

Widening the Knowledge Acquisition Bottleneck for Constraint-based Tutors

Pramuditha Suraweera, Antonija Mitrovic and Brent Martin,

*Intelligent Computer Tutoring Group, Department Computer Science and Software Engineering, University of Canterbury, Private Bag 4800, Christchurch, New Zealand
{pramudi, tanja, brent}@cosc.canterbury.ac.nz*

Abstract: Intelligent Tutoring Systems (ITS) are effective tools for education. However, developing them is a labour-intensive and time-consuming process. A major share of the effort is devoted to acquiring the domain knowledge that underlies the system's intelligence. The goal of this research is to reduce this knowledge acquisition bottleneck and better enable domain experts with no programming and knowledge engineering expertise to build the domain models required for ITS. In pursuit of this goal we developed an authoring system capable of producing a domain model with the assistance of a domain expert. Unlike previous authoring systems, the Constraint Authoring System (CAS) has the ability to acquire knowledge for both procedural and non-procedural tasks.

CAS was developed to generate the knowledge required for constraint-based tutoring systems, reducing both effort and the amount of knowledge engineering and programming expertise required: the domain expert only has to model a domain ontology and provide example problems (with solutions). We developed novel machine learning algorithms to reason about this information and thus generate a domain model.

A series of evaluation studies have produced promising results. The initial evaluation revealed that the task of composing the domain ontology aids the manual composition of a domain model. The second study showed that CAS is effective in generating constraints for non procedural database modelling and the procedural data normalisation. The final study demonstrated that CAS is also effective in generating constraints when assisted by only novice ITS authors; under these conditions it still produced constraint sets that were over 90% complete.

Keywords. authoring systems, domain model authoring, constraint-based ITSs, evaluation

INTRODUCTION

Numerous evaluation studies (e.g. Koedinger et al., 1997; Mitrovic & Ohlsson, 1999) have demonstrated that ITSs are effective tools for learning. ITSs maintain an up-to-date model of the student's knowledge based on the domain model (a formal representation of the domain), and can therefore adapt to each individual student. The task of building a domain model is a difficult process requiring much time and effort (Murray, 1997). This difficulty has imposed a major bottleneck in producing ITSs.

Constraint-based modelling (CBM) (Ohlsson, 1994) is a student modelling approach that eases the knowledge acquisition bottleneck to some extent by using a more abstract representation of the domain compared to other commonly used approaches (Mitrovic, Koedinger & Martin, 2003). CBM uses *constraints* to represent basic domain principles. Each constraint consists of a *relevance condition*, which identifies types of solution states to which the constraint is applicable, and a *satisfaction condition*, which then checks whether the student's solution is correct with respect to the

principle being tested. The two conditions are expressed in terms of solution features. During evaluation the student's solution is matched to all constraints, with the resulting diagnosis consisting of the lists of violated and (usually) satisfied constraints. The long-term student model is then updated accordingly.

Although representing domain knowledge in the form of constraints makes the authoring process easier, building a knowledge base for a constraint-based tutor still remains a major challenge. It requires multi-faceted expertise, such as knowledge engineering, programming and knowledge of the instructional domain itself. The goal of our research is to reduce the knowledge acquisition bottleneck for CBM tutors while opening the door for domain experts with little or no programming and knowledge engineering expertise to produce ITSs. In pursuit of this goal, we have investigated various forms of authoring support for composing a domain model for constraint-based tutoring systems. We now present CAS, an authoring tool that reduces the workload for composing domain models.

CAS is designed to hide the details of constraint implementation such as the constraint language and constraint structures. Rather than work at the constraint level, the author is instead required to model the ontology of the domain and provide example problems including their solutions. CAS then uses machine learning techniques to reason about the information provided and produce the constraints. Whilst the constraint details are hidden from novices, expert authors can directly modify the generated constraints if necessary using tools provided.

ITS authoring using CAS is a semi-automated process requiring the assistance of a domain expert. The expert initiates the process by modelling the domain as an ontology. They then define the form that solutions will take for problems in this domain (or *solution structure*) using concepts from the ontology. CAS's constraint-generation algorithms use both the ontology and the solution structure to produce constraints that verify the syntactic validity of solutions. The domain expert is also required to provide sample problems and their solutions, which are used by CAS generators to produce *semantic* constraints. These verify that a student's solution is semantically equivalent to a correct solution to the problem. Finally, the author has the opportunity to validate the generated constraints.

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

The paper is organised into five sections. The following section provides a survey of currently available domain knowledge authoring tools, including their strengths and weaknesses. Section three describes CAS, the authoring system developed for producing constraint bases, and outlines the authoring process and system architecture. Extensive evaluation studies conducted to evaluate the effectiveness of CAS, and their results, are presented in Section four. Finally, we present conclusions and future directions.

DOMAIN KNOWLEDGE AUTHORIZING TOOLS

Murray (1997, 1999) 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 (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.

We are interested in the performance-oriented authoring systems, in particular authoring systems for domain expert systems. Typically, students use these systems to solve problems and receive feedback customised to their attempt. Such systems 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 rules of the domain with the assistance of a domain expert - e.g. Diligent (Angros et al., 2002), Disciple (Tecuci, 1999) 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 an interactive simulation-based learning environment. It learns the required rules for the agent by observing a domain expert performing tasks. The system uses experimentation to generalise the pre-conditions of actions demonstrated by the expert. The experimentation involves repeating the task but, for each step, omitting that step from the original sequence during each repetition. The system uses a Version Space machine learning algorithm named INBF to generalise the set of pre-conditions for each operator.

The domain expert can review the procedural rules generated by the system during the experimentation stage 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 etc. Authoring also involves acquiring the linguistic knowledge required to explain procedural tasks to students.

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

Disciple

Disciple (Tecuci, 1998; Tecuci & Keeling, 1999; Tecuci et al., 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 verify the behaviour of the agent. Disciple acquires knowledge using a collection of complementary learning methods, including inductive learning from examples, explanation-based learning, learning by analogy and learning by experimentation. A completed Disciple agent can be used to interact with students and guide them while solving problems.

The Disciple shell first has to be customised by building two domain-specific interfaces and a problem solver for the domain. The first domain-specific interface is for the expert to express their knowledge, while the other is for the interaction between the agent and student. The customised Disciple agent acquires knowledge using a four-stage process: knowledge elicitation, rule learning, rule refinement and exception handling. The goal of the initial knowledge elicitation phase is to construct a semantic network that outlines the concepts and instances of the domain and the relationships between them. During the rule-learning phase the expert demonstrates how typical domain-specific problems are solved. Initially they provide a correct example from the domain. The agent then attempts to explain it based on the semantic network, and produces “plausible version space” rules using analogical reasoning. The next phase (rule refinement) improves these rules, while extending and correcting the semantic network, until all rules are correct. During this phase a rule is either generalised or specialised using a positive or negative example that is generated by the agent, provided by the expert or obtained during problem solving. The final phase of handling exceptions aims to reduce the number of exceptions of a rule by either refining the semantic network or replacing the affected rule with a set of rules with fewer exceptions.

The Disciple was used to produce an agent capable of generating problems in history. Students interacting with the agent are presented with multiple-choice questions and the agent evaluates their solutions and provides hints to guide the student towards the correct solution. Experts rated the developed rule base as 96% correct. A classroom-based experiment revealed that most students rated the questions as helpful and understandable. A group of teachers also rated the agent as a helpful tool for learning history. The statistics analysis agent generates new problems and assists students in performing routine tasks as they work through their assignments.

Demonstr8

Blessing (1997) developed Demonstr8, an authoring system that assists in the development of model-tracing tutors for arithmetic domains. The system aims to reduce the time and knowledge required to build model-tracing tutors. It infers production rules using programming-by-demonstration techniques, coupled with methods for further abstracting the generated productions.

The first task for the author is to create a student interface using a drawing tool-like interface. It consists of a cell widget tool to add input cells in the student user interface. All cells placed on the interface automatically become available as working memory elements (WMEs). 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 trying out each function. The author has to select the correct function in cases where multiple knowledge functions produce the desired outcome. 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. They are 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.

CTAT

The Cognitive Tutor Authoring Tools (CTAT) (Koedinger et al., 2004; Aleven et al., 2006) developed at Carnegie Mellon University assist the creation and delivery of model-tracing tutors. The main goal of these tools is to reduce the amount of Artificial Intelligence (AI) programming expertise required. CTAT allows authors to create two types of tutors: ‘Cognitive tutors’ and ‘Example-Tracing Tutors’ (previously called ‘Pseudo Tutors’ (Koedinger et al., 2004)). Cognitive tutors contain a cognitive 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, by dragging and dropping the available interface widgets. The second step involves 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 has to annotate the graph by adding hint messages to correct links and “buggy” messages to incorrect links. The author must also specify the skill required for each step.

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

Jarvis and co-workers (2004) have implemented an automatic rule authoring system for CTAT tools that generates JESS (Java Expert System Shell) rules. The rule generation system attempts to generalise the set of behaviour graphs developed for example-tracing tutors. The rule generation process requires the domain expert to list the set of skills (one per problem solving action) required to solve a problem. The author then selects inputs and outputs for each skill application from the demonstrated problem-solving paths by highlighting the corresponding widgets in the interface. The rule generation algorithm then generates a rule for each outlined skill. The right- and left-hand side of the rule are generated separately. The right-hand side is generated by searching through the space of all possible combinations of available functions and all possible permutations of variables, while the left-hand side is generated by generalising the positive examples of a sample behaviour graph.

The rule generation system was tested in the domains of multi-column multiplication, fraction addition and tic-tac-toe. The rules for all three domains were learned in a reasonable amount of time.

However, it was reported that the generated rules had overly general left-hand sides for the domains of multiplication and tic-tac-toe. Moreover, the tic-tac-toe domain required the creation of higher order WMEs to represent each formation of three consecutive cells.

Matsuda et al. (2007) report on SimStudent, another machine-learning approach to generating production rules within CTAT. SimStudent uses a programming-by-demonstration approach to develop domain models. 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 (such as adding a constant to both sides of an equation). SimStudent then generates a production rule for each problem-solving step. The IF part of the production rule specifies the focus of attention and any conditions that must hold for those interface components in order for the production rule to fire. The THEN part of the production rule specifies the action to be taken, in terms of the input values needed.

SimStudent learns the focus of attention in the IF part by generalizing examples. The conditions specified in the IF part are learnt using the FOIL machine learning algorithm, while depth-first search with iterative-deepening is used to learn operator sequences for the right-hand side of production rules. This approach has been used to generate production rules in the areas of arithmetic, algebra and chemistry, resulting in fairly good domain models.

Discussion

Designing and developing an authoring system capable of authoring a wide variety of domains has remained an elusive goal. General-purpose authoring systems for simulated domains are most useful for cases where a simulated environment already exists. However, as simulated environments for education are rare, the applicability of these authoring systems in education is limited. Most of existing authoring systems can support only very specific types of domains. An example of such authoring systems is Demonstr8, which has shown to be highly effective for simple algebraic domains, but the author admits that creating a tutoring system for geometry is difficult (Blessing, 1997) 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 an open-ended domain such as database modelling with Demonstr8 would be impossible.

All the previous authoring systems for domain expert systems have focused on modelling strictly procedural tasks. The model-tracing authoring systems (Demonstr8 and CTAT) are developed for tasks that can be broken down into atomic problem-solving actions. Students can arrive at the solution by carrying out problem solving actions in the correct order. None of the discussed authoring systems are capable of handling non-procedural tasks, where there is no algorithm for solving a problem.

The amount of knowledge engineering expertise required to use some of the state-of-the-art authoring systems is high. In the case of Disciple, the author is required to build a semantic network that includes instances of the target domain. In order for the learning algorithm to infer correct rules, the semantic network should contain all the domain concepts as well as all the instances that appear in problems and solutions. It is highly unlikely that the average domain expert with no special knowledge engineering training would produce a complete semantic network. Furthermore, there are a lot of possibilities for mistakes, because all elements participating in example solutions have to be added manually. As a semantic network that includes instances even for a simple domain tends to contain many links between nodes, more complex domains are bound to have very large semantic networks.

In order to use Demonstr8 and the rule generation of CTAT, the authors need to be familiar with model tracing. The production-rule generation process of Demonstr8 is highly dependent on the WMEs (both base and higher order) created by the author: the author needs to use the correct representation for WMEs for correct productions to be generated. 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 of the model-tracing approach. It seems unreasonable to assume typical educators (such as school teachers) would possess such knowledge. Similarly in CTAT, the domain author is required to list the set of skills required to solve a problem. This list should be a comprehensive list of problem solving actions. As the rule generation engine is fully dependent upon the list of skills, an incomplete skill list will result in an incomplete set of generated rules. The task of producing the skill list in addition to the behaviour graph would be very demanding for the domain expert.

Some authoring systems require customisations that the typical domain expert would not be able to perform. General-purpose authoring systems for simulated domains require an interface capable of communicating between the authoring system and the simulation environment. This interface can be a significant software engineering task for sophisticated simulated environments. 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. Developing the interfaces and problem solver requires software engineering and knowledge engineering expertise, with a developer working collaboratively with a domain expert.

CONSTRAINT AUTHORING SYSTEM

The goal of the Constraint Authoring System is to assist domain experts in developing domain models for constraint-based tutors. It was designed for lecturers or instructors who typically do not have expertise in knowledge engineering, constraint-based modelling and programming. Rather than work at the constraint level, the author is required to model an ontology of the domain and provide example problems including their solutions. CAS then uses machine learning techniques to reason about the information provided to produce the constraints. Unlike other authoring systems that require significant customisations, we do not envisage authoring a domain in CAS requiring major software development.

All previous authoring systems have focused on authoring domain models for procedural tasks. Procedural tasks have a well-defined sequence of steps to arrive at a solution. Non-procedural domains such as ER modelling have no such problem solving procedure. CAS is capable of authoring domain models for both procedural and non-procedural tasks.

Authoring in CAS is a semi-automated process, in which the domain expert defines the ontology of the chosen instruction domain and provides sample problems and their solutions. CAS then analyses the domain-specific information and generates syntax and semantic constraints.

The process of generating a domain model using CAS consists of six steps:

1. The author models the domain as an ontology
2. The author specifies the structure of solutions
3. CAS automatically generates syntax constraints

4. The author provides sample problems and their solutions
5. CAS automatically generates semantic constraints
6. CAS and the author validate the generated constraints

The author first models the ontology of the domain using CAS's ontology workspace. In the second step the author defines the general structure of solutions, expressed in terms of concepts from the ontology. CAS then generates syntax constraints by analysing the ontology and the solution structure. The author then adds sample problems and their correct solutions. During this step, the author is encouraged to provide multiple solutions for each problem, demonstrating different ways of solving it. CAS's semantic constraint generator analyses problems and their solutions to generate semantic constraints. Finally, the author can validate the constraints generated by CAS. We now present details of each authoring step.

Modelling Domain Ontology

An ontology is an "explicit specification of a conceptualisation" (Gruber, 1993). A domain ontology is therefore a description of concepts of a particular domain. It also contains information about a concept's properties and the inter-relationships between concepts. An ontology typically has a hierarchical structure with super- and sub-concepts.

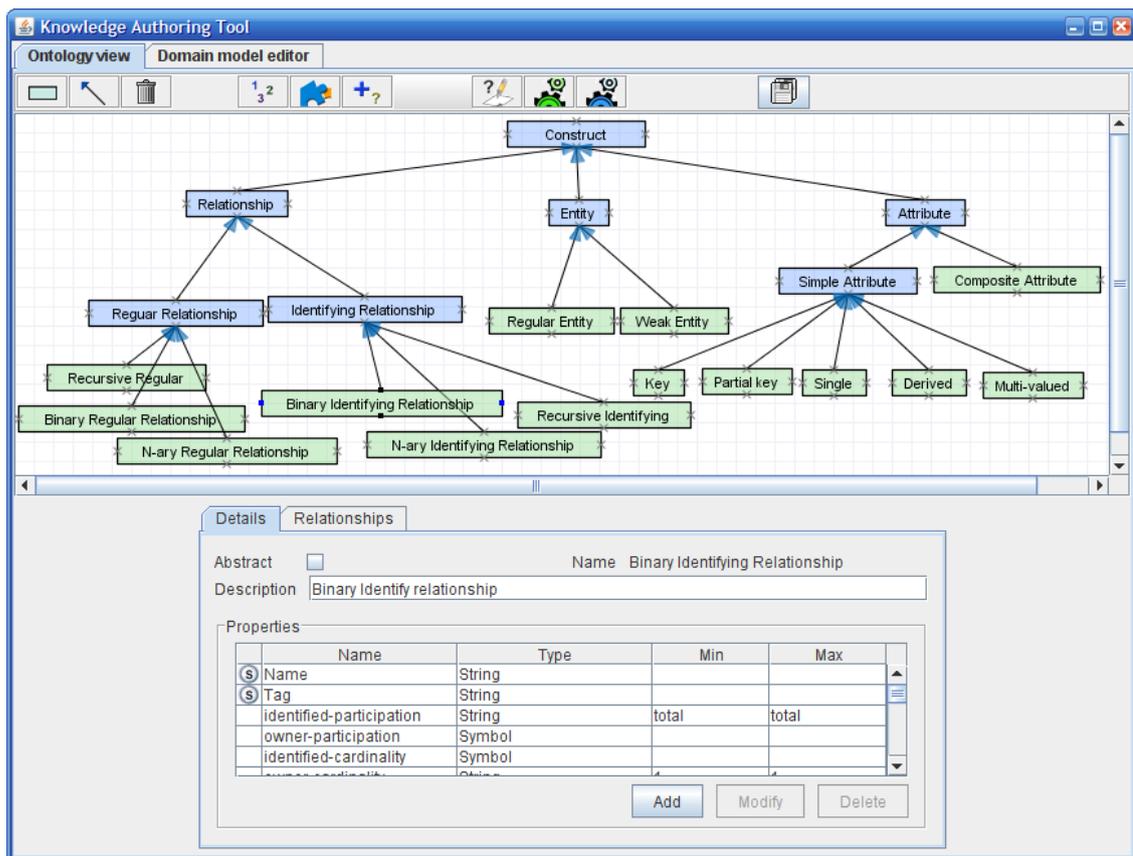


Fig. 1. Ontology Workspace

CAS contains an ontology workspace for composing domain ontologies, consisting of two panels (Figure 1). The top panel contains the ontology drawing tool, which allows users to compose a graphical representation of the ontology. Concepts are represented using rectangles and generalisation relationships are depicted with arrows. The author typically positions concepts to resemble a hierarchical structure; there are no restrictions on placing concepts within the workspace. The bottom panel shows the properties and relationships of the currently selected concept. It also allows users to add new properties and relationships.

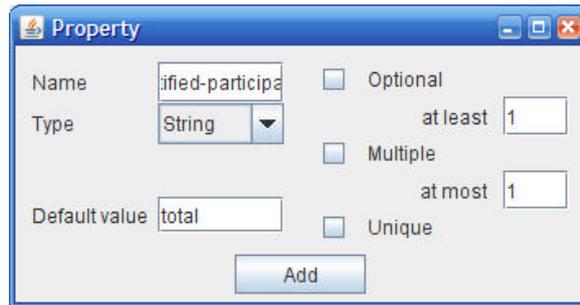


Fig. 2. Property Interface – specifying the *Identified-participation* property of the *Binary Identified Relationship* concept

Figure 1 shows an ontology for the domain of Entity Relationship (ER) modelling¹, a popular database modelling technique. It has *Construct* as the top-level concept and the three main types of constructs in ER modelling, *Relationship*², *Entity* and *Attribute* as its sub-concepts. *Relationship* is specialised into *Regular* and *Identifying*, the two relationship types in ER modelling. Similarly, *Entity* is specialised into *Regular* and *Weak*, while *Attribute* is specialised to *Simple* and *Composite*. The sub-concepts of *Simple attribute* include *Key*, *Partial key*, *Simple*, *Derived* and *Multi-valued*.

Details of concepts, such as its properties and relationships with other concepts, are outlined in the bottom panel. 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 can be added using the property input interface (shown in Figure 2), 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.

Other property restrictions that can be specified include requiring that the value of a property be unique, that it is optional or that it can contain multiple values. In the latter case the exact number of values that it 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 added using the relationship interface (shown in Figure 3) but are not visualised in the Ontology Workspace. To add a relationship the author defines its name and then selects the other concept(s) involved. The resulting relationship is

¹ ER modelling, originally proposed by Chen (1976), is widely used for the conceptual design of databases, and views the world as consisting of entities and relationships between them.

² The *Relationship* construct in ER modelling is different from relationships between concepts of an ontology.

between the concept initially selected in the graphical representation of the ontology and the concept(s) chosen in the relationship-composing interface. The author can also specify the minimum (*min*) and maximum (*max*) number of elements that should participate in the relationship. If *min* and *max* are set to 1, the relationship is binary; if they are greater than 1, the relationship is of higher order. The *Multiple* option allows the author to specify a set of concepts for the relationship; for example, attributes that belong to a *Regular Entity* could be *Single*, *Derived*, *Multi-valued* or *Key attributes*, and in this case, the author would specify these four concepts. The interface also allows the restriction of elements participating in combinations of relationships, such as whether an element must participate in two relationships. This is achieved by selecting a previously defined relationship under the ‘Compared to other relationships’ selection box and selecting either ‘equal’ or ‘mutually exclusive’. This is useful for denoting restrictions such as “an attribute can belong to either an entity or a relationship, but not both”.

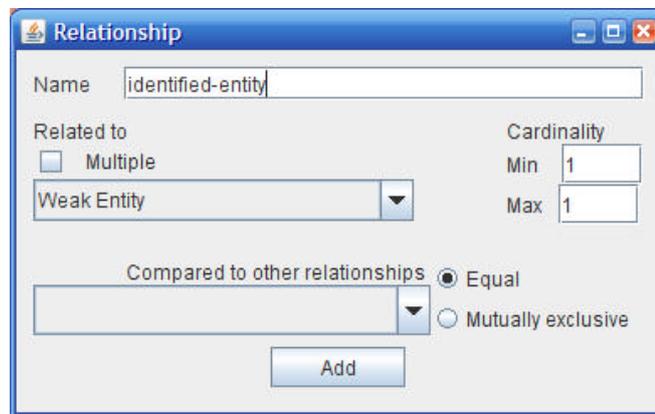


Fig. 3. Relationship Interface

Ontologies play the central role in the knowledge acquisition process, so it is imperative that the relationships are correctly defined. To ensure that the relationships are correct and are not overly general, the system engages the author in a dialogue by using a pop-up window. During this dialogue the author is presented with specialisations of concepts involved in the relationship and is asked to identify any specialisations that violate the principles of the domain. As an example, consider a scenario where the author has added a relationship between *Identifying Relationship* and *Attribute*³. CAS initially asks the author whether each specialisation of *Attribute* (*Key*, *Partial key*, *Single-valued* etc) is allowed to participate in this relationship. *Key* and *Partial key* attributes cannot be attributes of an *Identifying Relationship* according to the principles of ER modelling, so the author would indicate this. The ontology workspace then uses the outcome of this dialogue to automatically replace the original relationship with a more specific one (Figure 4).

The decision to design and implement an ontology editor specifically for CAS was taken after evaluating a variety of commercial and research ontology development tools. Although tools such as Protégé (2006), OilEd (Bechhofer et al, 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

³ This ontological relationship allows for attributes that belong to ER Identifying Relationships.

this research is to enable domain experts with little knowledge engineering background to produce tutoring systems for their domains. 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 ontology editor of CAS only allows 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.

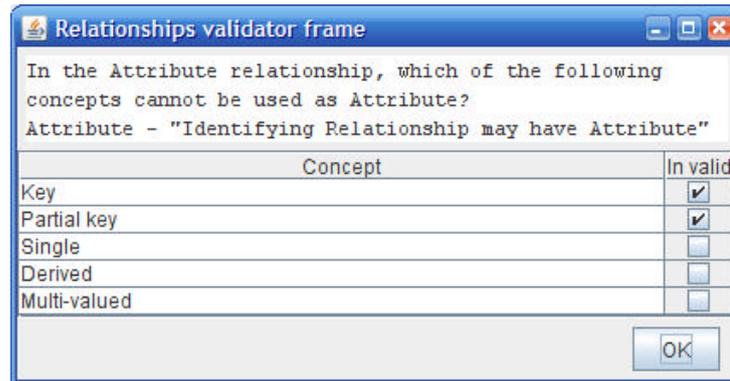


Fig. 4. Relationship Validation Dialog

Ontologies are stored in CAS using an XML representation defined specifically for this project. Although ontologies are stored using custom tags, they can be converted to a standard ontology representation form such as OWL (2004) via an XSLT transformation. CAS uses its own ontology XML schema because it directly reflects its internal ontology representation. This sped up the progress of the research project, avoiding the need for extensive research into ontology standards.

Modelling Solution Structure

The solution structure decomposes student solutions into meaningful sub-components. This is done for two reasons: to reduce cognitive load and to enable more focused feedback. Decomposition reduces cognitive load by presenting an explicit goal structure in terms of the components that need to be populated to complete a full solution. It also enables the tutoring system to analyse components of the solution independently and thus provide feedback targeted to a particular component. (However, this does not eliminate the need for analysing the solution as a whole.)

The number of solution components varies between domains and depends on the form of the expected solution. For example, if the tutoring system will present multiple-choice questions requiring a single response, the solution structure requires only a single component that contains the student's choice. On the other hand, a tutoring system for the domain of algebraic equations where the solution is an algebraic equation might decompose the solution into two components, namely a left- and right-hand-side expression.

Each solution component is described in terms of the ontology concepts that 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. To illustrate, solutions in the ER modelling domain consist of three components: entities, relationships and attributes. As shown in Figure 5, the entities component can hold instances of *regular* and *weak* entities. The relationship component can hold instances of 6 different types of relationships (*binary regular, recursive* etc.) outlined in the ontology. Similarly, the attributes component can hold the six types of attributes (*key, partial key* etc) outlined in the ontology.

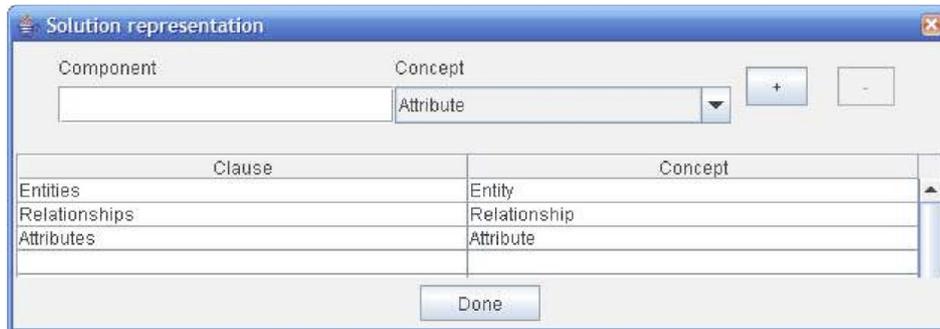


Fig. 5. Solution Representation for ER modelling

CAS was developed to support authoring for both procedural and non-procedural tasks. Procedural tasks, such as adding two fractions or solving a mathematical equation, require a strict procedure to be followed to accomplish the task. On the other hand, non-procedural tasks such as designing a database or writing a computer program do not have a rigid procedure that has to be followed to achieve their goals. Consequently, the expected structure of the solution for each problem-solving step has to be modelled for procedural tasks. The solution representation interface has the problem-solving step as a third column in the solution representation table for domains identified by the author as procedural,

Syntax Constraints Generation

An ontology contains a lot of information about the syntax of the domain. Composing a domain ontology using a graphical tool is much easier and quicker than manually composing syntax constraints (those responsible for ensuring that the student has used the correct syntax) in the formal constraint language. The goal of the syntax constraint generation algorithm is to extract all useful syntactic information from the ontology and translate it into constraints for the domain model.

Syntax constraints are generated by analysing relationships between concepts and concept properties specified in the ontology (Suraweera, Mitrovic & Martin, 2004b). The resulting constraints are applicable to both procedural and non-procedural tasks. An additional set of constraints is also generated for procedural tasks, which ensure the student adheres to the correct problem-solving procedure.

The syntax constraint generation algorithm is outlined in Figure 6. Step 1 concentrates on relationships between concepts. The specified minimum and maximum cardinalities result in syntax constraints that verify the correct numbers of elements participate in the corresponding relationships within the student solution. For example, the constraint “*Binary identifying relationship must have at least one Regular entity as the owner*” was generated based on a restriction on a relationship specified in the ontology.

As shown in Figure 3, the ontology editor allows restriction of the elements participating in two relationships: elements can be restricted to participate in only one of two relationships or simultaneously participate in both relationships. Step 1.b generates constraints that verify such restrictions within a student solution. “*An attribute can belong to either an entity or a relationship, but not both*” is an example of a constraint that was generated from such a restriction.

The results of the relationship validation dialogue (described previously) also contribute towards the generation of syntax constraints: step 2 results in syntax constraints being generated for each specialised relationship identified as invalid by the author.

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

Fig. 6. Syntax Constraint Generation Algorithm

After generating constraints from relationships, step 3 analyses the properties defined in the ontology. During this phase the constraint generator creates a constraint for each restriction on the domain and range of a property. Such restrictions include the minimum and maximum values allowed and whether the property is multi-valued or unique. The constraints such as “*The name of a Relationship has to be unique*” and “*The identified participation of a Binary identifying relationship must always be total*” were generated from restrictions specified on properties in the ER ontology.

All constraints generated from properties and relationships are applicable to both procedural and non-procedural tasks. For procedural tasks, the ITS must ensure that the student follows the correct sequence of steps. To achieve this CAS generates a constraint for each problem-solving step verifying that the student has at least made an attempt at populating the relevant part of the solution (step 4 in algorithm). For example, in the domain of adding fractions a constraint is generated that verifies 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 moving on before satisfying it.

Adding Problems and Solutions

The next step is to produce the semantic constraints, which check that the student’s solution has the desired meaning (i.e. it answers the question). Before this can happen the author must provide sample problems, along with several solutions for each depicting alternative ways of solving them. Problems are described by their problem statement, whereas the solutions are specified as sets of elements that relate to each other. The author enters a solution by populating the components of the solution structure using the solution composition interface illustrated in Figure 7. The interface outlines the solution structure and the author populates each component (*Entities*, *Relationships* and *Attributes*) by adding elements. For each component the interface restricts input to only those element types specified in the solution structure.

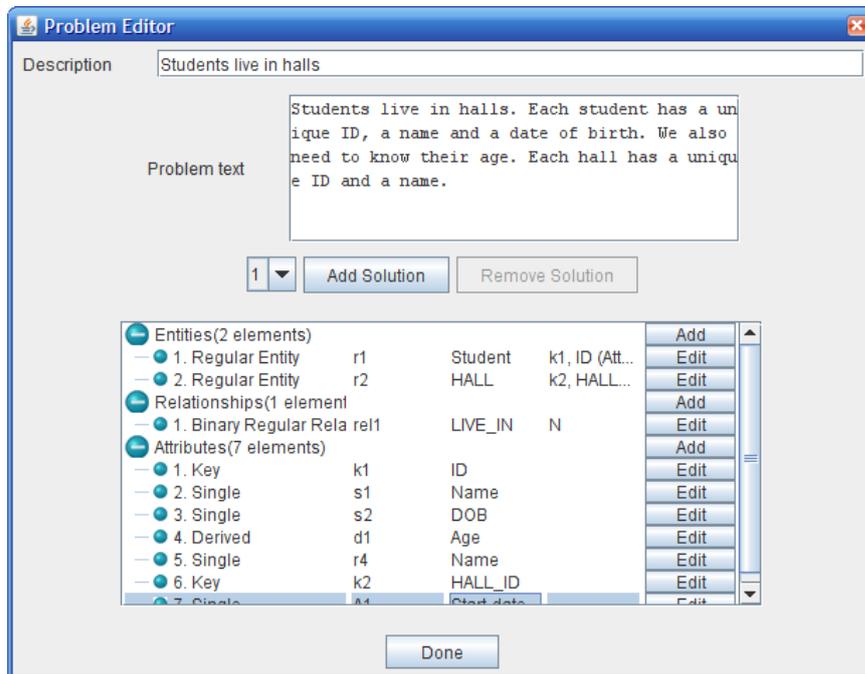


Fig. 7. Problem Solution Composing Interface

The solution interface displays the solution in a tabular form. Elements are represented in rows. The type of element is given in column one and the first three properties of each element are shown in the other columns. Figure 7 depicts the representation of ER diagram for students living in halls scenario. The ER diagram of the solution is given in Figure 8.

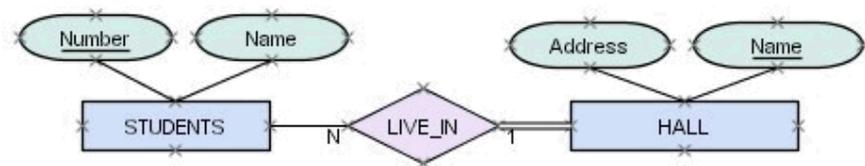


Fig. 8. ER diagram for 'Students living in Halls' scenario

The solution composition interface is designed to only allow solutions that strictly adhere to the ontology. Elements are created using a form-based interface, as shown in Figure 9, which is generated at runtime to reflect the current concept's definition. The forms contain input boxes for specifying values for concept's properties and drop-down-lists for selecting elements involved in relationships. Relationships between elements are specified by selecting the desired element and adding it to the respective container (the elements must have already been created). Figure 9 shows a screenshot of the interface at the time when the author is creating an instance of the *Regular Entity* concept. It contains two text boxes for populating the *name* and the *tag* properties of the chosen concept. It also contains three drop-down-lists for selecting elements that participate in each relationship (*attributes*, *connected to* and *key attribute*). Elements participating in a relationship are selected from the drop-down-list and added to the container under it. For example, the *Student* entity in Figure 9 has three *attribute* elements participating in the *attributes* relationship: *Name*, *DOB* and *Age*.

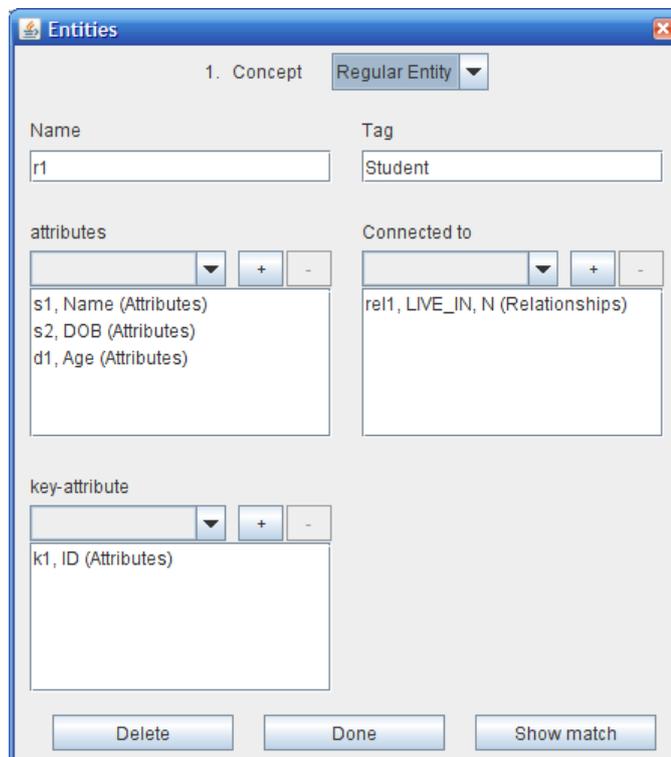


Fig. 9. Solution Element Creation Interface

In domains where multiple solutions exist, the system expects such solutions depicting alternate ways of solving the same problem. The interface 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. When a new solution is added, the solution area of the problem editor interface is populated with a copy of the first solution. The author can modify this solution by adding, modifying or dropping elements. This feature significantly reduces the workload of the author because most alternative solutions have a high degree of similarity.

Semantic constraints are generated by analysing the similarities and differences between two solutions to the same problem. It is therefore imperative that the system can accurately identify related

elements between the two solutions. CAS achieves this by maintaining a map of matching elements between the initial and alternative solutions. This breaks down however when new elements are added. In this situation CAS attempts to automatically find matches for new elements and if a successful match is found, the author is consulted to validate it. If the system fails to find a matching element, the author is requested to select the matching element, if one exists.

Semantic Constraints Generation

Semantic constraints are generated by an algorithm that learns from examples (Suraweera, Mitrovic & Martin, 2005) using the problems and solutions provided by the author during the previous step. 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; Suraweera & Mitrovic, 2004). The multiple alternative solutions specified by the author in the previous step enable the algorithm 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. The algorithm (outlined in Figure 10) generates semantic constraints by analysing pairs of solutions and identifying similarities and differences between them. The constraints generated from each pair contribute towards either generalising or specialising constraints in the main constraint base. The resulting set of constraints is capable of identifying different correct solutions produced using alternative problem-solving strategies.

The algorithm focuses on one problem at a time and generates a new set of constraints (represented as N-set in Figure 10) for each pair of solutions for that problem. All possible permutations of solution pairs (including each solution compared to itself) are used for generating constraints. One solution in each pair is treated as the student solution, while the other serves as the ideal solution. N-set contains constraints that compare each element of the ideal solution against a matching element in the student solution. An example constraint of this type would ensure that the student has modelled all the required regular entities (i.e. those that figure in the ideal solution). Additional constraints ensure that the relationships between elements of the ideal solution also exist between the corresponding elements of the student solution. The constraint that ensures the student has attached all the attributes with the correct entity (by comparing with the ideal solution) is an example of such a constraint.

The constraints generated from a pair of solutions for a problem are only applicable for that specific problem. These constraints are generalized by replacing the instance property values with variables and wild cards. As an example, consider the following constraints that were generated from two instances in the student solution ("student" regular entity and "undergrad" regular entity, with the tag E1) and their corresponding instances in the ideal solution ("course" regular entity and "class" regular entity, with the tag of E2).

Relevance: Entities component of IS has a (STUDENT, E1) Regular Entity

Satisfaction: Entities component of SS has a (UNDERGRAD, E1) Regular Entity

Relevance: Entities component of IS has a (COURSE, E2) Regular Entity

Satisfaction: Entities component of SS has a (CLASS, E2) Regular Entity

These two constraints will be generalized by replacing the "tag" property with a variable (?var), as the value of "tag" is constant within the constraint but different between two constraints. The

“name” property gets replaced with a wild card (?*) as it can have different values within the constraint and between two constraints.

Relevance: Entities component of IS has a (?, ?var) Regular Entity*

Satisfaction: Entities component of SS has a (?, ?var) Regular Entity*

1. For each problem P_i
2. For each pair of solutions S_i & S_j
3. Generate a set of new constraints (N-set)
4. Evaluate each constraint N_i in N-set against each previously analysed solution,
 - If N_i is violated, generalise or specialise N_i until it is satisfied (using algorithm in Figure 11)
5. Evaluate each constraint C_i in the main constraint base against solutions S_i & S_j used for generating N-set,
 - If C_i is violated, generalise or specialise C_i to satisfy S_i & S_j (using algorithm in Figure 11)
6. For each constraint N_i in N-set
 - If main constraint base has a constraint C_i that has the same relevance condition but a different satisfaction condition to N_i , add the satisfaction condition of N_i as a disjunctive test to the satisfaction condition of C_i
 - else if N_i does not exist in main constraint base, add it to main constraint base
7. Generate set of constraints that check for extra elements

Fig. 10. Semantic Constraint Generation Algorithm

The N-set, generated from a pair of solutions, is evaluated against all solutions used previously for generating constraints. This step (step 4) ensures that the newly generated set of constraints is consistent with all solutions analysed so far. Although all specified solutions are correct, this process may result in violated constraints if the generated constraints are too specific or general. If a constraint is violated by a previously processed solution, it is either generalised or specialised to satisfy that solution (using algorithm in Figure 11). However, in cases where generalisation and specialisation fail, the constraint is labelled as invalid and removed from the constraint set. The system keeps track of invalid constraints to ensure that similar newly generated constraints do not get added to the main constraint base. The next step (step 5) ensures that constraints in the main constraint base are consistent with the pair of solutions used to generate the new constraints (N-set). Each constraint in the main constraint base is evaluated against the solution pair, and violated constraints are either specialised or generalised similar to step 4.

After all constraints from the N-set and main constraint base are generalised/specialized as needed, the main constraint base is updated with constraints in the N-set (step 6). During this step, if the main constraint base contains a constraint C_i that has the same relevance condition as a constraint N_i (from the N-set) with a different satisfaction condition, a new satisfaction condition is generated as a disjunction of the two original satisfaction conditions from N_i and C_i . Constraints in the N-set that do not exist in the main constraint base get added to the main constraint base. Therefore, the main constraint base grows with each iteration.

Finally, after all problems and their solutions have been analysed, additional constraints are generated to check the student solution for extra elements. This is achieved by reversing the

constraints that exist in the constraint base. For example, the constraint base for ER modelling contains a constraint that ensures the student's solution contains a matching regular entity for each regular entity in the ideal solution. This constraint can be reversed as "the ideal solution should contain a regular entity for each regular entity in the student solution", which ensures that the student solution does not contain any extra regular entities.

Constraint Generalisation/Specialisation

Constraints produced in step 3 of the semantic constraint generation algorithm (Figure 10) that violate any solutions (step 4) are either too specific or general and need to be modified. This is achieved using the algorithm outlined in Figure 11. It is applied to each constraint individually, causing it to be generalised or specialised until it satisfies the violated pair of solutions. If this step fails the constraint is labelled as invalid and removed from the constraint set.

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

Fig. 11. Constraint Generalisation/ Specialisation Algorithm

The algorithm first attempts to correct the incorrect constraint by replacing it with a similar existing one that is more restrictive, i.e. that has the same tests in the relevance condition but with one or more additional conjunctive tests. It must also contain the same satisfaction condition, but may additionally contain other disjunctive tests. If such a constraint is found, the algorithm simply uses it to replace the incorrect constraint and completes the specialisation/generalisation process. If there is no such restrictive constraint, the algorithm attempts to generalise the incorrect constraint in step 2. First it searches for a constraint in the main constraint base (*C-set*) with the same relevance condition as the violated constraint but a different satisfaction condition. If the search is successful, both constraints are replaced by a new constraint that has the same relevance condition, and a disjunction of the two satisfaction conditions.

Generating constraints by looking at a problem instances can yield to contradicting constraints such as "If A then B better be true" and "If A then C better be true". Consider the constraint that says, "If the ideal solution contains a *Weak Entity* then the student solution should contain a matching *Weak Entity*". This constraint would be violated if the student chose to represent the *Weak Entity* as a *Composite Multi-valued Attribute* (*Weak Entities* can also be represented as *Composite Multi-valued Attributes*, as shown in Figure 12). As the new constraint set was generated from a *Weak Entity* in the ideal solution and a matching *Composite Multi-valued Attribute* in the student solution, the resulting

constraint would be “If the ideal solution contains a *Weak Entity* then the student solution should contain a matching *Composite Multi-valued Attribute*”. As these constraints contradict each other, they must be combined to ensure that the student is given the flexibility to model a *Weak Entity* using either a *Weak Entity* or a *Composite Multi-valued Attribute*. The resulting constraints would be “If the ideal solution contains a *Weak Entity* then the student solution should contain a matching *Weak Entity* or a matching *Composite Multi-valued Attribute*”.

During step 3 the relevance condition of constraint V is restricted to make it irrelevant for the solution that violated it (S_i). This is achieved by first searching for a solution (S_x) for which the constraint is satisfied. Any significant differences between S_i and S_x are used to restrict the relevance condition of V .

Consider an example of the constraint that ensures that the student solution contains an equivalent *Partial-key Attribute* to each found in the ideal solution. This constraint is violated if a *Weak Entity* is represented as a *Composite Multi-valued Attribute*, as shown in Figure 12. Even though the *Weak Entity* may have a *Partial-key Attribute*, A1, when it is represented as a *Composite Multi-valued Attribute*, A1 is represented as a *Single Attribute*. The ER diagram in Figure 12a, which satisfies this constraint, contains a *Weak Entity* that is not found in the ER diagram in Figure 12b. This difference is added as a new conjunctive test to the relevance condition of the constraint, thus making it irrelevant for the ER diagram in Figure 12a. The modified constraint will check that the student’s solution contains a *Weak Entity* as well as checking that a *Partial-key Attribute* exists (in the relevance condition).

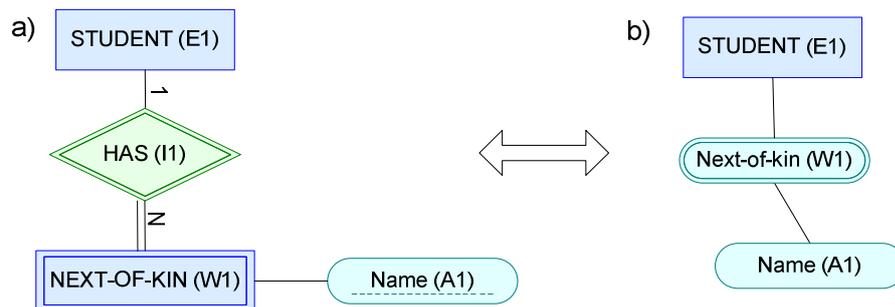


Fig. 12 Two equivalent ER diagrams: A Weak Entity can alternatively be represented as a Composite Multi-valued Attribute

Constraint Validation

In the final authoring phase the constraints generated by the system are validated with the assistance of the domain expert. This ensures that the generated constraints are accurate and that the resulting domain model can therefore be used in a tutoring system. During this phase the domain expert also adds meaningful feedback messages to constraints. The author is presented with a system-generated, high-level hint for each constraint, plus a high-level description of the relevance and satisfaction conditions. The hints are generated based on a pre-defined set of templates by analysing each constraint and determining whether they check for missing or extra elements/relationships. For example, the hints for constraints that deal with missing elements are produced using the template

“Some required <concept-name> elements are missing”. During hint generation, the <concept-name> tag is replaced with the appropriate name(s) of the concept.

During this process the author may come across incorrect constraints. These can be labelled as invalid and will be removed from the constraint base. The authors can also revisit step four of the authoring process and provide more example problems and solutions to demonstrate why the constraint is invalid. The constraint base can be regenerated at any time by executing the semantic constraint generator with the updated problem set.

The Architecture of CAS

CAS is implemented based on the client-server architecture depicted in Figure 13. The server is implemented in Allegro Common Lisp using the AllegroServe light-weight web server. The server is responsible for storing and retrieving all components of the domain model such as the ontology, problems, solutions and constraints. The client (implemented as a Java application) provides the interfaces for the domain expert to compose the required domain-dependent components. The Java application is also capable of generating syntax and semantic constraints, enabling it to function as a standalone application if a connection to the server is not available. The client stores and retrieves the domain model components from the server through HTTP requests.

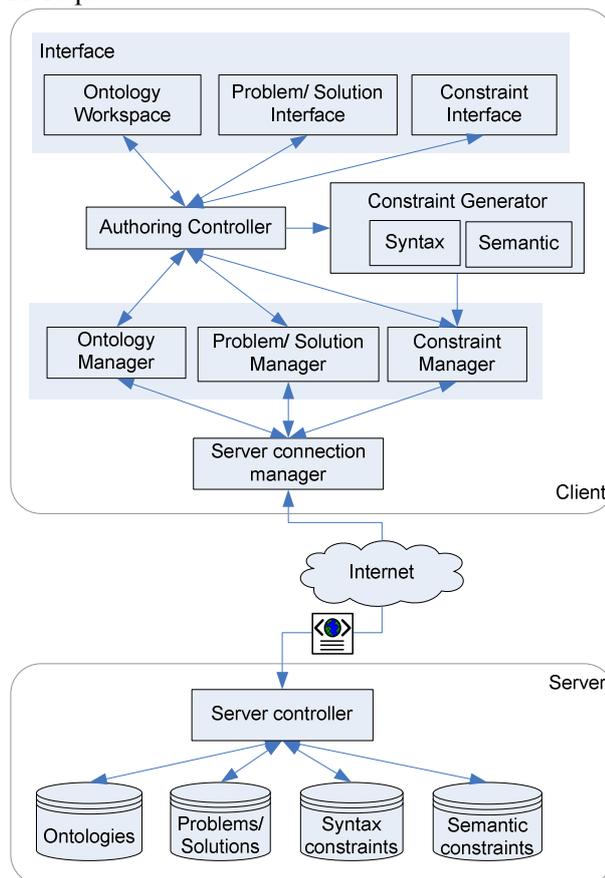


Fig. 13. CAS Architecture

The server is implemented as a simple web server that listens to HTTP requests and returns the appropriate response. Users must be authenticated prior to being granted permission to access any data. Authenticated users can retrieve their ontology, list of problems and constraints (syntax and semantic). They can also store domain model components composed using the client.

The client consists of an interface, the authoring controller, constraint generators and a set of managers responsible for persisting components of the domain model. The interface is a collection of three components that are used during different stages of the authoring process: *ontology workspace*, *problem/solution interface* and a *constraint interface*. The *ontology workspace* is provided for modelling an ontology of the domain. Problems and their solutions can be added using the *problem/solution interface*. It assists the authors by providing a form with input boxes based on the properties of the element's type (i.e. the concept). The constraint validation phase uses the *constraint interface* for validating constraints and adding meaningful feedback messages.

CAS supports authoring of domain knowledge but does not provide any means of running the resulting tutor. To do this the domain model must be loaded in to a tutoring server such as WETAS (Martin & Mitrovic, 2003). WETAS is a tutoring shell that provides all the domain-independent components for a web-based ITS, including a text-based user interface, pedagogical module and student modeller. The pedagogical module makes decisions based on the student model regarding problem/feedback generation and the student modeller evaluates student solutions by comparing them to the domain model and updates the student model. WETAS only requires a domain model in the expected format to serve a new web-based intelligent tutor.

EVALUATION

Evaluating the effectiveness of an authoring system is a complex task. There are various factors to be considered such as the nature of the domain, the experiences of the domain expert, time on task etc. We conducted three studies that focussed on different aspects of the system to evaluate the usefulness and effectiveness of CAS.

Domain ontologies play a central role in CAS. We conducted study 1 to test our hypothesis that developing domain ontology is useful even when constraints are authored manually. The experiment was carried out with novice authors using a tool designed to encourage the use of ontology. It revealed that domain ontologies do indeed facilitate the creation of complete/correct constraint bases (Suraweera, Mitrovic & Martin, 2004a).

In Study 2 we evaluated the effectiveness of constraint generation by using it to produce constraint-bases for ER modelling and Data normalisation. All three constraint bases were developed by the same author to ensure consistency between the domains. Analysis of the generated constraints revealed that CAS is extremely effective in producing constraints for these domains (Suraweera, Mitrovic & Martin, 2005).

Finally, a comprehensive evaluation (study 3) was carried out with a group of novice authors to evaluate the effectiveness of the system as a whole. They were assigned the task of using CAS to produce a complete domain model that could run in WETAS for the domain of fraction addition. This evaluation showed that CAS is capable of generating highly accurate constraint bases when used by novices. It also showed that CAS reduces the overall effort required to produce domain models (Suraweera, Mitrovic & Martin, 2007).

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

We hypothesised that it is highly beneficial for the domain model author to develop a domain ontology before manually composing the constraint set, as this helps the author to structure their thinking and reflect on the domain. We also believe that categorising constraints according to the ontology assists the authoring process. In doing so, the author needs to focus only on constraints related to a single concept, reducing their working memory load. The use of ontologies may therefore encourage authors to produce more complete constraint bases.

To evaluate our hypothesis, we conducted an empirical study (Suraweera, Mitrovic & Martin, 2004a) where authors produced constraints using a tool (called the domain composition tool) that supported organising constraints according to concepts of an ontology. The evaluation involved 18 students enrolled in the 2003 graduate course on Intelligent Tutoring Systems at the University of Canterbury. They were assigned the task of building an ITS using WETAS for teaching English adjectives. The tutor would present sentences to the student, to be completed by providing the correct form of a given adjective, such as “*My sister is much _____ than me (wise)*” – correct answer “wiser”. They were allocated a total period of three weeks. Prior to being assigned the task the participants had attended 13 lectures on ITSs, including five on CBM. They also had a presentation on WETAS, and were given a description of the task, instructions on how to write constraints, and the section on adjectives from a popular textbook on English spelling and vocabulary (Clutterbuck, 1990). They were also free to explore the domain model of LBITs (Martin, 2002), a similar tutor that teaches simple vocabulary skills.

The domain model composition tool for WETAS was designed to encourage the use of a domain ontology as a means of visualising the domain and organising the knowledge base. The tool consists of five tabs for each domain model component, as shown in Figure 14. The Ontology view supports modelling the ontology graphically (in the same way as supported by the Ontology Workspace in CAS) and also displays constraints related to the selected concept. Figure 14 shows an ontology developed by a participant, with three constraints developed for the selected domain concept (“*Ending with Y*”). The components of the first constraint (its feedback, relevance and satisfaction conditions, and type) are shown in the lower part of the screenshot. The tool also contains textual editors for composing syntax constraints, semantic constraints, macros and problems. The constraint editors divide the available text area based on the concepts of the ontology. The editors also provide syntax highlighting facilities to assist in constraint composition. The users were able to compose all the required domain model components using the authoring tool and deploy the produced domain model in WETAS to get a running tutor.

Seventeen students completed the task satisfactorily. One student lost his entire work due to a technical problem; that student’s data was not included in the analysis. The same problem did not affect other students (it was eliminated before they experienced it). Table 1 gives some statistics about the remaining students, including interaction times, numbers of constraints they developed and marks for constraint sets and ontology.

The participants took an average of 37 hours to complete the task, spending 12% of the time in the ontology view. This parameter varied widely, with a minimum of 1.2 and maximum of 7.2 hours. This can be attributed to different styles of developing the ontology. Some students may have developed the ontology on paper before using the system, whereas others developed the entire

ontology online. Furthermore, some students also used the ontology view to add constraints,⁴ which increased the total time spent in this view. However, the logs showed that this was not a popular option, with most students composing constraints in the constraint editors. One factor that may have contributed to this behaviour is the restrictive nature of the constraint interface, which displays only a single constraint at a time.

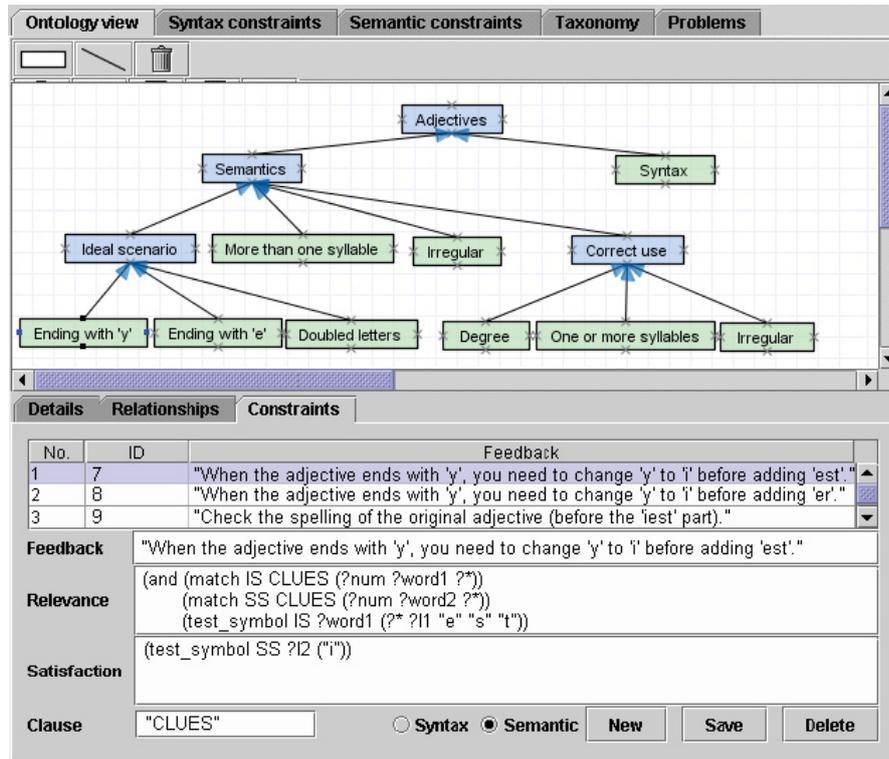


Fig. 14. Interface of Domain Model Composition Tool

In constraint-based tutors, constraints are classified as *semantic* (answers the question) or *syntactic* (obeys the syntax rules of the domain). In the domain of adjectives this distinction is sometimes subtle. For example, in order to determine whether the submitted adjective form is correct, the student's solution would be compared to the ideal solution. This involves checking whether the correct rule for determining the ending has been applied (semantics) as well as whether the resulting word is spelled correctly (syntax). Such constraints might be classified into either category, as is evident in the numbers of constraints for each category reported in Table 1. The averages of both categories are similar (9 semantic constraints and 11 syntax constraints). Some participants have included most of their constraints as semantic and others vice versa. Note that this classification does not affect the tutor's behaviour: all constraints are used to diagnose students' solutions. The participants composed a total of 20 constraints on average.

We assessed the participants' domain models by comparing them to one produced by an expert, the resulting marks being given under *Completeness* (the last two columns in Table 1). The expert's

⁴ Participants were able to add constraints in two ways: either directly in the ontology view, as shown in Figure 14, or via the syntax/semantic editor.

knowledge base consisted of 20 constraints. The *Constraints* column gives the number of constraints from the expert’s set that are accounted for in the participants’ constraint sets. Note that the mapping between the ideal and participants’ constraints is not necessarily 1:1 (i.e. the participants might use multiple constraints to represent a concept that the expert encoded in a single constraint, and vice versa). Two participants accounted for all 20 constraints. On average, the participants covered 15 constraints. The quality of the constraints was high.

The ontologies produced by the participants were given a mark out of five (the *Ontology* column in Table 1). All students scored highly, which we expected because the ontology was straightforward. Almost every participant specified a separate concept for each group of adjectives according to the given rules (Clutterbuck, 1990). However, some students constructed a flat ontology, which contained only the six groupings corresponding to the rules. Five students scored full marks for the ontology by including the degree (comparative or superlative) and syntax considerations such as spelling.

Table 1. Results

	Time (hours)		Number of constraints			Completeness	
	Total	Ontology view	Semantic	Syntax	Total	Constraints (out of 20)	Ontology (out of 5)
S1	38.16	4.57	27	3	30	20	5
S2	51.55	7.01	3	10	13	19	4
S3	10.22	1.20	14	1	15	17	4
S4	45.25	2.54	30	4	34	18	5
S5	48.96	4.91	11	5	16	20	4
S6	44.89	4.66	24	1	25	18	5
S7	18.97	2.87	1	15	16	17	4
S8	22.94	4.99	3	18	21	15	3
S9	34.29	4.30	11	4	15	18	5
S10	33.90	7.23	0	14	14	18	3
S11	55.76	3.28	16	1	17	17	5
S12	30.46	2.84	0	16	16	10	3
S13	60.94	3.47	1	15	16	13	3
S14	32.42	1.96	1	17	18	12	3
S15	33.35	4.04	1	14	15	11	3
S16	29.60	6.24	0	30	30	4	5
Mean	36.98	4.13	8.9	10.5	19.4	15.4	4.0
S.D.	13.66	1.72	10.5	8.2	6.6	4.4	0.9

Fourteen participants categorised their constraints according to the concepts of the ontology while others chose to ignore this. For the former, there was a significant correlation between the ontology score and the constraints score (0.679, $p < 0.01$). However, there was no significant correlation between the ontology and constraints scores when all participants were considered. This strongly suggests that the participants who made use of the ontology developed better constraint bases.

An obvious reason for this finding may be that more able students both produced better ontologies and produced a complete set of constraints. To test this hypothesis, we determined the correlation between the participant’s final grade for the course (which included other assignments) and

the ontology/constraint scores. There was indeed a strong correlation (0.840, $p < 0.01$) between the grade and the constraint score. However, there was no significant correlation between the grade and the ontology score. This lack of a relationship might be due to a number of factors. Since the task of building ontologies was novel for the participants, they may have found it interesting and performed well regardless of their ability. Another factor is that the participants had more practise at writing constraints (in another assignment for the same course) than with ontologies. Finally, the simplicity of the domain could also be a contributing factor.

The participants spent 2 hours per constraint (SD=1 hour). This is twice the time reported in (Mitrovic, 1998), but the participants are neither knowledge engineers nor domain experts, so the difference is understandable. In particular, the students were new to the constraint language. The participants reported that building an ontology made constraint identification easier. For example, the following comments were extracted from their reports: “*Ontology helped me organise my thinking*”, “*The ontology made me easily define the basic structure of this tutor*”, “*The constraints were constructed based on the ontology design*”, “*Ontology was designed first so that it provides a guideline for the tasks ahead.*”

The results indicate that ontologies do assist constraint acquisition: there is a strong correlation between the ontology score and the constraints score for the participants who organised the constraints according to the ontology. Subjective reports confirmed that the ontology was used as a starting point when writing constraints. As expected, more able students produced better constraints. In contrast, most participants composed good ontologies, regardless of their ability.

Effectiveness of the Constraint Generation Algorithms (Study 2)

We evaluated the effectiveness of CAS's constraint generation algorithms for two different domains: Database modelling and Data normalisation. The domains of Database modelling and Data normalisation were specifically chosen because we had previously developed two successful constraint-based tutors for the two domains: KERMIT (Suraweera & Mitrovic, 2002) and NORMIT (Mitrovic, 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 CAS.

The choice of domains was also influenced by the desire to evaluate CAS for procedural as well as non-procedural tasks. Data normalisation can be categorised as a procedural task, where a strict set of steps must be followed to arrive at the solution. On the other hand, the task of modelling a database is not well defined; there is no algorithm to be followed to produce an ER model.

As previously stated, CAS relies on the domain expert to provide a correct and complete ontology of the domain, as well as sample problems and a collection of solutions for each problem outlining different ways of solving it. To ensure that CAS is supplied with all correct and complete information, we provided the required information for both domains. Constraints for each domain were generated via CAS's six step authoring process, as described previously.

The completeness of the generated constraints was evaluated manually by the developer of CAS (the first author of this paper). The generated constraints were compared against the constraints found in KERMIT and NORMIT (referred to as benchmark constraints). As the languages used in the two constraint sets were different (the generated constraints were in pseudo-code and the benchmark constraints were in the proprietary constraint language) and as the granularity of the constraints was different, manual comparison was essential.

ER modelling

The ontology for ER modelling is shown in Figure 1. The solution structure consists of three components, which represent a set of entities, relationships and attributes respectively. As solutions can only contain instances of concrete (non-abstract) concepts, the “Entities” component can only hold instances of *Regular entity* and *Weak entity*. Similarly, the “Relationships” list can contain instances of the six concrete types of relationships (*Binary regular*, *N-ary regular* etc). Finally, the “Attributes” component contains instances of the concrete types of *Attribute* concept.

We added seven ER modelling problems and their solutions, selected from those available in KERMIT. For each problem we specified at least two alternative correct solutions. Although the solutions were chosen to illustrate different acceptable solutions, solutions in ER modelling tend to only have subtle differences. The solution composer enables the composition of secondary solutions by modifying the first solution, so the task of adding alternative solutions required little effort.

CAS produced a total of 49 syntax and 135 semantic constraints (Suraweera & Mitrovic, 2004). The syntax constraints ranged from simple constraints such as ‘*Entities* should have a name’ to more complex constraints such as ‘*Regular Entity* must have at least one *Key Attribute*’. The semantic constraints ensure that the student's ER diagram consists of all the required constructs and that they have been correctly connected to other constructs.

KERMIT's constraint base contains 35 syntax and 125 semantic constraints, all produced manually. Empirical studies have shown that the KERMIT's constraint base is effective in assisting students with their learning (Suraweera & Mitrovic, 2004). We compared the two constraint bases to evaluate the completeness of the constraints generated by CAS. The syntax constraints generated by CAS contained equivalents of all of those in KERMIT (i.e. 100% complete). Further analysis revealed that some generated syntax constraints were more specific than those in KERMIT, i.e. the two sets contained constraints of different granularity. In some cases several constraints generated by CAS were equivalent to a single constraint in KERMIT. This accounted for the difference in the numbers of constraints in the generated set (49) and KERMIT's syntax constraints (35).

The generated semantic constraints accounted for 90% of the 125 constraints found in KERMIT. Again the number of constraints differed between the two systems because the generated constraints were more specific. For example, KERMIT contains a constraint that verifies that the student's solution contains a matching key attribute attached to an entity. The domain model generated by CAS contained two constraints to account for the same problem state: one checks for the existence of the required key attribute in the solution, while the other checks whether the key attribute and the entity are connected.

The semantic constraints generator failed to produce 12 of KERMIT's semantic constraints. Some of these constraints ensure that constructs are represented using the correct type of construct. For example, they check that entities in the ideal solution are not represented as attributes or relationships in the student solution. Currently, CAS is not capable of generating such constraints. However, this problem state is covered by other constraints that ensure that there can be no extra constructs of each type. These constraints provide a more general feedback message that treats the misrepresented construct as an extra construct.

CAS also failed to generate a constraint from KERMIT that allows attributes of binary relationships to be assigned to an entity that participates in the relationship, as long as it has a cardinality of 1'. CAS is not able to generate this constraint as it deals with two equivalent relationship

instances (attribute belonging to a relationship and attribute belonging to an entity). Currently, CAS is only able to identify equivalent elements in a pair of solutions.

CAS also produced some constraints that do not have equivalents in KERMIT's constraint set. Further investigation revealed that Kermit's constraint set was imperfect. For example, although KERMIT allows a weak entity to be modelled as a composite multi-valued attribute, the student is required to have all the attributes of the weak entity with their types identical to their corresponding attributes in the ideal solution. However CAS correctly identified that when a weak entity is represented as a composite multi-valued attribute, the partial key of the weak entity has to be modelled as a simple attribute. Furthermore, the identifying relationship essential for the weak entity becomes obsolete.

Data Normalisation

Database normalisation is the process of refining a relational database schema to ensure all tables are of high quality (Elmasri & Navathe, 2003). The process proceeds in a top-down fashion by evaluating each table against criteria for each normal form and decomposing tables as necessary. There are four widely used normal forms: first, second, third (3NF) and Boyce-Codd normal form (BCNF). Typical problems in the domain of data normalisation focus on finding the normal form of a table given its functional dependencies. If necessary, the table is decomposed to satisfy BCNF normal form.

The procedure consists of 11 sequential steps. First the candidate keys of the table are determined. To verify that one or more attributes form the candidate key of a table, their closure has to be determined. Once the candidate keys are found, the prime attributes of the table must be identified. Next, the given functional dependencies (FD) are replaced by an equivalent set of FDs with simplified right-hand sides. To determine the normal form of the given table any partial FDs and other FDs that violate third normal form and BCNF are identified. Finally, in some cases the table must be further decomposed to satisfy BCNF after reducing the left hand sides of the FDs, and finding the minimal cover.

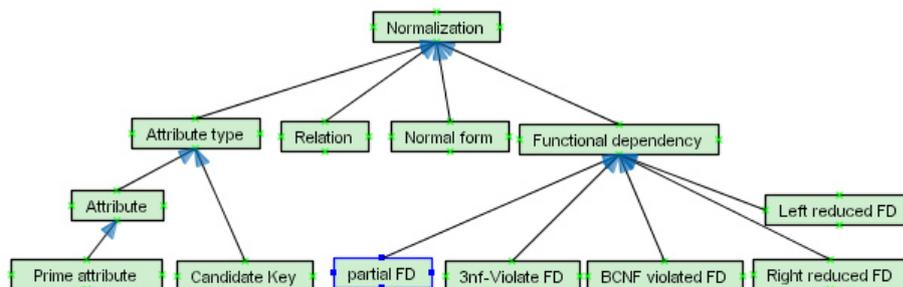


Fig. 15. Ontology for Data Normalisation

The ontology for the domain of Normalisation (Figure 15) contains four concepts: *Attribute type*, *Relation*, *Normal form*, *Functional dependency*. *Attribute type* is specialised to *Attribute* and *Candidate key*. *Attribute* is further specialised as *Prime attribute*. The *Functional dependency* concept is specialised according to the five types of functional dependencies; *Partial FD*, *3NF-violate FD*, *BCNF-violate FD*, *Right reduced FD*, *Left reduced FD*.

The student is required to provide a set of instances of attributes as the solution for the first step of determining the closure. The next step requires the student to outline candidate keys of the given table. Then instances of *Prime attributes* have to be specified. Steps four to seven require instances of

the different types of functional dependencies. Step eight requires an instance of the *Normal form* concept. Steps nine and ten both require instances of *Left reduced FD* as their solutions. Finally, an instance of *Relation* concept is required for decomposing the table to BCNF.

We selected three problems from NORMIT representing all domain concepts. These three problems were specifically chosen to ensure that the constraint generator encounters all significant concepts. The solutions were also obtained from NORMIT using its problem solver. The chosen problems had a single correct solution each. In situations where there is more than one correct path for solving a problem correctly, NORMIT's problem solver produces a correct solution based on the path taken by the student. As the solution to each step composed by a student is always compared to a single correct solution, the constraints generated by analysing a single solution would still be applicable for problems with multiple ways of solving them.

CAS produced 26 syntax constraints from the domain ontology, which check that the student followed the correct problem-solving path. For example a constraint was generated that ensured the student specified candidate keys in the second step. The generated set also contained constraints that ensured the functional dependencies produced by students adhered to the correct syntax. For example, a constraint was generated to ensure that every right-reduced functional dependency produced by the student contains only a single attribute in its right-hand side.

The generated constraints covered all but two of the 21 syntax constraints that exist in NORMIT (90%). Similarly to the experience with the domain of ER modelling the generated constraints were more specific than those in NORMIT. The two constraints that were not generated by CAS contained tests on more than one component of the solution. For example, the constraint that ensures the correctness of the specified partial FDs contains a test to ensure that the student has specified the table not to be in 2NF. It also contains a check that verifies the specified partial FDs have a candidate key in their left-hand and non-prime attributes in their right-hand. This constraint is a result of combining problem solving knowledge within constraints. A problem solver for normalisation, which is independent of constraints, would produce the correct solution for each step based on the problem-solving path taken by the student. Consequently, this constraint can be replaced by a simple constraint that compares each student specified partial FDs against the ideal solution's partial FDs. CAS generated such a constraint.

The semantic constraints generator produced a total of 45 constraints from the three problems and their solutions. Again the generated constraints were only relevant for a particular set of steps. For example, a constraint that ensures that each candidate key specified by the student has a corresponding candidate key in the ideal solution is only relevant when the student is working on the second step of identifying candidate keys of the table.

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

Summary

CAS produced high quality constraint sets for the domains of ER modelling and Data normalisation, as summarised in Table 2. Some constraints generated by CAS for the evaluated domains were too specific. The correct granularity of constraints is required to provide helpful feedback messages without enabling students to guess solutions: constraints that are too specific can encourage shallow learning, whereas constraints that are too general can frustrate students, due to their vague feedback

messages. Since experts may also disagree on the granularity of significant problem states, the correct granularity can only be determined by an empirical evaluation involving students (Martin & Mitrovic, 2005; 2006). Once an appropriate level of granularity is determined, the constraint algorithms may require adjusting to produce constraints with the desired level of granularity.

CAS failed to produce a few constraints for ER modelling (semantic) and data normalisation (syntax). In the case of ER modelling, the constraint generator failed to produce constraints that dealt with misrepresented constructs, plus the constraint that matches instances of ontological relationships. The syntax constraints generator failed to produce a few constraints for normalisation that had problem-solving logic embedded in them; such constraints would need to be added manually. However, since constraints operate independently, the generated constraints are sufficient for the domain model of an early version of a tutoring system. Testing the tutoring system might reveal the missing constraints.

Table 2: Summary of Constraint Generation Results

Domain	Expert set		Generated set		Completeness	
	Syntax	Semantic	Syntax	Semantic	Syntax	Semantic
ER Modelling	35	125	49	135	100%	85%
Data Normalisation	21	47	26	45	90%	100%

Effectiveness of CAS with Novice ITS Authors (Study 3)

The goal of CAS is to support novice authors to develop ITSs. In study 3 we evaluated the effectiveness of the system as a whole in meeting this aim. The study was performed with a group of 13 students enrolled in the 2006 graduate course on ITSs at the University of Canterbury. They were assigned the task of producing a complete ITS in WETAS for the domain of fraction addition, using CAS to author the domain model. We wanted to evaluate three hypotheses: 1) CAS makes it possible for novices to produce complete constraint bases; 2) the process of authoring constraints using CAS requires less effort than composing constraints manually; 3) the constraint generation algorithm depends on the quality of the ontology (i.e. an incomplete ontology will result in an incomplete constraint set).

The participants composed all the domain-dependent components necessary for CAS to generate constraints including a domain ontology, problems and solutions. Once this was done CAS generated syntax and semantic constraints in a high-level language (pseudo-code). At the time of the study, CAS was not able to convert these into the WETAS constraint language, so the participants were required to perform this step manually. They were also free to modify/delete constraints at this stage.

The participants had attended 13 lectures on ITSs, including five on constraint-based modelling before the study, similar to Study 1. They were introduced to CAS and WETAS, and given a task description that outlined the authoring process in CAS and described the WETAS constraint language. The participants were encouraged to follow the suggested authoring process. They were also given access to all the domain model components of LBITS (Martin & Mitrovic, 2003), as well as an ontology for database modelling as an example. We also provided the design for the student interface for the target ITS (one free-form text box per fraction). The participants had six weeks to complete the task; however, most students only started working on it at the end of the third week.

Twelve out of the 13 participants completed the task satisfactorily, i.e. produced working tutoring systems. One participant failed to complete the final step of converting the pseudo-code constraints to the target constraint language. The presented results only include the twelve participants who

completed the task. Analysis of CAS’s logs revealed that the participants spent a total of 31.3 (SD=13.4) hours on average interacting with the system. Six and a half hours (SD=4.34) of that was spent interacting with the Ontology view. There was a very high variance in the total interaction time, which can be attributed to each individual’s ability. The participants used the textual editors that were available under the “domain model editor” tab to modify/add domain model components required for WETAS, including problems, syntax constraints, semantic constraints and macros. The participants spent a mean total of 24.7 (SD=9.6) hours interacting with the textual editors.

Table 3: Total Numbers of Constraints Composed by Participants

	<i>Constraints</i>		<i>Completeness</i>		<i>Completeness %</i>	
	Syntax	Semantic	Syntax (8)	Semantic (13)	Syntax	Semantic
S1	5	12	5	7	63%	54%
S2	5	13	5	12	63%	92%
S3	4	12	4	12	50%	92%
S4	16	16	5	12	63%	92%
S5	14	18	8	13	100%	100%
S6	15	11	5	12	63%	92%
S7	2	5	3	3	38%	23%
S8	8	13	7	4	88%	31%
S9	5	8	4	4	50%	31%
S10	7	11	5	12	63%	92%
S11	4	18	5	12	63%	92%
S12	9	16	6	1	75%	8%
Mean	7.8	12.7	5.2	8.7	64.6%	66.7%
S.D.	4.7	3.9	1.3	4.5	16.7%	34.6%

The participants’ constraint bases were evaluated for completeness by comparing them to a constraint set composed by an expert (the *expert set*). This set contained eight syntax and 13 semantic constraints. The syntax constraints ensured that each component of a solution (such as the LCD and converted fractions) was of the correct format, and the student followed the correct problem-solving procedure. The semantic constraints ensure that the each part of the solution contains the correct values.

Table 3 lists the total numbers of constraints (syntax and semantic) composed by the participants under the “Constraints” column. The numbers of constraints in the expert set covered by each constraint base are listed under the “Completeness” column. The completeness of each constraint base (calculated as the percentage of constraints accounted for by each constraint base) is given under the “Completeness %” column. The participants accounted for five syntax constraints in the expert set (65%) and nine semantic constraints (67%). Only one participant (S5) produced all the necessary constraints. The majority of the others accounted for over half of the desired constraints. One participant (S12) struggled with semantic constraints and only managed to account for one desired constraint.

The expert set contains five syntax constraints for verifying the syntactic validity of inputs, such as whether the LCD is an integer and whether the entered fractions are syntactically valid. They need to be verified due to the generic nature of the interface the students were told to use, which consisted of a single text box for inputting a fraction. For example, constraints are required to verify that fractions are of the “numerator / denominator” form. However, these constraints are redundant for the

solution-composition interface produced by CAS from a complete ontology. It contains two text boxes for inputting a fraction (one for its numerator and the other for its denominator), ensuring that a fraction is of the correct format. Furthermore, these input boxes only accept values of the type specified for the relevant property in the ontology (numerator and denominator would both be defined as integer in this case). As a consequence, constraints such as those that verify whether the specified numerator is an integer are also redundant.

Table 4: Total Numbers of Constraints Generated by CAS

	Constraints		Completeness		Completeness %	
	Syntax	Semantic	Syntax (3)	Semantic (13)	Syntax	Semantic
S1	7	15	3	12	100%	92%
S2	8	12	3	12	100%	92%
S3	18	0	3	0	100%	
S4	6	23	3	1	100%	8%
S5	9	8	3	12	100%	92%
S6	0	23	0	1		8%
S7	13	26	3	1	100%	8%
S8	6	0	3	0	100%	
S9	11	23	3	1	100%	8%
S10	9	12	3	12	100%	92%
S11	7	12	3	12	100%	92%
S12	17	42	2	1	67%	8%
Mean	9.2	16.3	2.7	5.4	97%	49.9%
S.D.	4.9	11.9	0.9	5.8	9.9%	44.3%

The participants were free to make any modifications to the initial set of constraints produced by CAS, including deleting all of them and composing a new set manually if desired. For this reason we could not use the final constraint sets produced by the participants to evaluate our hypotheses, and therefore we used CAS to generate constraints from the domain information supplied by each participant. The generated constraint sets were analysed to calculate their completeness (Table 4). CAS generated the three syntax constraints necessary for CAS's solution interface for 10 participants. There was one situation where just two required constraints were generated (S12), as the result of an incorrectly specified solution structure. The syntax constraints generator failed to produce any constraints for participant S6 due to a software bug.

CAS could only generate 12 out of the 13 required semantic constraints, as it is unable to generate constraints that require algebraic functionality. CAS cannot generate a constraint that accounts for common multiples of the two denominators larger than the lowest common multiple. So the maximum degree of completeness that can be expected is 92%.

CAS generated the maximum possible 12 semantic constraints from domain-dependent components supplied by five participants. However, it was not successful in generating constraints for the remaining participants. Further analysis revealed that there were two main reasons. One of the reasons was that two of the participants (S4 and S9) had unknowingly added empty duplicate solutions for problems, which resulted in constraints that allowed empty solutions. This situation can be avoided easily by restricting the solution interface not to save empty solutions.

Another common cause for the failure to generate useful semantic constraints was an incomplete ontology. Four participants (S4, S6, S7 and S12) modelled the *Fraction* concept with only a single

property of type *String*. This results in a set of constraints that compare each component of the student solution against the respective ideal solution component as a whole. As these constraints are not of the correct level of granularity, the resulting feedback is limited in pedagogical significance. For example, the constraints would have the ability to denote that the student has made an error in the sum, but not able to pinpoint whether the mistake is in the numerator or the denominator. We believe that the decision to model the *Fraction* concept with a single property may have been influenced by the student interface that we required them to use. The participants may have attempted to produce an ontology and solution structure that is consistent with this particular student interface.

The constraint generator failed to produce any semantic constraints for two participants, S3 and S8. It failed to generate constraints for S3 due to a bug in the system. The other participant (S8) did not add any solutions, and therefore semantic constraints could not be generated.

a. Relevance:	Fraction-1 component of IS has a (?var1, ?*) 'Improper fraction'
Satisfaction:	Fraction-1 component of SS has a (?var1, ?*) 'Improper fraction'
b. Relevance:	(match IS Fraction-1 ("2." ?var1 ?IS-var2 ?*))
Satisfaction:	(match SS Fraction-1 ("2." ?var1 ?SS-var2 ?*))

Fig. 16. An example of translating a pseudo-code constraint into WETAS language

Although we assumed that generating pseudo-code version of constraints would assist the participants, there was little evidence in their reports that supported this assumption. Only one participant indicated that the generated constraints were useful. Since no explanation on the high-level constraint representation was provided to the participants, they may have struggled to understand the notation and to find commonalities between the two representations. For example, Figure 16a shows the pseudo-code representation of a constraint that ensures that the numerator of the first fraction specified by the student (SS) is the same as the corresponding value in the ideal solution (IS). The equivalent constraint in the WETAS language is given in Figure 16b.

The results from the evaluation study confirmed all our hypotheses: CAS was able to generate all the required syntax constraints for 10 of the 12 participants. Furthermore, CAS generated over 90% of the semantic constraints for half of the participants. Considering that the participants were given very little training in using the authoring system, the results are very encouraging. Providing the users with more training and improving CAS to be fully integrated with a tutoring server (similar to WETAS) would further increase its effectiveness.

The second hypothesis claims that CAS requires less effort than composing constraints manually. In order to obtain a measure for the effort required for producing constraints using CAS, we calculated the average time required for producing a single constraint. Only the participants whose domain model components resulted in generating nearly complete constraint bases were used for calculating the average effort, to ensure that incorrectly generated constraints were not accounted. Five participants (S1, S2, S5, S10, S11) spent a total of 24.8 hours composing the required domain-dependent information. They also spent a total of 115 hours interacting with the textual editors to produce a total of 107 constraints. Consequently, the participants spent a total of 1.3 hours on average to produce one constraint.

The average time of 1.3 hours per constraint is very close to the 1.1 hours per constraint reported by Mitrovic (1998) for composing constraints for SQL-Tutor. The time estimated by Mitrovic can be considered as biased since she is an expert of SQL and knowledge engineering, in addition to being an expert in composing constraints. Therefore, the achievement by novice ITS authors producing

constraints in a time similar to the time reported by Mitrovic is significant. Furthermore, the time of 1.3 hours is a significant improvement from the two hours required by a similar study reported in (Suraweera, Mitrovic & Martin, 2004a). Although that study involved composing constraints for the domain of adjectives in the English language, the overall complexities of the tasks are similar. Furthermore, after the experiment was completed, it was discovered that the setup of WETAS was not optimal, which required additional effort when writing the final constraints. First, WETAS can be set up to parse the input strings into appropriate sub-parts; without such a parser the constraints are much more difficult to write. Second, the choice of LBITS as a sample domain was not optimal because the solution structure is very different. The figure of 1.3 hours per constraint is therefore probably pessimistic.

Although at the time of the study CAS only generated constraints in a high-level language, it could be extended to generate constraints directly in the language required for execution. Assuming the generated constraints were produced in the required runnable form, a total of 99 syntax and semantic constraints were produced from 24.8 hours spent on composing the required domain information. Consequently, the participants would only require an average of 15 minutes (SD=0.25 hours) to generate each constraint. This is a dramatic improvement from 1.1 hours reported by Mitrovic (1998). This result is even more significant when we consider that the authoring process was driven by novice ITS authors. However, this does not take into account the effort required to validate the generated constraints; the constraint generation algorithm may not produce all the required constraints, so the domain author may also be required to modify the generated constraints or add new constraints manually.

Finally, the third hypothesis concentrates on the sensitivity of the constraint generation on the quality of the ontology. Developing a domain ontology is a design task that depends on the creator's perceptions of the domain. The ontologies developed by two users are very likely to be different, especially if the domain is complicated. In this study CAS generated constraints using ontologies developed by 12 participants. Although the ontologies were different, the syntax constraint generation algorithm managed to produce a full constraint set for almost all participants. However, the semantic constraint generation was more sensitive to the ontology. In particular, it was reliant on defining the *Fraction* concept with sufficient details. The semantic constraint generator managed to produce 92% complete constraint sets of the correct granularity for ontologies with a correctly defined *Fraction* concept. In contrast the constraints generated for ontologies with a partially defined *Fraction* concept were too general; they compared each fraction composed by students as a whole against its corresponding fraction in the ideal solution. These constraints result in feedback of limited pedagogical significance.

CONCLUSIONS AND FUTURE WORK

CAS is an authoring system that allows domain experts with little or no programming knowledge to produce domain models required for constraint-based ITSs. Unlike previous authoring systems, CAS is a general-purpose authoring system: the distinguishing feature of CAS is its ability to acquire knowledge for both procedural and non-procedural tasks. Authors develop domain models in CAS using a six-step process. Firstly, the author models an ontology of the domain using an ontology workspace provided by the system. Then, the author defines the solution structure. During the third step CAS uses the ontology and the solution structure to generate syntax constraints. The author

provides sample problems and their solutions in step four. In order for CAS to be able to generate semantic constraints that identify correct solutions arrived at using different approaches, the author needs to provide multiple solutions to problems, outlining different methods of solving them. The system then (in step five) generates semantic constraints by analysing the provided problems and their solutions. Finally, the domain expert validates the system-generated constraints by analysing the high-level descriptions of each constraint.

We conducted a series of evaluation studies to assess the effectiveness of CAS. The first study verified whether the creation of an ontology (and organising constraints according to its concepts) is beneficial in the process of manually composing constraints. The results revealed that the task of composing ontologies indeed assisted in the process of developing constraint bases. The effectiveness of CAS's constraint generation was then evaluated in a study that generated domain models for Database modelling and Data normalisation. Analysis of the generated constraints revealed that they accounted for over 90% of the constraints required for the domain. The final evaluation study evaluated CAS's effectiveness in generating constraints with the assistance of novice ITS authors, with promising results: CAS's syntax constraints generator produced all the required constraints for all but one of the participants. CAS also generated over 90% of the required semantic constraints for half of the participants.

This research opens a number of research avenues. First of all, further evaluation of CAS is needed in order to test the limits of its generality. We have tested CAS in a number of domains, with good results. However, further evaluation is needed in order to investigate the effects of the authoring procedure implemented in CAS on the quality of produced ITSs. For example, CAS supports the addition of alternative solutions by presenting the already specified solutions to the author, thus allowing the author to make modifications to it. This approach may introduce a bias, by encouraging the author to make minor modifications to existing solutions rather than thinking about radically different solutions.

Authoring a domain model using CAS currently requires the domain expert to model the entire domain ontology starting from a blank slate. The effort required to build an ontology can be reduced if the required ontology or a similar one can be imported from an ontology repository. This can be achieved by enhancing CAS to be able to import ontologies in standard representations such as OWL or DAML.

Constraint generation can also be improved in a number of ways. CAS currently produces constraints in a high-level pseudo-code form, thus requiring the author to manually translate the generated constraints into the form runnable in WETAS. The generation can be enhanced to produce constraints in the target constraint language directly. The generator can also be enhanced to produce constraints capable of ensuring that the students have entered the correct types of answers (ie. integers, strings etc).

Currently, the semantic constraint generation algorithm is only capable of analysing correct solutions provided by the domain expert. Typically, domain experts would know of common student misconceptions. They can use this knowledge to produce typical erroneous solutions. The constraint generation algorithm can be improved to make use of such erroneous solutions to specialise constraints to ensure that they are not overly general.

Currently CAS only has a collection of primitive types to describe concept properties. The list of types can be enhanced to include more sophisticated types such as algebraic expressions and algebraic equations. This would also require CAS to have a collection of built-in functions for each new type capable of identify equivalent solutions (e.g. $x*2 = 2x$). These functions can be used in semantic

constraints to compare the student answer against the ideal solution. This would enhance the ability to use CAS to generate domain models for mathematical domains.

CAS generates constraints with the assumption that there is a one-to-one mapping between problem solving interface widgets and properties of concepts in the ontology. This limits the nature of problem solving interfaces of the resulting ITS. This limitation can be overcome if CAS was enhanced with the ability to pre-process the solution entered by the student and convert it to the representation expected by CAS. This would enable an interface that allows free-form answers.

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 Artificial Intelligence programming. The contribution of this research is that it enables domain models to be generated automatically with the assistance of a domain expert such a teacher or a lecturer, obviating the need for knowledge engineering and programming skills. Consequently ITSs are poised to have a much significant and wider role in on education in the near future.

ACKNOWLEDGEMENTS

This research was supported by the University of Canterbury. We thank all members of ICTG for their support.

REFERENCES

- Altova – XML, Data Management, UML, and Web Services Tools, <http://www.altova.com>, 2005.
- Aleven, V., McLaren, B., Sewall, J., and Koedinger, K., The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains, in *Intelligent Tutoring Systems 2006*, Taiwan, 2006, pp. 61-70.
- Angros, R., Johnson, W. L., Rickel, J., and Scholer, A., Learning Domain Knowledge for Teaching Procedural Skills, in *First Int. joint conference on Autonomous agents and multiagent systems*, Bologna, Italy, 2002, pp. 1372-1378.
- Bechhofer, S., Horrocks, I., Goble, C., and Stevens, R., OilEd: a Reason-able Ontology Editor for the Semantic Web, in *14th Int. Workshop on Description Logics, DL2001*, Stanford, USA, 2001, pp. 396–408.
- Blessing, S. B., A Programming by Demonstration Authoring Tool for Model-Tracing Tutors, *Int. Journal of Artificial Intelligence in Education*, vol. 8, pp. 233-261, 1997.
- Chen, P. P., The Entity Relationship Model - Toward a Unified View of Data, *ACM Transactions Database Systems*, vol. 1, pp. 9-36, 1976.
- Clutterbuck, P. M., *The Art of Teaching Spelling: a Ready Reference and Classroom Active Resource for Australian Primary Schools*. Melbourne: Longman Australia Pty Ltd, 1990.
- Elmasri, R. and Navathe, S. B., *Fundamentals of Database Systems, Fourth Edition*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- Gruber, T. R., A Translation Approach to Portable Ontologies, *Knowledge Acquisition*, vol. 5, pp.199-220, 1993.
- Jarvis, M., Nuzzo-Jones, G., and Heffernan, N., Applying Machine Learning Techniques to Rule Generation in Intelligent Tutoring Systems, in *Intelligent Tutoring Systems 2004*, Maceio, Brazil, 2004, pp. 541-553.
- Koedinger, K., Aleven, V., Heffernan, N., McLaren, B., and Hockenberry, M., Opening the Door to Non-programmers: Authoring Intelligent Tutor Behavior by Demonstration, in *Intelligent Tutoring Systems 2004*, Maceio, Brazil, 2004, pp. 162-174.
- Koedinger, K. R., Anderson, J. R., Hadley, W. H., and Mark, M. A., Intelligent Tutoring Goes to School in the Big City, *Int. Journal of Artificial Intelligence in Education*, vol. 8, pp. 30-43, 1997.

- Martin, B., *Intelligent Tutoring Systems: The Practical Implementation of Constraint-based Modelling*, PhD thesis, University of Canterbury, 2002.
- Martin, B. and Mitrovic, A., Domain Modelling: Art or Science?, in *11th Int. Conference on Artificial Intelligence in Education*, Sydney, Australia, 2003, pp. 183-190.
- Martin, B. and Mitrovic, A., Using Learning Curves to Mine Student Models, in *User Modelling 2005*, Edinburgh, 2005, pp. 79-88.
- Martin, B. and Mitrovic, A., The Effect of Adapting Feedback Granularity in ITS, in *4th Int. Conference on Adaptive Hypermedia and Adaptive Web-Based Systems, AH2006*, Dublin Ireland, 2006, pp. 192-202.
- Matsuda, N., Cohen, W., Sewall, J., Lacerda, G., and Koedinger, K., Evaluating a Simulated Student using real students data for training and testing, in *User Modelling 2007*, C. Conati, K. McCoy, and G. Paliouras, Eds. Corfu, Greece: Springer, 2007, pp. 61-70.
- Mitrovic, A., Experiences in Implementing Constraint-Based Modelling in SQL-Tutor, in *Intelligent Tutoring Systems 1998*, San Antonio, 1998, pp. 414-423.
- Mitrovic, A., Scaffolding Answer Explanation in a Data Normalization Tutor, *Facta Universitatis, Series Elec. Energ.*, vol. 18, pp. 151-163, 2005.
- Mitrovic, A., Koedinger, K., and Martin, B., A Comparative Analysis of Cognitive Tutoring and Constraint-based Modeling, in *User Modelling 2003*, Pittsburgh, USA, 2003, pp. 313-322.
- Mitrovic, A. and Ohlsson, S., Evaluation of a Constraint-based Tutor for a Database Language, *Int. Journal of Artificial Intelligence in Education*, vol. 10, pp. 238-256, 1999.
- Murray, T., Expanding the Knowledge Acquisition Bottleneck for Intelligent Tutoring Systems, *Int. Journal of Artificial Intelligence in Education*, vol. 8, pp. 222-232, 1997.
- Murray, T., Authoring Intelligent Tutoring Systems: an Analysis of the State of the Art, *Int. Journal of Artificial Intelligence in Education, Part II of the Special Issue on Authoring Systems for Intelligent Tutoring Systems*, vol. 10, pp. 98-129, 1999.
- Murray, T., An Overview of Intelligent Tutoring System Authoring Tools: Updated analysis of the state of the art, *Authoring tools for advanced technology learning environments*, pp. 491-545, 2003.
- Ohlsson, S., Constraint-based Student Modelling, in *Student Modelling: the Key to Individualized Knowledge-based Instruction*, Berlin, 1994, pp. 167-189.
- OWL, OWL Web Ontology Language, <http://www.w3.org/TR/owl-features>, 2004.
- Protege, The Protege Ontology Editor and Knowledge Acquisition System, <http://protege.stanford.edu>, 2006.
- Suraweera, P. and Mitrovic, A., KERMIT: a Constraint-based Tutor for Database Modeling, in *Intelligent Tutoring Systems 2002*, Biarritz, France, 2002, pp. 377-387.
- Suraweera, P. and Mitrovic, A., An Intelligent Tutoring System for Entity Relationship Modelling, *Int. Journal of Artificial Intelligence in Education*, vol. 14, pp. 375-417, 2004.
- Suraweera, P., Mitrovic, A., and Martin, B., The Role of Domain Ontology in Knowledge Acquisition for ITSs, in *Intelligent Tutoring Systems 2004*, Maceio, Brazil, 2004, pp. 207-216.
- Suraweera, P., Mitrovic, A., and Martin, B., The Use of Ontologies in ITS Domain Knowledge Authoring, in *2nd Int. Workshop on Applications of Semantic Web for E-learning SWEL'04, ITS 2004*, Maceio, Brazil, 2004, pp. 41-49.
- Suraweera, P., Mitrovic, A., and Martin, B., A Knowledge Acquisition System for Constraint-based Intelligent Tutoring Systems, in *Artificial Intelligence in Education 2005*, Amsterdam, Netherlands, 2005, pp. 638-645.
- Suraweera, P., Mitrovic, A., and Martin, B., Constraint Authoring System: An Empirical Evaluation, in *Artificial Intelligence in Education 2007*, Marina Del Rey, Los Angeles, 2007, pp. 451-458.
- Tecuci, G., *Building Intelligent Agents: An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies*: Academic press, 1998.
- Tecuci, G. and Keeling, H., Developing an Intelligent Educational Agent with Disciple, *Int. Journal of Artificial Intelligence in Education*, vol. 10, pp. 221-237, 1999.
- Tecuci, G., Wright, K., Lee, S. W., Boicu, M., and Bowman, M. A Learning Agent Shell and Methodology for Developing Intelligent Agents, in *AAAI-98 Workshop: Software Tools for Developing Agents*, Madison, Wisconsin, 1998.