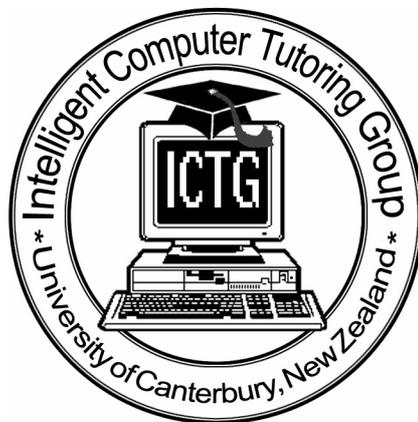




ASPIRE: Functional Specification and Architectural Design

Antonija Mitrovic
Brent Martin
Pramuditha Suraweera
Konstantin Zakharov
Nancy Milik
Jay Holland

Technical Report TR-06/05, 8 September 2005
Intelligent Computer Tutoring Group
Department of Computer Science and Software Engineering
University of Canterbury



Content

1. Introduction
 - 1.1. Background
 - 1.2. Constraint-Based Tutors
 - 1.3. The WETAS Tutoring Shell
2. The ASPIRE Architectural Framework
 - 2.1. ASPIRE-Tutor
 - 2.2. ASPIRE-Author
3. Data Model
 - 3.1. User Data
 - 3.2. Domain Model
 - 3.3. Student Modelling
 - 3.4. Sessions
4. Functionality: User Stories
 - 4.1. Student User Stories
 - 4.2. Author User Stories
 - 4.3. Administrator User Stories
 - 4.4. Teacher User Stories
5. Knowledge Representation Language
 - 5.1. Problem specification
 - 5.2. Constraint language
 - 5.3. Further extensions
6. Session Manager
7. Conclusions
8. References

1. Introduction

This document reports the work done during the initial two months of the ASPIRE project, funded by the e-Learning Collaborative Development Fund grant 502. In this project, we will develop a Web-enabled authoring system called ASPIRE, for building intelligent learning agents¹ for use in e-learning courses. ASPIRE will support the process of developing intelligent educational agents by automating the tasks involved, thus making it possible for tertiary teachers with little computer background to develop systems for their courses. In addition, we will develop several intelligent agents using the authoring system, so that we can evaluate its effectiveness and efficiency. The resulting e-learning courses will overcome the deficiencies of existing distance learning courses and support deep learning. The proposed project will dramatically extend the capability of the tertiary education system in the area of e-learning.

In this first report on the ASPIRE project, we start by presenting the background for the project, and then describe our previous work. Section 1.2 presents the basic features of constraint-based tutors, while Section 1.3 presents WETAS, a prototype of a tutoring shell. ASPIRE will be based on these foundations.

The first project milestone is the architecture of ASPIRE, which is discussed in Section 2. We first present the overall architecture of ASPIRE, and then turn to details of ASPIRE-Tutor, the tutoring server, followed by a similar discussion of ASPIRE-Author, the authoring server. Section 3 presents the data model and discusses individual classes. We then present the functionality of the system in terms of user stories in Section 4.

Section 5 presents the results of the second project milestone – designing the knowledge representation language used to generate domain models. The third milestone, the Session Manager, is presented in the last section.

1.1. Background

E-learning is becoming more and more popular with the wide-spread use of computers and the internet in educational institutions. Current e-learning courses are almost invariably developed using course management systems (CMS), such as WebCT or Blackboard. Although CMS tools provide support for some administrative tasks and enable instructors to provide instructional material, they offer no real support for learning. In such e-learning courses, the student has access to on-line material, simple multi-choice quizzes and chat tools. However, there is no ability to track the student's progress and adapt the learning material and instructional session to the individual student. In such courses, a student cannot obtain individualised domain-level feedback on his/her performance, which is critical for successful learning. This "one-size-fits-all" approach is the main deficiency in existing e-learning courses.

The solution to this situation is in adaptive intelligent educational systems, which track students' progress and generate models of students' knowledge, which are later used to adapt instructional sessions towards knowledge, needs and abilities of each individual student. Adaptation can be done in various ways: the student might be given feedback tailored towards the student's level of understanding, problems are selected so that they extend the student and provide new opportunities to e-learning and so on. Such educational systems have been shown to result in significant improvement over simplistic e-learning courses in domains that require extensive practice (Corbett et al., 1998; Koedinger et al., 1997; Mitrovic & Ohlsson, 1999).

Although intelligent educational systems are very effective, they are still not commonly available due to the high development costs. These systems require not only detailed knowledge of the instructional domain, but also extensive expertise in Artificial Intelligence, Computer Science, Education and

¹ In the report, we refer to intelligent learning agents also as intelligent tutoring systems or (adaptive) intelligent learning systems/environments. All these terms should be treated as synonyms.

Psychology. There has been some research done on developing authoring systems, which make the development of intelligent educational systems easier. However, there are only a few authoring systems developed worldwide, and all of them are only used within research laboratories. These authoring systems are highly specialised, and usually support a very limited set of instructional domains (such as algebra). Also all existing authoring systems require that the authors have substantial knowledge of the tool itself. The proposed project aims to resolve all these limitations.

The Intelligent Computer Tutoring Group (ICTG²) at the University of Canterbury have developed a methodology for building intelligent educational systems, which has been recognised internationally as one of the most promising approaches. Using this approach, we have developed several intelligent educational agents that have been thoroughly evaluated with local and international students (Mitrovic et al., 2004). We have proved that our intelligent agents support learning effectively, resulting in significant increase in students' knowledge. For example, the evaluation of SQL-Tutor, an intelligent tutoring system that teaches the SQL database language, showed that the students who used this system for only two hours achieved significantly higher marks in an exam than students who only had classroom instruction (the difference was 0.63 standard deviations) (Mitrovic & Ohlsson, 1999). In other studies students using our intelligent educational systems also achieved one grade higher than their peers in the traditional classroom situation. We discuss the general features of our intelligent learning agents in the following subsection.

1.2. Constraint-Based Tutors

Intelligent Tutoring Systems (ITS) are developed with the goal of automating one-to-one human tutoring, which is the most effective mode of teaching (Bloom, 1984). ITS offer greater flexibility in contrast to non-intelligent software tutors since they can adapt to each individual student. Although ITSs have been proven to be effective in a number of domains, the number of ITSs used in real courses is still extremely small (Mitrovic, Martin & Mayo, 2002). The goal of ICTG is to develop a powerful methodology for developing constraint-based tutors. Our methodology is based on Ohlsson's (1996) theory of learning from performance errors.

The typical architecture of constraint-based tutors is given in Figure 1. The tutors are developed in AllegroServe Web server, an extensible server provided with Allegro Common Lisp. All student models are kept on the server. At the beginning of interaction, a student is required to enter his/her name, which is necessary in order to establish a session. The session manager requires the student modeller to retrieve the model for the student, if there is one, or to create a new model for a new student. All student actions are sent to the session manager, to be linked to the appropriate session and stored in the student's log. The action is then sent to the pedagogical module (PM). If the submitted action is a solution to the current step, the PM sends it to the student modeller, which diagnoses the solution, updates the student model and sends the result of the diagnosis back to the PM, which generates feedback.

Domain knowledge consists of a set of constraints. Constraint-Based Modelling (CBM) (Ohlsson, 1994; Mitrovic & Ohlsson, 1999) is a student modelling approach that is not interested in the exact sequence of states in the problem space the student has traversed, but in what state he/she is in currently. As long as the student never reaches a state that is known to be wrong, they are free to perform whatever actions they please. The domain model is a collection of state descriptions of the form:

If <relevance condition> is true, then <satisfaction condition> had better also be true, otherwise something has gone wrong.

The knowledge base consists of constraints used for testing the student's solution for syntax errors and comparing it against the system's ideal solution to find semantic errors. The knowledge base enables the tutor to identify student solutions that are identical to the system's ideal solution. More importantly,

² <http://www.cosc.canterbury.ac.nz/~tanja/ictg.html>

this knowledge also enables the system to identify valid alternative solutions, i.e. solutions that are correct but not identical to the system's solution. Each constraint specifies a fundamental property of a domain that must be satisfied by all solutions. Constraints are problem-independent and modular, and therefore easy to evaluate. They are written in Lisp, and can contain built-in functions as well as domain-specific ones. For examples of constraints, please see (Mitrovic, 1998a, 2002; 2003; Suraweera & Mitrovic, 2001; 2002; Martin & Mitrovic, 2003; Mitrovic, Koedinger & Martin, 2003). If the satisfaction condition of a relevant constraint is met by the student solution, the solution is correct. In the opposite case, the student will be given feedback on errors.

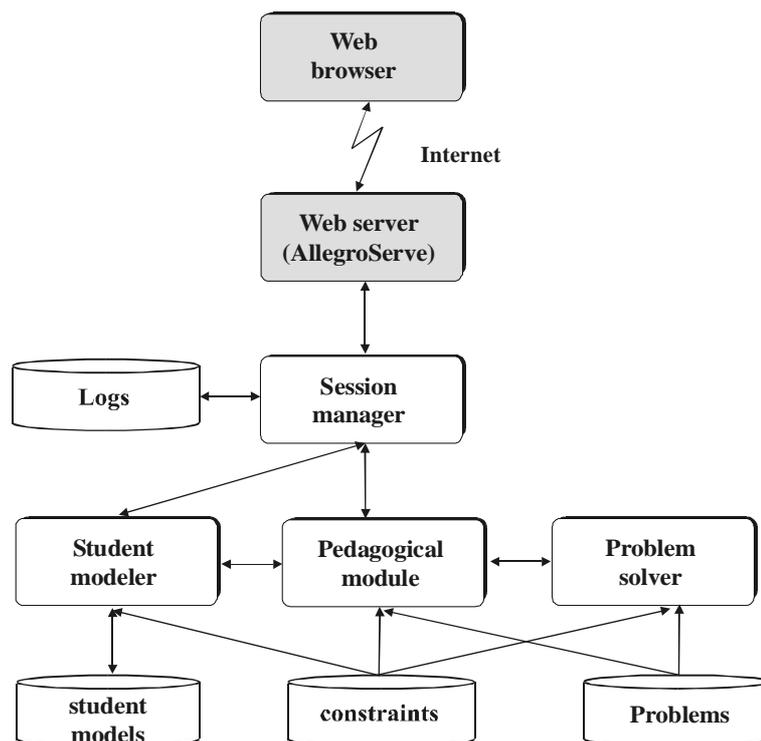


Figure 1. The architecture of constraint-based tutors

One of the advantages of CBM over other student modelling approaches (Mitrovic, Koedinger & Martin, 2003) is its independence from the problem-solving strategy employed by the student. CBM models students' evaluative, rather than generative knowledge and therefore does not attempt to induce the student's problem-solving strategy. CBM does not require an executable domain model, and is applicable in situations in which such a model would be difficult to construct (such as database design or SQL query generation). Furthermore, CBM eliminates the need for bug libraries, i.e. collections of typical errors made by students. On the contrary, CBM focuses on correct knowledge only. If a student performs an incorrect action, that action will violate some constraints. Therefore, a constraint-based tutor can react to misconceptions although it does not represent them explicitly. A violated constraint means that student's knowledge is incomplete or incorrect, and the system can respond by generating an appropriate feedback message. Feedback messages are attached to the constraints, and they explain the general principle violated by the student's actions. Feedback can be made very detailed, by instantiating parts of it according to the student's action.

The student modeller evaluates the student's solution against the knowledge base and updates the student model. The short-term student model consists of a list of violated and a list of satisfied constraints

for the current attempt. The long-term model records the history of usage for each constraint. This information is used to select problems of appropriate complexity for the student, and to generate feedback.

The pedagogical module (PM) is the driving engine of the whole system. Its main tasks are to generate appropriate feedback messages for the student and to select new practice problems. PM individualises these actions to each student based on their student model. Unlike ITSs that use model tracing (Anderson et al., 1996; Corbett et al., 1998; Koedinger et al., 1997), constraint-based tutors do not follow each student's solution step-by-step: a student's solution is only evaluated once it is submitted, although the student may submit a partial solution to get ideas on how to progress.

The feedback is grouped into six levels according to the amount of detail: *correct*, *error flag*, *hint*, *detailed hint*, *all errors* and *solution*. The first level of feedback, *correct*, simply indicates whether the submitted solution is correct or incorrect. The *error flag* indicates the type of construct (e.g. entity, relationship, etc.) that contains the error. *Hint* and *detailed hint* offer a feedback message generated from the first violated constraint. *Hint* is a general message such as "There are attributes that do not belong to any entity or relationship". On the other hand, *detailed hint* provides a more specific message such as "The 'Address' attribute does not belong to any entity or relationship", where the details of the erroneous object are given. Not all detailed hint messages give the details of the construct in question, since giving details on missing constructs would give away solutions. A list of feedback messages on all violated constraints is displayed at the *all errors* level. Finally, the complete solution is displayed at the *solution* level.

Initially, when the student begins to work on a problem, the feedback level is set to the *correct* level. As a result, the first time a solution is submitted, a simple message indicating whether or not the solution is correct is given. This initial level of feedback is deliberately low, as to encourage students to solve the problem by themselves. The level of feedback is incremented with each submission until the feedback level reaches the *detailed hint* level. Automatically incrementing the levels of feedback is terminated at the *detailed hint* level to encourage to the student to concentrate on one error at a time rather than all the errors in the solution. Moreover, if the system automatically displays the solution to the student on the sixth attempt, it would discourage them from attempting to solve the problem at all, and may even lead to frustration. The system also gives the student the freedom to manually select any level of feedback according to their needs.

When selecting a new problem, the PM firsts decides what concept is appropriate for the student on the basis of the student model. The concept that contains the greatest number of violated constraints is targeted. We have chosen this simple problem selection strategy in order to ensure that students get the most practice on the concepts with which they experience difficulties. In situations where there is no obvious "best" concept (i.e. a prominent group of constraints to be targeted), the next problem in the list of available problems, ordered according to increasing complexity, is given. We have also experimented with alternative problem-selection strategies, using Bayesian nets (Mayo & Mitrovic 2000; 2001) and neural networks (Wang & Mitrovic, 2002).

1.3. The WETAS Tutoring Shell

While CBM reduces the effort of building domain models for ITS, the task of building a constraint-based tutor is nevertheless still large. To reduce the authoring effort we have developed WETAS (Web-Enabled Tutor Authoring Shell), a web-based tutoring engine that provides all of the domain-independent functions for text-based ITSs (Martin & Mitrovic, 2003). WETAS removes much of the effort required to build an ITS, but it does not directly facilitate the building of the domain model, which is arguably one of the most difficult tasks.

WETAS was implemented in Allegro Common Lisp, and using the AllegroServe Web server. WETAS supports students learning multiple subjects at the same time; there is no limit to the number of domains it may accommodate. Students interact through a standard web browser such as Netscape or Internet Explorer. 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 modelling, feedback, and the user interface. The author provides the domain-dependent components, namely the structure of the domain, 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 provides both the infrastructure (e.g. student interface) and the “intelligent” parts of the ITS, namely the pedagogical module and the domain model interpreter. The former makes decisions based on the student model regarding what problem to present to the student next and what feedback they should be given. The latter evaluates the student’s answers by comparing them to the domain model, and uses this information to update the student model.

A prototype of WETAS has been implemented and used to evaluate its effectiveness in reducing the ITS building effort. To demonstrate the flexibility of WETAS we have re-implemented two existing constraint-based tutors, SQL-Tutor (Mitrovic, 1998a, 1998b, 1998c) and EER-Tutor (Zakharov, Mitrovic & Ohlsson, 2005), and developed a new Language Builder ITS (LBITS) which teaches vocabulary skills in English. These domains have very different problem/solution structures. We have evaluated LBITS in a New Zealand elementary school (Martin, 2003; Martin & Mitrovic, 2002), and the other two tutors at the University of Canterbury. The evaluations show that students’ learning increases significantly with the support of these ITSs.

Although WETAS has proved to be an effective tool for developing ITSs, the task of developing the domain model still requires significant knowledge and experience. The goal of the ASPIRE project is to build upon our experiences in developing constraint-based tutors and the WETAS authoring shell. ASPIRE will not only provide a tutor-deployment platform, as WETAS does; it will also provide support for the authoring phase. Most notably, it would support the author in developing the domain model.

2. The ASPIRE Architectural Framework

Figure 2 depicts the overall architecture of ASPIRE. Tutor delivery and authoring assistance will be delivered via the internet and a standard browser, and will be kept separate; we will develop two standalone Web servers, ASPIRE-Author and ASPIRE-Tutor. Communication between the servers will be afforded by the sharing of instructional content. The important architectural considerations are:

- The system will be developed using Allegro Common Lisp and the AllegroServe lightweight web server.
- To minimise compatibility problems, all functionality will be provided server-side as far as possible: Client-side scripting (e.g. Javascript) will be used only where there is no simple alternative, e.g. communicating with JAVA applets where they are deployed.
- Code maintenance will be made easier by the use of “Web Actions” – generic Web objects that consist of combinations of static HTML (where applicable) and Lisp functions. Web Actions may also include server-side scripting. Each Web service will be developed according to what is most efficient for that particular service.
- Each Web server will be divided into *modules*, where each module performs a major high-level function, e.g. the diagnostic module. This will be achieved by breaking the system into *packages*.
- The CLOS object-oriented system will be used to aid structural transparency and ensure maintainability. Packages will thus be further divided into many source code files according to what objects make up that package.

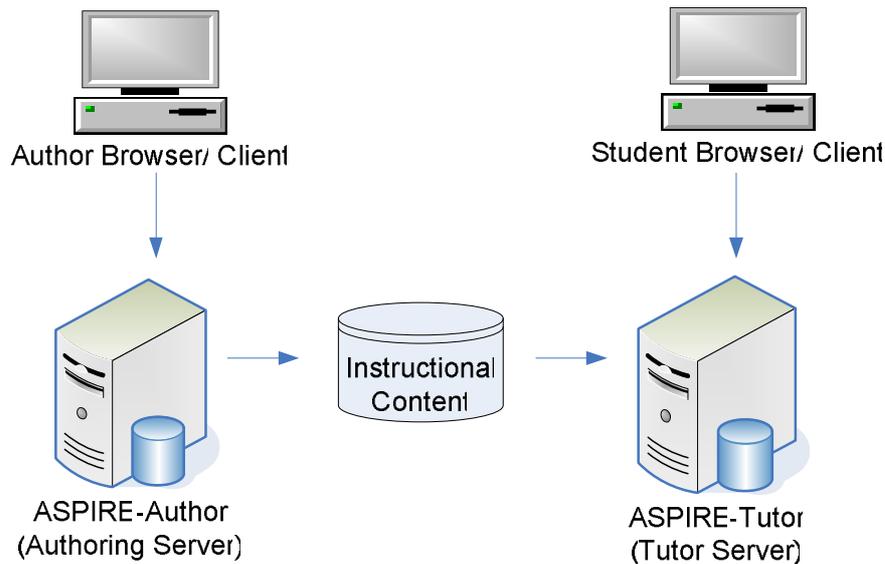


Figure 2. The overall view of ASPIRE

2.1. ASPIRE-Tutor

The ASPIRE-Tutor server is broken up into a set of modules, where each module has specific responsibilities in the serving of intelligent tutoring systems. ASPIRE-Tutor is based on the Model-View-Controller (MVC) architecture (Krasner & Pope, 1998). Model-View-Controller is a software architecture that separates the user interface, control logic and application's data model into three distinct components so that modifications to the system components, and in particular the interface, can be made with minimal impact to the data model. Such separation allows multiple views to share the same data model, which makes multiple clients easier to implement, test and maintain. When developing applications with multiple interfaces, the interface code should not be intertwined with non-interface code, because it will lead to duplicate efforts in testing, maintenance and modification. The data and the operations on data should be encapsulated in the *Model* component.

It is straightforward to map MVC concepts into the domain of Web based applications:

- The *Model* represents data and rules that govern access to and updates of this data;
- The *View* renders the contents of a Model. It accesses the data through the Model and specifies how the data should be presented. It is the View's responsibility to maintain consistency in its presentation when the Model changes.
- The *Controller* translates interactions with the View into actions to be performed by the Model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests.

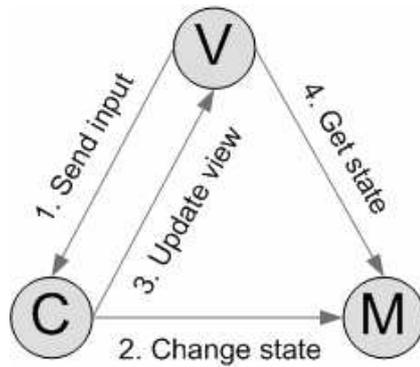


Figure 3. The MVC architecture

The typical sequence of interactions between the MVC components starts with the View sending input to the Controller. The Controller translates the input into an operation on the Model. When processing is complete, the Controller issues a command to update the View. Then the View updates itself by getting the current state of the Model. This interaction is illustrated in Figure 3.

The general architecture of ASPIRE-Tutor is illustrated in Figure 4. The individual modules are discussed in the following subsections.

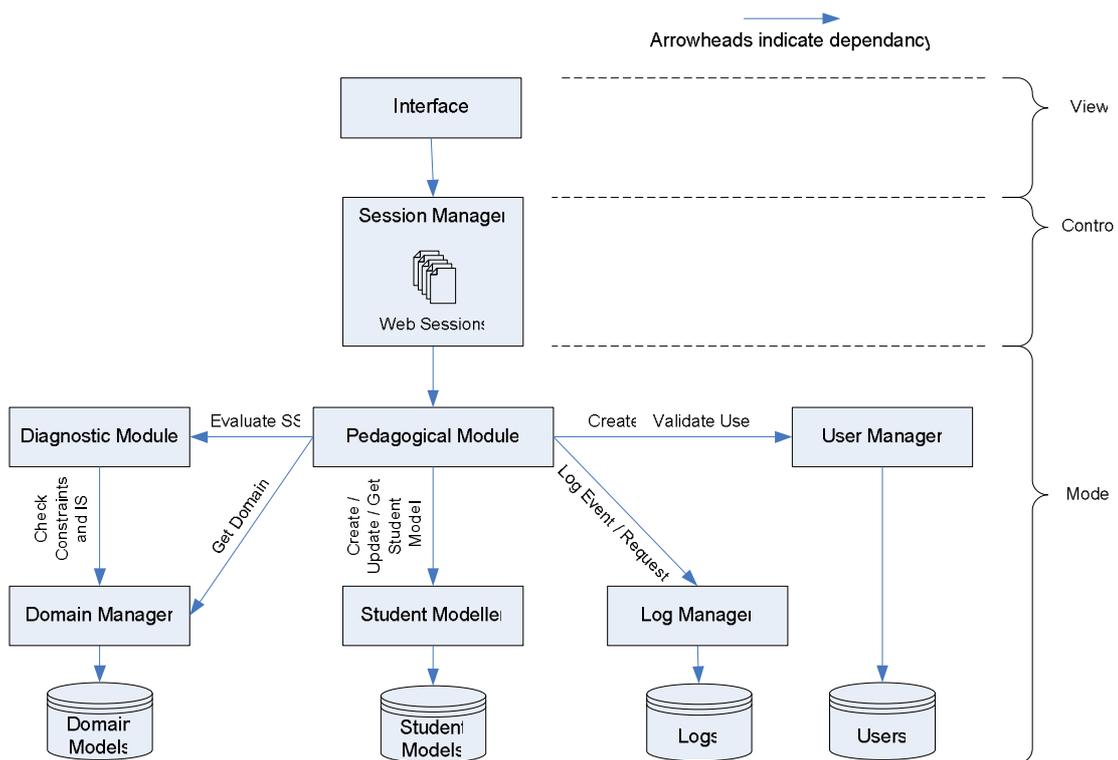


Figure 4. The architecture of ASPIRE-Tutor

Figure 5 illustrates the general “message passing” design of ASPIRE-Tutor. The general flow of control for user requests is the same for all functions; specific user requests are specialisations of this general approach.

1. The client application sends a request via the **Interface** (*the view*) to the **Session Manager** (*the controller*).
2. The Session Manager unpacks the request and passes the appropriate request(s) to the **Pedagogical Module** (*the model*). The Session Manager thus manages the *flow of control* of the interaction.
3. The Pedagogical Module decides what actions to take to fulfil the request, and does so by sending appropriate requests to the *functional modules*, i.e. any/all of the **Diagnostic Module**, **Domain Manager**, **Student Modeller**, **Log Manager** and **User Manager**. The Pedagogical Module thus manages the *pedagogical decisions* that determine what the response to each request will be.
4. Each request to a functional module results in a status and optional data being returned to the Pedagogical Module. In addition, the functional modules may access and/or update data objects, e.g. student model, domain model, logs. The various components of the model may also be updated.
5. The Pedagogical Module returns the final status and data to the Session Manager.
6. The Session Manager organises the result to be returned to the client, by packaging up a response and/or indicating what Interface object should be presented next.

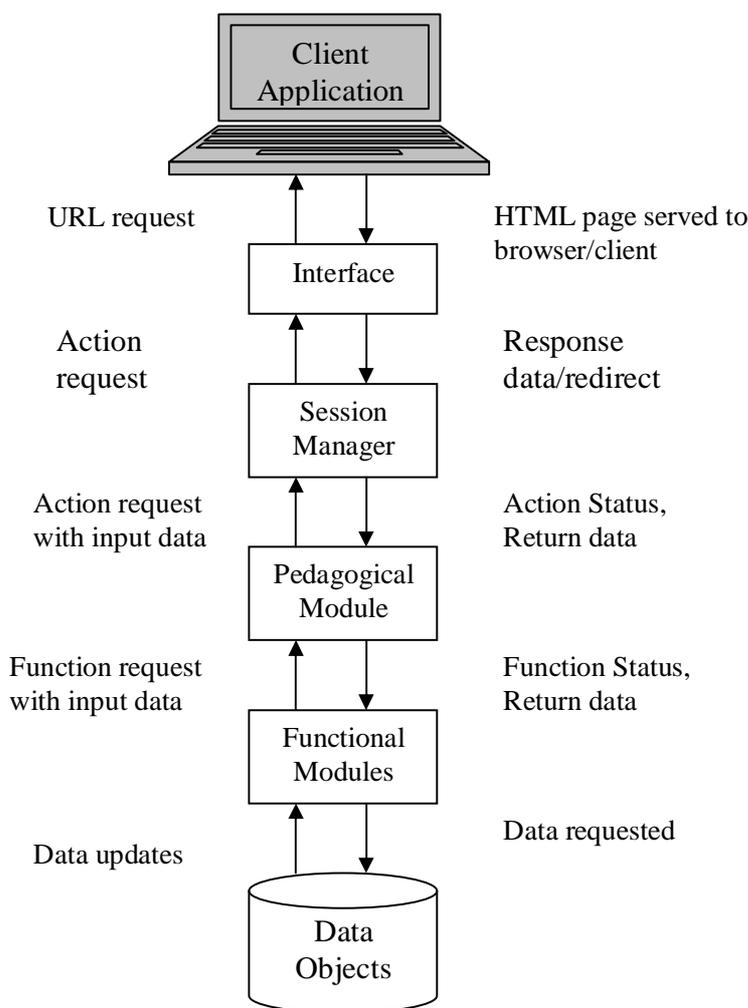


Figure 5. Flow of control in ASPIRE-Tutor

2.1.1. Interface

ASPIRE will offer three types of interfaces: Web interfaces, XML-RPC (eXtensible Markup Language Remote Procedure Call) interfaces and a typical window-based interface. Both Web and XML-RPC interfaces are based on the HTTP protocol. A Web interface is a ubiquitous way of accessing information and services on the World Wide Web. Web interfaces usually consist of sets of dynamic HTML pages.

XML-RPC is remote procedure calling using HTTP as the transport and XML as the encoding. XML-RPC is a software interface rather than a user interface: it does not have the means of visualising the data and controls required for interaction between the client and server, unlike Web interfaces, where HTML (Hyper-Text Markup Language) is used for visualisation of data and interface elements. Instead, ASPIRE will provide an XML-RPC entry point to its functionality, allowing any third-party client application to interface with ASPIRE and thus support intelligent tutoring. The implementation of the actual client interface communicating with the server through XML-RPC is left up to the designers of the client side.

A standard window-based interface is going to be used locally on the server for administration and maintenance tasks. This interface is not going to be available to authors and students.

We anticipate that the most commonly used interfaces to ASPIRE would be Web interfaces. A Web interface consists of a set of dynamic HTML pages that externalise the system functionality to the users. The Interface can be viewed in a standard Web browser. The initial page of the system may be accessed through a bookmark, link on another page, or the user may need to type the starting page URL into the location bar of the browser. For further interactions with the system, the Interface will provide navigational support like with any dynamic Web site. The user interaction with the rest of the system happens as follows:

- When the user performs input or navigates through the interface, on behalf of the user the browser sends URL requests to the Web server.
- On the server, each request is mapped to a response page.
- The server calls the functions responsible for processing the user input and actions associated with the request.
- When processing has been completed, the response page, possibly containing some dynamic content, is sent back to the browser and displayed to the user.
- The server will keep processing user requests in this way for the duration of the session.

Each client action will be represented by a separate URL. The URLs will be mapped by the Session Manager to the underlying operations on the user session data. The student functions (URLs) the interface will support are:

- Log in;
- Display main menu (a list of available tutors);
- Select/change sub-domain;
- Select new problem. There will be various options on how the problem may be selected. The user will be able to choose the next problem in the list of problems, choose any problem in the list of problems or ask for system's choice.
- Submit solution for evaluation;
- View history;
- View student model;
- Log out.

Besides the functionality mentioned above, the Interface will provide additional functionality for users other than students. However the basic principle of request/response communication will stay the same.

2.1.2. Session Manager

Communication between the browser and server is built on the basis of Hyper Text Transfer Protocol (HTTP). Since HTTP is stateless, ASPIRE is responsible for maintaining the state of each user session. The Session Manager is responsible for maintaining coherent communication with each client. Clients are usually Web browsers, but may optionally be some other applications. The Session Manager module is implemented on the basis of Allegro Web Actions framework. The Web server, AllegroServe is a part of the Web Actions framework. The Session Manager performs the following actions:

- The Web server accepts HTTP connections and retrieves URL requests associated with them;
- The Web server retrieves that data associated with each request and passes the request on to the Web Actions framework.
- For each request, the Web Actions framework associates the request with the appropriate user session.
- Based on the requested URL and data associated with it, the Session Manager passes on the request (and data) to the Pedagogical Module for processing.
- On the completion of processing, the Pedagogical Module returns the result to the Session Manager, which then updates the state of the user session.
- At this stage the server produces the appropriate response. This will either be in the form of an HTML page to be returned to the browser (either by providing content or by directing the browser to the appropriate next page), or optionally an XML response message if the client was some other application.
- Pass the response back to the client.

Detailed information about Session Manager is presented in Section 6.

2.1.3. User Manager

The User Manager is responsible for managing the set of users who are authorised to use the system. Functions include:

- Create/maintain lists of users, including usercodes and passwords, and the type of user (student, teacher, author, and administrator).
- Maintain the access rights of each user (i.e. which domains they may use).
- Validate users at login.

2.1.4. Diagnostic Module

The Diagnostic Module (DM) analyses a student's solution to the current problem by comparing it to the ideal solution using the domain model, which consists of the constraint set and associated macros. The result of this analysis is a list of errors that the student made, if there are any.

The Diagnostic Module takes as inputs the student's solution and the constraints from the Domain Manager. The outputs are the list of satisfied constraint, the list of violated constraints (including feedback messages), and the incorrect component tags (associated with each constraint).

2.1.5. Domain Manager

The Domain Manager manages the domain model and makes it available to the other modules, particularly the Diagnostic Module and the Pedagogical Module. The Domain Manager is also responsible for loading the domain models initially, when ASPIRE-Tutor is started. It also contains parsers for constraints and solutions. In addition, it externalises parsed constraints, problems and ideal solutions. More information about the domain model is given in Section 3.2.

2.1.6. Pedagogical Module

The Pedagogical Module (PM) decides how to respond to each student request, based on the current pedagogical strategy, the student model and their current request/attempt. It handles the following student's requests:

- *Select/Change sub-domain* (when applicable);
- *New problem*: There may be several options offered by the tutoring system for selecting a new problem. The options are:
 - *next problem*, in which case PM selects the problem from the list ordered by the increasing level of complexity;
 - *system's choice*, when PM selects the best problem based on the state of the student model;
 - *user's choice*, which provides the list of problems to the student and he/she can select a problem from that list.
- *Submit solution*. The pedagogical module requests the Diagnostic Module to analyse the student's submission. Based on the result of the diagnosis, PM either congratulates the students or offers feedback on errors.
- *View History*: This option allows the student to see the summary of his/her interactions with the tutoring system.
- *View Model*: This option presents a summary of the student model to the student, so that the student can reflect on their knowledge.

Implicit functions, which might be sub-modules of the PM, are:

- Curriculum sequencing (choose next sub-domain or domain concept)
- Problem selection (sequencing, dynamic problem selection)
- Feedback generation (selecting the appropriate level of feedback, possibly including dialogue generation)
- History visualisation
- Student model visualisation (and, potentially, model interaction)

2.1.7. Student Modeller

The Student Modeller (SM) is responsible for maintaining a long-term model for each student, and providing this information to other modules. SM is responsible for the following functions:

- Maintaining the state of each student model. The PM will call upon the SM to update the student model each time the student performs some significant action, such as submitting a solution. The PM will pass the required information (e.g. lists of satisfied and violated constraints) to the SM.
- Informing the PM of the student state. For example, the PM may request the current user's knowledge of a particular constraint, so that the PM can decide what level of feedback to give for this constraint.
- Student model persistence (load, save, update student models).

2.1.8. Log Manager

The Log Manager maintains and accesses logs of detailed information about all system usage, such as submissions to the system, requests for history, logging in and out, etc. Logging information is primarily used for infrequent requests for detailed information. Specific functions include:

- Write details of every student's request to the log database, including timestamps;
- Retrieve history for presenting to the student;
- Calculate learning curves and other specialised statistics for evaluation/analysis.

2.2. ASPIRE-Author

Authoring in ASPIRE-Author is a semi-automated process carried out with the assistance of an expert of the domain. The domain expert (i.e. the author) starts by developing the ontology for the chosen instructional domain, and provides the system with typical problems and their solutions. ASPIRE-Author then analyses the ontology and the problems to generate syntax and semantic constraints for the domain. Finally, the author validates the generated constraints by analysing the generated English descriptions of the constraints.

The process of generating a domain model for a domain using ASPIRE-Author involves six steps:

1. *Composing an ontology of the domain;*
During the ontology composition stage, the domain expert domain models the domain as a hierarchy of concepts using the provided Ontology Workspace.
2. *Modelling the solution representation structure;*
The author specifies the structure of solutions to problems in the chosen instructional domain.
3. *Designing the tutor's interface;*
During this step, the author specifies the characteristics of the tutoring system's interface.
4. *Adding problems and solutions;*
The author is requested to input problems and solutions of the domain during this stage. The interface of the problems and solutions editor is generated according to the solution representation structure and the interface characteristics specified during the previous step.
5. *Generating constraints (syntax and semantic);*
This step involves the generation of both syntax and semantic constraints. The Syntax Constraints Generator generates constraints by analysing the completed ontology. The Semantic Constraints Generator analyses both the ontology and the problems and solutions provided by the author to generate constraints.
6. *Validating the generated constraints.*
The constraint validation stage involves automatic validation of the constraint base by the Constraint Validation Module and manual validation by the domain expert. The domain expert would inspect a system generated high level description of each constraint and label invalid constraints. The author has to provide example problems to the system in order to illustrate the error.

ASPIRE-Author is composed of a set of modules, where each module is assigned a specific set of responsibilities in generating the domain model. The basic architecture of the ASPIRE-Author is depicted in Figure 6. The functionality of each of the modules is discussed in the following subsections.

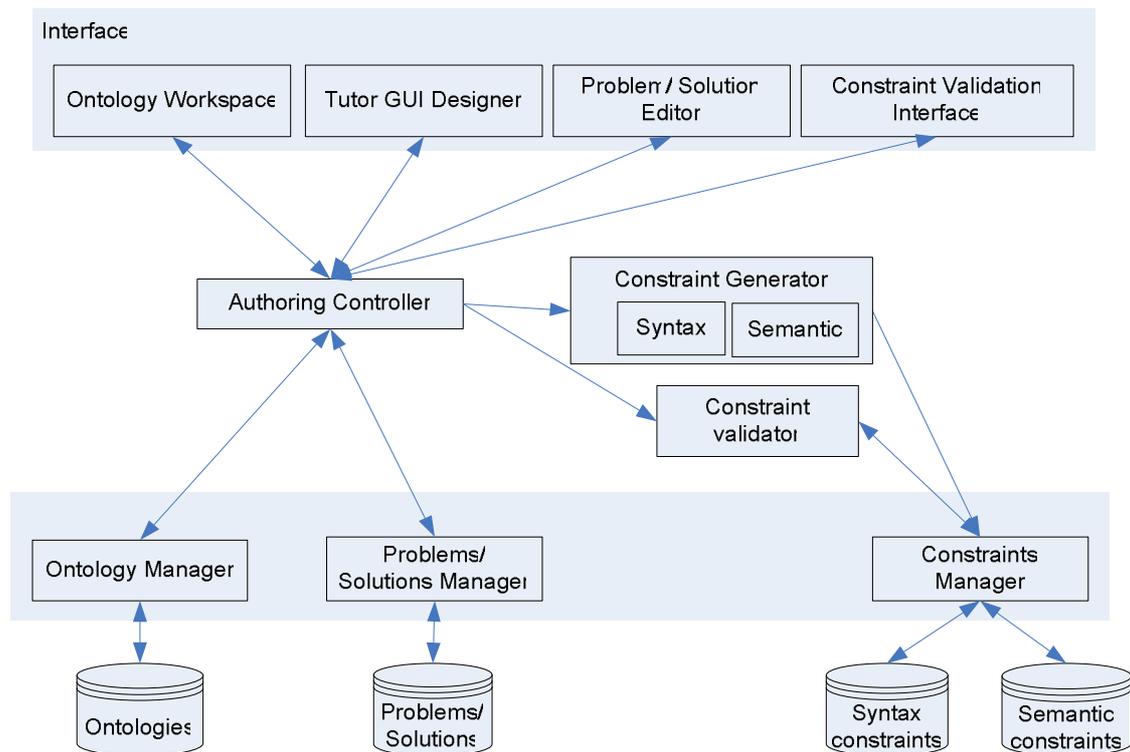


Figure 6. The architecture of ASPIRE-Author

2.2.1. Authoring Interface

The interface of ASPIRE-Author consists of four main components: Ontology Workspace, Tutor GUI Designer, Problem/Solution Editor and Constraint Validation Interface. Each component will be designed to assist the author during the respective stages of the authoring process and to ease the burden on the user.

The **Ontology Workspace** enables the author to describe the chosen instructional domain in terms of an ontology. An ontology is an explicit specification of a conceptualisation (Gruber, 1993). In other words, an ontology is a description of concepts of a domain. Typically, the domain concepts are organized in an ontology into a hierarchical structure with super-concepts and sub-concepts. The Ontology Workspace allows the author to model concepts of the ontology using the provided graphical representation. The workspace also allows the specification of properties of each concept and relationships between concepts.

The **Tutor GUI Designer** is used to perform two tasks:

- Modelling a representation for solutions
- Designing the tutoring system's student problem solving interface

The author specified a representation for solutions by decomposing the solution into components and specifying the type of elements held by each of them. Such a decomposition of a solution into meaningful components enables the student to focus only on a part of a solution while solving a problem. The diagnosis of a solution can also be structured according the solution's components.

The interface for an ITS strongly depends on the structure selected for representing solutions. The tutoring system's interface should have distinct input areas for each component of a solution, thus

visualising the solution structure. The interface design task involves the arrangement of the solution components within the interface and setting other visual characteristics such as colour, a logo etc.

The main goal of the **Problem/Solution Editor** is to provide an easy to use, intuitive interface for composing problems and their solutions. The author is encouraged to provide many correct solutions for a problem outlining different ways of solving the same problem for the process of generating constraints. The interface of the Problem/Solution Editor would reflect the interface designed using the Tutor GUI Designer. This enables the author to add problems and solutions using an interface similar to that of the student's problem-solving interface. This enables the author to evaluate the usability of the tutoring system's interface, while entering in problems and solutions.

The **Constraint Validation Interface** would be used for validating the constraints generated by the system. It will allow the user to perform the following functions:

- Inspect the generated constraints
- Mark constraints as illegal or incorrect
- Add feedback messages to constraints

2.2.2. Authoring Controller

The Authoring Controller is responsible for managing the authoring process and guiding the user through the various steps involved in authoring a domain model. The Authoring Controller receives all requests from the interface layer, initiates processes within other modules and guides the author to the different modules of the interface. The Authoring Controller receives a number of Web requests from the interface. The following requests are handled by the Authoring Controller:

- *Validate login*
 - Retrieve the user information from the Ontology Manager and if valid, invoke ontology workspace;
 - if invalid, invoke log in interface.
- *Composing ontology complete*
 - Pass on ontology to the Ontology Manager for persistence;
 - Invoke solution representation interface.
- *Solution representation modelling complete*
 - Pass on solution representation information to the Problem Manager;
 - Invoke Tutor GUI Designer.
- *Tutor GUI design complete*
 - Pass on the GUI design specification to the Ontology Manager;
 - Invoke the Problem Editor (pass the solution representation and GUI design information)
- *Problems and solution editing complete;*
 - Pass on Problems and solutions to problem-manager
 - Invoke syntax constraints generator (pass ontology)
 - Invoke semantic constraints generator (Pass ontology, problems and Solutions)
 - Invoke constraint validator (Pass generated constraints)
 - Invoke constraint validation interface (Pass generated constraints)
- *Constraint validation complete;*
 - Pass on constraints to constraint-manager for updating constraint bases
- *Generate tutoring system.*
 - Invoke *load domain model* of ASPIRE-Tutor.

2.2.3. Constraint Generator

The **Syntax Constraint Generator** is responsible for generating syntax constraints by analysing a completed ontology. Semantic constraints are generated by the **Semantic Constraint Generator**. It analyses the problems and their solutions as well as the ontology for generating semantic constraints. Multiple solutions provided for a single problem are used for generating constraints that are able to identify different correct solutions by comparing the student's solution to a single ideal solution.

2.2.4. Constraint Validator

The **Constraint Validator** is responsible for the validation of constraints generated by the constraint generators. The validation process involves a high level of interaction between the author and the system. Depending on the validation strategy used, the Constraint Validator may generate solutions to be validated by the author, or generate an English description of the constraint to be presented to the author for validation.

2.2.5. Ontology Manager

The **Ontology Manager** maintains ontologies and makes them available to the Authoring Controller. It is responsible for loading, saving and updating of ontologies and tutoring interface specifications. It also keeps a track of author details including their user ids and password information for authenticating users.

Transfer of data between the interface residing at the user's browser and the Authoring Controller is achieved using an XML representation. Converting the database representation of the ontology to XML and vice-versa is function of the Ontology Manager.

2.2.6. Problem/Solutions Manager

Problems and their solutions are managed by the **Problem/Solutions Manager**. It is responsible for saving, retrieving and modifying problems and solutions. Similar to the Ontology Manager, this module is also capable of converting the database representation of problems and solutions into an XML representation.

2.2.7. Constraints Manager

The Constraint Manager is responsible for managing both semantic and syntax constraints for a domain. It has the ability to load, save and update constraints that belong to both types.

3. Data Model

Central to the ASPIRE system is the representation of the relevant data structures, and how these will be processed. The main aspects are:

- What data structures are required?
- What “language” will they be represented in?
- How will they be stored in the system?
- How will they be processed?

Note the following information describes the *static* representation of the domain knowledge; these will be loaded when the server is started (or a domain reload request is received), and may be stored in some other format within ASPIRE. The main data objects required by the system are given in Figure 7, which shows the UML class diagram for the ASPIRE-Tutor system³.

3.1. User Data

All the classes discussed in this subsection belong to the *User Manager*. The **User** class contains information that is global to an individual user, including the username, password and any other general user profile information. There are different types of ASPIRE users (shown in Figure 6 as subclasses of the **User** class):

- **Student**: interacts with the tutoring system(s);
- **Author**: creates new domains / modifies the content of existing domains;
- **Teacher**: sets up access to the tutors (creates users and maps them to domains that are available on the system);
- **Administrator**: maintains the system, control deployment of new tutors, control the creation of “privileged” accounts (i.e. those that are not student accounts).

One of the responsibilities of a teacher is to set up a group of students who are allowed to use a specific tutoring system. The **Group** class allows the teacher to specify the students who are authorised to use a specific tutoring system. The teacher can also specify a particular **Pedagogical Setting** for a group of students. This class includes specific pedagogical strategies (such as problem selection or feedback selection) for each group, as well as other options, such as the date of pre-post tests.

³ This diagram is simplified for the purposes of this document. It does not include any of the operations/functions of the classes/objects and many of the attributes are only explained in the text of this section.

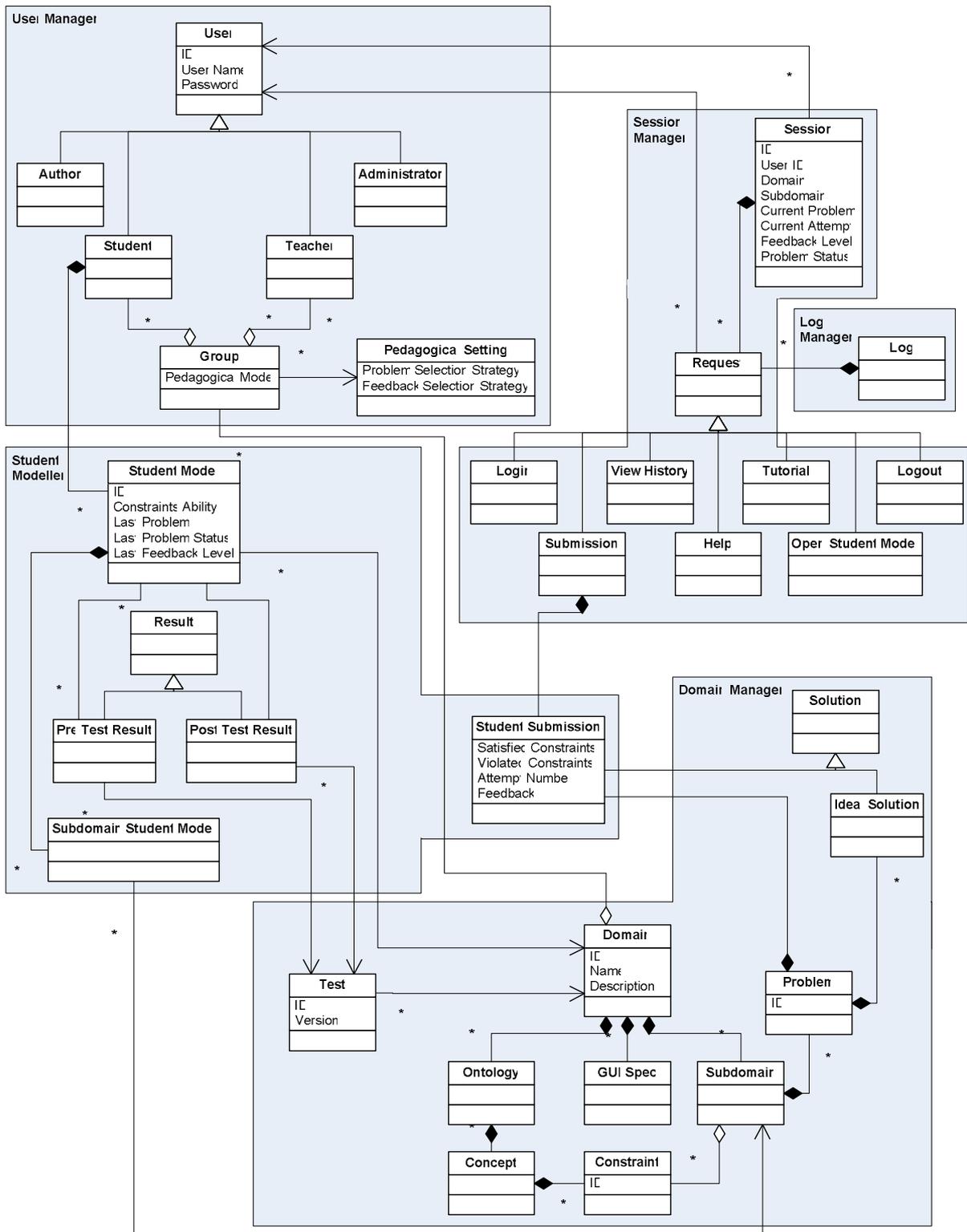


Figure 7. UML class diagram for ASPIRE-Tutor

3.2. Domain Model

The domain model consists of all knowledge necessary to solve problems in a particular instructional domain. Domain Models belong to the *Domain Manager*. Each domain model consists of the following components:

- Domain definition: specifies parameters that indicate how the system should operate, including style information.
- Sub-domain definition: a domain may have one or more sub-domains.
- Problems: Each problem consists of multiple parts, including the problem text (which may include machine-readable elements such as tags) and the solution. There may be multiple problem sets if there are multiple sub-domains for this domain.
- Constraints and associated macros/functions
- Ontology(s) for authoring and feedback control

Note that a domain may also contain sub-domains. In general, each of the above can appear at either the domain or sub-domain level, except problem sets, which always occur at the sub-domain level.

3.2.1. Domain/Sub-domain definition

The **Domain** and **Sub-domain** classes allow each instructional domain/sub-domain to be specified in terms of a number of parameters including:

- Dialogues or other domain/subset-specific pedagogical information
- Domain-wide appearance parameters, e.g.
 - Field sizes
 - Scaffolding information
 - Domain description (for menus)
 - What to call sub-domains (domain only)
 - Appearance information
- Generic feedback messages for this domain
- Domain-specific help (i.e. help text that is displayed if the student requests help on this domain/subset).

3.2.2. Ontology

An ontology is a specification of domain concepts and their mutual relationships (Gruber, 1993). The ontology for an instructional domain would be developed in the authoring phase. The ontology is used also in the deployment phase (i.e. ASPIRE-Tutor) by the pedagogical module to make various pedagogical decisions, such as instructional planning. The ontology is also used by the Domain Manager to organise constraints.

The **Ontology** class specifies the properties of a specific domain. It consists of a number of domain concepts. The **Concept** class specifies the features of each domain concept. Low-level domain concepts are related to constraints, which specify the features of constraints that are pedagogically relevant.

3.2.3. Problems

Each sub-domain has a set of problems defined for it. The **Problem** consists of:

- Problem number
- Problem name
- Problem difficulty
- Problem text (human-readable)
- Problem statement (machine-readable)
- Solution (human readable)
- Solution (machine readable)

For each problem, the author defines an ideal solution (i.e. a correct solution). The **Ideal Solution** class contains the correct solution for each problem.

3.2.4. Constraints

Constraints are the “knowledge units” of the domain model. They may be stored at the domain or a sub-domain level. The **Constraint** class consists of:

- Constraint number
- Relevance condition
- Satisfaction condition
- Feedback message(s)
- Related component of the solution
- Variable being tested (for returning a list of the erroneous parts)

The relevance and satisfaction conditions will be stored as logical expressions incorporating pattern matches. The language for specifying constraints is defined in Section 5.

3.2.5. Macros/Functions

Macros and functions are pieces of logic code that may be incorporated into constraints. They have three main functions:

- Reduce redundancy;
- Separate generic domain logic from declarative knowledge that is particular to the current problem set;
- Make the constraints more maintainable and readable.

Macros/functions could either be written in the same language as the constraints or be code fragments (e.g. LISP functions). The knowledge representation language is presented in Section 5.

3.2.6. Test

For each instructional domain the author may define a number of tests to be used to assess the student’s knowledge. The **Test** class stores the test number and the questions to be asked. For each question, there is a list of options offered as answers, the correct answer, and the feedback message that would be given to the student for each selected option.

3.3. Student Modelling

The User Manager will maintain general information about each student (including the user code and password) as attributes of the User class. Within ASPIRE-Tutor it is necessary to keep a model of the student's knowledge for each instructional domain. This information is distributed over several data classes covered in this subsection, all of which belong to the Student Modeller.

3.3.1. Pre/post Test Results

In instructional domains for which tests have been specified, the student will be given one of the tests (selected randomly) as a pre-test. The student's result on the pre-test would serve as an initial indication of the student's knowledge of the domain (i.e. pre-existing background knowledge). The pre-test would be given the first time the student log on to the tutoring system. When the teacher has specified that the students should sit a post-test⁴, a different test would be given to the student as a post-test. The result of the post-test serves as an indication of the student's knowledge after interacting with the tutoring system. The **Pre-test Result** and the **Post-test Result** classes store the date the test was taken, the student's answer to each question, and the total score.

3.3.2. Student Model

The **Student Model** class contains a model of the student's knowledge of a particular instructional domain. This includes:

- Performance on each domain constraint the student has used;
- Proficiency for the domain as a whole;
- The last sub-domain the student has worked on;
- The last problem the student has attempted;
- Information about the feedback the student received on the last submission;

3.3.3. Sub-domain Student Model

If the instructional domain contains several sub-domains, it is also necessary to store the student's level of proficiency for each individual sub-domain. The Sub-domain Student Model class includes:

- Proficiency for this sub-domain;
- The last problem worked on within the sub-domain;
- A list of the problems and their states (one of *new*, *attempted* or *solved*);

3.3.4. Student Submission

The **Student Submission** class records detailed information about every submission received by the system. It contains the following information:

- The date/time the submission was received;
- The selected instructional domain;
- The problem the student is working on;
- The attempt number on the current problem;
- The raw submission received;

⁴ Information about pre-post test dates is stored in the Pedagogical Setting class.

- The state of the solution (correct or incorrect);
- A list of satisfied constraints and their bindings;
- A list of violated constraints and their bindings;
- The feedback that the student has received on the submission.

3.4. Sessions

The Session class contains information about current sessions. For each session, the Session Manager maintains information about the user and the state of the session. Depending on the type of user (such as student or teacher), the Session Manager stores the information necessary for ASPIRE to provide continuous support. More information about this class can be found in Section 6.

4. Functionality: User Stories

As specified in Section 3.1, there are four classes of users in ASPIRE: student, author, teacher and administrator. In this section we present the use cases for all types of users.

4.1. Student User Stories

Figure 8 illustrates the use cases for a student interacting with the tutoring system. Each user case is then described via a “user story”.

4.1.1. A student logs in

- The student accesses the tutoring system via a Web browser or a client application (XML-RPC). For Web users, they are taken to a login page. A student logs in only once, and can then use whichever domains they have access to.
- The student enters a username and password.
- A “new guest account” checkbox is presented. Selecting this checkbox indicates the student only wishes to have access to those domains that are publicly available. Otherwise, the student needs to have been registered on the system.
- The user selects “submit” and a URL request (“log in”) is sent.
- The request is received by AllegroServe, initiating a Web action in the Session Manager. The Session Manager checks for an existing session (but fails to find one).
- The Session Manager sends a “check user” request to the Pedagogical Module, passing the student’s username and password.
- The Pedagogical Module sends a “check user” request to the User Manager.
- The User Manager checks the username/password is valid:
 - If the user has requested to be a *new guest user*, there *must not* be a matching username in the database. In this case the User Manager will create the new user record;
 - If the user has *not* requested to be a new guest user, there *must* be a valid user. In this case the User Manager will simply return a “success” status.

- The Pedagogical Module sends a “log event” request to the Log Manager, along with the relevant information.
- The Session Manager determines what to do next:
 - For a Web user, the “select domain” form is displayed;
 - For an XML-RPC user a “success” response is returned, along with a list of valid domains for this user.

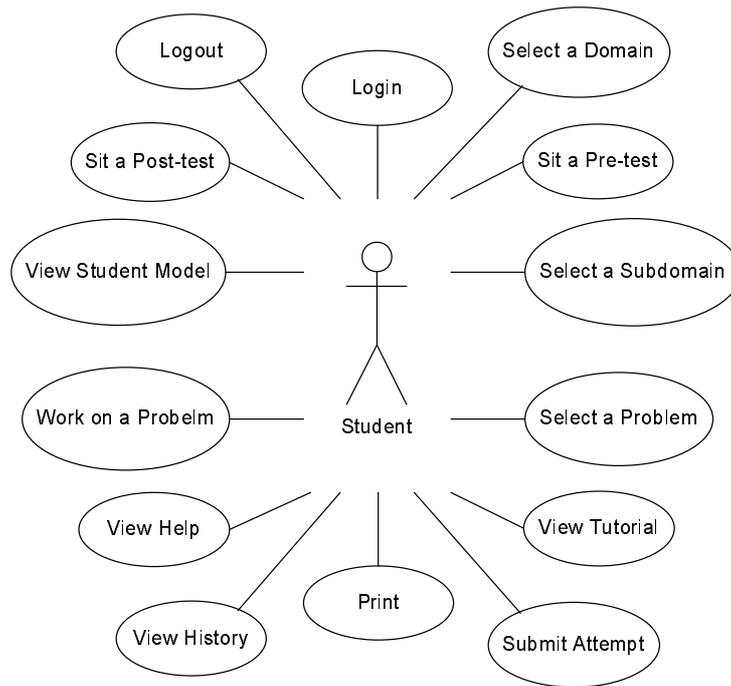


Figure 8. Student Use Cases

4.1.2. The student selects a domain to work on

- The student selects a domain (i.e. a tutoring system) to work on.
- The client (browser or application) sends a “set domain” request to the Session Manager.
- The Session Manager sends a “set domain” request to the Pedagogical Module, passing the student and domain names.
- The Pedagogical Module sends a “load student model” request for this student/domain to the Student Modeller:
 - If the student model exists, it is loaded;
 - If the student model does *not* exist, it is created by the Student Modeller.
- The Pedagogical Module sends a “log event” request to the Log Manager, along with the relevant information.
- The Session Manager determines what to do next:
 - For a web user, the “select subset” form is displayed;
 - For an XML-RPC user a “success” response is returned.

4.1.3. The student selects a sub-domain to work on

- The student selects the sub-domain via a Web page or in a client application.
- A “set subset” request is sent to the Session Manager.
- The Session Manager retrieves the user’s session state from the Web Session, and sends a “set sub-domain” request to the Pedagogical Module.
- The Pedagogical Module sends a “set sub-domain” request to the Student Modeller.
- The Student Modeller sets the current sub-domain for this domain/user in the appropriate student model.
 - If the student sub-domain model exists, it is loaded;
 - If the student sub-domain model does *not* exist, it is created by the Student Modeller.
- The Pedagogical module selects the next problem for the student to solve:
 - If the last problem they were working on is unsolved, this is selected, else;
 - If the student has unsolved problems in this domain/sub-domain, the most recent of these is selected, else;
 - The Pedagogical Module selects the most appropriate problem.
- The Pedagogical Module sends a “set problem” request to the Student Modeller.
- The Student Modeller sets the current problem to be the one selected. If the student has already attempted this problem, the last attempt is returned.
- The Pedagogical Module sends a “log event” request to the Log Manager, along with the relevant information.
- The Pedagogical Module returns the status and selected problem to the Session Manager.
- The Session Manager decides what to do next:
 - If a Web request, the main interface page is displayed, which serves up the selected sub-domain;
 - If an XML-RPC request, the status and selected sub-domain are returned.

4.1.4. The student sits a pre/post test

- The Pedagogical Module determines when the student is required to sit a pre/post test, and sends a “get pre-test” request to the Student Modeller.
 - If the student pre-test result exists, it is returned;
 - If the student pre-test result does *not* exist, “does not exist” status is returned.
- The Pedagogical Module decides which test version the student should sit, and requests it from the Domain Manager.
 - If the student pre-test result exists, a different test version is presented as a post-test;
 - If the student pre-test result does *not* exist, a randomly selected test version is presented as a pre-test.
- The Session Manager determines what to do next:
 - If a Web application, the test page is displayed.
 - If an XML-RPC application, the test is returned to the caller.
- The student sits the test and submits the solution.
- A “submit test” request is received containing:
 - The student/domain/sub-domain/test version;
 - The student’s solution.
- The Session Manager retrieves the session context.
- The Session Manager sends a “check test solution” request to the Pedagogical Module.
- The Pedagogical Module sends a “diagnose test solution” request to the Diagnostic Module, passing the solution and domain/subset/test version.

- The Diagnostic Module sends a “get test” request to the Domain Manager, which returns the solution.
- The Diagnostic Module checks the solution using the domain model, and returns the following to the Pedagogical Module:
 - The overall outcome (succeeded or failed);
 - A list of correct question numbers;
 - A list of incorrect question numbers.
- The Pedagogical Module sends an “update student model” request to the Student Modeller.
- The Pedagogical Module determines what action to take and communicates this back to the Session Manager. This could be one of:
 - Display feedback. The Pedagogical Module will retrieve the feedback from the Domain Manager (per question) and pass this back to the Session Manager.
 - Start a dialogue. The Pedagogical Module will retrieve the dialogue content from the Domain Manager and return it to the Session Manager.
- The Pedagogical Module sends a “log event” request to the Log Manager, along with the relevant information.
- The Session Manager determines what to do next:
 - If a Web request, the feedback is displayed according to the action type returned from the Pedagogical Module.
 - If an XML-RPC call, the feedback information is returned to the caller.

4.1.5. The student selects a problem to work on

- The student requests a problem to work on. This may be one of:
 - The NEXT problem in the domain
 - The SYSTEM’s choice
 - A specific problem number
- A “select problem” request is sent to the Session Manager.
- The Session Manager retrieves the user’s session state from the Web Session, and sends a “select problem” request to the Pedagogical Module.
- The Pedagogical Module requests information about problems solved from the Student Modeller.
- The Pedagogical Module selects the problem according to the method requested.
- The Pedagogical Module sends a “set problem” request to the Student Modeller.
- The Student Modeller sets the current problem to be the one selected. If the student has already attempted this problem, the last attempt is returned to the Pedagogical Module.
- The Pedagogical Module sends a “log event” request to the Log Manager, along with the relevant information.
- The Pedagogical Module returns the problem/solution information to the Session Manager.
- The Session Manager decides what to do next:
 - If a Web request, the main interface page is displayed, which serves up the selected problem;
 - If an XML-RPC request, the status and selected problem are returned.

4.1.6. The student works on a problem

- The student is presented with the appropriate problem interface. Each domain has its own interface with its appropriate fields/components.
- The student works through a problem on an attempt to solve it. This may include:
 - Entering an input;

- Making a selection.
- The client application provides the student with immediate cues in response to the student interactions and updates the interface when necessary.

4.1.7. The student submits an attempt

- A “submit” request is received from either a Web browser or an XML-RPC application, containing:
 - The student/domain/sub-domain/problem number;
 - The required feedback level;
 - The student’s solution.
- The Session Manager retrieves the session context.
- The Session Manager sends a “check solution” request to the Pedagogical Module.
- The Pedagogical Module sends a “diagnose solution” request to the Diagnostic Module, passing the solution and domain/subset/problem number.
- The Diagnostic Module sends a “get problem” request to the Domain Manager, which returns the *parsed* ideal solution.
- The Diagnostic Module sends a “parse solution” request to the Domain Manager, passing the student’s solution, and receives the parsed solution.
- The Diagnostic Module checks the solution using the domain model, and returns the following to the Pedagogical Module:
 - The overall outcome (succeeded or failed);
 - A list of satisfied constraint numbers;
 - A list of violated constraint numbers and identifiers (bindings) for the failed components.
- The Pedagogical Module sends an “update student model” request to the Student Modeller, passing the lists of constraints.
- The Pedagogical Module determines what action to take and communicates this back to the Session Manager. This could be one of:
 - Display feedback. The Pedagogical Module will retrieve the feedback from the Domain Manager (per constraint) and pass this back to the Session Manager.
 - Start a dialogue. The Pedagogical Module will retrieve the dialogue content from the Domain Manager and return it to the Session Manager.
- The Pedagogical Module sends a “log event” request to the Log Manager, along with the relevant information.
- The Session Manager determines what to do next:
 - If a Web request, the feedback is displayed according to the action type returned from the Pedagogical Module.
 - If an XML-RPC call, the feedback information is returned to the caller.

4.1.8. The student views help/history/tutorial/student model

- A “view” request is received from either a Web browser or an XML-RPC application. This may be one of:
 - view help
 - view history
 - view tutorial
 - view student model
- The Session Manager retrieves the session context, and sends the request to the Pedagogical Module.

- The Pedagogical Module determines where to retrieve the information from:
 - If a “view help”/“view tutorial” request, the request is sent to the Domain Manager.
 - If a “view history” request, the request is sent to the Log Manager.
 - If a “view student model” request, the request is sent to the Student Modeller.
- The Pedagogical Module sends a “log event” request to the Log Manager, along with the relevant information.
- The Pedagogical Module returns the information to the Session Manager.
- The Session Manager determines what to do next:
 - If a Web request, the information is displayed.
 - If an XML-RPC call, the information is returned to the caller.

4.1.9. The student prints

- The student chooses to print their current screen, and selects which sections of the interface to print. This may include:
 - The current problem text
 - The current attempt
 - The current feedback message(s)
- The Session Manager receives a “print” request and its parameters from the Web browser or client application and determines what to do next:
 - If a Web request, the “print preview” page is displayed followed by a print dialog to set the printer and related options.
 - If an XML-RPC call, the information is returned to the caller.

4.1.10. The student logs out

- The student requests to log out of the tutoring system.
- The Session Manager retrieves the session context, and sends a “log out”, request to the Pedagogical Module.
- The Pedagogical Module sends an “update student model” request to the Student Modeller, passing the last state of the current session, this may include:
 - The current problem and its state; whether new, attempted or finished.
 - The current attempt.
 - The level of the last feedback.
- The Pedagogical Module sends a “log out” request to the User Manager.
- The Pedagogical Module sends a “log event” request to the Log Manager, along with the relevant information.
- The Session Manager deletes the current student session once the Pedagogical Module returns “success” status.
- The Session Manager determines what to do next:
 - For a Web user, the “login” page is displayed;
 - For an XML-RPC user a “success” response is returned.

4.2. Author User Stories

Figure 9 illustrates the use cases for an author interacting with ASPIRE. Each user case is then described via a “user story”.

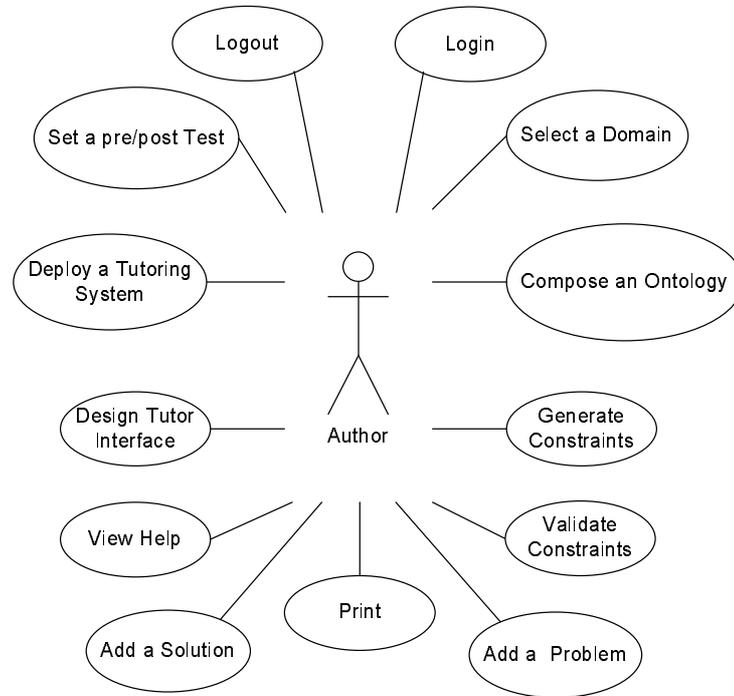


Figure 9. Author use cases

4.2.1 An author logs in

- The author accesses the authoring system via a Web. The users are taken to a login page. (NB: An author logs in only once, and can then use whichever domains they have created or create new domains.)
- The author enters a username and password.
- The user selects “submit” and a URL request (“log in”) is sent.
- The request is received by AllegroServe, initiating a Web action.
- The Authoring Controller sends a “check user” request to the ASPIRE-Tutor, passing the author’s username and password.
- The ASPIRE-tutor checks the user is valid:
 - If the author is valid, a “success” status is returned.
- The Authoring Controller retrieves a list of any previously created domains by this author.
- The “select domain” form is displayed.

4.2.2. The author selects a domain to work on

- The author selects a domain to work on.
- The client sends a “set domain” request to the Authoring Controller, passing the author and domain names.
- The Authoring Controller sends a “retrieve ontology” request for this author/domain to the Ontology Manager:
 - If the ontology exists, it is returned;
 - If the ontology does *not* exist, a new/blank one is created by the Ontology Manager.
- The “ontology workspace” form is displayed with retrieved ontology

4.2.3. The author composes an ontology

- The author composes, or edits an existing, ontology for their domain using the ontology workspace.
- The Authoring Controller receives the completed ontology, and passes it to the Ontology Manager.
- The Ontology Manager saves/persists the ontology and returns a “success” status.

4.2.4. The author designs the tutor interface

- The author chooses to design the tutor GUI interface.
- The Authoring Controller invokes the “solution representation” form for identifying the composition of a solution.
- At the completion of modelling the solution structure, the interface passes on the solution structure and interface parameters to the Authoring Controller.
- The Problem Manager receives the solution structure for persistence and returns a “success” status.
- The Ontology Manager receives the interface parameters for persistence and returns a “success” status.

4.2.5. The author adds a problem/solution

- The author chooses to add a problem to the current domain/ontology. The author may choose to divide the domain into sub-domains, each with its set of problems and solutions, and constraints.
- The Authoring Controller invokes the “problem editor” and passes the interface parameters.
- The author enters the problems and solutions. A problem may have multiple correct solutions. The author is encouraged to supply many problems, each with a set of different solutions to illustrate different ways of solving the same problem.
- The Authoring Controller receives “add problem”/“add solution” requests along with the complete set of problems/solutions.
- The Authoring Controller passes the problem/solution to the Problems Manager for persistence.

4.2.6. The author generates constraints

- The author chooses to generate constraints.
- The Authoring Controller retrieves the ontology from the Ontology Manager and invokes the “syntax constraints generator” passing in the ontology.

- Upon completion, the generated syntax constraints are passed to the Constraint Manager for persistence.
- The Authoring Controller retrieves the problems and solutions from the Problems Manager and invokes the “semantic constraints generator”, passing the ontology and the problems and solutions.
- The generated constraints are passed to the Constraint Manager for persistence.
- Authoring Controller invokes the “constraints validation” form.

4.2.7. The author validates the constraints

- The author chooses to validate the constraints.
- The Authoring controller retrieves constraints from constraint manger and passes them to the Constraint Validator for automatic validation.
- The Authoring Controller invokes the “Constraint validation interface” and passes the automatically validated constraints, for the author to validate.
- The author validates the constraints and adds appropriate feedback messages to the constraints.
- The author validated constraints and an “update constraints” request are sent by the Authoring Controller to the Constraint Manager.

4.2.8. An author deploys a tutoring system

- The author completes the development of the new tutoring system (or modifies an existing one) using ASPIRE-Author.
- ASPIRE-Author sends the domain model to ASPIRE-Tutor and requests to the domain to be (re)loaded.

4.2.9. The author prints

- The author chooses to print the current screen.
- The Authoring Controller receives a “print” request and its parameters from the Web browser or client application.
- The Authoring Controller invokes the “print preview” page followed by a print dialog to set the printer and related options.

4.2.10. The author views help

- The author requests to view the help page of the current task.
- The Authoring Controller receives the “view help” request, along with the current task name.
- The Authoring Controller selects the appropriate help page to display, and returns it to the interface.

4.2.11. The author logs out

- The author requests to log out of the authoring system.
- The Authoring Controller retrieves the session context, and determines what to do next:
 - If there are any unsaved changes, the author is asked whether to save or not.
- The “login page” is displayed.

4.3. Administrator User Stories

Figure 10 illustrates the use cases for an Administrator interacting with ASPIRE-Tutor, while Figure 11 shows the use cases for an administrator interacting with ASPIRE-Author. Each use case is then described via a “user story”.

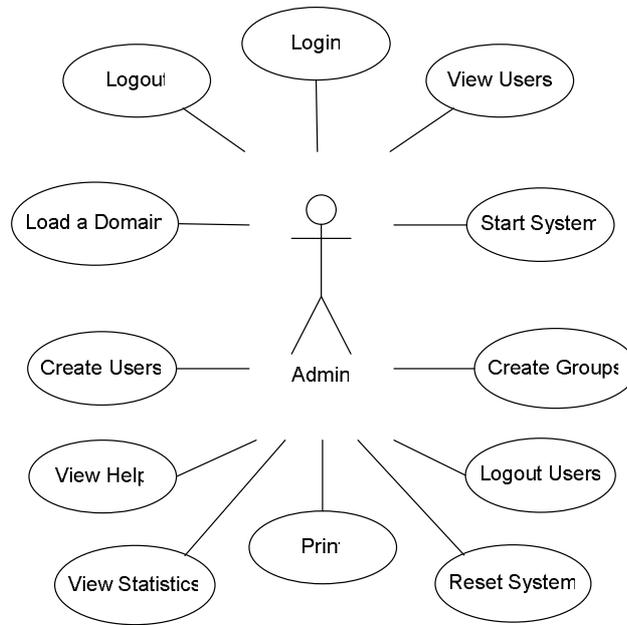


Figure 10. The ASPIRE-Tutor administrator use cases

4.3.1. The administrator starts ASPIRE-Tutor

- The administrator starts the ASPIRE-Tutor server.
- The system loads the system-wide parameters.
- For each domain, the system loads the domain parameters, the domain model (constraints and macros), the Ontology information, and the subsets (if any).
- The system publishes any domain-specific Web actions.
- The system starts the AllegroServe Web server.
- The ASPIRE-Tutor server is now available to accept requests.

4.3.2. The administrator logs into the admin system of ASPIRE-Tutor

- The user requests the administrator login screen (a URL).
- The user enters the administrator password.
- The user selects “submit” and a URL request (“log in”) is sent.
- A “login administrator” request is sent to the Session Manager.
- The request is received by AllegroServe, initiating a Web action in the Session Manager. The Session Manager checks for an existing session (but fails to find one).
- The Session Manager sends a “check user” request to the Pedagogical Module.

- The Pedagogical Module sends a “check user” request to the User Manager, with “administrator” as the username.
- The User Manager checks the username/password and responds accordingly.
 - If the username/password are valid, a “success” message is returned.
 - If not, a “failed: invalid password” message is returned.
- The Session Manager responds accordingly:
 - If the login was successful, the session is set. The browser is redirected to the administrator page.
 - If the login was unsuccessful, the browser is returned to the login page with an appropriate message.
- Once successfully logged in, the administrator is presented with the administration page, which gives the options of viewing the following information:
 - A list of domains that are currently deployed, including their status (loaded/not loaded).
 - For each domain, a list of current users.
 - Overall system status (e.g. system uptime).

4.3.3. The administrator (re)loads a domain

- The administrator requests the “domain administration” screen.
- The Session Manager sends a “get all domains” request to the Pedagogical Module.
- The Pedagogical Module requests all available domains from the Domain Manager.
- The Domain Manager returns the list of all domains, with notes on which are already loaded, back to the Session Manager.
- The list of domains is displayed, each domain showing its status (loaded/unloaded), and all domains are selectable for loading/reloading.
- Once an option is selected, the appropriate domains are loaded as per starting the system.

4.3.4. The administrator logs out of the admin system of ASPIRE-Tutor

- The administrator requests to log out of the tutoring system.
- The Session Manager retrieves the session context, and sends a “log out”, request to the Pedagogical Module.
- The Pedagogical Module sends a “log out” request to the User Manager.
- The Session Manager deletes the current administrator session once the Pedagogical Module returns a “success” message.
- The “login administrator” page is displayed.

4.3.5. The administrator logs a user or group of users out of ASPIRE-Tutor

- The administrator requests the “View users” screen.
- The Session Manager sends a “get logged-in users list” request to the Pedagogical Module.
- The Pedagogical Module requests all currently logged-in users from the User Manager.
- The list of logged-in users gets displayed, with options to log each user out.
- The administrator selects a user to log out.
- A “logout other user” request is sent to the Session Manager.
- The Session Manager retrieves the session context of the user requested for logout, and sends a “logout” request to the Pedagogical Module.

- If the user is a student, the Pedagogical Module sends an “update student model” request to the Student Modeller, passing the last state of the selected student session, this may include:
 - The current problem and its state; whether new, attempted or finished.
 - The current attempt.
 - The level of the last feedback.
- The Pedagogical Module sends a “log out” request to the User Manager.
- The Session Manager deletes the user session once PM returns a “success” message.
- The updated list of logged-in users is displayed.

4.3.6. The administrator views help for ASPIRE-Tutor

- The administrator requests help.
- The Session Manager retrieves the session context, and sends the request to the Pedagogical Module.
- A “view help” request is sent to the Domain Manager.
- The Pedagogical Module returns the information to the Session Manager.
- The Session Manager displays the help information.

4.3.7. The administrator views the ASPIRE-Tutor statistics

- The administrator requests the “view system statistics” screen.
- For each general system statistic, the Pedagogical Module is queried.
- The Pedagogical Module determines where to look for the information, and calculates the required values.
- The values are returned to the Session Manager.
- The Session Manager interprets the values according to the type of statistic, and returns a page representing these values.

4.3.8. The administrator resets ASPIRE-Tutor

- The administrator requests a system-reset.
- The Session Manager nulls all sessions, and sends a “delete all domains” request to the Pedagogical Module.
- The Pedagogical Module sends a “delete all domains” request to the Domain Manager.
- The Domain Manager nulls all loaded domains, and returns a “success” message.
- A page confirming the system-reset is displayed.

4.3.9. The administrator views ASPIRE-Tutor users

- The administrator requests the “View users” screen.
- The Session Manager sends a “get logged-in users list” request to the Pedagogical Module.
- The Pedagogical Module requests all currently logged-in users from the User Manager.
- The list of logged-in users gets displayed.

4.3.10. The administrator prints the screen

- The administrator chooses to print the current screen, which will be the statistics of a student or group of students, or general system statistics.
- The Session Manager receives a “print” request and the parameters on which statistics to print from the Web browser.
- The “print preview” page is displayed followed by a print dialog to set the printer and related options.

4.3.11. The administrator creates ASPIRE-Tutor users

- The administrator requests the create user screen.
- The administrator enters a username for the user, an email address, and selects a role.
- A “create user” request is sent to the Session Manager.
- The Session Manager retrieves the user’s session state from the Web Session, and sends a “create user” request to the Pedagogical Module.
- The Pedagogical Module sends a “create user” request to the User Manager.
- The User Manager checks to see whether another user exists with the same username.
 - If no such name exists the user profile is created on the system, and an email with a randomly generated password is sent to the user. A “success” message is returned to the Pedagogical Module.
 - If it does exist, a “failed: user exists” message is returned to the Pedagogical Module.
- The request result is returned to the Session Manager.
- If the creation was successful, the “User Created” confirmation page is displayed, otherwise an appropriate failure message is displayed.

4.3.12. The administrator creates groups

- The administrator requests the create group screen.
- The administrator enters a name for the group.
- A “create group” request is sent to the Session Manager.
- The Session Manager sends a “create group” request to the Pedagogical Module.
- The Pedagogical Module sends a “create group” request to the User Manager.
- The User Manager checks to see whether another group exists with the same name.
 - If no such name exists the group is registered on the system. A “success” message is returned to the Pedagogical Module.
 - If it does exist, a “failed: group exists” message is returned to the Pedagogical Module.
- The request result is returned to the Session Manager.
- If the creation was successful, the “Group Created” confirmation page is displayed, otherwise an appropriate failure message is displayed.

4.3.13. The administrator starts ASPIRE-Author

- The administrator starts the ASPIRE-Author server.
- The system loads the system-wide parameters from a file.
- The system publishes any domain-specific Web actions.
- The starts the AllegroServe Web server.

- The ASPIRE-Author server is now available to accept requests.

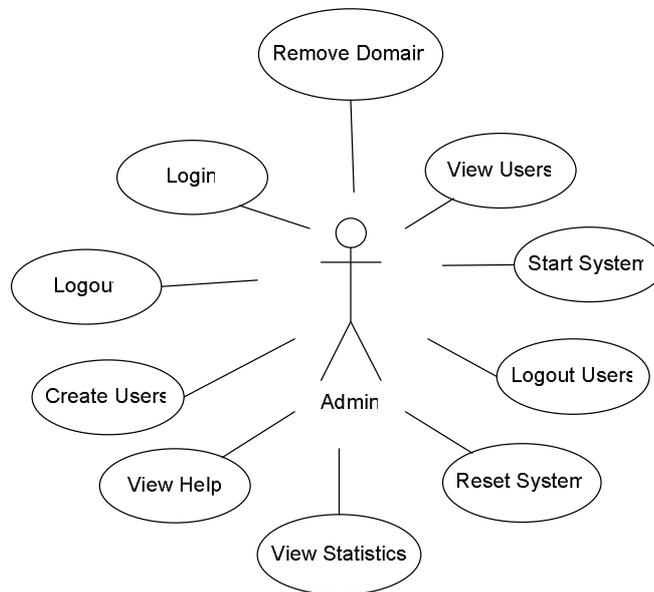


Figure 11. The ASPIRE-Author administrator use cases

4.3.14. The administrator logs into the admin system

- The user requests the administrator login screen (a URL).
- The user enters the administrator password.
- The user selects “submit” and a URL request (“log in”) is sent.
- A “login administrator” request is sent to the Authoring Controller.
- The Authoring Controller sends a “check user” request to the ASPIRE-Tutor.
- The Authoring Controller responds accordingly:
 - If the login was successful, the session is set. The browser is redirected to the administrator page.
 - If the login was unsuccessful, the browser is returned to the login page with an appropriate message.

4.3.15 The administrator logs out of ASPIRE-Author

- The administrator requests to log out of the authoring system.
- A “log out” request is received by the Authoring Controller.
- The user session is deleted.
- The “login administrator” page is displayed.

4.3.16. The administrator logs a user or group of users out of the system

- The administrator requests the “View users” screen.
- The Authoring Controller receives the request.
- The Authoring Controller sends a “get logged-in users list” request to the Authoring Controller

- The list of logged-in users is returned and displayed with options to log each user out.
- The administrator selects a user to log out.
- The session of the selected user is deleted.
- The updated list of logged-in users is displayed.

4.3.17. The administrator views help

- The administrator requests help.
- A “view help” request is sent to the Authoring Controller.
- The Authoring Controller returns the information to the interface.
- The browser displays the help information.

4.3.18. The administrator views the ASPIRE-Author statistics

- The administrator requests the ‘view system statistics’ screen.
- For each general system statistic, the Authoring Controller is queried.
- The Authoring Controller determines where to look for the information, and calculates the required values.
- The values are interpreted according to the type of statistic, and a page is displayed representing these values.

4.3.19. The administrator resets ASPIRE-Author

- The administrator requests a system-reset.
- All sessions are nulled.
- A page confirming the system-reset is displayed.

4.3.20. The administrator views ASPIRE-Author users

- The administrator requests the “View users” screen.
- The Authoring Controller receives the request.
- The Authoring Controller sends a “get logged-in users list” request to the ASPIRE-Tutor
- The list of logged-in users is returned and displayed.

4.3.21. The administrator creates ASPIRE-Author users

- The administrator requests the create user screen.
- The administrator enters a username for the user, an email address, and selects a role.
- A “create user” request is sent to the Authoring Controller, which passes it to the ASPIRE-Tutor
- The request result (success or failure) is returned to the Authoring Controller.
- If the creation was successful, the “User Created” confirmation page is displayed; otherwise an appropriate failure message is displayed.

4.3.22. The administrator removes a domain from ASPIRE-Author

- The administrator requests the “remove domain” screen.
- The list of currently installed domains is returned to the interface, with options to delete.
- The administrator selects a domain to delete.
- The request is sent to the Authoring Controller.
- The Authoring Controller removes all domain files.
- A success page is returned.

4.4. Teacher User Stories

Figure 12 illustrates the use cases for a Teacher interacting with ASPIRE-Tutor. Each use case is then described via a “user story”.

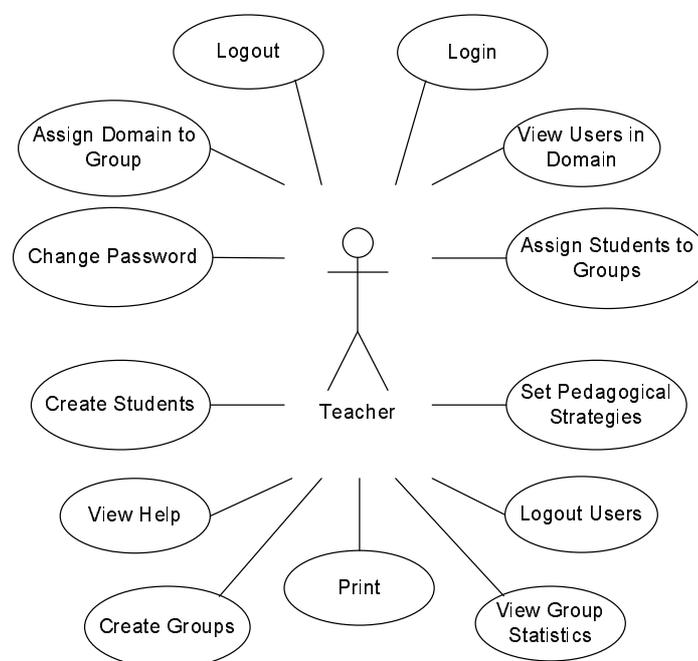


Figure 12. The Teacher Use Cases

4.4.1. The teacher logs into the system

- The user requests the teacher login screen (a URL).
- The user enters a username and password.
- The user selects “submit” and a URL request (“log in”) is sent.
- The request is received by AllegroServe, initiating a Web action in the Session Manager. The Session Manager checks for an existing session (but fails to find one).
- The Session Manager sends a “check user” request to the Pedagogical Module, passing the teacher’s username and password.
- The Pedagogical Module sends a “check user” request to the User Manager.

- The User Manager checks the username/password and role and responds accordingly.
 - If the login was successful, the browser is redirected to the administration page of the current teacher.
 - If the login was unsuccessful, the browser is returned to the login page with an appropriate message.
- Once successfully logged in, the teacher is presented with the teacher administration page, which contains a list of domains he/she is registered for.

4.4.2. The teacher logs out of the system

- The teacher requests to log out of the tutoring system.
- The Session Manager retrieves the session context, and sends a “log out”, request to the Pedagogical Module.
- The Pedagogical Module sends a “log out” request to the User Manager.
- The Session Manager deletes the current teacher session once the Pedagogical Module returns “success” status.
- The “login” page is displayed.

4.4.3. The teacher creates a student

- The teacher requests the create student screen.
- The teacher enters a username for the student, and an email address.
- A “create student” request is sent to the Session Manager.
- The Session Manager retrieves the user’s session state from the Web Session, and sends a “create student” request to the Pedagogical Module.
- The Pedagogical Module sends a “create user” request to the User Manager, with “student” as the role.
- The User Manager checks to see whether another user exists with the same username.
 - If no such name exists the user profile is created on the system, and an email with a randomly generated password is sent to the user. A “success” message is returned to the Pedagogical Module.
 - If it does exist, a “failed: user exists” message is returned to the Pedagogical Module.
- The request result is returned to the Session Manager.
- If the creation was successful, the “User Created” confirmation page is displayed, otherwise an appropriate failure message is displayed.

4.4.4. The teacher creates groups

- The teacher requests the create group screen.
- The teacher enters a name for the group.
- A “create group” request is sent to the Session Manager.
- The Session Manager retrieves the user’s session state from the Web Session, and sends a “create group” request to the Pedagogical Module.
- The Pedagogical Module sends a “create group” request to the User Manager.
- The User Manager checks to see whether another group exists with the same name.
 - If no such name exists the group is registered on the system. A “success” message is returned to the Pedagogical Module.

- If it does exist, a “failed: group exists” message is returned to the Pedagogical Module.
- The request result is returned to the Session Manager.
- If the creation was successful, the “Group Created” confirmation page is displayed, otherwise an appropriate failure message is displayed.

4.4.5. The teacher views statistics on users

- The teacher requests the “view statistics” screen.
- The “view statistics” screen is displayed, with selections for different types of statistics:
 - Statistics on individual groups.
 - Statistics on individual users.
 - Statistics across multiple groups.
- The teacher submits a request for a statistic.
- The Session Manager sends a “get statistic” request to the Pedagogical Module.
- The Pedagogical Module determines where to retrieve the information from, and calculates the required values. The result is returned to the Session Manager.
- The Session Manager interprets the values according to the type of statistic, and returns a page representing these values.

4.4.6. The teacher changes the pedagogical strategies of a group

- The teacher requests the “group-pedagogical strategy administration” screen.
- The Session Manager sends a “get groups for teacher” request to the Pedagogical Module.
- The Pedagogical Module sends the “get groups for teacher” request to the User Manager.
- The list of groups assigned to the given teacher is returned to the Session Manager, which is returned as a page, with options to select groups.
- The teacher selects a group.
- The Session Manager sends a “get all pedagogical strategies” request to the Pedagogical Module.
- The Pedagogical Module sends a “get all pedagogical strategies” request to the Domain Manager.
- The list of all possible pedagogical strategies gets returned to the Session Manager.
- The Session Manager returns a web page with the types of pedagogical strategies that are possible (e.g. Feedback), and what strategy is currently assigned to each type for the group.
- To change the strategy of a single type in the group:
 - The teacher selects the type of strategy.
 - A list of all possible strategies for that type is shown.
 - The teacher selects the strategy he/she would like to set for this type.
 - A “set strategy type to group” request is sent to the Pedagogical Module.
 - The Pedagogical Module sends a “set strategy type to group” request to the User Manager.
 - The User Manager assigns the strategy to the group for the current type, and returns “Success”.
- An updated view of the strategy list for this group is displayed.

4.4.7. The teacher assigns domains to a group

- The teacher requests the “group-domain administration” screen.
- The Session Manager sends a “get groups for teacher” request to the Pedagogical Module.
- The Pedagogical Module sends the “get groups for teacher” request to the User Manager.

- The list of groups assigned to the given teacher is returned to the Session Manager, which is in turn returned as a page, with options to select groups.
- The teacher selects a group.
- The Session Manager sends a “get all system domains” request to the Pedagogical Module.
- The Pedagogical Module sends a “get all system domains” request to the Domain Manager.
- The list of all system domains is returned to the Session Manager.
- The Session Manager returns a web page with the domains that are assigned to each group, and also a list of all possible domains, with options to add and delete domains from the current group.
- For the addition of a domain to the group:
 - The teacher submits the request.
 - An “add domain to group” request is sent to the Pedagogical Module.
 - The Pedagogical Module sends an “add domain to group” request to the User Manager.
 - The User Manager assigns the domain to the group, and returns “Success”.
- For the deletion of a domain from the group:
 - The teacher submits the request.
 - A “delete domain from group” request is sent to the Pedagogical Module.
 - The Pedagogical Module sends a “delete domain from group” request to the User Manager.
 - The User Manager removes the domain from the group, and returns “Success”.
- An updated view of the domain list for this group is displayed.

4.4.8. The teacher logs out student(s)

- The teacher requests the “View students” screen.
- The Session Manager sends a “get logged-in student list” request to the Pedagogical Module.
- The Pedagogical Module requests all currently logged-in students from the User Manager.
- The list of logged-in students is displayed, with options to log each student out.
- The teacher selects a student to log out.
- A “logout other user” request is sent to the Session Manager.
- The Session Manager retrieves the session context of the student requested for logout, and sends a “logout” request to the Pedagogical Module.
- The Pedagogical Module sends an “update student model” request to the Student Modeller, passing the last state of the selected student session, this may include:
 - The current problem and its state; whether new, attempted or finished.
 - The current attempt.
 - The level of the last feedback.
- The Pedagogical Module sends a “log out” request to the User Manager.
- The Session Manager deletes the student session once the Pedagogical Module returns “success” status.
- The updated list of logged-in students is displayed.

4.4.9. The teacher views which users are currently logged in

- The teacher requests the “view students” screen.
- The Session Manager sends a “get logged-in student list” request to the Pedagogical Module.
- The Pedagogical Module requests all currently logged-in students from the User Manager.
- The list of logged-in students gets displayed.

4.4.10. The teacher changes his/her password

- The teacher requests the “change password” screen.
- The teacher enters his/her new password in two separate fields.
- The “change password” request moves through the Session Manager, the Pedagogical Module, and the User Manager.
- The User Manager checks to see if the two entered passwords are the same.
 - If they are the same, the password is changed, and a “success” message is returned.
 - If they are not the same, a “failed: password mismatch” message is returned.
- The request result is returned to the Session Manager.
- The appropriate result page is displayed.

4.4.11. The teacher manages groups

- The teacher requests the “group-student administration” screen.
- A “get all assigned domains” request is sent to the Pedagogical Module.
- The Pedagogical Module requests the list of groups assigned to the teacher from the User Manager.
- A page is displayed this information, and all groups selectable.
- The teacher selects a group that he/she wants to add students to.
- A “get users for domain” request is sent to the Pedagogical Module.
- The Pedagogical Module requests the list of all users assigned to the selected domain from the User Manager. This list is returned to the Session Manager.
- A “get all users” request is sent to the Pedagogical Module.
- The Pedagogical Module requests a list of all users assigned to the system. This list is returned to the Session Manager.
- A page is displayed that contains a list of students currently in the group, and a separate list showing all students on the system not in this group, with options to add or delete students.
- For an addition to the group:
 - The teacher submits the request.
 - An “add student” request is sent to the Pedagogical Module.
 - The Pedagogical Module sends an “add student” request to the User Manager.
 - The User Manager assigns the student to the group, and returns “Success”.
- For a deletion from the group:
 - The teacher submits the request.
 - A “delete student” request is sent to the Pedagogical Module.
 - The Pedagogical Module sends an “add student” request to the User Manager.
 - The User Manager removes the student from the group, and returns “Success”.
- An updated view of the group and system user list is displayed.

4.4.12. The teacher prints the screen

- The teacher chooses to print the current screen, which will be the statistics of a student or group of students.
- The Session Manager receives a “print” request and the parameters on which statistics to print from the Web browser.
- The “print preview” page is displayed followed by a print dialog to set the printer and related options.

5. Knowledge Representation Language

A model of an instructional domain consists of the domain's ontology, a set of constraints and macros and a set of problems with their solutions. Constraints and macros are necessary for an ITS to be able to analyze students' solutions and offer feedback on them. As discussed above, an ITS contains a diagnostic module, which matches the student's solution to the ideal solution using constraints. In order to do this, it is also necessary to parse the problems, solutions and constraints into a form suitable for the diagnostic module.

WETAS uses a proprietary reasoning engine, which contains a bespoke pattern matcher for applying constraints to solutions. In ASPIRE, we will modify and extend this reasoning engine in the following manner:

- Pattern matching will be extended to allow *any* object to be matched (currently this is restricted to the ideal and student solutions only). This allows matches to be recursively nested (i.e. the result of a pattern match may itself be subjected to a further pattern match);
- Macros can accept variables that contain lists ("multipart" variables), such as named wildcards;
- Macros may be recursive, i.e. a macro may call itself.

The problem/solution specification structure is also currently ad hoc. For example, a problem may contain sub-problems, which are currently supported by structuring the ideal solution (to a maximum of one level). In ASPIRE we will allow entire problems to be nested to an arbitrary number of levels. The ontology language is still being defined, and forms part of the authoring system. The constraint and problem language representations are now described.

5.1. Problem specification

To define a problem, it is necessary to specify the following elements:

- A problem header, containing:
 - Problem number
 - Short description
 - Difficulty (optional)
 - Problem text (human readable)
 - Problem statement (machine readable - optional)
- The problem body: this is a list, where each item is either a component of the solution, or a nested problem. Each component contains:
 - Component name
 - Solution (machine readable)
 - Solution text (human readable – optional)
 - Default input (optional)

The following example is a problem from SQL-TUTOR, which does not require any nesting:

```
(169           ; problem number
"group names"  ; short description
1             ; difficulty
"Show the names of all groups, in descending order." ; problem text
NIL          ; machine readable problem text: not required
```

```

; Problem body - a list of components and their values
(("SELECT"      "distinct group_name")
 ("FROM"       "in_group")
  "WHERE"      "" )
 "GROUP-BY"   "" )
 "HAVING"     "" )
 "ORDER-BY"   "group_name desc"))

```

The following is an example of a problem with nesting:

```

(1 ; problem number
 "problem 1" ; short description
 5 ; difficulty
 NIL ; problem text (no text in this case)
 NIL ; machine readable problem statement not needed

; Now the problem body - consists of three nested problems
((1 ; sub-problem number
  NIL ; short name
  NIL ; Difficulty
  "long street" ; problem text
  NIL ; machine-readable problem text
   ("ANSWER" "road" NIL "ro")
  (2 NIL NIL "exciting journey" NIL
   ("ANSWER" "adventure" NIL NIL))
  (3 NIL NIL "stop for a while" NIL
   ("ANSWER" "rest" NIL NIL))))

```

In this example, problem 1 is composed of three sub-problems, which appear in the problem body. The student interface will present the problem accordingly, based on the structure of the problem body. Each sub-problem could potentially consist of further sub-problems. Sub-problems and solution components can also be mixed, allowing any arbitrary problem structure to be represented. Note that the author will enter problems via a GUI interface, rather than typed in as text. Hence much of the detail above will be hidden from the author.

5.2. The constraint language

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

```

(34
 "If there is an ANY or ALL predicate in the WHERE clause, then the
 attribute in question must be of the same type as the only expression of
 the SELECT clause of the subquery."

; relevance condition
(match SS WHERE (?* ?a1
 ("<" ">" "=" "!=" "<>" "<=" ">=")
 ("ANY" "ALL") (" "SELECT" ?a2 "FROM" ?* ")") ?*))

; satisfaction condition

```

```
(and (test SS (^type (?a1 ?type))
      (test SS (^type (?a2 ?type))))
```

```
"WHERE" )
```

This constraint tests that if an attribute is compared to the result of a nested SELECT, the attribute being compared and that which the SELECT returns have the same type (^type is an example of a macro, which are described in section 5.2.4). The constraint language consists of logical connectives (AND, OR and NOT) and three functions: MATCH, TEST, and TEST_SYMBOL. These are now described.

5.2.1. MATCH

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

```
(MATCH <what> <component name> (pattern))
```

<what> is the object to be matched. Normally this will be either the student solution (SS) or the ideal solution (IS). However, it may also be any other variable, such as a part of the result of a previous match. <component name> is the name of the solution component to which the pattern applies; for tests against a variable this is ignored. The notion of components is not domain-dependent; it simply allows the solution to be broken into subsets of the whole solution. (pattern) is a set of terms that match to individual elements in the solution being tested. The following constructs are supported:

- **?*** – **wildcard**: matches zero or more terms that we are not interested in. For example, (MATCH SS WHERE (?* ?a ?*)) matches to any term in the WHERE component of the student solution, because the two wildcards can map to any number of terms before and after ?a, so all possible bindings of this match gives ?a bound to each of the terms in the input;
- **?*var** – **named wildcard**: a wildcard that appears more than once, hence is assigned a variable name to ensure consistency. For example:

```
(AND (MATCH SS SELECT (?*W1 "AS" ?*W2)           (1)
      (MATCH IS SELECT (?*W1 ?N ?*))             (2)
```

First, (1) tests that the SELECT component of the student solution contains the term "AS". Then ?*W1 in (2) tests that the ideal solution also contains all the terms that preceded the "AS", and then maps the variable ?N to whatever comes next. The unnamed wildcard at the end of the second MATCH discards whatever comes after ?N;

- **?var** – **variable**: matches a single term. For example,

```
(MATCH IS SELECT (?what))
```

matches ?what to one and only one item in the SELECT component of the ideal solution.

- **"str"** – **literal string**: matches a single term to a literal value. For example, in

```
(MATCH SS WHERE (?* ">=" ?*))
```

one of the terms in the WHERE component of the ideal solution must match exactly to ">=".

- (lit1 lit2 lit3...) – **literal list**: list of possible allowed values for a single term. For example:

```
(MATCH SS WHERE (?* ?N1 (">=" "<=") ?N2 ?*))
```

assigns the variable ?N1 to any term preceding either a ">=" or a "<=", and ?N2 to the term following it. Note that because ?N1 and ?N2 are not wildcards, they must map to a single term each, hence if the ">=" or "<=" is either at the start or the end of the component this match will fail, because one (or both) of ?N1 and ?N2 will fail to match.

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

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

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

5.2.2. TEST

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

```
(2726
"Check you have used the correct logical connective in WHERE to represent
a range of numbers."

(and (match SS WHERE (?* ?n1 ?op1 ?what1 (("and" "or") ?lc)
                        ?n1 ?op2 ?what2) ?*) (1)
      (match IS WHERE (?* ?n1 "between" ?*)) (2))

(test SS ("and" ?lc)) (3)
"WHERE")
```

Constraint 2726 first tests for an attribute (?n1) in the WHERE component of the student solution that is being compared to two different values (?what1 and ?what2) in (1). Then, (2) looks for the same attribute being used in a BETWEEN construct in the ideal solution. If this is the case, the two tests in the student solution must be ANDed together. The TEST function call in (3) checks that this is the case, by ensuring that the logical connective (represented by the variable ?lc) equates to "and". The syntax of the TEST function is:

```
(TEST <solution name> (test-term))
```

where <solution name> is again IS or SS, and (test-term) is a single value test, such as a test against a literal or list of literals. In the previous example, a single value test is made for the value

"and". In effect, TEST performs the same function as MATCH, but where the pattern contains just a single match term, on a list that contains just the value of the variable in question, in this case ?lc.

5.2.3. TEST_SYMBOL

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

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

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

```
(TEST_SYMBOL <solution> <var> (pattern))
```

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

5.2.4. Removing domain-specific functions: macros

To overcome shallow domain-specific knowledge (such as the set of allowed words in an English tutor), we use macros to represent partial pattern matches that are used often. For example, the macro for ^attribute-of in SQL (which specifies all valid attribute/table combinations) is:

```
(^attribute-of (??a ??t)
  (TEST SS ((("TITLE" "MOVIE") ("LNAME" "DIRECTOR")...)
    (?a1 ?t1)))
```

The syntax of a macro definition is:

```
(<MACRO NAME> (<parameters>) <body>)
```

The name must always begin with a “^” so that macros can be easily identified by the constraint compiler. Similarly, the parameter names are preceded by “??” so that they can be distinguished from local variables in the macro body. The body may be any valid condition including logical connectives, MATCH functions and other macro calls. Consider the following example from SQL:

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

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

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

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

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

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

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

In this case, the *first* parameter to `^attr-name (?n)` is matched to a term in the input string, with `?a` and `?t` being either tested or instantiated by the macro, depending on whether or not they are already bound.

5.2.5. Allowing recursion

In WETAS, when the constraints are compiled the macro calls are recursively removed such that the resulting code contains purely pattern matches with no sub-functions. This prevents recursion, since this will result in an infinite loop of substitutions because the stopping criteria for recursion cannot be tested at compile-time. In ASPIRE macros will be resolved at runtime, allowing recursive calls. For example, the following macro checks for a valid expression, which may consist of either a word or two expressions ANDed together:

```
(^expression (??exp)
(or-p (test ?? (^word ?exp))      ; test for a single word
      ; else test for two expressions ANDed together
      (and (match ??exp NIL (?*e1 "AND" ?*e2))
            (test ?? (^expression ?*e1)
                      (test ?? (^expression ?*e2))))))
```

In the above example, the MATCH statement matches variables `?*e1` and `?*e2` to the entire part of the solution preceding and following the AND, respectively. The two TEST calls then check that each of these is in itself a valid expression. This allows conjoined expressions to be nested to an arbitrary level.

5.3. Further extensions

We will investigate implementing the reasoning engine using an existing third-party engine such as JESS (Friedman-Hill, 2003). JESS supports all of the functionality described, and offers additional functionality, such as:

- The ability to match arbitrary structures in a single pattern match (our pattern matcher assumes the object of any given pattern match is a flat list);
- Arithmetic functions;
- The ability to call external functions.

We will prototype using JESS and test its capabilities. In particular, we will measure its performance. The JESS reasoning language is very similar to the WETAS language. Also, we would produce parsers that convert ASPIRE's constraint representation into rules that JESS could use to test solutions.

6. Session Manager

The Session Manager has several primary functions:

- user authentication;
- user authorization;
- maintaining user sessions, and
- providing monitoring functions.

As discussed earlier, the clients access ASPIRE-Tutor through a Web interface (based on HTTP) or through the XML-RPC interface. Both HTTP and XML-RPC protocols are stateless, i.e. each request between the client and the server is an atomic transaction independent of the previous transactions. However, ASPIRE must maintain information across these atomic transactions in order to provide continuous support. To provide adaptive instruction, the student model needs to be updated after each action, and the Pedagogical Module needs to make pedagogical decisions based on the whole instructional session and the student's progress. Therefore, the server (the *Model* component in MVC terms) is responsible for maintaining the internal state of the application. The Session Manager encapsulates the internal state of the data, providing a unified interface to all types of interfaces that may be used to access the system. Here the term interface has a double meaning. The first meaning is associated with how the system is accessed, i.e. what protocol is used for interaction with the system. The other meaning of interface is associated with different types of users. For example, both students and administrators will be interacting with the same system with its underlying database; however, the view of the system delivered to the user will differ based on the type of the user. The Session Manager will provide a single point of access to the Pedagogical Module.

To provide continuity of user session, the Session Manager will maintain in memory a set of transient session objects for all running session. The Session class was discussed in Section 3.4. For each user logged on to ASPIRE at any particular time, the Session Manager creates a session object, which contains all the necessary information for normal continuation of interaction. For example, to support a student learning with an ITS, the session object must contain the session id, the student's name, information about the current domain and problem, the timestamp of the last student's action, as well as some other specific information related to the student's action/request. The data stored in the session object will be updated on the basis of each request and response. For example, if the student chooses a new problem from the list of unsolved problems, the client request will contain the problem number the student wishes to work on next. In that case, the Session Manager will update the session object by storing the new problem number.

The Session Manager is also responsible for user authentication, which refers to verifying the user's identity. The Session Manager will be responsible for authenticating users based on user names and passwords. User names and hashed passwords will be stored in the database as slots of persistent objects created for each user. The Session Manager will perform authentication by verifying user's credentials. For security reasons, the user database will store hash strings (obtained with one-way hashing algorithm MD5) instead of plain-text user passwords. The Session Manager will compare the hash strings obtained from submitted credentials with stored hash strings. On successful authentication, the Session Manager will create a transient session object for each authenticated user.

Authorisation refers to the user's permission to perform certain operations on data. For example, ASPIRE will be accessible to various types of user, such as students, authors, teachers and server administrators. These types of users will be allowed to perform various actions, described in User Stories Section. Students will have the least privileges, only enabling them to access particular instructional domains (i.e. ITSs). Teachers will be given the privileges allowing them to create new student accounts, view statistics of their classes' performances, etc. Similarly, server administrators will have even higher levels of privileges, allowing them to perform server administration tasks. Based on the user privileges, the interface of the system might have additional navigation features associated with higher levels of privileges. When processing requests requiring higher privileges, Session Manager will verify the privilege status of request sender. The Session Manager will use Secure Socket Layer (SSL) encryption to provide a reliable level of network security.

The Session Manager also provides a set of functions that allows monitoring the workload of the server with varying levels of details. For example, the administrator may want to see the memory status of the server or view the list of individual user sessions. To conserve server resources, Session Manager will be responsible for terminating sessions for inactive users.

7. Conclusions

This report covers the first three milestones of the ASPIRE project. We discussed our previous experience in building constraint-based tutoring systems, as well as the WETAS tutoring shell. We then presented the goal of ASPIRE, its overall architecture, and the architecture of the two servers, ASPIRE-Author and ASPIRE-Tutor. The knowledge representation language was designed. We have also implemented the first module of ASPIRE-Tutor, the Session Manager.

The next two milestones we will work on include one module from each of the servers. We will be implementing the Student Modeller (ASPIRE-Tutor) and the Tutor GUI Designer module from ASPIRE-Author.

8. References

1. Allegro Common Lisp www.franz.com
2. Anderson, J. R., Corbett, A., Koedinger, K. & Pelletier, R. (1996) Cognitive Tutors: Lessons Learned. *Journal of Learning Sciences*, 4 (2), 167-207.
3. Bloom, B. S. (1984) The 2-sigma Problem: the Search for Methods of Group Instruction as Effective as one-to-one Tutoring. *Educational Researcher*, 13, 4-16.
4. Corbett, A. T., Trask, H. J., Scarpinato, K. C. & Hadley, W. S. (1998) A Formative Evaluation of the PACT Algebra II Tutor: Support for Simple Hierarchical Reasoning. In Goettl, B. P., Halff, H. M., Redfield, C. L. and Shute, V. J. (eds.). *Proc. 4th Int. Conf. Intelligent Tutoring Systems*, San Antonio, Texas, pp. 374-383.
5. Friedman-Hill, E. (2003) *Jess in Action: Java Rule-based Systems*. Manning.
6. Gruber, T.R. (1993) A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199-220.
7. Koedinger, K. R., Anderson, J. R., Hadley, W. H. & Mark, M. A. (1997) Intelligent tutoring goes to school in the big city. *Artificial Intelligence in Education*, 8(1), 30-43.
8. Krasner, G.E., Pope, S.T. (1998) A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Programming*, 1(3), 26-49.
9. Martin, B., Mitrovic, A. (2002) Authoring Web-Based Tutoring Systems with WETAS. In: Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson, C-H Lee (eds) *Proc. Int. Conf. Computers in Education ICCE 2002*, pp. 183-187.
10. Martin, B. (2003) *Intelligent Tutoring Systems: The practical implementation of constraint-based modelling*. PhD thesis, University of Canterbury.
11. Martin, B., Mitrovic, A. (2002) Authoring Web-Based Tutoring Systems with WETAS. Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson, C-H Lee (eds) *Proc. Int. Conf. Computers in Education ICCE 2002*, Auckland, 183-187.
12. Martin, B., Mitrovic, A. (2003) Domain Modelling: Art or Science? In: U. Hoppe, F. Verdejo & J. Kay (ed) *Proc. 11th Int. Conf. Artificial Intelligence in Education*, IOS Press, pp. 183-190.
13. Mayo, M., Mitrovic, A. (2000) Using a probabilistic student model to control problem difficulty. *Proc. Int. Conf. Intelligent Tutoring Systems ITS'2000*, G. Gauthier, C. Frasson and K. VanLehn (eds), Springer, pp. 524-533.
14. Mayo, M., Mitrovic, A. (2001) Optimising ITS Behaviour with Bayesian Networks and Decision Theory'. *Int. Journal on Artificial Intelligence in Education*, 12(2), 124-153.
15. Mitrovic, A. (1998a) Learning SQL with a Computerized Tutor. *29th ACM SIGCSE Technical Symposium*, pp. 307-311.
16. Mitrovic, A. (1998b). A Knowledge-Based Teaching System for SQL. In: T. Ottmann, I. Tomek (eds.), *Proc. ED-MEDIA'98*, pp. 1027-1032.
17. Mitrovic, A. (1998c) Experiences in Implementing Constraint-Based Modelling in SQL-Tutor. In: B. Goettl, H. Halff, C. Redfield, V. Shute (eds.), *Proc. Int. Conf. Intelligent Tutoring Systems ITS'98*, pp. 414-423.
18. Mitrovic, A. (2002) NORMIT, a Web-enabled tutor for database normalization. Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson, C-H Lee (eds) *Proc. Int. Conf. Computers in Education ICCE 2002*, Auckland, pp. 1276-1280.
19. Mitrovic, A. (2003) Supporting Self-Explanation in a Data Normalization Tutor. In: V. Alevan, U. Hoppe, J. Kay, R. Mizoguchi, H. Pain, F. Verdejo, K. Yacef (eds) *Supplementary proceedings AIED 2003*, pp. 565-577.
20. Mitrovic, A., Devdizic, V. (2004) A Model of Multitutor Ontology-based Learning Environments. *Int. J. Continuing Engineering Education and Life-Long Learning*, 14(3), 229-245.
21. Mitrovic, A., Koedinger, K., Martin, B. (2003) A Comparative Analysis of Cognitive Tutoring and Constraint-Based Modelling. P. Brusilovsky, A. Corbett, F. de Rosis (Eds.) *Proc. 9th Int. Conf. User Modelling UM 2003*, Springer-Verlag, LNAI 2702, pp. 313-322.
22. Mitrovic, A., Martin, B., Mayo, M. (2002) Using Evaluation to Shape ITS Design: Results and Experiences with SQL-Tutor. *Int. J. User Modelling and User-Adapted Interaction*, 12(2-3), 243-279.
23. Mitrovic, A., Ohlsson, S. (1999) Evaluation of a Constraint-Based Tutor for a Database Language. *Int. J. Artificial Intelligence in Education*, 10(3-4), 238-256.
24. Mitrovic, A., Suraweera, P., Martin, B., Weerasinghe, A. (2004) DB-suite: Experiences with Three Intelligent, Web-based Database Tutors. *Journal of Interactive Learning Research*, 15(4), 409-432.
25. Model-View-Controller pattern <http://java.sun.com/blueprints/patterns/MVC-detailed.html>
26. Ohlsson, S. (1994) Constraint-based Student Modelling. In *Proc. of Student Modelling: the Key to Individualized Knowledge-based Instruction*, Springer-Verlag, Berlin, pp. 167-189.

27. Ohlsson, S. (1996) Learning from Performance Errors. *Psychological Review*, 103, pp. 241-262.
28. Suraweera, P., Mitrovic, A. (2001) Designing an Intelligent Tutoring System for Database Modelling. In: Smith, M. J. and Salvendy, G. (eds.). *Proc. of 9th Int. Conf. Human-Computer Interaction HCII 2001*, New Orleans, vol. 2, pp. 745-749.
29. Suraweera, P. & Mitrovic, A. (2002) KERMIT: a Constraint-based Tutor for Database Modelling. In: S. Cerri, G. Gouarderes and F. Paraguacu (eds.) *Proc. 6th Int. Conf on Intelligent Tutoring Systems ITS 2002*, Biarritz, France, LCNS 2363, pp. 377-387.
30. Suraweera, P., Mitrovic, A. (2004) An Intelligent Tutoring System for Entity Relationship Modelling. *Int. J. Artificial Intelligence in Education*, 14(3-4), 375-417.
31. Wang, T., Mitrovic, A. (2002) Using neural networks to predict student's behaviour. In: Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson and C-H Lee (eds.) *Proc. Int. Conf, Computers in Education ICCE 2002*, Los Alamitos, CA: IEEE Computer Society, pp. 969-973.
32. Weerasinghe, A., Mitrovic (2005) Facilitating Deep Learning through Self-Explanation in an Open-ended Domain. *Int. J. of Knowledge-based and Intelligent Engineering Systems*, 9, 1-17.
33. Zakharov, K., Mitrovic, A., Ohlsson, S. (2005) Feedback Micro-engineering in EER-Tutor. In: C-K Looi, G. McCalla, B. Bredeweg, J. Breuker (eds) *Proc. Artificial Intelligence in Education AIED 2005*, IOS Press, pp. 718-725.