The current issue and full text archive of this journal is available at
www.emeraldinsight.com/0264-0473.htm

Received 5 September 2007
Revised 24 October 2007
Accepted 31 October 2007

An architecture for e-learning system with computational intelligence

Marc El Alami, Nicolas Casel and Denis Zampunieris
Cellule d'Ingénierie et de Conseil en e-Learning, Faculty of Sciences, Technology & Communication, University of Luxembourg, Luxembourg, Grand-Duchy of Luxembourg

Abstract

Purpose – The purpose of this paper is to introduce a new kind of learning management system: proactive LMS, designed to improve the users' online (inter)actions by providing programmable, automatic and continuous intelligent analyses of the users' behaviours, augmented with appropriate actions initiated by the LMS itself.

Design/methodology/approach – Proactive systems adhere to two premises: working on behalf of, or pro, the user, and acting on their own initiative, without the user's explicit command. The proactive part of the LMS is implemented as a dynamic rules-based system, and is added next to the initial LMS. They both use the same database as their source of information on the users, their activities, the available resources and the current state of the whole system.

Findings – How the proactive part of the LMS was implemented on the basis of a dynamic expert system is shown. Also how it looks like from a user's point of view is sketched. Finally, examples of intelligent analysis of users' behaviours coded into proactive rules are given.

Research limitations/implications – Future work should include the design and the implementation of sets of rules (packages) dedicated to common users' needs, enabling useful proactivity on the basis of elaborated intelligent analysis.

Originality/value – Current learning management systems (virtual educational and/or training online environments) are fundamentally limited tools. Indeed, they are only reactive software: these tools wait for an instruction and then react to the user's request. Students using these online systems could imagine and hope for more help and assistance tools: LMS should tend to offer some personal, immediate and appropriate support as teachers offer in classrooms. The proactive LMS can, for example, automatically and continuously help and take care of e-learners with respect to previously defined procedures rules, and even flag other users, like e-tutors, if something wrong is detected in their behaviour.

Keywords Computer based learning, Learning, Management techniques

Paper type Research paper

1. Introduction

Learning management systems (LMS) or e-learning platforms are dedicated software tools intended to offer a virtual educational and/or online training environment. Unfortunately, current LMS are fundamentally limited tools. Indeed, they are only reactive software developed like classical, user-action oriented software. These tools wait for an instruction and then react to the user request.

Students using these online systems could imagine and hope for more help and assistance tools, based on an intelligent analysis of their (lack of) actions. LMS should

The Electronic Library
Vol. 26 No. 3, 2008
pp. 318-328

0264-0473
DOI 10.1108/02640470810879473

tend to offer some personal, immediate and appropriate support like teachers do in classrooms.

In this paper, we introduce a new kind of LMS: a proactive e-learning management system, designed to improve the users' online (inter)actions by providing programmable, automatic and continuous intelligent analyses of the users' behaviours, augmented with appropriate actions initiated by the LMS itself.

Our proactive LMS can, for example, automatically and continuously help and take care of e-learners with respect to previously defined procedures rules, and even ?ag other users, like e-tutors, if something wrong is detected in their behaviours; it can also automatically verify that awaited behaviours of e-users have been carried out, and it can react if these actions did not happen.

In the following text, we show how we implemented the proactive part of our LMS on the basis of a dynamic rule-based expert system, and we sketch how it looks like from a user's point of view. Finally, we give some examples of computational intelligence coded into proactive rules.

2. User interface

Recent works (see Brusilovsky (2003) for example) also propose how to improve current web-based educational systems by adding intelligence in these systems, but these intelligent add-ons modules are as static as the initial LMS was. Indeed, they still need a click or an action from the user to activate it. Our goal was to design and develop a LMS that is able to analyze a situation and to act spontaneously with respect to this situation without queries from its environment. In our system, the user can receive information, help or hints sent by the proactive system at any time and with no actions needed from him. These messages should not disturb him in his work, that's why the interface has been thought in such a way that the information will be viewable at any time and in any context in the LMS.

A messages zone has been dedicated in the header (see Figure 1). This alert zone is a Flash application which is able to display server messages in real-time. Messages are following each other vertically and are colored differently according to their importance. By clicking on a message, the user opens the Messages Manager (see Figure 2) and he/she can read more details on her/his alerts and save them.

3. Proactive computing

Proactive systems (see Tennenhouse, 2000; Salovaara and Oulasvirta, 2004) act on their own initiative on behalf of the user and without waiting for user's explicit command. In other words, the user is "above the loop", that means that the system can

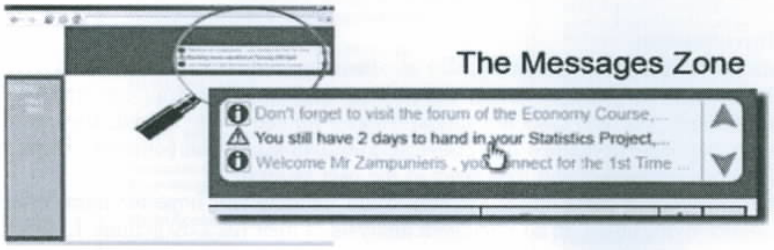

Figure 1.
The message zone

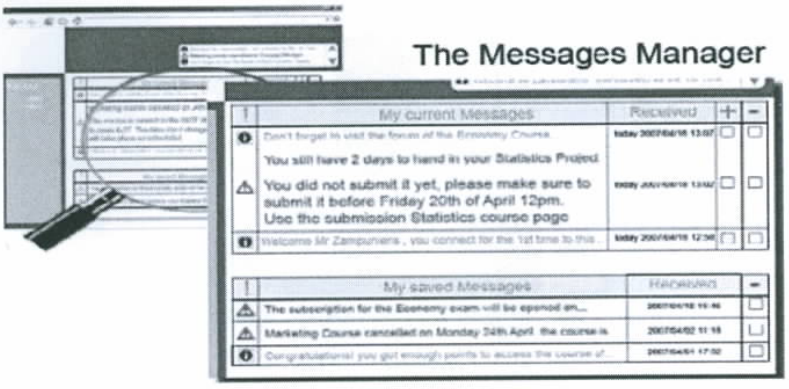

Figure 2.
The message manager

manage himself with no human interactions. The user is here to supervise the system and take last decisions. These systems are more reactive to extern events and can communicate automatically with users using the usual ways of messaging like popup, emails, alerts zone or logs for administrators. The continuous computing of users (inter)actions is possible thanks to a set of programmed behaviours augmented with appropriated actions.

The aim of proactive LMS is to help users to better interact online in a learning environment. Guiding new user through the LMS is a common problem: dynamic help will be more efficient than an elementary static interface. Moreover, pro-activity is useful in case of urgent situation: it enables focusing and dealing immediately with relevant information. For instance, a learner may have difficulties with an online exercise. Our system catches analyses and transmits pertinent data to the tutor who can quickly respond. With standard LMS, the same piece of information is merely stored in a database.

How does it work? The proactive part of the LMS is based on few principles:

- Every proactive behaviours is coded as a Rule in the Rules Running System (RRS).
- Rules are pushed in a FIFO queue.
- The RRS is activated for a time frequency F and performs the N first rules of the FIFO queue.
- Once executed, a rule is deleted from the system.
- To regenerate, a rule should clone itself to be pushed again in the FIFO queue and be reactivated later.

Figure 3 describes the different actors interfering in the system. The RRS is the extra element that wouldn't appear in a traditional LMS architecture. It is independent from the initial LMS server but works in complementarity with it by sharing the same database. The RRS also directly communicates with the Client and the LMS server. This point is explained in section 4: "RRS architecture".

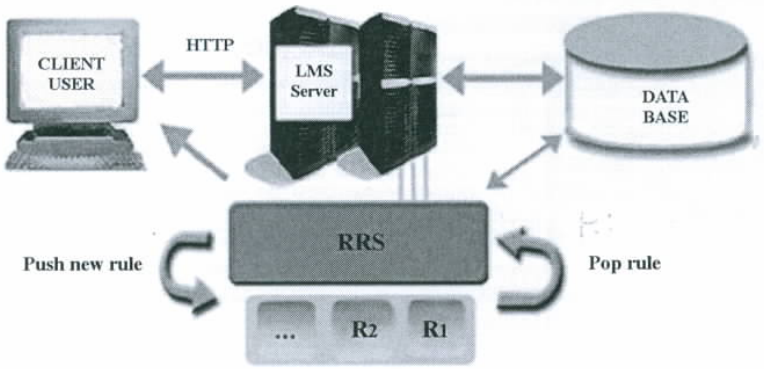

Figure 3.
The rules running system (RRS)

4. Overview of rules contents

The proactive part of our LMS is based on a dynamic rules-based system that is described in the next Section: "The dynamic rules running system". But let us first explain the contents of a rule. Figure 4 represents the algorithm of a rule. As one can see a rule is made of five parts: data acquisition, activation guards, conditions, actions and rules generation. These parts are successively briefly described hereunder. We will not enter into syntactic details. Rules do not have arguments (parameters) but the rule generation procedure is parameterized. That is, when a rule is created, an abstract version of the rule is instantiated into a concrete one with definite values. These choices are made for improving the simplicity and the efficiency of the parsing and running processes: no type checking for parameters, no stack memory management, and so on. Running a large set of rules is then performed quickly. Moreover, as examples given in Section: "Examples of rules declarations and uses" will show, it does not restrict much the expressivity: useful and powerful rules can still be written in this language.

The first part (data acquisition) allows a rule to get information from the LMS in order to use these data in its other parts. This implies that the data acquisition part is the first one to be performed when a rule is run. These data are stored into variables local to the rule; their values can not be modified by the rule – these are read-only variables (but they can be used as references to access and modify values into the LMS database) and are discarded once the rule is run.

Examples of data acquisition are:

- To get from the database the profile of an e-student, based on its identifier.
- To get the connected status of an e-tutor, based on its identifier.
- To get the current date and hour of the LMS.

The second part (activation guards) is performed after the data acquisition part and is made of a set of *and* connected tests on local variables that, once evaluated, determine if the conditions and actions parts will be performed afterwards. Note that the last part of the rule, rules generation, is always performed.

Data acquisition

i. **repeat** for each data acquisition request DA
   a. perform DA
   b. **if** error **then** raise exception on system manager console and go to step vii
      else create new local variable and initialize it with result of DA
ii. create new local Boolean variable "activated" initialized to false

Activation guards

iii. **repeat** for each activation guard test AG
   a. evaluate AG
   b. **if** result == false **then** go to step vi
      else **if** AG == last activation guard test
      then activated = true

Conditions

iv. **repeat** for each conditions test C
   a. evaluate C
   b. **if** result == false **then** go to step vi

Actions

v. **repeat** for each action instruction A
   a. perform A
   b. **if** error **then** raise exception on system manager console and go to step vii

Rule regeneration

vi. **repeat** for each rule generation R
   a. perform R
   b. insert newly generated rule as the last rule of the system
vii. delete all local variables
viii. discard rule from the system

Figure 4.
The algorithm to run a rule

Examples of activation guards tests are:

(1) Is the current date and/or time higher than a given date and/or time?

(2) Is a specific e-tutor (the one identified in the first part, for instance) currently connected to the LMS?

If all the activation guards are evaluated positively, then the conditions and actions parts are performed. On the contrary, these parts are ignored when running the rule. There is a special and automatically defined local Boolean variable called "activated" which value is set accordingly to the result of the guards evaluation.

The third part (conditions) is made of a set of *and* connected tests on local variables that, once evaluated, determine if the actions part will be performed afterwards. Syntax and semantics of conditions tests are equivalent to activation guards tests.

The fourth part (actions) is made of a list of instructions that will be performed in sequence if all the conditions part tests are evaluated positively.

Examples of action instructions are:

- Send a LMS local e-mail to a specific student.
- Show a message box on the screen of a specific currently connected user.

The fifth and last part (rules generation) is performed at the end. It allows the rule to generate other(s) rule(s) that will be performed afterwards (see the following Section: "The dynamic rules running system"). With this mechanism, one can program long-lasting rules that perform actions over a period of time.

5. Examples of rules declarations and uses

Here follow two examples of rules, targeted to different uses and/or users. Of course, these examples do not reflect all the possible uses of the proactive system. The examples are simple and intend to show how the proactive system can automatically take care of e-learners, and even notify an e-tutor if something "wrong" is detected in the e-learner behavior. These two rules can be automatically added to the system when an e-student (id = S) is registered to an e-course (id = C) under the coaching of an e-tutor (id = T).

The first rule gives some welcome and recommendation words to the e-student the first time he/she connects to the e-course.

Data acquisition

```
es = get_user(S)

ec = get_course(C)
```

Activation guards

```
es.isConnectedToCourse(C) == true
```

Conditions

Actions:

```
showMessageBox(es.session, "Welcome to the course" + ec.name)

showMessageBox(es.session, "Do not forget to take a look at the forum"

+ "dedicated to the course" + ec.name + "called" + ec.forum)
```

Rules generation

```
if(activated = false)

then cloneRule(self)
```

The second example is the declaration hereunder of a rule that is intended to check that an e-student (id = S) started to explore/learn the module 3 of an e-course (id = C) at a given date (D) and if not, to notify it to the e-professor of the e-course (id = P) by a message box on its screen when he is connected to the LMS. (Please note that it would be a better design choice to send an email to the e-professor P immediately when the LMS faces that situation than to wait that P is connected to the LMS in order to show him a message box.)

Data acquisition

```
es = get_user(S)

ep = get_user(P)

ec = get_course(C)

module = get_module(ec, 3)

date = get_date()
```

Activation guards

```
date >= D

ep.isConnected() == true
```

Conditions

```
es.numberOfConnections(module) == 0
```

Actions

```
showMessageBox(ep.session, "Warning − student" + es.name + "did not enter"
  + "module" + module.name + "yet")
```

Rules generation

```
if(activated = false)

then cloneRule(self)
```

6. RRS architecture

Before starting to develop the system, some technological choices were made:

- The system should be simple, effective and fast: therefore we choose the C language that has these advantages.

- The LMS is able to send information to the client at anytime without waiting for a client query. Indeed, using sockets was clearly necessary in order to keep a permanent connection where both parts can be servers.
- The client is able to treat and post text in a browser automatically without refreshing.

Flash is the right client application to meet these requirements. Widespread (97 per cent of the browsers) and cross platform, it can be coupled with several servers technologies (Java, JSP, PHP, ASP, etc) and the most important property is that it can handle a socket connection.

Figure 5 shows an overview of the Rules Running System. The RRS is able to pop a rule from the FIFO queue, check the type of this rule and parameters included. The reference to the function (or proactive behaviour) which performs the rule is given to the RRS. Actually, the RRS accesses a library that defines every rule and instructions to execute it. For instance, these actions can be sending a message to the client through the socket connection or by email. In this example, the rule is to send a welcome message to the user no. 46. The message is build in the RRS and put in the shared memory. The server checks the shared memory and post the welcome message to the right online user. The use of the shared memory allows the RRS and the server to work asynchronously. The RRS can continue to handle new rules while the server is in charge of sending the message to the client.

Figure 6 shows the path followed by a message from the state of a rule on server-side to the print on the client screen. First, the RRS pops the rule from the queue, tests the activation guards (ex: test if the user is connected) and executes the rule. The message is built and put in a shared memory where the socket server can easily access and read it. Before reading the shared memory, the server checks if there is a new message to send. Meanwhile, a client is connected and if the message concerns him, the server will use the open socket to address him the data read from the shared memory. Finally the client will send back a positive acknowledgement. If there is no response from the client after a given time, the server creates a log to inform the administrator. The RRS waits for the socket server to read the message in the shared memory before continuing.

With the continuous evaluation of large sets of rules, an efficiency problem might appear with the system saturation. Indeed, as every rule makes an average of two to

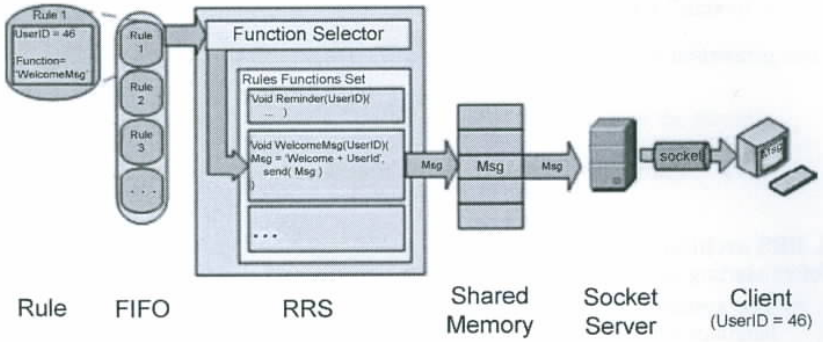

Figure 5.
The RRS architecture

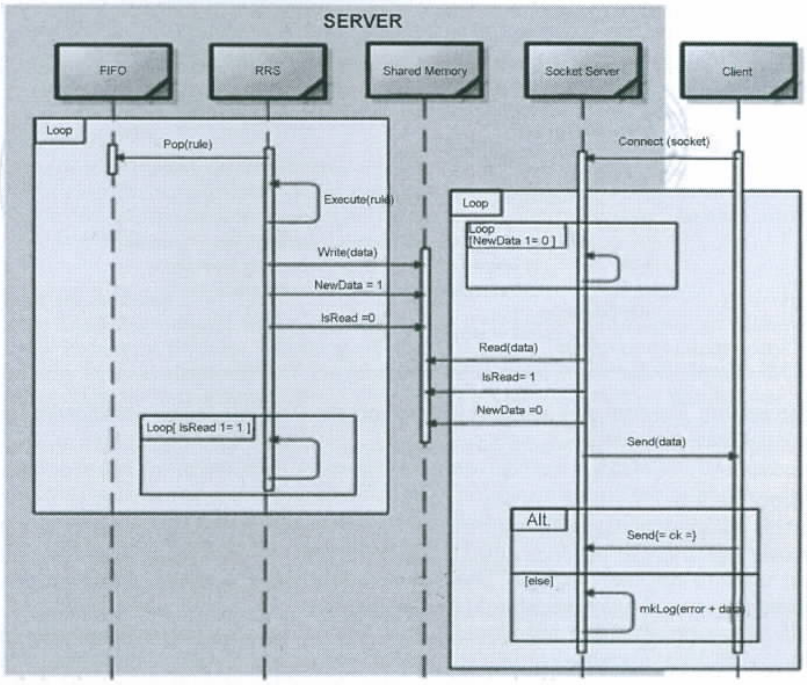

Figure 6.
The message sending sequence diagram

three requests to the LMS database, the mean reaction time of the LMS to a user request is increased, sometimes in a severe way. This problem can be solved with the introduction of "lazy evaluation" in the main algorithm. It is an optimization technique that attempts to delay computation of expressions until the results of the computation are known to be needed. For the interested reader, more information about this solution can be found in Zampunieris (2006a, b).

7. Rules for intelligent analysis

These proactive behaviours coded into rules are the bases for an automatic and intelligent analysis for the current state of the whole system and of its changes implied by user actions and interactions. Figure 7 represents the structure of a specific rule that is intended to check that the e-student S used at least once the LMS forum communication tool dedicated to the e-course C one week after his registration to that e-course and if not, to notify it by an LMS email to the e-tutor T so that he can check with the e-student what is the problem. In the data acquisition, the system gets all the information needed (information about the student, the tutor, the course and the current date), then the activation guards are performed to determine if the rule should be activated or not; if yes, the rule enters the conditions part that determine if the rule should be processed. In that case the rule enter the actions part and accomplishes a list of instructions (in our case, the instruction is to send an email to the tutor with all the

```
data acquisition :
        es = get_user(S)
        et = get_user(T)
        ec = get_course(C)
        date = get_date()
activation guards :
        date > ( ec,startDate + 7 days)
conditions :
        es.numberOfConnections(ec.forum) == ()
actions :
        sendLMSeMail(to = et.name, subject = "Warning", data = "e-student"
                + es name + " did not use the forum " + ec.forum.name
                + " after one week...please check with her/him")
rules generation :
        if (activated == false)
        then cloneRule(self)
```

Figure 7.
Algorithm to run the "forum-warning" rule

information). The fifth and last part (rules generation) allows to regenerate rules that have not been activated.

8. Conclusion

Current learning management systems are fundamentally limited software tools: they are only reactive, user-action oriented software. These tools wait for an instruction and then react to the user request. They do not offer some personal, immediate and appropriate support like teachers do in classrooms.

In this paper we have introduced a new kind of learning management system: proactive LMS. These e-learning platforms with computational intelligence are designed to improve their users' online interactions by providing programmable, automatic and continuous analyses of users actions augmented with appropriate actions initiated by the LMS itself. We have showed how to implement such a proactive LMS on the basis of a dynamic rules-based expert system. We also gave some examples of proactive rules declarations. Finally, we sketched how it looks like from the users' point of view.

Future work includes the design and the implementation of sets of rules (packages) dedicated to common users needs, as well as to elaborated intelligent analysis coded into proactive rules.

These sets of rules will be of two kinds: standard packages that one will be able to use as is, as well as abstract packages (templates) that one will have to tailor to its specific needs by using appropriate tools.

References

Brusilovsky, P. (2003), "A component-based distributed architecture for adaptive web-based education", in Hoppe, U., Vardejo, F. and Kay, J. (Eds), *Artificial Intelligence in Education: Shaping the Future of Learning through Intelligent Technologies, Proceedings of AI-ED 2004, Sydney, Australia, July 20-24, 2004*, OIS Press, Amsterdam, pp. 386-8.

Salovaara, A. and Oulasvirta, A. (2004), "Six modes of proactive resource management: a user-centric typology for proactive behaviors", *Proceedings of the Nordic Conference on Human-Computer Interaction, ACM International Conference Proceedings Series 82 (2004)*, pp. 57-60.

Tennenhouse, D. (2000), "Proactive computing", *Communications of the ACM*, Vol. 43 No. 5, pp. 43-50.

Zampunieris, D. (2006a), "Implementation of a proactive learning management system", in Reeves, Th.C. and Yamashita, Sh.F. (Eds), *Proceedings of E-Learn World Conference on E-Learning in Corporate, Government, Healthcare, & Higher Education, Hawaii, USA, October 2006*.

Zampunieris, D. (2006b), "Implementation of efficient proactive computing using lazy evaluation in a learning management system", paper presented at m-ICTE – International Conference on Multimedia and Information and Communication Technologies in Education, Seville, Spain, 2006.

About the authors

Nicolas Casel is a software developer at the Cellule d'Ingénierie et de Conseil en e-Learning (CICeL), Faculty of Sciences, Technology & Communication, University of Luxembourg. He received a Bachelor degree (IUP IT Engineering HCI) from the Université de Metz in 2003, a Masters degree in the same subject from the University of Limerick, Ireland in 2004 and a second Masters degree in Ergonomics HCI from the Université René Descartes, Paris in 2005. Since February 2006 he has been involved in designing and programming e-learning platforms (Quattropole and Spotlight) at the University of Luxembourg. Prior to that he was an ergonomics consultant (covering usability and accessibility audit, benchmarking, contextual inquiries, tasks analysing, user testing, storyboarding, and internal studies on project management) at SQLI in Paris. He has also worked on project management and web solutions programming at Mikado Online in Luxembourg and analysing and developing data processing tools at Fortis Luxembourg and Citibank Luxembourg.

Denis Zampunieris holds a PhD in computer science from the University of Namur, Belgium and is professor at the University of Luxembourg in the faculty of Sciences, Technology and Communication, where he is the Director of Studies of the Bachelor of Engineering in Computer Science. His current main research interests are the design of new software technologies and tools for e-learning, with a focus on the integration of proactive behaviors. He is the founder and the academic head of the R&D team "CICeL – Cellule d'Ingénierie et de Conseil en e-Learning" (http://cicel.uni.lu). He is the corresponding author and can be contacted at: denis.zampunieris@uni.lu