ASPIRE: An Authoring System and Deployment Environment for Constraint-Based Tutors

Antonija Mitrovic, Brent Martin, Pramuditha Suraweera, Konstantin Zakharov, Nancy Milik, Jay Holland, *Intelligent Computer Tutoring Group, Department Computer Science and Software Engineering***,** *University of Canterbury, Private Bag 4800, Christchurch, New Zealand*
http://www.ictg.canterbury.ac.nz/
{tanja.mitrovic,brent.martin}@canterbury.ac.nz

Nicholas McGuigan, *Lincoln University, Lincoln, New Zealand*
mcguigan@lincoln.ac.nz

Abstract: Over the last decade, the Intelligent Computer Tutoring Group (ICTG) has implemented many successful constraint-based Intelligent Tutoring Systems (ITSs) in a variety of instructional domains. Our tutors have proven their effectiveness not only in controlled lab studies but also in real classrooms, and some of them have been commercialized. Although constraint-based tutors seem easier to develop in comparison to other existing ITS methodologies, they still require substantial expertise in Artificial Intelligence (AI) and programming. Our initial approach to making the development easier was WETAS (Web-Enabled Tutor Authoring System), an authoring shell that provided all the necessary functionality for ITSs but still required domain models to be developed manually. This paper presents ASPIRE (Authoring Software Platform for Intelligent Resources in Education), a complete authoring and deployment environment for constraint-based ITSs. ASPIRE consists of the authoring server (ASPIRE-Author), which enables domain experts to easily develop new constraint-based tutors, and a tutoring server (ASPIRE-Tutor), which deploys the developed systems. ASPIRE-Author supports the authoring of the domain model, in which the author is required to provide a high-level description of the domain, as well as examples of problems and their solutions. From this information, ASPIRE generates the domain model automatically. We discuss the authoring process and illustrate it using the development process of CIT, an ITS that teaches capital investment decision making. We also discuss a preliminary study of ASPIRE, and some of the ITSs being developed in it.

Keywords: authoring systems, ITS deployment environment, constraint-based ITSs, evaluation

INTRODUCTION

It is widely accepted that ITSs promise to revolutionize education, due to their high effectiveness in supporting learning (e.g., Koedinger, Anderson, Hadley, & Mark, 1997; Mitrovic & Ohlsson, 1999; VanLehn et al., 2005). Despite this promise, ITSs are still not common in classrooms because their development requires extensive expertise, effort and time (Murray, 1997).

We at ICTG have developed a number of successful constraint-based tutors over the last decade. Our approach to building ITSs is based on Ohlsson's theory of learning from performance errors (Ohlsson, 1996), which resulted in the methodology known as Constraint-Based Modeling (CBM) (Ohlsson, 1994; Mitrovic & Ohlsson, 1999). Ohlsson proposed that knowledge should be represented in the form of constraints, which specify what ought to be so, rather than generating problem-solving paths, as done in model-tracing tutors. Domain knowledge (i.e., constraints) is thus used as a way of prescribing abstract features of correct solutions. As we discussed elsewhere (Ohlsson & Mitrovic, 2007), constraints are easier to develop in contrast to production rules, which are used in model tracing as a recipe for performing tasks in a domain (e.g., Koedinger et al, 1997). Constraints support evaluation and judgment, not inference, and are used to represent both domain and student knowledge.

Although representing domain knowledge in the form of constraints makes the authoring process easier (Mitrovic, Koedinger, & Martin, 2003; Ohlsson & Mitrovic, 2007), building a knowledge base for a constraint-based tutor still remains a major challenge. Developing a constraint-based tutor, like any other ITS, is a labour-intensive process that requires expertise in 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.

This paper presents ASPIRE, an authoring and deployment environment that assists in the process of composing domain models for constraint-based tutors and automatically serves tutoring systems on the web. The proposed system builds upon our experiences with constraint-based tutors and WETAS (Martin & Mitrovic, 2002, 2003), 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 building the domain model, automating some of the tasks involved, and seamlessly deploys the resulting domain model to produce a fully-functional, web-based ITS.

Developing ITSs in ASPIRE is much easier than developing them manually, as ASPIRE automates much of the work involved. ASPIRE hides the details of constraint implementation, such as the constraint language and constraint structures, from authors. Rather than having to learn the constraint language and write constraints manually, in ASPIRE the author is only required to model the domain ontology and provide example problems and their solutions. ASPIRE generates constraints from the information the author provides automatically. Ontologies used in ASPIRE consist of concepts, described in terms of their properties and restrictions on properties, with relationships representing various associations among concepts. ASPIRE uses a machine-learning approach to induce constraints from the domain ontology and examples of solved problems, which we earlier investigated and developed in the Constraint Authoring System (CAS) (Suraweera, Mitrovic, & Martin, 2004a, 2004b, 2005, 2007). Whilst the constraint details are hidden from novices, expert authors can directly modify the generated constraints if necessary.

The paper commences with a brief introduction to related authoring systems for building ITSs. The next section presents ASPIRE's architecture, followed by a discussion of ASPIRE-Author, the outline of the domain authoring process and ASPIRE-Tutor. We also present CIT, a constraint-based tutor developed in ASPIRE, and the evaluation study we performed with it. Finally, the last section presents conclusions.

RELATED WORK

Murray (1999, 2003) classifies ITS authoring tools into two main groups: pedagogy-oriented and performance-oriented. Pedagogy-oriented systems focus on instructional planning and teaching strategies, assuming that the instructional content will be fairly simple (e.g., a set of instructional units containing canned text and graphics). Such systems provide support for curriculum sequencing and planning, authoring tutoring strategies, composing multiple knowledge-types (e.g., facts, concepts and procedures) and authoring adaptive hypermedia. On the other hand, performance-oriented systems focus on providing rich learning environments where students learn by solving problems and receiving dynamic feedback. The systems in this category include authoring systems for domain expert systems, simulation-based learning and some special purpose authoring systems focussing on performance.

ASPIRE would be classified as a performance-oriented authoring system under those criteria, and therefore in this paper we discuss only authoring systems with similar characteristics. Students typically use ITSs to solve problems and receive feedback customised to their attempts. ITSs have a deep model of expertise, enabling the tutor to both correct the student's errors and provide assistance on solving a problem. Authoring systems of this kind focus on generating the rules that form the domain model. They typically use sophisticated machine learning techniques for acquiring domain rules with the assistance of a domain expert. Examples include Diligent (Angros, Johnson, Rickel, & Scholer, 2002), Disciple (Tecuci, 1998), and Demonstr8 (Blessing, 1997). We now describe some of these approaches.

Diligent

Diligent (Angros et al., 2002) is an authoring system that acquires the knowledge required for a pedagogical agent in simulation-based learning environments. It learns the agent rules by observing the expert demonstrating the skill to be taught. The system experiments with the recorded traces in order to understand the role of each step in the procedure. The expert can directly modify learned procedures by providing clarifications at the completion of the experimentation stage. Authoring also involves acquiring the linguistic knowledge required to explain procedural tasks to students.

The domain expert can review the procedural rules generated by the system by examining a graph of the task model, or alternatively by allowing the agent to demonstrate its acquired knowledge by teaching it back to the domain expert. The expert can also directly modify the task model by adding or removing steps, ordering constraints and so on.

Diligent was evaluated on a series of tasks, revealing that the produced models were significantly better that those produced manually, and that Diligent's assistance was most beneficial with complex, error-prone procedures. Although the evaluation results are promising, Diligent is only suited for authoring knowledge in simulated domains where the consequences of each action are readily observed. It would, therefore, not be effective in environments such as medical diagnosis, where the diagnosis steps are determined from the patient's condition, rather than from the effects each step has on the patient.

Disciple

Disciple (Tecuci, 1998; Tecuci & Keeling, 1999; Tecuci, Wright, Lee, Boicu, & Bowman, 1998) is 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. The expert is also required to supervise and correct the agent's behaviour. Disciple uses 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 they solve problems.

Disciple first has to be customised by building two domain-specific interfaces and a problem solver for the domain. The first domain-specific interface provides the expert with a means of expressing their knowledge, while the second is the problem-solving interface for the student. The extent and the nature of the problem solver depend on the type and purpose of the learning agent being developed. Developing the interfaces and problem solver requires software engineering and knowledge engineering expertise, with a developer working collaboratively with a domain expert.

The Disciple agent has been applied to a number of domains including history, statistical analysis and engineering design. The task of customising the Disciple agent requires extensive programming skills and considerable effort: the authors must develop a problem-solving interface for the domain and a problem solver. Furthermore, building a problem solver in some domains may be extremely hard, if not impossible. The semantic network contains information about domain concepts as well as instances. The process of specifying the semantic network is highly repetitive and tedious because all elements participating in example solutions must be added manually. Furthermore, even for small domains the semantic networks tend to contain many links between nodes. Larger domains tend to have very complex semantic networks; locating nodes becomes very difficult and domain experts may even become disoriented.

Demonstr8

Demonstr8 (Blessing, 1997) is an authoring system that assists in the development of model-tracing tutors for arithmetic domains. The system infers production rules using programming-by-demonstration techniques, coupled with methods to further abstract the generated productions.

The author begins by creating a student interface using a drawing tool-like interface. All cells placed on the interface automatically become available as working memory elements (WMEs). The author then identifies higher-order WMEs (collections of atomic WMEs that have a significant meaning). The system relies on *knowledge functions* for any declarative knowledge not depicted directly in the interface. These functions are implemented as two-dimensional tables that enumerate the function’s value for all combinations of the two inputs.

The author then demonstrates the problem-solving procedure by solving example problems. The system selects the relevant knowledge function for each step by exhaustively going through the list of available functions to locate the one that produces the demonstrated result. In cases where multiple knowledge functions return the desired outcome, the system asks the author to select the correct one. After each author action, the system generates a production rule and displays it. The author can modify the rule by selecting a more general/specific condition from a drop-down list. The author is also required to specify a goal and a skill covered by the generated rule, and to provide four help messages with increasing levels of detail.

The production rule generation process is highly dependent on the WMEs (both base and higher order) the author created: for correct productions to be generated the author needs to use the correct representation for WMEs. Furthermore, each production rule generated by the system is displayed to the author for fine-tuning. In cases where a production rule needs to be decomposed into sub-goals, this also has to be directly specified by the author. To achieve these tasks successfully the author needs to be knowledgeable about the model-tracing approach. It seems unreasonable to assume typical educators (such as school teachers) would possess such knowledge.

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

CTAT

The Cognitive Tutor Authoring Tools (CTAT) (Aleven, McLaren, Sewall, & Koedinger, 2006; Koedinger, Aleven, Heffernan, McLaren, & Hockenberry, 2004) also assist the creation and delivery of model-tracing tutors. CTAT allows authors to create two types of tutors: Cognitive tutors and Example-Tracing Tutors (previously called Pseudo Tutors). Cognitive tutors contain a model capable of tracing student actions while solving problems. In contrast, an example-tracing tutor contains a fixed trace from solving one particular problem.

To develop an example-tracing tutor, the author starts by creating the student interface, followed by demonstrating correct and incorrect actions to be taken during problem solving. All actions performed by the author are visualised in the form of a behaviour graph. The author annotates the graph by adding hint messages to correct links and “buggy” messages to incorrect links. The author must also add labels to the links representing the skill behind each problem-solving step. These skill labels are used for generating a *skill matrix*, which outlines the knowledge elements required to solve each problem.

The ratio of development time to instructional time with CTAT is 23:1 on average, which compares favourably to the corresponding estimate of 200:1 for manually constructed, fully functional model-tracing tutors (Koedinger et al., 2004). Although in theory both example-tracing and cognitive tutors exhibit identical interaction with the student, an example-tracing tutor is specific to the problem used to produce it. While new problems can be added to a full cognitive tutor with little effort, example-tracing tutors require all possible problem-solving paths to be demonstrated for each new problem. This task becomes increasingly tedious as the number of similar problems increases and the complexity of alternate paths rises.

Jarvis and co-workers have implemented an automatic rule authoring system for CTAT tools (Jarvis, Nuzzo-Jones, & Heffernan, 2004) that generates Jess rules. Their goal is to generate the required production rules (given domain knowledge and examples of problem-solving steps) through programming-by-demonstration techniques. The rule generation system generalises the set of behaviour graphs developed while building an example-tracing tutor; the correct steps demonstrated by the expert are used as positive examples and incorrect steps are used as negative examples. It was tested in the domains of multi-column multiplication, fraction addition and tic-tac-toe. The rules for all three domains were learned in a reasonable amount of time, but were overly general in some cases (Jarvis, Nuzzo-Jones, & Heffernan, 2004). In this process, the author needs a thorough knowledge of model tracing. Furthermore, listing the complete set of skills required for the domain is a task that requires cognitive modelling experience. The rule generation engine is fully dependent upon the list of skills, so an incomplete skill-list would result in an incomplete set of generated rules. Furthermore, producing the skill-list in addition to the behaviour graph would be very demanding for the domain expert.

Matsuda, Cohen, Sewall, Lacerda, and Koedinger (2007) report on SimStudent, another machine-learning approach to generating production rules within CTAT. In SimStudent, the author demonstrates how a particular problem can be solved by performing a sequence of problem-solving steps. Each step is represented in terms of the focus of attention (i.e., the interface components being modified), the value(s) entered by the author, and the skill demonstrated. SimStudent then generates a production rule for each problem-solving step. This approach has been used to generate rules for arithmetic, algebra and chemistry, resulting in fairly good domain models. However, this approach cannot be used for non-procedural tasks.

THE OVERVIEW OF ASPIRE

The goal of ASPIRE is to simplify the process of developing and deploying constraint-based tutors. Our primary intention is to reduce the amount of background knowledge required from authors in order to develop ITSs.

ASPIRE[1] consists of ASPIRE-Author, the authoring server, and ASPIRE-Tutor, the tutoring server that delivers the resulting ITSs to students (see Figure 1). ASPIRE-Author makes it possible for the human expert (i.e., the author) to describe the instructional domain and the tasks students will be performing, and to specify problems and their solutions. Once an ITS has been specified in ASPIRE-Author, the tutoring server delivers the developed system to the student.

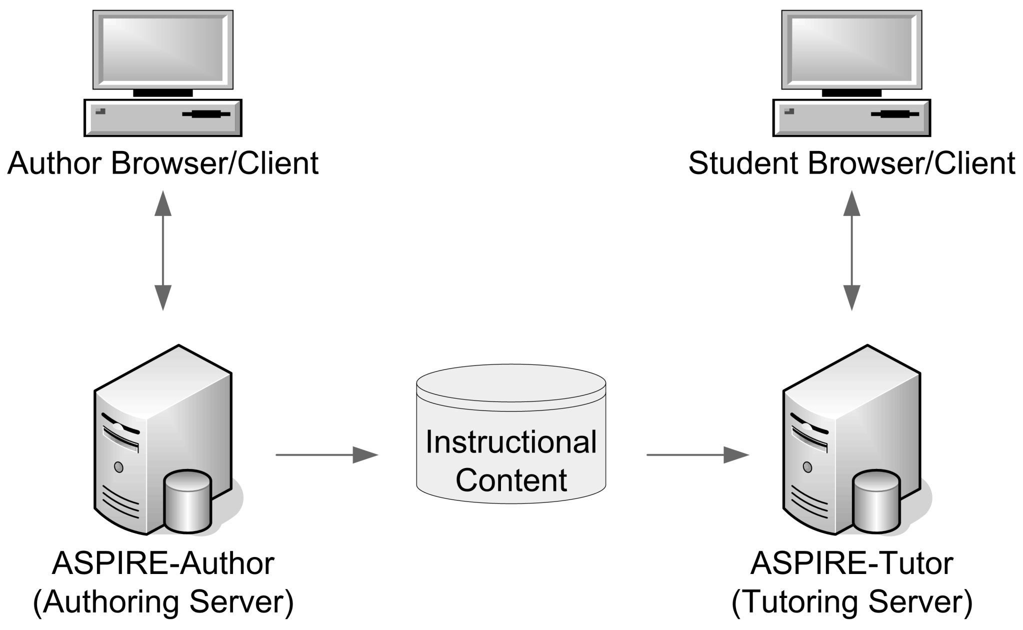


Fig. 1. The Architecture of ASPIRE

[1] In this paper, we discuss only some of ASPIRE's features and functionality. For the full discussion, please see ASPIRE User Manual available at http://aspire.cosc.canterbury.ac.nz

ASPIRE-AUTHOR

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 Table 1, consists of eight phases. Initially, the author specifies general features of the chosen instructional domain, such as whether it consists of sub-domains focusing on specific areas, and whether or not the task is procedural. For procedural tasks, the author describes the problem-solving steps. The author then develops the domain ontology using ASPIRE's ontology workspace. In the third phase, the author defines the problem structure and the general structure of solutions, expressed in terms of concepts from the ontology. The author then adds sample problems and their correct solutions. During this phase, the author is encouraged to provide multiple solutions for each problem, demonstrating different ways of solving it. ASPIRE can then generate syntax constraints by analysing the ontology and the solution structure. The semantic constraint generator analyses problems and their solutions to generate semantic constraints. Finally, the author can deploy the developed ITS.

Table 1. The phases of the authoring process

1. Specify the domain characteristics
2. Compose the domain ontology
3. Model the problem and solution structures
4. Design the student interface
5. Add problems and solutions
6. Generate syntax constraints
7. Generate semantic constraints
8. Deploy the tutoring system

The architecture of ASPIRE-Author is illustrated in FigureFig. –2. The *Authoring Controller* is the central component which manages the authoring process and communication between the various components of ASPIRE-Author. The *Domain Structure Modeller* supports the first phase of the authoring process by allowing the author to specify the general characteristics of the chosen instructional domain. This information is stored as the initial part of the domain model. The author then specifies the domain ontology using the *Ontology Workspace* (phase 2) and the structure of problems/solutions in the *Problem/Solution Structure Modeller* (phase 3). The *Student Interface Builder* supports the author in specifying the initial version of the student interface (phase 4), which will be used to communicate with students. The author uses the *Problem/Solution Editor* to provide examples of problems and their solutions (phase 5). The *Constraint Generator* uses the specified information to develop the domain knowledge necessary for the ITS to be able to analyse students' solutions (phases 6 and 7). This knowledge is represented in terms of constraints, which describe the syntax and the semantics of the instructional domain. Finally, the developed domain models are maintained by the *Domain Model Manager*.

Phase 1: Specifying the Characteristics of the Domain

We will illustrate the authoring procedure for the procedural task of adding fractions. The problem-solving procedure can be broken down into four steps. 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 necessary. Figure 3 shows the screenshot of the *Domain* tab of ASPIRE, in which the author has specified the problem-solving steps for this domain. Note that in this particular case the author wanted each step to be shown on a separate page.

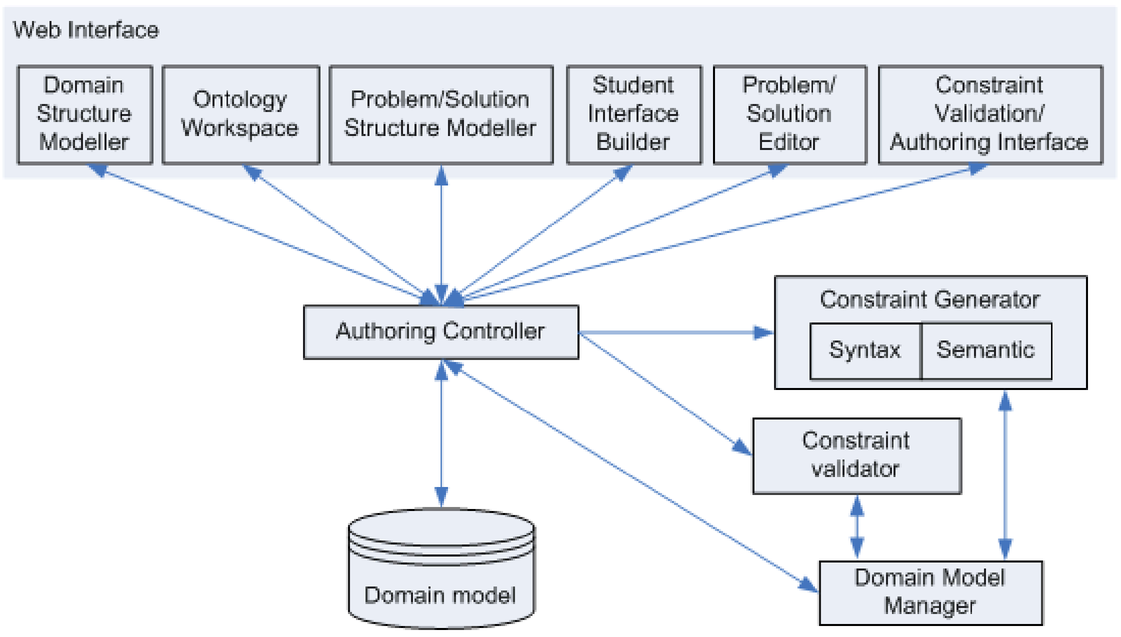


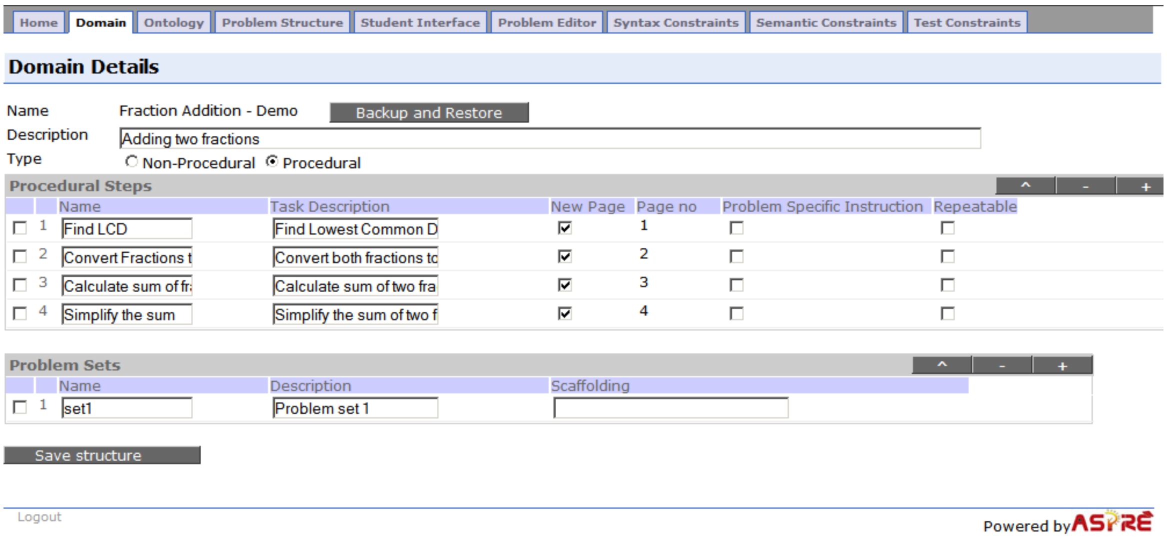
Fig. 2. The Architecture of ASPIRE-Author

The first authoring phase requires the author to identify the steps for the chosen instructional task. This is not a trivial task, as the author needs to decide on the pedagogical approach to teaching the task. For example, it is possible to come up with a different set of steps for adding two fractions in comparison to the one shown in Figure 3: the author may want to combine the initial two steps into one, or even combine the initial three tasks into one. The decisions would depend on the author's teaching approach and/or the target population of students. The author also needs to decide on how to structure the student interface: will the task be presented on the same page or on multiple pages?

In Figure 3, the author defined a sequence of steps, due to the nature of the task. Please note that even though there are four steps defined for the Fraction Addition domain, only some problems will require each step to be completed. If, for example, the two given fractions have the same denominator, the student will not have to do anything in the initial two steps. ASPIRE supports more complicated tasks as well, in addition to sequences of steps. Note the "*Repeatable*" check boxes: the author can use them to specify whether a particular step needs to be repeated many times.

Phase 2: Specifying the Domain Ontology

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. A domain ontology describes the domain by identifying important concepts and relationships between them. The ontology defines 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 (Suraweera, Mitrovic & Martin, 2004a). The study showed that ontologies help authors to reflect on the domain, organise constraints into meaningful categories and produce more complete constraint bases.



Domain Details

Name	Fraction Addition - Demo	
Description	Adding two fractions	
Type	Non-Procedural [✓ Procedural]	

Procedural Steps

		Name	Task Description	New Page	Page no	Problem Specific Instruction	Repeatable
☐	1	Find LCD	Find Lowest Common D	✓	1	☐	☐
☐	2	Convert Fractions t	Convert both fractions tc	✓	2	☐	☐
☐	3	Calculate sum of fr	Calculate sum of two fra	✓	3	☐	☐
☐	4	Simplify the sum	Simplify the sum of two f	✓	4	☐	☐

Problem Sets

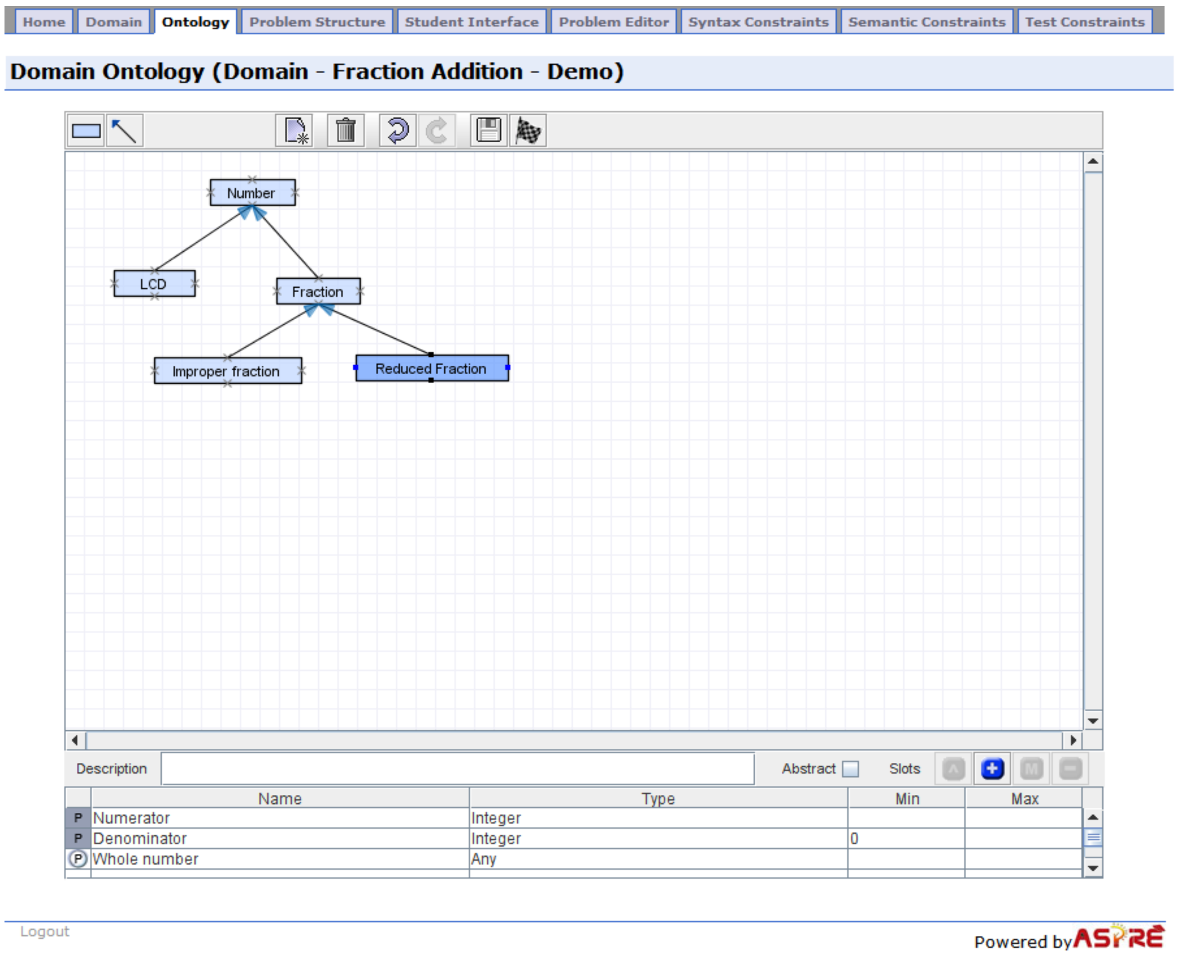
		Name	Description	Scaffolding
☐	1	set1	Problem set 1	

Fig. 3. Problem-solving procedure for fraction addition

In ASPIRE, ontologies are represented as hierarchies in which concepts are related via the *is-a* relationship. The Ontology Workspace (shown in Figure 4) is a graphical ontology-development tool, which supports a rich knowledge model. The taxonomy is represented as a set of concepts (rectangular boxes) connected with arrows, representing the *is-a* relationship. The bar at the top of the drawing area contains a set of tools that can be used to draw the ontology and manage it. The rectangle and arrow tools are used to draw the hierarchy (i.e., to draw concepts and relationships between them). In addition to these two tools, there is also a tool for starting a new ontology (the empty page tool), deleting the currently selected element of the ontology (the trash can tool), undoing/redoing the last action, saving the ontology (shown as diskette) and the finish tool, which has the effect of saving the ontology and leaving the ontology workspace (shown as the finish flag). Below this tool bar there is a drawing pane, where the domain hierarchy can be drawn.

To define the ontology the author must create the identified domain concepts and specify their properties and relationships. One possible ontology for the domain of adding fractions is illustrated in Figure 4. It contains *Number* as the most generic concept, which has two specialisations, *LCD* and *Fraction*. *Fraction* is further specialised into *Improper* and *Reduced*. The specialization/generalization relationships between domain concepts are visually represented as arrows between concepts.

Details of concepts, such as their properties and relationships with other concepts, are outlined in the bottom panel. Figure 4 shows the properties of the (currently selected) *Reduced Fraction* concept: *Numerator*, *Denominator* and *Whole Number*. The first two properties are inherited from the *Fractions* concepts, while the last one is defined locally, and may only occur if the resulting fraction needs to be simplified. 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. New properties and relationships can be added using the interface shown in Figure 5, which allows the specification of a default value for String and Boolean properties. It also allows the range of values for Integers and Floats to be specified in terms of a minimum and maximum. When creating a property of type Symbol the list of valid values must be specified.



Domain Ontology (Domain - Fraction Addition - Demo)

	Name	Type	Min	Max
P	Numerator	Integer		
P	Denominator	Integer	0	
Ⓟ	Whole number	Any		

Fig. 4. The ontology for fraction addition

Property values may be specified as unique, optional and/or multivalued. In the latter case, the exact number of values that a property may hold is specified using the ‘at least’ and *at most* fields of the property interface. The *at least* field specifies the minimum number of values a property may hold while the ‘at most’ field specifies the maximum number of such values.

The ontology workspace visualises only generalisation/specialization relationships. Other types of relationships, such as part-of relationships between concepts, can be specified, but are not shown visually. To add a relationship the author defines its name and then selects the other concept(s) involved. The resulting relationship holds between the concept initially selected in the graphical representation of the ontology and the concept(s) chosen in the relationship-composing interface. In some cases, a relationship may involve one of a set of concepts. For example, when specifying an assignment (a statement that assigns a value to a variable), the author may specify that the allowed concepts on the right-hand side are constants (e.g., "x = 1"), variables (e.g., "x = y"), functions (e.g., "x = max(a,b,c)") or arithmetic expressions (e.g., "x = y + 3"). The `List` box allows the author to specify such a case, and then the corresponding concepts can be added to the container by selecting the appropriate concept from the drop-down list and clicking the + button. Figure 6 shows the *assigned value* relationship, when the first related concept has been added (i.e., the *Number* concept).

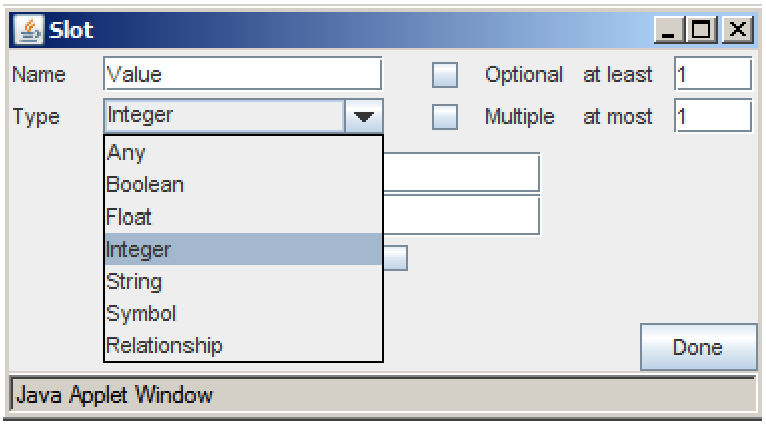


Fig. 5. Adding a new property

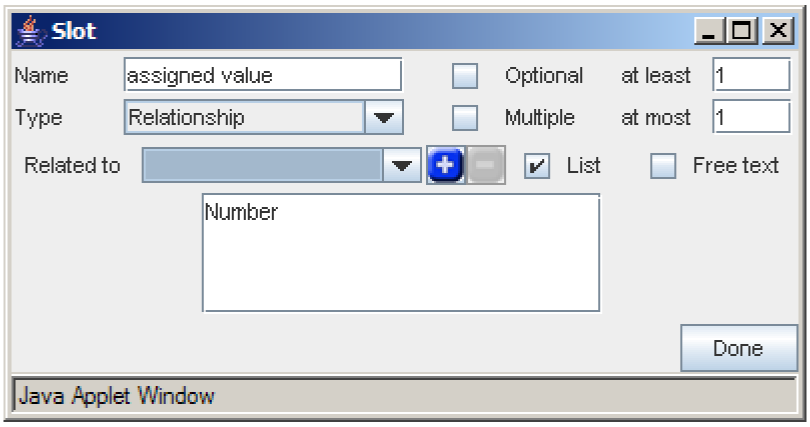


Fig. 6. Adding a relationship

We decided to design and implement an ontology editor specifically for ASPIRE after evaluating a variety of commercial and research ontology development tools. Although tools such as Protégé (2006), OilEd (Bechhofer, Horrocks, Goble, & Stevens, 2001) and Semantic Works (Altova, 2005) are sophisticated and possess the ability to represent complicated syntactical domain restrictions in the form of axioms, they are not intuitive to use. They were designed for knowledge engineering experts, and 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 ITSs for their courses. To achieve this goal, the system should be easy and intuitive to use. We therefore decided to build an ontology editor specifically for this project to reduce the required training for users. The resulting ontology editor was designed to compose ontologies in a manner analogous to using a drawing program. Ease-of-use was achieved by restricting its expressive power: in contrast to Protégé where axioms can be specified as logical expressions, the ASPIRE ontology workspace only requires a set of syntactic restrictions to be specified through its interface. We believe this is sufficient for the purpose of generating syntax constraints for most instructional domains.

The ontology workspace does not offer a way of specifying restrictions on different properties attached to a given concept, such as that the number of years of work experience should be less than the person's age. It also does not contain functionality to specify restrictions on properties from different concepts, such as that the salary of the manager has to be higher than the salaries of employees for whom they are responsible. Such arbitrary restrictions can be specified in Protégé using the Protégé Axiom Language (PAL). As the constraints in ASPIRE analyse the ontology as well as sample problems and their solutions, a complete ontology that includes all the restrictions is not required to generate a complete constraint base.

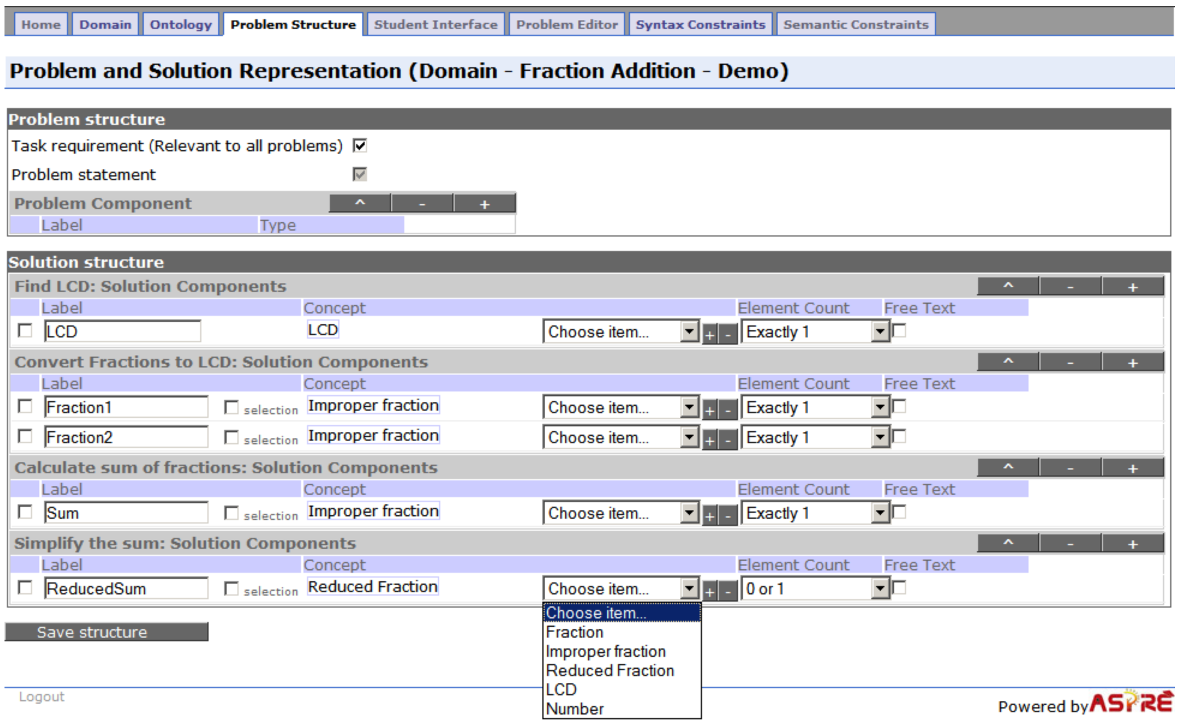
Phase 3: Modelling the Problem/Solution Structures

In phase 3 the author specifies the problem and solution structure for each problem set, using the *Problem Structure* tab illustrated in Figure 7. ASPIRE assumes that each problem contains a problem statement. In addition to the problem statement, the author can specify the task requirement, that is the description of the task and additional instructions that will be given to students for each problem. For example, in the fraction addition domain the task requirement may be "Add the following two fractions," while the problem statement will specify the two fractions to be added (e.g., 1/5 + 2/3). As another example, let us take a look at a language tutor, which contains a set of problems dealing with turning verbs into nouns. All problems of this type would have the same task requirement entered just once by the author: "Turn the following verb into a noun." Each verb would then be entered separately as the problem statement.

A problem may also contain a collection of sub-components that add more information to the problem statement. Problem components are described by their label and type. The label is displayed in the student interface next to the problem component. Each component can be either textual or graphical. The components are problem-specific, and are therefore specified in the Problem Editor.

The solution structure for a non-procedural task consists of a list of solution components. A solution component is described in terms of the ontology concept(s) it represents. The task of modelling the solution structure therefore involves decomposing a solution into components and identifying the type of elements (in terms of ontology concepts) each component may hold. The author specifies the label for the solution element, selects one or more concepts from the ontology (using the drop-down lists labelled *Choose item*), and the number of elements it may hold (*Element Count)*. Additionally, there is a *Free text* box for each component, which needs to be ticked if the student can freely type the content of the component. The free text components will be displayed in the student problem-solving interface as text boxes.

For procedural tasks, each problem-solving step might require several parts, and therefore the author needs to specify the solution components for each step. Consequently, the solution structure for procedural domains consists of a collection of solution component lists, one for each problem-solving step. For example, in the first step of adding fractions, the student needs to specify the lowest common denominator, which is a single number. To add this component, the author would specify the label the student will see (*LCD* in Figure 7), select the LCD concept from the options listed, and finally specify that there is only one number to be added.


Home | Domain | Ontology | Problem Structure | Student Interface | Problem Editor | Syntax Constraints | Semantic Constraints

Problem and Solution Representation (Domain - Fraction Addition - Demo)

Problem structure

Task requirement (Relevant to all problems) ☑

Problem statement ☑

Problem Component

Label	Type

Solution structure

Find LCD: Solution Components

Label	Concept		Element Count	Free Text
☐ LCD	LCD	Choose item...	Exactly 1	☐

Convert Fractions to LCD: Solution Components

Label		Concept		Element Count	Free Text
☐ Fraction1	☐ selection	Improper fraction	Choose item...	Exactly 1	☐
☐ Fraction2	☐ selection	Improper fraction	Choose item...	Exactly 1	☐

Calculate sum of fractions: Solution Components

Label		Concept		Element Count	Free Text
☐ Sum	☐ selection	Improper fraction	Choose item...	Exactly 1	☐

Simplify the sum: Solution Components

Label		Concept		Element Count	Free Text
☐ ReducedSum	☐ selection	Reduced Fraction	Choose item...	0 or 1	☐

Choose item...
Fraction
Improper fraction
Reduced Fraction
LCD
Number

Save structure

Logout

Powered by ASPIRE

Fig. 7. Problem and solution structures for the Fraction Addition domain

Phase 4: Specifying the Student Interface

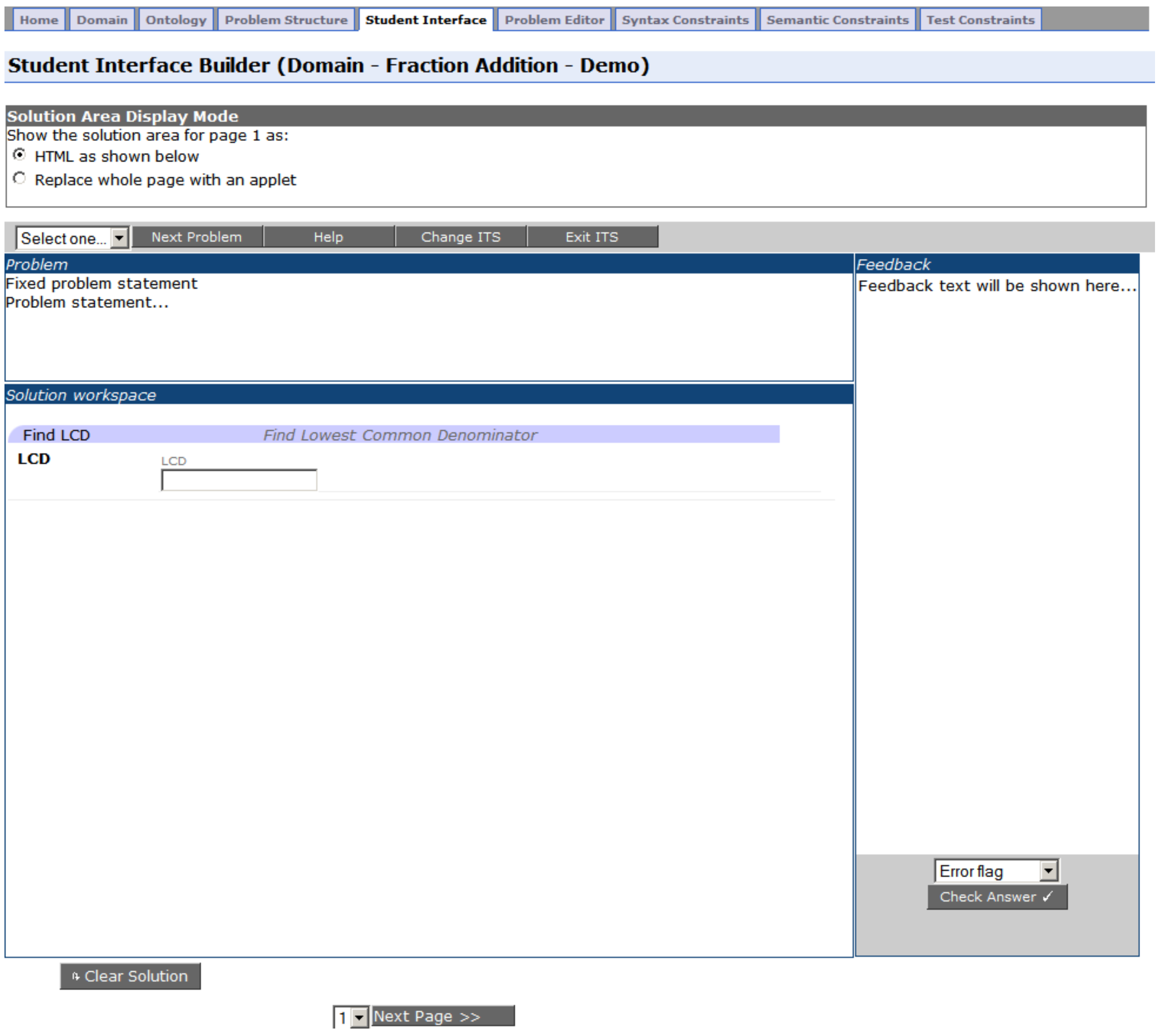
After specifying the problem/solution structures, ASPIRE will show the student interface tab (Figure 8). At the top of the page the author is asked to supply information about the display mode. The default option is the HTML interface as shown in this figure. If the author accepts this default option, the student will be given an interface automatically generated by ASPIRE consisting of an input area for each component defined in the solution structure.

The default HTML interface expects the student to type in the components of the solution. However, in some domains this is not a realistic expectation. For example, in a Mechanics tutor the student may draw a force diagram, so textual input is not appropriate. In such cases, the author may provide a domain-specific applet to support the student in performing one or more steps. Please note that we do not expect the author to develop the applet himself/herself, as such development would require programming expertise. The applet would need to be developed by software professionals. We discuss this issue in more depth later in the paper. If there is an applet to be added, the author needs to specify that the student interface will contain such an applet. If the author selects the second option, the applet will replace the HTML interface, and will include all solution components for that page. The author then needs to upload the applet to ASPIRE.

At the top of the student interface, there is a set of standard buttons that students may use to select problems, get help on how to use the system, change the system or log out. Under the buttons, the interface displays the problem area, including the general instruction and the problem statement.

Phase 5: Adding Problems and Solutions

In the fifth phase, the author adds problems and their solutions. The interface for this is similar to the default student interface (i.e., HTML interface generated by ASPIRE from the domain definition). To generate this interface, ASPIRE-Author uses the previously specified problem/solution structures. Therefore, when the author starts adding the first problem for the domain, the Problem Editor provides the author with the necessary interface widgets based on the problem structure, and expects the author to populate them.



Student Interface Builder (Domain - Fraction Addition - Demo)

Solution Area Display Mode

Show the solution area for page 1 as:

[✓ HTML as shown below]

Replace whole page with an applet

Problem

Fixed problem statement

Problem statement...

Feedback

Feedback text will be shown here...

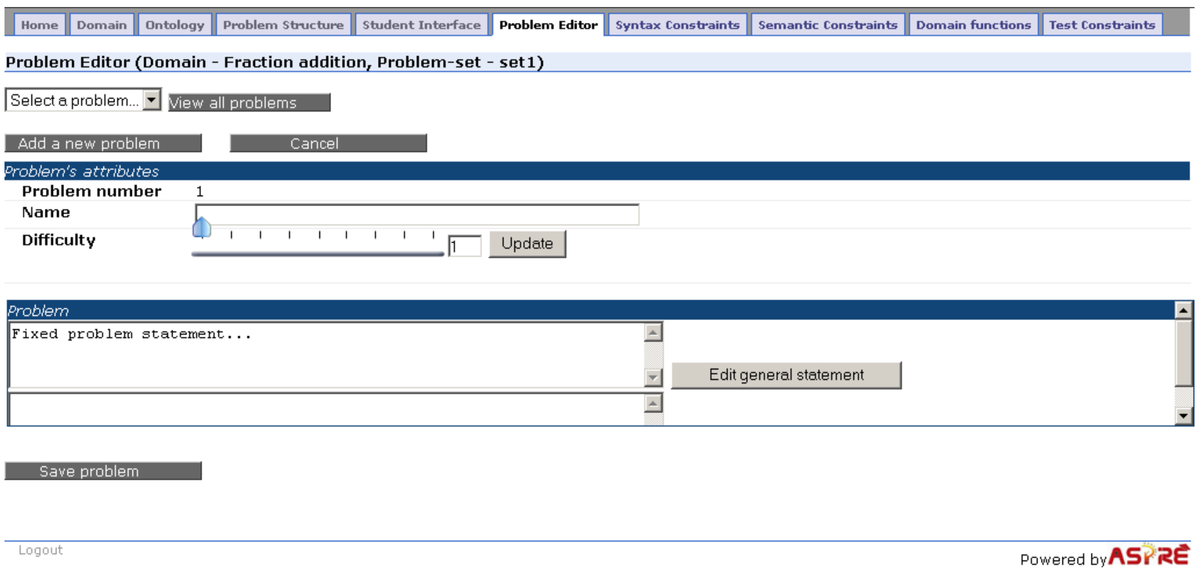
Solution workspace

Find LCD — Find Lowest Common Denominator

LCD — LCD

Fig. 8. The default student interface for the first step of the fraction addition task

There are several general problem features to specify, shown in the *Problem's attributes* area (Figure 9). The unique problem number is generated automatically by the system (1 in this case, as the author is adding the first problem for the chosen problem set). The author may then specify an optional name for the problem. If the problem name is specified, it will be shown to students together with the problem number; otherwise, students will only see the problem number. The author must specify the problem difficulty (ranging from 1 for the simplest problems to 9 for the most complex problems), the problem statement and (optionally) problem components.



Problem Editor (Domain - Fraction addition, Problem-set - set1)

Problem's attributes

Problem number	1
Name	
Difficulty	1

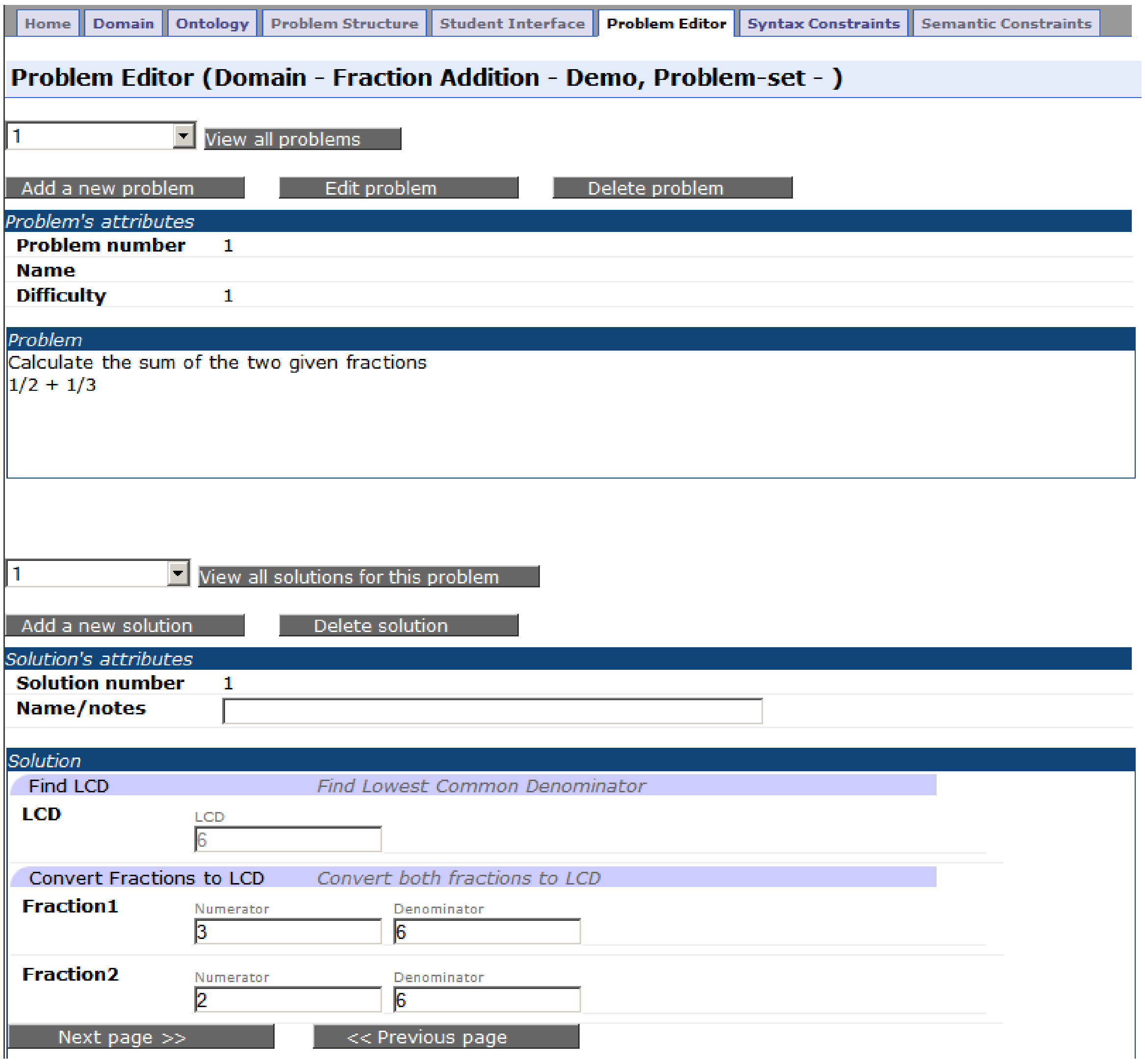
Fixed problem statement...

Fig. 9. Adding a problem

After saving the problem, the author can add one or more solutions to it, as illustrated in Figure 10. For procedural tasks, if there are multiple steps for solving a problem, the solution workspace allows the author to enter all the steps simultaneously rather than navigating through them one at a time as the students would. This eliminates the navigation effort needed between steps, making it quicker and easier for the author to add and inspect the full solution for a problem. Each step is displayed along with its name and the description that the students would see, and is separated by borders to make a clear distinction between steps. The author needs to specify the solution components for each problem-solving step. In domains where there are multiple solutions per problem, the author should enter all practicable alternative solutions. The solution editor reduces the amount of effort required to do this by allowing the author to transform a copy of the first solution into the desired alternative. This feature significantly reduces the author's workload because alternative solutions often have a high degree of similarity.

Phases 6 and 7: Constraint Generation

Syntax constraints are generated on author's request, from the domain ontology. The ontology contains a lot of information about the syntax of the domain. The syntax constraint generation algorithm extracts all useful syntactic information from the ontology and translates it into constraints. Syntax constraints are generated by analysing relationships between concepts and concept properties specified in the ontology (Suraweera, Mitrovic & Martin, 2004b). For example, any restrictions specified on relationships, such as minimum and maximum cardinalities, will be translated into constraints automatically. The same happens with data types and value ranges specified for properties. An additional set of constraints is also generated for procedural tasks, which ensures that the student performs the problem-solving steps in the correct order (also called *path constraints*). For example, in the domain of fraction addition a constraint is generated that verifies that the lowest common denominator (LCD) part of the student's solution is not empty, which becomes relevant when the student is working on this step and prevents them from moving on to the following step before satisfying it.



Problem Editor (Domain - Fraction Addition - Demo, Problem-set -)

Problem: 1

Problem's attributes

Problem number	1
Name	
Difficulty	1

Problem

Calculate the sum of the two given fractions
1/2 + 1/3

Solution: 1

Solution's attributes

Solution number	1
Name/notes	

Solution

Find LCD — *Find Lowest Common Denominator*

LCD: LCD 6

Convert Fractions to LCD — *Convert both fractions to LCD*

Fraction1: Numerator 3, Denominator 6

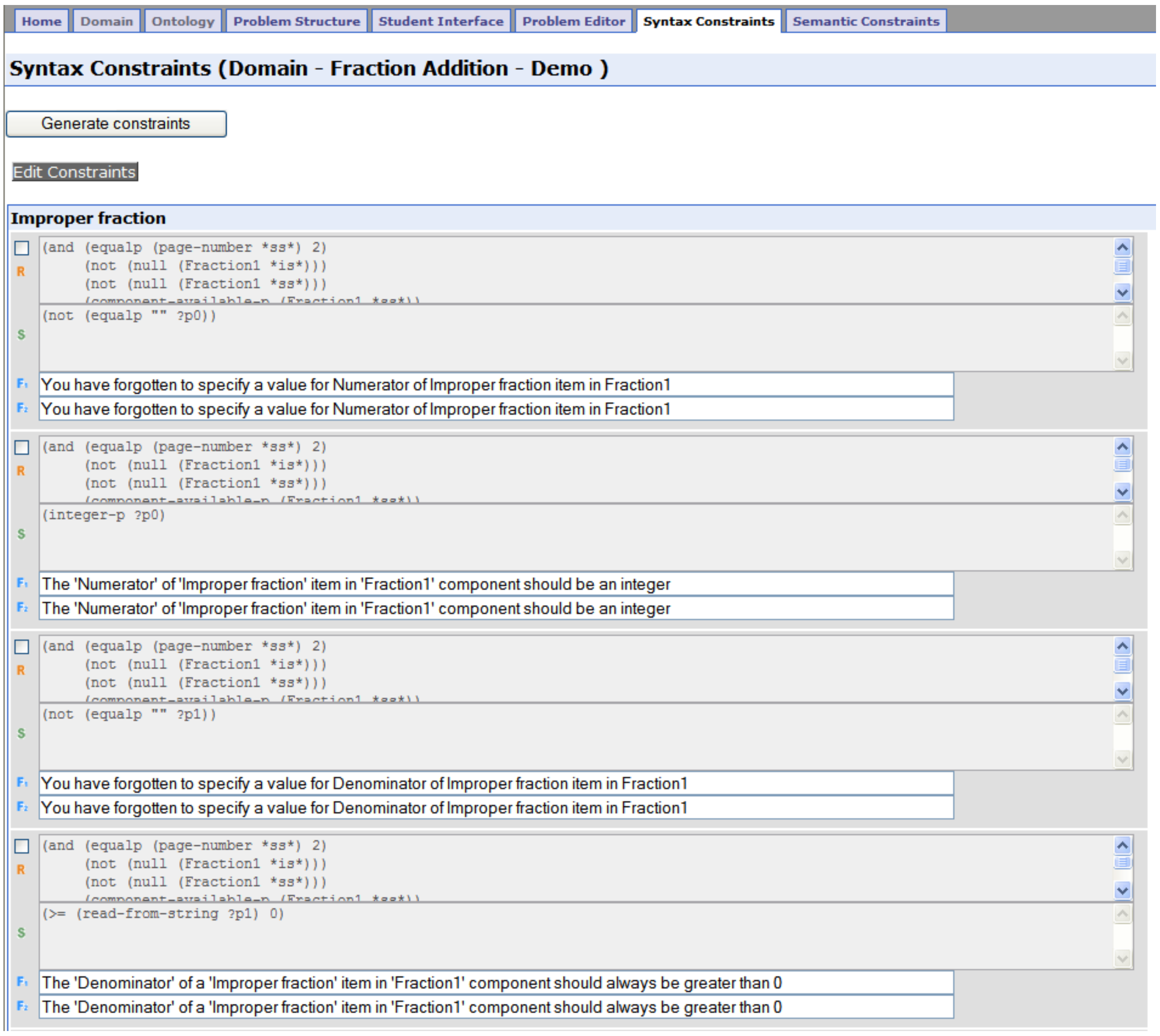
Fraction2: Numerator 2, Denominator 6

Fig. 10. Adding a solution

Figure 11 illustrates some syntactic constraints generated for the fraction addition domain. The constraints are arranged in groups corresponding to domain concepts they have been generated from. The figure shows four constraints generated for the *Improper fraction* concept. Each constraint consists of two conditions (relevance and satisfaction condition) followed by two feedback messages. The two feedback messages will be shown to the student one at a time, first when the constraint is violated, and then when the student asks for more feedback. ASPIRE generates these messages automatically. However, the author can modify them to make them more useful for the student.

The tick-box at the beginning of each constraint allows the constraint to be selected for further action. For example, to delete one or more constraints the author first selects them, and then clicks the *Delete* button at the bottom of the page.

As discussed previously, we do not assume authors will be able to understand the constraint language. However, ASPIRE supports other types of users, such as developers, who are familiar with the constraint language. ASPIRE allows developers to modify constraints, or even add constraints manually; authors are not permitted to do this.



Syntax Constraints (Domain - Fraction Addition - Demo)

Improper fraction

```
R (and (equalp (page-number *ss*) 2)
       (not (null (Fraction1 *is*)))
       (not (null (Fraction1 *ss*)))
       (component-available-p (Fraction1 *ss*))
S (not (equalp "" ?p0))
```
F1 You have forgotten to specify a value for Numerator of Improper fraction item in Fraction1
F2 You have forgotten to specify a value for Numerator of Improper fraction item in Fraction1

```
R (and (equalp (page-number *ss*) 2)
       (not (null (Fraction1 *is*)))
       (not (null (Fraction1 *ss*)))
       (component-available-p (Fraction1 *ss*))
S (integer-p ?p0)
```
F1 The 'Numerator' of 'Improper fraction' item in 'Fraction1' component should be an integer
F2 The 'Numerator' of 'Improper fraction' item in 'Fraction1' component should be an integer

```
R (and (equalp (page-number *ss*) 2)
       (not (null (Fraction1 *is*)))
       (not (null (Fraction1 *ss*)))
       (component-available-p (Fraction1 *ss*))
S (not (equalp "" ?p1))
```
F1 You have forgotten to specify a value for Denominator of Improper fraction item in Fraction1
F2 You have forgotten to specify a value for Denominator of Improper fraction item in Fraction1

```
R (and (equalp (page-number *ss*) 2)
       (not (null (Fraction1 *is*)))
       (not (null (Fraction1 *ss*)))
       (component-available-p (Fraction1 *ss*))
S (>= (read-from-string ?p1) 0)
```
F1 The 'Denominator' of a 'Improper fraction' item in 'Fraction1' component should always be greater than 0
F2 The 'Denominator' of a 'Improper fraction' item in 'Fraction1' component should always be greater than 0

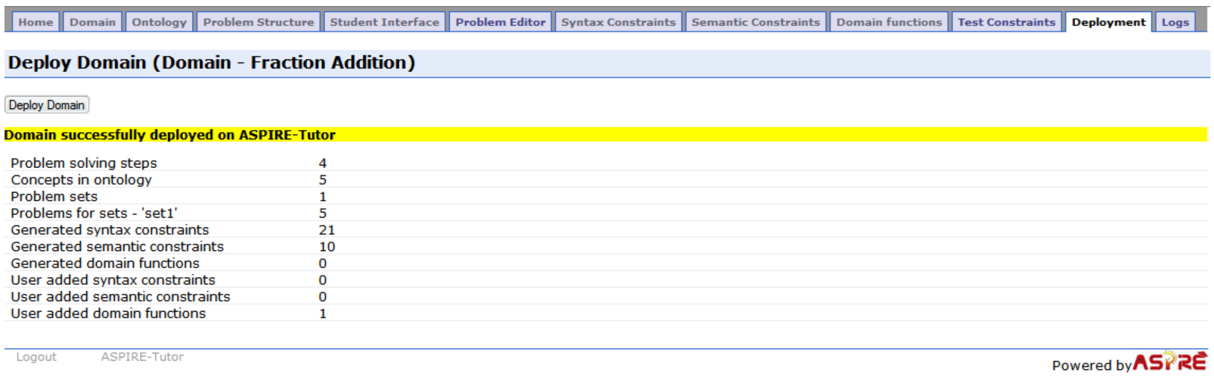
Fig. 11. Generated syntactic constraints

In the next phase, the author requests the semantic constraints to be generated. Those constraints check that the student's solution has the desired meaning (i.e., it answers the question). Constraint-based tutors determine semantic correctness by comparing the student solution to a single correct solution to the problem; however, they are still capable of identifying alternative correct solutions because constraints are encoded to check for equivalent ways of representing the same semantics

(Mitrovic & Ohlsson, 1999; Ohlsson & Mitrovic, 2007; Suraweera & Mitrovic, 2004). Such constraints are generated by ASPIRE from analysing alternative correct solutions for the same problem supplied by the author. ASPIRE analyses the similarities and differences between two solutions to the same problem. The multiple alternative solutions specified by the author enable the system to generate constraints that will accept alternative solutions by comparing the student's solution to the stored (ideal) solution in a manner that is not sensitive to the particular approach the student used. A detailed discussion of the constraint-generation algorithms is beyond the scope of this paper; the interested reader is referred to (Suraweera et al., 2005).

Phase 8: Deploying the Domain

Once the author has completed all the authoring steps, he/she may wish to see the tutoring system running. This allows the author to interact with the final tutoring system, solving problems and receiving feedback in a manner similar to students. The task of starting a tutoring system (to run on ASPIRE-TUTOR) is called *deployment*. On the author's request, ASPIRE-Author will perform a number of checks on the domain to test that the information supplied by the author is consistent and the domain model is complete. Figure 12 shows a screenshot of the deployment page of a domain where ASPIRE has not found any inconsistencies. In such cases the author can simply click on the Deploy Domain button. The author can then try the tutoring system on ASPIRE-Tutor.



Deploy Domain (Domain - Fraction Addition)

Domain successfully deployed on ASPIRE-Tutor

Problem solving steps	4
Concepts in ontology	5
Problem sets	1
Problems for sets - 'set1'	5
Generated syntax constraints	21
Generated semantic constraints	10
Generated domain functions	0
User added syntax constraints	0
User added semantic constraints	0
User added domain functions	1

Fig. 12. Deploying a domain

ASPIRE-TUTOR

ASPIRE-Tutor (Fig. ~~Figure ~~13) is also designed as a collection of modules based on the typical ITS architecture. It is capable of serving a collection of tutoring systems in parallel. The student accesses an intelligent tutoring system in ASPIRE through a Web browser. The interface module is responsible for producing an interface for each tutoring system deployed on the server. 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 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. When evaluating a student's submission and providing feedback, the Pedagogical Module delegates the task of evaluating the solution to the diagnostic module and decides on the appropriate feedback by consulting the student model. The student modeller maintains a long term model of the student's knowledge.

Requests to other modules result in status and optional data being returned to the Pedagogical Module. In addition, the functional modules may access and/or update data objects (e.g., student model, domain model, logs) which are stored in the Allegro Cache database. The Pedagogical Module returns the final status and data to the Session Manager. The Session Manager returns the result to the interface by packaging up a response and/or indicating which interface object should be presented next.

The Diagnostic Module analyses students' solutions, and identifies any mistakes students made. In order to be able to perform this task, the Diagnostic Module needs the services of the Domain Manager, the component that is in charge of all knowledge developed for various intelligent tutoring systems. On the basis of the diagnosis performed by the Diagnostic Module, the Student Modeller updates the student model (i.e., the system's view of the student's knowledge). The student model is used to adapt instructional actions to meet the needs and abilities of each individual student (for example, by selecting problems at the appropriate level of complexity for the student).

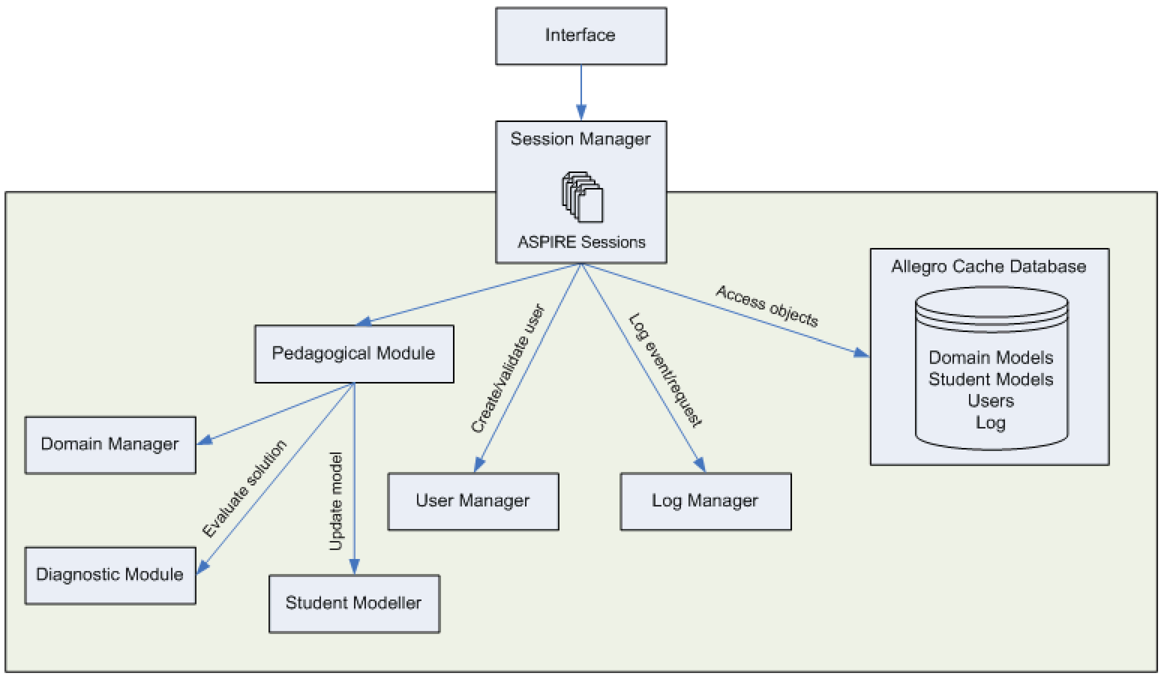


Fig. 13. The architecture of ASPIRE-Tutor

All actions students perform in ASPIRE are logged by the Log Manager. Finally, the User Manager maintains user information and ensures that only authorized users can access ASPIRE and the various intelligent tutoring systems defined within it. There are five types of users in ASPIRE:

students, teachers, administrators, developers and authors. Each user group has specific privileges and rights and can access different parts of the system. The User Manager makes sure that users can access the part of ASPIRE they need.

For example, the teacher can specify the progression of feedback levels. By default, ASPIRE offers the following feedback levels: *Quick Check* (specifying whether the answer is correct or not), *Error Flag* (identifying only the part of the solution that is erroneous), *Hint* (identifying the first error and providing information about the domain principle that is violated by the student's solution), *Detailed Hint* (a more detailed version of the hint), *All Errors* (hints about all errors) and *Show Solution.* By default, ASPIRE starts with *Quick Check* and progresses with each consecutive submission of the same problem to *Detailed Hint*, unless the student asks for a specific type of feedback. Information about all errors and the solution are only available at request. However, the teacher can override this default behaviour by limiting the types of feedback, prohibiting the full solution from being shown, specifying the minimal number of attempts before the full solution can be seen, or by specifying the maximal level of feedback to be provided automatically. The teacher can also specify the problem selection mechanisms available to students.

As discussed earlier, both authors and developers can develop new ITSs in ASPIRE-Author. The difference between those two groups of ASPIRE users is that developers can add, delete and modify constraints generated by ASPIRE, while authors can only modify feedback messages.

EVALUATION

In previous work, we evaluated the constraint generation algorithms employed in ASPIRE (Suraweera et al., 2005, 2007). The effectiveness of constraint generation was evaluated by producing constraint sets for three domains: conceptual database design using Entity-Relationship (ER) modelling, Fraction addition and Data normalisation. Database design and Data normalisation were specifically chosen because we had previously developed two successful constraint-based tutors for the two domains: KERMIT (Suraweera & Mitrovic, 2002, 2004) and NORMIT (Mitrovic, 2003, 2005), respectively. The constraint bases of these tutors were therefore used as benchmarks to evaluate the correctness and completeness of the constraint bases generated by those algorithms. The choices of domains were also influenced by the desire to evaluate the algorithm on both procedural and non-procedural tasks. Data normalisation and Fraction addition can be categorised as procedural domains, where a strict set of steps must be followed to arrive at the solution. On the other hand, database design is an ill-defined task; there is no strict procedure to be followed to produce an ER model. The results of the evaluation showed that the generated constraints accounted for over 90% of the constraints required for the domain; the missing constraints needed to be added manually. Constraints that were not generated required additional problem-solving or algebraic functionality that is not currently supported by the constraint-generation algorithms. However, since constraints operate independently, the generated constraints are sufficient for an early version of a tutoring system.

We also used ASPIRE in COSC420, a graduate ITS course at the University of Canterbury taught by the first author of this paper. The students enrolled in this course had previously learnt about ITSs and constraint-based tutors in 12 lectures, and had some practical experience of using various ITSs. Therefore, they had more ITS experience and less domain expertise than what we expect from a typical ASPIRE author. They were assigned the task of producing a complete ITS in ASPIRE for the domain of balancing chemical equations. We provided a high-level description of the system's behaviour: each problem should specify the chemical compounds on the left- and right-hand side of the equation (reactants and products, respectively), and the student should determine the coefficients required (i.e., to specify the number of molecules of each reactant/product in the equation). We introduced ASPIRE, and the suggested authoring process, and provided a list of unbalanced chemical equations. The participants were free to design domain ontologies and problem/solution structures as they wished. They all had developer accounts for ASPIRE, which allowed them to modify the constraint sets produced by ASPIRE. The time period for completing the assignment was three weeks. Out of nine students enrolled in the class, only one did not complete the assignment, due to the high load in other courses. All other students produced running tutors. At the time when the course was offered, ASPIRE was still being developed, and therefore the students experienced some system crashes and identified some bugs in the system. We therefore do not treat this study as an evaluation study, but it did provide us with valuable experiences and suggestions for improving the system. The students agreed that ASPIRE eased the development of ITSs in comparison to manual development. They liked the ontology workspace, but pointed out that ASPIRE was a complex system, and that it did require a lot of training to be used properly.

Additionally, we held a full-day tutorial at the AIED 2007 conference, in which we introduced ASPIRE to participants and engaged them with small hands-on activities in ASPIRE. Based on the experiences gained at the tutorial, one of the participants, Prof. Glenn Blank, decided to use ASPIRE in his graduate class on ITSs (CSE497) at Lehigh University in 2007. There were nine students enrolled in the class, and they were assigned the task of building a tutor that taught fraction addition in both ASPIRE and CTAT. The students learnt about constraint-based and model-tracing tutors in lectures, and had access to both authoring environments and accompanying manuals. Some students had difficulties with the authoring procedure. For example, ASPIRE needs multiple problems and solutions in order to generate general constraints. If the author only provides a single problem/solution, the constraint generated will be overly specific (i.e., they will contain all the specifics of the single solution provided). In order for constraints to be generalized, ASPIRE needs several problems. One student in this class had a problem understanding this requirement, but after a clarification was able to complete the ITS.

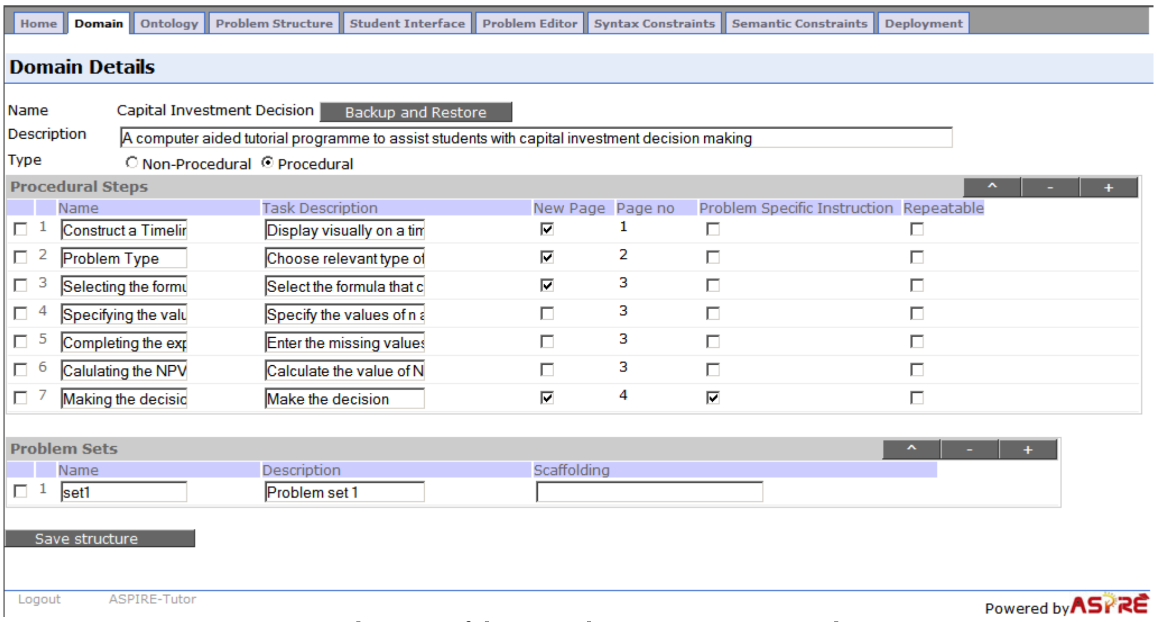
It should be pointed out that both ASPIRE and CTAT were still under development at the time, and therefore some advanced features were missing from both environments. Furthermore, the students were not given the full training that the developers of ASPIRE and CTAT envision for proper use. The following findings therefore need to be taken as preliminary only. The comments we received from this group of students pointed out that more initial learning was needed to use ASPIRE effectively in comparison to CTAT, which is not surprising as ASPIRE is a general authoring system. The students pointed out that CTAT was easier to use initially, but was much more restrictive. They stated that in CTAT it was easy to create a problem, but to achieve good coverage of the domain, lots of problems needed to be created manually[2]. The other shortcoming of CTAT was that the resulting tutors were inflexible, requiring the student to follow the pre-specified paths. On the other hand, the students praised the generality of ASPIRE and its applicability to a variety of instructional domains in contrast to CTAT, which is aimed at procedural tasks. The other observation was that the resulting constraint-based tutors were more flexible than tutors developed in CTAT.

[2] "The current version of CTAT is capable of supporting fully-flexible fraction addition tutors, and makes it easy to "mass produce" tutors for problems with isomorphic solution spaces." (*V. Aleven, personal communication, 28 December 2008*)

We at ICTG have used ASPIRE to develop a number of ITSs. However, to test whether ASPIRE has reached its goal, we also asked people outside of ICTG to use ASPIRE. Two university instructors without ITS expertise used ASPIRE to develop tutors for the courses they teach at two different universities in New Zealand. We describe one of those systems, the Capital Investment Tutor (CIT), which was developed by the last author of this paper. We present the process of developing CIT and the evaluation study performed with it. The other ITS (in the area of thermodynamics) is still under development.

CIT: Teaching Capital Investment Decision Making

Capital investment decision-making plays a crucial role in the financial evaluation of non-current assets within contemporary organisational practice. Teaching experience shows that capital investment evaluation techniques, namely, the accounting rate of return, net present value and the internal rate of return, are problematic for students to master. Students find the principles of capital investment decision-making difficult to comprehend and lack the ability to translate from theory to practice. It was envisaged that CIT would enable students to apply theoretical financial decision-making to real-life simulated business environments. It was with this in mind that CIT was developed by the last author of this paper (who had no ITS/programming experience) as one of the crucial evaluation stages of the ASPIRE project. The author developed CIT in consultation with the ASPIRE team, as discussed later in this section.



Domain Details

Name: Capital Investment Decision

Description: A computer aided tutorial programme to assist students with capital investment decision making

Type: Non-Procedural [✓ Procedural]

Procedural Steps

		Name	Task Description	New Page	Page no	Problem Specific Instruction	Repeatable
☐	1	Construct a Timelir	Display visually on a tin	☑	1	☐	☐
☐	2	Problem Type	Choose relevant type of	☑	2	☐	☐
☐	3	Selecting the formu	Select the formula that c	☑	3	☐	☐
☐	4	Specifying the valu	Specify the values of n a	☐	3	☐	☐
☐	5	Completing the exp	Enter the missing values	☐	3	☐	☐
☐	6	Calulating the NPV	Calculate the value of N	☐	3	☐	☐
☐	7	Making the decisic	Make the decision	☑	4	☑	☐

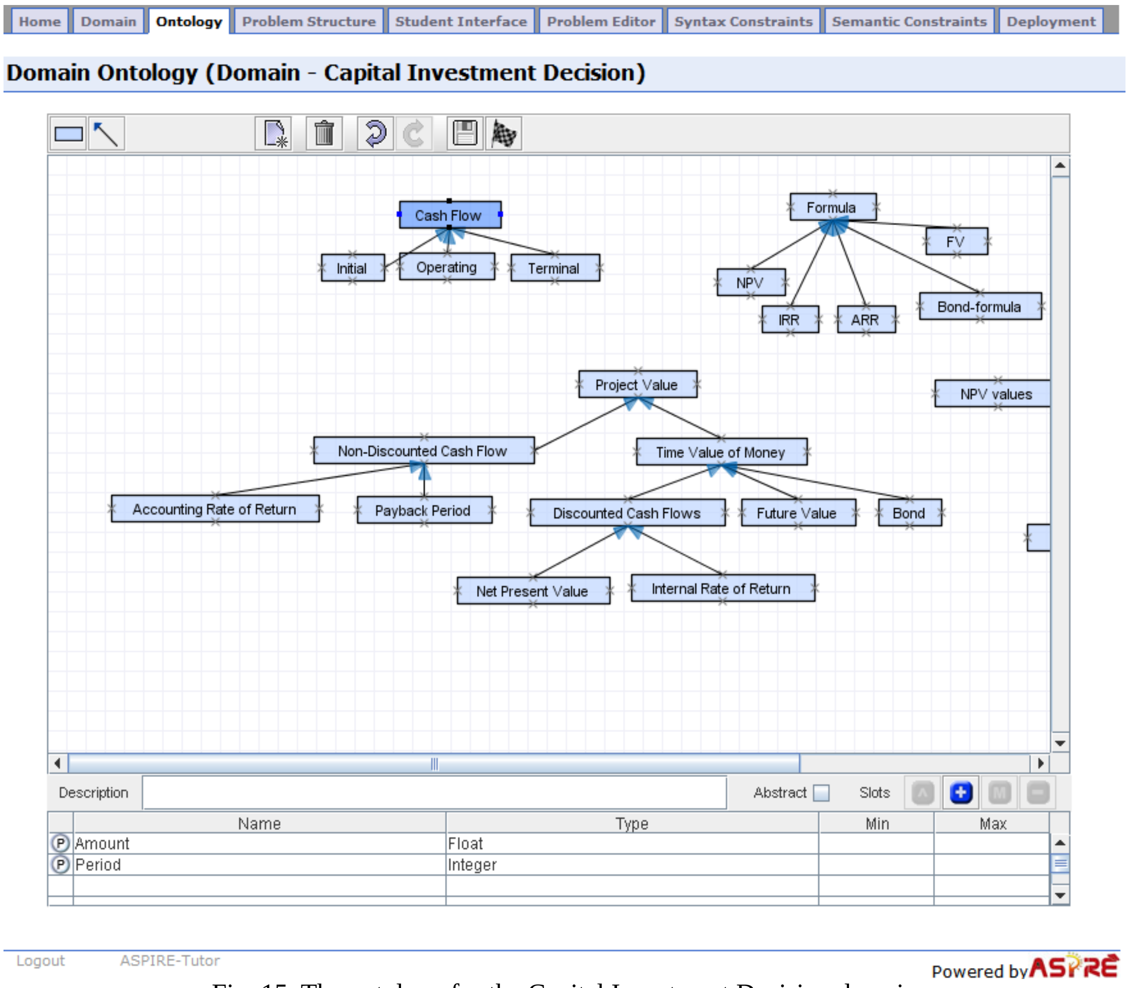
Problem Sets

		Name	Description	Scaffolding
☐	1	set1	Problem set 1	

Fig. 14. The steps of the Capital Investment Decision domain

Figure 14 shows a screenshot of the Domain tab of ASPIRE-Author for CIT (which corresponds to the first phase of the authoring procedure in which the author describes the domain and specifies problem-solving steps). The task the student needs to perform is a procedural one, consisting of seven steps. In the first step, the student constructs a timeline of project costs from the information given in the problem statement. This step is shown to the student on its own on the first page. In step two (shown on its own on a new page), the student identifies the relevant problem type in terms of which variable needs to be calculated. Step 3 requires the student to select the formula corresponding to the chosen variable. They then enter the parameters for the formula in step 4. In steps 5 and 6, the student enters the known values into the selected formula and then specifies the computed value. Based on this computed value, the student then makes the final decision regarding capital investment in step 7. In CIT there is only a single problem set, although ASPIRE allows for multiple problem sets to be defined for a domain.

The domain ontology for CIS is illustrated in Figure 15, showing the important domain concepts and their relationships. There are 30 concepts in the ontology, each containing one or two properties. For example, the *Cash Flow* concept is specialised into *initial*, *operating* and *terminal* cash flows. The author then specified the problem structure; in CIT, each problem has a problem statement and an attached image. The solution structure is given in Table 2.



Name	Type	Min	Max
Amount	Float		
Period	Integer		

Fig. 15. The ontology for the Capital Investment Decision domain

From this information, ASPIRE was able to generate the default HTML student interface. The author wanted the final student interface to use applets, rather than text boxes supplied by the default student interface, to make CIT more visually attractive. The only action the author performed in phase 4 was to indicate that applets would replace the default student interface. Please note that the applets themselves were not developed by the author, as the expertise required for developing applets is far above the normal expectations of ASPIRE authors.

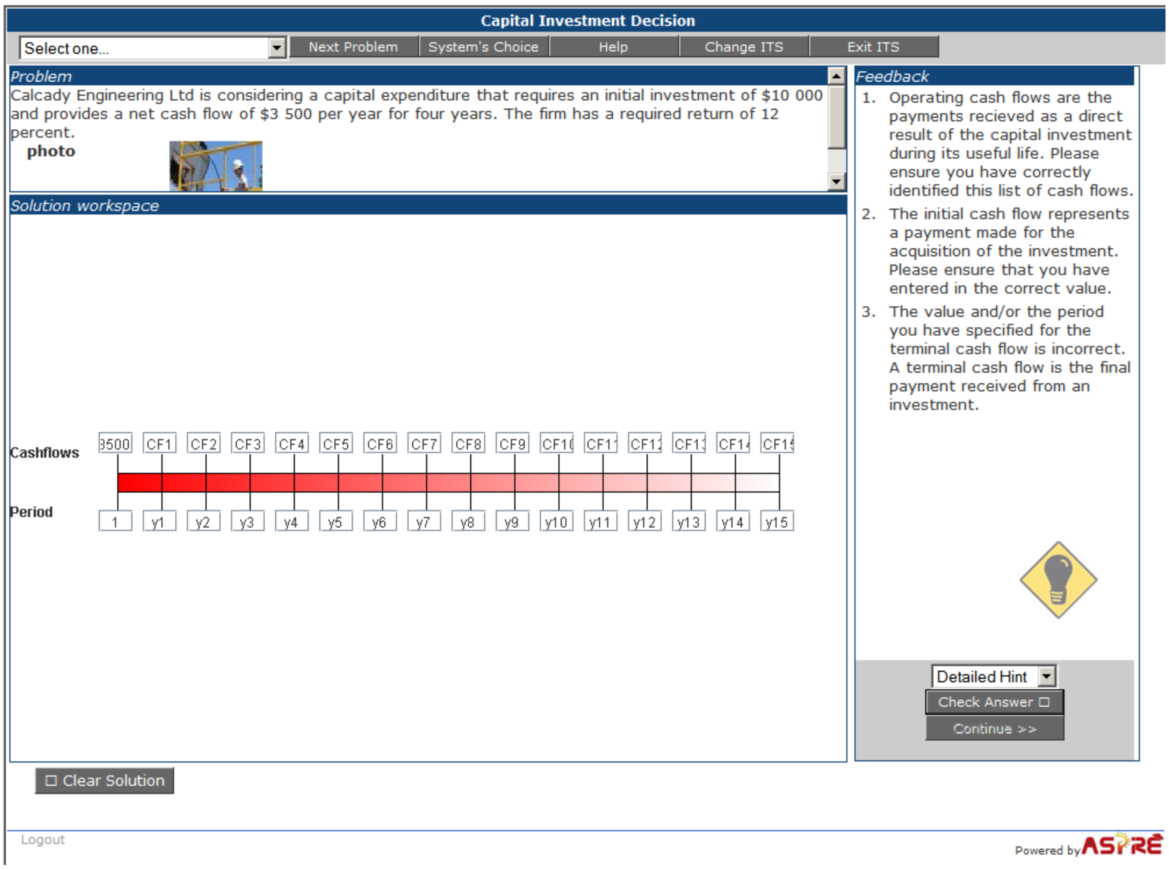
Table 2. Solution structure for CIT

Problem-solving step	**Solution components**
1. Construct a timeline	Cash flows (initial, operating and terminal)
2. Identify problem type	One of *Accounting Rate of Return, Bond, Future Value, Internal Rate of Return, Net Present Value (NPV), Payback Period*
3. Select the formula	One of the pre-specified set of formulae
4. Specify the parameters for the formula	n, k
5. Complete the formula	All components of the NPV formula
6. Enter the NPV value	NPV value
7. Make the final decision	Decision

In phase 5 the author entered twelve problems and their solutions. In this domain, there is only one correct solution per problem. Syntax and semantic constraints were generated, and the author modified the automatically generated feedback messages to be more helpful to students. Finally the author deployed the domain, which resulted in the domain information being transferred to ASPIRE-Tutor. The author/teacher also defined a group, consisting of students who would have access to the system, and tailored the behaviour of the system by specifying the feedback and problem selection options available to the students.

Figure 16 shows the student interface with the applet for the first step in CIT. The top area of the page provides controls for selecting problems, obtaining help, and changing/leaving the ITS. The problem statement is shown together with the photo describing the situation. The problem-solving area for this step consists of an applet that visualizes the timeline. The student needs to label the period on this timeline and enter the amounts corresponding to the various types of cash flows. In the situation illustrated in Figure 16, the student has incorrectly labelled the initial time as period 1 on the timeline, entered the incorrect value for the initial cash flow, and failed to specify the rest of the timeline. The feedback shown in the right-hand panel corresponds to the *All Errors* level: the first hint discusses operating cash flows that are missing, the second discusses the initial cash flow for which the student supplied the wrong value, while the last one discusses the terminal cash flow. The student can change the solution based on the feedback provided, and submit the solution to CIT again.

Since CIT is a procedural tutor, the student needs to complete each step correctly before being allowed to move on to the next one. Figure 17 shows a situation when the student has already completed the first four steps successfully: completed the timeline, selected NPV as the appropriate evaluation mechanism to be used in the problem, picked the correct formula for NPV and specified the correct values for *n* (the number of years) and *k* (the interest rate), which are the parameters used in the formula for NPV. The applet shows the current step, in which the student must fill in the values into the formula for calculating NPV. Please note that the applet shows the expanded version of the summation formula, showing the four terms corresponding to the operating cash flows and the last term (pi) corresponding to the initial cash flow. The goal of this step is to check whether the student can differentiate between the initial cash flow (the term subtracted from the others) and the operating cash flows, and also whether the student understands the various time periods involved and corresponding cash flows. The student has specified the initial cash flow correctly, but has not specified any of the operating cash flows and the corresponding time periods. The feedback shown in Figure 17 is on the *Hint* level, and discusses the operating cash flows, which are missing from the student's solution.



Calcady Engineering Ltd is considering a capital expenditure that requires an initial investment of $10 000 and provides a net cash flow of $3 500 per year for four years. The firm has a required return of 12 percent.

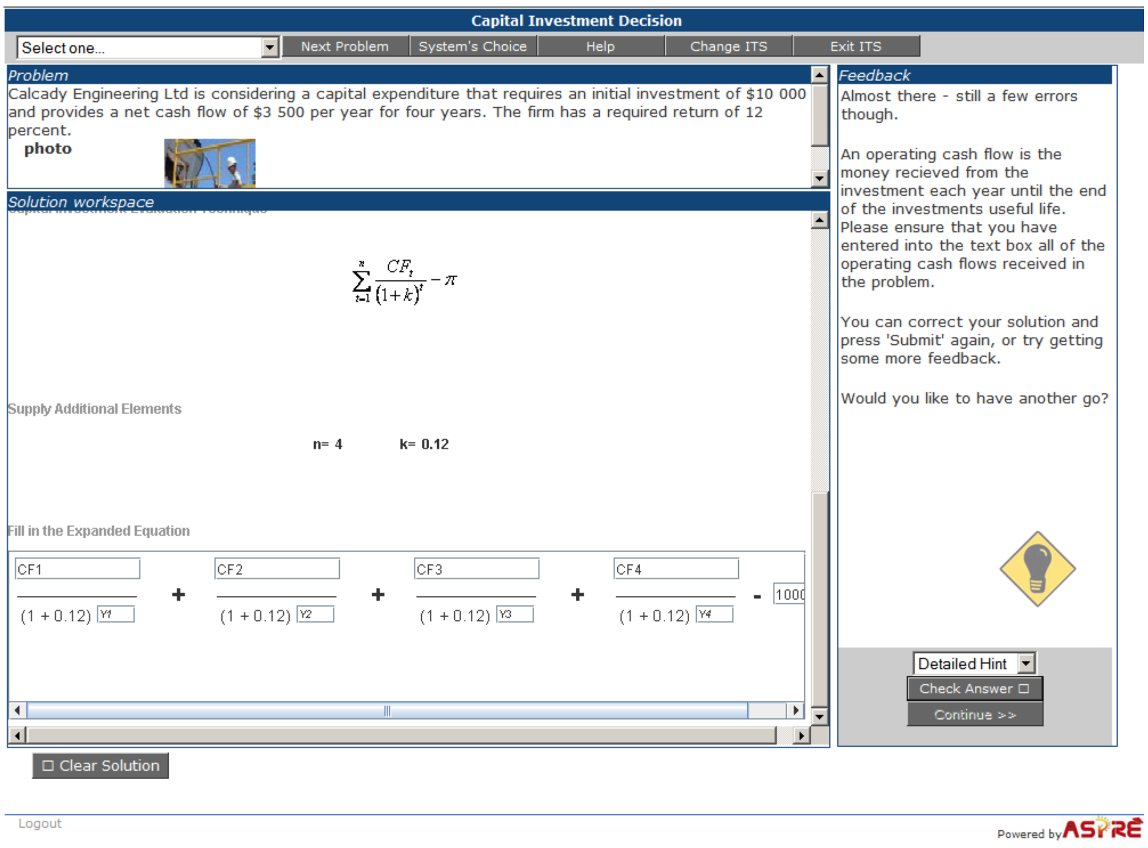
1. Operating cash flows are the payments recieved as a direct result of the capital investment during its useful life. Please ensure you have correctly identified this list of cash flows.
2. The initial cash flow represents a payment made for the acquisition of the investment. Please ensure that you have entered in the correct value.
3. The value and/or the period you have specified for the terminal cash flow is incorrect. A terminal cash flow is the final payment received from an investment.

Fig. 16. Student interface showing the first step of the procedure (the timeline)

CIT uses applets to make interaction more attractive visually, although the task itself can be executed using purely textual input. However, in some domains it might be impossible to use a purely textual interface. For example, if the student needs to develop some kind of diagram, a drawing applet would be necessary. Our experience shows that the time and effort needed to develop applets is higher than the requirements for the other development tasks in ASPIRE. Additionally, applets require software engineering experience, and for that reason we do not expect authors to be able to develop them by themselves.

Experiences in Developing CIT

CIT has been developed over a period of approximately six months from the initial conception through to the finished system. Please note that the development of CIT overlapped with the ASPIRE development; therefore, the ASPIRE documentation (i.e., authoring manual) was not available at the time. This necessitated meetings with the ASPIRE team, in order to familiarize the author with ASPIRE's functionality.


Capital Investment Decision

Problem
Calcady Engineering Ltd is considering a capital expenditure that requires an initial investment of $10 000 and provides a net cash flow of $3 500 per year for four years. The firm has a required return of 12 percent.

Solution workspace

$$\sum_{t=1}^{n} \frac{CF_t}{(1+k)^t} - \pi$$

Supply Additional Elements

n= 4 k= 0.12

Fill in the Expanded Equation

$$\frac{CF1}{(1+0.12)^{Y1}} + \frac{CF2}{(1+0.12)^{Y2}} + \frac{CF3}{(1+0.12)^{Y3}} + \frac{CF4}{(1+0.12)^{Y4}} - 1000$$

Feedback
Almost there - still a few errors though.

An operating cash flow is the money recieved from the investment each year until the end of the investments useful life. Please ensure that you have entered into the text box all of the operating cash flows received in the problem.

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

Would you like to have another go?

Fig. 17. Student interface showing the fifth step of the problem

Contact with the ASPIRE team proved beneficial, providing a way forward when the author was unable to visualise the overall system, causing slow progress and frustration. This was evident during the initial stages, when the author experienced difficulties constructing the ontology for CIT. This was partly due to an inadequate understanding of the ontology, as it can be difficult for content experts to map out what they are teaching in a structured and detailed manner. In the case of CIT, it was difficult for the author to clearly ascertain what was required of him and what purpose the ontology served. In future work, it may prove beneficial to develop a partial ontology, and follow the authoring process through to its entirety, completing the remaining ontology requirements along the way. This would allow the author to obtain a holistic understanding of the process.

The authoring process implemented in ASPIRE is relatively straight forward for an average computer user. The author of CIT had an academic background in business and education and therefore had a limited knowledge of computer applications. A familiarity with computer programming or software applications is not necessary; however, belonging to a younger generation the author has had exposure to computer technology. The author found that, in constructing CIT, the overall ASPIRE system was easy to use with an ability to clearly map the course-learning material to the systems application. ASPIRE was able to meet the requirements needed for the design of the

Capital Investment Decision Making tutor to include a visual image of the case study, clear student interface and templates, and progressive feedback.

The manual uploading of problems and suggested solutions was straightforward and required little development time, proving to be one of the advantages to using the ASPIRE system. The feedback provided to students proved to be one of the most useful elements of CIT as it could be tailored to the student group based on the learning outcomes of the course and the task at hand. The manual modification of generated feedback messages was conducted towards the end of the system's development and was jointly achieved by the ASPIRE team and author. Having not had a computer programming background, the author found this stage relatively difficult, relying on face-to-face assistance to complete the task. An amount of detail could be provided at this stage to assist the author: for example, a text box underneath each constraint, providing an overview of the constraint in English, may prove beneficial.

Evaluation of CIT

We conducted an evaluation study of CIT in a summer school course at Lincoln University in February 2008. The participants were 21 students enrolled in ACCT102 (Accounting and Finance for Business), an introductory business decision-making course. Prior to the study, the students had participated in lectures covering the relevant material. The course had two scheduled tutorial streams, and we randomly selected one of them to serve as the experimental group, while the other served as the control group. The length of tutorials was 90 minutes. During this time, the students took a short pre-test, interacted with the system (experimental) or solved the problems individually and then discussed them with a human tutor (control), and then took a post-test. Both groups spent 45 minutes on problem-solving. The experimental group participants also filled in a user questionnaire, which solicited their impressions of CIT.

Table 3 reports some statistics from the study. There was no significant difference between the pre-test scores, indicating that the prior knowledge of the two groups of participants was comparable. Both groups improved on the post-test, with the control group having a significant ($p<0.04$), and the experimental group a marginally significant improvement ($p=0.066$). There was no significant difference between the gains for the two groups. We attribute the relatively low results on the post-test to the short session length.

The control group students worked individually through the problems just like they would if they were using the system. The students progressed at their own speed, and managed to complete from 7 to 10 problems. They did not solve exactly the same number of problems due to differences in their abilities and prior knowledge. The human tutor facilitated by providing assistance when needed, similar to CIT.

The time the experimental group students spent interacting with CIT ranged from 36 to 45 minutes, with an average of 42 minutes (SD=3 min). They solved fewer problems than the control group, ranging from 0 (two students) to 4 (four students), with an average of only 2.5. This is low because the experimental students had to learn how to use CIT and understand its interface: one of the items in the user questionnaire asked students to estimate the time they needed to learn how to use the interface, and they reported that they needed an average of 9 minutes (SD=7 minutes). These students, therefore, effectively had less time to devote to problem-solving. This figure also does not take into account that working through a problem using the interface may have been slower than on paper. We also attribute the lower post-test results to this difference in time, and the consequent difference in the number of solved problems. This is also confirmed by the positive correlation (0.24) between the interaction time and the post-test performance for the experimental group students, and the much higher correlation (0.57) between the total numbers of attempts and the post-test.

Table 3. Statistics from the 2008 evaluation of CIT

Group	**No of students**	**Pre-test Mean (sd)**	**Post-test mean (sd)**	**Gain mean (sd)**	**Problems solved**
Experimental	14	32% (26%)	42% (18%)	10% (25%)	2.5 (1.4)
Control	7	38% (28%)	50% (13%)	21% (27%)	7-10

We also analyzed how students acquire constraints; the resulting learning curve is shown in Figure 18. The x-axis shows the opportunity to use a given constraint during the session on any attempt (please note that an attempt is a partial solution submitted for a particular step of the task). The average number of attempts was 78 (sd = 31). The y-axis shows the error probability. The data points were averaged over all constraints and all students who interacted with CIT. The minimal number of attempts per student was 42, and therefore the graph shown in Figure 18 represents all students from the experimental group. The initial probability was computed for 649 constraints, and the cut-off point for the graph was attempt 15, when the number of constraints used was 2/3 of the initial number. The data exhibits an excellent fit to a power curve, thus showing that students do learn effectively in CIT. Furthermore, the learning rate (i.e., the exponent of the power curve equation) is very high, showing that students acquired the necessary knowledge quickly. The initial error probability of 0.26 dropped by more than 50% to 0.12 after only four attempts.

The questionnaire responses were also analyzed. For the questions we discuss below, the students selected a response on a scale ranging from 1 (not at all) to 5 (very much). The students enjoyed interacting with the system (mean=3.5, sd=1.1), and believed that their understanding of the domain was improved as a consequence of using CIT (mean=3.8, sd=1). Eleven out of the fourteen experimental group participants indicated they would recommend CIT to other students. The students rated CIT’s interface as fairly easy to use (mean=3.7, sd=1.1). Most of the students found the feedback useful (mean=3.3, sd=1.3), although a few pointed out that they would prefer a little more information.

The results obtained from this initial evaluation of CIT are encouraging, and are similar to the results we have obtained in previous studies with manually developed constraint-based tutors. However, the amount of collected data was limited due to the small class size. We plan to repeat the evaluation study in 2009, when more students are expected to take the same course. We will then extend the length of the study to a couple of sessions.

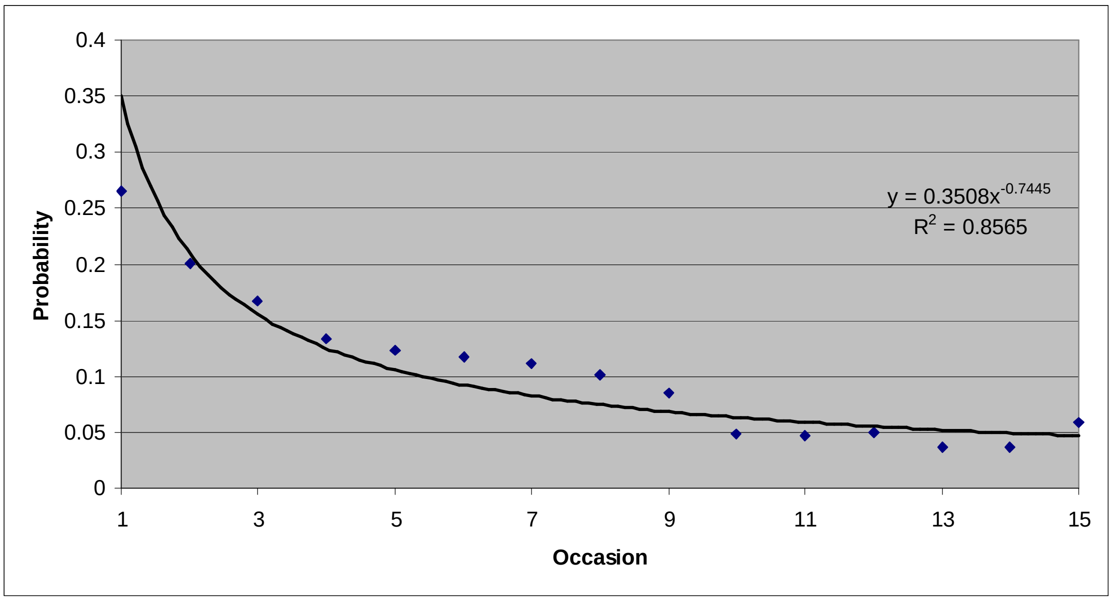


Fig. 18. Learning curves for CIT

CONCLUSIONS AND FUTURE WORK

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 in which the author is required to provide a description of the domain in terms of an ontology, specify the structure of problems and solutions, and provide examples of both. From this information, ASPIRE induces the domain model and produces a fully-functional web-based ITS that can then be used by students. ASPIRE also provides additional support for administrators to create user accounts and maintain the activities in ASPIRE. It also supports teachers in tailoring ITSs to their classes. The paper presented the features of ASPIRE and illustrated the authoring process on the example of the fraction addition domain. We also presented some initial evaluation results, as well as the process of developing CIT, an ITS that teaches Capital Investment decision-making. A preliminary evaluation of CIT showed that students do learn from interacting with this ITS. This evaluation serves as a proof the ITSs developed in ASPIRE are effective.

ASPIRE eases the development of ITSs by generating constraints automatically, thus not requiring programming and AI expertise from authors. As discussed in the paper, the constraint generation algorithms used in the current version of ASPIRE are not perfect: in evaluations we performed, they were capable of generating 90% of constraints needed. At the moment, these algorithms cannot generate constraints involving functions (arithmetic or domain-dependent ones), and therefore such constraints need to be defined manually. One of the avenues for future work is improving the constraint-generation algorithms. Even though the resulting constraint sets were not complete, the majority of constraints were provided to the authors. If authors have programming expertise, ASPIRE allows them to add or edit the automatically generated constraints manually.

Another important feature of ASPIRE is its generality. We designed ASPIRE to be able to support development of ITSs in any problem-solving task, be it procedural or non-procedural. This is a very important advantage, as most existing authoring systems are aimed at a particular type of instructional task. Of course, the consequence of generality is a steep learning curve for novice ASPIRE authors. Our authors report that initially they had difficulty understanding what ontologies were, how they could be defined and what influence they had on the authoring process. Another reported difficulty was understanding the effects each authoring phase has on the rest of the process.

ASPIRE generates student interfaces automatically, in the form of HTML-based textual interfaces. The default student interface can be replaced with applets. We do not expect the author to be able to develop applets on his/her own, as that requires significant programming expertise. In the future, we will extend ASPIRE with a GUI-based tool for specifying the student interface.

CIT is only one of the ITSs developed in ASPIRE. We have also been developing ITSs in ASPIRE for the areas of thermodynamics, engineering mechanics, chemistry and arithmetic. The authors involved in this work have various types of backgrounds, ranging from teachers to Computer Science postgraduate students. We have not performed rigorous evaluation studies and formal interviews yet, but the initial experiences have been positive. Although initially authors needed to learn about ontologies and the development philosophy supported by ASIRE, they found it a flexible and powerful tool. ASPIRE is freely available on the Web, and we hope more teachers will use it to develop ITSs for their courses. We plan to perform more studies on how authors interact with ASPIRE, with special focus on the reduction in time needed to produce domain models. We will also keep enhancing ASPIRE in light of feedback from authors.

Intelligent Tutoring Systems are being increasingly used in real classroom settings, producing significant learning gains. Ideally, teachers themselves 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 requires months of work that can only be carried out by experts in knowledge engineering and programming. The contribution of this research is that it enables domain models to be generated automatically with the assistance of a domain expert such as a teacher or a lecturer, obviating the need for AI programming skills. The reported research into the ASPIRE authoring system provides a way for ITSs to become more significant and to play wider role in education in the near future.

ACKNOWLEDGEMENTS

The ASPIRE project was supported by the eCDF grants 502 and 592 from the Tertiary Education Commission of New Zealand. We gratefully acknowledge the support of all members of ICTG. We thank Glenn Blank for informing us of the experiences with ASPIRE in his course, and also thank the anonymous reviewers, whose comments enabled us to improve this paper.

REFERENCES

Altova – XML, Data Management, UML, and Web Services Tools, (2005) http://www.altova.com

Aleven, V., McLaren, B., Sewall, J., & Koedinger, K. (2006). The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. In M. Ikeda, K. Ashley, & T-W Chan (Eds.) *Intelligent Tutoring Systems ITS 2006* (pp. 61-70). Berlin: Springer.

Angros, R., Johnson, W. L., Rickel, J., & Scholer, A. (2002). Learning Domain Knowledge for Teaching Procedural Skills. In M. Gini, T. Ishida, C. Castelfranchi, W.L. Johnson (Eds.) *Autonomous Agents and Multiagent Systems AAMAS 2002* (pp. 1372-1378). New York: ACM.

Bechhofer, S., Horrocks, I., Goble, C., & Stevens, R. (2001). OilEd: a Reason-able Ontology Editor for the Semantic Web. In F. Baader, G. Brewka, T. Eiter (Eds.): *KI 2001: Advances in Artificial Intelligence* (pp. 396–408). Berlin: Springer.

Blessing, S. B. (1997). A Programming by Demonstration Authoring Tool for Model-Tracing Tutors. *International Journal of Artificial Intelligence in Education*, 8, 233-261.

Gruber, T. R. (1993). A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5, 199-220.

Jarvis, M., Nuzzo-Jones, G., and Heffernan, N. (2004). Applying Machine Learning Techniques to Rule Generation in Intelligent Tutoring Systems. In J. Lester, R. M. Vicari. F. Paraguacu (Eds.) *Intelligent Tutoring Systems 2004* (pp. 541-553). Berlin: Springer.

Koedinger, K., Aleven, V., Heffernan, N., McLaren, B., & Hockenberry, M. (2004). Opening the Door to Non-programmers: Authoring Intelligent Tutor Behavior by Demonstration. In J. Lester, R. M. Vicari. F. Paraguacu (Eds.) *Intelligent Tutoring Systems 2004* (pp. 162-174). Berlin: Springer.

Koedinger, K. R., Anderson, J. R., Hadley, W. H., & Mark, M. A. (1997). Intelligent Tutoring goes to School in the Big City. *International Journal of Artificial Intelligence in Education,* 8, 30-43.

Martin, B., & Mitrovic, A. (2002). Authoring Web-Based Tutoring Systems with WETAS. In Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson, C-H Lee (Eds.) *Proceedings of the International Conference on Computers in Education* (pp. 183-187) Los Alamitos: IEEE Computer Society.

Martin, B., & Mitrovic, A. (2003). Domain Modelling: Art or Science? In U. Hoppe, F. Verdejo & J. Kay (Eds.) *Artificial Intelligence in Education AIED 2003* (pp. 183-190) Amsterdam: IOS Press.

Matsuda, N., Cohen, W., Sewall, J., Lacerda, G., & Koedinger, K. (2007). Evaluating a Simulated Student using real students data for training and testing. In C. Conati, K. McCoy, and G. Paliouras (Eds.) *User Modelling 2007* (pp. 61-70) Berlin: Springer.

Mitrovic, A. (2003). Supporting Self-Explanation in a Data Normalization Tutor. In V. Aleven, U. Hoppe, J. Kay, R. Mizoguchi, H. Pain, F. Verdejo, K. Yacef (Eds.) S*upplementary proceedings, AIED 2003* (pp. 565-577) Amsterdam: IOS Press.

Mitrovic, A. (2005). Scaffolding Answer Explanation in a Data Normalization Tutor. *Facta Universitatis, Series Electronics and Energetics,* 18, 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. de Rosis (Eds.) User Modelling 2003 (pp. 313-322). Berlin: Springer.

Mitrovic, A., & Ohlsson, S. (1999). Evaluation of a Constraint-based Tutor for a Database Language, *International Journal of Artificial Intelligence in Education,* 10, 238-256.

Mitrovic, A., Martin, B., & Suraweera, P. (2007). Intelligent tutors for all: Constraint-based modeling methodology, systems and authoring. *IEEE Intelligent Systems,* 22(4), 38-45.

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,* 10, 98-129.

Murray, T. (2003). An Overview of Intelligent Tutoring System Authoring Tools: Updated Analysis of the State of the Art. In T. Murray, S. Blessing, & S. Ainsworth (Eds.) *Authoring tools for advanced technology learning environments* (pp. 491-545) Norwell, MA: Kluwer Academic Publishers.

Ohlsson, S. (1994). Constraint-based Student Modelling. In J. Greer, G. McCalla (Eds.) *Student Modelling: the Key to Individualized Knowledge-based Instruction* (pp. 167-189) Berlin:Springer.

Ohlsson, S. (1996). Learning from performance errors. *Psychological Review,* 103, 241-262.

Ohlsson, S., & Mitrovic, A. (2007). Fidelity and Efficiency of Knowledge Representations for Intelligent Tutoring Systems. *Technology, Instruction, Cognition and Learning*, 5(2), 101-132.

OWL (2004). OWL Web Ontology Language, http://www.w3.org/TR/owl-features

Protege (2006). The Protege Ontology Editor and Knowledge Acquisition System, http://protege.stanford.edu

Suraweera, P., & Mitrovic, A. (2002). KERMIT: a Constraint-based Tutor for Database Modeling. In S. Cerri, G. Gouarderes and F. Paraguacu (Eds.) *Intelligent Tutoring Systems 2002* (pp. 377-387). Berlin:Springer.

Suraweera, P., & Mitrovic, A. (2004). An Intelligent Tutoring System for Entity Relationship Modelling. *International Journal of Artificial Intelligence in Education*, 14, 375-417.

Suraweera, P., Mitrovic, A., & Martin, B. (2004a). The Role of Domain Ontology in Knowledge Acquisition for ITSs. In J. Lester, R. M. Vicari. F. Paraguacu (Eds.) *Intelligent Tutoring Systems 2004* (pp. 207-216). Berlin: Springer.

Suraweera, P., Mitrovic, A., & Martin, B. (2004b). The Use of Ontologies in ITS Domain Knowledge Authoring. In L Arroyo, D. DIcheva (Eds.) *Applications of Semantic Web for E-learning SWEL'04* (pp. 41-49) Eficiencia.

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* (pp. 638-645). Amsterdam: IOS Press.

Suraweera, P., Mitrovic, A., & Martin, B. (2007). Constraint Authoring System: An Empirical Evaluation. In R. Luckin, K. Koedinger, & J. Greer (Eds.) *Artificial Intelligence in Education* (pp. 451-458) Amsterdam: IOS Press.

Tecuci, G. (1998). *Building Intelligent Agents: An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies.* San Francisco: Morgan Kaufmann.

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. W., Boicu, M., & Bowman, M. (1998). A Learning Agent Shell and Methodology for Developing Intelligent Agents. In *AAAI-98 Workshop: Software Tools for Developing Agents*, (pp. 37-46). Madison, Wisconsin: AAAI.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., & Wintersgill, M. (2005) The Andes Physics Tutoring System: Lessons Learned. *International Journal of Artificial Intelligence in Education*, 15, 147-204.