ICE: AN AUTOMATED TOOL FOR TEACHING ADVANCED C PROGRAMMING

Ruben Gonzalez
School of ICT, Griffith University
Southport, Queensland, Australia

ABSTRACT

There are many difficulties with learning and teaching programming that can be alleviated with the use of software tools. Most of these tools have focused on the teaching of introductory programming concepts where commonly code fragments or small user programs are run in a sandbox or virtual machine, often in the cloud. These do not permit user software to directly interact with system resources and accordingly do not directly support advanced programming concepts such as multiprocessing, inter-process communication (IPC), Device IO, concurrency, synchronization, and platform independence. This paper presents a new tool for teaching advanced programming called the Interactive C Environment (ICE) that is designed to support teaching of all these programming concepts by automating many tasks for both the learner and teacher. It consists of an integrated development environment (IDE) with a built-in editor, compiler, automated testing framework, and a built-in makefile processor. It supports programs written in C-99 and a subset of C++ using both MS Windows and Linux System APIs. ICE not only provides secure, automated testing and assessment of projects but also group learning support in terms of comparative statistics.

KEYWORDS

C-programming, Programming Based Learning, Automated Assessment

1. INTRODUCTION

The importance of teaching programming within the computer science discipline and the associated learning difficulties is well recognized (Thomas et.al, 2010; Robins et.al, 2003). A constructivist approach (Jonassen 2003) to teaching programming is commonly taken in the form of what some called “active” or experiential learning (Kolb 1984), where students learn by practising problem solving skills through writing software. Many tools have been developed to assist in both teaching and/or assessing programming skills with some success. Some tools such as Codingbat (Stanford 2011) focus on providing a simplified environment (IDE) for representing problems to students with immediate feedback. Other tools (Ahomiemi and Reinikainen, 2006; Ala-Mutka 2005, Whalley 2011) focus on automatic assessment of student work and a third class of tools, such as MOSS, JPLAG and various others, are specialized for plagiarism detection (Aiken 1994; Prechelt et.al, 2002; Pawelczak 2013; Lancaster and Culwin 2004). Further, some tools provide a combination of these capabilities. Many of the tools in the first and second category are focused on teaching introductory programming. Some tools such as Alice (Cooper 2001) and Scratch (Marji 2014) focus on teaching programming theory without the semantics of production languages through the use of graphical or visual programming metaphors (Kelleher 2005; Johnston 2004). Others focus on teaching the syntax and semantics of production languages as well as simple algorithms and the application of this knowledge towards solving simple computational problems (Pullan 2013). Commonly, in these cases the user programs are run in a sandbox environment or virtual machine (Pawelczak and Baumann 2104) where software exceptions are caught by the tools and handled in a safe manner thereby avoiding students having to deal with the unexpected consequences of poorly written code which might confuse a beginner. Since sandboxing isolates the user program from directly accessing the hardware and operating system resources, this approach inhibits the use of advanced programming concepts such as multiprocessing, concurrency, synchronization, inter-process communication, device IO and platform specific dependencies. Accordingly these tools are, in

general, not suited to teaching and learning in advanced programming courses such as systems programming and network programming or computationally efficient, low level programming.

To address the issue of supporting teaching and learning of more advanced programming the Interactive C Environment (ICE) is specifically designed for the teaching of systems and distributed programming. ICE provides automated mechanisms for self-assessment and feedback for programs that goes beyond just teaching language semantics using relatively simple "sandboxed" sequential programs that do not interact with the underlying computing platform. ICE is being used in the Systems Programming undergraduate course at Griffith University. The capabilities of ICE and its support for teaching and learning are presented in the following sections.

2. INTERACTIVE C ENVIRONMENT

ICE is a client-server system. The ICE server consists of a number of remotely hosted PHP scripts that are responsible for overall system and user administration including validating registered users, managing available exercises and test harnesses, providing the appropriate set of exercises for individual users, logging user activity including assessment and performance scores communicated to it by the client, performing plagiarism checking and returning statistical information of the performance of the user relative to other users. The ICE server communicates with ICE clients by tunnelling over the HTTP protocol using a cipher block chaining encryption scheme.

The ICE client consists of an integrated IDE with a built-in editor, compiler, an automated testing framework, and a built in makefile processor. Versions of the client exist for both MS Windows and Linux. It supports programs written in C-99 with inline assembly and a subset of C++. Irrespective of the host operating system the client supports user programs that target both MS Windows and Linux System APIs. The client also provides the user with performance feedback in the form of a visual representation of each user's standing in terms of comparative statistics with respect to the other members of particular courses.

In contrast to many online tools, the ICE client software executes compiled user programs natively on the local computer hardware granting them direct access to all the hardware and operating system resources of the underlying platform. ICE provides special support to test programs that make use of these enhanced capabilities. The ICE client encrypts and stores all the user code and test harnesses locally. Accordingly, it can run in either online or offline mode, with the limitation that while offline it cannot update the available exercises or user performance statistics. The ICE user interface consists of a window divided into three main panels: the exercise browser panel on the left, the editor panel on the right and the console panel beneath it.

2.1 ICE User Interface

In contrast to the many Integrated Development Environments (IDEs) that support hundreds of features and settings, which can be confusing to the novice user, the ICE user interface is designed to be as simple as possible. The program window (Figure 1) is divided into three main panels: the exercise browser panel on the left from which users select what problem to work on, the editor panel, which is the primary work area, and the console panel beneath it, which is used for interacting with users' programs. At the bottom left of the window is a prominent button labelled "Build" which activates the entire tool chain when clicked: it saves and compiles the source code, runs the applicable tests, reports the results to the ICE server and obtains updated statistical information about the user's performance.

The exercise browser panel consists of a drop down list of courses at the top. Below that is a collapsible tree widget, listing available exercise questions, followed by a prominent build/execute button and beneath that, a file browser subwindow showing a list of files in a project appears when required for multi-file projects. The tree widget also shows a visual depiction of the completion status of each exercise. An empty box indicates an exercise has not been attempted. A ticked white box indicates the exercise was satisfactorily completed. A red crossed box indicates the program failed static and syntactical tests. A white crossed box indicates the user program passed all of the static and syntax tests and compiled successfully but completely failed the functional tests. A white box with an 'o' indicates that the user's program passed the majority of functional tests but did not fully satisfy the task's requirements.

When the user first logs into ICE, it attempts to communicate with the ICE server. If successful, the course list box at the top of the exercise browser is populated with the courses that the user has access to. Selection of a given course from this list will populate the tree widget beneath it with the appropriate exercises grouped into (weekly) sets. If the ICE server cannot be contacted, the user only has access to the exercises, test harnesses and data that was accessed in previous online sessions. Also, ICE will store any user generated data that would normally be interactively sent to the server until the next time that ICE is in online mode when it will send the bundled stored data to the server.

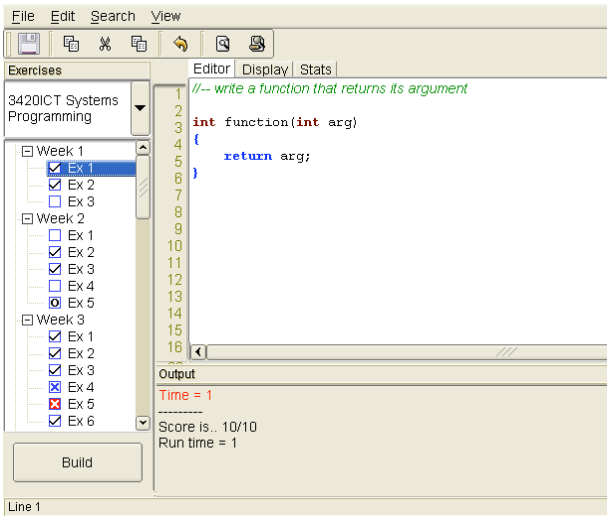

Figure 1. ICE Main Window for a Single File Project Showing User Program

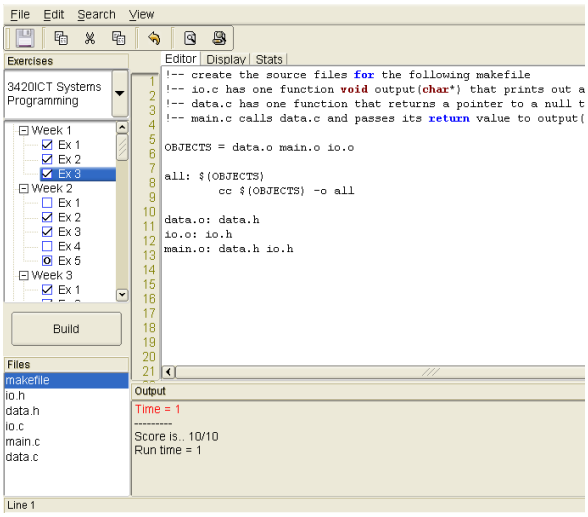

Figure 2. ICE Main Window with a Multi-File Project Showing Makefile Content and File Browser

Selecting an exercise from the exercise browser will cause the editor panel to be loaded with relevant boilerplate/template code at the first instance, and thereafter by the user's updated code. If the exercise is predefined to be a multi-file project, the editor panel is loaded with a relevant makefile template and the file list subwindow is made visible (Figure 2). Once the project dependencies have been defined in the makefile and the project is 'built', the nominated C source files are then created and displayed in the file list subwindow. Users can load any particular project file into the editor by selecting them in the subwindow, which automatically saves the currently loaded file.

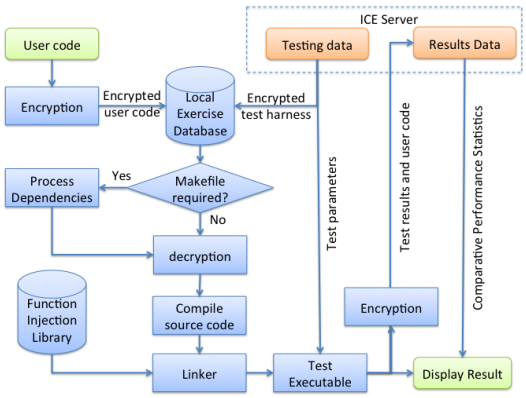

Figure 3. ICE Client Functions

When an exercise is built any compile or make errors will be reported in the console panel. Clicking on any error with the mouse will load the relevant source code file into the editor if it is not already loaded, highlight the relevant line and place the cursor on it. After successful compilation the user's program is automatically run in user mode (without test harness if it is not a program fragment) with the standard input

and standard output redirected to the console panel. If appropriate, the user can supply command line arguments for their programs using a dialog box. If a test harness is present, the program will be run again, possibly multiple times depending on the test conditions, under the control of the test harness. Each time a build is attempted telemetry data will be sent to the server consisting of the user program's encrypted source and object code and the test and execution time scores. These functions and processes implemented by the ICE client are depicted in Figure 3.

2.2 Single and Multi-file Programming

ICE supports three basic types of programming exercises. Those that consist of code fragments, complete single file programs and multi-file programs. Exercises that require code fragments to be written typically involve students creating or correcting code consisting of a single function, data structure or macro. These are always executed within predefined test harnesses that are invisible to the user. This class of exercise is typically used for students to focus on, and master, specific programming constructs and algorithms without needing to consider supporting functions that might be required to exercise those constructs or algorithms.

Single file exercises are small standalone programs with a defined main entry point. These are either executed independently, possibly taking command line arguments, or they are executed within a predefined test harness. This class of exercise is typically used for students to master a wide variety of more complex aspects of programming that are highly focused and require larger programs to be developed.

Multi-file exercises require the users to define the project's dependencies in a makefile but are otherwise handled in the same way as single file exercises. This permits students to master the concepts and use of traditional automated build systems and to develop much more comprehensive software programs. Additionally, multi-file programs permit the creation of multi-process programs by defining each required child process of the main process as a separate makefile target.

2.3 Automated Standard Testing

ICE supports a variety of automated white-box and black-box testing features. At the simplest level the client can impose syntax restrictions on the user's programs. These involve Boolean algebraic expressions containing specific program symbols that either must be or are not permitted to be utilized in the user program. The program will not be compiled unless these restrictions are followed. This permits certain software solutions to be enforced or prohibited to ensure that students gain experience in practicing specific skills to create alternate solution forms and encourage divergent thinking (Plucker 2010). Syntax errors and common semantic errors are detected and reported during the compilation phase.

At the next level, ICE can repeatedly apply black-box testing without the use of a test harness by evaluating the standard output of a program in the absence of, or in response to, a series of supplied command line input arguments. One problem with automated blackbox testing is that once users determine the specific output that will be tested for, they can create workarounds that disregard exercise requirements and instead simply produce the expected test output. Having a series of tests with different input arguments means that a program that is created to produce a fixed output will fail all but the one input condition it is designed to give the correct output for. At this level the testing regime is, by necessity, fairly simple: is the output correct or not for given input conditions? Accordingly, testing reports specific failures to the users such as "got 1,2,3 but expected 5,4,3" and scores the supplied solution to the exercise accordingly in response to whatever tests have been run on the user program.

The most comprehensive testing capability is provided through the use of test harnesses. These broaden the scope of what can be tested, beyond textual output in response to static input. In addition to permitting command-line arguments to be adjusted dynamically, they permit the dynamic injection of data into the standard input stream and any other IO streams used by running program. This permits a test harness to simulate user interaction with a program under test or manipulate the user program's interaction with the file-system. This feature allows testing to assess a user program's handling of human computer interaction issues and its interaction with dynamic file systems. Test harnesses also have access to all of a program's output streams such as standard output, files and display buffer and can analyze them for correct operation.

ICE test harnesses also allow white-box testing of a user program by permitting individual, pre-specified user functions to be inspected from inside and out. From the outside, pre-specified functions can be invoked

in isolation from the program within an environment created specifically for the purpose by the test harness to test their handling of boundary and random conditions. Inspection from within is performed by function injection: the remapping of symbols used by the user program to another defined within the test harness itself or injection library, e.g. intercepting calls to printf() and redirecting them to myprintf(). This permits the test harness to inspect the memory and file structures created by a program either before of after a pre-specified function call ensure that they are correct and assess any side effects the function might have.

One of the issues with automated testing is that the test harness often includes a reference solution that the user code is tested against for various input conditions. Without a reference solution, test harnesses are limited in the scope and range of testing that can be performed on user code since they can only partially model the expected program behaviour. However, the reference solutions must be kept hidden from the users to avoid them simply copying them. The approach taken by ICE is for the server to encrypt the test harness so that while it is being downloaded or stored by the client it can not be deciphered by users. A related issue is inhibiting users from sharing their own solutions in bulk with others or having them stolen. Encrypting the locally stored user programs using customized per-user keys as well as all client communication also solves this problem. While users may copy solutions to individual exercises from the editor and paste them somewhere else, ICE stores them securely so other users cannot decipher them. This permits users to copy their solutions from one computer to another for their own use, but not for the benefit of others.

Additionally, the ICE client evaluates the computational complexity of the user programs via measuring program run times. Since the programs are run in native mode on the user's individual machine, there is likely to be differences based on the hardware configuration of their machines. Accordingly the measured results are normalized by the runtime of a benchmarking function.

2.4 Advanced Testing Capabilities

Some of ICE's standard testing features can be found in other tools. Where ICE is differentiated is in its unique support for white-box testing of advanced programming concepts. These include handling of platform dependencies, multi-processing, inter-process communication and distributed systems programming.

ICE test harnesses can select platform specific dependencies to apply irrespective of the underlying operating system that ICE is running on. This permits the appropriate platform specific API to be exposed to the user program. Currently WIN32 and UNIX variant dependencies can be individually or jointly specified. If jointly specified, user programs must be portable, that is, they must compile and run on both WIN32 and UNIX variant operating systems. ICE will then compile and run the user program twice each time applying the respective platform dependencies. Where system APIs are incompatible, users are required to use conditional compilation via preprocessor directives. A significant subset of the system specific APIs for each operating system is currently supported including file system, multithreading, synchronization, device IO, IPC, timers and process management functions.

ICE provides support for user programs to perform process management including manipulation of environment variables. In regard to white-box testing of multi-process exercises, ICE test harnesses can act as either the parent or a child process in a solution to ensure that the corresponding process behaves correctly in relation to the other. In the case of process creation, the different process model used by MS Windows makes the Unix `fork()`, method very difficult to replicate. When running on MS Windows, the ICE implementation of fork, departs from its standard behaviour by starting the child from the main entry point, but then diverting execution at the fork call towards the child code. It inherits all of the resources of the parent process, but not the values of the automatic or global variables.

ICE further supports testing of multiprocessing exercises by intercepting all data on IPC channels. This permits test harnesses to inspect or modify the content of any communication between cooperating processes and control the behaviour of the processes involved by injecting or altering the data communicated. This includes the IPC mechanisms of anonymous Pipes, FIFOs (named pipes), files, standard IO, Message Queues, Sockets and Shared memory.

To facilitate network programming, ICE provides a fully configurable dispatch queue-based multithreaded TCP server that can be invoked by user programs or test harnesses. This TCP server permits the definition of a callback function for processing requests made to the server. This reduces the barriers for students to developing network software by avoiding the chicken and the egg problem of requiring either a TCP server or TCP client to be working in order to test the other. User exercises can focus on developing

socket client programs since ICE provides a reliable TCP server to test against. Test harnesses can also perform detailed white-box testing of TCP client user programs as test harnesses can configure the server callback function to handle and assess client requests, while simultaneously controlling the behaviour of the client via standard input injection and assessing its reactions to server responses.

3. TEACHING WITH ICE

ICE has been used in the teaching of advanced programming classes covering systems programming and distributed systems programming at Griffith University. The laboratory environment used in these classes includes both MSWindows and Cygwin/Linux operating systems. Specifically, ICE has been used to teach/assess the following set of concepts: language constructs for C/C++ (optional inline assembly); dynamic data structures; algorithms; network programming; file system and device IO; multiprocessing and inter-process communication and multithreading and synchronization. Students are permitted to do the practical lab work using the development tools of their choice.

ICE provides a means for students to thoroughly test their programs to ensure they correctly meet exercise requirements. Some students choose to initially implement their solutions using native tools such as gcc/make and MS Visual Studio and then use ICE's group learning features to check the performance of their solutions against the rest of the class. ICE is also used in assessed laboratories to automatically grade student work. ICE also supports plagiarism/similarity detection and provides comparative performance statistics.

Automatic plagiarism detection relies on evaluating some measure of similarity between student programs (Sheneamer 2016; Roy 2009; Đurić 2013). While a variety of different techniques are used the most common approach is to measure the similarity between sequences of tokens. These sequences are formed by removing all comments, whitespace and identifier names from software programs. The remaining code statements are replaced with tokens and the resulting sequences are directly compared. In ICE the Levenshtein Distance (Levenshtein 1966) is used to compare the sequences. In addition to comparing program source code, ICE also measures the similarity of the object code of the compiled files. These measures are reported to students so that they can see how original their solution to any given exercise is and to discourage them from simply cloning other students' solutions.

The code similarity measure is just one element of information that ICE provides to users regarding their performance. Each time an exercise is compiled, the test-harness-generated score and normalized program execution time is sent to, and collated by, the ICE server. The ICE server then responds by sending the ICE client updated statistics about the performance of the entire student group.

Users can access this information from the "Stats" tab of the main screen as is depicted in Fig. 4. The top half of the screen presents information about the individual student's performance for the selected exercise against the rest of the student group. The bottom half gives overall information about the performance of the individual student relative to the rest of the student group over all of the set exercises. The top row of bar and pie charts presents the student's and the groups' test harness scores for the selected exercise. The second row of bar charts presents user's code similarity measure or "uniqueness" and also the execution time relative to the rest of the student group. The third row of bar and pie charts presents the total number of successfully completed exercises for the individual compared to the rest of the group. The last row presents the total score achieved by the individual over all exercises attempted compared to the rest of the group. In each bar chart the horizontal line represents the score of the user whereas the bars are the sorted individual scores of the rest of the group. Permitting students to be immediately aware of their standing in a group of their peers encourages a sense of competitiveness in some and an awareness of a need for improvement in others. Both of these responses can work towards improving student learning outcomes.

The student group was surveyed to ascertain their perception of the usefulness of the ICE environment, the performance statistics data and other features. The results are given in table 1. A total of 75% of respondents found the comparative performance information useful in their learning with the remaining 25% being uncommitted. Students in general found a number of ICE features very useful, such as the simplified build environment/user interface. A majority of students found the ability of ICE to perform both WIN32 and UNIX builds and testing of their code rather than needing to use multiple tools on different operating systems useful. In general, on a scale of 1 to 5, where 1 indicated that ICE was not useful and 5 was that it was potentially very useful, ICE was awarded an average score of 4, with 50% of students awarding ICE a score

of 5. Students identified a number of teething issues and/or feature enhancements that they wanted. These include more a powerful text editor with infinite undo, more detailed reporting of compiler errors, and an ability to download their past exercise solutions from the server without having to access the copies that are automatically saved locally.

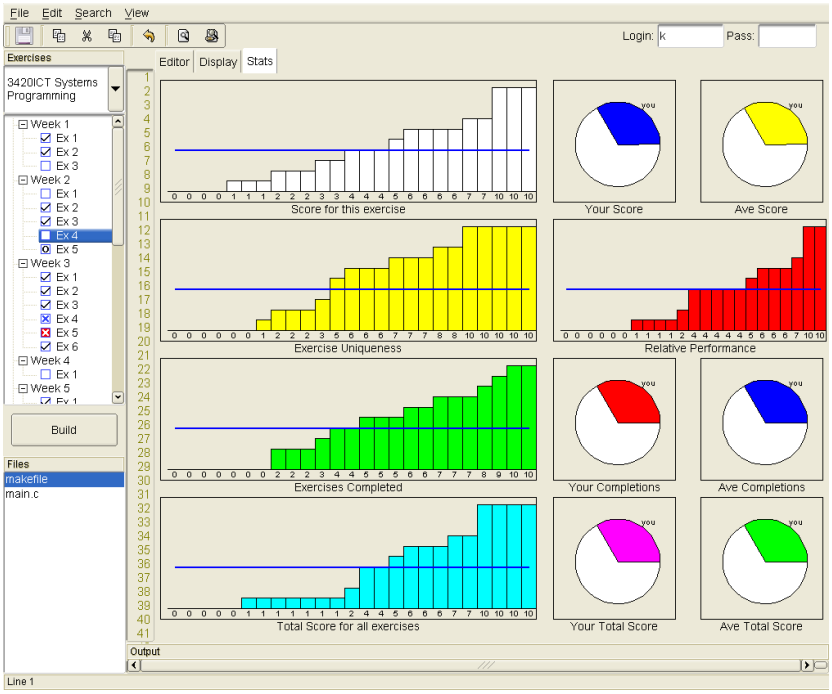

Figure 4. View of Statistical Data for an Individual Exercise

Table 1. ICE User Survey Responses

ICE Feature	Useful	Unsure	No Useful	Did not use
Performance Statistics	**75%**	25%		
Both Win32 & Unix	**62%**		12%	25%
Simple User Interface	**75%**		25%	
Automatic Testing	**50%**	38%	12%	

From a teaching perspective, ICE encourages students to complete their laboratory exercises as it gives students visibility as to their individual performance relative to the rest of the student group. From the collated student scores, it gives the teacher an instantaneous view into the performance of the entire student body and the performance of individual students within it. ICE discourages students from plagiarism and provides immediate feedback to students regarding the correctness of their solutions. It facilitates the grading of student work. It also simplifies issues related to needing to support multiple development environments for laboratory work. Most importantly it permits a class of advanced programming techniques to be assigned and assessed that is not possible with other software tools.

4. CONCLUSION

ICE is a specialized software tool for facilitating teaching and learning of advanced programming concepts. While it shares some of the objectives of other tools, it is unique in its capabilities for simplifying the testing of platform specific code with different dependencies by simulating each separate platform and for providing support for testing advanced programming features such as interprocess communication, device IO, multiprocessing and network programming, etc. In addition to functional testing, ICE also automatically

evaluates execution time and tests for plagiarism. The results of each of these tests are immediately available to the user for comparison to those obtained by others in the same learning group. The majority of students surveyed found ICE to be useful in their learning and staff likewise finds it useful for supporting teaching.

REFERENCES

Aiken A. 1994. MOSS: A System for Detecting Software Plagiarism. http://theory.stanford.edu/~aiken/moss/. {Online: accessed 12-January-2013}.

Ala-Mutka K. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. Computer Science Education vol.15, No.2, pp. 83-102.

Ahoniemi T, Reinikainen T, 2006, ALOHA - a grading tool for semi-automatic assessment of mass programming courses, Proc. 6th Baltic Sea conference on Computing education research: Koli Calling 2006, Feb. 01-01, Sweden,

Cooper S, Dann W, and Pausch R. (2000). Alice: a 3-D tool for introductory programming concepts. In Proceedings of the fifth annual CCSC northeastern conference on the journal of computing in small colleges (CCSC '00), John G. Meinke (Ed.). Consortium for Computing Sciences in Colleges, USA, pp.107-116.

Drew S, Pullan W, 2014. Technology immersion and Managed Learning for Student Programmers, presented at the Learning in Higher Education (LiHE'14) symposium, Greece, June 1-5,.

Đurić Z, Gašević D, 2013. A source code similarity system for plagiarism detection. *The Computer Journal*, Vol. 56, No. 1, pp.70–86.

Jonassen, D. H., 2003. *Learning to solve problems with technology: a constructivist perspective (2nd ed*.). Upper Saddle River, N.J.: Merrill.

Johnston, W.M.; et al, 2004. "Advances in dataflow programming languages". *ACM Computing Surveys*. Vol.36 , No.1, pp.1–34

Kelleher C, Pausch R, 2005, "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers", ACM Computing Surveys (CSUR), vol. 37, no. 2, pp. 83-137.

Kolb, D. A. 1984. Experiential learning: experience as the source of learning and development. Englewood Cliffs, N.J.: Prentice-Hall.

Lancaster T, Culwin F, (2004) A Comparison of Source Code Plagiarism Detection Engines, Computer Science Education Vol.14 , No. 2, pp.101-112

Levenshtein, Vladimir I. (February 1966). "Binary codes capable of correcting deletions, insertions, and reversals". *Soviet Physics Doklady*. Vol.10, No. 8, pp.707–710.

Marji, M (2014). Learn to Program with Scratch. San Francisco, California: No Starch Press. pp. xvii, 1–9, 13–15. .

Pawelczak D., Baumann A,. 2014. Virtual-C - a programming environment for teaching C in undergraduate programming courses. *In Proc. of Int'l Conf. Frontiers in Education: CS and CE*, pp.1142-1148

Pawelczak D., 2013 Online Detection of Source-code Plagiarism in Undergraduate Programming Courses. In Proc. of the 9th Int. Conf. on Frontiers in Education: Computer Science and Computer Engineering (Las Vegas, USA), pp.57-63

Plucker, J. A., & Makel, M. (2010). Assessment of creativity. In J. Kaufman & R. Sternberg (Eds.), The Cambridge handbook of creativity (pp. 48–73). New York, NY: Cambridge University Press

Prechelt, L., et al, 2002. JPlag: Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science, Vol.8, No.11, pp.1016–1038.

Pullan W. et.al, 2013. A Problem Based Approach to Teaching Programming. *In Proc. of the 9th Int. Conf. on Frontiers in Education: Computer Science and Computer Engineering* (Las Vegas, USA), FECS'13, pp.403-408

Robins A. et al, 2003, Learning and teaching programming: A review and discussion. *Computer Science Education*, Vol. 13, pp.137-172

Roy C, et al, 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming Journal*. Vol.74, No.7 (May 2009), pp.470-495.

Thomas, l. et al, 2010. Threshold Concepts in Computer Science: an Ongoing Empirical Investigation. In J. H. F. Meyer, et al (Eds.), *Threshold Concepts and Transformational Learning* (pp. 241-257). Rotterdam: Sense Publishers.

Sheneamer A., Kalita J. 2016 A Survey of Software Clone Detection Techniques. *International Journal of Computer Applications* vol. 137, No.10, pp.1-21, March 2016. Published by Foundation of Computer Science (FCS), NY, USA

Stanford University, 2011, CodingBat, from http://www.codingbat.com

Whalley J. L. and Philpott A. 2011, A unit testing approach to building novice programmers' skills and confidence. *In Proc. of the 13th Australasian Computing Education Conf*. (Perth, Australia), ACE 2011, pp.113-118