Enhanced JavaScript Learning using Code Quality Tools and a Rule-based System in the FLIP Exploratory Learning Environment

Sokratis Karkalas, Sergio Gutierrez-Santos
Department of Computer Science and Information Systems
Birkbeck College, University of London
London, United Kingdom
sokratis@dcs.bbk.ac.uk, sergut@dcs.bbk.ac.uk

***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 the sub-system that is 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.**

Keywords- rule-based system, exploratory learning, personalisation

I. INTRODUCTION

FLIP is an Exploratory Learning Environment (ELE) used for teaching introductory programming to University students. This paper presents the architectural design of a sub-system in FLIP that provides personalised support to students based on their initial misconceptions. This system can also adapt to the students' particular circumstances based on past experience.

Computer programming is one of the major challenges in computing education [1, 2]. It is a composition-based task that imposes major problems to novices [1]. Students at that stage may suffer from a wide range of difficulties and deficits [3] which consequently can have a negative impact in their studies and their future career choices [3].

Programming is a craft and as such it can only be learnt by doing exercises in the lab. Teaching in the practical sessions requires a major effort from academic staff. It is obvious that there is an analogy between the effectiveness of the processes in the lab and the actual learning outcome that can be achieved. If students can utilise as much as possible of the resources available and 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 the architecture of a system that provides individualised teaching to students in the lab with no extra cost in terms of resources. The system is adaptive and provides personalised support depending on students' initial misconceptions and past experience.

II. RELATED WORK

Intelligent Tutoring Systems (ITS) started appearing in education in the late '70s. Typical examples of these systems are given in [5, 6, 26]. These systems target bugs in procedures. A system that resembles FLIP is PROUST [26] in the sense that it utilizes a knowledge base of programming plans along with the common misconceptions associated with them. Another system that teaches LISP is ELM-ART [27]. This is a rather different approach since the aim in this case is to provide an intelligent courseware delivery service. Other more recent attempts are presented in [28, 29, 30, 31]. The SQL tutor [28] is an intelligent but not adaptive web-based tutoring system that teaches SQL. It uses constraint-based modelling to represent domain knowledge and compares student code with correct solutions to known problems that have been specified by tutors. Another constraint-based system is J-LATTE [30]. This system teaches Java and provides multi-level support that includes both design and implementation. An expert system supported by decision trees that also teaches Java is [29]. A different approach in terms of knowledge representation is used by [31]. This is a web-based system that teaches Prolog utilizing an ontology. Systems like the E-Lab [19] are too narrow in scope since they focus solely on assessment.

The above systems provide assistance to well-defined Problem-Based Learning (PBL) scenarios [4] in a relatively controllable way. FLIP does not belong to this category. Help in FLIP is not provided in an intrusive way. Support is always available but is only given on demand. Students are given Inquiry-Based Learning (IBL) scenarios which by definition are open-ended ill-defined problems [4] and try to discover knowledge in an exploratory manner.

To the best of our knowledge there are no other systems that teach introductory programming in this manner. Systems that could be used as ELE like BlueJ [7], Greenfoot [8], Alice [9], Karel [10, 11], ToonTalk [12], LOGO-based Microworld [13], Scratch [14] and CodeSkulptor [20] are sophisticated Integrated Development Environments (IDE) that lack the intelligent support component. The problem is not to teach novices about language constructs and semantics but to provide a framework where certain compositions of these constructs make sense [15, 16]. Experts know much more than syntax and semantics [17, 18]. These stereotypical solutions to problems as well as the strategies for using them must be explicitly taught to the students [1] either by human or artificial tutors.

III. THE PROBLEM

Programming is a craft and as such it presupposes the development of technical skills. These skills can only be developed through practical training in computer

978-1-4799-4038-7/14 $31.00 © 2014 IEEE
DOI 10.1109/ICALT.2014.35

laboratories. Students are given IBL or PBL exercises and work under the supervision of a facilitator (tutor). They normally ask for help when they feel they cannot tackle some problem regarding the syntax or the logic of their code. If the students cannot receive the amount of help needed in a timely fashion they may not be able to overcome their problems and that can have a negative effect on their confidence and their studies in general.

Ideally the learning experience in the computer laboratory must be a sequence of successive iterations that follow Colb's learning cycle [21, 22]. Students attempt to solve the given task in cycles. In every round they try to develop something that moves them a step forward towards the completion of the given task. This process is often suspended when they hit the boundaries of the inner circle within their particular ZPD [23]. In such cases the tutors try to provide enough help so that the students can move on and resolve the issue. The intention is always to provide only enough and relevant help so that the student can overcome the problem and carry on with the process. During this intervention the facilitator helps the students to understand the issues raised and to develop an abstract conceptualization that can then be transformed to active experimentation in the next cycle [21, 22]. After each successful cycle a little bit of learning is achieved and gradually (possibly after many iterations) the ZPD [23] circles expand. In a busy classroom immediate and focused help cannot be guaranteed. The tutor – student ratio might be a limiting factor. Other factors that determine the effectiveness of this process might be the competency and the general (personal, social etc.) background of the tutor. Previous knowledge of the expected typical student misconceptions is not guaranteed.

The intention of this project is to offload the (human) tutors and delegate as much as possible of this work to virtual intelligent tutors. If the system knows what to expect in terms of misconceptions and knows how to adapt to the students' particular context, then immediate, focused and personalised support can be guaranteed. Interaction between students and teachers will be kept to a minimum and that will promote independent and constructive [24] learning. Tutors on the other hand, will be able to provide more valuable support at a higher level.

IV. THE LAYERED ARCHITECTURE OF THE STUDENT SUPPORT SYSTEM

JavaScript is an interpreted language. As such it does not provide messages that you would normally get from a compiler about syntax problems. You expect to receive such messages at run time. It is self-evident that the sooner a coding problem becomes known to the programmer the better. This way the problems will be more evenly distributed in time and the programmer will not have to face all of them at the end of the development process. The development process supported by FLIP is implemented in four layers.

A. Layer 1 (L1)

The first layer is implemented within the code editor. The component used by FLIP for this purpose is the Ace editor (http://ace.c9.io) which natively supports syntax checking based on JSHint (http://www.jshint.com). JSHint flags suspicious usage of JavaScript code. Syntax checking takes place dynamically as the programmer types in the code. These warnings appear along the left side of the editor and can be ignored by the programmer as their existence does not prevent the system from moving on to the next step.

B. Layer 2 (L2)

The second layer is implemented by a separate component that is part of the FLIP platform and is based on JSLint. JSLint (http://www.jslint.com) is a code quality control tool for JavaScript programs. It functions like JSHint but its API is not hidden within the editor. This component is directly accessible and therefore more configurable. It is used to capture cases that, although valid according to the language's syntax, may not represent good programming practices. If there are suspicious patterns, then the tool reports back to the user offering possible automatic changes in the code (refactoring) and/or visualisations that help them develop a better understanding of the issue. Issues that fall into this category could be undeclared variables, use of plain equality instead of strict equality, lack of curly braces for blocks of statements etc. This layer corresponds to a pre-processing phase that takes place before execution. Problems identified at that stage can be ignored by the programmer as they will not prevent execution.

C. Layer 3 (L3)

The third layer is again implemented as a separate component in the FLIP platform. This component uses the Esprima (http://esprima.org) parser to produce the abstract syntax tree (AST) of the code and then performs static analysis on it. The program is effectively transformed into a vector of characteristics. The vector is then given as input to a rule-based reasoning system that identifies patterns in the code likely to indicate potential student misconceptions. These misconceptions correspond to the Concept Inventory (CI) presented in [25]. If such misconceptions are detected the component starts interacting with the student. During this process it makes decisions on how to respond based on the type of problem and previous experience. The help can take various forms like a few tips and references to the language documentation, code refactoring, correctness validation with test wrapper functions and code tracing visualisations. The intention in this case is to provide the students with individualised help to the greatest possible extent so that they can safely diagnose the problems, understand their misconceptions and consequently embed the new concepts into their knowledge structures. This component is expected to respond to known student misconceptions / problems that are not necessarily related to bad coding practices and therefore may not be detectable by code-quality control tools. Examples of such misconceptions are off-by-one errors when using arrays in iterative loops, the notion of the type of variables, comparisons between values of different types, unnecessary code repetition etc.

D. Layer 4 (L4)

The fourth layer is a logical extension of the previous one but it is implemented as a distinct component. The difference is that it requires the prior selection of a given activity (task-dependent support). The code in this case is checked against the requirements of the activity by a decision tree classification system. This system is able to provide information regarding similarity of the given algorithm in relation to a number of existing algorithms in the learning set (database). The code is again converted into a vector of characteristics. Then the vector is checked against the decision tree and the system classifies the corresponding algorithm as being (or not) the one expected. If the system identifies missing components a number of questions/suggestions are presented to enable the programmer to isolate the problems indicated with the help of code visualisation tools and discover the solution in an exploratory manner. If the algorithm does comply with the requirements, the system applies dynamic analysis to verify its correctness. The whole process is adaptive and fully automated in the sense that it is able to take under consideration all the algorithms added to the database regardless of the time of insertion.

V. AN OVERVIEW OF THE STUDENT SUPPORT SYSTEM

The focus of this paper is the design of a sub-system that corresponds to layer 3 of the system presented in the previous section. The purpose of this sub-system is to mimic the support expected to be provided by a human tutor in problems that do not correspond to specific given tasks. During development, the student encounters issues that hinder the problem-solving process. These issues typically correspond to misconceptions related to either the correct interpretation and use of language constructs or the lack of algorithmic thinking skills. In both cases, the assumption is that the produced code is correct in terms of syntax and the suspicious pattern may not be captured by code quality tools.

FLIP utilises a buggy ruler for this purpose. From a conceptual point of view, the reasoner exists in a world that comprises (false) concepts that represent known misconceptions and students that deposit their understanding through code and direct interaction with the reasoner.

The concepts exist in the reasoner in the form of rules. These rules have been statically inserted by Experts and in the present system no rule induction is possible. The student's understanding is dynamically inserted into the reasoner in the form of facts. If the student's understanding triggers the activation of a concept which subsequently fires, the system will start interacting with the user. An overview of the system is depicted on Figure 1. Teaching entails a few tips and some references to the language documentation or a direct coding suggestion (using refactoring) or a test for correctness (using test-wrapper functions and dynamic analysis) or a presentation of some visualisation (using code tracing). The teaching session is an iterative process. Thus the result of these actions might provide the reasoner with new data and subsequently provoke further action. The process will carry on until either it is interrupted by the user or there is no indication of a misconception in the code that is under investigation.

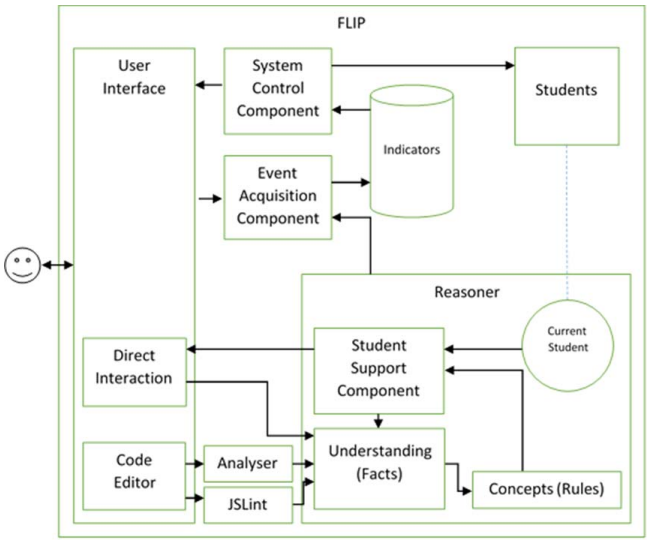

Figure 1. The FLIP Architecture

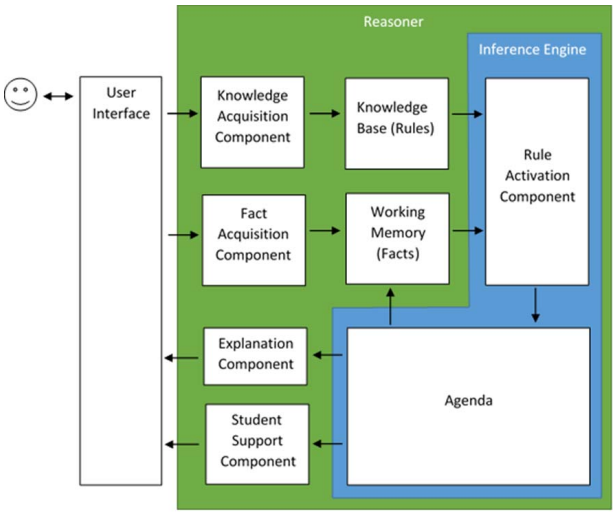

Figure 2. The Reasoner Architecture

As said above, the reasoner might respond to the user in different ways. The selected method depends on the level of difficulty the user experiences. This is detected using the information stored in the learner model. This is the second major entity in the conceptual view of this system. If the reasoner is invoked to deal with a particular misconception for the first time then the selected response provides the minimum possible help to the student. If the reasoner is asked to teach again an issue that involves the same misconception then the selected method will provide a little bit more help than the previous time. The state of the learner model is taken into account every time a decision needs to be made regarding the amount of help to be provided. This object gets continuously updated by FLIP in parallel with

any other processes that may take place at the same time so that it reflects as accurately as possible the current situation of the student. If the student has repeatedly used the reasoner for the same misconception at the highest level of difficulty then the system generates an event that provokes the intervention of a (human) tutor.

Events that are generated by the user interface or the reasoner are collected by the event acquisition component and accumulated in a database of indicators. This database is like a journal of historic data that stores all the events of interest. The system, upon arrival of new entries, sends a notification to the system control component which then depending on the type of indicator(s) received may update the user interface or the users' state.

VI. THE REASONER ARCHITECTURE

The reasoner is a rule-based expert system that interacts with the student when there is a misconception that needs 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. An abstract view of this system is given in figure 2.

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. The consequent part of the rules corresponds to the action that needs to take place in case they fire. The rule formation takes place in the Knowledge Acquisition Component (KAC) and then the resulting structures are stored permanently in the Knowledge Base (KB). Experts can utilise a visual component that is part of the User Interface (UI) to synthesise rules and instruct the KAC on how these rules should be constructed.

Facts are inserted dynamically into the system during the development process. The fact formation takes place in the Fact Acquisition Component (FAC). The insertion process is not direct. Users select a part of code in the editor and ask for help. The reasoner invokes FAC to generate the facts (see an example below). The facts are objects that are formed as a result of static code analysis. The selected code is parsed and analysed and the resulting constructs (if any) depict the code status. The patterns identified in the code are effectively transformed into a vector of characteristics (name/value pairs). This vector is submitted to the Working Memory (WM) as facts.

If there are facts that satisfy the conditional part of one or more rules, then these rules get selected by the Rule Activation Component (RAC) and placed in the Agenda (A).

If there are more than one rule in A then the system selects the one that has the lower number of references to characteristics and fires it. Firing the rule entails the execution of its consequent. This can be either some form of output to the user through the Student Support Component (SSC) or the creation of a new fact. In this case the fact is inserted into WM directly by A. The process carries on until there is no active rule to be processed or in other words until there is no misconception to be resolved.

As teaching takes place, the system provides feedback to the user in the form of help and/or questions. If the user is asked a question by the reasoner, then the answer may result in a direct formation of a fact through FAC.

Feedback is also given in the form of justifications as to how the decisions have been made by the reasoner. This service is provided by the Explanation Component (EC).

VII. A WORKED EXAMPLE

The student after a number of unsuccessful attempts to solve a problem issues a call for help. The selected code in the editor follows:

```
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 code is analysed by the FAC and the WM is populated by a series of facts. The RAC activates the following three rules:

1. Understanding the necessity of variables/constants (10 Facts).
2. Understanding off-by-one errors when using arrays in loop structures (21 Facts).
3. Understanding the difference between block scope and function scope (7 Facts).

Each rule corresponds to a potential student misconception [25]. One fact that satisfied the conditional part of the 3rd rule was generated by JSLint (Level 2). All the rest of the facts were generated by the analyser (Level 3). The reasoner decides to fire the rule that relates to the lowest number of facts (No 3). Simple problems have a priority over more difficult ones.

The reasoner decides on the amount of help to provide based on the current learner model [25]. If this needs to be taught for the first time a number of links are presented, directing the user to the part of the language reference that explains issues related to block/function scope. Then the user amends the code and asks for help again. If the change is successful then the misconception is considered resolved. If the misconception remains, then the system provides the option to refactor the code automatically. The user can accept the suggested change or ask for more help. The latter implies that the user is still skeptical about the correctness of the suggested change. The system executes a test function to verify the correctness of the code. The test might fail due to other problems that may exist in the code. If that happens the system suggests to keep a snapshot of this case for later reference, fix the problem and move on to the next one. The whole process is repeated for every rule in the Agenda. In

parallel the system updates the user database with the misconceptions found and the level of support provided. The ultimate level of support in the system is a code tracing visualisation. If help at that level has been given many times then the system suggests a human intervention. After a teaching session is finished, the system suggests a review of existing misconceptions by presenting snapshots of the user's code from previous attempts.

VIII. CONCLUSIONS

This paper has described the general architecture of the FLIP system, a system that integrates a combination of off-the-shelf and own components to provide intelligent support to early students of Javascript programming in the context of an open-ended exploratory programming session. The paper has focused especially on the reasoner that responds to students' misconceptions and provides support for them, including a detailed example showing a fragment of real code and how support is generated for the student.

Preliminary tests have shown that the system has achieved its original design objectives and it operates as described in the paper. An evaluation that will measure the responsiveness and scalability of the system is scheduled for the next edition of the JavaScript course in the summer term and thus falls out of the scope of this paper.

The system in its present state does not support automatic rule induction. We envision to make the system more versatile in that respect possibly by replacing the Knowledge Base component along with the inference engine with a Web Ontology Language (OWL) reasoner. Having a richer knowledge base without extra administrative overhead will facilitate a better and more focused student support.

REFERENCES

[1] Soloway, Elliot. "Learning to program= learning to construct mechanisms and explanations." *Communications of the ACM* 29, no. 9 (1986): 850-858.

[2] Jenkins, Tony. "On the difficulty of learning to program." In Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, vol. 4, pp. 53-58. 2002.

[3] Robins, Anthony, Janet Rountree, and Nathan Rountree. "Learning and teaching programming: A review and discussion." *Computer Science Education* 13, no. 2 (2003): 137-172.

[4] Savery, John R. "Overview of problem-based learning: Definitions and distinctions." *Interdisciplinary Journal of Problem-based Learning* 1, no. 1 (2006): 3.

[5] Brown, John Seely, and Richard R. Burton. "Diagnostic models for procedural bugs in basic mathematical skills." *Cognitive science* 2, no. 2 (1978): 155-192.

[6] Reiser, Brian J., John R. Anderson, and Robert G. Farrell. "Dynamic Student Modelling in an Intelligent Tutor for LISP Programming." In *IJCAI*, pp. 8-14. 1985.

[7] Kölling, Michael, Bruce Quig, Andrew Patterson, and John Rosenberg. "The BlueJ system and its pedagogy." *Computer Science Education* 13, no. 4 (2003): 249-268.

[8] Kölling, Michael. "The greenfoot programming environment." *ACM Transactions on Computing Education (TOCE)* 10, no. 4 (2010): 14.

[9] Dann, Wanda, Stephen Cooper, and Randy Pausch. "Making the connection: programming with animated small world." In *ACM SIGCSE Bulletin*, vol. 32, no. 3, pp. 41-44. ACM, 2000.

[10] Roberts, Jim, and Richard Pattis. Karel++: A gentle introduction to the art of object-oriented programming. Vol. 1. New York: Wiley, 1997.

[11] Becker, Byron Weber. "Teaching CS1 with karel the robot in Java." *ACM SIGCSE Bulletin* 33, no. 1 (2001): 50-54.

[12] Morgado, Leonel, and Ken Kahn. "Towards a specification of the ToonTalk language." *Journal of Visual Languages & Computing* 19, no. 5 (2008): 574-597.

[13] Jenkins, Craig, and Caerleon Campus. "Microworlds: Building Powerful Ideas in the Secondary School." *Online Submission* (2012).

[14] Maloney, John H., Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. "Programming by choice: urban youth learning programming with scratch." *ACM SIGCSE Bulletin* 40, no. 1 (2008): 367-371.

[15] Spohrer, James G., and Elliot Soloway. "Analyzing the high frequency bugs in novice programs." (1986): 230-251.

[16] Spohrer, James C., and Elliot Soloway. "Novice mistakes: Are the folk wisdoms correct?." *Communications of the ACM* 29, no. 7 (1986): 624-632.

[17] Adelson, Beth, and Elliot Soloway. "The Role of Domain Experience in Software Design." *Software Engineering, IEEE Transactions on* 11 (1985): 1351-1360.

[18] Brooks, Ruven. "Towards a theory of the comprehension of computer programs." *International journal of man-machine studies* 18, no. 6 (1983): 543-554.

[19] Delev, Tomche, and Dejan Gjorgjevikj. "E-Lab: Web Based System for Automatic Assessment of Programming Problems." (2012).

[20] Ben-Ari, Mordechai Moti. "MOOCs on introductory programming: a travelogue." *ACM Inroads* 4, no. 2 (2011): 58-61.

[21] Kolb, David A. Experiential learning: Experience as the source of learning and development. Vol. 1. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[22] Konak, Abdullah, Tricia K. Clark, and Mahdi Nasereddin. "Using Kolb's Experiential Learning Cycle to improve student learning in virtual computer laboratories." *Computers & Education* 72 (2014): 11-22.

[23] Vygotskiĭ, L. Lev Semenovich. Mind in society: The development of higher psychological processes. Harvard university press, 1978.

[24] Huitt, W. "Constructivism. educational psychology interactive." *Retrieved April* 2 (2003): 2008.

[25] Karkalas, Sokratis, and Sergio Gutierrez-Santos. "Intelligent Student Support in the FLIP Learning System based on Student Initial Misconceptions and Student Modelling" (under review)

[26] Johnson, W. Lewis, and Elliot Soloway. "PROUST: Knowledge-based program understanding." Software Engineering, IEEE Transactions on 3 (1985): 267-275.

[27] Brusilovsky, Peter, Elmar Schwarz, and Gerhard Weber. "ELM-ART: An intelligent tutoring system on World Wide Web." Intelligent tutoring systems. Springer Berlin Heidelberg, 1996.

[28] Mitrovic, Antonija. "An intelligent SQL tutor on the web." International Journal of Artificial Intelligence in Education 13.2 (2003): 173-197.

[29] Sykes, Edward R., and Franya Franek. "A Prototype for an Intelligent Tutoring System for Students Learning to Program in Java (TM)." Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education, June 30-July 2, 2003, Rhodes, Greece. 2003.

[30] Holland, Jay, Antonija Mitrovic, and Brent Martin. "J-LATTE: a Constraint-based Tutor for Java." (2009).

[31] Peylo, Christoph, et al. "An Ontology as Domain Model in a Web-Based Educational System for Prolog." FLAIRS Conference. 2000.