Implementation of a Proactive Learning Management System

Denis Zampunieris
CICeL – Faculty of Sciences, Technology and Communication
University of Luxembourg
Grand-Duché de Luxembourg
denis.zampunieris@uni.lu

Abstract: This paper proposes a new kind of Learning Management Systems: proactive LMS, designed to help their users to better interact online in an educational or training environment, by providing programmable, automatic and continuous analyses of users (inter)actions augmented with appropriate actions initiated by the LMS itself. We explain how to implement such a proactive LMS on the basis of a dynamic rules-based software system. We also give examples of rules *e.g.* to show how the proactive system can automatically take care of users and check some awaited users' behaviors and react if these actions do not happen.

Introduction

Learning Management Systems (LMS), or e-learning platforms, are dedicated software tools intended to give a virtual educational and/or training environment online. Despite their lots of functionalities covering a large number of users needs for a variety of different users acting specific roles in these environments, current LMS are fundamentally limited tools: they are only reactive software.

Indeed, even the more widely used (free or commercial) LMS like ATutor ("*An Open Source Web-based Learning Content Management System*", see http://www.atutor.ca/), Moodle ("*A Free, Open Source Course Management System for Online Learning*", see http://moodle.org/) or WebCT ("*A leading provider of e-learning systems for educational institutions*", see http://www.webct.com/) were 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. One could imagine and hope for more helping and assisting tools, especially because:

1) users could be inexperienced online software users who would expect some guidelines (what to do and how to do it) from the system instead of a static user interface,
2) some particular users like e-tutors have to peruse lots of data in order to try to efficiently manage specific other users' needs and would expect some highlighting (where to search and what to look for) from the system instead of a static database.

Moreover, interactions between the LMS actors, which are at the heart of every learning process, take place mainly by electronic means. If one of these users encounters some difficulties to use online communication tools, how can (s)he ask for some help? how can someone try to help this user online?

In classrooms, teachers can see or feel or check that something "wrong" is happening with a student, and immediately try to help her/him. In face to face situations, special behaviors like the repetitive absence of a student or her/his lack of workload management, are "naturally" notified to teachers who can react on time. LMS should also allow doing or supporting that, efficiently and online.

In this paper, we propose a new kind of LMS: proactive LMS, designed to help their users to better interact online in an educational or training environment, by providing programmable, automatic and continuous analyses of users (inter)actions augmented with appropriate actions initiated by the LMS itself. We show how to implement such a proactive LMS on the basis of a dynamic rules-based software system. We also give examples of rules *e.g.* to show how the proactive system can automatically take care of users and check some awaited users' behaviors and react if these actions do not happen.

Proactive Software Systems

Following (Tennenhouse 2000), interactive computing deliberately places human beings “in the loop” (human-centered computing), while shrinking time constants and sheer numbers demand research into proactive modes of operations in which humans are *above* the loop. This is a major change from interactive computing, in which we lock a system into operating at exactly the same frequency as we do. This is exactly what we propose with our new LMS: the (proactive part of the) software system makes the repetitive and continuous analysis of users’ demands and actions, and notifies the concerned human user(s) only if needed with respect to a given set of rules.

As described in (Salovaara & Oulasvirta 2004), proactive systems adhere to two premises: working on behalf of, or *pro*, the user, and acting on their own initiative, without user’s explicit command. Proactive behaviors are intended to cause changes, rather than just to react to changes. Following the user-centric topology they propose, a proactive system can act with respect to six modes: preparing resources, optimizing resources, advising on the use of resources, manipulating resources, inhibiting resources and finalizing resources. As our LMS system is able to act along every of these six modes, as shown in the Section: “Examples of Rules Declarations and Uses”, it could be ranked as a *full proactive* software system.

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.

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. Examples of activation guards tests are:

- is the current date and/or time higher than a given date and/or time
- 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 email 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.

```
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
```

Figure 1: The algorithm to run a rule.

The Dynamic Rules Running System

The set of rules is stored by the LMS 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 behavior of the rules running system:

- F = the time frequency of its activation periods
- N = the (maximum) number of rules it runs in an activation period.

These parameters are set by the LMS system manager and can be changed at run time.

The LMS proactive system activates (starts) the rules running system 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 LIFO list (if available), one at a time with respect to their ranks, using the algorithm showed in (Fig. 1). This algorithm is written in pseudo-code and without low-level details for clarity purposes.

Once run, a rule is discarded from the system. If one wants the rule to stay present 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.

Examples of Rules Declarations and Uses

Here follow four 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.

Example 1 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 behavior. Example 2 shows how the proactive system can automatically check some awaited behaviors of e-students, and react if these actions did not happen. Example 3 gives a flavor of automatic management of the LMS by using the proactive system.

Example 4 demonstrates how access conditions (prerequisites) to e-learning modules (imposed by authors and/or administrative staff) can be implemented using dynamic rules.

Here follow, as the first example, 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).

The first rule is intended to give some welcome and recommendation words to the e-student the first time 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 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.

```
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)
```

A 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)
```

The third example shows how the rules system can also be used for system management of the LMS. For instance, the two rules declarations hereunder allow the system manager to order a backup of the LMS data at date D, starting around hour H provided no user is connected to the system. During the backup procedure, the LMS is locked: it does not accept anymore an user connection.

The first rule is added to the system by the system manager. It is intended to start the backup procedure, if it is possible, or otherwise to send by email a notification to the system manager.

```
data acquisition :
        sys = get_system()
        date = get_date()
        hour = get_hour()
activation guards :
        date == D
        hour >= H
conditions :
        sys.numberOfConnectedUsers() == 0
actions :
        sys.lock()
        sys.startBackup()
rules generation :
        if (activated == false)
        then cloneRule(self)
        else  if (sys.isLocked() == false)
            then sendLMSeMail(to = sys.manager, subject = “Warning”,
              data = “failed to start backup because of connected users”)
            else addRule(R)
```

The second rule (id = R) would be added to the system by the first rule when its actions part is completed, that is: the backup procedure is started. It is intended to unlock the LMS after the backup procedure.

```
data acquisition :
        sys = get_system()
activation guards :
        date == D
        hour >= H + 1 hour
conditions :
        sys.isLocked() == true
actions :
        sys.unlock()
rules generation :
        if (activated == false)
        then cloneRule(self)
```

Finally, here follows the fourth example showing how prerequisites to e-learning modules can be implemented using dynamic rules of the proactive system. For instance, the following rule declaration is intended to check if an e-student (id = S) registered to an e-course (id = C) is authorized to enter the module 4 of this e-course. The conditions imposed to the student are i) to have passed (note higher than 10 out of 20) the final exam of the module 3 and ii) to have completed the satisfaction form for this e-course. It is supposed here that when the e-student will want to enter the fourth module of the e-course and will not be allowed to do so by the LMS system, she/he will contact her/his e-tutor. Another solution would be to modify the rule to automatically send an email to the e-student and/or the e-tutor when the rule detects this problem.

This rule can already be added automatically to the system immediately when the e-student is registered to the e-course, on the basis of prerequisites defined for this course, even if the activation of the rule will only occur weeks later.

```
data acquisition :
        es = get_user(S)
        ec = get_course(C)
        module3 = get_module(ec, 3)
        module4 = get_module(ec, 4)
activation guards :
        es.examTaken(module3) == true
conditions :
        es.finalNoteForModule(module3) > 10
        module3.isSatisfFormCompleted(es) == true
actions :
        module4.grantAccessTo(es)
rules generation :
        if (activated == false)
        then cloneRule(self)
```

Conclusions and Future Work

Despite their lots of functionalities covering a large number of users needs for a variety of different users acting specific roles in these environments, current LMS are fundamentally limited 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 this paper, we proposed a new kind of LMS: proactive LMS, designed to help their users to better interact online in an educational or training environment, by providing programmable, automatic and continuous analyses of users (inter)actions augmented with appropriate actions initiated by the LMS itself. We also showed how to implement such a proactive LMS on the basis of a dynamic rules-based software system.

We gave examples of rules, targeted to different uses and/or users: *e.g.*, to show how the proactive system can automatically take care of e-students, and even notify an e-tutor if something “wrong” is detected in an e-learner behavior; to show how the proactive system can automatically check some awaited behaviors of e-students, and react if these actions did not happen; to show how automatic management of the LMS can be performed by using the proactive system; to show how access conditions (prerequisites) to e-learning modules can be implemented using dynamic rules of the proactive system.

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

References

Salovaara, A., & Oulasvirta, A. (2004). Six modes of proactive resource management: A user-centric typology for proactive behaviors. *Nordic conference on human-computer interaction, 2004,* ACM international conference proceedings series, 82, 57-60.

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