ARTICLE IN PRESS

PERGAMON

Expert Systems with Applications xx (0000) 1–16

Expert Systems with Applications

www.elsevier.com/locate/eswa

Building an intelligent system using modern Internet technologies

Goran Šimić[a], Vladan Devedžić[b,*]

[a]Military educational center for signal, computer science and electronic warfare, Veljka Lukića Kurjaka 1, Belgrade 11000, Yugoslavia
[b]FON—School of Business Administration, University of Belgrade, Jove Ilića 154, P.O. Box 52, Belgrade 11000, Yugoslavia

Abstract

This paper is a detailed case study of building Code Tutor, a Web-based intelligent tutoring system (ITS) in the domain of radio communications. It is ontologically founded and was built using CLIPS and Java-based expert system tools, latest integrated graphical CASE tools for software analysis and design, and Java servlets. In Code Tutor, Apache HTTP Server stores and serves static HTML pages, and Apache JServ Java package enables dynamic interpretation of user defined servlet classes and generation of active HTML pages. XML technology is used to generate files that Code Tutor uses to provide recommendations to the learners. Such a rich palette of integrated advanced technologies has greatly alleviated the system design and implementation, and has also led to interesting solutions of a number of problems common to many ITSs. The paper describes these solutions and useful design decisions, and discusses several practical issues related to architectures of intelligent Web-based applications.
© 2003 Published by Elsevier Science Ltd.

Keywords: Expert systems; Intelligent tutoring systems; Knowledge representation; Ontology

UNCORRECTED PROOF

1. Introduction

Using current Internet technology to support learning in the classroom is recently getting much easier and much more feasible than it used to be. If a network of computers or workstations is available in the classroom, it is easy to install and use Apache, Orion, Tomcat, or another Web server on a dedicated server machine to distribute HTML pages generated statically or dynamically by an educational application. Client computers/-workstations should only have an Internet browser. Hardware and software requirements for the client machines are minimal. We acquired experience with this technology when developing an intelligent tutoring system (ITS) called *Code Tutor*.

Code Tutor is a small Web-based tutor designed for fast students' briefing in the area of radio-communication. The learners are telecommunications college students. After they complete a course in radio-communication theory, they are supposed to exercise using expensive radio-station equipment. It is the teachers' responsibility to ensure that the equipment is always in a good condition and that it is used appropriately. There is little time for checking each student's capabilities for independent practical work when the course is over and before the exercises begin. Code Tutor is used instead.

The first version of Code Tutor has been actively used in the classroom since mid-2001. The teachers' opinion is that it is very useful, and the students favor this kind of learning. However, the first version has some limitations. Its expert module is implemented as a rule-based expert system (ES) using an ES shell that was too old, without support for network-based applications. Also, due to these weaknesses of the shell the entire system is limited to 'closed-world' standalone applications, without connections and data interchange with the environment.

These facts have motivated us to build a new version. The entire system is implemented in Java, using *JBuilder 6* tool for developing Web-based and Java-based applications (Borland Corporation, 2001). The system's midlayer is designed using *Rational Rose* tools for software analysis and design (Rational Corporation, 2001). The new Code Tutor integrates many different current technologies:

- *CLIPS*, a tool for building ES (CLIPS, 2002) is used to generate knowledge base files.
- Java-based ES shell *Jess* is used to interpret these files (Jess, 2001; Friedman-Hill, 2002).
- Students communicate with the system through a standard Web browser.

* Corresponding author. Tel.: +381-11-3474123; fax: +381-11-461221.
E-mail address: devedzic@galeb.etf.bg.ac.yu (V. Devedžić).

0957-4174/03/$ - see front matter © 2003 Published by Elsevier Science Ltd.
doi:10.1016/S0957-4174(03)00049-6

ESWA 1072—21/3/2003—19:56—SIVABAL—66295— MODEL 5

ARTICLE IN PRESS

Java™ Servlet technology (Sun Microsystems Corporation, 2001; Hall, 2001a,b) is used to implement the system's interactions with the students.
Apache server (The Apache Software Foundation, 2001a) is used to store static HTML pages.
Apache JServ (The Apache Software Foundation, 2001b) is used to interpret the servlets.
XML technology (W3C, 2001) is used to generate files that Code Tutor uses to provide recommendations to the students.

When developing the new Code Tutor, our goal was not to further the research in Web-based ITS per se. Rather than that, we wanted to re-design and re-develop a *working* system, to make it *work* in a Web-based learning context, to make it *intelligent* at least to an extent, to have both the teachers and the learners actually *use* it, to document its design (made using current Web technologies), and to acquire valuable practical experience for further research and development of intelligent Web-based learning systems. Also, with the system's purpose (students' briefing) in mind we wanted to keep the system small and simple. For the sake of all these goals, in the design of Web-based version of Code Tutor we have deliberately sacrificed some features that are certainly important in Web-based learning systems (e.g. adaptivity and integration of stand-alone domain service systems). In that sense, it should be noted that Code Tutor is actually Web-enabled and Web-ready, intended primarily for use in the classroom, rather than a full-fledged Web-based ITS built with the major goal of using it adaptively over the Web.

The paper describes the new version of Code Tutor thoroughly. The next section briefly overviews related work of other authors relevant for the design and development of the new version, providing the context within which Code Tutor is best understood. Section 3 analyzes the system's use cases and shows its architecture. Design and implementation details are discussed in Section 4, while Section 5 specifically addresses Code Tutor's intelligent behavior. Section 6 covers important issues of Code Tutor's user interface design.

2. Related work

In designing the new version of Code Tutor as a Web-based ITS, our situation was much like the one in which authors of other ITS have been already: we needed to convert a stand-alone ITS to one that operates on the World Wide Web. There were a number of architectural paths from which we might have chosen. Hence, we first studied the scope of Web-based ITS architectures. Extensive discussion on categorization of such architectures by Alpert, Singley and Fairweather (1999) and Mitrović and Hausler (2000) was our starting point. They have found out that architectures of many Web-based ITS are either *centralized* (the application server performs all tutoring functions), or *replicated* (the entire tutor resides in a Java applet that needs to be downloaded and is executed on the student's machine), or *distributed* (tutoring functions are distributed between the client and the server). Each category has some advantages and some disadvantages, described elsewhere as well. For example, Johnson, Shaw, and Ganeshan (1998) discuss feasibility of client-side tutoring deployed in their pedagogical agent called Adele.

Then we studied architectures of a number of specific Web-based ITS, in order to find out about their characteristics and to relate them to our idea of Code Tutor as a Web-based ITS. In the first version of Code Tutor we used rule-based ES reasoning in the GUIDEON style (Clancey, 1983), and we wanted to continue using it in the new, Web-based version. An example of a Web-based ITS that uses an ES is PAT Online, a model-tracing Web-based algebra tutor (Ritter, 1997). We also wanted to deploy XML technology in Code Tutor, and a good example of how to do it is ActiveMath, a generic Web-based learning system that dynamically generates interactive math courses with the content represented in XML-based format and presented to the learner via a standard Web browser (Melis et al., 2001).

The architecture proposed by Retalis and Avgeriou (2002) is important for our work in that it is based upon standards and practices from international standardization bodies, as well as on the practices of well-established software and hypermedia engineering techniques. We also studied the architectures of VALIENT, a Web-based database design learning environment (Hall & Gordon, 1998), and ILESA, a Web-based tutor for linear programming problems (López, Millán, Pérez-de-la-Cruz, & Triguero, 1998). More recently, we also found other ideas that have something in common with Code Tutor—Prentzas, Hatzilygeroudis, and Garofalakis (2002) use hybrid rules for knowledge representation in Web-based tutoring, Rebai and de la Passardiere (2002) try to capture educational metadata for Web-based learning environments, and Ahmad and Lajoie (2001) use an integrated learning model to facilitate Web-based instruction.

Theoretical work of Brusilovsky (1999), as well as ELM-ART (Brusilovsky, Schwartz, & Weber, 1996) and ELM-ART II (Weber & Brusilovsky, 2001) systems for learning programming in LISP cover a number of important issues related to adaptivity of Web-based learning environments, such as providing adaptive navigation support to the learner, links annotation, and adaptive curriculum sequencing. We are aware of the importance of adaptivity of Web-based tutors, but we haven't designed the present version of Code Tutor to have such features.

3. System analysis and architecture

Code Tutor is a client-server learning environment designed as a Web classroom (Fig. 1). Students and teachers

UNCORRECTED PROOF

ARTICLE IN PRESS

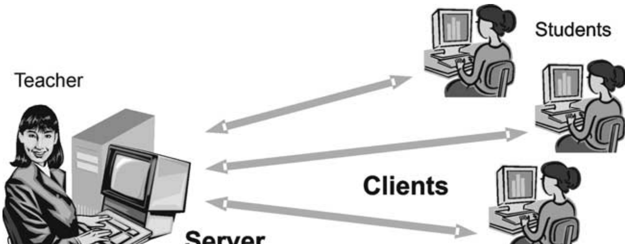


Fig. 1. A Web classroom.

work in a real or in a virtual classroom; in both cases, students learn individually and Web technology connects the server and the client sides.

There are two major actors in the system: the student on the client side, and the teacher on the server side. Therefore, there are two general sets of use cases in Code Tutor—Client-side use cases, and Server-side use cases.

3.1. Client side use cases

There are four modes of students' interaction with Code Tutor, each one being modeled as a distinct use case (see the use-case diagram in Fig. 2).

The student's (user) interaction with the system starts with 'Authentication'—logging in for a new session. 'Learning' begins with selecting one of the chapters to learn from (Fig. 3). Each chapter is composed of several lessons. Currently, lesson is the elementary learning unit (i.e. lessons are not further divided in sections, because we wanted to keep Code Tutor's design simple). The lessons are short, distilled and composed of 3–5 pages. When learning a chapter, the student reads the illustrated lessons. Some of the lesson pages are filled with text and graphics,

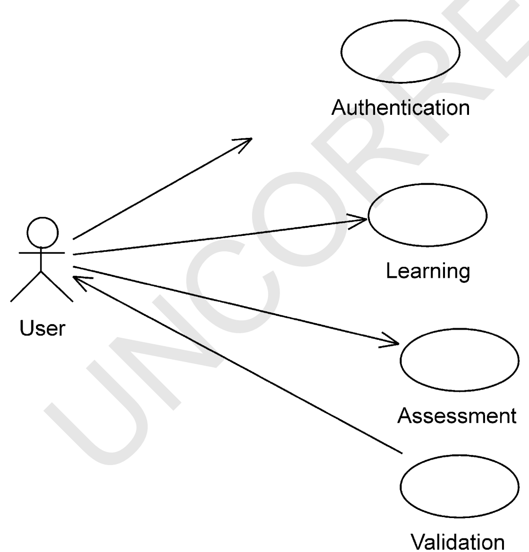


Fig. 2. Student-side use cases.

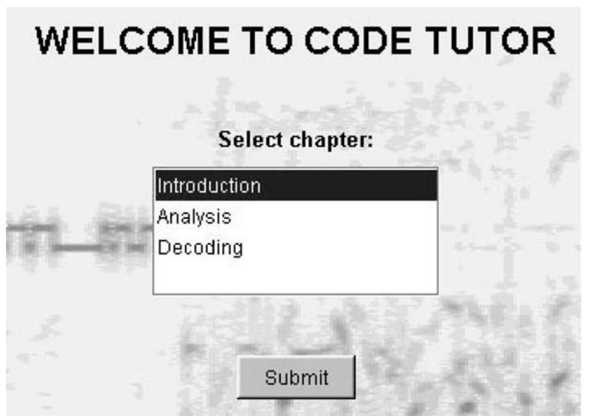


Fig. 3. A fragment of the start servlet HTML page.

and some of them also have audio clips of radio-emissions and/or spectral-analysis diagrams of radio signals (Fig. 25).

Learning a lesson is followed by an appropriate assessment. In 'Assessment', the student answers the questions and submits them (Fig. 4). This event triggers 'Validation', the use case in which Code Tutor checks and updates the student model by estimating the user's knowledge about analyzing and receiving different types of radio-emissions. The system marks each answer given by the student and calculates the student's final score. These marks are aggregated in the student's score (Fig. 12). If the user has at least one negative mark, the system returns him to the beginning of the relevant lesson (Fig. 5).

There are two options for a student who completes a lesson successfully. The first one is to choose another lesson, and the process is the same as described in the previous paragraphs. Code Tutor helps the student by recommending him what to learn next (Section 5.1).

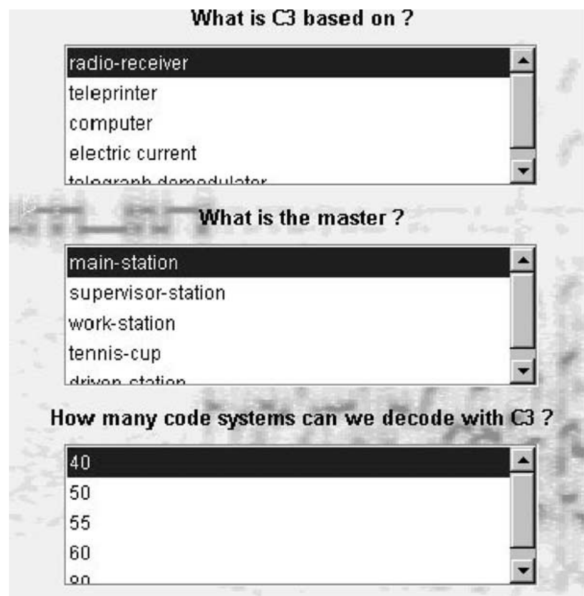


Fig. 4. A test servlet HTML page.

UNCORRECTED PROOF

ARTICLE IN PRESS

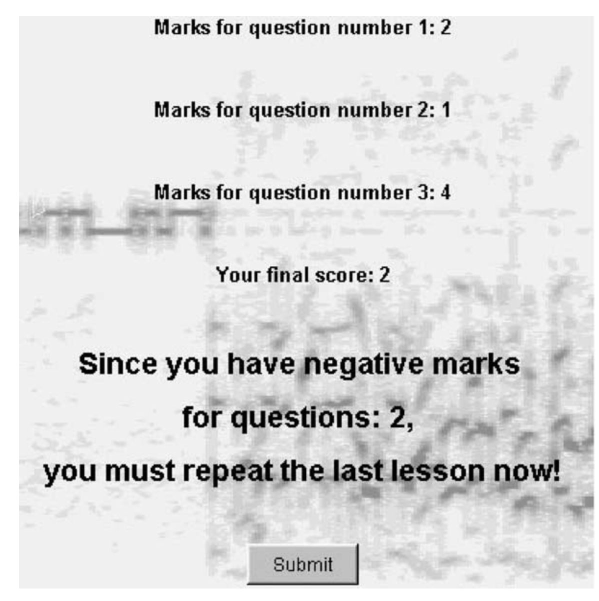


Fig. 5. The fragment of the final servlet HTML page.

The second one is to repeat the same lesson and try to get a better score. In this case, the new score overwrites the previous one.

3.2. Server-side use cases

The teacher is on the server side. His tasks include authentication, starting the server, monitoring the student's sessions, editing the knowledge base and stopping the server (Fig. 6). Some of the tasks are very different from those on the student side—for example, editing the knowledge base (Section 4.2.4). Unlike the student-side GUI, where there is only a browser, the teacher-side GUI has a number of options.

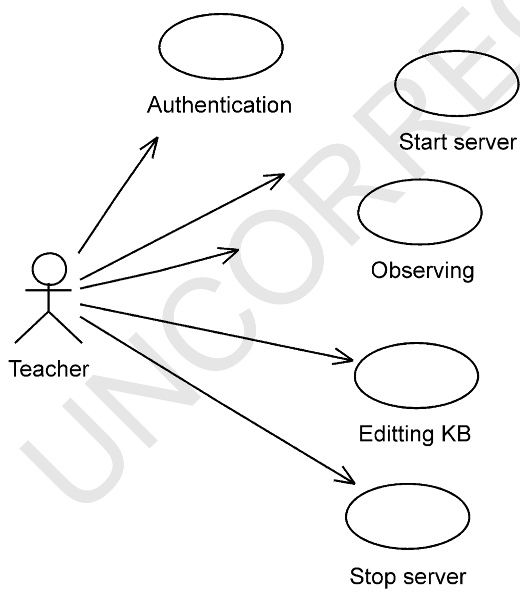


Fig. 6. Teacher-side use cases.

3.2.1. Authentication and starting the server

The teacher's authentication ('Authentication' in Fig. 7) looks much like the student's authentication, but with different authorities. The teacher can start the Web server ('Start server') and only then the tutoring system is available to the students. Of course, starting the server comes after the teacher's authentication.

3.2.2. Monitoring the students' sessions

When the students log in, the server side application creates a new instance of the *User* class for each student. Every such an instance stores the student's user-ID and some other relevant facts (see Section 4.2.3 for details), and is used by the appropriate *Session* object. When the student selects a chapter, the chapter name is also stored in the Session object. In the end of the session, after the tutor evaluates the student's answers, the score will also be stored in the corresponding instance of the User class. The objects used in the session are serializable. The teacher can access all data of each student at any time ('Observing').

3.2.3. Editing the knowledge base

An important and distinct module on the server side of the application is the knowledge-base editor. In the 'Edit KB' use case, the teacher adds, edits or deletes lessons. For example, the teacher can select a chapter and one of its lessons, and then edit the parts he wants to change. When the lesson is modified, the teacher submits the updates (Fig. 16). As the major effect of this operation, Code Tutor generates a script which updates the knowledge base file (CLIPS knowledge base files have the extension.*clp*).

3.2.4. Stopping the server

All the students must finish their sessions before the teacher stops the Web server ('Stop server'). This is necessary for the system in order to update student models (scores) consistently.

3.3. Code Tutor's architecture

From the above use-case analysis, it follows that the system architecture can be represented as in Fig. 7. The following sections further describe some of its details. We opted for a centralized architecture, since Code Tutor had to support the idea of a Web classroom, with a centralized repository of student models and simultaneous support multiple students.

4. Design and implementation

Based on the above use-case diagrams and the system workflow descriptions, it is possible to draw UML sequential diagrams and collaboration diagrams (Fowler, 1997) for specific use cases.

UNCORRECTED PROOF

ARTICLE IN PRESS

449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476

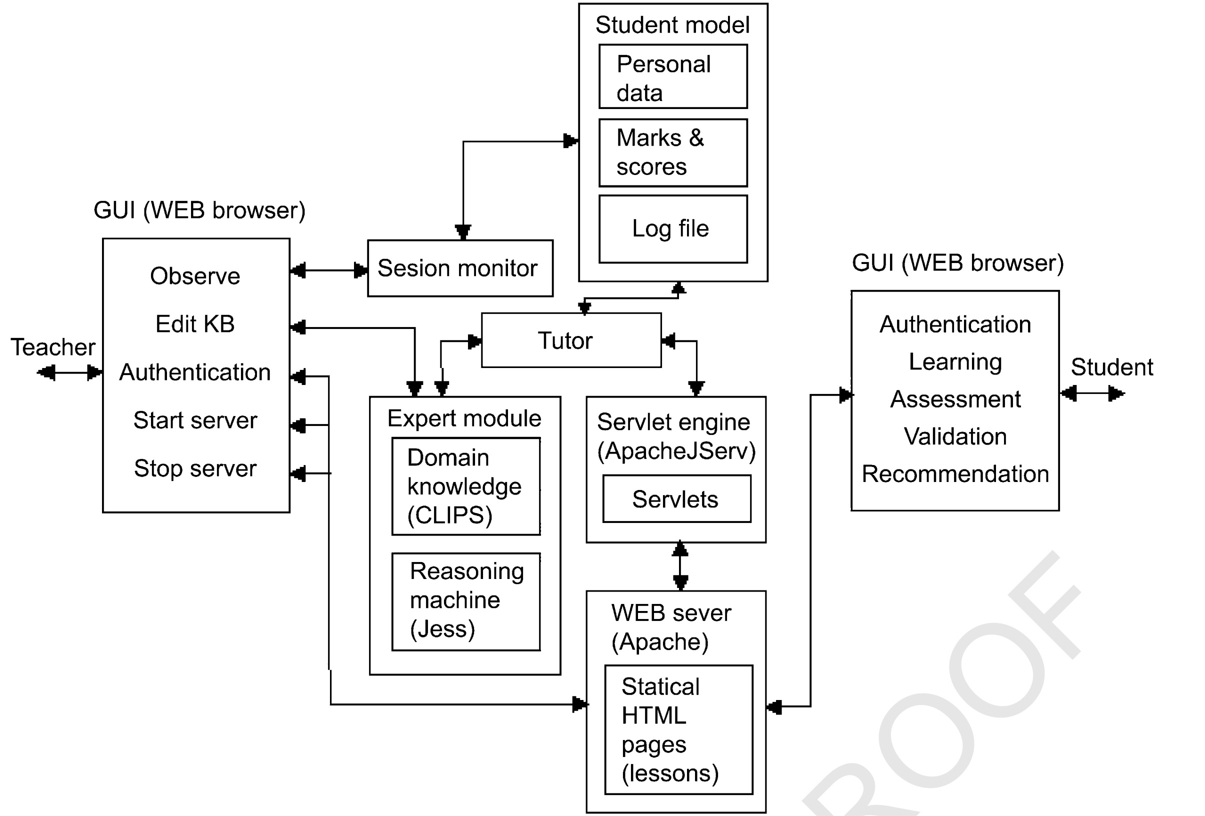


Fig. 7. Code Tutor's overall architecture.

4.1. The student session

4.1.1. Logging in

After the user starts the browser, he enters the first servlet's URL. As a result, an HTML page is generated with the appropriate text fields for the user to log in (the username and the password). During the initialization, the servlet creates an instance of the *Coordinator* class, Fig. 8, and attaches it to the Session object. We designed this using the well-known *GRASP* pattern-controller (Larman, 1999; Eckel, 2000). The initialization is further propagated through the model in the following way. The Coordinator

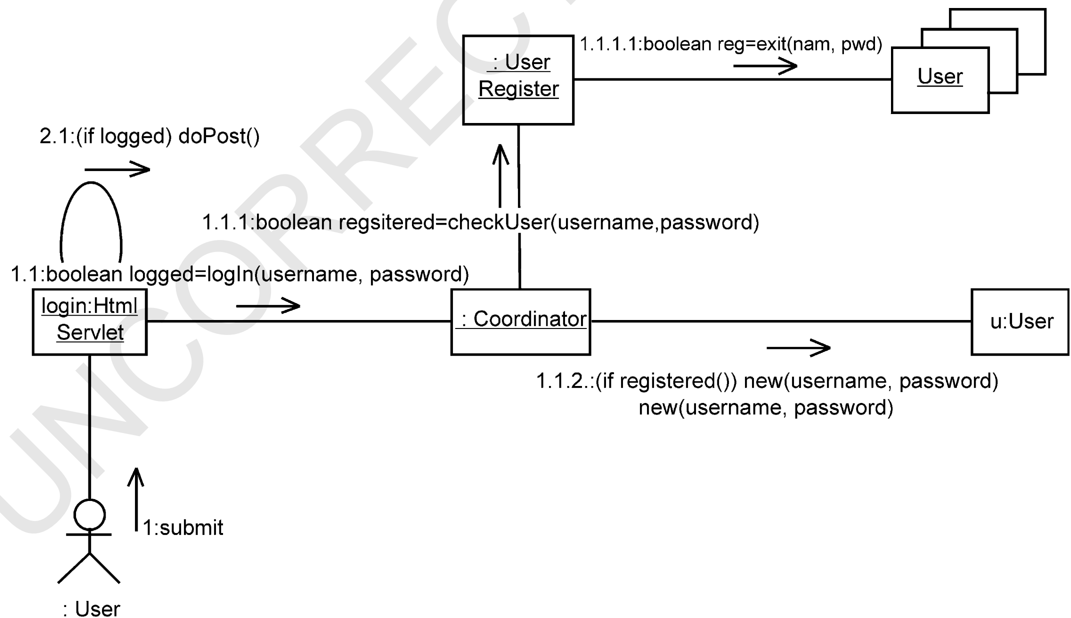


Fig. 8. Collaboration diagram—student login.

477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504
505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560

UNCORRECTED PROOF

ARTICLE IN PRESS

creates an object of the *UserRegister* class. This object is filled in from an external data file. The Coordinator then checks the user's authorities. If the user is registered, the coordinator creates a new User object, and inserts it into the user hash table. Then the Coordinator sends a message to the servlet. This message activates the next servlet.

4.1.2. Learning

The first Learning servlet is also the last one. When the user selects the chapter he wants to learn, the Coordinator remembers the chapter's name for using it later, in the assessment. The students can navigate through the pages and learn. The servlet system references statically inter-linked HTML pages that can contain a number of multimedia contents. In this phase, only the Learning servlet is invoked.

4.1.3. Assessment

To initiate assessment, the system invokes the 'Test' servlet (Fig. 9). The name of the chapter is the argument passed to the Test servlet (through a Session object). In the Coordinator object, this parameter is mapped to the KB file path (*.clp file). The Coordinator takes the path string and passes it to the *executeCommand* method (Fig. 9, step 1.1) that starts the Jess reasoning engine (*Rete* object). This action fills the working memory with the relevant contents (queries and answers) of the .clp file (Section 4.2.4.2). In the following steps (1.2–1.5) Coordinator takes queries and answers from the Jess working memory and uses them to fill in an instance of the *Lesson* class. Lesson objects reference the relevant lessons' contents, questions and answers, etc. (Using a Lesson object instead of just the corresponding questions and answers simplifies other implementation details). Then the Test servlet calls the Coordinator's methods in a loop, in order to get other questions and answers as well, and fills the components of the HTML page.

4.1.4. Validating the student's knowledge

The user selects the answers and submits them. This event puts the answers in the Session object and triggers the validation servlet. In the validation servlet (Fig. 10), the Coordinator evaluates the user's knowledge using the *validateAnswers* method. This is done by sending the answers to the Jess inference engine. It calculates the marks for every answer and finally creates the user's score. The servlet shows these results through an HTML page it generates, and stores them in the user's *Score* object (which is serializable). If the user has at least one negative mark, the session continues with the Learning servlet (which generates the appropriate HTML page).

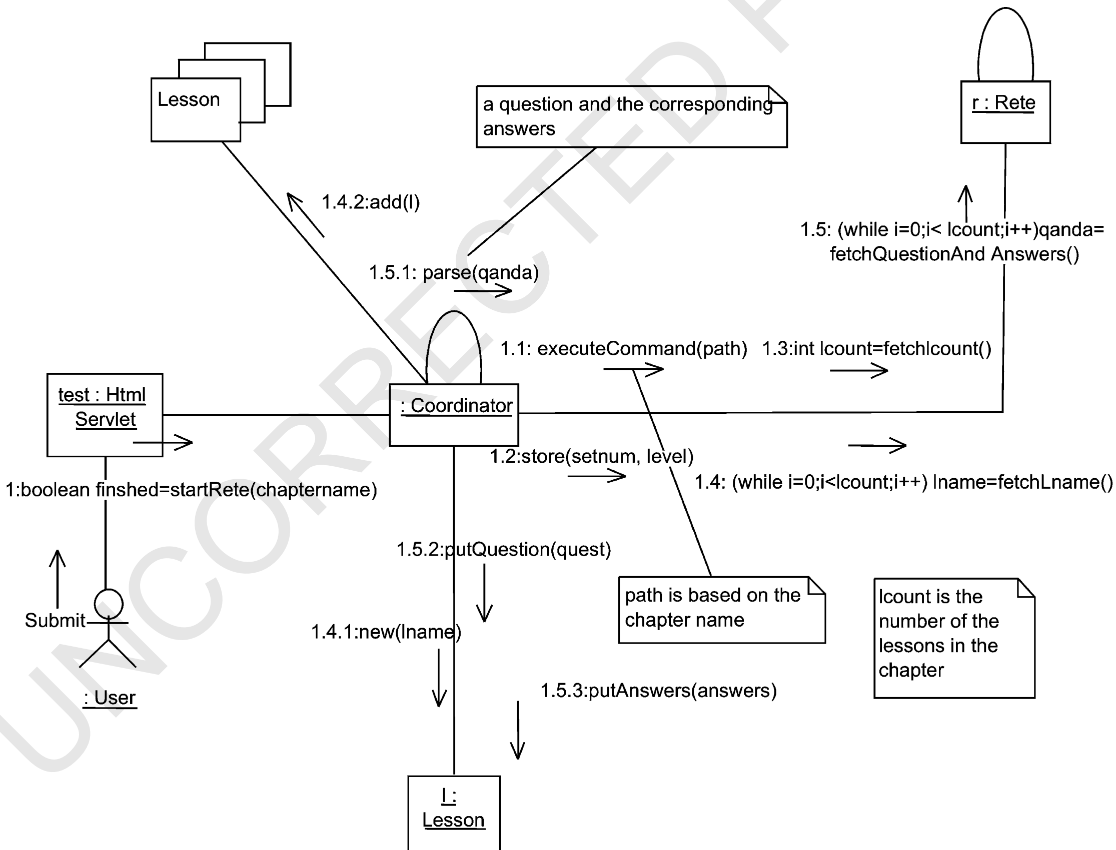


Fig. 9. Collaboration diagram—assessment.

UNCORRECTED PROOF

ARTICLE IN PRESS

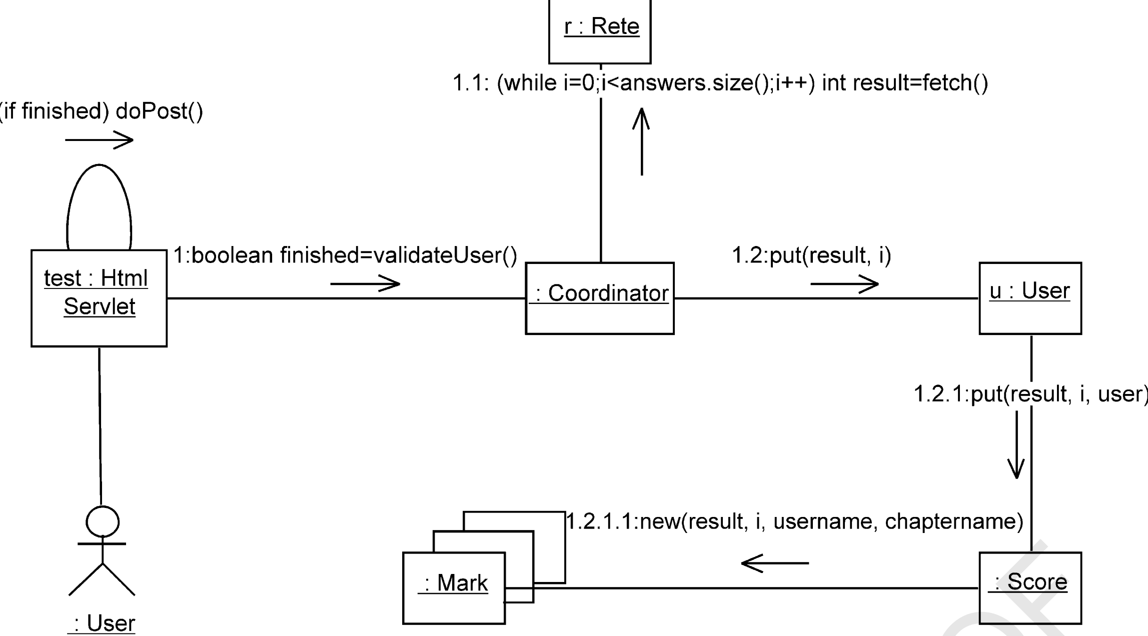


Fig. 10. Collaboration diagram—student knowledge evaluation.

4.2. The teacher session

Code Tutor is designed as a Web classroom with minimal hardware and software requirements, hence a form-based application is run at the teacher's side (not a servlet-based one), for the sake of system flexibility.

4.2.1. Logging in and starting the Web server

In order to start the Web server, the teacher must be logged onto the system. The system only checks the teacher's authorities.

4.2.2. Selecting and editing the students' records

Along with being responsible for modifying the knowledge base, the teacher is also responsible for administrative tasks such as updating the students' records. All available students are stored in the UserRegister object (Section 4.1.1). When the teacher selects the 'Observing' option in the 'Session' menu, the system fills the UserRegister object and shows the *Edit users* dialog (Fig. 11). The students' names, passwords and authorities are stored all in one file. Using a database and a DBMS here for a small number of records would be too costly and would decrease the system's performance, so we designed Code Tutor to read and edit these students' data using a flat text file (XML file), called *Users*. When the teacher selects a user (student) in the *Edit users* dialog, he may modify the student's data (e.g. change the student's personal data or his authorities). In response, the system will update the user's record in the Users file. This can happen only within a teacher session. The user's learning results are stored in a separate file. This second file is changed only within the corresponding student session.

4.2.3. Monitoring user sessions

Parts of the student model storing the user name, password, the selected lesson, the marks from the tests and the final score are all aggregated in the User class. The adopted principles of user validation (Section 4.1.4) suggest three important parts of the User class shown (with their inter-dependencies) in Fig. 12.

The student's personal data are members of the object belonging to the User class. The Score class is an aggregate of the *Mark* class instances. The functionality of some data members in this model is described in Section 5.

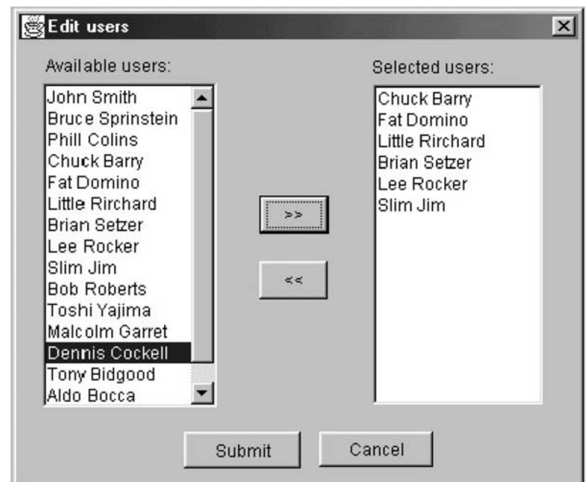


Fig. 11. The beginning of the Edit users dialog.

UNCORRECTED PROOF

ARTICLE IN PRESS

Fig. 12. Important parts of the student model.

Whenever the student starts a new session, Coordinator checks the student's authorities and creates a User object which contains the student's data. These are filled from a specific log file that the system maintains for each student in each of his sessions (Fig. 7). Each time the user logs onto the system, after the user's validation, and when the user logs off the system, Code Tutor serializes all of his personal data, marks, scores, and other data relevant for the history of his learning activities into the appropriate flat-text log file (XML file). Also, every time the student completes some tests the results are added to this file. It is interpreted and displayed only by the teacher-side application. The teacher has 'read only' authorities over the data in the log file and can see them in a specific form (window). If the teacher wants to observe the student's activities, his marks and the score in the log file (Fig. 13), it is not even necessary to run the server.

4.2.4. Editing the knowledge base

4.2.4.1. The ontology. Code Tutor is essentially a rule-based system and its knowledge is contained in *rules*, *chapters*, *lessons*, *sets of questions and answers* (*quandas* or *qandas*), and *globals* (*global variables*). The knowledge structure is represented by the ontology shown in Fig. 14. The main class in the ontology is *KnowledgeBase*. It contains the other concepts of the Code Tutor's knowledge ontology. The instances of the Chapter class are related to the chapters the students can learn. Lessons are parts of a chapter. Each lesson has one or more associated quanda sets. Each quanda set implements a multiple-choice question and includes the question string and a collection of answer objects. An object of the *Answer* class contains the answer string and the appropriate mark.

Global variables are used to store the addresses of the parts of the domain knowledge (Chapter instances) that the students learn. Another role of a global variable is to store the number of lessons in a chapter. *Global* is a class with only put and get methods, like a Java Bean. If the teacher adds a new Lesson, the corresponding global variable (which stores the number of Lessons) is incremented, and vice versa. If the teacher only edits an existing Lesson, this global variable does not change.

4.2.4.2. Knowledge-base script files. At the implementation level, Code Tutor's knowledge base is a collection of KB script files, one for each chapter. This way, the system loads

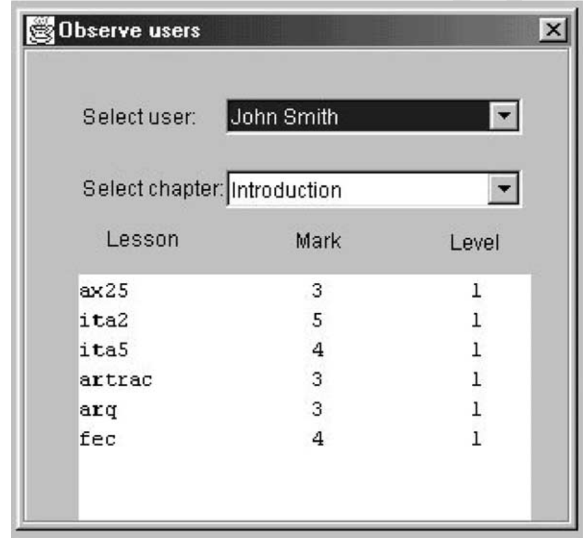


Fig. 13. Showing the student's results.

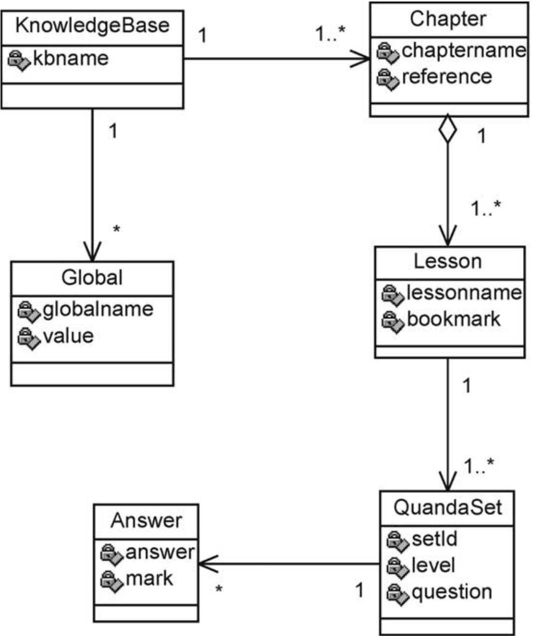


Fig. 14. The ontology of Code Tutor's knowledge.

ARTICLE IN PRESS

only the necessary facts and rules into the working memory, and it's reasoning is more focused. Also, the teacher may edit the part of the knowledge base corresponding to a certain chapter, while the students simultaneously use the other KB files. The changes in one KB script do not affect the others.

In a KB script file, a chapter corresponds to the CLIPS *deftemplate* ⟨*chaptername*⟩ declaration, and lessons correspond to the CLIPS *slot* ⟨*lessonname*⟩ declarations, Fig. 15. The system's interaction with the student is designed and implemented in rules, and is reflected in the KB script files. Code Tutor has five types of rules:

- the start rule (initiates the operation of the reasoning machine);
- the queries-and-answers rule (displays questions to the student);
- the receive-answer rule (collects the student's input when he is answering questions);
- the working rules (evaluate the student's answers); and
- the goal rule (calculates the student's final score).

Internally, two kinds of slots correspond to each lesson: the slots used by the receive-answer rule to get the student's input (the answer), and the slots used by the working rules to mark and quantify the student's answers and determine the final score. Each mark-slot corresponds to an answer-slot. These design details are hidden from the teacher (Section 4.2.4.3).

The structure of a KB script file looks like the one shown in Fig. 15. It contains the definitions of the concepts specified in the ontology. The global variables are on top (section (A)). The chapters and lessons are below (B). Everything under the chapters (C)–(E) are the rules: the start, queries-and-answers, and receive-answer rules (C), the working rules (D) and the goal (final) rule (E).

4.2.4.3. Editing lessons. When the teacher edits a lesson, things look like in Fig. 16. When the lesson editing is completed, the system generates a new KB script. This means that the rules containing references to any data item that the teacher has modified in the lesson will be changed as well.

4.2.4.4. What about the rules?. Where are the rules in the ontology, i.e. in its conceptual diagram shown in Fig. 14? Nowhere in the Code Tutor ontology, all knowledge is represented in the question-and-answers sets (quandas). However, since the Jess inference engine interprets rules, not quandas, the rules are automatically generated from quandas and are put in the *.clp file that Jess interprets. Whenever the teacher adds/removes/edits a lesson, or updates domain knowledge in any other way, the changes are reflected in the corresponding rules in the knowledge base automatically. This approach gives the teacher the possibility to make changes in the knowledge base easily, and requires no knowledge of programming or the CLIPS' scripting language. This way, all parts of the knowledge base have low coupling and high cohesion, and the entire knowledge base scales well for different tutoring tasks.

4.2.5. Rules design details

4.2.5.1. The start and queries-and-answers rules. Upon the start of a student session, the Coordinator object takes the level and set numbers (Section 5 for the meaning of these numbers) from the User object and sends them to the Jess reasoning machine (the Rete instance). Then the queries-and-answers rule displays questions to the user (student), through the objects of the middle layer (corresponding to the concepts in the ontology) and front-end servlets. This communication runs via global variables (also called common variables, or routers). The sample code fragment below describes this (Fig. 17). The queries-and-answers rule calls the *store* method defined in the CLIPS shell (which we used to represent the rules), and sets the values of the corresponding *LESSON** and *QANDA** (Question AND Answers) variables. Each *LESSON** variable is set to the string-value representing the corresponding lesson name. Coordinator uses this name to create the new Lesson instance. Each *QANDA** variable is set to the corresponding multiple-choice question-and-answers string. Fig. 18 shows how a dedicated method of the Coordinator class gets and customizes a *QANDA** value. The Coordinator object fills the string *sqanda*, tokenizes the string and puts the tokens in a vector, which passes them to the corresponding Chapter object. The Chapter object creates and fills the Lesson object with the question-and-answers, which are extracted from the vector (see the collaboration diagram in Fig. 9).

4.2.5.2. The receive-answer rule. The receive-answer rule does not have a premise (Fig. 19)—it fires and waits for the student input (the selected answer). This way the receive-answer rule behaves as a listener of the reasoning machine. The mid-layer Coordinator object sends the student's answers to the reasoning machine (Fig. 20). The receive-answer rule accepts them, calling CLIPS' *fetch* method, and keeps the resulting states of global variables (*?*u** in Fig. 19), using CLIPS' *set-reset-globals nil*, until the answers are marked.

4.2.5.3. The working rules. After the receive-answer rule fills the appropriate answer-lessons, the working rules have the necessary firing conditions. Every working rule has its premise and its action part ('then' clause), Fig. 21. The premise uses an answer-slot that contains the student's answer, and the action part contains the corresponding mark-slot and sets the marks for the answer. The identifiers of the answer and the appropriate mark-slots differ only in that the mark-slot's identifier starts with the prefix 'o'. The number of working rules is the same as the number of the

UNCORRECTED PROOF

ARTICLE IN PRESS


```
(defglobal ?*i* = 3)                                                    A
....
(deftemplate Setup                                                      B
        (slot setnum (type INTEGER))
        (slot level (type INTEGER) (default 1)))
(deftemplate Introduction
        (slot basics (type STRING))
        (slot master (type STRING))
        (slot numdec (type STRING))
        (slot obasics (type INTEGER))
        (slot omaster (type INTEGER))
        (slot onumdec (type INTEGER))
        (slot omarkintro (type FLOAT)))

(defrule start                                                          C
      =>
      (bind ?*a* (assert (Setup(setnum(fetch SETNUM))(level(fetchLEVEL)))))
      (set-reset-globals nil)
)

(defrule queriesandanswers1
        ?fact1 <- (Setup(setnum ?s)(level ?q))
        (test (eq ?s 1))
        (test (eq ?q 1))
        =>
        (store LESSONCOUNT ?*i*)
        (store LESSON1 "basics")
        (store QANDA1 "What is C3 based on?/radio-receiver/teleprinter/computer/electric
current/FSK demodulator")
...
(defrule receiveanswer
  =>
  (bind ?*u* (assert (Introduction(basics (fetch ANSWER1))(master(fetch
ANSWER2))(numdec(fetch ANSWER3)))))
  (set-reset-globals nil)
)

(defrule validate_basics51                                              D
  ?fact11 <- (Introduction(basics "computer")(obasics nil))
  ?fact12 <- (Setup(setnum ?s)(level ?q))
  (test (eq ?s 1))
  (test (eq ?q 1))
  =>
  (modify ?fact11(obasics 5))
  (bind ?*u* ?fact11)
  (set-reset-globals nil)
)
(defrule validate_basics41
  ?fact11 <- (Introduction(basics "FSK demodulator")(obasics nil))
  ?fact12 <- (Setup(setnum ?s)(level ?q))
  (test (eq ?s 1))
  (test (eq ?q 1))
  =>
  (modify ?fact1(obasics 4))
  (bind ?*u* ?fact1)
  (set-reset-globals nil)
...
(defrule validate                                                       E
  ?fact1 <- (Introduction(obasics ?x)(omaster ?y)(onumdec ?z)(omarkintro ?f))
  (test (neq ?x nil))
  (test (neq ?y nil))
  (test (neq ?z nil))
  =>
  (bind ?f(round(/ (+ ?x ?y ?z) ?*i*)))

  (bind ?*u* ?fact1)
  (set-reset-globals nil)
  (store NUMMARKS (+ 1 ?*i*))
  (store O1 ?x)
  (store O2 ?y)
  (store O3 ?z)
  (store O4 ?f)
)
```

Fig. 15. The structure of a KB script file.

UNCORRECTED PROOF

Edit questions and answers

Hyperlink: ../../ctutor_a/Intro.htm Select chapter: Introduction

Bookmark #master Select lesson: master Level: 1 Set number: 1

Edit question: What is the master ?

Edit answer1: tennis-cup Mark: 1

Edit answer2: radio-station Mark: 2

Edit answer3: work-station Mark: 3

Edit answer4: main station Mark: 4

Edit answer5: supervisor station Mark: 5

Submit Cancel

Fig. 16. An edit-lesson dialog.

```
(defrule queriesandanswers1
        ...
        =>
        (store LESSONCOUNT ?*i*)
        (store LESSON1 "basics")
        (store QANDA1 "What is C3 based on?/radio-receiver/.../... ")
        (store LESSON2 "master")
        (store QANDA2 "What is the master?/work-station/.../.../...")
        (store LESSON3 "numdecod")
        (store QANDA3 "How many code systems can decode with
C3?/.../.../...")
        (bind  ?*a* ?fact)
       (set-reset-globals nil))
```

Fig. 17. A queries-and-answers rule script.

```
for(int i=0;i<ilessoncount;i++){
        //ES passes lesson name via LESSONn (used to create new lesson)
        String lessonnkey="LESSON"+(i+1);
        Value vlessonn=(r.fetch(lessonnkey));
        //Value type is defined in the Jess package
        String lessonname=vlessonn.stringValue(r.getGlobalContext());
        String key="QANDA"+(i+1);
        //var that receives the query&answer string for the lesson)
        Value vqa=(r.fetch(key));
        String sqanda=vqa.stringValue(r.getGlobalContext());
        //Conversion into the standard Java type
       ....
```

Fig. 18. The Coordinator object receives queries-and-answers (quandas).

RRECTED PRO

ARTICLE IN PRESS


```
defrule receiveanswer
  =>
  (bind ?*u* (assert (Introduction(basics(fetch ANSWER1))(master(fetch
ANSWER2))(numdec(fetch ANSWER3))
  (set-reset-globals nil)
)
```

Fig. 19. The receive-answer rule.

```
....
for(int i=0;i<answ.size();i++){
      String sanswer=(String)answ.elementAt(i);
      if (sanswer!=null){
           String temp="ANSWER"+(i+1);
           r.store(temp, new Value(sanswer, RU.STRING));
         ....
```

Fig. 20. The Coordinator object sends the student's answers.

student's answers. This means that every working rule's role is to evaluate an appropriate answer.

4.2.5.4. The goal rule. The processing of the reasoning machine is finished when the goal rule is fired. There is only one goal rule in the knowledge base (Fig. 22). Like the working rules, the goal rule has a premise and an action part. The main condition for the goal rule firing is that the student is validated (i.e. the student's knowledge is evaluated) for all the questions in a certain lesson. The goal rule's task is to calculate the student's final score as the average of all marks received in a certain lesson.

After the goal rule fires, all the marks and the final score are sent to the mid-layer objects, which store the data in the user log file. These marks are shown to the student through the final servlet. The mid-layer objects check for negative marks. If the student has any, the system returns him to the beginning of the lesson. On the other hand, the student can read some other lessons and test himself in them. The teacher can see the student's success.

5. Code Tutor's intelligent behavior

It may seen from the previous sections that Code Tutor operates as a procedural program—the student selects a chapter to learn from, then the system shows different HTML pages with the text, graphics and audio, then the student answers the questions, and finally the system marks the student's performance. However, intelligent interactions between Code Tutor and the student continue from this point onward.

5.1. Recommendations to the student

One manifestation of the system's intelligence is the recommendation to the student (what to learn). For example, after evaluating the student's performance, Code Tutor can suggest the student to read a certain lesson again. This recommendation is based on the results of the assessment. There are separate knowledge-base modules (separate files) for this purpose. The system takes the assessment results

```
....
(defrule validate_master3
      ?fact1 <- (Introduction(master "work-station")(omaster nil)(level
1)(setnum 1))
      ?fact12 <- (Start(setnum ?s)(level ?q))
      (test (eq ?s 1))
      (test (eq ?q 1))
      =>
      (modify ?fact1(omaster 3))
      (bind ?*u* ?fact))
         ....
```

Fig. 21. A fragment of a working rule.

UNCORRECTED PROOF

ARTICLE IN PRESS


```
....
(defrule final
      ?fact <- (Introduction(obasics ?x)(omaster ?y)(onumdec
?z)(omarkintro ?f))
      (test (neq ?x nil))
      (test (neq ?y nil))
      (test (neq ?z nil))
      =>
      (bind ?f(round(/ (+ ?x ?y ?z) ?*i*)))
      (bind ?*u* ?fact)
      (set-reset-globals nil)
      (store NUMMARKS (+ 1 ?*i*))
      (store O1 ?x)
      (store O2 ?y)
      (store O3 ?z)
      (store O4 ?f)
)
```

Fig. 22. The goal rule sample.

(via the User object) and searches for the worst mark. If there are two or more marks with the same value, the system uses the first one it finds. Then it takes the worst mark's lesson-name and finds the mapped reference to the HTML page (paragraph). The final result of this reasoning process (Fig. 23) is a servlet-generated link, which is the reference to a certain HTML page (or paragraph). The student can click the recommendation link and the session is continued by reading the referenced chapter/lesson. Otherwise, if the student is satisfied then submitting the results ends the session.

The recommendation directs the student to an appropriate HTML page or a paragraph in the lesson. The mapping is done through a hash table called *References*. This hash table is maintained by the Coordinator object and is filled at the moment the system starts. XML files are used to store the references (Fig. 24). Lessons and chapters contain the reference attribute. Each chapter has a reference to a page (link) or a paragraph in the page (link + bookmark). The system reads the XML file and fills the hash table. The hash table key is the lesson name. The servlet that generates the score page evaluates the student's marks and generates the reference (link) with the corresponding lesson name. The session continues if the student goes to this referenced page. The student's score is remembered before Code Tutor shows the final score page (in the method of the score servlet).

5.2. Multilevel learning

Multilevel learning is another manifestation of Code Tutor's intelligent behavior. Expert knowledge in Code

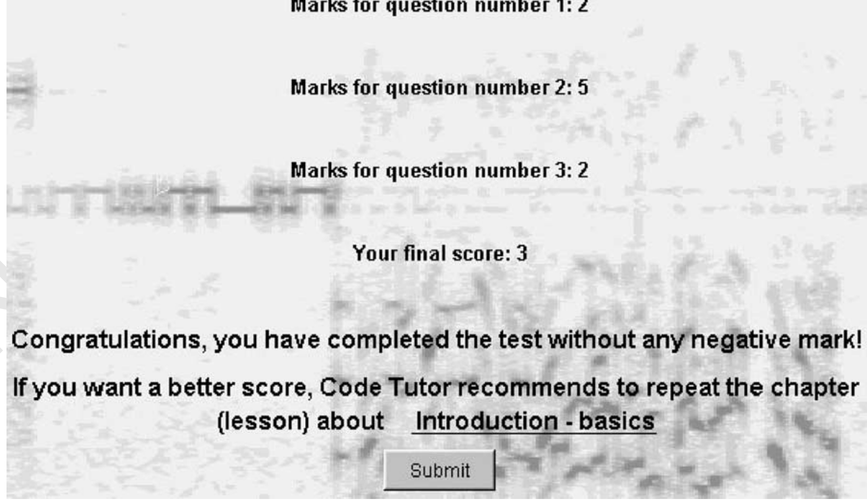


Fig. 23. Example of recommendation to the student who passed the test.

ARTICLE IN PRESS


```
<c3ontology>
....
 <chapter name="Introduction" href="../../ctutor_a/introduction.htm">
  <lesson name="master" href="#master">
   <qanda level="1" setid="1">
    <question>What is the master?</question>
    <answer mark="1">electric current</answer>
    .....
   </qanda>
   .....
  </lesson>
  .....
 </chapter>
 ...
</c3ontology>
```

Fig. 24. A fragment of the XML file storing the references.

Tutor is structured in two levels: the basic and the advanced level. A novice student has to learn the basic concepts of the radio-telecommunications first. If he passes the basic lesson test without negative marks, he can study the corresponding advanced-level lesson.

5.3. The testing multiset

Experience with Code Tutor has shown the students' tendency to remember the questions and wrong answers in order to pass the test in the next step. With this inadequate motivation (not to learn, but to get the positive mark), the student quickly completes reading the lesson and immediately starts the test. He is certain that he will succeed this time. Hence, one set of questions per lesson is not enough, and we have developed a number of question sets per lesson. This means that the student who gets at least one negative mark will be asked a new set of questions when repeating the same lesson. This forces the student to read and learn the lesson with attention. For this reason the system is a little more complicated.

The role of the tutor is very important for students' motivation. The tutor's goal is to help the students understand the domain and acquire the knowledge they need. If the teacher is too rigorous, and he does not understand the students' needs, the motivation drops. Code Tutor solves this problem by repeating the test, and by giving recommendations to the student (Section 5.1). Browsing freely through the lesson pages reassures the student that he can succeed. Also the multilevel learning gives the student a gradual learning satisfaction.

6. Interface design

One of the most important things in a tutoring system is to keep the students focused and motivated during their learning sessions. As with human teachers, the more interactions with the learner result in higher student's focus and better performance. This requires a multilevel approach. The basic student perception (visual and audio perception) must be considered, as well as his cognitive capabilities (reading, learning, interpreting, implying the knowledge). All of the interactions between the student and an artificial tutor happen through the user interface.

6.1. Lesson contents and design

As mentioned above, the chapters contain text, graphics, and audio clips, which are accessible through the Web browser (as HTML pages). The background of the pages is designed as a dimly colored picture reflecting the context of the lesson—spectral or time diagrams of some radio-emissions—but they do not distract the student from learning (Fig. 25). Taken from the earlier version of Code Tutor, each lesson has a unique background. This can help the student remember, where he can find certain data. Like the background, the color of the text, the font size and the font face (weight) increase the lesson readability and help the student learn the lesson with less effort.

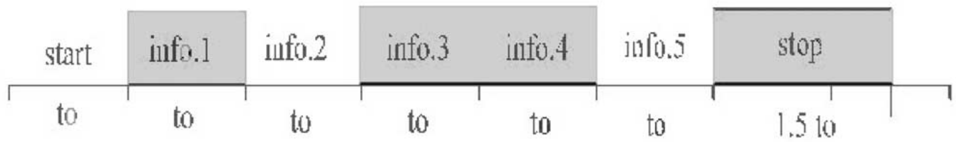
When perceiving the real world is not possible, text can be the best way to represent it. In some cases this is the only way to do it. The main benefit of using text is that the students can read it as quickly as they need to. Also, the students can repeatedly read the same text as many times as they want. On the other hand, long text sequences make the students loose attention. Moreover, the reader can interpret the text in his mind in a wrong way. This is analogous to representing an object in a relational database (a real thing or being is represented only as a set of data items).

Therefore, the lessons in Code Tutor come with a lot of illustrations and audio samples. Every picture and audio clip is related to the appropriate text paragraphs. The keywords in the lessons are highlighted and tagged to detailed

UNCORRECTED PROOF

3. FIB ITA2 audio clip spectral diagram

FIBITA 2 is the oldest kind of FSK transmission that is still in use. The protocol implies 1 start bit, 5 information bits, and 1 stop bit. The stop bit's length can be equivalent to the length of 1, 1.5, or 2 lengths of an information bit. Hence each character's length is 7, 7.5, or 8 bits. See the figure at the bottom.



Usual Baud rates are 50, 75, and 100 Bd. Bit inversion is quite common. One of the program's options is the possibility to use 32 masks (masking templates) that we can set up easily from the keyboard; this is indicated by displaying the text "MASK" and the selected mask templete number next to it. Autoclassifier's performance in this task is excellent, even in detecting the polarization mode. Line intervals can differ a lot; older systems have larger intervals. The "U" key is very useful with ITA-2 transmission, especially when receiving longer telegrams (with alphabetical or numeric characters) where it is used to block switching from letters to digits or vice versa.

Fig. 25. A sample of lesson design.

explanations. If the students forget something, the system lets them browse through the lower layer lessons.

Another important aspect is the complexity of the lesson pages. The pages overloaded with text, pictures, and control components (commands, radio buttons, check boxes, etc.) are tiresome. Hence the lessons in Code Tutor are divided in a number of HTML pages, but there are fewer items in one page. Also, the lessons' ontology is layered—the student can always return to the lower-level pages, but cannot go to the upper-level ones before completing the lower-level pages with a positive score.

6.2. Design of test pages

There are many different techniques to evaluate the student's knowledge. As the lessons are divided in two levels, there is the same number of test levels (although the system is actually designed to support more than two levels). In all tests the student either answers multiple-choice questions (if a text-based answer is required) or enters numeric values in text fields.

For the sake of design simplicity, the lower-level tests are designed to evaluate the student's answers to each question are categorized as either correct or incorrect. The system notices when the student passes the test. The student can then read lessons from the higher level and can undergo the corresponding higher-level tests. Higher-level tests let the system evaluate the student's knowledge on the scale from one to five.

Code Tutor does not constrain the number of questions per lesson. We recommend the following rule: it is better to have more lessons and only a few questions per lesson. However, this rule contradicts the fact that evaluation is always better if there are more questions. In practice, the teacher's experience decides what the best number of questions is.

7. Conclusions

The present version of Code Tutor is developed using the latest technology. The major advantage of the new version of Code Tutor over the first one is its design flexibility. Big technological changes would not mean big changes in Code Tutor. The changes would only affect parts of the system. The object-oriented design contributes to the system extensibility.

One of the first steps in further development of Code Tutor is to make it capable of learning by itself. This means that the teacher should be able to easily mark up the lessons (to mark up the keywords and give the names to the chapters) and put them in the knowledge base (to put the lesson file(s) in a selected directory and insert the file reference into the system accordingly). After that, the system should be able to interlink the teacher's input and generate the appropriate XML file(s). Also, the system should generate lesson templates and the appropriate forms based on the keywords. The marked-up facts in the lesson can be used to generate the rules. We discuss these ideas further in (Devedžić, 2002, 2003).

It should be stressed that, in order to use Code Tutor that way, the teacher should understand the domain and the basic principles of lesson marking, but is not required to understand details of the system design.

UNCORRECTED PROOF

ARTICLE IN PRESS

7. Uncited reference

Sandia National Laboratories, 2002. The World-Wide Web Consortium, 2001.

References

Ahmad, A., & Lajoie, S. (2001). The integrated learning model: a design experiment in web instruction. *Proceedings of the 10th World Conference on Artificial Intelligence in Education AIED'01*, pp. 354–364.

Alpert, S. R., Singley, M. K., & Fairweather, P. G. (1999). Deploying intelligent tutors on the web: an architecture and an example. *International Journal of Artificial Intelligence in Education*, *10*, 183–197.

Borland Corporation (2001). JBuilder6 Enterprise 6.0.438.0 Documentation. Available at: http://www.borland.com, last visited December 26th, 2002

Brusilovsky, P. (1999). Adaptive and intelligent technologies for web-based education. In C. Rollinger, & C. Peylo (Eds.), *Künstliche intelligenz 4* (pp. 19–25). *Special issue on intelligent systems and teleteaching*.

Brusilovsky, P., Schwartz, E., & Weber, G. (1996). ELM-ART: an intelligent tutoring system on the World Wide Web. *Proceedings of the Third International Conference on Intelligent Tutoring Systems, Montreal, Canada*, pp. 261–269.

Clancey, W. (1983). The epistemology of a rule-based expert system: a framework for explanation. *Artificial Intelligence*, *20*(3), 215–251.

CLIPS (2002). CLIPS, a tool for building expert systems. Available at: http://www.ghg.net/clips/CLIPS.HTML last visited December 26th, 2002

Devedžić, V. (2002). Understanding ontological engineering. *Communications of the ACM*, *45*(4), 136–144.

Devedžić, V. (2003). Next-generation web-based education. *International Journal of Continuing Engineering Education and Life-Long Learning*, Special Issue on The Issues of Technological Support for New Educational Perspectives (forthcoming).

Eckel, B (2000). Thinking in Java, 2nd edition, Prentice-Hall. Available at: http://planetpdf.com/, last visited December 26th, 2002

Fowler, M. (1997). *UML distilled*. Reading, MA: Addison-Wesly.

Friedman-Hill, E.J (2002). Jess, the expert system shell for the Java Platform, v. 6.1a4 User's Manual. Available at: http://herzberg.ca.sandia.gov/jess/, last visited December 26th, 2002

Hall, M. (2001a). *Core Servlets and JavaServer pages*. Englewood Cliffs, NJ: Sun Microsystems Press/Prentice Hall.

Hall M (2001b). Tutorial on Core Servlets and JSP. Available at: http://www.coreservlets.com/, last visited December 26th, 2002

Hall, L., & Gordon, A. (1998). Synergy on the net: integrating the web and intelligent learning environments. *Proceedings of The Workshop on Web-Based ITS, electronic edition. San Antonio, TX.*

Johnson, W. L., Shaw, E., & Ganeshan, R. (1998). Pedagogical Agents on the Web. *Proceedings of The Workshop on Web-Based ITS, electronic edition. San Antonio, TX.*

Larman, C. (1999). *Applying UML and patterns*. Englewood Cliffs, NJ: Prentice-Hall.

López, J. M., Millán, E., Pérez-de-la-Cruz, J. L., & Triguero, F. (1998). Design and Implementation of a Web-based Tutoring Tool for Linear Programming Problems. *Proceedings of The Workshop on Web-Based ITS, electronic edition. San Antonio, TX.*

Melis, E., Andrès, E., Büdenbender, J., Frischauf, A., Goguadze, G., Libbrecht, P., Pollet, M., & Ullrich, C. (2001). Activemath: a generic and adaptive web-based learning environment. *International Journal of Artificial Intelligence in Education*, *12*, 385–407.

Mitrović, A., & Hausler, K. (2000). Porting SQL-tutor to the web. *Proceedings of the International Workshop on Adaptive and Intelligent Web-based Educational Systems. Montreal, Canada*, 50–60.

Prentzas, J., Hatzilygeroudis, I., & Garofalakis, J. (2002). A web-based intelligent tutoring system using hybrid rules as its representational basis. *Proceedings of 6th International Conference on Intelligent Tutoring Systems, ITS, Biarritz, France, and San Sebastian, Spain*, pp. 119–128.

Rational Corporation (2001). Rational Rose Enterprise 2000. Available at: http://www.rational.com/, last visited December 26th, 2002

Rebai, I., & de la Passardiere, B. (2002). Dynamic generation of an interface for the capture of educational metadata. *Proceedings of 6th International Conference on Intelligent Tutoring Systems, ITS, Spain*, pp. 249–258.

Retalis, S., & Avgeriou, P. (2002). Modeling web-based instructional systems. *Journal of Information Technology Education*, *1*(1), 25–41.

Ritter, S. (1997). PAT online: a model-tracing tutor on the world-wide web. *Proceedings of the Workshop: Intelligent Educational Systems on the World Wide Web, Kobe, Japan*, Japan pp. 11–17.

Sandia National Laboratories (2002). Jess the Rule Engine for the JavaTM Platform. Available at: http://herzberg.ca.sandia.gov/jess/, last visited December 26th, 2002. Livermore, CA

Sun Microsystems Corporation (2001). Servlet Specification v 2.3. Available at: http://www.sun.com/products/servlet/, last visited December 26th, 2002

The Apache Software Foundation (2001). Apache HTTP Server Version 1.3.20 Documentation. Available at: http://www.apache.org/, last visited December 26th, 2002

The Apache Software Foundation (2001). Apache JServ Version 1.1.2. Available at: http://java.apache.org/dist/, last visited December 26th, 2002

The World-Wide Web Consortium (2001). XML schema, Recommendations for XML Document's Syntax, Structure, semantics and design. Available at: http://www.w3.org/TR/2001/REC-XMLschema-1-20010502/, last visited December 26th, 2002

Weber, G., & Brusilovsky, P. (2001). ELM-ART: an adaptive versatile system for web-based instruction. *International Journal of Artificial Intelligence and Education*, *12*, 351–384.

UNCORRECTED PROOF