Including Malicious Agents into a Collaborative Learning Environment*

Natalia López, Manuel Núñez, Ismael Rodríguez, and Fernando Rubio

Dept. Sistemas Informáticos y Programación. Facultad de Informática.
Universidad Complutense de Madrid, E-28040 Madrid. Spain.
{natalia,mn,ir,fernando}@sip.ucm.es

Abstract. In this paper we introduce a collaborative environment for the development of medium-size programming projects. Our system provides the usual facilities for communication among members of the group as well as a friendly programming environment for the functional programming language Haskell. A relevant feature of our learning environment is that some of the *students* may be, in fact, *virtual* students. It is worth to point out that these agents will not always behave as *helpers*. On the contrary, it can happen that they produce, on purpose, wrong programs. By doing so, we pretend that students get the abilities to detect mistakes not only in their own code, but also in the code generated by other team-mates.

Keywords. Distributed learning environments, cooperative systems.

1 Introduction

The rapid development of network-based technologies has allowed distance partners to work collaboratively regardless of their location and time availability. In particular, there has been a great proliferation of systems providing collaborative learning environments (see e.g. [4,5,3,7]). We think that students should get used to work as part of a team already in their early stages of learning. In fact, team work encourages students to get some skills that are not covered by classroom lessons. In particular, we advocate that the last requirement of an introductory programming course should consist in the development of a medium-size programming project, where students are grouped into teams. Even though a teacher will evaluate the final result of the project, it is rather difficult to control all the activities of the group. That is why collaborative environments monitoring, among other tasks, the behavior of individual members of the group, may provide the appropriate feedback to the teacher.

In this paper we present a collaborative learning environment that can be used to develop such a medium-size project. We have chosen the programming language Haskell [12]. The reason for this is the same as the one for developing

* Research supported in part by the CICYT projects TIC2000-0701-C02-01 and TIC2000-0738, and the Spanish-British Acción Integrada HB 1999-0102.

WHAT [8].[1] In contrast with other programming languages proposed to teach to first-year students, there are not friendly programming environments for Haskell. Therefore, in order to fairly compare the difficulties the students have to deal with their first programming language, a better interface is needed. Even though Haskell could be a good choice, the current environments are certainly not. However, the architecture of our system can be easily adapted to work under any other programming language.

Next, we briefly sketch the main capabilities of our system.

- It provides an environment to develop collaborative projects. In particular, it allows a *hierarchical* structure of the project. A group of students is partitioned into several teams. Each team has a distinguished member: The *team-leader*. Access privileges to different parts of the project will be given depending on the role of the corresponding student (the team where she[2] is located, considering if she is a team-leader, etc).
- It provides students with a friendly interface to write programs in Haskell. Let us remark that usual Haskell environments are quite verbose and far from friendly.
- It allows students to communicate among them. As pointed out in [10], these communications are specially relevant for the final development of the project, as they allow to monitor[3] the individual behavior of students inside a team. So, it will be possible to determine students attributes as activity/passivity, leadership capabilities, collaborative skills, etc.
- It allows students to test programs written by other team-mates. Students are not only supposed to write their assigned parts. On the contrary, they are responsible of the overall development of the project. So, they must control the programs produced by other team/group mates.

In addition to the previous characteristics, our system will incorporate intelligent agents[4] as part of the project. More precisely, we may include *virtual* agents as group members. They will have the same duties as a usual student. Thus, they are not supposed to (and they will not) be the *perfect student* who is solving all the problems appearing in the development of the project. Actually, they will also create (again, by simulating the behavior of usual students) problems: They may write wrong programs, they may be lazy, they may disobey its team-leader commands, etc. In conclusion, our agents are not really *malicious*. We use this term in order to clarify that their behavior do not correspond with the usual *angelic* conception of intelligent agents.

[1] WHAT (Web-based Haskell Adaptive Tutor) is a tool designed to help first year students having their first contact with the programming language Haskell.

[2] From now on, we assume that the gender of students and teachers is always female.

[3] In order to be able to monitor all the interactions, *physical* meetings among members of the project are restricted (as much as possible). In particular, *real* identities of project members are hidden.

[4] In terms of [14], our agents present as information attitude knowledge (versus belief) while as pro-attitudes we may remark commitment and choice (versus intention and obligation).

The main reason for including this kind of agents in our system is that we pretend to strengthen students collaborative skills. By adding a *distracting* colleague in the group, we pretend that students *learn by being disturbed* [2,1]. So, students will get used to pay attention not only to their assigned tasks but also to the ones developed by other team members. In this line, it is our intention that agents do not reveal their *true* identities. If students find out that some of their team-mates are (malicious) agents, they may be tempted to discard all the work performed by these *peers*. In order to *cover* agents we have the following *advantages*:

- Students are not identified by their real names.
- The number of students taking *Introduction to Programming* is around three hundred, distributed into six groups. These groups have different time-tables and they are located in different class-rooms.
- Students do not expect that some team-members are in fact artificial agents.

The rest of the paper is structured as follows. In Section 2 we briefly comment on the main capabilities of WHAT. Even though our collaborative learning environment is completely independent from WHAT, the teacher will use its *classes mechanism* to classify students in order to form groups/teams. In Section 3 we present our system. First, we describe the early stages of the project, where the teacher plays a fundamental role. Next, we show the system from the student point of view, that is, the possibilities that students have while interacting with our environment. Finally, we describe the behavior of our agents in the system. In Section 4 we present our conclusions and some lines for future work.

2 A Brief Introduction to WHAT

In this section we briefly review the main features of WHAT: A Web-based Haskell Adaptive Tutor [8]. The main duty of WHAT consists in providing exercises to students according to their current command on the language. In order to do that, the system does not only take into account the information gathered about the current student, but also all the information about previous *similar* students. Students are similar when they belong to the same classes. WHAT manages both *static* and *dynamic* classes. The former consider attributes as whether the student is taking the course for the first time, whether she already knows an imperative language, etc. Dynamic classes are used to handle information that is not known at the beginning of the course and that may vary along the academic year. For example, dynamic classes consider whether a student learns fast or whether she has good memory. In contrast with static classes, as dynamic attributes change (on the basis of interactions with the system) students can be automatically relocated into different dynamic classes.

WHAT handles three main categories of exercises: Typing functions, Evaluating expressions, and Programming assignments. For each problem, students are allowed to ask for hints. The proposed problems are chosen according to the student current knowledge of the language. At each moment, students are free

to choose the category of exercises they want to practice. However, the system automatically points to the topic that better fits the current learning necessities of the student. In order to do that, WHAT provides a personalized following, by creating individualized profiles, that allows to offer better assistance to the student.

WHAT incorporates a complete help system for Haskell. Any concept covered during the academic year can be consulted by means of a friendly navigation interface.

Finally, our tutor provides the teacher with information about the skills the students are getting, how they are obtaining these skills, their typical mistakes, and their main difficulties. Thus, the teacher can improve her lessons in forthcoming years, explaining with more detail the parts of the course that seem to be harder.

3 The Collaborative Environment

In this section we introduce our collaborative environment. First, we present how the project is distributed among teams, how teams are formed, and the different roles played by students in the project. Next, we describe the system from the student point of view, explaining the different features. The view of the system will vary depending on the role of a student. For example, only team-leaders may communicate with other team-leaders. Finally, we present how we add agents into the collaborative environment. In particular, we define the theoretical models underlying the behavior of agents.

3.1 Distributing the Work

One of the main objectives of our system is that students learn good habits in programming (see e.g. [13]). A fundamental issue when developing real applications is the interaction with other programmers: Distributing the work, assuming/delegating responsibilities, putting pressure on other members of the team, etc. Thus, after a first course covering the basic concepts of programming, students confront a final project. In order to simulate a real software engineering application, this final assignment will be developed collaboratively by several students, organized in different groups. In this case, each project is assigned to a group of fifteen to twenty students. Besides, depending on the nature of the project, this group of students will be split into three to five teams. Given the fact that students still lack the necessary skills to develop on their own the whole project, the teacher plays a fundamental role in the specification of the different tasks. In particular, she will play both the roles of project owner and project manager.

In order to improve the coordination between teams, simple organizational design of projects structures are considered. There exists a project owner and a project manager. Each team has a *team-leader*. In addition to their duties as team-members, they have some additional responsibilities. First, they are in charge of the work assigned to the team. For example, they can (dynamically)

decide to change the distribution of tasks in their teams, in the case that a member is too slow or is having problems with a particular task. They will also be responsible for putting pressure on their team-mates as well as on other teams. Actually, they are the *interface* of the team with the *outside world* (other teams and project manager/teacher).

As it is well known (see e.g. [6]), the learning efficiency of the individual students is strongly influenced by the relations among the members of the group. Thus, a critical teaching decision consists in the design of effective groups that guarantee educational benefits to the individual students. Fortunately, during the course, the learning of students was *controlled* by WHAT, so the teacher has been provided with relevant information about the characteristics of students. Hence, the composition of teams will be mainly determined by the attributes defined by WHAT (considering both static and dynamic classes of students). In particular, team-leaders will be appointed on the basis of their trajectory in the course.

The project manager (the teacher) is in charge of dividing the project into modules. Each module will be assigned to a different team of students. She will also perform the second refinement where each module is split into a set of tasks. Let us remark that this is not a typical software engineering strategy. In fact, it is a quite erroneous strategy in normal applications, as the project manager should abstract the distribution of the work inside each team. However, as we are dealing with first-year students, students are not yet prepared to perform these organizational tasks. Let us remark that even though tasks are initially distributed by the project manager, we allow students to *commerce* with their assigned tasks. Thus, each student can propose exchanging some tasks to other member of the team. Besides, she can also accept or reject such a proposal. In case students commerce adequately, a better distribution of tasks will be done, so that students specialize themselves on different topics, improving the overall performance of the project. As a byproduct we expect that students increase their *negotiation* skills, so that they can perform the tasks that are more attractive/easy for them. Let us remark that students are not allowed to commerce with members of other teams. This is a usual restriction and pretends to avoid tasks going out of the scope of the corresponding module.

3.2 The Student Point of View of the System

Students may access two different interfaces inside our collaborative environment: One devoted to *programming* and another one devoted to *communications*. The programming interface (left part of Figure 1) provides an area to write programs, another one to test programs already completed (by defining test cases), and several options to help the user (compile, run, load, save, etc.). In addition to that, students can always check the status of the functionalities of the project. They may consult the date/time of the last release of a given task, whether a finished task has been already tested, etc. Students can freely choose which of their pending tasks they want to work on. The state of this task is automatically updated in the programming environment. Finally, as our system is connected to

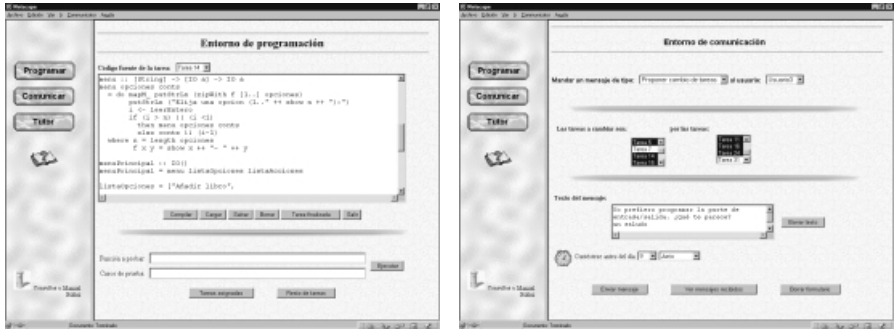

Fig. 1. Programming (left) and communication (right) environments

a Haskell compiler, both compilation errors and results from actual executions can be shown to the user.

The communication with the rest of users is done through a different interface (see right part of Figure 1). A student can either send messages to other peers, or read received messages (a list of her sent/received messages is always available). In order to be able to automatically monitor the communications among users, before a message is sent, students have to select the kind of message they want to transmit. By doing so, log files store all the relevant information about communications. These files are used to detect some characteristics (initiative, participation, leadership, etc.) of each student. The messages that can be sent are classified according to the following categories:

- Teams coordination.
- Tasks commerce.
- Project status.
- Other messages.

The first category of messages is enabled only for team-leaders, as they are responsible for coordinating the different modules of the project. They can put pressure on other teams to work on a specific functionality, to fulfill a deadline for performing a partial evaluation of the project, etc. Team-leaders will be also allowed to communicate with the project manager/teacher.

Regarding commerce of tasks, students may either accept or reject other users proposals. Moreover, students are also allowed to propose exchanges of a list of tasks with other users. Finally, team-leaders can also decide to redistribute the tasks of other member, just in case that a member is delaying the whole group.

Regarding the project status, a student can ask other members of her own team to work harder on a particular part of their tasks (because that part is critical for one of her pending tasks). The student has also the possibility to notify that she has already finished one of her assigned tasks. Finally, she can also detect that a functionality provided by other member is incorrect, and notify it. Let us remark that students do not have access to the source code

being developed by other members of the group, but only to the interface of the modules. Thus, students must abstract implementation details and treat the programs of the other members as *black boxes*.

The last group of messages includes those communications associated with verbal conversations, so that, for instance, students can ask for help to other members of the team. Following the approach presented in [10], we allow students to send different kind of messages, like questions, answers, opinions, suggestions, encourages, acknowledgments, etc. As we have said, the monitoring of these communications allows us to detect the roles students are playing inside the teams, like discovering who is the real *leader* of the team.

3.3 Including Agents into the Collaborative Environment

In addition to students, intelligent agents will be added to the teams. Actually, students will not be informed about the fact that some of their team-mates could be *virtual* students. Thus, in order to hide their nature, they communicate using the same mechanisms as the rest of students, the only difference being that they do not need to use the web interface. Let us remark that, from the user point of view, both agents and real students are seen in the same way, as their user identities do not reveal the nature of them.

The characteristics of the agents will depend on the team members, so that they balance the *necessities* of the team. For instance, in case there are many brilliant students,[5] the behavior of the agent could be that of a not so brilliant student. Let us remark that there will be as many different *kinds* of agents as considered classes.[6] Thus, there will be different types of agents, depending on the skills learned in the course, depending on whether they learn fast or not, etc. Regarding the *malicious* adjective, as we already commented in the introduction, our agents are not always really *bad*: They mainly simulate the behavior of students. Let us remark that agents have access to the full code of two implementations of the project: the correct and complete one implemented by the teacher, and the incomplete one that the students are developing. That is, in order to provide programs solving their assigned tasks, they will (initially) use the right code (as provided by the teacher). However, they may introduce some errors. In order to write a wrong program, they will modify the right code according to the most commom mistakes of the (classes of) students that they are simulating. This information will be provided by WHAT (the tutor that students have been using along the course).

As pointed out in [6], the characteristics of the best learning group, for a particular user, are different at each moment. Thus, team components should ideally change dynamically. However, this solution is not feasible for students. Fortunately, our virtual students will dynamically adapt their attributes so that

[5] Let us remark that the student model is obtained by using WHAT.

[6] Formally, this number will be given by the cardinal of the set containing all the possible combinations of different (static and dynamic) classes (that is, a cartesian product of the corresponding sets).

they fit better the proper characteristics of the team. For instance, if a team temporarily lacks a person who press the rest of the members, an agent could adopt such behavior. Let us remind that the system dynamically detects the characteristics of each of the students, knowing whether they have initiative, participation, enthusiasm, etc. In addition, agents do not only balance groups/teams with behaviors that the components do not have. They can also strengthen some skills of students, as self-confidence and communication expertise. For instance, an agent could be lazy, so that it delays the whole group. The team should press the lazy member, and even decide to distribute its assigned tasks among the rest of members. We find this feature particularly useful for first-year students, as they are only used to solve problems individually.

In the following we present how agents react to the different interactions with real students. As we do not want students to know that some of their mates are virtual students, the way in which agents perform communications should be as similar as possible to the behavior of a real student. Obviously, it is not easy to perfectly simulate the communications performed by a student, as that is nearly equivalent to passing the Turing test. Logically, our aim is more modest: our agents will be able to ask simple questions, and also to answer them by using simple structures, as we show in the following paragraphs.

Regarding the commerce of tasks, the agent assigns a programming cost to each of the tasks. It suggests exchanging tasks when it can decrease its programming effort, and it accepts exchanges if they do not increase it. Actually, the underlying model is based on (a simplification of) PAMR [11]. This process algebra is very suitable for the specification of concurrent systems where *resources* can be exchanged between different components of the system. In our case, agents have a *utility function* relating all the tasks of the team, according to the programming costs. So, if a member of the group proposes an exchange, the agent will compute the corresponding utilities (before and after the exchange) and it will accept the offer if its utility does not decrease.

An important feature in the implementation of the agents consists in the definition of adequate response times. For example, if an agent is assigned to implement a difficult function, it could not be credible that the agent provides the code after a few seconds. It would be also erroneous to consider that the agent returns its programs after a fix amount of time, regardless the complexity of the corresponding program. In order to add realism to our agents, we have considered a model based on continuous time semi-Markov chains extended with probabilistic information (this theoretical framework is based on the work presented in [9]). Next, we briefly sketch how these models guide the temporal behavior of agents. For each of the tasks assigned to the agent, its response time will be defined by a discrete probability distribution function associated with the answer that the agent will provide. As we said before, agents may provide either a right answer (with probability p) or different (wrong) variations of the correct answer (the total probability of these answers is equal to $1-p$). Besides, each probabilistic decision is associated with a random variable. These random variables decide the time that the agent will be delayed until the corresponding

program is presented to the group. It is important to note that random variables will depend on the kind of answer that they are associated with. More precisely, the *wronger*[7] an answer is, the faster the agent will come out with a program. For the sake of simplicity, assume that all the random variables are exponentially distributed. Consider a *very wrong*, a *not so wrong*, and a *right* answers, having associated with them exponential distributions with parameters λ_1, λ_2, and λ_3, respectively.[8] We should have $\lambda_1 > \lambda_2, \lambda_3$ while $\lambda_2 \approx \lambda_3$. Let us remark that the probabilistic-stochastic model is not static. In other words, both probability distributions and random variables will vary not only according to the complexity of the given task but also according to the corresponding response time of other members of the group. For example, if the agent detects that it is faster than the rest of members, then forthcoming random variables will be *delayed*. In this line, agents consider a well-known property: Productivity (notably) increases as the deadline approaches. Finally, if the rest of the group urges the agent to finish a task, it may (probabilistically) decide to reduce the corresponding delays. However, if these delays are shortened, the probability distributions are also recomputed so that the probability associated with wrong answers increases.

Finally, an agent can also press other members of the project to work on a task, as the agent knows the tasks dependencies graph. In the same way, agents can also detect that a functionality provided by other user is wrong. Given the fact that to check equality of functions is undecidable, agents must use a *testing* procedure. Taking into account that agents have access to the right solution, provided by the teacher, they generate test cases covering the most problematic cases of the program. Then, they compare the results returned by the students implementation with the ones returned by the teacher implementation.

4 Conclusions and Future Work

In this paper we have described a collaborative environment for the development of medium-size programming projects. The system is designed to help first-year students to learn good programming habits. In order to obtain this objective, students need to collaborate to solve a common programming assignment. An important feature of our system is that not all the members of the group are real students (but students do not know this fact). Virtual students are added to the system in order to form effective groups that guarantee educational benefits to individual students. From the student point of view, our agents are not always helpers, as they can produce wrong programs, and they can even delay the work of the whole group. Nevertheless, agents are *real helpers*, as they help students to learn how to work inside heterogeneous groups.

As future work we plan to develop a similar tool for other programming languages, namely Pascal and Java. In fact, one of the main aims of our work

[7] The relative correctness is given by the type and number of errors that the agent is introducing.

[8] Let us remind that the mean of an exponential distribution with parameter λ is given by $\frac{1}{\lambda}$. So, the bigger λ is, the faster the delay is consumed.

is to compare the difficulties that first-year Mathematics students have learning different languages. By providing similar tools for the different languages that have been used in our department, we hope to help clarifying which language suits them better. As the environments will be analogous, we will be able to really compare the languages, without being disturbed by their environments. Besides, currently our system is now only in Spanish, as it is designed to be used by our students. However, it is trivial to translate it to any other language.

References

1. E. Aïmeur, H. Dufort, D. Leibu, and C. Frasson. Some justifications for the learning by disturbing strategy. In *AI-ED 97, World Conference on Artificial Intelligence and Education*, 1997.
2. E. Aïmeur and C. Frasson. Analyzing a new learning strategy according to different knowledge levels. *Computer and Education. An International Journal*, 27(2):115–127, 1996.
3. E. Aïmeur, C. Frasson, and H. Dufort. Co-operative learning strategies for intelligent tutoring systems. *Applied Artificial Intelligence. An International Journal*, 14(5):465–490, 2000.
4. G. Ayala and Y. Yano. GRACILE: A framework for collaborative intelligent learning environments. *Journal of the Japanese Society of Artificial Intelligence*, 10(6):156–170, 1995.
5. H.U. Hoppe. The use of multiple student modeling to parameterize group learning. In *7th Conference on Artifical Intelligence in Education*, pages 234–241, 1995.
6. A. Inaba, T. Supnithi, M. Ikeda, R. Mizoguchi, and J. Toyoda. How can we form effective collaborative learning groups? –Theoretical justification of "opportunistic group formation" with ontological engineering–. In *ITS 2000, LNCS 1839*, pages 282–291. Springer, 2000.
7. P. Jermann, A. Soller, and M. Muehlenbrock. From mirroring to guiding: A review of state of the art technology for supporting collaborative learning. In *1st European Conf. on Computer-Supported Collaborative Learning*, pages 324–331, 2001.
8. N. López, M. Núñez, I. Rodríguez, and F. Rubio. WHAT: A Web-based Haskell Adaptive Tutor. Submitted for publication, 2002.
9. N. López and M. Núñez. A testing theory for generally distributed stochastic processes. In *CONCUR'2001, LNCS 2154*, pages 321–335. Springer, 2001.
10. M. McManus and R. Aiken. Monitoring computer-based problem solving. *International Journal of Artificial Intelligence in Education*, 6(4):307–336, 1995.
11. M. Núñez and I. Rodríguez. `PAMR`: A process algebra for the management of resources in concurrent systems. In *FORTE 2001*, pages 169–185. Kluwer Academic Publishers, 2001.
12. S.L. Peyton Jones and J. Hughes. Report on the programming language Haskell 98, 1999. `http://www.haskell.org`.
13. A. Vizcaíno, J. Contreras, J. Favela, and M. Prieto. An adaptive, collaborative environment to develop good habits in programming. In *ITS 2000, LNCS 1839*, pages 262–271. Springer, 2000.
14. M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.