

Proceedings

# Innovations in Teaching Programming

Ray Kemp and Ben du Boulay

MONDAY, JULY 21ST, 2003

VOLUME VII OF AIED2003 SUPPLEMENTARY PROCEEDINGS

## WORKSHOP SCIENTIFIC COMMITTEE

### Program Chair:

- **Ray Kemp** (Co-Chair) *Institute of Information Sciences and Technology, Massey University, Palmerston North, New Zealand*
- **Ben du Boulay** (Co-Chair) *School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK*

### Committee members:

- **Paul Brna**, *Northumbria University, UK*
- **Richard Cox**, *University of Sussex, UK*
- **Dianne Hagan**, *Monash University, Australia*
- **Elizabeth Kemp**, *Massey University, New Zealand*
- **Rudi Lutz**, *University of Sussex, UK*
- **Tanja Mitrovic**, *University of Canterbury, New Zealand*
- **Paul Mulholland**, *Open University, UK*
- **Robert Rist**, *University of Technology, Sydney, Australia*
- **Pablo Romero**, *University of Sussex, UK*

## TABLE OF CONTENTS

Foreword.....	396
<b>FULL PAPERS</b>	
Towards a debugging tutor for object-oriented environments .....	399
<i>B du Boulay, P Romero, R Cox, R Lutz</i>	
Supporting teaching/learning activities using an environment based on virtual tools .....	408
<i>L Giraffa, S Marczak, G Almeida</i>	
Adopting exploratory + collaborative learning in an adaptive CSCL environment for introductory programming .....	417
<i>A Gogoulou, E Giyku, M Grigoriadou</i>	
Model-based generation of demand feedback in a programming tutor .....	425
<i>A N Kumar</i>	
A didactic interface in a programming tutor .....	433
<i>M A de Lemos, L N de Barros</i>	

## FOREWORD

Systems to teach and monitor programming have been developed and evaluated throughout the history of AIED. In many ways, programming has been a very productive domain in the evolution of most aspects of the field, including student modelling, knowledge representation and pedagogy. Effective programming requires a range of problem-solving and diagnostic strategies. Program code can provide a rich insight into the reasoning processes of its creator. Programming, therefore, provides an interesting domain for studying learning and cognitive processes.

The aim of the workshop is to bring together researchers who have new ideas or have developed innovative techniques for using the computer to help students learn programming. In addition to contributing to progress in understanding the learning process in general, we hope that we will be able to have a direct impact on support for teaching programming. More than ever, this is an important area particularly in tertiary institutions where there are increasing numbers of students wishing to learn the subject, and where it is difficult to provide the individual tuition that they need. There has been a recent proliferation of institutions investing in distance learning, too, and we need to provide appropriate methods of teaching this key subject to students learning remotely.

It is clear from the papers submitted to this workshop that there are a wide range of approaches that are currently being taken to the teaching of programming. Historically, program synthesis has been the focus of many systems. Although there are obviously still a number of groups working in this area, most of the submissions consider other aspects including teaching and aiding error detection/correction, predicting program behaviour, understanding the program development process and appraising existing programs. Techniques used include agents, scaffolding, kbs methods, visualizations and case-based reasoning. Correspondingly, there are a wide range of teaching styles including collaboration, coaching, problem based learning and feedback on demand.

du Boulay, Romero, Cox and Lutz look at a neglected area of the programming process – debugging. They argue that many environments provide debugging tools but few offer any *interactive* assistance to students who need to learn how to carry out this vital task. Students are usually given general guidelines and heuristics, but, for the most part, are expected to develop expertise by trial and error over a period of time.

Two experiments were designed to determine whether specific patterns of behaviour were associated with effective debugging. Students were provided with a software debugging environment for Java which included both textual and visual representations of the programs. In general, the more successful students were better able to use the trace and visualization features to spot discrepancies than the unsuccessful ones.

Using their experience from these experiments, the authors indicate how a debugging tutor might function. By tracking students' focus of attention, the system could monitor their behaviour and re-direct their attention as required. For example, if the student was spending a great deal of time examining the code then the system could suggest that they take a look at one of the diagrams instead. Determining *why* a student is exhibiting a particular pattern of behaviour is the key, of course, since the feedback needs to be geared to the causes of individual patterns of attention switching.

Another way in which students could be helped is by tracking their tracing behaviour. Assuming the system knows where the bugs are, the students could be steered away from unproductive parts of the code, and guided towards points where the errors occur or manifest themselves.

Giraffa, Marczak and Almeida have accumulated a number of tools and techniques over a long period to aid their students in an Algorithms and Programming course. Their approach puts the onus very much on the students. Although the framework for the course

is traditional, with a weekly schedule of classes and set assignments, the students have many available resources and choose which they think are most appropriate for the task in hand. The authors characterize this as a constructivist approach but also include teacher coaching and assistance.

In the current version of their scheme, the human teacher has control of all the resources including a blackboard, the content of help systems and forums. A system called PROOGRAMA has been designed to take over the management of much of this work. An agent called AMIGO facilitates the interaction between the teacher and students, and also organizes the information about each student. Other tools aid the student in program construction, mediate on-line discussion, and help the teacher maintain a FAQ file. The authors believe that this scheme will alleviate much of the work that the teacher currently has to carry out in order to maintain the resources and interactions with and between the students.

Gogoulou, Giyku and Grigoriadou have developed a set of guidelines called ECLiP for helping tutors design an integrated set of learning activities in any domain. It is a three step process to ensure the tutorial material helps students acquire, construct, and apply/refine their knowledge. Collaboration with other students is a central theme, and this mode of learning encourages students to externalize their knowledge, justify their points of view and evaluate their own and other solutions. They have also designed an environment called SCALE which facilitates collaborative learning. It has intelligent agents for providing encouragement and for guiding the learning activities. These activities address learning outcomes at four levels: comprehension, application, checking/critiquing and creation.

In their paper, the authors explain how the ECLiP framework could be applied in the SCALE environment to provide suitable activities for students learning introductory programming. Using typical scenarios, they assess in what ways SCALE could be used to facilitate learning in this domain. For example, at the knowledge acquisition stage, the student may learn about loops by being given problems that require this construct. The system will facilitate discussion within collaborating groups to encourage them to discover a way of repeating operations. Alternatively, if students are required to work alone, SCALE will provide communication and scaffolding tools to enable them to work independently towards a solution. Similarly, proposals are made for how their scheme could help during the knowledge building and application/refining stages.

Kumar believes that an effective way of learning how to program is to have explanations of what individual pieces of code are for and how they interact. Obviously, comments on lines of code can be provided manually for specific programs but he wants to be able to provide automatic generation of explanations for any arbitrary program. To this end, he has used model-based reasoning to model C++ code. Such schemes have been used routinely to simulate the behaviour of electrical and mechanical devices. Here the approach is applied to computer programs.

The program is considered to be a unit which is composed of a number of functions and each of these contains declarations and statements, and so on. As the program is run, the interaction between these components is modelled. At each level, the behaviour of a component is specified completely by the behaviour and interaction of its subcomponents rather than by employing a state-based approach. The only exception is at the bottom level where the states of variables are represented in state transition diagrams.

Typically, the tutor will have a program or subprogram that he or she wishes the student to understand. It may be correct, or it may contain errors. Kumar's interpreter executes the code and simultaneously generates an explanation of what is happening. In addition, if there are any errors then component level error messages are generated. The feedback can be adjusted depending upon the student's level of knowledge. For example, a

novice would be given a complete description of the behaviour of components. This scheme has already produced encouraging results in experiments where students are required to predict behaviour of (output from) a program and identify semantic and run-time errors.

de Lemos and de Barros believe that the interface to any ITS is crucial, and should provide suitable support for the student so they can utilize cognitive knowledge to solve the task in hand. In their scheme for teaching program development, suitable building blocks are provided. Using these the student is encouraged to develop an appropriate mental model of the problem and of the program.

For a given programming problem, there are high level goals that might be chosen in order to produce a program to solve this problem. Corresponding to each goal there is a plan to achieve that goal, and each plan can be decomposed into actions. These items are stored in a *cognitive tree*. The actual C code corresponding to the actions selected by the student is automatically generated and shown in a separate window. The idea is that they can learn to understand programs by producing these high level mental models and then observing the corresponding code that is produced.

Currently, feedback can be supplied when the C program is executed. If the mental program model is incorrect then the interpreter produces errors. The authors plan to extend the system to provide appropriate feedback to students when they choose incorrect items from the cognitive tree, or try to place them in the wrong place in their high level program.

As can be seen, the problems addressed and approaches that are currently being taken to teaching programming are many and varied. We anticipate that there will be a lively debate of these ideas and methods. In addition, we would also like to consider other timely issues at the workshop such as employing problem-based teaching, teaching OO, new technology such as PDAs, the web, and digital whiteboards and the wider challenge of incorporating the analysis and design of systems rather than focussing on the coding process in isolation.