# Helping Teachers Build ITS with Domain Schema

Brent Martin and Antonija Mitrovic

Intelligent Computer Tutoring Group
University of Canterbury, Christchurch New Zealand
{brent.martin, tanja.mitrovic}@canterbury.ac.nz

**Abstract.** Authoring ITS domain models is a difficult task requiring many skills. Tools such as ASPIRE that model domains using ontology reduce the problem by allowing the author to work at a higher level of abstraction (and thus avoid low-level code writing), but such tools tend to be complex and the task is not intuitive for many people. To overcome this problem we have developed a framework for *domain schema*: high-level abstractions that describe the semantics of the domain model for a class of domains. Using domain schema reduces the authoring effort to one of describing only those aspects that are unique to this particular domain; the schema provides the rest of the model. We describe the framework we have implemented and give some examples of domain types for which schema have been built.

## 1. Introduction

Intelligent Tutoring Systems increasingly show promise as a technology that will expand the horizons of education from those able to attend a bricks-and-mortar institution to anyone with an Internet connection. Acting as an enhancement to traditional distance learning offerings, they promise to augment laboratories and tutorials by allowing students to practice the skills they are learning from home. In recent years tutors such as the Geometry and Algebra tutors, and the Addison-Wesley database place suite (SQL-Tutor, ER-Tutor and NORMIT) have made it out of the lab and into the classroom [1], [2].

Constraint-Based Modeling (CBM) [3] is an effective approach for building Intelligent Tutoring Systems (ITS) that supports the building of domain and student models. Constraint-based tutors are effective: for example, students using our database design tutor have shown significant gains in learning after as little as one hour of exposure to this system [4]. Also, CBM seeks to minimize the authoring effort by requiring the author model only states, rather than solution paths [5]. Nevertheless, the task of building an ITS is still large. To reduce the authoring effort we have developed a number of tools, including WETAS [6] and ASPIRE, an authoring system that allows the complete development of an ITS without ever writing a line of code [7].

ASPIRE makes it feasible for teachers with no prior knowledge to develop ITS, by automating most tasks or providing GUI interfaces allowing the author to easily define the elements of the final system, such as the general domain parameters (procedural versus non-procedural etc) and the structure of solutions that will be submitted by the student. The most complex part of the authoring task, that of modeling the domain, is achieved by creating an ontology of the domain concepts using a custom graphical tool. ASPIRE was developed to support constraint-based modeling. The domain model used by the final system (i.e. the set of constraints) is generated automatically from the ontology. ASPIRE's approach dramatically reduces the effort required to build an ITS, but nonetheless it is still a formidable task. In particular, developing domain ontology is a process that does not come naturally to all authors. For example, from a group of 12 students at the 2006 e-learning summer school at the University of Dublin, only half produced usable domain models for a simple hypothetical search engine language, of which only one was completely correct; the other half had considerable difficulty grasping the complexity of the modeling task, while nearly all participants were unable to model the recursive nature of the domain [8]. Also some authors have developed domains in ASPIRE entirely independently, but others have required help. This is a common problem in ITS authoring: the more general the tool, the harder it is to use. Many authoring systems overcome this problem by being limited to a particular type of domain. For example, Demonstr8 [9] is tailored for arithmetic.

Our goal for ASPIRE is that it be a tool for authoring ITS for *any* domain. To do this it must be easily extensible. Since different authors will have differing semantic requirements it must be possible for new domain types to be supported without changes to the core ASPIRE system. To facilitate this we have developed an additional abstraction layer, *domain schema*. Domain schema define the behavior of ASPIRE for a subset of domains that share a common structure and task type. New schema can be added to ASPIRE at any time by creating the appropriate XML documents and uploading them. The schema automates the authoring process still further by performing those tasks that are consistent across all domains of this type, such as providing the main structure of the domain ontology. Authors then work with the appropriate schema, rather than ASPIRE directly.

The next section briefly introduces constraint-based modeling (CBM), and describes two CBM authoring systems. Section three outlines how domain schema work, with an implemented example in the area of critiquing images. In section four we show how the approach can be generalized to other domain types. We conclude in Section five and discuss our long-term goals; to created distributed ITS via the semantic web, and to disconnect the domain model from the modeling and reasoning approaches.

## 2. Constraint-Based Modeling, WETAS and ASPIRE

CBM is based on the theory of learning from performance errors [10]. It models the domain as a set of state constraints, where each constraint represents a declarative concept that must be learned and internalized before the student can achieve mastery.

Constraints represent restrictions on solution *states,* and take the form:

*If* <*relevance condition*> *is true for the student's solution,*
*THEN* <*satisfaction condition*> *must also be true*

The relevance condition of each constraint checks whether the student's solution is in a pedagogically significant state. If so, the satisfaction condition is checked. If it succeeds, no action is taken; otherwise the student has made a mistake and appropriate feedback is given. *Syntactic* constraints check that the solution is syntactically correct. Conversely, *semantic* constraints check whether the student's solution has solved the problem, usually by comparing it to an "ideal" solution supplied by the teacher. The constraints implicitly encode semantics by testing for all of the different possible encodings of the semantic concept they are attempting to test. The student is thus permitted to use a different problem-solving strategy to the author, or even to mix strategies, provided no fundamental domain concepts are violated.

WETAS is a constraint-based web-enabled tutoring engine that provides all of the domain-independent functions for text-based ITS. It is implemented as a web server, written in Allegro Common Lisp, and using the AllegroServe Web server [11]. WETAS performs as much of the implementation as possible in a generic fashion. In particular, it provides the following functions: problem selection, answer evaluation, student modeling, feedback, and the user interface. The author need only provide the domain-dependent components, namely the structure of the domain (e.g. any curriculum subsets), the domain model (in the form of constraints), the problem/solution set, the scaffolding information (if any), and possibly an input parser, if any specific pre-processing of the input is required. WETAS has been used to build several tutors, including EER-Tutor [2] and Collect-UML [12]. It has also been used for four years by a graduate University class in Intelligent Tutoring Systems at the University of Canterbury.

ASPIRE is a high-level authoring tool that automates the encoding of constraints based on an ontology that the author provides via a graphical ontology editing tool. Unlike WETAS, at all stages in ASPIRE the author interacts with a GUI tool when authoring the domain; no additional files are required. ASPIRE is a general tool that has been used to build tutors across a variety of domains, such as basic accounting, thermodynamics and solid mechanics. However, the ASPIRE approach can still be improved in at least two ways. First, the author has little control over the interface; any non-standard user-interaction must be provided via bespoke applets which are uploaded into ASPIRE. Second, authoring in ASPIRE is still far from a trivial exercise. In particular, the task of ontology authoring is specialized and difficult. We hypothesized that we could make it easier to build ontologies by providing an ontology schema that reduces the ontology vocabulary to only concepts required for a particular subset of domains, thus making the author's job much more straightforward. We combined this with the ability to specialize ASPIRE's student interface for such domains. Finally, we further hypothesized that the semantic interpretation of the ontology for all domains of a given subset would also be the same. The combination of ontology schema, interface and ontology semantics is a *domain schema*.


**Fig. 1.** Example tutor for critiquing x-ray images.

## 3. Domain Schema

A domain schema is a collection of documents that describe parts of the domain model that will be common to all domains of the same general type, such as critiquing a set of images. The documents tell ASPIRE how to perform many parts of the authoring process that would be otherwise performed manually. The documents are:

- Ontology schema (XML) and ontology generation rules (XSLT)
- Constraint generation rules (XSLT)
- Solution structure generation rules (XSLT)
- Student interface (HTML, with optional Java applets)

In the following sections we will use an example domain type to illustrate how domain schemas work: for this domain type the student is shown a set of two or more images and is asked to choose the one with a particular characteristic and to identify features in the image that support their choice. This domain type could apply to many different subject areas (domains), such as: which of two buildings is Ionian; which x-ray image is better quality; which forest is the most damaged by acid rain; which painting is by Van Gogh; which x-ray shows an intestinal stricture. The interface consists of an applet for displaying, panning and zooming images, a control for selecting one of the images and a list of *features* that may or may not contribute to the decision; for each feature the student will select an appropriate *feature value*. Figure 1 shows this interface in action for an example of this domain type: x-ray power.

For each domain type the ontology will have the same basic form. The *ontology schema* defines this form by specifying concepts common to all domains of this type (typically the top of the ontology hierarchy), and describing the types of other concepts that the author can create and the relationship between these and the common concepts. Figure 2 shows part of the ontology for an ITS of the domain type "critique images", in this case the x-ray power domain, viewed using ASPIRE's

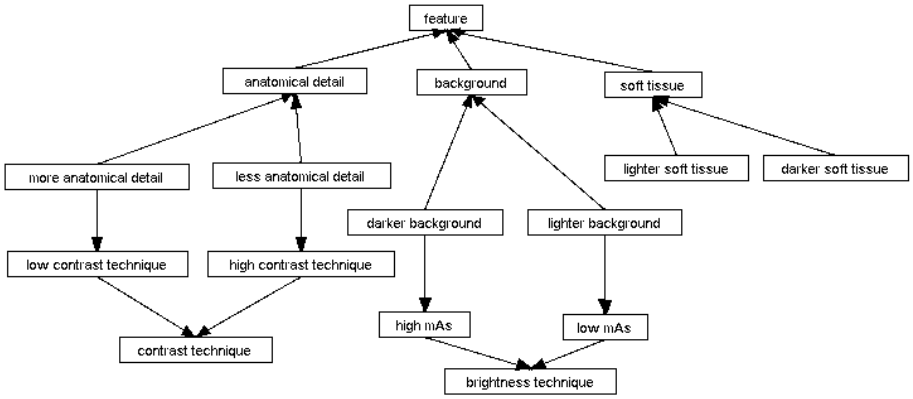


**Fig. 2.** Ontology for x-ray power

ontology editor. All ontologies for this domain type contain the "feature", "image" and "selection" concepts. The "feature" concept is then specialized for the actual features that the student will look for in this domain. The author can also specify *abstract* features if they wish; these are used for adding information that is common to more than one of the actual features. In figure 2 the actual features are "anatomical detail", "background" and "soft tissue"; abstract features are "contrast technique" and "brightness technique". Each feature is then further specialized into *feature values*, which are the values the student can choose between, such as "more anatomical detail" and "lighter soft tissue". The "image" concept is used to describe the images being shown to the student, in terms of the features present in this image (whether or not they contribute to the correct answer). Finally, the "selection" concept represents the choice the student must make between images.

The ontology schema is shown in Figure 3. The second part of this ontology schema describes the two concept types the author can create (feature and feature value). For each it also describes the attributes of that concept the author will be required to provide; in this case a *feature* can have two feedback messages (one—hint—to use when the student has overlooked this feature, and the other—wrong—for when the feature has been erroneously used). Similarly, a *feature value* has a summary and detailed feedback message, and another (positive) to be displayed as reinforcement when the student has correctly answered the question. Finally, the author can specify that one concept is an example of another; in figure 2 "more anatomical detail" is an example of "low contrast technique". Once the author has filled in the details for the features and feature values, the information is saved as an XML document and converted to a standard ASPIRE ontology using XSLT.

The ontology is then converted to constraints using an XML transform. This XSLT encodes the semantic interpretation of the ontology, by specifying how each concept should be turned into one or more constraints. For the domain type under discussion the constraint generation rules are as follows:

1. **Correct selection:** for each "selection" concept check the student has supplied the correct selection value

2. **All features specified:** For each feature, if a value is specified in the ideal solution, the student must also have specified a value
3. **No extraneous features:** for each feature, if the student has specified a value, the ideal solution must also specify a value
4. **Correct feature value:** If the student has specified a feature value, and one was required, is it the same as that in the ideal solution
5. **Feature value supports selection:** if the student has selected a feature value that is present in their chosen selection, check that the selection is correct.

For each constraint the hint and feedback messages (for the concept from which it is generated) are incorporated into boilerplate text to give the actual messages the user will see when the constraint is violated. For this domain type the semantics are very straightforward; other domain types are more complex (see Section 4).

The domain schema also defines how to generate from the ontology the solution structure (i.e. what the student must submit) and the default interface, again using

```
<!-- Default ontology, inserted directly -->
<baseOntology>
  <concept id='1' label='selection' name='selection' abstract='false'>
    <property  name='value' id='value' type='Any' unique='false'
     max-cardinality='1' min-cardinality='1'/>
  </concept>

  <concept label='feature' name='feature' abstract='true'></concept>

  <concept id='2' label='image' name='image' abstract='false'>
    <attribute name="input" value="author"/>
    <property  name='name' id='name' type='Any' unique='false' />
    <property  name='URL' id='URL' type='Any' unique='false' />
  </concept>
</baseOntology>

<!-- concept types that the author can create -->
<conceptType name="feature" label="Feature" input="true"
     propertyOf="image">
  <attribute name="name" label="Name" type ="text"/>
  <attribute name="abstract" label="Abstract?" type="boolean"/>
  <attribute name="negativeHint" label="Hint" type="text">
  <attribute name="negativeWrong" label="Wrong" type="text"/>
  <relationship name="exampleOf" label="Example Of"
     type="specialisation" range="feature">

  <conceptType name="featureValue" label="Feature value">
    <attribute name="name" label="Name" type ="text"/>
    <attribute name="summary" label="Summary feedback" type="text"/>
    <attribute name="detail" label="Detailed feedback" type="text"/>
    <attribute name="positive" label="Positive feedback" type="text"/>
    <relationship name="exampleOf" label = "Example of"
            type="specialisation" range="featureValue"/>
  </conceptType>
</conceptType>
```

**Fig. 3.** Ontology Schema for “critique images” (in XML)

XSLT. By default the solution structure consists of all non-abstract concepts. In the case of the domain type under discussion, each concept of type "feature" becomes a field in the student solution. The feature values are used to create the appropriate interface widget (e.g. a set of radio buttons) that the student will use to select values. The default interface displays controls for the entire solution structure. The author may then specialize the interface by specifying which parts of the solution structure are to be used for a given type of question in this domain, and they may override how it will be displayed. For example, in this domain type the author can specify that for certain questions only the "soft tissue" and "anatomical detail" features will be presented to the student, and that they will be represented using combo boxes. This allows the same domain model to be used for a variety of (related) tutoring tasks.

## 4. Other Domain Types

We are using domain schema to develop VIPER (Virtual Instructional and Practice Educational Resource) in conjunction with the Christchurch Polytechnic Institute of Technology (CPIT). For this project there are five domain types, all of which are visual: critique images; label an image; identify a feature in the image (i.e. point to it); perform measurements on an image; experiment with the parameters of an image. In all cases the domain model is feature-based, and as a result the semantics are straightforward. Another domain type we are developing is programming languages. In this type of tutor the student is given a task to perform where they must write a snippet of code in free text form. The ontology for this type of ITS describes the grammar of the language being used. For example, consider the domain of writing logical expressions. In this domain each concept represents some part of the language (e.g. "conjunct"); concept properties represent the "part-of" relationship between a concept and the language constructs that make up that concept; for example a conjunct consists of an expression, followed by "and" followed by a second expression). The constraint generation rules for checking semantics of this domain type are as follows:

1. **Concept necessary:** for each concept, if it appears at least once in the ideal solution, it must also appear in the student solution;
2. **Concept superfluous:** for each concept, if it appears at least once in the student solution, it must also appear in the ideal solution;
3. **All concept instances present:** for each instance of each concept in the ideal solution where the student solution contains at least one instance of this concept, there must exist an equivalent instance in the student solution;
4. **No concept instances superfluous:** for each instance of each concept in the student solution where the ideal solution contains at least one instance of this concept, there must exist an equivalent instance in the ideal solution;
5. **Correct components:** for each concept instance in the student solution, if all but one component is equivalent to an instance in the ideal solution, the remaining component must also be equivalent.

For the logical expressions domain the author describes each of the concepts in the same way as they would describe a grammar in BNF. However, this is not sufficient because they also need to define equivalence. For example, “dog and cat” is equivalent to “cat and dog”. They do this by defining additional concepts. In the previous example, conjunction is defined twice, with one definition being the exact reverse of the other. Each concept can then specify an “is equivalent to” relationship with another. In some cases the concept will be one that does not already appear in the grammar. For example, for logical expressions we can define de Morgan’s law:

$$\overline{(A \wedge B)} \equiv \overline{A} \vee \overline{B} \tag{1}$$

We specify this law by defining both de Morgan forms and indicating they are equivalent. The constraint generation rules then use this information as follows. First, whenever a concept detected in one solution (e.g. the ideal solution) is being looked for in the other solution (i.e. the student solution), the default logic is to look for the exact same concept instance in both solutions. However, if the concept instance is an example of a concept for which an equivalent form exists, the constraint will instead check that either the same concept instance exists in the other solution *or* an equivalent concept instance exists. Second, when checking for a particular concept instance, the constraint will also check whether it *forms part of* another concept that takes part in an equivalence relationship, and the alternate form exists in the other solution. If so, the check is dropped. For example, when checking for all “and”s, if the “and” in question is part of a De Morgan form and the student used the alternate form, the check for “and” will be dropped. We are currently evaluating this domain type in the areas of logical expressions, Java and SQL. This approach is also potentially useful for natural languages, provided the domain is sufficiently constrained. We are also exploring this possibility.

Another example of a completely different domain type is arithmetic procedural domains, such as multi-column addition. These can be catered for by extending the framework described as follows. First, for such domains the properties of a concept must be able to be collections. For example, an addition problem is made up of a collection of columns; each column contains a carry, a collection of addends and a sum. Second, the author must be able to specify arithmetic value restrictions for properties. For example (again from multi-column addition):

$$sum(n) = \left[carry(n) + SUM\left(addends(n)\right)\right] MOD\ 10 \tag{2}$$

$$carry(n) = \left[carry(n+1) + SUM\left(addends(n+1)\right)\right] DIV\ 10 \tag{3}$$

Note that *n* is the column number (more generally, *n* is the instance number). SUM and DIV are built-in primitives. As well as giving the formula for the restriction, the author also specifies two associated feedback messages: one that describes what the restriction means in words (used to correct the student when they violate the restriction) and one that describes the dependencies implied by the RHS of the restriction (used to indicate why the student should not be specifying this value yet,

because the restriction cannot be tested). The constraints are now generated from both the concepts in the ontology plus the restrictions, as follows:

1. **All values specified:** For each concept instance, check whether this instance has been completed, e.g. "*You have not filled in the sum for column 3.*" Note that the restrictions imply dependencies between concept instances, which also need to be checked. If the dependent concept instances are not complete yet this constraint will not be relevant.
2. **Ordering:** For each concept that is on the LHS of a restriction, if the student has supplied an instance of this concept, check that the necessary parts in the RHS have been specified and give the "dependency" error if not, e.g. "*You cannot compute the carry for a column until you have completed the column to the right.*"
3. **Correct value:** For each concept that is on the LHS of a restriction, test its value, and give an error if wrong, e.g. "*Check your sum in column 3. The sum should add up to the sum of addends in this column, plus the carry, if any*", or "*Check the value of the carry in column 2. The carry should be 1 if the addends and carry in the next column to the right add up to 10 or more.*"

This logic is sufficiently general to apply to other arithmetic domains, such as fraction addition.

## 6. Conclusions and Future Work

ITS authoring is a difficult task. Whilst generic authoring tools such as ASPIRE dramatically reduce the authoring effort required, domain authoring nevertheless remains a specialized task. We have introduced a framework, called domain schema, that allows a generic ITS authoring tool to be tailored to specific domain types to ease the authoring process, but which is still general in the sense that it can be readily extended to support new domain types. We are using this framework in the VIPER project to create an authoring environment suited to domains where the student tasks involve interacting with images. We have also shown how the approach is suited to other very different domain types such as programming languages and arithmetic. VIPER will be trialed by teachers at CPIT in mid 2008.

Most (if not all) existing ITS tools are monolithic: the student logs into the ITS system, which then serves them content. This is in direct contrast with other web-based educational content delivery approaches, which are *content-oriented*. In content-oriented systems, learners seek out appropriate educational content from any source that their client software supports (e.g. SCORM). In the ITS equivalent teachers would develop exercises for their students, who complete them and submit their answer to an appropriate reasoning engine for evaluation. The reasoning engine would be a lightweight expert system that can run rules written in some standard language (e.g. ruleML [13]). Each page of content contains links to domain content in a form similar to what we have just described: the domain is represented by an ontology plus additional information indicating how the ontology should be used to

generate evaluation rules. This would allow re-use of the same ontology for different types of evaluation. For example, the domain information for multi-column addition described in section four is sufficient to generate production rules for a model-tracing tutor [14]. Further, ontology could be cobbled together from existing ones, or customized by overriding some parts. The framework we have described is a step towards this because it separates the reasoning rules, ontology and reasoning engine.

Intelligent tutoring systems are a promising tool for delivering education remotely. To date a key problem has been the effort required to build such systems, even when sophisticated authoring tools are used. Domain schema is a promising step towards making ITS a realistic option for education practitioners everywhere.

## References


1. Koedinger, K.R., Anderson, J.R., Hadley, W.H., and Mark, M.A.: Intelligent Tutoring Goes To School in the Big City. International Journal of Artificial Intelligence in Education. 8, 30-43 (1997)
2. Mitrovic, A.: Large-Scale Deployment of three intelligent web-based database tutors. In: Proceedings of ITI. Cavtat, Croatia, pp. 135-140. (2006)
3. Ohlsson, S.: Constraint-Based Student Modeling. In: J. Greer and G. McCalla (eds.) Student Modeling: The Key to Individualized Knowledge-Based Instruction, pp. 167-189. Springer-Verlag, New York (1994)
4. Suraweera, P., Mitrovic, A., An Intelligent Tutoring System for Entity Relationship Modelling. Int. J. Artificial Intelligence in Education, 14, 3-4, 375-417 (2004)
5. Mitrovic, A., Koedinger, K.R., and Martin, B.: A Comparative Analysis of Cognitive Tutoring and Constraint-Based Modelling. In: Ninth International Conference on User Modeling UM 2003, pp. 313-322. Springer-Verlag. (2003)
6. Martin, B. and Mitrovic, A.: WETAS: A Web-Based Authoring System for Constraint-Based ITS. In: Second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems. Malaga, pp. 543-546. Springer. (2002)
7. Mitrovic, A., Suraweera, P., Martin, B., Zakharov, K., Milik, N., and Holland, J.: Authoring constraint-based tutors in ASPIRE. In: ITS 2006. Taiwan, pp. 41-50. (2006)
8. Martin, B., Mitrovic, A., and Suraweera, P.: Domain Modelling with ontology: a case study. In: International workshop on authoring adaptive education systems at UM07. Corfu, Greece, pp. 4-11. (2007)
9. Blessing: A Programming by Demonstration Authoring Tool for Model-Tracing Tutors. International Journal of Artificial Intelligence in Education. 8, 233-261 (1997)
10. Ohlsson, S.: Learning from Performance Errors. Psychological Review. 3(2), 241-262 (1996)
11. Allegroserve, http://opensource.franz.com/aserve/
12. Baghhaei, N. and Mitrovic, A.: A Constraint-based Collaborative Environment for Learning UML Class Diagrams. In: ITS2006. Taiwan, pp. 176-186. Springer. (2006)
13. Boley, H., Tabet, S., and Wagner, G.: Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In: The first Semantic Web Working Symposium, SWWS'01. Stanford, California USA, pp. 381-401. (2001)
14. Anderson, J.R., Corbett, A.T., Koedinger, K.R., and Pelletier, R.: Cognitive Tutors: Lessons Learned. Journal of the Learning Sciences. 4(2), 167-207 (1995)