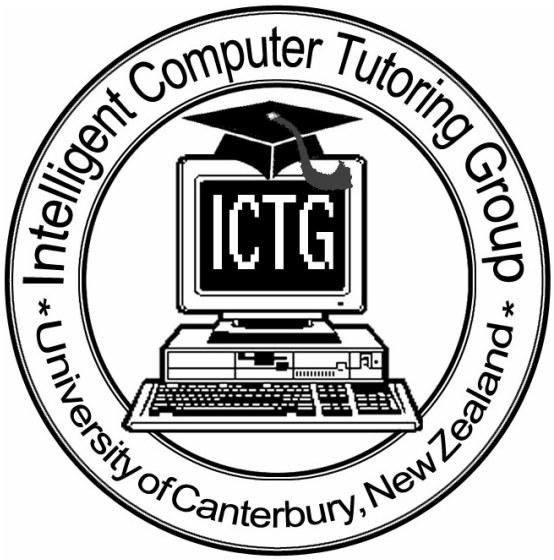
ASPIRE: Student Modelling and Domain Specification

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

Technical Report TR-08/05, 8 December 2005
Intelligent Computer Tutoring Group
Department of Computer Science and Software Engineering
University of Canterbury



Content

1. Introduction

This document reports the work done from 1.9.2005 to 30.11.2005 on 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 agents for use in e-learning courses. ASPIRE will support the process of developing intelligent educational systems by automating the tasks involved, thus making it possible for tertiary teachers with little computer background to develop systems for their courses. The resulting educational systems 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 the first report on the ASPIRE project (Mitrovic et al., 2005), we presented the background for the project, functional specifications and the overall architecture of ASPIRE. ASPIRE consists of ASPIRE-Author, the authoring server, and ASPIRE-Tutor, the tutoring server which delivers the resulting intelligent educational systems to students. The first report also discussed the functionality of the system in terms of user stories, the knowledge representation language used for developing domain models, and finally presented the Session Manager, the first component of ASPIRE to be developed.

This second report focuses on the work performed on implementing components of both ASPIRE-Author and ASPIRE-Tutor, and builds upon the information presented in the first report. Section 2 discusses the newly developed components of ASPIRE-Author. We start with the authoring process, and then in Section 2.2 describe the implementation of the general framework necessary for authoring, on which the further development is based. We present the package structure and discuss the classes developed. Next, Section 2.3 describes the Domain Structure component of the authoring interface, which allows the author to specify some features of the chosen instructional domain. Section 2.4. discusses the interface component for specifying the structure of problems and solutions. These specifications are stored as a part of the domain model, and later used for constraint induction. They are also necessary in order for the author to develop the student interface, and Section 2.5 describes the Student Interface Designer.

Section 3 follows the same structure, starting with the description of the implemented framework, classes and packages of ASPIRE-Tutor. Then we describe the Diagnostic Module in Section 3.2, and the Student Modeller in Section 3.3.

In order to provide persistence of student data, it was necessary to design and implement logging procedures. ASPIRE is being implemented in Allegro Common Lisp (Steele, 1990), an object-oriented language which supports the CLOS standard[1]. For that reason, we decided to use AllegroCache[2] (Aasman, 2005), an object-oriented database management system which is a component of the ACL IDE[3]. Section 4 discusses the work performed with AllegroCache. Finally, conclusions are presented in Section 5.

2. ASPIRE-Author

ASPIRE-Author supports authors in developing new intelligent educational systems. The main task in authoring is the development of the domain model. Manual development of a domain model is a time-consuming and difficult task, as domain models are typically large, and require careful explicit representation of the domain knowledge. For that reason, ASPIRE-Author supports the author by substituting the programming task with a much simpler task of providing examples of problems and solutions, from which the system induces the necessary knowledge elements.

[1] http://www.lisp.org/table/references.htm
[2] http://www.franz.com/products/allegrocache/index.lhtml
[3] http://www.franz.com/

This section describes the work done on ASPIRE-Author in the second reporting period. We start by describing the authoring process in general, and then describe the implementation of the framework for ASPIRE-Author. This work was necessary to provide an environment for further development of the components planned for this reporting period. We then present the components allowing the author to specify domain characteristics, problem and solution structures, followed by a description of the Student Interface Builder.

2.1. General Authoring Process

As described in the previous report (Mitrovic et al, 2005), authoring in ASPIRE-Author is a semi-automated process carried out with the assistance of an author (i.e. domain expert). The authoring process presented previously was extended to account for the addition of several instructional domains by the same author. The modified process includes an additional initial step where the author adds a new domain and specifies its general characteristics, such as its name and description, as well as any necessary subdomains. This step serves as the starting point for developing the domain model.

Once the instructional domain is specified, the author develops its ontology, which is a hierarchy of the domain concepts ordered by the 'is-a' relationship (The 'is-a' relationship is the association between a subclass and its parent; super-class). Each domain concept may additionally be associated with other domain concepts by various author-specified relationships. The developed ontology will later be used for constraint generation. After specifying the ontology, the author decides on the problem and solution structures, and provides typical problems with 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.

Consequently, the process of generating a domain model for a domain using ASPIRE-Author involves seven steps:

1. *Modelling the domain structure;* the author specifies the structure of the domain. This includes specifying the name of the chosen instructional domain, a short description, and whether the domain is procedural or not. For procedural domains, the author specifies the steps to be performed by the student. The author also specifies the subdomains (if any exist).
2. *Composing an ontology of the domain;* during the ontology composition stage, the author models the domain as a hierarchy of concepts using the Ontology Workspace.
3. *Modelling the problem and solution structures;* the author specifies the structure of problems and solutions to problems in the chosen instructional domain.
4. *Designing the student interface;* during this step, the author specifies the interface for communicating with students.
5. *Adding problems and solutions;* the author is requested to enter examples of problems and solutions during this stage. The Problem/Solution Editor will be developed according to the specified structures of problems and solutions and the interface characteristics specified during the previous step.
6. *Generating constraints (syntax and semantic);* this step involves the generation of both syntax and semantic constraints. The Syntax Constraints Generator analyses the domain ontology and induces syntax constraints directly from it, without any further assistance of the author. The Semantic Constraints Generator analyses both the ontology and the problems and solutions provided by the author to generate semantic constraints.
7. *Validating the generated constraints;* the constraint validation stage involves automatic validation of the constraints 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.

The reporting period covered by this document included work on steps 1, 3 and 4. Section 2.2 describes the implementation of the general framework which was necessary in order to work on ASPIRE-Author. Then, the following sections describe the components supporting these three authoring steps.

2.2. Implementation of the Framework

Figure 1 illustrates the architecture of ASPIRE-Author, which is a slightly updated version of the architecture developed in the first reporting period. As shown in the diagram, the Web Interface module (i.e. the authoring interface) consists of six components to cater for each knowledge authoring step outlined in Section 2.1 (note that generation of constraints is handled by other components of the system). The three components (Domain Structure Modeller, Problem/Solution Structure Modeller and the Student Interface Designer) relating to authoring steps 1, 3 and 4 have been completed during this reporting period.

The Authoring Controller module is the driving engine of the ASPIRE-Author. It acts as a mediator between the interface layer and the data layer. The controller receives requests from the interface layer, which are passed on to the relevant module. The controller is also responsible for accessing objects from the database. The controller functions necessary for the implemented interface modules have been completed.

The Domain Model Manager consists of all the domain model classes for representing domain models. In this period, we have completed the domain model classes necessary for representing domain details, problem/solution structures and ontologies. As the Ontology Workspace is planned to be developed in the next period, currently we have only developed the classes necessary to store the ontology. This was necessary as the problem and solution structure depend on concepts of the ontology.

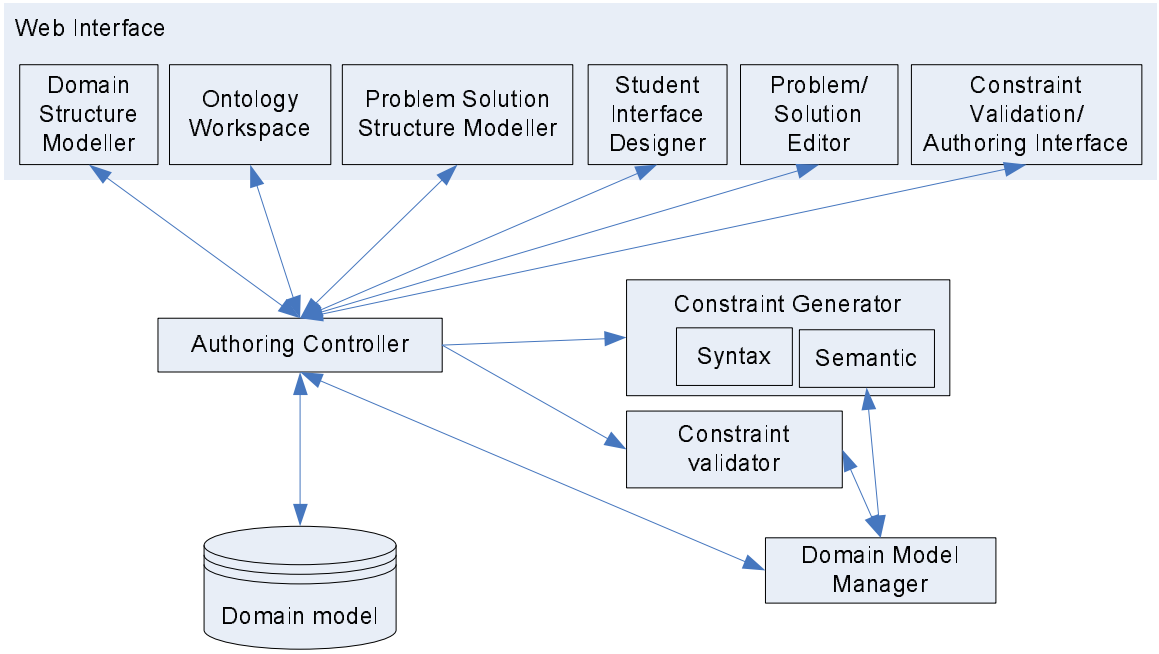


Figure 1. The architecture of ASPIRE-Author

2.2.1. Packages structure

The package structure of ASPIRE-Author (shown in Figure 2) was implemented to reflect its architecture. Each module is implemented as a separate package. The framework of the system had to be developed in order for the individual modules to be able to communicate with each other. The packages such as the constraint-generator are currently empty and will be completed in future milestones.

The web interface is handled by the `web` package. The base web package is responsible for initialising and starting up the web server. It consists of two sub packages: `web.access` and `web.interface`. `web.access` contains all web access functions for handing web requests from the user. The `web.interface` package consists of all clp functions, which produce the dynamic content of HTML pages (http://opensource.franz.com/aserve/aserve-dist/webactions/doc/webactions.html). Currently the base web package is complete. The functions necessary for implementing the three interface components scheduled for this reporting period have been implemented.

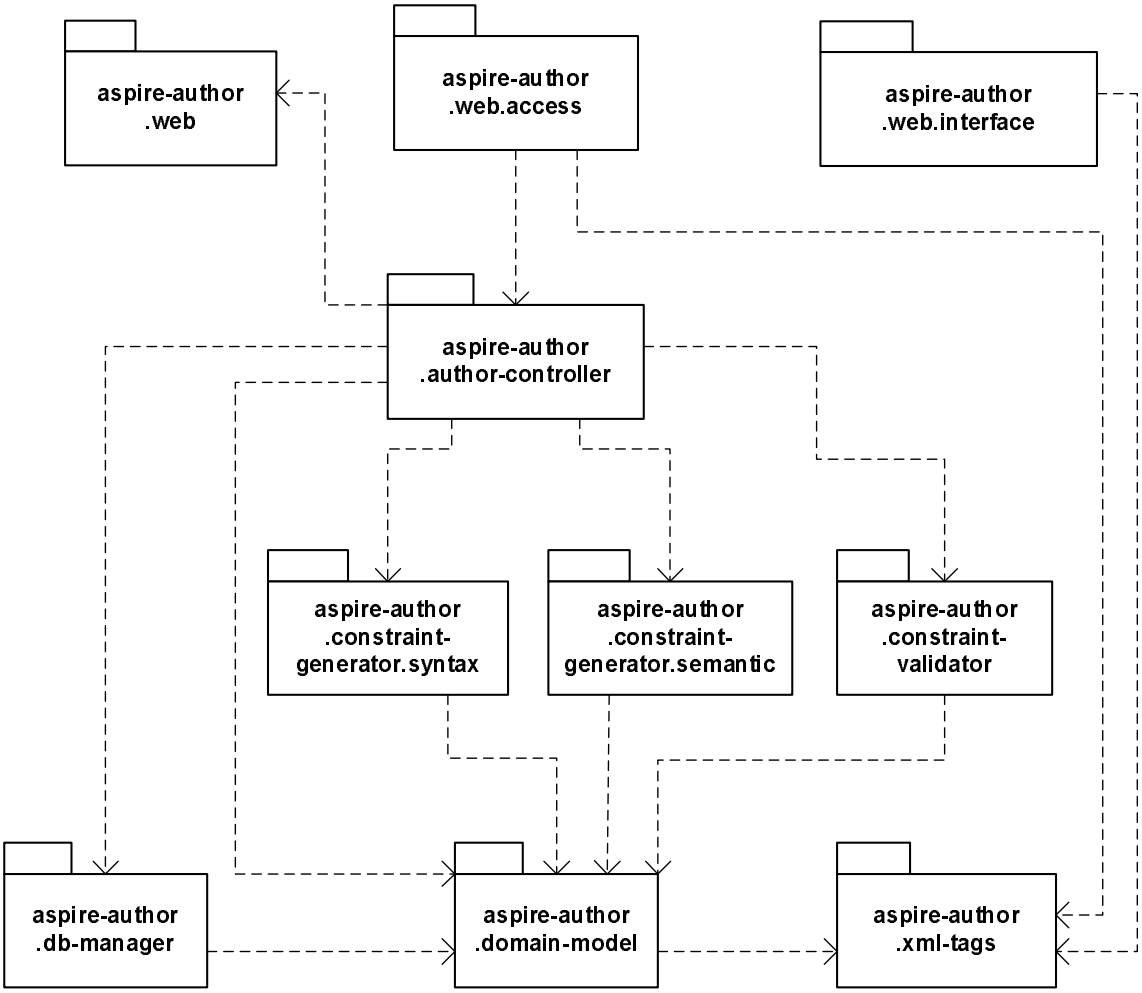


Figure 2. The package structure of ASPIRE-Author

The `controller` package acts as a middle layer between the data layer and the interface layer. It provides the only access point to the `web.access` package. The controller package is responsible for starting up ASPIRE-Author server, communicating with the `db-manager` for

accessing the required components from the database and issuing commands for other packages. The controller package also communicates with the domain model for modifying and retrieving domain model components. Currently the controller package contains the functions essential for the implemented interface components to communicate with the domain model package.

The data classes for storing the domain model are contained in the `domain-model` package (see Section 2.2.2). The implementation of classes required for representing domain details and problem solution structure is complete. They are also capable of producing XML representations of themselves and of updating themselves according to an XML representation of a domain model.

The `xml-tags` package consists of all constants, especially tags used for producing XML documents. The xml-tags package is used by the domain-model, and the web package for assembling and decrypting XML representations of the domain model. The controller also uses the xml-tags package for accessing relevant constants.

The ASPIRE-Author project is implemented according to the directory structure outlined in Figure 3. The project consists of seven top level directories for grouping various files. The db directory holds the physical database file. The compiled executable version of the system would reside in the `dist` directory. All user and development documentation resides within the `doc` directory. The `lib` directory contains any extra libraries that the system would use. Any other resources reside within the `resources` directory. The directories under `src`, which hold Lisp source code files, reflect the system's package structure. Each source file is stored under the appropriate directory. All HTML pages including their resources such as style sheets, images and scripts are stored under the `web` directory.

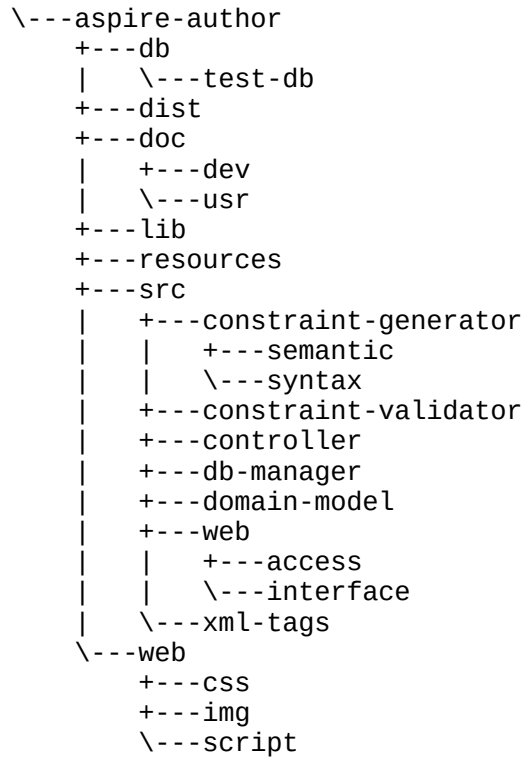


Figure 3. The directory structure of ASPIRE-Author

2.2.2. Domain Model Classes

In order to be able to work on the milestones for this period, we needed to define the classes belonging to the domain model. Figure 4 shows the UML class diagram for the domain model. For simplicity, the class diagram only focuses on the implemented classes that are directly related to achieving the milestones in this reporting period. Please note that ASPIRE-Author and ASPIRE-Tutor share domain models, and therefore this work is relevant for both sides of ASPIRE.

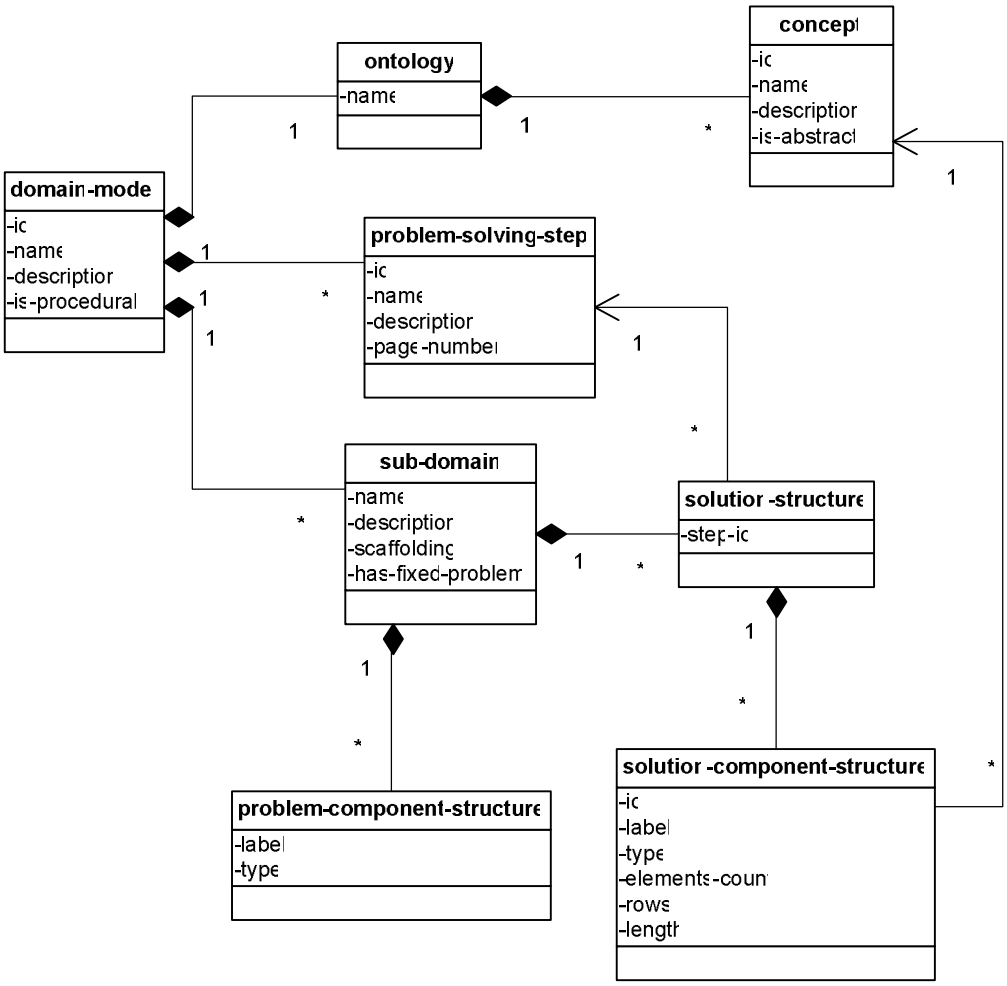


Figure 4. The domain model classes[4]

As discussed in the previous report (Mitrovic et al, 2005), a domain model consists of all knowledge necessary to analyse students' solutions to problems in a particular instructional domain. Domain models are maintained by the Domain Manager. A domain has a unique id and a unique name. The author may also enter a description of the domain; this description is not processed by ASPIRE in any way, and will only be used by humans to familiarise themselves with instructional domains. An instructional domain either contains procedural or declarative (i.e. non-procedural) tasks, as specified by the author (see Section 2.3), and the type of tasks is stored as the `is-procedural` attribute in the corresponding class. For procedural tasks, the author specifies each step in the task. This information is maintained in the `problem-solving-step` class. For

[4] The links between the structures and concrete objects in Figure 4 are *dependency* links -- a dotted line with an open arrowhead that shows one entity (object) depends on the behaviour of another entity (object). Dependencies are typically used to represent that one class instantiates another or that it uses the other class as an input parameter.

each problem-solving step, the author specifies its id (unique within the task), name, description, and the page this step will be served on. Further explanation of procedural task specification is available in Section 2.5.

Each domain has a domain ontology (produced by the author), which is identified by its name, and consists of a number of domain concepts. A domain concept has a unique id, name, description, and an attribute which keeps track of whether it is abstract. Domain concepts are organised into a hierarchy via the 'is-a' relationship; the concepts appearing as leaves in the hierarchy are elements of student solutions, while the concepts on higher levels are abstract concepts, and are used to describe the structure of the domain. Currently, we have only implemented the `ontology` and `concept` classes as they were relevant for the solution structures. The next reporting period involves the implementation of the ontology editor, which would require the remainder of the classes for representing an ontology such as `relationship` and `property` to be implemented (not shown in the class diagram).

Each domain may have a number of subdomains, where a subdomain covers a particular problem set. For example, in the case of SQL-Tutor, the database query language tutor (Mitrovic, 1998; Mitrovic et al., 2002), a subdomain consists of a set of problems using a particular database, such as a MOVIES database. All problems in a subdomain share the same database, and the author can specify the scaffolding information only once. The scaffolding information can be any additional material that the author feels is necessary to aid the students in solving the presented problems, in the SQL-Tutor this is the names and fields of the tables in the database (i.e. the database schema). The constraints describing the SQL concepts would belong to the domain, not to the subdomain, as they do not depend on a particular database. In other domains, however, there might be constraints that are domain specific. For example, in the case of LBITS, an intelligent educational system that teaches English (Martin & Mitrovic, 2003), the various types of problems (such as turning verbs into nouns, or specifying the plural version of a noun given the singular form) will also have specific constraints. In that case, a subdomain will contain a number of problems and also constraints specific to that subdomain. Therefore, each subdomain contains the specification of the problem structure and solution structure. In the case when all the problems are of the same type, there will be a single subdomain representing them.

The problem structure consists of a list of `problem-component-structure` objects, which keep information about the problem's components, such as its name and type. A problem component can be either textual (such as the text of the problem) or graphical (e.g. a diagram accompanying the problem). If the component is textual, then the author would type in the problem statement while adding problems, whereas if it is graphical, the author would have to upload an image.

Each subdomain has its own solution structure. The `solution-structure` objects keep track of the structure of a complete solution for a particular step. Sub-domains of declarative domains have only one solution structure, whereas procedural domains would have a solution structure associated with each problem-solving step. A `solution-structure` consists of a list of `solution-component-structure` objects, which keep track of the structure of a part of the complete solution. Each `solution-component-structure` object has an id, label, type and length and width. It also knows about the type of elements it can hold (i.e. concept from ontology). The types and dimensions of widgets that form the student problem solving interface depends on properties of the `solution-component-structure` object, as discussed in Section 2.5.

Problems and solutions for a subdomain are based on these problem/solution structures. The number of components for a problem, the number of solutions for each component and the ordering in which they appear are governed by these specifications.

2.3. Modelling the domain structure

The Domain Structure component of the authoring interface, illustrated in Figure 5, allows the author to specify the general characteristics of the chosen instructional domain. The author enters the domain's name and a short description of the domain in the textboxes provided. The author also chooses whether the domain contains non-procedural (i.e. declarative) or procedural tasks. By default, a domain is non-procedural, as is the case in Figure 5.

Domains can be divided into subdomains (referred to as problem sets) to group similar problems. Each domain must have at least a single problem set, described by its name, description and any scaffolding information. Problem sets can be added and removed by clicking the + and – buttons. Figure 5 illustrates the domain structure for a tutoring system that teaches SQL, the popular database query language. The final tutoring system, named SQL-Tutor, would present problems to students to be solved in a non-procedural manner. SQL-Tutor would consist of two sets of problems: `movies` and `company`.

Once the author is satisfied with the modelled domain structure, it can be saved by pressing on the `Save structure` button. A successful save action presents the author with the `Ontology Editor` interface for modelling an ontology of the domain, which is the next step in the authoring process. The author may also return to the `Domain structure` page at any time to make any necessary changes or to add new problem sets.

If the author specifies that the domain is procedural, the authoring interface allows the author to specify the problem-solving steps. In the case illustrated in Figure 6, the author is developing a physics tutor, and has specified four problem-solving steps. The four steps are: choosing the appropriate equation, outlining the known variables, composing the chosen equation with the known variables substituted and finally solving the equation for the unknown variable. The author needs to decide whether the solutions for all steps are composed in one web page or whether it is spread out to a set of web pages. In the case of APT, each problem solving step would be presented to the student on a separate page, where the student has to correctly complete the current step they are working on, in order to move on to the next step. Once the student has correctly completed a step, the interface for composing the solution for the next step is presented along with the solutions for all previously completed steps.

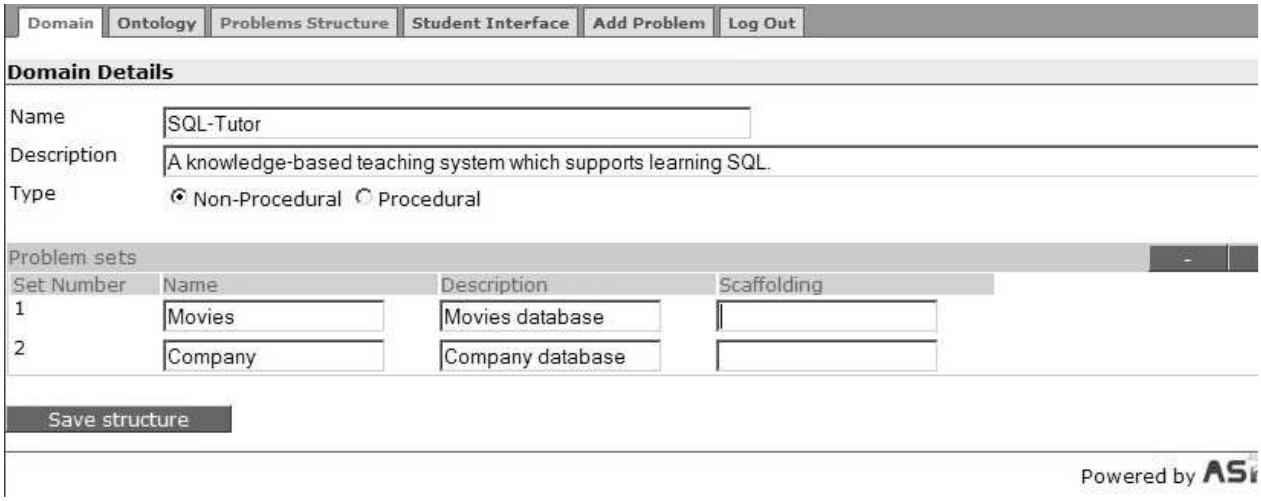


Figure 5. The Domain Structure Modeller

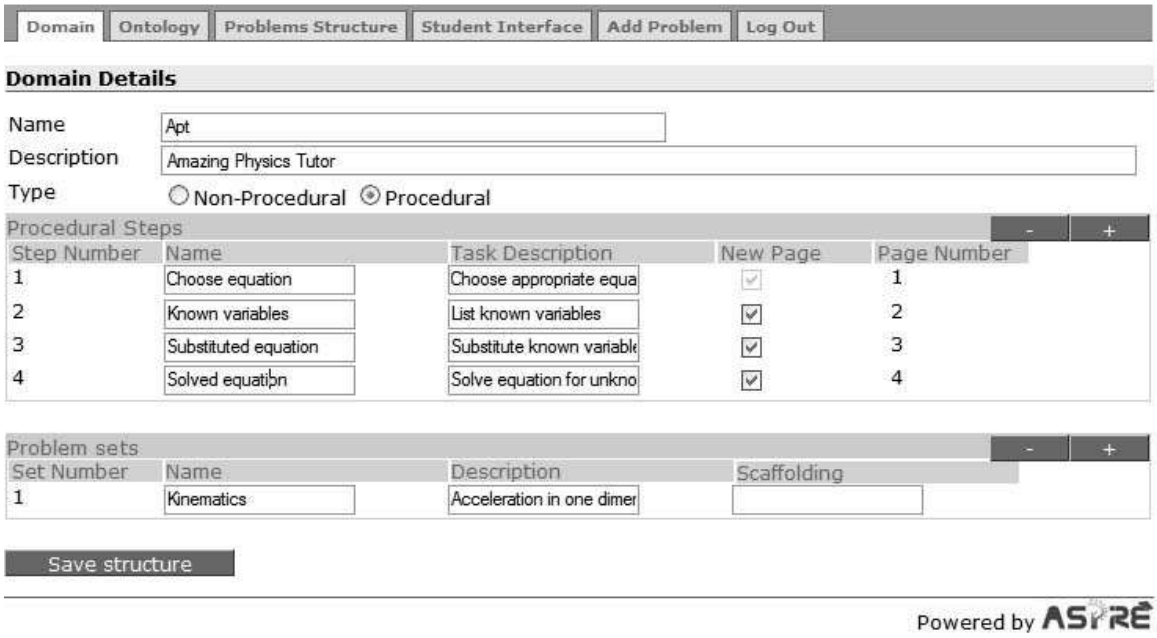


Figure 6. Specifying steps of a procedural task

The author may add as many steps as required by clicking the + button. Clicking the + button results in a new blank row, which can be populated to add one new step. For each step, the author specifies its name and description in the textboxes provided, and also chooses whether to display the step's solution components on the same page as the preceding step or on a new page. The author may also remove any unnecessary steps by clicking on the - button. Each click on the - button removes the last step in the procedural steps list.

2.4. Specifying Problem/Solution Structure

After specifying the domain ontology, the author needs to specify the structure of problems and solutions in the chosen instructional domain during the third step of the domain authoring process. We have completed the interface for modelling the structure of problems and solutions, which enables authors to specify components of problems and model the structure of solutions expected from students. The structure of solutions depends on whether the domain is procedural or declarative. A declarative task requires a single solution that may consist of a number of components. On the other hand, a procedural task requires a solution for each step of the problem solving procedure. As the result, the structure of solutions for each step has to be modelled.

2.4.1. Declarative domains

Figure 7 illustrates the interface for modelling the structure of problems and solutions for a declarative domain. The interface is divided into two separate sections. The top section is for the author to model the structure for problems and the bottom is for modelling the structure of solutions.

In some cases, all problems in a subdomain may have the same general description about what needs to be done. For example, in the language tutor, there is a set of problems dealing with turning verbs into nouns. All problems of this type would have the same task requirement entered just once by the author: "Turn the following verb into a noun", and then each verb would be entered separately as the problem statement. The author needs to specify whether there is such a task requirement for the current subdomain, using the tick box associated with the first element (called `Task requirement`) of the problem structure interface. The author does not need to do

anything about the second element (called `Problem statement`), as it is assumed that every problem will have a specific problem statement. This element is included in the interface to make it obvious that a problem statement will always be a part of the problem specification.

A problem may also contain a collection of sub-components that add detail to the problem statement. The composition of these problem components can be modelled by populating the problem components table, where problem components can be added by clicking on the + button and removed using the – button. Clicking the + button results in a new row, which can be populated to add a new problem component. Problem components are described by their label and type. The label is displayed in the student problem solving interface next to the problem component. Each component can be either textual or graphical. For example, Figure 7 illustrates the problem structure for SQL-Tutor. Each problem in SQL-Tutor contains a description of the task, and no additional components.

The solution structure for a declarative task consists of a list of solution components. The components can be added and removed in a manner similar to the addition and removal of problem components, by clicking the + and – buttons. Each solution component has a label, the type of elements it may hold (i.e. the concept from the domain ontology), a type and the number of elements it may hold. The type determines the choice of input widget that would be ultimately displayed in the student problem solving interface. Solution components can be either 'text', 'boolean' or 'choice', which results in a textbox, check box or drop-down list respectively. In the case of specifying 'text' as the type for the solution component, the author also has to specify the dimensions of the resulting text box, labelled as length and rows.

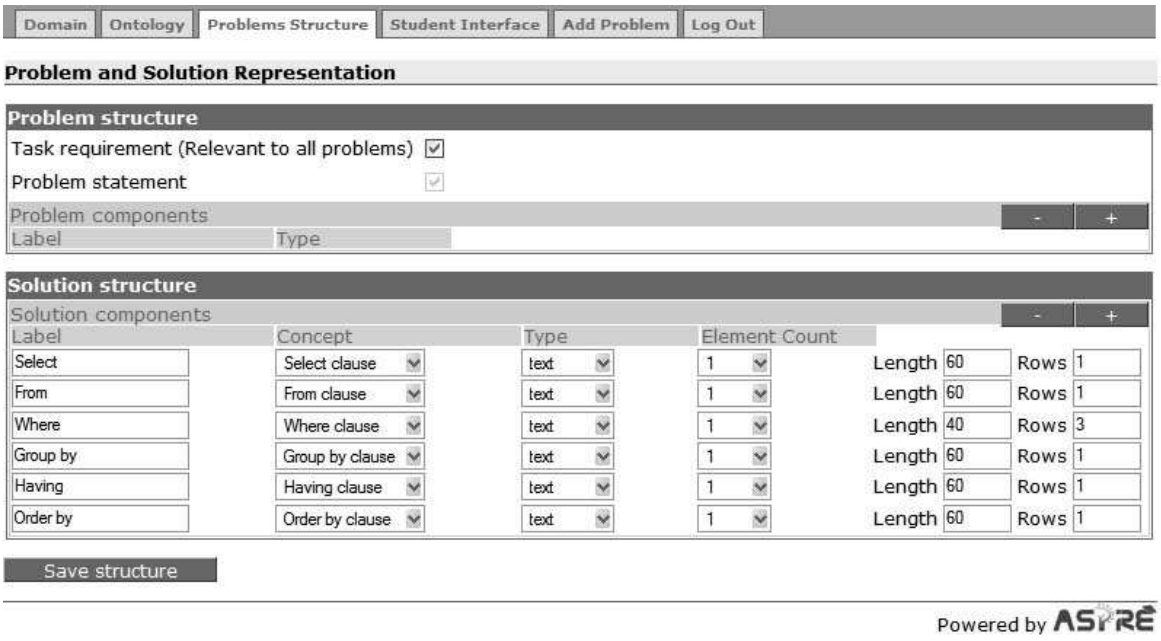


Figure 7. The interface for specifying problem/solution structure

The solution structure specified for SQL-Tutor (Figure 7) reflects the six clauses that exist in SQL SELECT statements: Select, From, Where, Group by, Having and Order by. The solution components can hold textual elements of their respective types. All but one text box to be displayed in the solution workspace of the student interface are 60 columns wide and 1 row tall. The `where` clause is 40 columns wide and 3 rows tall.

2.4.2. Procedural domains

The interface for modelling the structure of problems and solutions for procedural tasks (shown Figure 8) is similar to the interface for declarative tasks described in the previous section. The main difference is the presentation of the solution structure. As each problem solving step requires a solution which may contain several parts, the composition of solutions for each step has to be modelled separately. Consequently, the solution structure for procedural domains consists of a collection of solution component lists, one for each problem solving step.

The structure for problems in APT (Physics tutor), illustrated in Figure 8, consists of only the default problem statement. As problems of APT consist of four steps, the structure of solutions for each step has to be modelled using the interface. The solution for the first step (`choose an appropriate equation`) consists of only one component which may hold an equation. Solutions for the second step, which involves identifying the known variable, may contain up to four components. In other words, students can identify up to four variables depending on the given scenario. The final two steps also contain a single component, which require an equation as their solutions.

2.5. Student Interface Designer

In step 3 of the authoring process, the author specifies the student interface. Once developed, this interface becomes the communication medium that students will use when interacting with the developed intelligent educational system. Therefore, when the author finishes designing the interface, the specification becomes a part of the domain model, and will be transferred to ASPIRE-Tutor to be served to students.

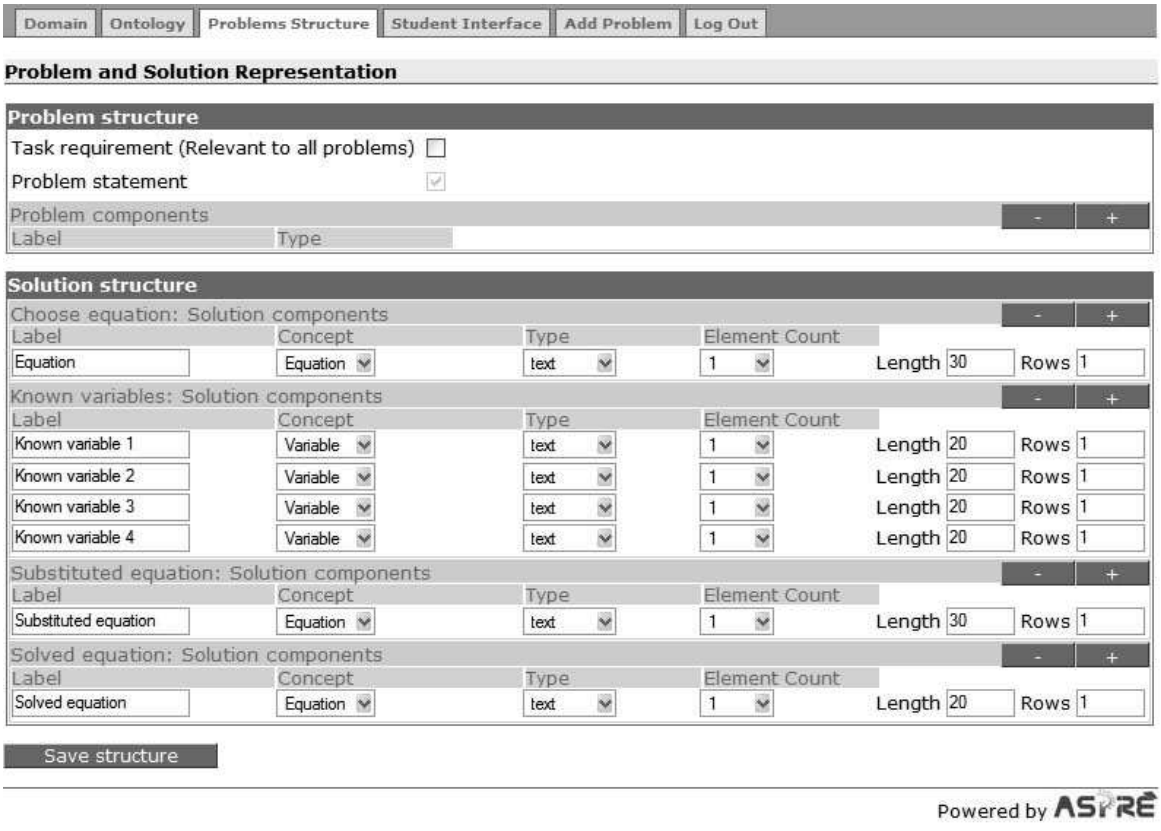


Figure 8. Specifying problem/solution structure for a procedural task

We have simplified the interface building task by providing an interface layout, dynamically calculated from the previously specified problem/solution structures. This interface layout serves as the initial phase in the process (illustrated in Figures 9 and 10). The assumption here is that every interface will consist of four basic areas:

- the navigation area at the very top of the page, providing the buttons that students interact with;
- the problem area at the top of the page, providing the problem statement and any additional problem components the author specifies;
- the solution area at the centre of the page, providing the workspace for students to enter their solutions; and
- the feedback area at the right side of the page, which will contain the system's feedback to the student, and buttons for the student to request feedback.

The navigation area's options include:

- the problem selection drop-down menu where students can select the next problem they would like to work on;
- the `Next Problem` button, which requires the educational system to select the best problem for the student to solve, based on the student model;
- the `History` button, which shows the history of interactions in the current session;
- the `Student Model` button, which provides an assessment of the student's proficiency level;
- the `Tutorial` button, which provides a tutorial on how to use the system;
- the `Help` button, which gives quick pointers on how to use the system and/or go about solving the problems;
- the `Logout` button, which allows the student to exit the tutor at any time.

The starting point for constructing the student interface is the previous specification of the problem and solution structures. The ASPIRE-Author examines the specified components and displays them in accordance to their characteristics in the problem/solution areas. The navigation and feedback areas, however, are displayed to the author without any additional calculations. The author is then able to check the interface layout. The current implementation of the Student Interface Builder does not allow the author to move the various components of solutions/problems, but we plan to implement that kind of support in the second year of the project.

If there are any necessary changes to the structures, the author may go back to the problem and solution structures page to make those changes, which will be reflected in the interface as soon as they are saved. Once the author is satisfied with the problem/solution structures and their interface representation, the author is able to move to the next authoring step of adding problems and their solutions (which are covered in milestone 9). The next two sections provide examples of this authoring step for both types of domain tasks; declarative and procedural.

2.5.1. Declarative domains

Figure 9 shows a screenshot of the Student Interface Builder for our SQL-Tutor example. The Student Interface Builder examines the problem and solution structures and presents the corresponding interface to the author. In this case, the problem structure does not contain any additional components, so only problem statement will be displayed. The six solution components are displayed one after the other in the solution area. As all the solution components are of type "text", a `textbox` is drawn beside each component's name according to its specified size; length and rows.

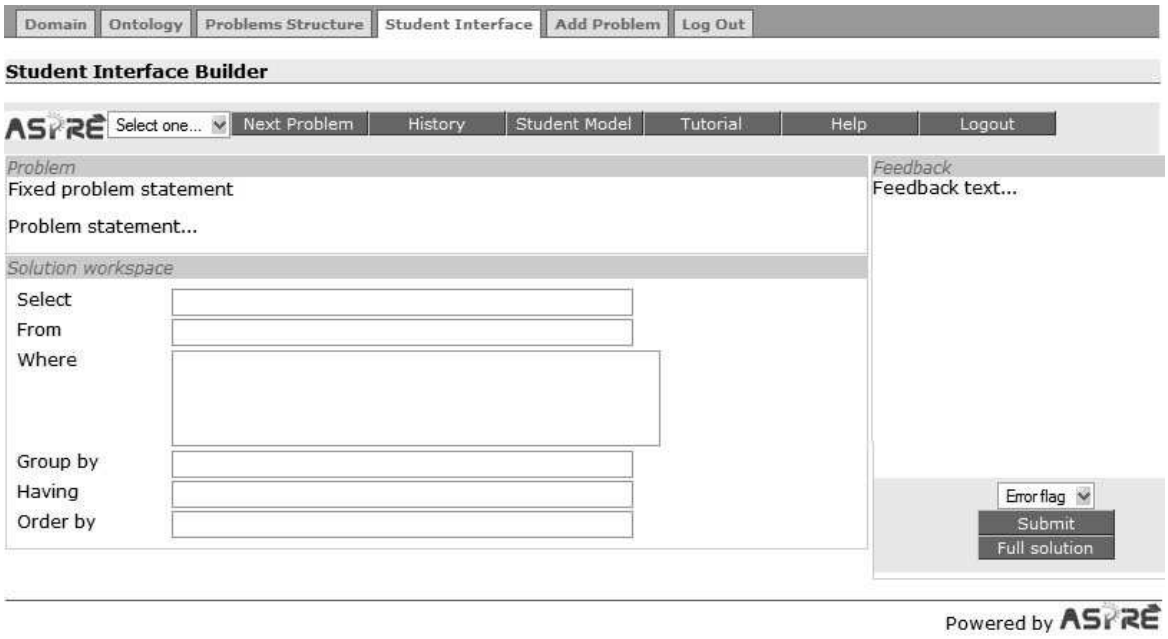


Figure 9. Student interface for the SQL domain

2.5.2. Procedural domains

Figure 10 shows the screenshot of a student interface produced by the Student Interface Builder for the physics tutor, which teaches a procedural task. Similar to declarative tasks, the interface areas populated with the appropriate structures are presented to the author. The main difference here is the presentation of each problem solving step. As specified by the author previously, each step may be presented on a separate page. Therefore, although the components of the steps are examined and presented in the same manner as for declarative domains, there are additional checks for presenting them on their appropriate pages. Figure 10 shows what the student interface will look like for the first step of the physics problem.

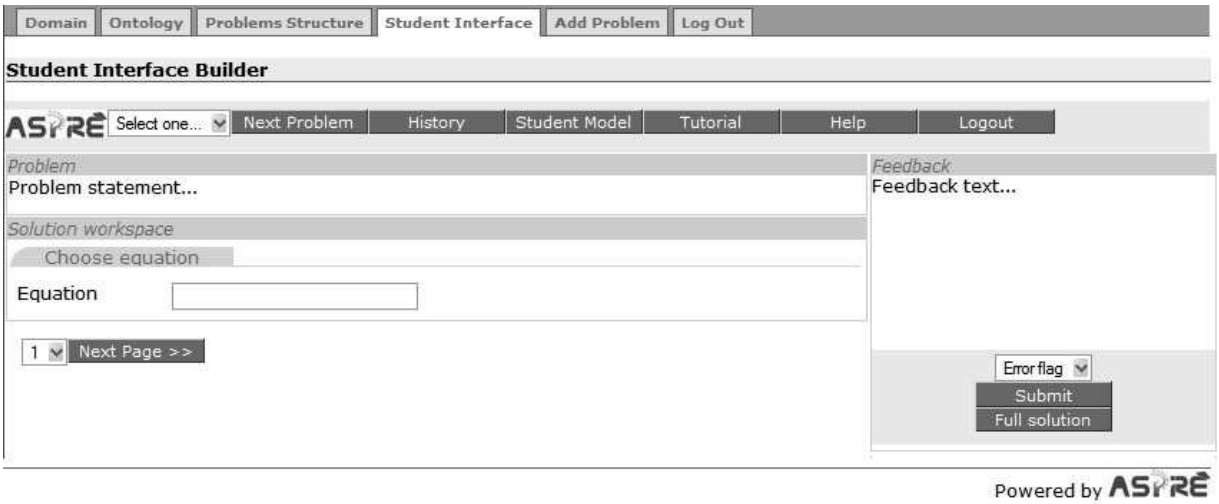


Figure 10. Student interface for a procedural task

As discussed above, all steps share the same problem structure, but differ in their solution structure. The solution structure of a particular step is only presented on the page corresponding to that step. Given that these problem-solving steps may depend on each other, the solutions of previous steps are displayed in the problem area as part of the current step's problem structure.

The author can navigate through the pages of a procedural task by clicking on the `Next Page >>` or `<< Previous Page` buttons, or by choosing the page number from the pages drop down menu.

3. ASPIRE-Tutor

ASPIRE-Tutor is responsible for the delivery of intelligent educational systems developed in ASPIRE-Author. Figure 11 illustrates the general architecture of ASPIRE-Tutor, which was discussed in the previous report (Mitrovic et al., 2005).

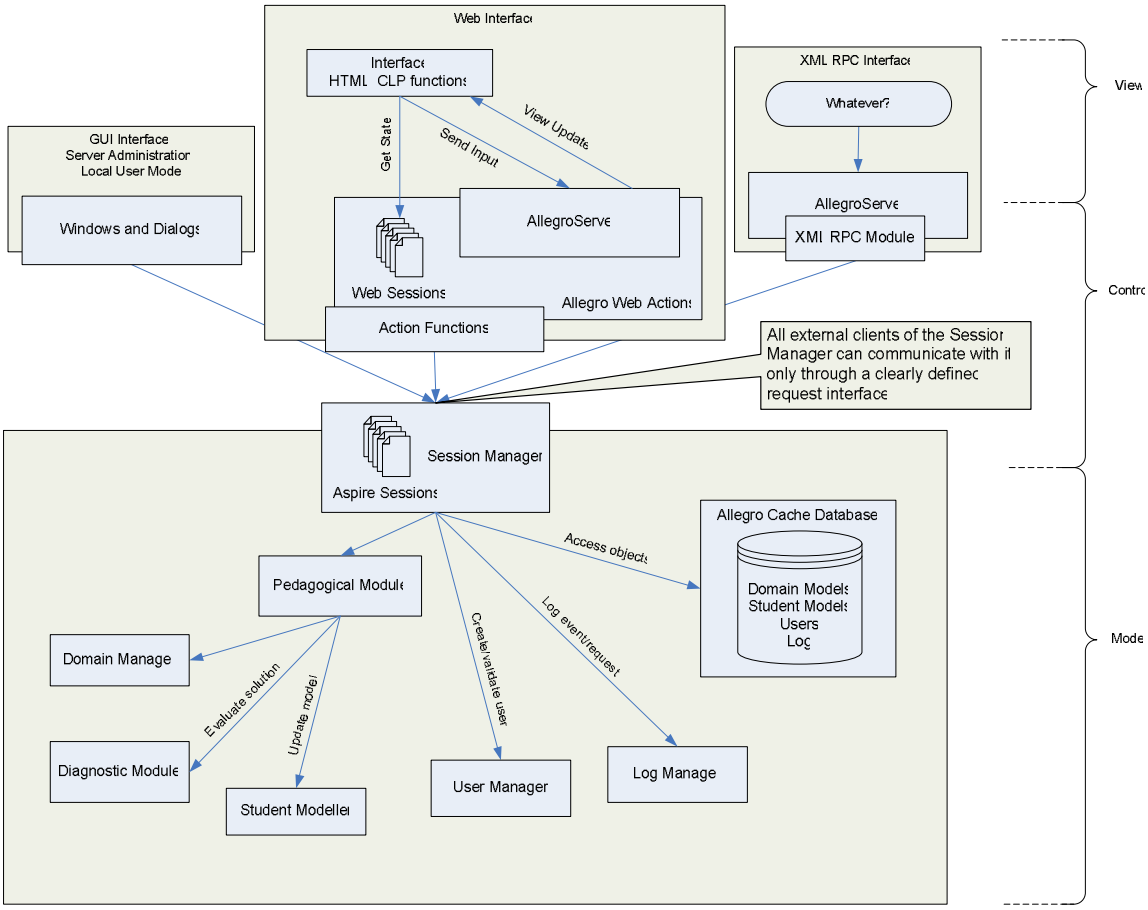


Figure 11. The architecture of ASPIRE-Tutor

3.1. Implementation of the Framework

The plan for the current reporting period includes the Diagnostic Module and the Student Modeller. However, in order to be able to implement these two components, we also had to implement all the packages and the basic functionality of all the modules.

3.1.1. Packages Structure

The package structure of ASPIRE-Tutor (shown in Figure 12) was implemented to reflect its architecture. To minimise coupling[5] and increase cohesion[6] of the system modules, each module is implemented as a separate package centred around the functionality provided by the module. The `session-manager` package represents the component of the system which manages the system data and delegates operations on the data to other modules. Each module encapsulates the operations on data and hides the implementation behind it via a clearly defined public interface abstracted from implementation details. This approach, naturally supported by the Object-Oriented programming paradigm, minimises further development and maintenance efforts. Even before the system is complete, it is possible to run the system during development and testing stages.

The `session-manager` runs the web and database server controlling the access to data. The Session Manager module translates external requests and operations coming from the client into a unified language of requests understood by the rest of the system. In this way, the Session Manager translates all commands arriving from users (via XML-RPC or Web clients) into operations that are further handled by the other packages. Session Manager dispatches the operations to the modules responsible for handling specific requests. For example, login and logout requests are handled by the User Manager, implemented in the `user` package, and students' solutions are processed by the Pedagogical Module, implemented in the `pedagogy` package. The `session-manager` package decouples the packages dependent on it so that they do not communicate directly with each other.

The Pedagogical Module depends on the Diagnostic Module, Domain Module and Student Modeller implemented in the corresponding packages. The `pedagogy` package, similarly to the `session-manager` package, acts as a client of the packages it depends on. In this way `diagnosis`, `student-model` and `domain` packages are independent of each other, with the `pedagogy` package being responsible for making high-level decisions. The finer-grain details of these decisions are provided by the functionality encapsulated in the lower-level packages. This separation of module functionality and interaction is encouraged by the principle of `separation of concerns`. Each module and the corresponding package encapsulate the minimum functionality necessary for carrying out its tasks. For example, the `student-model` package does not need to be responsible for finding, loading and saving individual student model objects corresponding to certain users. Instead it simply manipulates student model objects passed to it by routines called from the `pedagogy` package. These routines, however, are contained in the `student-model` package, but it is up to the Pedagogical Module to decide when they should be called upon.

[5] Coupling or dependency is the degree to which each module relies on any other module. Low coupling in programming means that one module does not have to be concerned with the internal implementation of another module, and interacts with modules via a stable interface. Therefore, with low coupling, a change in one module will not require a change in the implementation of another module. Low coupling is a desirable sign of a well structured computer system.

[6] Cohesion is a measure of how well the lines of the source code within a module work together to provide a specific piece of functionality. Modules with high cohesion tend to be preferable as high cohesion is associated with several desirable traits of software including robustness, reliability, and reusability.

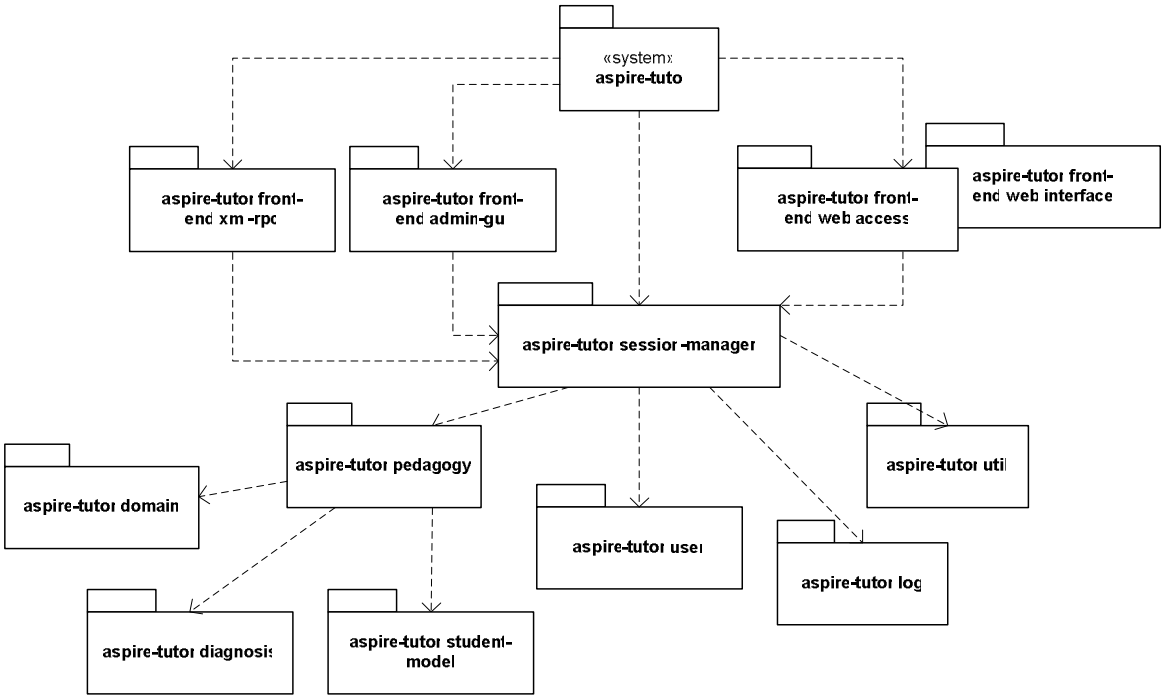


Figure 12. The package structure of ASPIRE-Tutor

The directory and source code layout of ASPIRE-Tutor is shown in Figure 13. The project consists of seven top level directories for grouping various files. The `db` directory holds the physical database file. The compiled executable version of the system would reside in the `dist` directory. All user and development documentation resides within the `doc` directory. The `lib` directory contains any extra libraries that the system would use. Any other resources reside within the `resources` directory. The directories under `src`, which hold Lisp source code files, reflect the system's package structure. Each source file is stored under the appropriate directory. All HTML pages including their resources such as style sheets, images and scripts are stored under the `web` directory.

```
\---aspire-tutor
    +---db
    |   +---test-db
    |   \---tmp-db
    +---dist
    +---doc
    |   +---dev
    |   \---usr
    +---lib
    +---resources
    +---src
    |   +---diagnosis
    |   +---domain
    |   +---front-end
    |   |   +---admin-gui
    |   |   |   \---icons
    |   |   +---web
    |   |   |   +---access
    |   |   |   \---interface
    |   |   \---xml-rpc
    |   |       \---access
    |   +---pedagogy
    |   +---session-manager
```

```
|   +---student-model
|   +---user
|   \---util
\---web
    +---base
    |   +---css
    |   +---img
    |   \---scripts
    \---domains
```

Figure 13. The directory structure of ASPIRE-Tutor

3.1.2. ASPIRE-Tutor classes

The UML diagram in Figure 14 shows the classes developed for ASPIRE-Tutor. Please note that the domain model classes are shared between the tutoring and authoring side, and have already been discussed in Section 2.2.2. The Diagnostic Module, the Student Modeller, and the classes specific to these components are described in Sections 3.2 and 3.3 respectively.

Information about users is stored in the `User` class, and also the appropriate subclasses (depending on the type of users). The affiliation of users would be defined by the administrators, when the user account is created. ASPIRE-Tutor recognises three types of users: students, teachers and administrators. The teacher defines the specific features of an intelligent educational system he/she would like to use for a group of students, and these details will be stored as an instance of the `group` class.

Information about each Web session will be maintained using an object of the `session` class. Different types of sessions correspond to the three types of users: students, teachers and administrators. The `student-session` class is of special importance for this report, as we have been working on the functionality related to students. At any one time, the student's session will be related to an instance of the `pedagogical-state` class, representing information about the current instructional domain the student is working in, the id of the selected problem, and the attempt number on the current problem.

A student will have an instance of the `student-model` class for each intelligent educational system he/she uses. The `student-model` class stores the long-term model of the student's knowledge. This class contains the student's proficiency for the domain as a whole. For each sudomain of the instructional domain the student has used, there would be an instance of the `subdomain-student-model` class, storing details of the student's behaviour in the subdomain.

If there is a defined pre/post test for the chosen instructional domain/system, the result of the test would be stored as an object of the `test-result` class. Instances of `test-result` class store pre/post-test results, including the student's answer to each question, and the total score. In instructional domains for which pre/post-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 logs on to the educational system. When the teacher has specified that the students should sit a post-test, a different test will 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. Pre/post-test dates are stored in the `group` class. There should be an instance of `pedagogical-settings` class associated with each `group` instance.

For each solution attempt the student submits, there would be an instance of the `solution-attempt` class, storing information about the problem the student attempted, the timestamp, the

raw solution received, the attempt number, the outcome of the diagnosis (the lists of violated/satisfied constraints and appropriate binding lists), as well as the feedback the student received.

The student model also contains constraint histories, implemented via the `constraint-use-case` class. An object of this class exists for every constraint relevant to a student's submission, and stores the timestamp (i.e. the time and date), information about whether the constraint was used correctly or not (the `satisfaction` attribute), information about the domain the constraint belongs to, and the problem number within that domain.

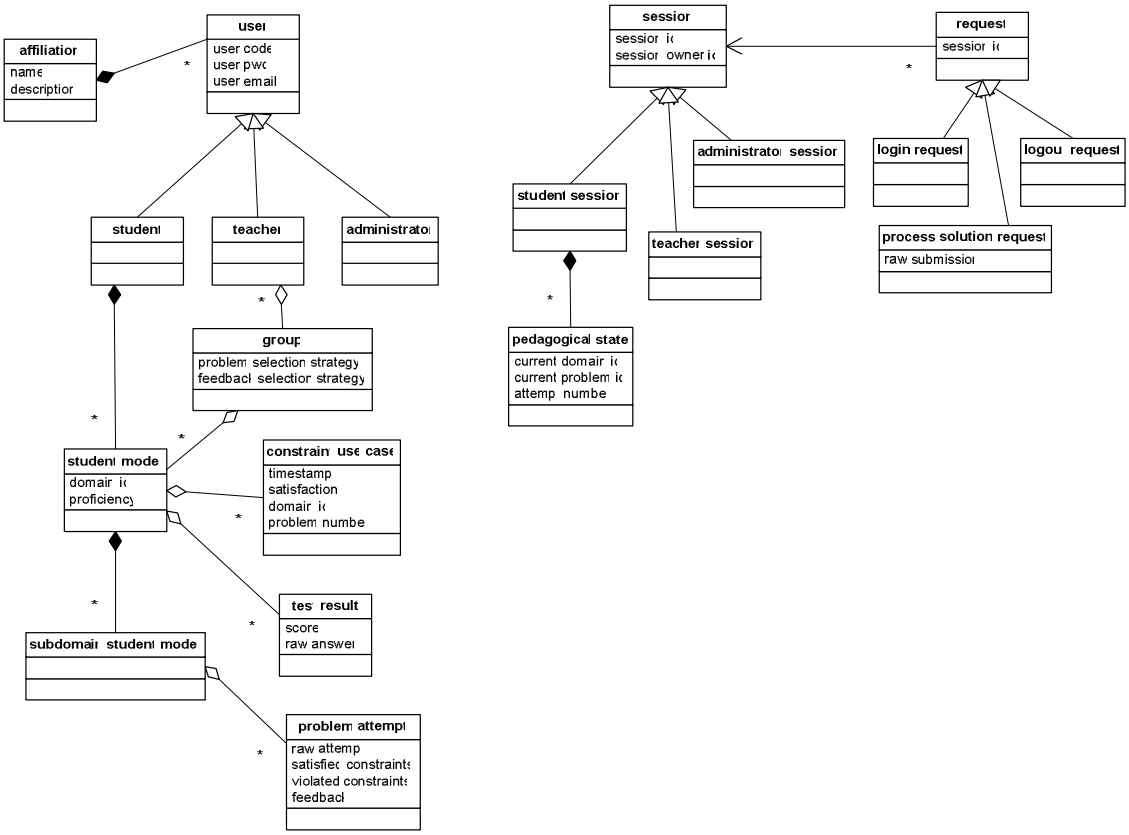


Figure 14. The class diagram for ASPIRE-Tutor

3.1.3. Request Protocol

This section describes in detail the interaction between external client interfaces and the core application server. User actions are delivered to the Session Manager module in the form of HTTP request or XML-RPC commands through the application web-server listening for incoming network connections. In line with the Model-View-Architecture (Krasner & Pope, 1998), the interface layer is decoupled from the core application. In ASPIRE-Tutor, the communication between the interface layer and the Session Manager is carried out through an internal request protocol. The basic principle of this protocol is that every user action, no matter what type of interface it is coming from, is delivered to the Session Manager in a specific request type form. For example, a login attempt initiated in a web browser as an HTTP POST request is handed over to the Manager as an instance of the `login-request` class. Although the class diagram in Figure 14 shows only three types of requests, there are `request` classes corresponding to other types of user actions supported by the system. In this report we only concentrate on the three types of request which are handled by the components we have focused on in the current period.

The Session Manager is aware of the request specialisation, and thus each request is handled in a unique appropriate way determined by its request type. The implementation of this request handling protocol is based on generic functions and specialised methods that are the building blocks of the Common Lisp Object System.

All requests handled by the Session Manager can be divided into two broad categories: persistent and transient requests. Persistent requests are those that carry certain value of the pedagogical or administrative nature; these requests are automatically saved in the database as the system activity log. Login, logout and solution submission requests are examples of persistent requests. Persistent requests and the system's responses to them are used internally by the system for providing optimal learning environment and for external analysis and monitoring of learning progress. Transient requests, on the other hand, are small atomic transactions that do not need to be saved in the database. For example, requests cantered around retrieval of information required for rendering of parts of the interface are represented as transient requests. These requests are processed, but are not logged in the system database.

3.2. Diagnostic Module

The Diagnostic Module undertakes the evaluation of solutions submitted by students, and also returns the results of these evaluations. Initially, the Pedagogical Module passes the information about the student's solution to the Diagnostic Module, including the domain name, problem id, and the components of the solution. The Diagnostic Module then creates an instance of the Diagnosis class to represent the solution, and starts evaluating it. Once the evaluation is complete, the Diagnosis object will contain the result of the evaluation (the satisfied and violated constraints, and the variable bindings), which are then passed back to the Pedagogical Module.

A student's solution is processed in two phases. Firstly, the relevance conditions of all domain constraints are matched to the student solution and the ideal solution for the same problem. In the second phase, the satisfaction condition of the relevant constraints are also matched. This process results in the lists of relevant and satisfied/violated constraints, as discussed in more detail in Section 3.2.2. The Student Modeller uses this information to update the long-term student model (as discussed in Section 3.3), and the Pedagogical Module uses this information in order to select appropriate feedback for the student.

3.2.1. Diagnostic Module interface

There is only one operation exposed to the Pedagogical Module:

```
process-solution (domain-id problem solution-components)
```

This operation passes the information about the student's solution and requires that it be processed. The Diagnostic Module creates an instance of the Diagnosis class, and evaluates the solution against the ideal solution and the domain constraints. Once completed, the `Diagnosis` object will contain the lists of satisfied and violated constraints that have resulted from the evaluation, and possibly other relevant data (e.g. for visually identifying where in a solution the errors occurred).

3.2.2. Matching constraints

Constraint-based intelligent tutoring systems represent domain knowledge in terms of constraints, which denote the basic domain principles. 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.

CBM starts from the observation that all correct solutions in a particular instructional domain share one property: no correct solution violates any domain principles. Therefore, domain knowledge may be represented in terms of state descriptions of the form:

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

As discussed in the previous report (Mitrovic et al., 2005), constraints test the student's solution for syntax errors and compare 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, 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 functions.

Each constraint has a relevance condition, and a satisfaction condition. These conditions may be simple tests, or multiple tests connected with `AND`, `OR` or `NOT`. A test may be any Lisp function or the call to the `match` function, which implements pattern matching. Pattern matching is a well-known Artificial Intelligence technique in which a pattern (i.e., an expression containing variables) is compared to a constant expression (with no variables) to test whether they are similar. Patterns contain variables, that can represent any kind of expressions, and constants, which must appear in the constant expression as they are given in the pattern. Pattern matching finds substitutions for variables in the pattern that will make the pattern identical to the constant expression. In ASPIRE, patterns are specified by using the following symbols:

- `?var` - A simple variable, which matches any one expression
- `?*var` - Matches zero or more expressions
- `?+var` - Matches one or more expressions
- `??var` - Matches zero or one expression
- `(?if exp)` - Tests if `exp` is true
- `(?is var predicate)` - Tests the specified predicate on one expression
- `(?or pat ...)` - Matches any pattern on one expression
- `(?and pat ...)` - Matches every pattern on one expression
- `(?not pat ...)` - Succeeds if pattern(s) do not match

Constraints differ significantly in their complexity. An example of a very simple constraint from SQL-Tutor is:

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

The relevance pattern of this constraint is `t`, which is always satisfied; therefore, this constraint is relevant to all the student's solutions. The satisfaction pattern specifies that the SELECT clause of the student's solution (represented by the variable `ss`) cannot be empty.

However, constraints can be much more complex. In order to test for specific features of solutions, patterns need to be specified for the `match` function. Other conditions may be any LISP functions, and they can use the binding lists obtained from the matching process to perform additional tests on the student's solution. Constraint 186, given below, contains complex conditions. Its relevance condition contains a conjunction of seven tests. For the constraint to be relevant to a submitted solution, all seven tests need to be met. The initial two tests make sure that the `where` component of the student solution and the ideal solution are not empty, respectively. If that is the case, the third test binds the variable ?n to the names the student used in the WHERE component (note that this might result in multiple bindings for variable ?n, if there is more than one name in WHERE). The fourth test keeps only those values of ?n which are valid attributes in the current database (ensured by using the attribute-p predicate), and then the fifth tests is met only by numerical attributes. The first match requires a specific pattern to appear in the student's solution, where the value of the attribute is greater than a specific numeric constant (?c1), while the second match function contains a similar pattern to be found in the ideal solution, including the same attribute which is greater than or equal to another constant (?c2). If that is the case, the satisfaction condition ensures that the constant in the student's solution is 1 less than the constant in the ideal solution.

```
(p 186
  "Check the numerical constant you are using in the WHERE clause!"
 (and (not (null (where ss)))
      (not (null (where is)))
      (bind-all ?n (names (where ss)) bindings)
      (attribute-in-db (find-schema (current-database *student*)) ?n ss)
      (equalp (find-type ?n ss) 'numeric)
      (match '(?*d1 ?n ">" (?is ?c1 numericp) ?*d2) (where ss) bindings)
      (match '(?*d3 ?n ">=" (?is ?c2 numericp) ?*d4) (where is)
bindings))
(equalp (string-to-number ?c2) (1+ (string-to-number ?c1)))
  "WHERE")
```

We have developed functions that allow any test to be specified in the constraint language. The actual processing of a student's solution is further discussed in the following section.

3.2.3. Speeding up the evaluation of student solutions

As discussed earlier, CBM is computationally very simple because it reduces student modelling to pattern matching. The conditions of constraints are patterns matched against the student's solution. Pattern matching is a simple, but potentially time-consuming operation, especially in situations when the number of patterns is large. It is therefore often done in AI systems by using RETE networks (Forgy 1982). A detailed discussion of the RETE pattern matching algorithm is beyond the scope of this report and we briefly give the fundamental ideas only. RETE networks are designed for many pattern/many object situation, typical for expert systems, where there is a large number of facts (objects) describing the current problem state, and a large number of rules (patterns). A RETE network is the result of compiling patterns so that all conditions that appear in many patterns are applied only once. RETE networks consist of a number of nodes, which apply the conditions on objects they are given.

In constraint-based educational systems, all constraints belonging to an instructional domain need to be used when diagnosing students' solutions. In an environment support multiple educational systems, each of which could have hundreds of constraints, speed could potentially become an issue, especially with many users accessing the system simultaneously. A complete evaluation of a solution taking only a few seconds can become a greater inconvenience once CPU time is shared and evaluation times extend. In order to ensure efficient processing of student solutions, ASPIRE-Tutor compiles constraints into structures that resemble RETE networks. We call these structures `constraint networks`. For a given domain, we create a pair of constraint

networks, as illustrated in Figure 15. The relevance network contains compiled relevance conditions, and the other network compiles satisfaction conditions of all domain constraints. The student's solution is first propagated through the relevance network to determine which constraints are relevant. In the second phase, the solution is propagated through the satisfaction network, but only for those constraints that are relevant. Finally, the satisfaction network produces a list of violated and a list of satisfied constraints.

There are three kinds of nodes in the constraint networks: input, internal and output nodes. Each input node may be connected to one or more internal nodes, referred to as its children. Each internal node has one input (coming from the input node, or another internal node), and may be connected to a number of other nodes (internal or output). Each internal node applies a test (corresponding to a condition in a constraint) to either SS or IS and, if the test is satisfied, propagates the resulting bindings to its children.

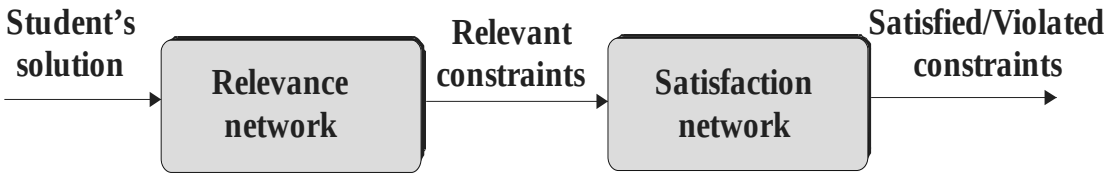


Figure 15. The processing of a student's solution

Relevance and satisfaction networks have slightly different structures. The number of input nodes in a relevance network is equal to the number of distinct conditions which appear as first conditions in any constraint. If several constraints have identical initial test in their relevance conditions, the relevance network will have an input node containing that test, and this input node will be shared by all these constraints. This way, the pattern-matching results can be re-used, and the amount of processing is reduced.

Each output node in the relevance condition corresponds to one of the domain constraints. Once the student's solution reaches an output node in the relevance network, the relevance condition of the corresponding constraint has been met, and the constraint is added to the list of relevant constraints, together with the appropriate binding list.

Each input nodes of a satisfaction network contains a constraint id. If the constraint is relevant for the student's solution, the satisfaction condition will be applied to it, by propagating the solution to the children of the input node. When the solution reaches an output node, the satisfaction network adds the constraint number to the list of satisfied constraints (with the binding list). If any of the tests in the satisfaction condition (i.e. the tests in the nodes connected to the input node) fail, the constraint is violated, and is added to the list of violated constraints.

Let us illustrate constraint networks on the example of the following constraints from SQL-Tutor[7]:

```
(p 110
   "You need the ON Keyword in the FROM clause!"
    (member "JOIN" (from-clause ss))
    (member "ON" (from-clause ss))
    "FROM")
```

[7] Please note that these constraints are given in a simplified form, to make the understanding of the example easier.

```
(p 358
   "Check the syntax for the JOIN and ON keywords in the FROM clause!"
   (and (member "JOIN" (from-clause ss))
        (member "ON" (from-clause ss)))
   (match '(?*d1 ?t1 ??s1 "JOIN" ?t2 ??s2 "ON" ?a1 "=" ?a2 ?*d2)(from-
clause ss) bindings)
   "FROM")

(p 11
   "If the JOIN keyword is used in the FROM clause,
    the same clause should contain a join condition specified
    on a pair of attributes from corresponding tables being joined."
    (and (member "JOIN" (from-clause ss))
         (member "ON" (from-clause ss)))
         (match '(?*d1 ?t1 ??s1 "JOIN" ?t2 ??s2 "ON" ?a1 "=" ?a2 ?*d2)
(from-clause ss) bindings)
         (valid-table ?t1)
         (valid-table ?t2)
         (attribute-of ?t1 ?a1))
    (and (attribute-of (find-table ?t2 (current-database *student*)) ?a2)
         (equalp (find-type ?a1 ss) (find-type ?a2 ss)))
    "FROM")

(p 187
   "If the JOIN keyword is used in the FROM clause,
    the same clause should contain a join condition specified
    on a pair of attributes from corresponding tables being joined."
    (and (member "JOIN" (from-clause ss))
         (member "ON" (from-clause ss)))
         (match '(?*d1 ?t1 ??s1 "JOIN" ?t2 ??s2 "ON" ?a1 "=" ?a2 ?*d2)
(from-clause ss) bindings)
         (valid-table ?t1)
         (valid-table ?t2)
         (attribute-of ?t1 ?a2))
    (and (attribute-of (find-table ?t2 (current-database *student*)) ?a1)
         (equalp (find-type ?a1 ss) (find-type ?a2 ss)))
    "FROM")
```

The relevance network for these constraints is given in Figure 16. Input and internal nodes are shown as rectangles, while output nodes are shown as ovals. Constraint 110 has only a single test in its relevance condition (checking that the JOIN keyword appears in the FROM clause of the student's solution), and if it is met by the students solution, the output node is reached, meaning that the constraint is relevant. Constraint 358 shares the same test with constraint 110, but additionally requires that the ON keyword also appears with JOIN. If both of these conditions are met, the constraint is relevant. Constraint 11, in addition to these two tests, also has four new tests, which are checked in order. Constraints 187 and 11 have almost identical relevance conditions; the only difference is in the last test, and that is why there are two branches from the internal node testing whether variable ?t2 has a value which is a valid table.

Note how tests are shared: the input node would apply the initial test to the student's solution, and if it is met, the joining internal nodes will apply more tests to the student solution. Instead of having to apply the same test five times in order to test the relevance condition of each constraint independently, the constraint network re-uses the results produced by network nodes, thus reducing the total number of tests applied on the student's solution.

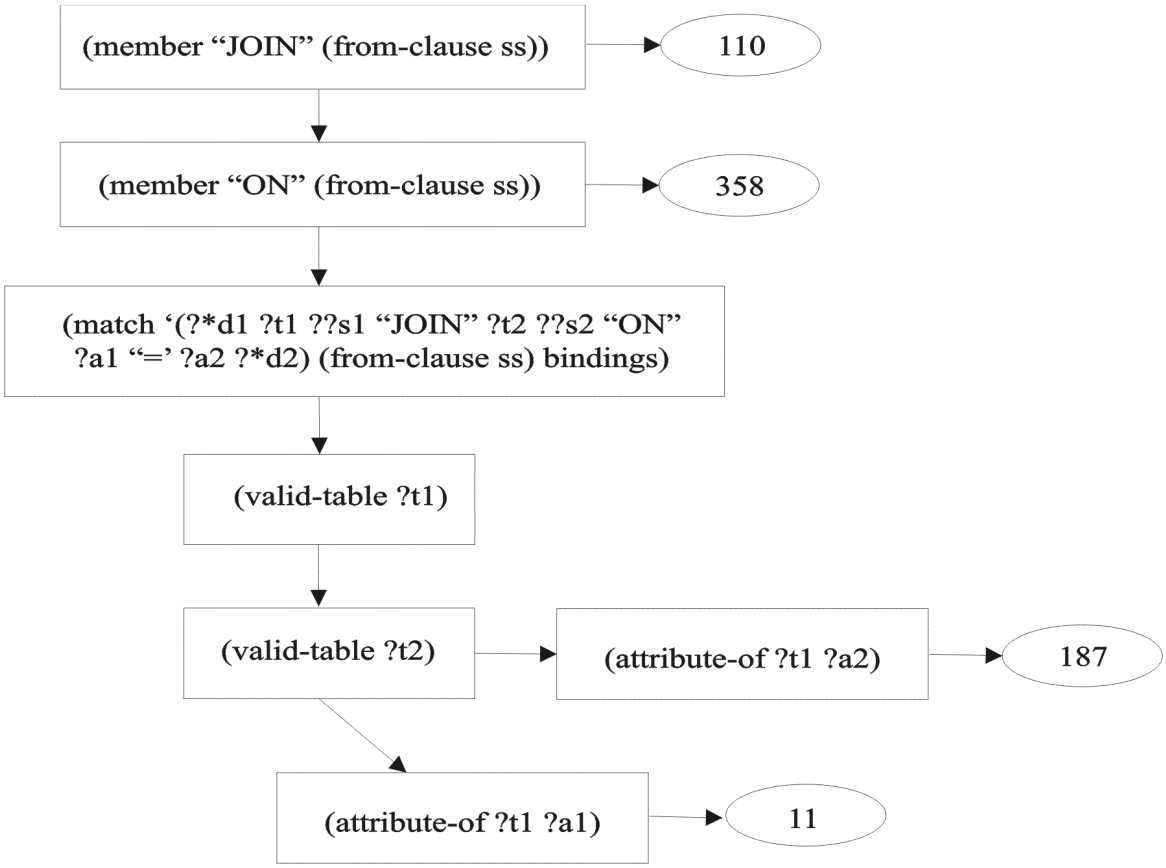


Figure 16. An example relevance network

3.2.4. Locking system

Constraint networks must be created at start-up, with the constraints from the chosen domain. As discussed earlier, there is a pair of constraint networks (i.e. relevance and satisfaction networks) created for each domain. The total number of networks would be determined by the number of instructional domains served by ASPIRE-Tutor. On start-up, the Diagnostic Module iterates through all instructional domains, and for each domain performs the following steps: gets the constraints for each domain from the Domain Manager, creates a pair of constraint networks, and stores references to them.

Due to the nature of constraint networks, only one solution can be propagated through a network at a time. Because of this, a locking system has been created for situations when there is simultaneous need for a network (i.e. multiple students submit solutions to the same instructional system). When a process-solution call has been made, the Diagnosis object makes a request to the repository of constraint networks corresponding to the domain. If the network is not in use, it is returned, and a `Request-Lock` is set. If the network is locked, the request continues to wait until the network becomes available.

Once the submission is propagated through the constraint network, the lock is removed, and other requests can now retrieve the network.

3.2.5. Diagnostic Module Classes

Figure 16 illustrates the classes that the Diagnostic Module uses for its specific functions. The Diagnosis class, as discussed in Section 3.1.2, stores all the necessary information about the student's solution and the results of its evaluation. Other classes included in the diagram in Figure 16 are related to the constraint networks. As discussed in Section 3.2.4, the constraint networks are generated on system start-up, from the domain constraints obtained from the Domain Manager. For each network, an instance of the `Constraint-Network` class is created. Input nodes and internal nodes of relevance networks are represented as instance of the `Node` class, as they have the same structure (i.e. each node contains a test to be evaluated). The output nodes of both relevance and satisfaction networks have the same structure, and are represented as objects of the `Output-node` class. The input nodes of satisfaction networks are different from the input nodes of relevance networks, as they only contain the constraint id, and not a test.

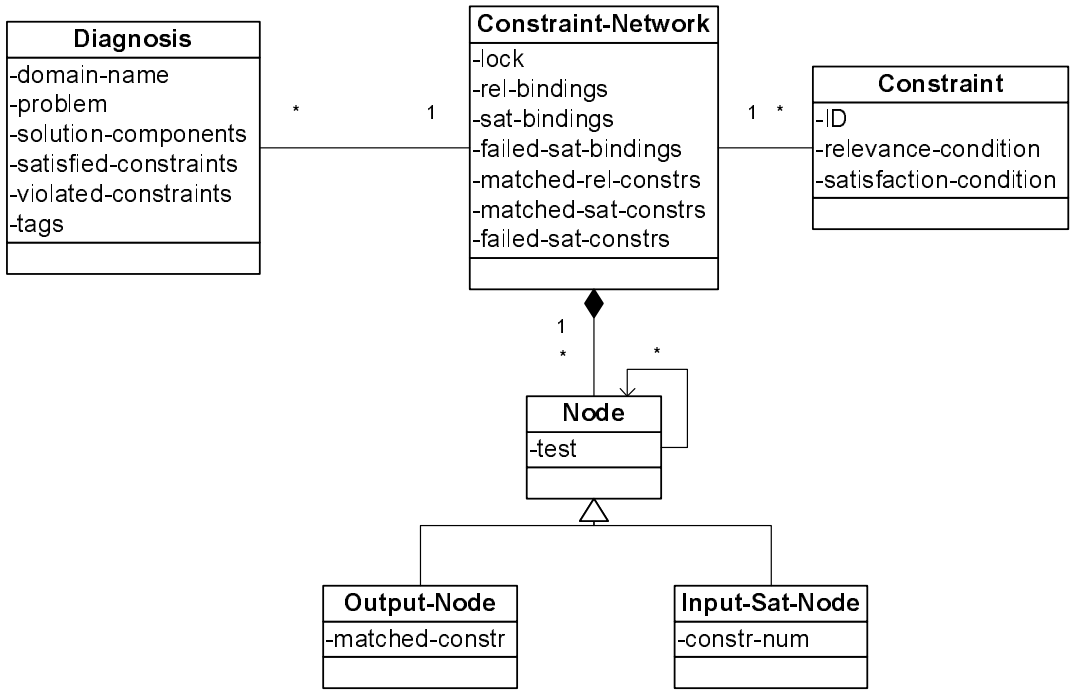


Figure 16. The class diagram for the Diagnostic Module

3.3. Student Modeller

Student Modeller is responsible for maintaining long-term models of students' knowledge. Whenever a student submits the solution to a problem, the Diagnostic Module will analyse it, and produce the short-term model (as discussed in Section 3.2), which it passes to the Pedagogical Module. This module then requires the Student Modeller to update the long-term model.

ASPIRE stores general information about each user (such as user code and password) as attributes of the `user` class. The `student` class extends the `user` class and provides capabilities for storing information that describes the student's knowledge of a particular domain. Within ASPIRE-Tutor it is necessary to keep a model of the student's knowledge for each instructional domain the student has worked on. This information is distributed over several classes, described in Section 3.1.2. The Student Modeller module is responsible for maintaining a long-term model

of each student's knowledge, aggregating and providing data about individual student models to the Pedagogical Module.

3.3.1. Student Modeller Interface

The operations exposed to the Pedagogical Module include:

- `make-student-model (domain-id)`

  Creates a new student-model object for the specified domain.

- `update-student-model (student-model solution-attempt)`

  Updates the student-model instance based on the evaluated solution-attempt containing required information (e.g. lists of satisfied and violated constraints). The Pedagogical Module will call this method to update the student model each time the student performs cognitively or pedagogically important action, such as submitting a solution to a problem.

- Queries
  - `student-level (student-model)` Gets the student's proficiency level for the domain.
  - `attempted-problems (student-model)` Gets the problems attempted by the student. This may include problems attempted but not completed as well as fully completed problems. This may be useful for problem selection.
  - `solved-problems (student-model)` Gets the problems solved by the student. This only includes the successfully completed problems. This is useful for problem selection.
  - `constraint-knowledge (student-model constraint-id)` For feedback and problem selection strategies the Pedagogical Module needs to know the student's comprehension of each particular constraint.
  - `visualise-student-model (student-model)` Gets a summary of the student-model for visualisation.

4. Allegro Cache

AllegroCache is a high-performance, persistent, dynamic object caching system (Aasman, 2005). It supports a full transaction model with long and short transactions, and meets the classic ACID requirements[8] for a database. AllegroCache allows us to work directly with objects as if they were in memory while in fact the object data is always stored persistently. It provides both a single user and client-server (multiple-user) configurations. In the client-server environment, different clients (or processes) on different processors can access the same database over the network. In the context of ASPIRE project we are using AllegroCache in the client-server configuration.

Purely for performance considerations HTTP requests received by the application web server on behalf of either ASPIRE-Tutor or ASPIRE-Author are normally processed in separate Lisp threads. When a client connects to the port on which AllegroServe web server is listening, AllegroServe passes that connected socket to a free worker thread which then wakes up and calls the internal function to process the given connection[9]. AllegroCache database serving as the backend datastore for both ASPIRE-Tutor or ASPIRE-Author allows multiple simultaneous connections to the database. However, from the perspective of AllegroCache, only one thread at a

[8] Every production database management system needs to achieve four goals: atomicity, consistency, isolation and durability (ACID). Databases that fail to meet any of these four goals is not considered reliable. AllegroCache meets all these essential ACID requirements.

[9] http://opensource.franz.com/aserve/aserve-dist/doc/aserve.html

time should be using a given database connection (AllegroCache Manual). Using a database connection includes:

- reading or writing the slot of a persistent object
- locating a persistent object from an index
- commit or rollback
- creating a persistent object
- deleting a persistent object

To be sure threads do not overlap during their use of a connection, we create a pool of database connections and have each thread pick a connection out of the pool for its exclusive use, returning the connection to the pool when the thread is finished. The Session Manager ensures that for each thread processing an HTTP request there is also a dedicated database connection. AllegroCache is based on optimistic concurrency model (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconoptimisticconcurrency.asp). When the database transaction associated with each request is complete, the Session Manager commits the changes to the database.

AllegroCache makes object persistence absolutely transparent. When an object of a persistent class is created or modified, the database reflects the transaction globally after the commit operation. During each transaction the persistent database objects can be manipulated just like transient objects.

5. Conclusions

This report covered the second reporting period of the ASPIRE project, and presented the work done on both ASPIRE-Author and ASPIRE-Tutor. In both cases, it was necessary to implement all the packages and the basic functionality of all modules, in order to be able to complete the planned components. On the authoring side, we have implemented three components of the authoring interface: Domain Structure Modeller, Problem/Solution Structure Modeller, and the Student Interface Builder. We also implemented two components of ASPIRE-Tutor: Diagnostic Module, and Student Modeller. We also briefly discussed AllegroStore, the underlying object-oriented database storing all the necessary data.

In this period, we have also started working on constraint generation, and this milestone will be completed in March 2006. The next reporting period includes two new components of the authoring interface: the Ontology Workspace and the Problem/Solution Editor.

6. References

1. Aasman, J. AllegroCache: A High-Performance Object Database for Large Complex Problems. 1005th International Lisp Conference, Stanford University, June 19-22, 2005.
2. Forgy, C.L. Rete: a fast algorithm for the many pattern/many object pattern match problem'. Artificial Intelligence 19, pp. 17-37, 1982.
3. Krasner, G.E., Pope, S.T. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. J. Object Oriented Programming, 1(3), 26-49, 1998.
4. Martin, B., Mitrovic, A. 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, 2003.
5. Mitrovic, A. Experiences in Implementing Constraint-Based Modeling in SQL-Tutor. B. Goettl, H. Halff, C. Redfield, V. Shute (eds.), Proc. ITS'98, pp. 414-423, 1998.
6. Mitrovic, A., Martin, B., Mayo, M. 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, 2002.
7. Mitrovic, A., Martin, B., Suraweera, P., Zakharov, K., Milik, N., Holland, J. ASPIRE: Functional Specification and Architectural Design. Tech. Report TR-COSC 05/05, University of Canterbury, 2005.
8. Mitrovic, A., Ohlsson, S. Evaluation of a Constraint-Based Tutor for a Database Language. Int. J. Artificial Intelligence in Education, 10(3-4), 238-256, 1999.
9. Ohlsson, S. Constraint-based Student Modelling. In Proc. of Student Modelling: the Key to Individualized Knowledge-based Instruction, Springer-Verlag, Berlin, pp. 167-189, 1994.
10. Steele, G.L. Common Lisp - the Language. Digital Press, 2nd edition, 1990.