Intelligent Student Support in the FLIP Learning System based on Student Initial Misconceptions and Student Modelling

Sokratis Karkalas and Sergio Gutierrez-Santos

Department of Computer Science and Information Systems, School of Business, Economics and Informatics, Birkbeck, University of London, Malet Street, London WC1E 7HX, U.K.

Keywords: Student Modelling, Exploratory Learning, Rule-based System.

Abstract: The 'FLIP Learning' (Flexible, Intelligent and Personalised Learning) is an Exploratory Learning Environment (ELE) for teaching elementary programming to beginners using JavaScript. This paper presents a sub-system in FLIP that can be used to generate individualised real-time support to students depending on their initial misconceptions. The sub-system is intended to be used primarily in the early stages of student engagement in order to help them overcome the constraints of their Zone of Proximal Development (ZPD) with minimal assistance from teachers. Since this is an ongoing project we also report on issues related to potential changes or enhancements that will enable a more optimised use under real classroom conditions.

1 INTRODUCTION

This paper describes the design of an Exploratory Learning Environment (ELE) for teaching introductory programming to University students. It has long been established that teaching and learning programming at that level is particularly challenging for tutors and students. Computer programming is one of the major challenges in computing education (Bennedsen and Caspersen, 2007) and as composition-based task it imposes major problems to novices. Students at that level may suffer from a wide range of difficulties and deficits (Robins et al., 2003). Programming courses are considered to be hard to attend and statistically that leads to high dropout rates (Robins et al., 2003; Bennedsen and Caspersen, 2007).

It is evident that this part of computing education is particularly sensitive and as such it requires a major effort in terms of design, preparation and implementation from academic staff. Programming is a craft and it can only be learnt by doing. Most of the learning takes place in computer laboratories, where students under the supervision of tutors, attempt to solve programming exercises. It is obvious that there is an analogy between the effectiveness of the processes that take place during the practical exercises in the lab and the actual learning outcome that can eventually be achieved. If students can utilise as much as possible of the resources available and the system is flexible enough so that individualised support is provided in a timely fashion then it is more likely for them to achieve an optimum result.

The focus of this paper is to present a system that is able to provide a flexible and personalised learning experience to the students without extra investment in terms of resources. Students will be able to work on their own in an exploratory manner and receive immediate attention from intelligent virtual tutors whenever needed. These tutors can provide individualised help based on the students' activity logs and the initial misconceptions they may have.

2 RELATED WORK

The need for intelligent computer-supported help in teaching is not a recent development. Attempts towards this direction had started in the late '70s. Typical examples of these attempts are intelligent tutoring systems (ITS) like (Brown and Burton, 1978; Reiser et al., 1985; Johnson and Soloway, 1985) targeting bugs in procedures. A rather different approach is the ELM-ART (Brusilovsky et al., 1996) which is more an intelligent adaptive courseware delivery system rather than a buggy ruler. Other more recent attempts include (Mitrovic, 2003; Sykes and Franek, 2003; Holland et al., 2009; Peylo et al., 2000). According to (Brusilovsky and Peylo, 2003) systems may be intelligent and/or adaptive. An intelligent system is one that applies techniques from the field

Karkalas S. and Gutiérrez Santos S..
Intelligent Student Support in the FLIP Learning System based on Student Initial Misconceptions and Student Modelling.
DOI: 10.5220/0005127603530360
In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development* (KEOD-2014), pages 353-360
ISBN: 978-989-758-049-9

of AI in order to provide better support to the users. An adaptive system is one that takes into account the user's individuality in order to change the way it interacts with the users. The system in (Mitrovic, 2003) is intelligent but not adaptive. It compares student solutions to correct solutions specified by tutors using domain knowledge that is represented in the form of constraints. Constraint-based modelling is also used in J-LATTE (Holland et al., 2009). This system teaches Java and provides support in both design and implementation aspects of the language. Another system that teaches Java is (Sykes and Franek, 2003). This is an expert system supported by decision trees to provide intelligent support and adaptiveness. A rather different approach is adopted by (Peylo et al., 2000). Domain knowledge in this case is represented as an ontology. The system is web-based and teaches Prolog.

All the aforementioned systems, despite their differences, converge in one thing. They all presuppose the selection of a given task and they offer guidance in a controllable manner. The difference between FLIP and systems like the above is the fact that FLIP is an Exploratory Learning Environment (ELE). Typical ITS of the past are designed to provide assistance to problem-based learning (PBL) scenarios (Savery, 2006) in a very controllable way. A student is given a specific well-defined problem to solve and is not 'allowed' to make mistakes during the process. The system automatically detects mistakes and corrects them. The scenario with FLIP is quite different. Students are given open-ended problems (inquiry-based learning scenarios (Savery, 2006)) and try to discover knowledge in an exploratory manner. FLIP does not intervene in this process in a strict and intrusive way. The process is not controllable. Help is always available but is only given on demand. Another important difference is that FLIP is designed to target student misconceptions and not bugs in code. A bug is a section of code that does not conform to the program specification. In FLIP there is no program specification at that level. Furthermore, bugs may not always be related to student misconceptions. In that respect, fixing a bug might not be equally beneficial to fixing a misconception in terms of learning outcome.

3 THE PROBLEM

Teaching programming is a very labour intensive process especially if the intended class is a cohort of beginners. Programming is a craft and as such it involves hands-on laboratory work. This is normally an Inquiry-Based Learning (IBL) or Problem-Based Learning (PBL) based practical exercise that takes place in a computer laboratory. Learners develop their knowledge through this process under the assistance of a facilitator (tutor). Students normally ask for assistance every time they feel they cannot cope with some problem relating to the syntax or the logic of their program. Debugging is a tedious process and can be a very daunting experience especially for young programmers. If the student cannot receive the amount of help needed in a timely fashion to overcome the problem this can have a significant effect on their confidence and consequently a negative impact on their studies.

Ideally the learning experience in the computer laboratory must be a sequence of successive iterations that follow Kolbs learning cycle (Kolb et al., 1984; Konak et al., 2014). The students' engagement with the task should be a cyclical process. In every round the students make an attempt to develop something that moves them closer to the completion of the given task. This attempt is interrupted when they hit the boundaries of the inner circle within their particular ZPD (Vygotskiĭ et al., 1978). This is where the facilitator comes into play. The student asks for help and the available tutor inspects the problem and tries to motivate the student to move on by providing help in small increments. The intention in this case is to provide only enough and relevant help so that the student can overcome the issue and carry on with the rest of the development without problems. During this little interval the student and the facilitator are engaged within a conversation where the facilitator helps her to understand the issues raised and to develop an abstract conceptualisation that is then transformed to active experimentation in the next cycle (Kolbs Learning Cycle) (Kolb et al., 1984; Konak et al., 2014). After each successful cycle a little bit of learning is achieved and gradually (possibly after many iterations) the ZPD (Vygotskiĭ et al., 1978) circles expand. This, of course, is based on the assumption that there is real substance in these cycles and actual learning is achieved. If students cannot receive the help they need in a timely fashion then it is less likely to achieve their full potential in the given time for laboratory work. In other words it is less likely to perform as many cycles as they would if help was immediately available. Another issue is related to the relevancy and the amount of help. Experienced tutors that are aware of the typical student misconceptions can easily switch from problem to problem and focus quickly on the type of help that needs to be provided both in terms of size and relevancy. Tutors, though, are human beings and their performance might be influenced by a wide variety of factors: personal, social

etc. Therefore, relevancy and proper size of help cannot be guaranteed either.

Human tutors cannot be substituted by machines. This is not the intention of this work. The intention is to delegate as much as possible of these tasks to intelligent agents. If the system knows what student misconceptions to anticipate and how to provide relevant help then the students will be able to receive immediate help whenever needed. This help will always be focused and consistent with the actual need. That will also keep the interaction between the students and the teachers to a minimum and promote independent and constructive learning. The students will learn to work autonomously and utilise as much of their background knowledge as possible. This is in full compliance to the constructivist approach (Huitt, 2003; Bruner, 1966; Vygotskiĭ et al., 1978), which is today's predominant educational paradigm: new knowledge is constructed on the basis of existing knowledge. The students will be able to engage in a highly constructive and rewarding learning experience where they will appreciate their own contribution and they will build up their confidence. The process will also be more inclusive since the amount and depth of help received will always be the same regardless of sociocultural factors. On the other hand tutors will be given space to offer more high level support and deal with aspects of support that machines are unable to cope with. The net result is expected to be a better and more effective learning experience which in turn will result in higher student retention in a subject that traditionally suffers from high dropout rates (Robins et al., 2003).

4 MODELLING CONCEPTS AND LEARNERS

The two most important entities in FLIP are concepts and students (learners). The main components of the learner model can be summarised as follows:

1. Misconceptions identified (journal of temporal data): This information can be a useful indicator of the student's progress. If the system keeps recording the same misconceptions, then probably no progress is being made. The frequency of occurrence per session can be particularly useful in that respect. This data can also be useful for the automatic production of recommendations to the student in terms of material, exercises etc.
2. Misconceptions covered (journal of temporal data): This information can be used to make teaching adaptive to the student's needs. If the help given the first time was not enough, then the system can provide a deeper analysis of the problem possibly using the aid of visual or other tools. If it is established that the system cannot provide more help, then the human tutor can be asked to intervene.
3. Frequency of help use: This information, in conjunction with the previous sets, can be used to show the usefulness of the intelligent support subsystem. It can also be used to profile students' behaviour. These profiles can then be used to provide more insight to the intelligent support tools.
4. Student activity/inactivity: Activity in this case means actions performed in the editor. If the student uses the editor to write and experiment with code then the system considers the session active. Long inactivity periods during the laboratory sessions might be an indication of problems. These problems cannot be detected by the intelligent support subsystem because the detection mechanism is dependent on code. If inactivity is identified, then the system provokes the intervention of the tutor.
5. Use of language reference: In practical terms this can enhance the student's experience by providing direct access (links) to parts of the language reference that are more relevant to the student's needs. These parts can also be prefetched and cached for efficiency (responsiveness).

As said above concepts represent known student initial misconceptions. In practical terms these misconceptions correspond to certain formations of source code. These formations (patterns) could effectively be represented as sets of characteristics in the form of logical expressions. The first step in the process of modelling is to identify the patterns that indicate such misconceptions. As stated before in this text, FLIP's intended target group is students with no prior experience in programming. It is assumed that these students will be using the system in an introductory programming course that does not fully cover all the complexities of the JavaScript language and the more advanced object-oriented features of it. Therefore, in this very first stage of FLIP the intention was to capture only the misconceptions that correspond to basic language use and elementary algorithmic thinking. In this project the concept elicitation process involved 111 students from 3 introductory programming courses. Two of them were postgraduate courses taught in Java comprising 42 and 44 students respectively and the remaining one was a undergraduate course taught in JavaScript. The intention was to identify misconceptions that are com-

mon in both languages. The research was conducted at the Department of Computer Science and Information Systems, Birkbeck, University of London during the academic years 2012-13 and 2013-14. Collecting data from the JavaScript course proved to be particularly useful since we were able to identify language-specific misconceptions too. The general research approach followed was more of an exploratory nature rather than confirmatory. We thought that it would be best not to constrain the domain by interviewing volunteers using existing classifications of already recognised misconceptions in the literature. Therefore, the data collection technique used was the systematic registration of every issue that took place in the practical laboratory sessions during these three terms (following a process similar to Grounded Theory (Strauss and Corbin, 1994)). The Java courses were fast-paced, intense courses and covered much more material and in greater depth. Therefore, the data collection part for these courses took place only for the first four sessions of each term. The data collection was carried out by two teaching assistants and the lecturer. The teaching assistants collected the data which was then reviewed by the lecturer. The author was a teaching assistant in the Java courses and the lecturer of the JavaScript course. The co-author was the lecturer of the Java courses. After the data was collected we classified it using the Concept Inventory (CI) presented in (Goldman et al., 2008). According to the results, the following categories were applicable in our sample:

We did not expect our sample to exhaustively cover all the concepts identified by the Delphi experts in (Goldman et al., 2008). The concepts covered were 16 out of 32 in the CI (50%). The 3 concepts in the grey area were not part of the original CI proposed in (Goldman et al., 2008). The actual misconceptions that emerge in each course may depend on many factors like the students' background, the language and programming paradigm used, the material covered, the intended learning outcomes, the tutors' contribution and so on. Assuming that these things remain fixed for our courses, the aim of this work was to identify what is the actual need of our students in terms of help. The fact that the concepts identified cover a large part of the Delphi CI and overlap at a proportion of 60% with the student misconceptions identified in (Kaczmarczyk et al., 2010) is an indication that the elicitation process was effective. The asterisk under the 'D' column in the above table indicates that the same concept was also found in (Kaczmarczyk et al., 2010). There were no object-oriented misconceptions identified (apart from one) since JavaScript is a prototype object-based (class-less) language and the related misconceptions would not be relevant.

Table 1: Concept Categories Identified in the Sample.

Procedural Programming Concepts			
	ID	Topic	D
1	PA1	Parameters/Arguments I	
2	PA2	Parameters/Arguments II	
3	PA3	Parameters/Arguments III	
4	PROC	Procedures/Functions/Methods	
5	TYP	Types	*
6	BOOL	Boolean Logic	
7	COND	Conditionals	*
8	SVS	Syntax vs Semantics	
9	AS	Assignment Statements	*
10	SCO	Scope	

Object Oriented Concepts			
1	PVR	Primitive and Ref Variables	*

Algorithmic Design Concepts			
1	IT2	Iteration/Loops II	*
2	IT3	Iteration/Loops III	
3	IT4	Iteration/Loops IV	
4	IT5	Iteration/Loops V	
5	REC	Recursion	
6	AR1	Arrays I	*
7	AR2	Arrays II	
8	AR3	Arrays III	

Furthermore, most of the difficult aspects of object-oriented development, like inheritance, are not part of the JavaScript introductory course. Also, it was probably very difficult for the program design concepts to reveal themselves in the code, at least in this initial stage. There have been recorded incidents in the lab regarding design-related concepts but there was no usable code involved. Therefore, this category was excluded from the set. The CI that was formed after our analysis is presented in the Appendix.

5 SUPPORTING STUDENTS

FLIP utilises a rule-based expert system that interacts with the student when there is a misconception to be resolved. It takes the role of the teacher and repetitively exchanges information with the student in order to assess the current situation, identify the problems and provide individualised support whenever possible.

This system accepts two inputs: rules (misconceptions) and facts (current student understanding). Rules are inserted by experts and form the knowledge base of the system. The conditional part of these rules corresponds to one or more characteristics identified in the code (facts). The consequent part corresponds

to the action that needs to take place in case they fire. Facts are inserted dynamically into the system during the development process. The student's code is statically analysed and divided into distinguishable characteristics that can be extracted in the form of a vector. These characteristics take the form of atomic formulae (variable/predicate pairs) and are inserted automatically into the system as facts. One or more characteristics combined with Boolean operators can form an equivalent representation of a potential misconception. Facts can also be generated by the system itself.

5.1 A Typical Usage Scenario

The student is trying to do an exercise in the lab. Part of this exercise is to compute the sum of numeric values in an array. She has tried many times using different approaches but the result is still not satisfactory and she decides to ask for help. She selects the (following) problematic part of the code and presses the button 'help'.

```
1 var x = [2,5,1,8,9];
2
3 for (var i = 0; i <= 5; i++)
4 {
5     var sum = 0;
6     sum += x[i];
7 }
8
9 alert(sum);
```

The system parses/analyses the code and generates a series of facts. An extract of this vector is given in the following figure:

(index)	subject	relation	property
0	"x"	"is"	"array"
1	"x"	"location"	"{"start":{"line":1,"column":4}…
2	"x"	"length"	5
3	"s1"	"is"	"structure"
4	"b1"	"is"	"block"
5	"b1"	"location"	"{"start":{"line":4,"column":0}…
6	"sum"	"is"	"var"
7	"sum"	"location"	"{"start":{"line":5,"column":5}…
8	"l1"	"is"	"literal"
9	"l1"	"location"	"{"start":{"line":5,"column":11…
10	"l1"	"value"	"0"
11	"sum"	"value"	"l1"
12	"b1"	"includes"	"sum"

Figure 1: Facts (Code Status).

Entities that are not explicitly named in the code such as programming structures (if, for) and blocks of code are given identifiers by the analyser (s1, s2, .., sn) and (b1, b2, .., bn) respectively.

The rules that correspond to the following misconceptions get activated (their conditional part is satisfied by the facts):

1. (SCO-4) Understanding the difference between block scope and function scope. (7 Facts)
2. (AR1-1) Understanding off-by-one errors when using arrays in loop structures. (17 Facts)
3. (SVS-1) Understanding the difference between variable values and literal values. (7 Facts)
4. (SVS-2) Understanding the necessity of variables/constants. (5 Facts)
5. (SVS-3) Understanding the necessity of variables when referring to array length. (9 Facts)

The following figure shows the contents of the 'agenda' which is the component that stores the active rules in the reasoner:

(index)	name	factCount
0	"SCO-4"	7
1	"AR1-1"	17
2	"SVS-1"	7
3	"SVS-2"	5
4	"SVS-3"	9

Figure 2: Active Rules.

The number of facts (factCount) corresponds to the facts that activated the rule. In practical terms these are the expressions presented in figure 1 that have been evaluated in the conditional part of the rule.

The reasoner does not make a decision on which rule to fire. It informs the user that potential misconceptions have been identified in the code and expects her to make a decision. In general the reasoner stays as discreet as possible throughout this process. The only indication that it has a preference as to which rule needs to be fired first is the colour of the corresponding button (red). This decision is based on the number of facts each rule refers to. The rule with the smallest number of facts is probably simpler to resolve and simple problems should have a priority over more difficult ones. The following figure shows

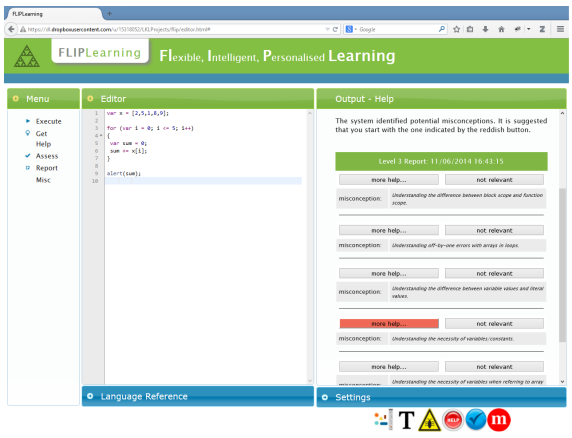

Figure 3: FLIP.

what is presented to the user after the 'help' button is pressed:

The user is free to discard misconceptions as not relevant to the problem under consideration or to select one and start a tutorial. The latter is equivalent to firing the corresponding rule. If that happens the reasoner consults the current learner model to see whether this needs to be taught for the first time. If this is the case, the system displays a brief explanation of the issue and prompts the user to read the related part of the language documentation. This is displayed in the language reference section. The user is expected to read the suggested text, possibly watch a video if there is one available, review the code and try to solve the problem alone. If that happens the user can press the help button again to verify the correctness of the change. If the change is successful then the system will identify only three problems and consider the conception learnt. If the problem is still there or the same concept has been taught in the past, then the system displays a hint as to what might be the solution to the problem. If, after the next attempt, the problem is still there, the system tries to help with a visualisation that shows a memory map and a code tracing utility.

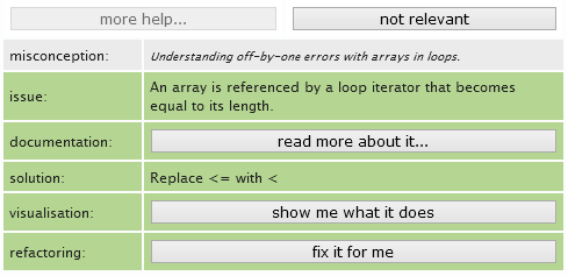

Figure 4: Levels of Support.

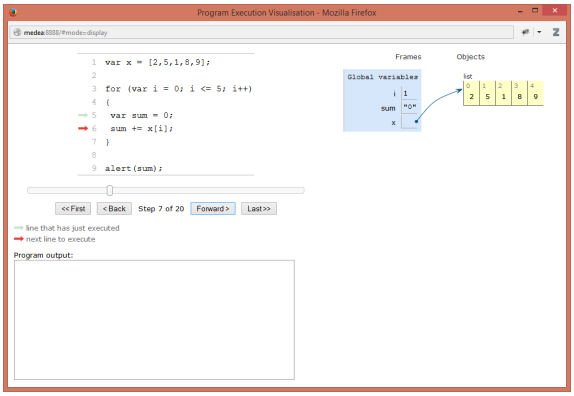

Figure 5: Code Tracing Visualisation.

If the problem persists even after the visualisation, the system offers automatic code refactoring. This is a feature that is not heavily used since there is a penalty that needs to be paid in terms of rule complexity. Therefore it might not be available in every rule. The information about the location of the problem and how it can be fixed is encoded in the rules and that makes the rules lengthy and complicated. This is an issue that will prossibly be addressed in future versions of the system. If the rule appears again after this process, then the system concludes that human intervention is required and records the relevant indicator.

6 CONCLUSIONS

This paper has described how data has been collected, analysed, and systematically classified to get a taxonomy of common misconceptions of novice programmers in JavaScript, building on previous work on other programming languages; it has also shown how this information is used to provide personalised feedback in the context of an open-ended exploratory programming session.

Preliminary testing has shown that the system has achieved its original design goals and it operates as described in the paper. An evaluation that will measure the amount of work that the system can offload from human tutors in the classroom is scheduled for the next edition of the JavaScript course in the Autumn term and thus falls out of the scope of this paper.

We envision the system to become more sensitive and receptive to user actions and it would also be desirable to obtain the users' previous knowledge, needs, interests, motives, constraints possibly from other sources like social networking platforms. Having a richer and more representative user profile will facilitate a better and more personalised learning experience.

REFERENCES

Bennedsen, J. and Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32–36.

Brown, J. S. and Burton, R. R. (1978). Diagnostic models for procedural bugs in basic mathematical skills*. *Cognitive science*, 2(2):155–192.

Bruner, J. S. (1966). *Toward a theory of instruction*, volume 59. Harvard University Press.

Brusilovsky, P. and Peylo, C. (2003). Adaptive and intelligent web-based educational systems. *International Journal of Artificial Intelligence in Education*, 13(2):159–172.

Brusilovsky, P., Schwarz, E., and Weber, G. (1996). Elm-art: An intelligent tutoring system on world wide web. In *Intelligent tutoring systems*, pages 261–269. Springer.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., and Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. *ACM SIGCSE Bulletin*, 40(1):256–260.

Holland, J., Mitrovic, A., and Martin, B. (2009). J-latte: a constraint-based tutor for java.

Huitt, W. (2003). Constructivism. *Educational psychology interactive*.

Johnson, W. L. and Soloway, E. (1985). Proust: Knowledge-based program understanding. *Software Engineering, IEEE Transactions on*, (3):267–275.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., and Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 107–111. ACM.

Kolb, D. A. et al. (1984). *Experiential learning: Experience as the source of learning and development*, volume 1. Prentice-Hall Englewood Cliffs, NJ.

Konak, A., Clark, T. K., and Nasereddin, M. (2014). Using kolb's experiential learning cycle to improve student learning in virtual computer laboratories. *Computers & Education*, 72:11–22.

Mitrovic, A. (2003). An intelligent sql tutor on the web. *International Journal of Artificial Intelligence in Education*, 13(2):173–197.

Peylo, C., Teiken, W., Rollinger, C.-R., and Gust, H. (2000). An ontology as domain model in a web-based educational system for prolog. In *FLAIRS Conference*, pages 55–59.

Reiser, B. J., Anderson, J. R., and Farrell, R. G. (1985). Dynamic student modelling in an intelligent tutor for lisp programming. In *IJCAI*, pages 8–14.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172.

Savery, J. R. (2006). Overview of problem-based learning: Definitions and distinctions. *Interdisciplinary Journal of Problem-based Learning*, 1(1):3.

Strauss, A. and Corbin, J. (1994). Grounded theory methodology. *Handbook of qualitative research*, pages 273–285.

Sykes, E. R. and Franek, F. (2003). A prototype for an intelligent tutoring system for students learning to program in java (tm). In *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education, June 30-July 2, 2003, Rhodes, Greece*, pages 78–83.

Vygotskiĭ, L. S., Cole, M., and John-Steiner, V. (1978). Mind in society.

APPENDIX

The following list presents the misconceptions identified in our work. Every misconception is given an identifier comprising the ID of the concept that corresponds to the Delphi CI plus a numeric value that distinguishes it from similar misconceptions associated with the same concept.

1. PA1-1: Understanding the difference between 'call by reference' and 'call by value' semantics.
2. PA1-2: Understanding the implications of masking an object reference within a function.
3. PA2-1: Understanding the difference between 'formal parameters' and 'actual parameters'.
4. PA3-1: Understanding the scope of parameters, correctly using parameters in procedure design.
5. PROC-1: Understanding the difference between definition and execution of function.
6. PROC-2: Understanding the role of the return value.
7. PROC-3: Understanding the usability of a function.
8. TYP-1: Understanding the difference between numeric values and their textual representation.
9. TYP-2: Understanding the difference between Boolean values and their textual representation.
10. TYP-3: Understanding the difference between null value and its numeric representation.
11. TYP-4: Understanding the difference between arrays of values and their possible numeric representation.
12. TYP-5: Understanding the implications of leaving variables uninitialised (undefined).
13. BOOL-1: Understanding the fact that a Boolean expression must yield a Boolean value.
14. COND-1: Understanding how to remove unnecessary conditions from multiple selection structures.
15. COND-2: Understanding the correct use of the else clause.
16. COND-3: Understanding the potential danger of testing different variables in multiple selection structures.
17. COND-4: Understanding the potential danger of not using break in switch structures.
18. COND-5: Understanding that repetition of code inside multiple selection blocks imply non-dependency.
19. SVS-1: Understanding the difference between variable values and literal values.
20. SVS-2: Understanding the necessity of variables/constants.
21. SVS-3: Understanding the difference between variable declaration and variable reference.

22. AS-1: Understanding the difference between assignment and equality operation.
23. SCO-1: Understanding the implications of not declaring a variable.
24. SCO-2: Understanding the difference between a global and a local variable.
25. SCO-3: Understanding that a (homonymous) local variable masks a global one.
26. SCO-4: Understanding the difference between block scope and function scope.
27. PVR-1: Understanding the difference between variables which hold data and variables which hold memory references.
28. IT2-1: Understanding that loop variables can be used in expressions that occur in the body of a loop.
29. IT3-1: Understanding the implications of having an empty body in a loop structure.
30. IT4-1: Understanding the semantics behind different loop structures.
31. IT5-1: Understanding that loop variables can help in loop termination.
32. REC-1: Understanding the implications of not using a base case in a recursive function.
33. REC-2: Understanding the implications of not using the return values from recursive calls within a function.
34. AR1-1: Understanding off-by-one errors when using arrays in loop structures.
35. AR2-1: Understanding the difference between a reference to an array and an element of an array.
36. AR3-1: Understanding the declaration of an array and correctly manipulating arrays.