Implementation of Efficient Proactive Computing Using Lazy Evaluation in a Learning Management System

Denis Zampunieris, University of Luxembourg, Luxembourg

ABSTRACT

In Zampunieris (2006) we proposed a new kind of learning management system, proactive LMS, designed to help users to better interact online by providing programmable, automatic, and continuous analyses of the users' actions, augmented with appropriate actions initiated by the LMS itself. The proactive part of our LMS is based on a dynamic rules-based system. However, the main algorithm we proposed in order to implement the rules-running system suffers some efficiency problems. In this article, we propose a new version of the main rules-running algorithm that is based on lazy evaluation in order to avoid unnecessary and time-costly requests to the LMS database when a rule is not activated, that is, when its actions part will not be performed because preliminary check(s) failed.

Keywords: *electronic learning (e-learning); proactive computing; technological innovations; Web-based learning systems*

INTRODUCTION

Learning management systems (LMSs), or e-learning platforms, are dedicated software tools intended to offer a virtual educational and/or training environment online. Despite a large number of functions covering a large number of user needs for a variety of different users acting in specific roles in these environments, current LMSs are fundamentally limited tools. Indeed, they are only reactive software, developed like classical, user-action-oriented software. These tools wait for an instruction, most likely given through a graphical user interface, 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. LMSs should tend to offer some personal, immediate, and appropriate support like teachers do in classrooms.

Moreover, some particular users like e-tutors have to peruse lots of data in order to try to efficiently manage specific users' needs and would expect some highlighting (where to search and what to look for) from the system instead of a static database.

In Zampunieris (2006) we proposed a new kind of learning management system, proactive LMS, designed to help users better interact online by providing programmable, automatic, and continuous analyses of the users' (inter)actions, augmented with appropriate actions initiated by the LMS itself.

Proactive systems (see, e.g., Tennenhouse, 2000; Salovaara & Oulasvirta, 2004) adhere to two premises: working on behalf of, or for, the user, and acting on their own initiative without a user's explicit command. Proactive behaviours are intended to cause changes rather than just react to changes. This is a major change from interactive computing, in which we lock a system into operating at exactly the same frequency as we do.

Our proactive LMS can automatically and continuously take care of e-students with respect to previously defined procedure rules, and even notify an e-tutor if something wrong is detected in some e-learner's behaviour; it can also automatically check some awaited behaviours of e-students and react if these actions did not happen. Automatic and user-specific checks of generic access conditions (prerequisites) to e-learning modules can be implemented using dynamic rules in the proactive system. Finally, some automatic management processing of the LMS can also be performed by using the proactive part of the system.

The proactive part of our LMS is based on a dynamic rules-based system. However, the algorithm we proposed in order to implement the rules-running system suffers some efficiency problems mainly due to lots of database requests when running the rules, some of them being superfluous.

In this article, we propose a new version of the main rules-running algorithm that is based on lazy evaluation in order to avoid unnecessary and time-costly requests to the LMS database when a rule is not activated, that is, when its actions part will not be performed because preliminary checks failed. In computer programming, lazy evaluation is a technique that attempts to delay the computation of expressions until the results of the computation are known to be needed.

USER INTERFACE

Several recent works also propose to improve current Web-based educational systems by adding intelligence in these systems, but these add-on 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 an LMS that is able to analyze a situation and to act spontaneously with respect to the situation, without queries from its environment.

In our system, the user receives information, help, or hints sent by the proactive system at any time and with no actions needed from him or her. As these messages should not disturb his or her current work (like pop-up windows do, for example), the user interface has been thought of in such a way that the information is viewable at any time and in any context in the LMS in a small screen area.

A message zone has been dedicated in the header (see Figure 1). This alert zone is a Flash application that is able to display the server messages in real time. Messages follow each other vertically and may have different colours according to their importance.

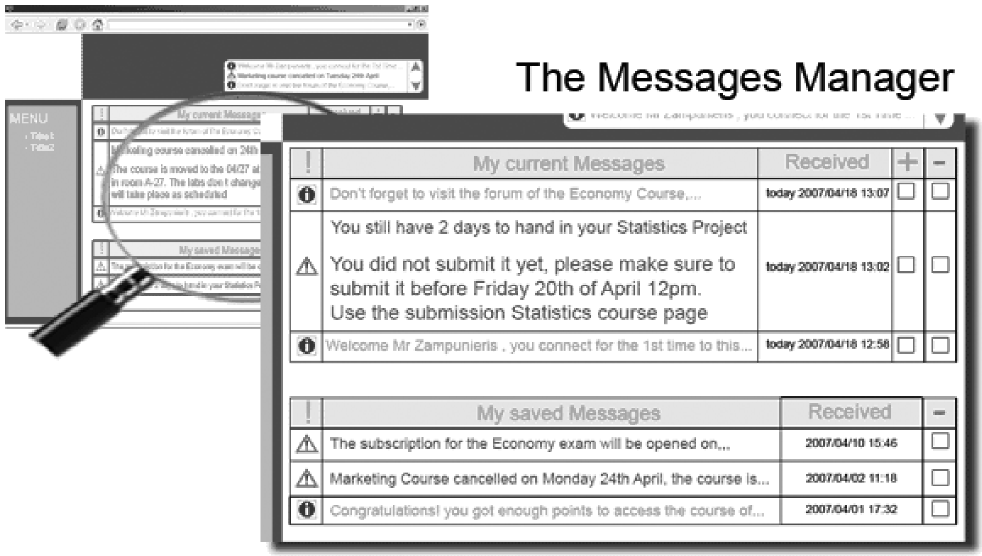
By clicking on a message, the user opens the message manager (see Figure 2) and then can read more details on alerts and can decide to save them or not for later sessions.

DYNAMIC RULES-RUNNING SYSTEM

The set of rules is stored by the proactive system into a FIFO list: The oldest generated rule is at the beginning of the list and will be run first. Two parameters influence the behaviour of the rules-running system: *F* is the time frequency of its activation periods, and *N* is the (maximum) number of rules it runs during an activation period. These parameters are set by the system manager and can be changed at run time. The LMS activates (starts) the rules-running system

Figure 1. The message zone

Figure 2. The message manager



with respect to the parameter *F*. If the rules-running system is already activated, it continues its current activation.

Once activated, the rules-running system executes the *N* first rules of the FIFO list (if available), one at a time with respect to their ranks, using the algorithm shown at the end of this section.

Once run, a rule is discarded from the system. If one wants the rule (or more precisely, the proactive behaviour this rule implements) to stay active in the system for a longer time, the rule has to clone itself in order to be included in the next activation of the rules-running system.

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.

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 cannot 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.

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 condition and action parts will be performed afterward. Note that the last part of the rule, rules generation, is always performed. 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* whose value is set according to the result of the guard 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 afterward. The 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 condition-part tests are evaluated positively.

The fifth and last part (rules generation) is performed at the end. It allows the rule to generate other rules that will be performed afterward. With this mechanism, one can program long-lasting rules that perform actions over a period of time.

The main algorithm to run a rule is as follows. This algorithm is written in pseudo-code and without low-level details for clarity purposes:

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

EXAMPLES OF RULES CONTENTS AND USES

Here follow the declarations of two rules that 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), even if these two rules will be activated only weeks later. This first example is intended 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 behaviour.

The first rule is intended to give some welcome and recommendation words to the e-student the first time she or he 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 rule is intended to check that the same e-student S used at least one time the LMS forum communication tool dedicated to the e-course C 1 week after the start of that e-course and if not, to notify it by an LMS e-mail the e-tutor T so that she or he can check with the student to see what the problem is.

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) == 0
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)

The second example gives a flavour of the automatic management of the LMS by using the proactive system. The rule hereunder automatically collects the number of users connected to the LMS every 5 minutes and stores it in a dedicated table in the LMS database for later statistics purposes.

Data Acquisition:
sys = get_system()
time = get_time()
nb_users = sys.getNumberOfConnectedUsers()
Activation Guards:
time >= T
% T is a parameter of the rule
Conditions:
Actions:
sys.dbStore(table = "statistics", values = time ++ nb_users)
Rules Generation:
if (activated == false)
then cloneRule(self)
else cloneModifiedRule(T / T + 5 min) *% T is replaced by T + 5 min. in the new rule*

Other examples of rules can be found in Zampunieris (2006).

LAZY EVALUATION OF RULES

In the last example, for the sake of efficiency, the database request for the current number of connected users should be performed in the actions part, that is, only if the rule is activated. Indeed, if the rule is not activated (when the activation guard is evaluated as false) there is no need to know these data. Hence, the database request, which is time and resource costly, is not useful and should be avoided.

This efficiency problem might appear as minor in this example but becomes major with the continuous evaluation of large sets of rules. Indeed, as every rule makes an average of two to three requests to the LMS database, and as several desired proactive behaviours combined with a large number of targeted e-students and e-tutors result in a very large number of rules to continuously evaluate, the consequence is an even larger number of successive database requests in order to run the whole set of rules. During the processing of these database accesses, 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. In computer programming, lazy evaluation is a technique that attempts to delay the computation of expressions until the results of the computation are known to be needed.

In our new LMS, the lazy evaluation of rules in the proactive system is used in order to avoid unnecessary and time-costly requests to the LMS database when a rule is not activated, that is, when its actions part will not be performed because preliminary checks failed.

Technically speaking, a variable is no more composed of a pair <*name*, *value*> but of a triple <*name*, *definition*, *value**> where *definition* is the expression that has been or will be evaluated in order to give a value to the variable, and *value** is either a real value or the special value *to_be_computed*.

When a rule is run, in its data acquisition part, local variables to the rule are created but are not given values; instead a triple <*name*, *definition*, *value**> is generated for each variable, with its *definition* component equal to the expression to be computed and its *value** component equal to the special value *to_be_computed*.

When running the other parts of the rule, if the value of a variable is requested, then either its *value** component is equal to the special value *to_be_computed* or it is a different one. In the first case, the expression attached to this variable, stored in its *definition* component, is computed and the result of this evaluation is stored into its *value** component. This data is then usable as the value of the variable.

In the second case, it means that the expression attached to this variable in the data acquisition part of the rule has already been computed previously to give a value to the variable, and therefore this value can be directly used.

Back to the last example in the previous section (production of statistics), the sentence << nb_users = sys.getNumberOfConnectedUsers() >> in the data acquisition part will not result in a database request because the value of the local variable *nb_users* is not computed at that time. This database request will only be performed later when the value of the variable is requested, that is, when the sentence << sys.dbStore(table = "statistics," values = time ++ nb_users) >> in the actions part of the rule is run.

CONCLUSION

Current LMSs are fundamentally limited software tools: They are only reactive, user-action-oriented software. These tools wait for an instruction, most likely given through a graphical user interface, and then react to the user request.

In Zampunieris (2006), we proposed a new kind of learning management system, proactive LMS, designed to help users better interact online by providing programmable, automatic, and continuous analyses of user (inter)actions augmented with appropriate actions initiated by the LMS itself. The proactive LMS can automatically and continuously take care of e-students with respect to previously defined procedure rules, and even notify an e-tutor if something wrong is detected in some e-learner's behaviour; it can also automatically check some awaited behaviours of e-students and react if these actions did not happen.

The proactive part of our LMS is based on a dynamic rules-based system. However, the main algorithm we proposed in order to implement the rules-running system suffered some efficiency problems resulting in an increase of the mean reaction time of the LMS to a user request, sometimes in a severe way.

In this article, we proposed a new version of the main rules-running algorithm that is based on lazy evaluation in order to avoid unnecessary and time-costly requests to the LMS database when a rule is not activated, that is, when its actions part will not be performed because preliminary checks failed. When using such a delayed evaluation mechanism, an expression is not evaluated as soon as it gets bound to a variable, but when the evaluator is forced to produce the expression's value.

Future work includes the design and implementation of sets of rules (packages) dedicated to common user needs that one will be able to use as is, as well as abstract packages (templates) that one will have to tailor to specific user needs by using appropriate tools.

REFERENCES

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

Tennenhouse, D. (2000). Proactive computing. *Communications of the ACM, 43*(5), 43-50.

Zampunieris, D. (2006). Implementation of a proactive learning management system. In T. Reeves & S. Yamashita (Eds.), *Proceedings of e-learn: World Conference on E-Learning in Corporate, Government, Healthcare, & Higher Education 2006* (pp. 3145-3151). Chesapeake, VA: AACE.

Denis Zampunieris holds a PhD in computer science from the University of Namur (B) and is professor of at the University of Luxembourg (L) in the faculty of Sciences, Technology and Communication (http://www.uni.lu), 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 'Ingénierie et de Conseil en e-Learning".