K. Ashley & T.-W. Chan, eds, 'Intelligent Tutoring Systems 2006', Springer, Jhongli, Taiwan, pp. 707–709.

Mitrovic, A. [1998*a*], Experiences in implementing constraint-based modelling in SQL-Tutor, *in* B. P. Goettl, H. M. Halff, C. L. Redfield & V. J. Shute, eds, '4th International Conference on Intelligent Tutoring Systems', San Antonio, pp. 414–423.

Mitrovic, A. [1998*b*], Learning SQL with a computerised tutor, *in* '29th ACM SIGCSE Technical Symposium', Atlanta, pp. 307–311.

Mitrovic, A. [2002], NORMIT, a web-enabled tutor for database normalization, *in* Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson & C. H. Lee, eds, 'International Conference on Computers in Education 2002', Auckland, New Zealand, pp. 1276–1280.

Mitrovic, A. [2003*a*], 'An intelligent SQL tutor on the web', *International Journal of Artificial Intelligence in Education* **13**, 171–195.

Mitrovic, A. [2003*b*], Supporting self-explanation in a data normalization tutor, *in* V. Aleven, U. Hopppe, J. Kay, R. Mizoguchi, H. Pain, F. Verdejo & K. Yacef, eds, 'Supplementary proceedings, AIED 2003', pp. 565–577.

Mitrovic, A. [2005*a*], The effect of explaining on learning: a case study with a data normalization tutor, *in* C.-K. Looi, G. McCalla, B. Bredeweg & J. Breuker, eds, 'Artificial Intelligence in Education AIED 2005', IOS Press, pp. 499–506.

Mitrovic, A. [2005*b*], 'Scaffolding answer explanation in a data normalization tutor', *Facta Universitatis, Series Elec. Energ.* **18**(2), 151–163.

Mitrovic, A., Koedinger, K. & Martin, B. [2003], A comparative analysis of cognitive tutoring and constraint-based modeling, *in* P. Brusilovsky, A. Corbett & F. d. Rosis, eds, '9th International conference on User Modelling UM2003', Springer-Verlag, Pittsburgh, USA, pp. 313–322.

Mitrovic, A., Mayo, M., Suraweera, P. & Martin, B. [2001], Constraint-based tutors: a success story, *in* L. Monostori, J. Vancza & M. Ali, eds, '14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-2001)', Springer-Verlag, Budapest, pp. 931–940.

Mitrovic, A. & Ohlsson, S. [1999], 'Evaluation of a constraint-based tutor for a database language', *International Journal of Artificial Intelligence in Education* **10**(3-4), 238–256.

Mitrovic, A., Suraweera, P., Martin, B. & Weerasinghe, A. [2004], 'DB-suite: Experiences with three intelligent, web-based database tutors', *Journal of Interactive Learning Research (JILR)* **15**(4), 409–432.

Mitrovic, A., Suraweera, P., Martin, B., Zakharov, K., Milik, N. & Holland, J. [2006], Authoring constraint-based tutors in aspire, *in* M. Ikeda, K. Ashley & T.-W. Chan, eds, 'Intelligent Tutoring Systems 2006', Springer, Jhongli, Taiwan, pp. 41–50.

Munro, A., Johnson, M. C., Pizzini, Q. A., Surmon, D. S., Towne, D. M. & Wogulis, J. L. [1997], Authoring simulation-centred tutors with RIDES, *in* 'International Journal of Artificial Intelligence in Education', Vol. 8, pp. 284–316.

Murray, T. [1997], 'Expanding the knowledge acquisition bottleneck for intelligent tutoring systems', *International Journal of Artificial Intelligence in Education* **8**, 222–232.

Murray, T. [1999], 'Authoring intelligent tutoring systems: an analysis of the state of the art', *International Journal of Artificial Intelligence in Education, Part II of the Special Issue on Authoring Systems for Intelligent Tutoring Systems* **10**, 98–129.

Murray, T. [2003], 'An overview of intelligent tutoring system authoring tools: Updated analysis of the state of the art', *Authoring tools for advanced technology learning environments* pp. 491–545.

Noy, N. F., Sintek, M., Decker, S., Crubézy, M., Fergerson, R. W. & Musen, M. A. [2001], 'Creating semantic web contents with Protégé-2000', *IEEE Intelligent Systems* **16**(2), 60–71.

Ohlsson, S. [1987], 'Some principles of intelligent tutoring', *Artificial intelligence and education; vol. 1: learning environments and tutoring systems* pp. 203–237.

Ohlsson, S. [1994], Constraint-based student modelling, *in* 'Student Modelling: the Key to Individualized Knowledge-based Instruction', Springer-Verlag, Berlin, pp. 167–189.

Ohlsson, S. [1996], 'Learning from performance errors', *Psychological Review* **103**(2), 241–262.

O'Shea, T. & Self, J. A. [1983], *Learning and Teaching with Computers*, Brighton, Harvester Press.

OWL [2004], 'OWL web ontology language', http://www.w3.org/TR/owl-features.

Protege [2006], 'The Protégé ontology editor and knowledge acquisition system, http://protege.stanford.edu/'.

RDF [2006], 'Resource description framework (RDF)', http://www.w3.org/RDF.

Rich, E. [1989], 'Stereotypes and User models', *User Models in Dialog Systems* pp. 35–51.

Shute, V. J., Glaser, R. & Raghavan, K. [1989], 'Inference and discovery in an exploratory laboratory', *Learning and individual differences: Advances in theory and research* pp. 279–326.

Soloway, E., Guzdial, M. & Hay, K. [1994], 'Learner-centered design: the challenge for HCI in the 21st century', *interactions* **1**(2), 36–48.

Storey, M., Musen, M., Silva, J., Best, C., Ernst, N. & Noy, R. F. N. [2001], 'Jambalaya: Interactive visualization to enhance ontology authoring and knowledge acquisition in Protégé'.

Suraweera, P. [2004], Automatic acquisition of knowledge for constraint-based tutors, *in* 'Student Track, Intelligent Tutoring Systems 2004', Maceio, Brazil.

Suraweera, P. & Mitrovic, A. [2002], KERMIT: a constraint-based tutor for database modeling, *in* S. Cerri, G. Gouarderes & F. Paraguacu, eds, 'Intelligent Tutoring Systems 2002', Biarritz, France, pp. 377–387.

Suraweera, P. & Mitrovic, A. [2004], 'An intelligent tutoring system for entity relationship modelling', *International Journal of Artificial Intelligence in Education* **14**(3,4), 375–417.

Suraweera, P., Mitrovic, A. & Martin, B. [2004*a*], The role of domain ontology in knowledge acquisition for ITSs, *in* J. Lester, R. Vicari & F. Paraguaçu, eds, 'Intelligent Tutoring Systems 2004', Springer, Maceio, Brazil, pp. 207–216.

Suraweera, P., Mitrovic, A. & Martin, B. [2004*b*], The use of ontologies in ITS domain knowledge authoring, *in* J. Mostow & P. Tedesco, eds, '2nd Int. Workshop on Applications of Semantic Web for E-learning SWEL'04, Intelligent Tutoring Systems 2004', Maceio, Brazil, pp. 41–49.

Suraweera, P., Mitrovic, A. & Martin, B. [2005], A knowledge acquisition system for constraint-based intelligent tutoring systems, *in* C.-K. Looi, G. McCalla, B. Bredeweg & J. Breuker, eds, 'Artificial Intelligence in Education 2005', IOS Press, Amsterdam, Netherlands, pp. 638–645.

Suraweera, P., Mitrovic, A. & Martin, B. [2007], Constraint authoring system: An empirical evaluation, *in* 'Artificial Intelligence in Education 2007', IOS Press, California, USA, p. to appear.

Tecuci, G. [1998], *Building Intelligent Agents: An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies*, Academic press.

Tecuci, G., Boicu, M., Marcu, D., Stanescu, B., Boicu, C. & Comello, J. [2002], 'Training and using Disciple agents: A case study in the military center of gravity analysis domain', *AI Magazine* **23**(4), 51–68.

Tecuci, G. & Keeling, H. [1999], 'Developing an intelligent educational agent with Disciple', *International Journal of Artificial Intelligence in Education* **10**, 221–237.

Tecuci, G., Wright, K., Lee, S., Boicu, M. & Bowman, M. [1998], A learning agent shell and methodology for developing intelligent agents, *in* 'AAAI-98 Workshop: Software Tools for Developing Agents', Madison, Wisconsin.

van Lent, M. & Laird, J. E. [2001], Learning procedural knowledge through observation, *in* 'International conference on Knowledge capture', ACM Press, pp. 179–186.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J., Shelby, R., L., T., Treacy, D., Weinstein, A. & Wintersgill, M. [2005], 'The Andes physics tutoring system: Lessons learned', *International Journal of Artificial Intelligence and Education* **15**(3), 147–204.

Weerasinghe, A. & Mitrovic, A. [2003], Effects of self-explanation in an open-ended domain, *in* U. Hoppe, F. Verdejo & J. Kay, eds, '11th Int. Conference on Artificial Intelligence in Education AIED 2003', IOS Press, pp. 512–514.

Wikipedia [2005], 'Ontology (computer science)', http://en.wikipedia.org/wiki/Ontology_(computer_science).

Zakharov, K., Ohlsson, S. & Mitrovic, A. [2005], Feedback micro-engineering in EER-Tutor, *in* C.-K. Looi, G. McCalla, B. Bredeweg & J. Breuker, eds, 'Artificial Intelligence in Education AIED 2005', IOS Press, Amsterdam, Netherlands, pp. 718–725.