StudTest – A Platform Supporting Complex and Interactive Knowledge Assessment

doi:10.3991/ijet.v3i1.722

V. Glavinić, M. Čupić and S. Groš
University of Zagreb, Zagreb, Croatia

***Abstract*—This paper describes the model and prototype implementation of a knowledge assessment framework based on problem management components. In order to support student testing with complex problem types and enable usage of rich graphical user interfaces for solution entry, we have developed an e-examination model in which the core concept is a component that can generate complex questions and evaluate students' solutions with additional explanation generation, which we named *prlet*. The respective system implementation is described, which can operate under heavy loads.**

***Index Terms*—authoring tool, e-learning, knowledge assessment**

I. INTRODUCTION

Computer based systems for supporting and enhancing faculty courses are nowadays increasingly used, some of the more significant examples being [1], [2], [3]. However, since the main goal of those systems is to be comprehensive, in a way that offers capabilities for course organization, course material repositories and student management and knowledge assessment, some of those capabilities are not worked out as they should be, which introduces limiting factors for their usage. One notable representative is the assessment of students' knowledge, which is typically based on quizzes with limited capabilities; in the university setting quizzes are often implemented as multiple choice questions [15], [16], [17]. In this paper we describe an open framework based on portable technologies and designed with extensibility in mind, specialized for assessment of students' knowledge. This framework heavily relays on the concept of *prlets* (pronounced as "pearl-ets"), a concept introduced following the one of servlets, today widely accepted and used as a core Java based technology [4] for Web applications, and standardized by Sun [5]. A similar approach is under development at Ramapo College of New Jersey, targeting the design of more-than-usually-capable problems known as "problettes" [6], which can be used in most Java enabled Web browsers in the form of Java Applets.

This paper is organized as follows. In Section 2 we give an overview of typical quizzing capabilities offered today. Section 3 provides an overview of practical considerations on quizzes and tests in general. In Section 4 we introduce the model of StudTest, the system being built around the *prlet* concept, and briefly describe its prominent components. In Section 5 we present our prototype implementation of StudTest framework and discuss it briefly. Section 6 presents conclusion and future work directions.

II. QUIZZES OVERVIEW

Since the problems of which quizzes are composed are the core of knowledge assessment, in this section we describe typical problem capabilities offered today regarding this issue. We analyze four factors describing each problem: auto evaluation capability, dynamics capability, presentation randomization capability, and multi-technology presentation capability.

Auto evaluation capability means that a problem can automatically evaluate student answers and determine their correctness. Representative of problems lacking this capability is the "essay-like problem", where students write natural language answers. Conversely, in "ABC-type questions", possessing this capability, the correct answer is known in advance by the system. Some problems where students must enter textual answer can even have this capability, although in a limited sense, if the answer is constrained either to only a few predetermined words or even to multiple words if students are required to select the correct one. In the former case, the words can thus be verified by means of regular expressions, albeit this approach has its issues regarding synonyms, typos etc. In the latter one, it should be noted that a student's recall capabilities are however dominantly checked while her knowledge only in a lesser extent.

Dynamics capability discriminates problems that can utilize some form of template for question generation vs. problems having statically preloaded question texts (and depending on problem-type possible answers). For example, typical ABC questions supported in many popular e-learning systems do not possess this capability as question texts and possible answers must be preloaded by a human. On the other hand, they have *presentation randomization* capability, meaning that the same question will (usually) not be presented in the same way to two different students, e.g. offered options will have a randomized order of appearance. The simplest form of a problem having dynamics capability is the one that can be stated in a pseudo-language as "What is the result of addition of {$a} and {$b}", with the correct solution given as "{$a} + {$b}" and constraints like "a, b are integer in [0, 50]". A more advanced dynamics capability is associated with problems capable to dynamically generate multimedia objects and incorporate them as part of questions (e.g. per student images).

Multi-technology presentation capability means that a problem can be presented to a user by a variety of technologies, e.g. through some local windows based application, through a Web browser, through a cell-phone, etc. This factor is also important since it determines how the user can answer a question (by clicking/selecting, by

entering some text/number or perhaps by drawing). It should be noted that this capability is a widely lacking one.

There are also some important factors concerning quizzes themselves as the following list of desiderata suggests. E.g. what problems will the quiz be composed of, how many questions will be asked, will (in)correctness in answering a previous question influence the selection of the next one? All of these issues point to a common factor – adaptability, itself opening the issue of its implementation mechanism. Namely, adaptability can be based on some simple algorithms or on more complex ones relying on methods from the AI field.

Taking into consideration the range of the above issues, we chose not to fix any of them, but to build a framework that should be capable of supporting all of them. Since quizzing represents only one form of testing students' knowledge, in the remainder of the paper we will be using the more general term of "tests".

III. PRACTICAL CONSIDERATIONS

In order to devise an effective and efficient assessment system, besides the problems themselves, there is a number of other considerations to be analyzed resulting from real system usage (of which we have some experience, working with simultaneous groups of over 120 students, and a total course population of over 1100 students).

First of all, *security* is an important issue. This tackles the question of who will be able to access the tests and when. If testing is used in a supervised way for the whole student population enrolled in the course, which is subdivided into smaller groups at a time, what is the most likely situation (e.g. because of the limited number of available computers), care must be taken to disallow access to tests for students not under staff supervision. Presently this is typically accomplished by password protection (password being communicated to students present in the examination room). To disable password leakage to outdoor students (e.g. through cell-phones), IP based control can also be utilized, and passwords can be changed.

Course policy is another issue that should be taken into account. A typical example is a policy stating that "A student cannot get test X, if she has not passed test Y", or "A student must pass test X, where the number of attempts is unlimited." Also, some courses can have the following policy: "A student can solve test X as many times as she wants; we will grade her by her last attempt" (which is commonly used when trying to ensure that the student effectively learned the course material). Another example of course policy is: "Test X can be taken for no longer than 15 minutes."

The last issue to be mentioned is *scalability and heavy load handling*. Here the system must be correctly dimensioned, so that it can handle a large total number of users (e.g. in the order of thousands). However, depending on specific course organizations, situations can arise where many of the students will use the system simultaneously during time-constrained testing scheduled at fixed times, resulting in heavy peak-loads. In such conditions it is critical that the system insures small response times. This can be achieved in two ways: either by building a clustered system with load-balancing support (this being more expensive), or by building a system based on asynchronous operations that can postpone less important operations during heavy load periods (this being more acceptable in financial terms).

In order to offer both solutions, we have defined a framework that can easily be clustered, and that is based on asynchronous operations.

IV. FRAMEWORK MODEL

To facilitate the implementation of a variety of possibilities and capabilities as listed in Section 2, we have defined the concept of *prlet*, and constructed the rest of the framework to be a *prlet container* – a component based environment that executes *prlets* and supports pluggable objects. *Prlets* are intended to represent pluggable components, which have public names and can be globally referenced thus making them easily sharable. They also contain the complete logic needed for problem editing, instantiation, and possibly evaluation. This framework operates with several categories of objects, as follows.

A. Framework Core Elements

The framework core elements provide the basic functionality for test modeling and problem representation. They include the following ones: TestDescriptor, Test, TestInstance, ProblemType, ProblemRenderer, ProblemGenerator, ProblemEditor, ProblemInstantiator and ProblemEvaluator, see Figure 1.

TestDescriptor is the description of a test and enables selection and inclusion of container supported mechanisms, which are dynamically discovered by active plug-in examination. It enables the inclusion and configuration of available security constraints as well as the TestController (determining e.g. whether the test is adaptive and which problems are to be included) and TestGrader (determining the score assignment policy) components.

Test is a wrapper built for each single user. Test can be visualized as a folder containing all user's attempts to solve a specified TestDescriptor.

TestInstance is a concrete test presented to the user. For each user and each TestDescriptor, TestInstances are grouped into collections by means of Test.

ProblemType provides the user the mental model of a problem. Typical problem types include the following ones: single-correct-ABC-question, multiple-correct-ABC-question, input-text/number-question, input-list-of-text/number-question and CustomProblemPanel. The latter problem type is defined in order to support problems to be presented only in a graphical user interface, and must offer rich tools for solution entry e.g. by drawing. Within the StudTest framework we decided to separate this information from the *prlet* itself, in order to detach type presentation from problem logic issues, leaving only the latter as part of a *prlet*. This separation also enables the implementation of multi-platform presentation capability. Namely, when the user client contacts StudTest, as a part of the handshaking process it must send a Technology identifier telling StudTest what technology for test presentation the client supports. Basing on this parameter, StudTest can then select the appropriate ProblemRenderer component for each ProblemType. This component possesses the information on how to present the user the

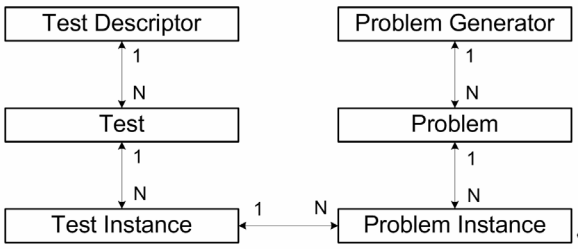

Figure 1. StudTest core objects.

given problem type applying her technology, which is notably most often HTML, where the CustomProblemPanel type can easily be supported by means of Java applets.

Hence, *ProblemRenderer* is a component used for the presentation of a problem of a specified type using the selected technology. Thanks to this separation, in order to add support for new technologies, all that is required is to implement an additional set of renderers, while *prlets* won't be aware of the change.

Within this context the *prlet* is an aggregated component composed of the following components: a ProblemGenerator, one or more ProblemEditors, a ProblemInstantiator and a ProblemEvaluator, see Figure 2.

ProblemGenerator stores the basic information on a *prlet* i.e. its public name, its problem type and whether or not it can automatically evaluate answers.

ProblemEditor is a component allowing customization of a problem template upon which concrete questions (later titled ProblemInstances) are subsequently created. ProblemEditor includes a supported technology identifier. Namely, as the main concern of knowledge assessment systems should be a wide range of supported technologies for problem presentation, editing of problem template parameters can be supported in a smaller range of technologies of which standard HTML should be mandatory. Hence, for each technology a new editor must be written.

ProblemInstantiator is a component in charge of concrete problem generation, based on current parameters of the problem template. Most of the power of the described framework lies exactly here: for problem instantiation we have separate components which can use anything they need (e.g. communicate with other servers on the Internet, use Web services for help, etc.) in order to create new problem instances. Since ProblemInstantiator knows the type of the problem it creates, all necessary data imposed by that type contract will be stored in the ProblemInstances repository.

ProblemEvaluator is a component that evaluates and generates comments on a user solution, determines its correctness using a predefined measure, and generates the correct solution if this is computable/supported and the user provided no correct answer. Due to this responsibility division, evaluators are able not only to implement complex algorithms themselves, but also to use other resources for evaluation purposes, such as contacting other servers, to use clusters prepared for the required calculations, etc.

For each problem type there is a mandatory interface that must be supported by all ProblemType implementations, consisting of methods for obtaining user help, explanations generated during the evaluation process and information on the correct solution. If it is available (i.e. has an evaluator generated one), and unique (i.e. if more correct solutions exist), we define mechanisms for presenting only one, since there exist situations where the number of correct solutions can be infinite.

Many of the above mentioned components use private repositories for information storage. The StudTest model defines Repositories as a name-distinguished collection of Repository objects. Each Repository object contains two separate containers: KeyRepository and AttachmentRepository. KeyRepository is a collection of key-value pairs, where keys are textual objects, while values are arrays of bytes hence enabling storage of any content type. AttachmentRepository is a collection of name-distinguished attachments, each having a name, mime-type and content in form of an associated byte array.

B. Other Framework Elements

Beside the above core elements of the framework directly associated either with tests or with problems, components for enforcing test security constraints and course policy, management of examination process and test instance grading complement the whole picture.

TestStartCheckers are components for enforcing both test security constraints and course policy. These components can at TestDescriptor creation time be associated with it by the person creating the test, and configured accordingly. The most important method these components provide is *isStartAllowed*, which checks whether the user satisfies its constraints and subsequently grants permission for test start. If working with more than one TestStartChecker, all of them must be satisfied for a successful start. Foreseeable applications for checkers are starting: from an allowed IP address range, after a required password is entered, at a specified time frame, during a predefined interval after the supervisor explicitly enabled start, only if other test prerequisites are fulfilled (e.g. passed other tests), etc.

During the examination process, other components titled *TestSupervisors* are used for examination process supervision. They can be associated with TestDescriptor by a person creating the test and configured accordingly. These components have two tasks: to supervise the process of test writing, and to generate status information

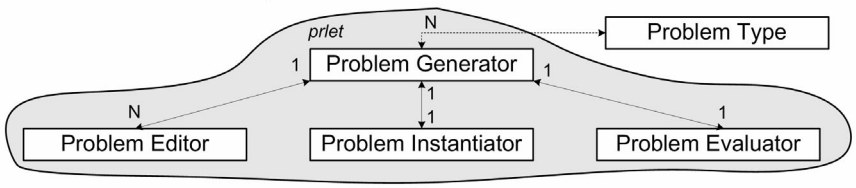

Figure 2. Overview of *prlet* structure and other associated elements.

for the user. Foreseeable applications for the supervisor is the restriction of time within which the test must be solved, informing the user of the time remaining, etc.

The component in charge for management of the examination process, i.e. the examination workflow, is *TestController*. There can be many TestControllers plugged in a system, but during TestDescriptor creation, the person defining it must select one of the available components and configure it accordingly. TestController determines what is to be presented to the user and when. These decisions can be made dynamically. Namely, the first time the user accesses her test, the selected TestController is requested to determine what question(s) are to be presented. When the user solves these questions, TestController is again requested what to do next. Thus, depending on the TestController implementation various scenarios can be generated, ranging from simple quizzes determining all of their questions on the first call and up to adaptive intelligent tests asking only the first batch of questions, analyzing the evaluation results, asking additional questions taking into consideration these results, etc.

When test writing is done, it remains to calculate the total score. Recognizing the fact that this is a very sensitive area, we decided to model this process with an additional component named *Grader*. During the evaluation of user solution correctness, ProblemEvaluator assigns a correctnessMeasure parameter as a value in the interval [0, 1], 0 being totally wrong and 1 being absolutely correct. Also, during the process of test writing, the user can be offered to either enter her confidencyMeasure, a number in the interval [0, 1], or that the default value of 1 is used (the former being preferred). Based both on the parameters correctnessMeasure and confidencyMeasure and the information whether the user has solved the question or left it unsolved, various grading strategies can be adopted e.g. positive/negative score, scoring proportional to correctness and confidence, etc.

C. Framework Helper Elements

By analyzing many university courses and problems suitable for student knowledge assessment, we discovered the following fact: within a given course many questions can be stated having in mind very few course-related concepts whose representation is typically rather complex. To illustrate this, let us imagine a Computer Networks course. A number of questions can be stated beginning with: "A computer network is shown on Figure 1. How to ...". However, generating an image which shows a computer network, and includes symbols for routers, hubs, switches, servers and (regular) computers, determining where and how to place each component is not a trivial task at all. This is the main obstacle in the creation of new problems, especially dynamic ones, where each student can be given her own network. In order to foster the usage of computer generated problems we have provided a component-based facility that can alleviate these problems. The general idea is to introduce a set of components called Helpers, which can produce the required multimedia content (most often images), based on given parameters. In the case of StudTest, these components have direct access to the allowed ProblemInstance repository content and can generate the requested content based on the parameters found in the repository. On the other hand, ProblemInstantiator knows what helper it will use, and from the helper contract where it must leave the necessary data. During the instantiation process, ProblemInstantiator has the facility to include the helper reference, which will than be executed during the problem presentation phase. Within an HTML technology context, this reference is rendered as an IMG tag, which causes the browser to make an additional request to the server, in turn starting the helper that generates the requested content and eventually returns it to the client.

V. FRAMEWORK PROTOTYPE IMPLEMENTATION

We implemented the StudTest framework outlined in Section 4 in the Java programming language. Java technology was chosen for two reasons: first, thus far only Java offers a stable and portable platform for application development. Because of its broad acceptance and existence of Virtual Machine implementations for almost all widely used operating systems, Java is definitely the only optimal choice. The second reason is the fact that we wanted to support graphically rich problems with complex user interfaces and complex solution entry methods. Considering the trends to move assessment systems to the Web and HTML, again the only portable platform is offered by Java Applets [7]. An additional reason is the fact that in order to support dynamic problems, some kind of scripting language is needed. Although the language of choice for such purposes is nowadays JavaScript, it is not a full-fledged OO language as it has never been meant to be such but only a scripting language for client side simple evaluations and event handling. Thus we have based all of our "scripting" needs on the regular and widely accepted OO language with modern language constructs and built-in support for concurrency [8].

Our implementation of the StudTest framework is illustrated in Figure 3. As it can be seen, the implementation is a standalone module relying on a database for data persistence (although it does not have to be a relational database, thanks to the persistence virtualization layer, see Figure 4), and provides its services to clients through developed connectors. Currently only one connector is implemented – the TCP/IP based binary connector with connection pooling. This was done to improve performance. However, other connectors are under development as well, one of which is the Web Services connector to expose StudTest functionality through Web Services over HTTP.

Since our default persistence storage is MySql [9], which is a relational database management system, we needed an appropriate object-relational mapper. Instead of implementing this from scratch, we decided to use Hibernate 2 [10], which is a wide accepted O/R mapper for Java.

To allow an easier system distribution and to enable a

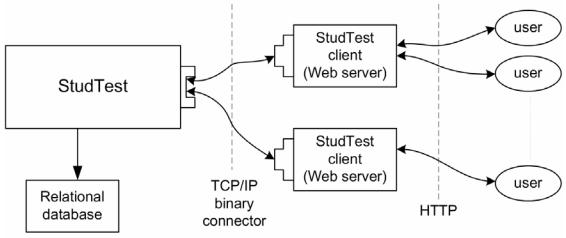

Figure 3. StudTest model implementation.

better peak-load handling, most of StudTest operations are implemented as asynchronous ones, while the communication with the components is hidden behind suitable interfaces, which allows an easy component replacement/reimplementation. Most notably, there are two distinguished queues: the ProblemInstantiation queue and the ProblemEvaluation queue, both being implemented as priority queues. When there is a need to instantiate problems, the process is not started immediately; instead, the instantiation request is added to the ProblemInstantiation queue. Similarly, when there is need to evaluate a problem instance, the request is added to the ProblemEvaluation queue. During system setup, the configured number of InstantiatorWorkers and EvaluatorWorkers is started. These workers continually read requests from appropriate queues and execute them. When a large number of requests is generated, they will be processed gradually, and the system will continue to function normally instead of collapsing.

This design can also support scenarios of primitive and dedicated clustering configurations, where both queues can be exposed over TCP/IP connectors, and in which additional StudTest systems can be setup and dedicated to execution of instantiation and evaluation requests, while the main StudTest server could handle examination workflow operations and the users answer storage. Even more, since communication with Helpers also happens through well-defined interfaces, it is possible to setup additional systems for Helper execution. This is especially important since Helpers can consume large amounts of memory, e.g. during generation of multimedia contents.

A. *Implemented Components*

We have so far implemented a number of components extending system capabilities, as circumstances required to cover the needs of two large courses being taught at the University of Zagreb, Faculty of Electrical Engineering and Computing (viz. Digital Electronics [13] and Digital Logic [14]); these include TestStartCheckers, a TestSupervisor, a TestGrader and a TestController.

The TestStartCheckers we have implemented are the following: *QueueStartChecker* (requires student to register for test and provides simultaneous enabling of registered test instances), *TimeFrameStartChecker* (allows the test to start within the predefined time period), *TimeWindowStartChecker* (works with QueueStartChecker and disables the start of test writing after a predetermined amount of time since granting it), *IPAddressStartChecker* (allows the test to start only from a selected IP address range), *PasswordProtectionStartChecker* (enables start of tests only if students enter the correct password) and *PassedTestPrerequisiteStartChecker* (enables start of tests only if prerequisite tests were solved and passed).

The only instance of TestSupervisors implemented so far is *TestDurationSupervisor*. It is configurable to allow a fixed time for solving, which starts with the student beginning to solve the test. It can be also fixed to a time/date deadline. In both cases it reports the information on time remaining to the end of the test.

The implemented TestController provides the following capabilities:

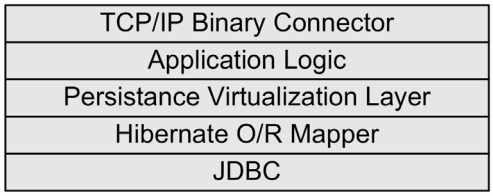

Figure 4. StudTest layered structure.

- static problem selection from a given problem group, which in turn can contain subgroups, or even exclusive subgroups,
- fixed number of presented questions,
- configuration for a number of questions to be displayed at once,
- forward/backward navigation with the possibility to turn off backward navigation,
- on/off switch for direct navigation to each question,
- maximum achievable score for test,
- threshold for test passing, and
- on/off switch for enabling of multiple solution attempts.

The latter is the functionality we had not anticipated to be necessary, but emerged when the Bologna process was implemented at our Faculty, and brought forward to the examination procedures the concept of homework. Homeworks are a specific challenge, since they break apart the conventional examination procedure where a student opens her test, solves and submits it, all under heavy supervision of all available and configured security mechanisms. Namely, here students are able to access their homework (which is, from the system standpoint, only another test), solve some questions, suspend the solving process, resume it the next day, etc., within a predetermined period of time (e.g. during a week). However, due to proper system modeling and design, this functionality was easily added.

We have also completed one implementation of Graders, offering rich configuration capabilities through the use of a simple scripting language, exemplified in Figure 5.

B. *Integration with other systems*

StudTest is a standalone module for user knowledge assessment – and nothing more. StudTest neither defines nor provides any technology or connector through which users could directly work with it, e.g. from a Web browser. Instead, in order to be used StudTest must be included into some other system that provides the user

```
if $isSolved then
  if $isCorrect then
    return 10;
  else
    return -2;
  end if;
else
  return 0;
end if;
```

Figure 5. Simple grader configuration.

interface and communicates with StudTest through its respective connector. A typical system targeted for StudTest inclusion is a Web based e-learning system or a simpler Web based course management system, which provide the suitable user interface. Namely, even when the client accesses StudTest and claims to use HTML technology, the respective TestRenderer (a component that renders a whole test in a given technology) will not create the whole Web page. It will instead deliver through its connector two chunks of HTML pages: one to be included in the HEAD part of the document, and the other to be included in the BODY part, as shown in Figure 6. The StudTest client is than free to add all surrounding data or adjust the look&feel of the generated HTML document before it sends it back to the user.

In order for clients to use and communicate with StudTest, they must utilize the client side of the connector,

```
<HTML>
 <HEAD>
   <!-- head part of document -->
 </HEAD>
 <BODY>
   <!-- body part of document -->
 </BODY>
</HTML>
```

Figure 6. HTML document structure.

which is also written in Java. And to further simplify StudTest utilization, we have prepared for inclusion a simple ready-to-use Servlet.

We have quite extensively tested StudTest along with its connectors within our course management system Nescume [11] supporting two courses enrolling over 2200 students, without any serious objections or complaints.

C. *Support for prlet development*

Since the natural environment for *prlets* is the *prlet* container, which is a large and complex environment, *prlet* development typically poses some challenges. Namely, before deploying a *prlet* into the *prlet* container, it should be completed and tested. To support an easier *prlet* development, we have also prepared a *prlet* development framework helping the process by simulating the *prlet* lifecycle. The *prlet* lifecycle encompasses creation of a new problem template, editing of an existing problem template, problem instantiation, presentation of a problem to the user and problem solving by user (in this case, the developer), and finally problem instance evaluation and generation of comments and correct solution (if available/supported by *prlet*).

D. *Example of developed prlets*

An example of developed *prlets* is shown on Figure 7. A random Boolean function is generated (and displayed) for each student, as is visible in the illustration. The student must design a CMOS circuit which implements the given function. For purposes of circuit design, the problem is imaged using a Java Applet with a simple program for schema drawing. The drawing components (like MOSFETs, inputs, outputs etc.) are accessible from a popup menu. In the evaluation phase, solution correctness is checked using an appropriate CMOS simulator. In this case the problem generating code, which permits the

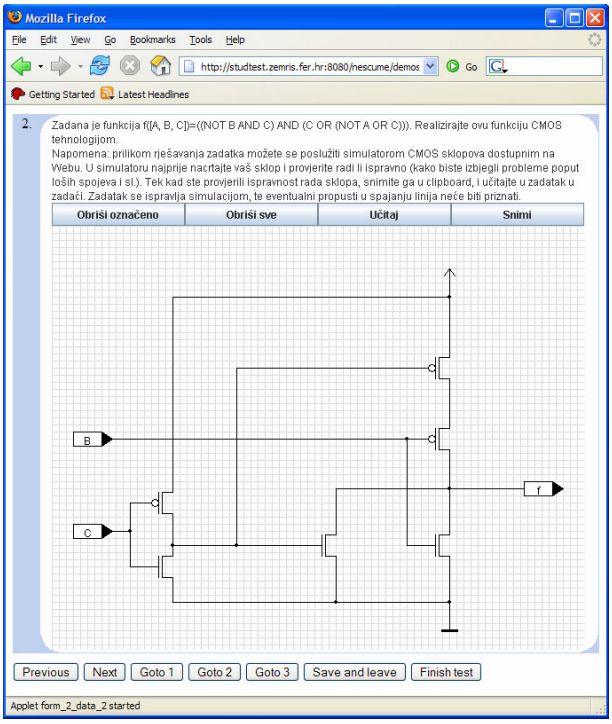

Figure 7: Screenshot of test containing applet based problem addressing CMOS circuit design

student to solve the problem and subsequently evaluates the solution correctness, consists of a single *prlet*.

Figure 8 shows another example illustrating a random PLA based circuit, rendered by an appropriate Helper, where the student is requested to determine the counting sequence for the implemented sequential circuit. During problem definition, *prlet* randomly creates one cycle for each student, programs the PLA based circuit and prepares the data required by the associated helper to draw its schematics. Next, it randomly generates three cycle instances and presents the whole to a student as a Single Choice question itself.

VI. CONCLUSION AND FUTURE WORK

The StudTest framework was developed as an effort to offer rich and complex examination capabilities for generated problems and to form a specialized subsystem for knowledge assessment, which should be independent of user technology. This framework is therefore designed with the following goals in mind: extensibility, good peak-load handling and scalability. More important, it represents a well-defined model, which supports the prototype implementation. The system has been used on several thousands students, within two courses (Digital Electronics and Digital Logic) in eliminatory lab entry tests and final scoring tests, as well as in student homework management. Because of its advanced graphical capabilities, we have been able to implement *prlets* such as the one requiring students to draw a CMOS schema of a circuit implementing a randomly generated Boolean function, and automatically verify the correctness of the respective design. Since students accessed tests through Nescume, which is Web based, these complex problems were presented to students by means of Java Applets. In total, we have developed over 65 *prlets* used

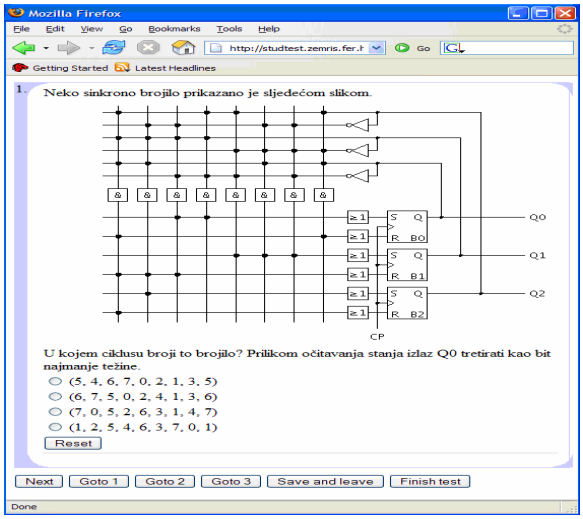

Figure 8: Screenshot of test addressing PLA based circuit design

in homeworks, and about 300 static problems used in laboratory exercises.

As a direction for future work, we plan to work out and implement a better clustering support and implement an adaptive TestController that would use an ontologically described course structure and its relationships to the existing problems (based on RDF and RDFS) for problem selection. Support for reasoning about this knowledge will probably be obtained through the Sesame system [12].

ACKNOWLEDGMENT

This paper describes the results of research being carried out within the project 036-0361994-1995 Universal Middleware Platform for e-Learning Systems, as well as within the program 036-1994 Intelligent Support to Omnipresence of e-Learning Systems, both funded by the Ministry of Science, Education and Sports of the Republic of Croatia.

REFERENCES

[1] WebCT: http://www.webct.com/ (visited on 2007-10-15)
[2] BlackBoard: http://www.blackboard.com/ (visited on 2007-10-15)
[3] Moodle: http://www.moodle.org/ (visited on 2007-10-15)
[4] Java: http://java.sun.com/ (visited on 2007-10-15)
[5] JSR-000053 Java Servlet 2.3 Specification http://www.jcp.org/aboutJava/communityprocess/final/jsr053/ (visited on 2007-10-15)
[6] Problettes – The Home Page http://phobos.ramapo.edu/~amruth/grants/problettes/ (visited on 2007-10-15)
[7] Applets: http://java.sun.com/applets/ (visited on 2007-10-15)
[8] G. Booch, Object-oriented analysis and design with applications, Second edition, Addison Wesley, 1994.
[9] MySql: http://dev.mysql.com/downloads/ (visited on 2007-10-15)
[10] Hibernate: http://www.hibernate.org/ (visited on 2007-10-15)
[11] V. Glavinić, M. Čupić, S. Groš, "Nescume - A System for Managing Student Assignments", Proceedings of the First International Conference on Internet Technologies and Applications (ITA 05), Wrexham, North Wales, UK, 2005. pp. 233-238.
[12] Sesame: http://www.openrdf.org/ (visited on 2007-10-15)
[13] Course Digital electronics, http://www.fer.hr/predmet/digel (visited on 2007-05-03)
[14] Course Digital logic, http://www.fer.hr/predmet/diglog (visited on 2007-05-03)
[15] R. W. Brown, "Multiple-choice versus descriptive examinations". 31st ASEE/IEEE Frontiers in Education. IEEE, 2001.
[16] T. S. Roberts, "The use of multiple choice tests for formative and summative assessment". ACE 2006. Australian Computer Society, 2006.
[17] K. Woodford, P. Bancroft, "Multiple choice questions not considered harmful". ACE 2005. Australian Computer Society, 2005.

AUTHORS

V. Glavinić (e-mail: vlado.glavinic@fer.hr) is with the Department of Electronics, Microelectronics, Computer and Intelligent Systems, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia.

M. Čupić (e-mail: marko.cupic@fer.hr) is with the Department of Electronics, Microelectronics, Computer and Intelligent Systems, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia.

S. Groš (e-mail: stjepan.gros@fer.hr) is with the Department of Electronics, Microelectronics, Computer and Intelligent Systems, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia.

This article was modified from a presentation at the International Conference of Interactive Computer Aided Learning ICL2008, September 24 - 26, 2008 in Villach, Austria. Manuscript received 5 November 2008. Published as submitted by the authors.