Tech., Inst., Cognition and Learning, Vol. 4, pp. 00-00
Reprints available directly from the publisher
Photocopying permitted by license only

Problem-Solving Support in a Constraint-based Tutor for UML Class Diagrams

NILUFAR BAGHAEI, ANTONIJA MITROVIC AND WARWICK IRWIN

Department of Computer Science and Software Engineering
University of Canterbury, Private Bag 4800, New Zealand

We present COLLECT-UML, a constraint-based tutoring system that teaches object-oriented analysis and design using Unified Modelling Language (UML), a popular object-oriented modelling technology. Constraint-Based Modelling (CBM) has been used successfully in several tutoring systems, which have proven to be effective in evaluations performed in real classrooms. In this paper, we present problem-solving support available in COLLECT-UML. The system observes students' actions and adapts to their knowledge and learning abilities. We describe the system's architecture and functionality. The effectiveness of the system has been evaluated in two studies with students taking ITS and software engineering courses. Objective data shows that students' performance increases significantly while interacting with the system, and that they do learn the domain concepts. The students have enjoyed the system's adaptivity and found it a valuable asset to their learning.

Keywords: Problem-solving support, Constraint-based modelling, UML class diagrams, ITS evaluation

INTRODUCTION

Constraint-based tutors are Intelligent Tutoring Systems (ITS) which use Constraint-Based Modelling (CBM) [Ohlsson, 1994] to generate domain and student models. These tutors have been proven to provide significant learning gains for students in a variety of instructional domains. As is the case

*Corresponding authors: E-mail: {n.baghaei, tanja, w.irwin}@cosc.canterbury.ac.nz

with other ITSs [Brusilovsky & Peylo, 2003], constraint-based tutors are problem-solving environments; in order to provide individualized instruction, they diagnose students' actions, and maintain student models, which are then used to provide problem-solving support and generate appropriate pedagogical decisions. Constraint-based tutors have been developed in domains such as SQL (the database query language) [Mitrovic 1998; Mitrovic & Ohlsson, 1999; Mitrovic, 2003], database modelling [Suraweera & Mitrovic, 2002; 2004], data normalization [Mitrovic 2002, 2005], punctuation [Mayo & Mitrovic, 2001] and English vocabulary [Martin & Mitrovic, 2003]. All three database tutors were developed as problem solving environments for tertiary students [Mitrovic et al., 2004]. Students solve problems presented to them with the assistance of feedback from the system. The tutors for punctuation and English vocabulary were developed for 9-12 year old school children.

This paper presents our experiences in implementing a constraint-based tutor in the area of object-oriented (OO) analysis and design using the Unified Modelling Language (UML). The chosen task is very complex, as it requires sound knowledge of requirements analysis, design and UML. The text of the problem is often ambiguous and incomplete, and students need a lot of experience to be successful in analysis. UML is a complex language, and students have many problems mastering it. Furthermore, UML modelling, like other design tasks, is not a well-defined process. There is no single best solution for a problem, and often there are several alternative solutions for the same requirements.

Although many tutorials, textbooks and other resources on UML are available, we are not aware of any attempt at developing an ITS for UML modelling. However, there has been an attempt [Soller & Lesgold, 2000] at developing a collaborative learning environment for OO design problems using Object Modeling Technique (OMT) – a precursor of UML. The system monitors group members' communication patterns and problem solving actions in order to identify (using machine learning techniques) situations in which students effectively share new knowledge with their peers while solving OO design problems. The system first logs data describing the students' speech acts (e.g. *Request Opinion, Suggest*, and *Apologise*) and actions (e.g. *Student 3 created a new class*). It then collects examples of effective and ineffective knowledge sharing, and constructs two Hidden Markov Models which describe the students' interaction in these two cases. A knowledge sharing example is considered effective if one or more students learn the newly shared knowledge (as shown by a difference in pre-post test performance), and ineffective otherwise. The system dynamically assesses a group's interaction in

the context of the constructed models, and determines when and why the students are having trouble learning the new concepts they share with each other. The system does not evaluate the OMT diagrams and an instructor or intelligent coach's assistance is needed in mediating group knowledge sharing activities. In this regard, even though the system is effective as a collaboration tool, it would probably not be an effective teaching system for a group of novices with the same level of expertise, as it could be common for a group of students to agree on the same flawed argument.

We start by describing the chosen instructional domain in Section 2. Section 3 describes the overall architecture of the system. COLLECT-UML supports problem-solving in two ways. The interface provides information about the domain of instruction, and its design is heavily influenced by the chosen domain. Section 4 discusses how the interface supports the learner while solving problems. Secondly, problem-solving is supported via the feedback that the system provides, which is discussed in Section 5. Section 6 presents the results of two evaluation studies performed. Conclusions are given in the last section.

DIFFICULTIES OF LEARNING OBJECT-ORIENTED MODELLING

An OO approach to software development is now commonly used [Sommerville, 2004], and learning how to develop good quality OO software is a core topic in Computer Science and Software Engineering curricula. When OO first entered the mainstream of software engineering, it served (only) as a programming language paradigm. Subsequently, its influence broadened to provide a paradigm for the design of software, known as Object-Oriented Design (OOD), and broadened yet further to encompass Object-Oriented Analysis (OOA). In OO analysis, the same OO principles for structuring systems are used when performing requirements analysis to represent the concepts, behaviours and relationships found in some problem domain.

OO systems consist of *classes* (with *structure* and *behaviour)*, and *relationships* between them. Relationships have multiplicity, names and can be of different types (association, aggregation, composition, inheritance or dependency). In OOA and OOD, these structures exist independently of any programming language, and consequently many notational systems have been developed for representing OO models without the need for source code. UML is the predominant notation in use today. Software engineering courses that teach OO analysis and design typically do so using UML.

UML consists of many types of diagrams, but *class diagrams* are the most fundamental for OO modelling, as they describe the static structure of an OO system: its classes and relationships. For readers unfamiliar with OO or UML, class diagrams can be viewed as conceptually akin to the *entity-relationship diagrams* used for data modelling, with support for OO features such as inheritance and methods [Booch et al., 1999].

The research described in this paper concentrates on teaching students how to construct a UML class diagram to represent the OO concepts present in informal textual descriptions of software requirements. This type of exercise has been used successfully for several years in our introductory software engineering course, with the support of human tutors. The ITS described in this paper was designed to supplement the existing teaching programme by presenting additional problems and providing automated tutoring.

Let us illustrate the process of designing a class diagram on a simple example. A student is given the following description of the target system:

> *Design a class diagram for a School. A school is known by its name, address, and phone number and has one or more departments. Each department has a name and is assigned a number of instructors. Each instructor has a name and teaches several courses within the department. Each course is known by its name and course ID. A student has a name and student ID and attends a number of courses offered by the department. The school has a number of students and can add students, remove students, add departments and remove departments. Students may enrol in a number of courses, drop courses and transfer credits. Each department can add instructors and remove instructors.*

From the description, the classes *school*, *department*, *student*, *course*, and *instructor* can be identified. The student may start by drawing these classes first. For each class, attributes and methods are described. For example, each department contains a name, and methods to add and remove instructors. All the attributes and methods are explicitly mentioned in the requirements.

The student also needs to identify the relationships between these classes. For example, each school has one or more departments, and this is mentioned in the second sentence of the problem text. The student needs to decide which relationship type would be most appropriate to use. Once all the relationship types are identified, the student needs to determine the multiplicities and names of the relationships.

The UML class diagram for the *School* software system is illustrated in Figure 1. As can be seen from this simple case, there are many things that the student has to know and think about when developing a UML diagram. The student must understand both the basic building blocks available and the restrictions specified on them. In real situations, the text of the problem is likely to be much longer, often ambiguous and incomplete. The student must be able to reason about the requirements and use his/her own world knowledge to make reasonable assumptions. UML modelling is not a well-defined process, and the task is open ended. There is no algorithm to derive the UML class diagram for a given set of requirements. There is no single, best solution for a problem, and often there are several correct solutions for the same requirements. In our experience students typically have many problems learning how to construct good quality OO models.

Although the traditional method of learning UML modelling in a classroom environment may be sufficient as an introduction to the concepts of OO analysis and design, students cannot gain expertise in the domain by attending lectures only. Even if some effort is made to offer students individual help through tutorials, a single tutor must cater for the needs of the entire group of students, and it is inevitable that they obtain only limited personal assistance. Therefore, the existence of a computerized tutor, which would support students in acquiring such design skills, would be highly useful.

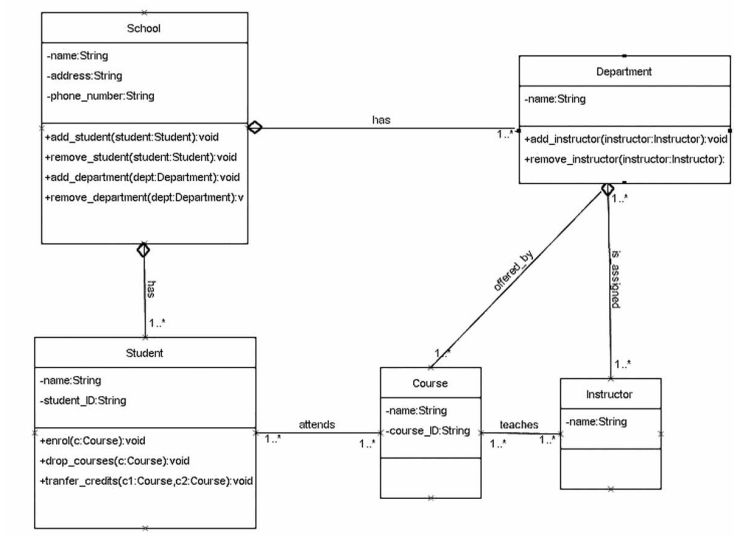


FIGURE 1
The UML Class diagram for School.

THE ARCHITECTURE OF COLLECT-UML

COLLECT-UML is a web-based problem-solving environment, in which students are required to construct UML class diagrams that satisfy a given set of requirements. The system is designed as a complement to classroom teaching and when providing assistance, it assumes that the students are already familiar with the fundamentals of OO software design and UML. It assists students during problem solving and guides students towards a correct solution by providing feedback.

COLLECT-UML has a distributed architecture [Mitrovic, 2003], where the tutoring functionality is distributed between the client and the server. Because the task is very demanding and interactive, it was desirable to perform some pedagogical action on the client, in order to speed up interaction. The client intervenes in situations when the student makes simple syntax errors, such as submitting a diagram with missing component names. The system is implemented in WETAS [Martin & Mitrovic, 2002; 2003], a constraint-based authoring shell. WETAS itself is implemented in Allegro Common Lisp, which provides a development environment with an integrated Web Server [AllegroServe].

The system's components are illustrated in Figure 2. 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. Each action a student performs is sent to the session manager, as it has to link it to the appropriate session and store it in the student's log. Then, the action is sent to the pedagogical module. If the submitted action is a solution to the current problem, the student modeller diagnoses the solution, updates the student model, and sends the result of the diagnosis back to the pedagogical module, which generates appropriate feedback.

COLLECT-UML does not have a problem solver, as developing a general problem solver for UML modelling is extremely difficult. One of the major obstacles that would have to be overcome is natural language processing (NLP), as the problems in the domain are presented using natural language text. However, the NLP problem is far from being solved. Other complexities arise from the nature of the task. There are assumptions that need to be made during the development of UML diagrams. These assumptions are outside the problem description and are dependent on the semantics of the problem itself. Although this obstacle can be avoided by

explicitly specifying these assumptions within the problem description, ascertaining these assumptions is an essential part of the process of constructing a solution and would over-simplify the problems.

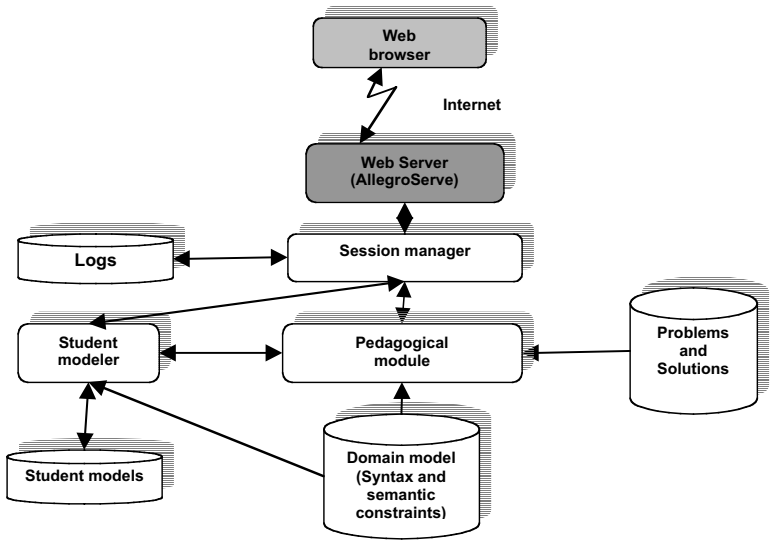


FIGURE 2
The architecture of the system.

Although there is no problem solver, COLLECT-UML is capable of diagnosing students' solutions. The system contains an ideal solution for each problem, which is compared to the student's solution according to the system's domain model, represented as a set of constraints. Constraint-Based Modeling [Ohlsson 1994] is a student modeling 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.* If the relevance condition of a constraint is true (i.e. constraint is relevant to the student's solution being processed), the satisfaction condition should also be true. Otherwise the constraint is violated and the feedback message attached to that constraint is presented to the student.

The system's domain model contains 88 semantic and 45 syntax constraints that describe the basic principles of the domain. Semantic constraints are usually more complex than syntax constraints. In order to develop constraints, we studied material in textbooks, such as [Fowler, 2004], and also used our own experience in teaching UML and OO analysis and design. Figure 3 illustrates two constraints from the UML domain. Constraint 41 is a syntax constraint; it checks that there are some attributes or methods defined for each class in the student's solution (SS). The constraint contains a message which would be given to the student if the constraint is violated. The last two elements of the constraint specify that it covers some aspects of classes, and also identifies the class to which the constraint was applied. Constraint 52 is a semantic constraint. Its relevance condition identifies a superclass and a subclass in the ideal solution, which have the same method defined. Then, the relevance condition looks for a matching class and a superclass in the student's solution, with the same method defined for the superclass. The student's solution is correct if there is a method with the same name defined in the subclass, which overrides the method defined in the superclass.

```
(41
 "Check your classes. Each class must have at least one attribute or method."
 ; Relevance condition
 (match SS CLASSES (?* "@" ?class_tag ?*))
 ; Satisfaction condition
 (or-p (match SS ATTRIBUTES (?* "@" ?tag1 ?attr_name ?c1ass_tag ?*))
       (match SS METHODS (?* "@" ?tag2 ?method_name ?class_tag ?*)))
 "classes"
 (?class_tag))

(52
 "Check your inheritance relationships. Some of your subclasses must override one or more methods
defined in the superclass. The ability of a subclass to override a method in its superclass allows a class to
inherit from a superclass whose behavior is similar, and then override methods as needed."
; Relevance condition
  (and (match IS SUPERCLASSES (?* "@" ?c1_tag ?*))
       (match IS SUBCLASSES (?* "@" ?c2_tag ?c1_tag ?*))
       (match IS METHODS (?* "@" ?m1_tag ?name ?c1_tag ?*))
       (match IS METHODS (?* "@" ?m1_tag ?name2 ?c2_tag ?*))
       (match SS SUPERCLASSES (?* "@" ?c1_tag ?*))
       (match SS SUBCLASSES (?* "@" ?c2_tag ?c1_tag ?*))
       (not-p (test SS ("null" ?c1_tag)))
       (not-p (test SS ("null" ?c2_tag)))
       (match SS METHODS (?* "@" ?m1_tag ?name3 ?c1_tag ?*)))
; Satisfaction condition
   (match SS METHODS (?* "@" ?m1_tag ?name4 ?c2_tag ?*))
   "methods"
   (?c1_tag ?c2_tag ?m1_tag))
```

FIGURE 3
Examples of constraints from COLLECT-UML.

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 generate feedback.

INTERFACE

Students interact with COLLECT-UML via its interface (Figure 4) to view problems, construct UML class diagrams, and view feedback. The top pane contains buttons that allow the student to select a problem, view the history of the session, inspect his/her student model (Figure 5), ask for help, or print the solution. The central part is a Java applet, which shows the problem text and provides the UML modelling workspace. The applet was implemented using Java 1.4.2, and contains 4 packages, 75 Java classes and 6853 lines of code. Feedback is presented on the right, while the bottom part allows the student to select the feedback level, and submit solutions.

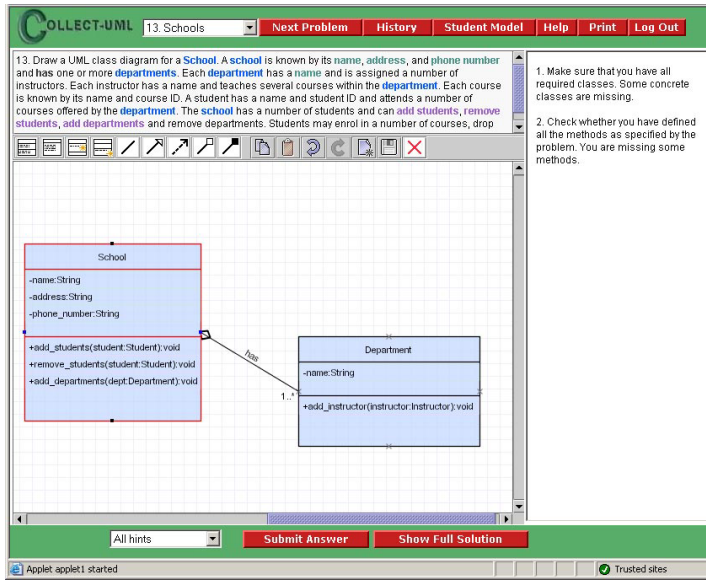


FIGURE 4
The interface of COLLECT-UML.

The interface is not purely a communication medium: it also serves as a means of supporting problem solving. The interface provides information about the domain of study: as can be seen from Figure 4, the applet contains a drawing bar with UML constructs. Students can therefore remind themselves of the basic building blocks to use when drawing UML diagrams. The symbols used for UML modelling are shown in Figure 6. In order to draw a UML diagram, the student selects the appropriate drawing tool from the drawing toolbar and then positions the cursor on the desired place within the drawing area.

COLLECT-UML requires the student to name each newly added construct by using a word/phrase from the problem text as its name. A name can be selected by highlighting a phrase from the problem text. It is not possible to name a construct by typing. This is useful from the point of view of the student modeller for evaluating solutions [Suraweera & Mitrovic, 2002]. There is no standard that is enforced in naming classes, methods, attributes or relationships. Since the names of the components in the student solution may not match the names of construct in the ideal solution (IS), the task of finding correspondence between the constructs of the SS and IS is difficult. This problem is avoided by forcing the student to use the names that come from the problem text directly.

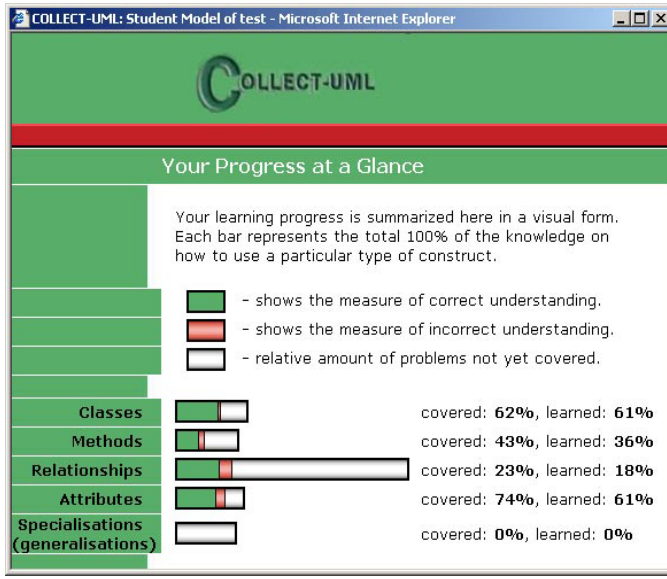


FIGURE 5
The open student model.

This requirement enforces two of the most important practices in software design: using the end-users' language and reflecting on the requirements. By selecting names for various diagram components directly from the problem text, the student has to think about the requirements. The interface highlights the previously selected parts of the problem text that correspond to various types of UML constructs using different colours, making it easier for the student to review how much of the problem has been covered. Subjective evaluation of the system (described later in the paper) showed that several participants pointed out this feature, when asked what they liked in particular about COLLECT-UML.

Currently, the system contains 14 problems, which cover different aspects of UML modeling, and their ideal solutions. Figure 7 shows a sample problem and the internal representation of its ideal solution, which consists of 6 components (i.e. *RELATIONSHIPS, ATTRIBUTES, METHODS, CLASSES, SUPERCLASSES* and *SUBCLASSES*). The problem text is represented internally with embedded tags that specify the mapping to the constructs in the ideal solution. The tags are not visible to the student since they are extracted before the problem is displayed.

Symbol	Component
	Concrete Class
	Interface
	Attribute
	Method
	Association
	Inheritance
	Dependency
	Aggregation
	Composition

FIGURE 6
UML components supported by the system.

The applet saves the solutions submitted by students as XML files, which are converted to internal representation using an XSLT style-sheet. The constraints are applied to the internal representation of the solutions and feedback is given to students, using messages attached to the violated constraints.

```
(13                    ; problem number
10                     ; difficulty
"Draw a UML class diagram for a <E1> School </E1>. A <E1> school </E1> is known by its <E1A1>
name </E1A1>, <E1A2> address </E1A2>, and <E1A3> phone number </E1A3> and <R1> has </R1>
one or more <E2> departments </E2>. Each <E2> department </E2> has a <E2A1> name </E2A1> and
<R2> is assigned </R2> a number of <E3> instructors </E3>. Each <E3> instructor </E3> has a <E3A1>
name </E3A1> and <R3> teaches </R3> several <E4> courses </E4> within the <E2> department </E2>.
Each <E4> course </E4> is known by its <E4A1> name </E4A1> and <E4A2> course ID </E4A2>. A
<E5> student </E5> has a ..."

(("RELATIONSHIPS" "@ R1 aggregation E1 E2 null 1..* null null has @ R2
aggregation E2 E3 null 1..* null null is_assigned @ R3 association E4
E3 1..* 1..* null null teaches ...")
("ATTRIBUTES" "@ E1A1 name E1 String private no @ E1A2 address E1 String
private no @ E1A3 phone_number E1 String private no @ E3A1 name E3
String private no @ E4A1 name E4 String private no @ E4A2 course_ID E4
String private no @ E5A1 name E5 String private no ...")
("METHODS" "@ E1A4 add_student E1 void public no 1 student_ID String null
null null null @ E1A5 emove_student E1 void public no 1 Student_iD
String null null null null ...")
("CLASSES" "@ E1 School concrete @ E3 Instructor concrete @ E4 Course
concrete @ E5 Student concrete @ E2 Department concrete ")
("SUPERCLASSES" "")
("SUBCLASSES" ""))
"13.jpg"
"Schools")
```

FIGURE 7
A sample problem and its ideal solution.

FEEDBACK GENERATION

COLLECT-UML evaluates the student's solution once it is submitted, and provides feedback. During evaluation, the student modeller identifies the constraints that the student has violated. The feedback is offered at five levels of detail: *Simple Feedback*, *Error flag*, *Hint*, *All Hints* and *Full solution*. The first level of feedback simply indicates whether the submitted solution is correct or incorrect. The *Error flag* indicates the type of construct (e.g. class, relationship, method, etc.) that contains the error. *Hint* offers a feedback message generated from the first violated constraint such as "*Make sure that you have all required classes. Some concrete classes are*

missing." A list of feedback messages on all violated constraints is displayed at the *All hints level*. The UML class diagram of the complete solution is displayed when the user clicks on *Show Full Solution* button.

Initially, when the student begins to work on a problem, the feedback level is set to the *Simple Feedback* 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 *Hint* level. In other words, if the student submits the solutions three times the feedback level would reach the *Hint* level, thus incrementally providing more detailed messages. The system was designed to behave in this manner to reduce any frustrations caused by not knowing how to develop UML diagrams. Automatically incrementing the levels of feedback is terminated at the *Hint* level to encourage the student to concentrate on one error at a time rather than all the errors in the solution. The system also gives the student the freedom to manually select any level of feedback according to their needs. This provides a better feeling of control over the system, which may have a positive effect on their perception of the system. In the case when there are several violated constraints and the level of feedback is different from *All hints*, the system will generate the feedback on the first violated constraint. The constraints are ordered in the knowledge base by the human teacher, and that order determines the order in which feedback would be given.

EVALUATION

As the credibility of an ITS can only be gained by proving its effectiveness in a classroom environment or with typical students, we have conducted two evaluation studies on COLLECT-UML, described in this section.

Pilot Study

The pilot study was conducted as a think-aloud protocol in March 2005. The study aimed to discover students' perceptions about various aspects of the system, mainly the quality of feedback messages and the usability of the interface. The participants were 12 postgraduate students enrolled in an Intelligent Tutoring Systems course at the University of Canterbury. At the

time of the study, the participants had completed 50% of the ITS course lectures, and were expected to have a good understanding of ITS. All participants except two were already familiar with UML modelling.

The study was carried out in the form of a think-aloud protocol [Ericsson & Simon, 1984]. This technique is increasingly being used for practical evaluations of computer systems. Although think-aloud methods have traditionally been used mostly in psychological research, they are considered the single most valuable usability engineering method [Nielsen, 1993]. Each participant was asked to verbalise his/her thoughts while performing a UML modelling task using COLLECT-UML. Participants were able to skip the problems without completing them and to return to previous problems. Data was collected from video footages of think-aloud sessions, informal discussions after the session and researcher's observations.

The majority of the participants felt that the interface was nicely designed and the drawing area was big enough for them to work on the problems given. Three participants felt that some of the hints provided by the system were not helpful enough for them to correct their mistakes. The difficulty with the feedback came from the students not being able to interpret given messages. For example, the feedback message of constraint 41 (Figure 3) is "*Check your classes. Each class must have at least one attribute or method*." If the diagram contains many classes, the student might have difficulty identifying the class that the feedback message is relevant for. We have modified the system to highlight the part of the diagram related to the feedback message in red, making it easy for students to localize errors. Two participants also expressed their desire to have access to a glossary and a tutorial on how to use the system. These features will be added to the system in the future.

In order to name a new component (class, attribute, method or relationship), the students were required to highlight phrases from the problem text. Although some participants found this somewhat restrictive initially, they became more comfortable with the interface once they had a chance to experiment with it. Pop-up dialog windows were added to help the users with naming the classes/methods/attributes once they were created.

The majority of the participants felt that the feedback messages helped them to understand the domain concepts that they found difficult. For this study, the feedback level was restricted to *All Hints* only. For the full evaluation study (described in the next section), the system was modified to include the five different levels of feedback, shown in Figure 4.

The constraints were implemented so that they would only check for necessary constructs that the students were supposed to have included in their UML diagrams (i.e. classes, attributes, methods and relationships). Therefore, the participants were allowed to define extra methods for example, if they thought they were needed. This was a feature several participants particularly liked about the system.

Evaluation Study

The evaluation study was carried out at the University of Canterbury in May 2005, after COLLECT-UML was enhanced in the light of the findings from the pilot study. The study involved 38 volunteers from students enrolled in the Introduction to Software Engineering course offered by the Computer Science and Software Engineering department. This second year course teaches UML modelling as outlined by Fowler [2004]. The students learnt UML modelling concepts during two weeks of lectures and had some practice during two weeks of tutorials prior to the study.

The study was conducted in two streams of two-hour laboratory sessions. Each participant sat a pre-test, interacted with the system, and then sat a post-test and filled a user questionnaire. The pre-test and post-test (given in Appendices A and B) each contained four multiple-choice questions, followed by a question where the students were asked to design a simple UML class diagram. Both tests included questions of comparable difficulty, dealing with inheritance and association relationships.

Table 1 presents some general statistics about the study. The participants spent two hours interacting with the system, and solved half of the problems they attempted.

TABLE 1
Some statistics about the study.

	Average	s. d.
Time spent on problem solving (hours)	1.52	0.43
Attempted problems	5.71	2.59
Solved problems	47%	33%
Attempts per problem	7.42	4.76
Pre-test	52%	21%
Post-test	76%	17%

Learning

The most important measure of the ITS effectiveness is the improvement in performance. The average mark on the pre-test for the students who participated in the study was 52% (Table 1). The students' performance on the post-test was significantly better ($t = 2.71$, $p = 4.33E\text{-}08$).

We have also analyzed the log files, in order to identify how students learn the underlying domain concepts. Figure 8 illustrates the probability of violating a constraint plotted against the occasion number for which it was relevant, averaged over all constraints and all participants (*All constraints*). The data points show a regular decrease, which is approximated by a power curve with a close fit of 0.93, thus showing that students do learn constraints over time. The probability of 0.19 for violating a constraint on the first occasion of application has decreased to 0.09 at its tenth occasion, displaying a 47% decrease in probability.

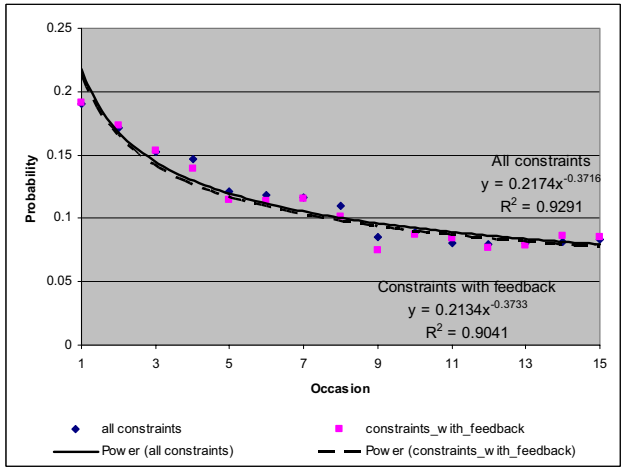


FIGURE 8
Probability of constraint violation.

The other power line in Figure 8 labelled *Constraint with feedback* illustrates the probability of violating a constraint plotted against the occasion number for which it was relevant, averaged over all participants, but only for constraints on which participants obtained specific feedback (i.e. when the participants asked for *Hint* or *All Hints* feedback levels). The student logs show that 53% of the participants asked for the *All Hints* feedback level. The *All constraints* learning curve has 2053 data points at the first occasion, while the

Constraints with feedback curve has 40% less, because students often received feedback other than hints. The learning curve is again very regular, with a very high R^2 fit (0.9), and almost identical initial error probability (0.19) and learning rate (-0.37). We believe that the difference between these two curves is small because the participants often could recover from their errors by being shown where the error is (i.e. by being given the *Error Flag* feedback), or could correct slips by being told that there are problems in their solutions (*Simple feedback*).

We found out that 22 constraints were never violated by the participants, meaning that the students already knew the corresponding domain concepts. These constraints can be divided into several groups: 1) constraints that make sure the name of each class is unique; 2) constraints that check whether classes, attributes, inheritances, compositions and aggregations are represented in the student's solution using appropriate UML constructs; 3) a constraint making sure that each method parameter has a name; 4) a constraint that checks the correct use of dependencies between classes; 5) constraints that check inheritances in students' diagrams, making sure that there are no cycles, and finally 6) a constraint that makes sure each subclass is connected to a superclass.

There were also five constraints that were never satisfied, meaning that the participants did not learn the corresponding domain concepts during the session. The constraints in this group cover aggregation and composition, making sure that the student has used the correct UML construct to represent them. Also this group includes a constraint that checks that multiple inheritance is only specified for interfaces.

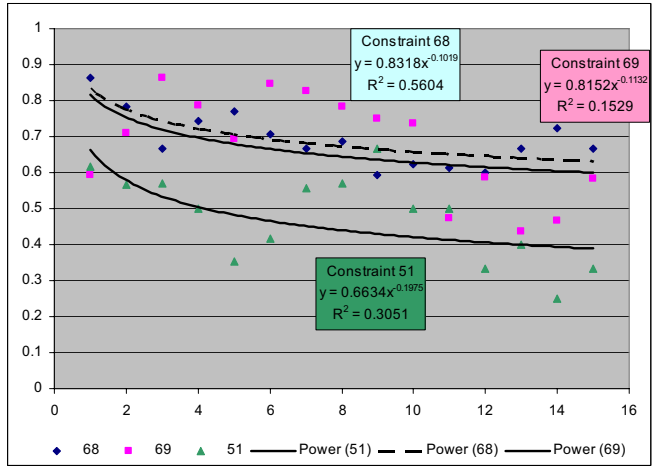


FIGURE 9
Learning curves for three difficult constraints.

We have also looked at learning curves of individual constraints, trying to identify constraints that were especially difficult for our students. Figure 9 illustrates the learning curves for three constraints which were hard for participants. The learning rates for all three constraints are much lower that the ones in Figure 8, as well as the R^2 fit. Constraint 68, the most difficult of the three, checks whether the participant has specified the types of attributes correctly. Constraint 69, the second hardest, checks whether static attributes were specified as such. Finally, constraint 51 checks whether the correct parameters have been specified for methods. In all three cases, the constraints are very specific, and it is likely that the student will focus on these elements of the solution only when the solution is predominantly correct. Furthermore, we have noticed that some problem texts do not contain enough detail for the student to be able to complete the relevant parts of the solution, and therefore we believe that the probability of violating these constraints could be decreased by including more detail in the problem descriptions.

Subjective analysis

All the participants were given a questionnaire (Appendix C) at the end of their session to determine their perceptions of the system. Table 2 presents a summary of the responses. The students found the interface easy to learn and use. 60% of the participants were familiar with UML modelling from lectures and some work, and the rest had previous experience only from the lectures. Most of the participants (65%) responded they would recommend the system to other students.

The mean response when asked to rate how much they learnt by interacting with COLLECT-UML was 2.9, on the scale of 1 (nothing) to 5 (very much). As Table 1 shows, the students spent 1.52 hours on problem solving in average. Some participants indicated that they would have learnt a lot, if they had more time to interact with the system.

Students were offered individualised feedback on their solutions upon submission. The mean rating for the usefulness of feedback was 2.8. 67% of the participants had indicated that they would have liked to see more details in the feedback messages, especially the ones dealing with types of attributes and number of parameters for each method. These two common remarks point out that the problem texts do not contain enough information for students to make correct decisions related to these issues, as we have already noted from the analysis of individual constraints' learning curves.

The problem texts will be modified in future, in order to provide such information. The comments we received on open questions also pointed out several features of the system, which can be improved.

TABLE 2
Mean responses from the user questionnaire for the evaluation study.

	Average	s. d.
Time to learn interface (min.)	10	8
Amount learnt	2.9	0.9
Enjoyment	2.9	1
Ease of using interface	2.8	1
Usefulness of feedback	2.8	1

Discussion

The results show COLLECT-UML is an effective learning environment. The participants achieved significantly higher scores on the post-test, suggesting that they acquired more knowledge in UML modelling. The learning curves also prove that students do learn constraints during problem solving. Subjective evaluation shows that most of the students felt spending more time with the system would have resulted in more learning and that they found the system to be easy to use.

The questionnaire responses suggested that most participants appreciated the feature of being able to view the complete solution and found the hints helpful. Responses showed that the participants found the problems challenging and enjoyed the user friendliness and learning support of the system. There were a few suggestions for further improvement such as including short cut keys, including more details in some of the feedback messages and tool tip boxes, providing tutorials on how to use the system and including general explanations of the full solutions, when they are being displayed to the user.

There were other encouraging signs suggesting that COLLECT-UML was an effective teaching tool. A number of students who participated in the study inquired about the possibility of using COLLECT-UML in their personal time for practicing UML modelling.

CONCLUSIONS

This paper discussed the design and implementation of COLLECT-UML, an ITS developed to assist students learning UML modeling. We presented the system's architecture and functionality, with emphasis on problem-solving support. COLLECT-UML supports problem solving through its interface, which provide domain-specific information and enforces good practices in the domain. The system also provides feedback on students' solutions. COLLECT-UML's effectiveness in teaching UML class diagrams was evaluated in the two classroom experiments. The results of both subjective and objective analysis proved that COLLECT-UML is an effective educational tool. The participants performed significantly better on a post-test after short sessions with the system, and reported that the system was relatively easy to use. The reported studies evaluated the system as a whole; in the future studies, we will focus on a single feature of the system, such as feedback or adaptation.

The goal of future work is to extend the system to support collaborative learning, addressing both collaborative issues and task-oriented issues. The enhancement process will include implementation of the shared workspace, modification of the pedagogical module to support groups of users and designing and implementing the group-modeling component, which will generate feedback messages related to effective collaboration. CBM has been used to effectively present knowledge in several ITSs supporting individual learning. The comprehensive evaluation studies of the multi-user version of the system will provide a measure of the effectiveness of using the CBM technique in intelligent computer-supported collaborative learning environments.

Acknowledgements

The work presented here was supported by the University of Canterbury PhD scholarship awarded to the first author. We thank Konstantin Zakharov for helping with the statistical analyses and Pramudi Suraweera for advice during the earlier stages of constraint development. This research could not have been done without the support of other past and present members of ICTG.

APPENDIX A: PRE-TEST

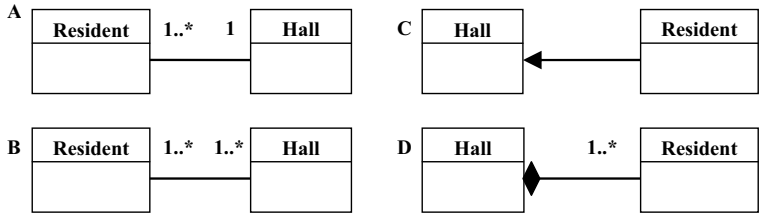
1. System analysts often examine textual requirements descriptions for domain model information. *Nouns* suggest:

A. *Classes*
B. *Attributes*
C. *Methods*
D. *Relationships*
E. *A and B*
F. *C and D*

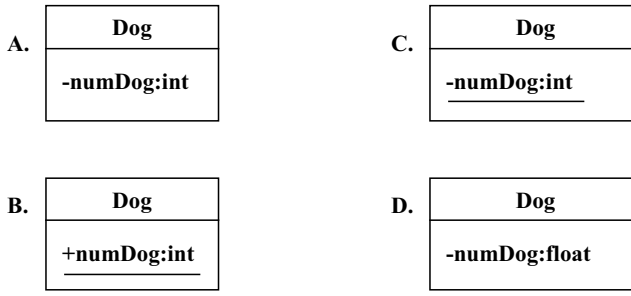
2. Which type of UML relationship would be used when one object merely *invokes methods* of another object?

A. *Inheritance*
B. *Dependency*
C. *Association*
D. *Aggregation*
E. *All of the above*
F. *None of the above*

3. Select the most appropriate option that best describes the given situation: "**Residents live in a student hall**".



4. Which diagram best describes "**numDogs**" attribute? **numDogs** contains a count of the number of Dog instances. This count is accessed only within the class Dog.



Draw a UML class diagram to represent order payments. An order has a number and a price. There are two payment options: credit card and cheque. For each payment option, we store the payment date. For credit card option, the card number and the expiry date are recorded. For cheque option, the cheque number is stored.

APPENDIX B: POST-TEST

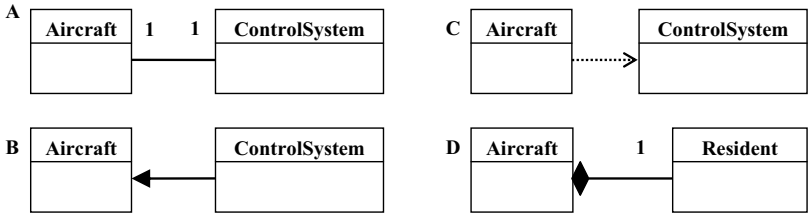
1. System analysts often examine textual requirements descriptions for domain model information. *Verbs* suggest:

A. *Classes*
B. *Attributes*
C. *Methods*
D. *Relationships*
E. *A and B*
F. *C and D*

2.Which type of UML relationship cannot have a relationship name?

A. *Composition*
B. *Association*
C. *Inheritance*
D. *Aggregation*
E. *All of the above*
F. *None of the above*

3. Select the most appropriate option that best describes the given situation: "**an aircraft has a control system.**"



4. In object-oriented software, attributes usually have a visibility of .....… and methods have a visibility of .....…

A. Public, Protected
B. Private, Protected
C. Private, Public
D. Public, Private

5. Draw a UML class diagram to represent customers. A customer has a name and an address and places one or more orders. Each order has a number and a date it was received. A customer can be either personal or corporate. For personal customers, the credit card number is recorded and for corporate customers, the credit card rating and limit are stored.

APPENDIX C: QUESTIONNAIRE

Thank you for using COLLECT-UML. Your feedback will be crucial for further improvements of the system and we would be most grateful, if you could take time to fill in this questionnaire. The questionnaire is anonymous, and you will not be identified as an informant. You may at any time withdraw your participation, including withdrawal of any information you have provided. By completing this questionnaire, however, it will be understood that you have consented to participate in the project and that you consent to publication of the results of the project with the understanding that anonymity will be preserved**.**

1. What is your previous experience with UML modelling? (Please circle one)

A - Only lectures B – Lectures plus some work C – Extensive use

2. How much time did you need to learn about the system's functions? (Please circle one)

(a)	Substantial time (most of the session)
(b)	30 minutes
(c)	10 minutes
(d)	Less than 5 minutes

3. How much did you learn about UML modelling from using the system? (Please circle one)

Nothing				Very much
1	2	3	4	5

4. Did you enjoy learning with COLLECT-UML? (Please circle one and add a comment)

Not at all				Very much
1	2	3	4	5

5. Would you recommend COLLECT-UML to other students? (Please circle one)

   A – No B- Don't know C - Yes

6. Did you find the interface easy to use? (Please circle one and add a comment)

Not at all				Very much
1	2	3	4	5

7. Did you find the feedback from COLLECT-UML useful? (Please circle one and add a comment)

Not at all				Very much
1	2	3	4	5

8. Would you prefer more details in feedback? (Please circle one and comment)

   A – No B- Don't know C - Yes

9 Did you encounter any software problems or system crashes? If yes, please specify
10. What did you like in particular about COLLECT-UML?
11. Is there anything you found frustrating about the system?
12. Do you have any suggestions for improving COLLECT-UML?

REFERENCES

AllegroServe - a Web Application Server. Retrieved 31.5.2005 from http://www.franz.com/

Booch, G., Rumbaugh, J., Jacobson, I. (1999) *The Unified Modelling Language User Guide*. Reading: Addison-Wesley.

Brusilovsky, P., Peylo, C. (2003) Adaptive and Intelligent Web-based Educational Systems. *Artificial Intelligence in Education*, 13, 159-172.

Ericsson, K. A., Simon, H. A. (1984) *Protocol Analysis: Verbal Reports as Data*. Cambridge: MIT Press.

Fowler, M. (2004) *UML Distilled: a Brief Guide to the Standard Object Modelling Language*. Reading: Addison-Wesley, 3rd edition.

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*, pp. 183-187.

Martin, B., Mitrovic, A. (2003) Domain Modeling: Art or Science? In: U. Hoppe, F. Verdejo & J. Kay (ed) *Proc. 11th Int. Conference on Artificial Intelligence in Education*, IOS Press, 183-190.

Mayo, M., Mitrovic, A. (2001) Optimising ITS behaviour with Bayesian networks and decision theory. *Artificial Intelligence in Education*, 12(2), 124-153.

Mitrovic, A. (1998) Learning SQL with a Computerised Tutor. *29th ACM SIGCSE Technical Symposium*, pp.307-311.

Mitrovic, A. (2002). NORMIT, a Web-enabled Tutor for Database Normalization. *Proc. ICCE 2002*, pp.1276-1280.

Mitrovic, A. (2003) An Intelligent SQL Tutor on the Web. *Artificial Intelligence in Education*, 13(2-4), 173-197.

Mitrovic, A. (2005) The Effect of Explaining on Learning: a Case Study with a Data Normalization Tutor. In: C-K Looi, G. McCalla, B. Bredeweg, J. Breuker (eds) *Proc. 12th Int. Conf. Artificial Intelligence in Education*, IOS Press, pp. 499-506.

Mitrovic, A., Ohlsson, S. (1999) Evaluation of a Constraint-based Tutor for a Database Language. *Artificial Intelligence in Education*, 10(3-4), 238-256.

Mitrovic, A., Mayo, M., Suraweera, P., Martin, B. (2001) Constraint-based Tutors: a Success Story. *Proc. 14th Int. Conf. Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Berlin: Springer-Verlag LNAI 2070, pp.931-940.

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.

Nielsen, J. (1993) *Usability Engineering*. San Diego, CA: Academic Press Inc.

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

Soller, A., Lesgold, A. (2000) Knowledge Acquisition for Adaptive Collaborative Learning Environments. *AAAI Fall Symposium: Learning How to Do Things*.

Sommerville, I. (2004) *Software Engineering*. Pearson/Addison-Wesley, 7th ed.

Suraweera, P., Mitrovic, A. (2002) KERMIT: a Constraint-based Tutor for Database Modeling. In: Cerri, S., Gouarderes, G. and Paraguacu, F. (eds.) *Proc. 6th Int. Conf. Intelligent Tutoring Systems*, pp.377-387.

Suraweera, P., Mitrovic, A. (2004) An Intelligent Tutoring System for Entity Relationship Modelling. *Artificial Intelligent in Education*, 14(3-4), 375-417.