An Evaluation of the Constructive Teaching Methodology of Programming Concepts

Manas Hardas
Computer Science Department, Kent State University, Kent, OH.
mhardas@cs.kent.edu.

***Abstract*— In this paper we present the results from a class room setting based experiment to test the validity of the bottom up technique employed in teaching concepts in the domain of computer programming like branching, looping, nested looping, procedures, recursion, etc. Programming knowledge concepts are generally taught in an order of increasing complexity of comprehension, like branching before looping before recursion and so on. The null hypothesis is that with the knowledge of simpler concepts, students can then learn and apply increasingly complex concepts. We test this technique by asking students to review example LOGO programs and their corresponding correct outputs and then perform increasingly complex programming tasks which will require them to apply the acquired knowledge from the examples. It is expected that after reviewing complex programs the students too will be able to perform increasingly complex tasks thus demonstrating concept knowledge. The student performance on the tasks is measured by comparing the student generated program and the expert generated correct solution in terms of a parameter called cognitive complexity (CC). It is seen from the preliminary results that although students consistently learn and apply simpler concepts, they do not learn and apply complex concepts to solve tasks as often.**

Keywords-component; concept learning; LOGO; cognitive complexity;

I. INTRODUCTION

Jean Piaget [14] proposed the theory of constructivism as an exhaustive explanation for learning at different stages of a human life. The theory gives an explanation for cognitive development of concepts in human concept learning where in at every stage of learning, the cognitive concept map a human is assumed to possess is modified as new concepts is learned and prior concepts are discarded. According to the theory, a learner acquires new knowledge through the processes of assimilation and accommodation. Based on the previously acquired knowledge, a learner decides to put the new incoming concepts into context following one of the processes. New concepts at higher levels of abstractions are formed as knowledge is internalized from information. In assimilation the human cognitive concept map is not restructured, but in accommodation the map needs to be restructured to fit in the new concepts. Thus, according to the constructivist view of teaching students should learn higher level programming concepts from lower level concepts, which justifies the bottom up technique of teaching programming concepts. The general method of teaching programming concepts domain is a bottom up approach where the educators starts from the simplest concept and builds the knowledge by incrementally introducing more complex concepts. This teaching technique mimics the constructivist view of teaching.

An experiment is conducted in which the students are first presented with LOGO programs and the corresponding correct output for every program. Students are allowed to study the programs and form concepts i.e. attach semantics to the structural constructs of the program in their minds.

```
to circ :radius
repeat 360 [fd 2*3.14*:radius/360 rt 1]
end
circ 10
```

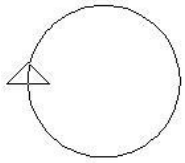

Figure 1. Example LOGO program to draw a circle using "repeat" statement

The students then take an untimed test where in they have to answer "constructive questions". Constructive questions are simply question such that they require the knowledge of prerequisite concepts to be able to be answered correctly and they progressively are based off the ability of answer the previously posed questions correctly. The programs generated by the students and the ideal solution programs for the problems are then evaluated for their cognitive complexity and compared. Cognitive complexity (CC) is chosen as a measurable parameter for two reasons namely, CC is fundamentally inherent to any software code or any piece of information for that matter and also because there has already been extensive research on the computation of CC using software metrics for simple software systems. Concept learning involves lot of other activities which go hand in hand with learning like comprehension, storing, encoding, decoding and retrieval of concept knowledge [3].

Parker and Becker [11] conduct a very similar experiment though they do not base their study on a conceptual knowledge representation. In another work, Soloway [4] details how construction of mechanism and explanations occurs in minds in the form of detailed plans as a result of programming assignments. Although the work lacks empirical measures it gives interesting explanations of how programmers selectively chunk information in an incremental manner to form explanations and mechanism. It too lacks a conceptual view of knowledge; however it still reiterates the existence of constructivism in learning programming concepts.

II. GRAPHICAL REPRESENTATION OF CONCEPT KNOWLEDGE

Cognitive concept maps are representations of concept knowledge in the human mind. We give a simple graphical approach for representation of cognitive maps based on constructivist theory of learning.

The concept map is a directed cyclic graph in which a node represents a concept and the edges represent the relationship between the concepts. The relationships can have various semantics associated with them however all of them are abstracted to a higher level of relation called *has-prerequisite* relationship. The children nodes of a concept node represent the prerequisites for that concept node. It means that the knowledge required to comprehend a concept is given by the immediate children of the concept node thus emulating the importance of prior knowledge in comprehension of a node. In essence, to comprehend a particular node it is essential to comprehend all its children nodes first, and to comprehend the children nodes, we need to comprehend their children nodes and so on. For example in programming concepts concept map shown in Figure 2, to comprehend the concept of "looping", student first needs to comprehend all its prerequisite concepts i.e. while and repeat. Given all the requisite knowledge required to comprehend a particular concept the students should voluntarily engage in an active discovery of knowledge and construction of concepts at a higher level of abstraction. The objective of any teaching technique is to impart this graphical representation to a student. A student is said to have "learned" the programming concept knowledge domain when a student has acquired, for example, a concept map like Figure 1.

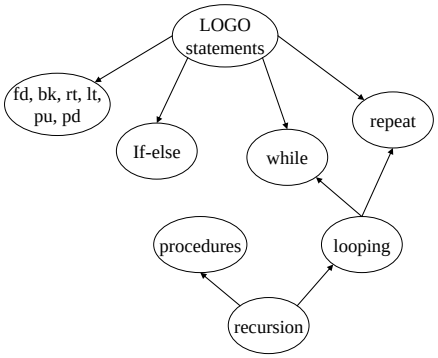

Figure 2. Example concept map for programming concepts

III. EXPERIMENT METHODOLOGY AND SETUP

Students are given LOGO programs along with their correct outputs. LOGO is an educational programming language developed by a group led by Seymour Papert at MIT in the 1950's and is based on the principle of constructivism. The most popular component of LOGO is the turtle graphics component which allows the users to draw simple or very complex shapes by commanding the turtle to move in a desired way. Writing programs to first draw simplistic shapes and then progressively modify the program to draw more and more complex shapes exhibits typical constructivist bottom up concept learning behavior. Another reason that LOGO was chosen was that it is extremely simple to learn in a very short span of time. In fact no syntactical knowledge or knowledge of any programming paradigm is required to play with turtle graphics. Any person with reasonable knowledge of very simple concepts likes programs, loops, branches etc. can look at a LOGO program and should be able to figure out the working of a program.

There have been a number of studies on the effectiveness of LOGO in education. It is debated whether programming is mostly a syntactical learning issue and should not be considered more than that, or if programming had a lot of implicit advantages in developing the cognitive thinking abilities of students [8, 12]. Papert conducted an extensive research project on the effectiveness of LOGO, named as the Brookline LOGO project. However it turned out that LOGO was not a factor in development of cognitive skills in students. Kurland et al. [8] in a study examined if LOGO aided in the development of general thinking skills other than programming and found the observations to the negative. R. Pea, Kurland et al. [7] also studied the cognitive effects of learning computer programming and mental models of children while learning recursive LOGO programming. All of these studies were extensive both in period of time and observations, which harmed the original intent of observing concept learning and skills acquisition.

In this study we attempt to measure the how effective is the technique of teaching concepts in a constructivist theory fashion by the use of LOGO programs. Students are first asked to fill out a pre-test questionnaire consisting of questions about their programming experience, programming knowledge, number of projects undertaken, etc. This information is later used to calculate a parameter of the student behavior as governed by his/her experience, prior knowledge etc. Then the students are asked a set of questions, first based on simple programming concepts like sequential statements, branching (if-then-else) and looping (repeat). Progressively the questions get complex, in the sense that they progressively are based on concepts at a higher level of abstraction in the 'programming concepts' concept map like in Figure 2. Gradually the questions are based on procedures, nested looping and recursion. The intuition behind this is that after having seen a few example programs demonstrating the concepts in practice, the students assimilate the knowledge of these basic concepts and then constructively use these concepts to internalize newer concepts at a higher level of abstraction. Parker and Becker [10] list some interesting properties which any constructivist question set should have and we believe our set of questions does follow these properties.

IV. COGNITIVE COMPLEXITY

The cognitive structures in the mind are not directly visible so we need a theory to use the observable behavior as one of the parameters to evaluate a student's performance [15]. Here the observable behaviors are the programs produced by the student which are a direct result of the student's acquired concept knowledge. We parameterize the code produced by the student in an attempt to give solution to the question, by the cognitive complexity of the code. The cognitive complexity of software is defined as "*those characteristics of the software which affect the level of resources used by a human/system to*

perform a given task on it" [2]. In more general terms, Basili [1980] defined cognitive complexity of software as the resources expended by a system, human or other, while interacting with a piece of software to perform a given task [1]. The resources referred to in these definitions are the concepts which the student has acquired from learning the previous concepts. Since the student obviously can comprehend a program written by him/her, it follows that the cognitive complexity measure of the final product program is a good indicator of his/her level of comprehension and consequently learning.

A. Cognitive processes in program comprehension

Chunking and tracing [17] are the two main processes which are generally believed to occur during the comprehension of programs. Chunking is the process in which a student recognizes a group of statements (not necessarily in order) and extracts information from these statements to form abstraction about the software. The process of navigating through the code to look for chunks is called as tracing. The processes of chunking and tracing are essential elements of the physical comprehension process and therefore directly contribute to the cognitive complexity. Therefore, we device an approach to calculate the total cognitive complexity based on the chunking complexity and tracing complexity from previously published approaches.

There are generally two views of comprehension by chunking. According to Sheiderman and Mayer learner utilizes both semantic knowledge and syntactic knowledge [17]. By chunking syntactic knowledge common concepts are grouped to a single concept at a higher level of abstraction forming a multi-leveled representation of the program. Semantic knowledge is nothing but conceptual representation of program structure like simple loops to more complex structures like whole algorithms. This semantic knowledge is relatively independent of the syntactical knowledge of programming languages. This view of comprehension fits very well with our conceptual representation and constructivist learning theory. Another view is that of Ehrlich and Soloway [5], which says that learner's form control flow plans and variable plans by chunking pieces of code based upon appropriate roles. We believe this again reverts back to the semantics of conceptual representation wherein roles represent concepts.

B. Factors affecting chunking

Cant et.al. [2] list a number of factors affecting the complexity of chunks. We classify these factors according to Rauterberg's generic definition of cognitive complexity of a system in terms of behavioral complexity (BC), system complexity (SC) and task complexity (TC). Cognitive complexity (CC) is then defined as CC=SC+TC-BC. The factors affecting chunk complexity classified accordingly are,

- Task Complexity
  - F1- Chunk size
  - F2 - Type of Basic Control Structure [19] in which the chunk is enclosed.
- System Complexity
  - F3 - Recognizability e.g. programming paradigm, rules of discourse, cohesion.
  - F4 - Visual Structure layout (grouping of chunks etc.).
- Behavior Complexity
  - F5 - Familiarity (experience, speed of recall etc.).

Thus according to Rauterberg's CC formula [15],

$$CC = F1 + F2 + F3 + F4 - F5 \quad (1)$$

C. Calculation of Factors

The first factor F1, chunk size is calculated as the standardized chunk size in lines of code. The programs used for the purpose of this experiment were very small and therefore we decided to use a normalizing factor of 100.

$$standardized\ chunk\ size = \frac{LOC}{100} \quad (2)$$

TABLE I. WCBCS WEIGHTS

Category	BCS	Weight WCBCS
Sequence	Sequence	1
Branch	If-then-else	2
	Case	3
Iteration	For-do	3
	Repeat-until	3
	While-do	3
Embedded	Function call	2
	Recursion	3
	Nesting	3
Concurrency	Parallel	4
	Interrupt	4

The calculation of F2 is a bit complex and equally important factor in the calculation of cognitive complexity. The complexity of the code enclosing the chunk is important because the structure will represent some acquired concept in the cognitive map after student has internalized the information obtained from the chunk. Therefore we calculate the information complexity (IC) of chunk enclosing code in this factor. Wang has identified 7 types of structures called as the Basic Control Structures (BCS) which can enclose the chunk and can therefore contribute to this factor [18, 19]. These BCS's are weighted according to their intuitive complexity in comprehension and empirical estimation. As it turns out,

concepts at a higher level of abstraction in the programming tasks concept map have a higher weight value (WCBCS). The weights for types of BCS are given in Table 1.

In our experiments, the maximum weight of the BCS we used was 3. To calculate the information complexity (IC) we use Khuswaha and Misra's [9, 10] definition given by,

$$IC = \frac{Information\,content\ of\ each\ LOC}{total\ LOC} * WCBCS \quad (3)$$

Information content in each LOC is calculated as the summation of all the operators, operands and identifiers in that line of code.

$$Info.(LOC)_k = operators_k + operands_k + identifier\,s_k \quad (4)$$

For our LOGO environment, we have defined as the LOGO reserved words or commands as the operators, any variables, words or lists used as operands and any other words like procedure names as identifiers examples as shown in Table 2.

TABLE II. ELEMENTS OF INFORMATION IN LOC'S.

Operators	Operands	Identifiers
FD, BK, LT, RT, REPEAT, TO, END, HOME, CS, PU, PD, etc.	:count, :size, :sidea, :sideb etc.	circle, square, spiral, sqrspiral etc.

The formula to measure WCBCS is given by amplify the effect of weights of the BCS's.

$$WCBCS = BCS_1^2 + BCS_2^2 + ... + BCS_n^2 \quad (5)$$

Other complexity calculation parameters which are often used are Halstead's software metrics, McCabe's cyclomatic complexity, Klemola's KLCID complexity metric, etc.

From the behavior data we collected, all except 1 student from the test set had 0 knowledge of LOGO environment. Therefore we decided to assign a value of 1 for system complexity combined for factors 3 and 4. The value of behavior complexity factor F5 is calculated from the data obtained from the pre-test questionnaire. A total of 19 students took this test, 2 of whom were girls, 5 were non computer science majors. The students were either labeled as experts (>3 years of programming experience) or novices (<3 years). The students were asked to rate themselves in their knowledge of programming languages, experience in programming, experience in LOGO, number of projects etc. The values collected were then standardized for the group, i.e. the value for each student was divided by the highest value in that category and the behavior complexity was then calculated by simply averaging all these values.

Once all the values for all the factors were calculated the total cognitive complexity was calculated using the eq.1. These values were calculated for each solution for each student.

D. *Example IC calculation for a simple LOGO program*

```
to spiral :size
if :size > 50 [stop]
fd :size rt 15
spiral :size * 1.02
end
spiral 10
```

Figure 3. Example LOGO program to draw a spiral

Operators=7 (in 6 LOC)

Operands=4 (:size in 4 LOC)

Identifiers=3 (spiral in 3 LOC)

Total = 14

Total LOC = 6

BCS=(sequential, if, recursion, procedure call)

WCBCS=[1^2+2^2+3^2+2^2]=18

IC = 14/6*18 = 42

V. OBSERVATIONS AND RESULTS

Once we calculated the cognitive complexities for each solution for all the students we compare them to the cognitive complexity of the correct solution for each question. The correct program solution is generated by an expert in the field and does not contain erroneous or extraneous statements or any elements which can artificially inflate or deflate the values of cognitive complexity. We compare the behavior of the graphs for each of the 19 students, plotted from the CC values generated from the solutions for the 9 questions in the test, against the expert's graphs generated from the CC values generated from the ideal solution. Therefore the desired graph would be one which follows the ideal graph (or referred to as expert's graph). A lower CC for a particular solution implies, the students' solution did not contain BCS's with the weights as much as the expert's solution contained. This in turn implies that the student wasn't able to reproduce in practice the BCS with greater weights i.e. ones which are at a higher level of abstraction. This could have 2 possible explanations, the first one is that student did not understand the basic concepts itself on which to base the higher concepts on, or students did not understand higher level concepts in spite of comprehending lower level concepts thus demonstrating an absence of constructivist concept learning. If the former is true, then that observation can be validated by checking the corresponding

CC values for those lower level solutions. The plots for the students against the expert's graph can be seen in Figure 4.

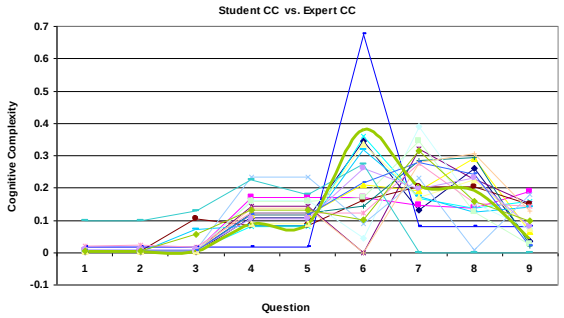

Figure 4. Student CC vs. Expert CC

It can be seen from the graph that most of the student CC graphs follow the expert CC graphs in behavior although not in values. It can also be seen that till question 5 most of the student CC graphs are equal to or above that of expert CC graph implying the comprehension of lower level concepts asked in the questions at the beginning of the test. However for question 6 and further the behavior of the student CC graphs becomes erratic. Some graphs follow the experts graph while some don't, but the consistency of behavior is not maintained. Particularly for solution to question 6, the graphs are especially erratic. This maybe because question 6 required the student to demonstrate the knowledge of nested looping. Although quite a few students were not able to answer this question correctly there were a few students who demonstrated through their solutions the ability to internalize higher level concepts like nested looping from simple looping. Figure 5 shows the behavior of average and median of student CC graphs against the expert CC graph to get a more general view of the behaviors.

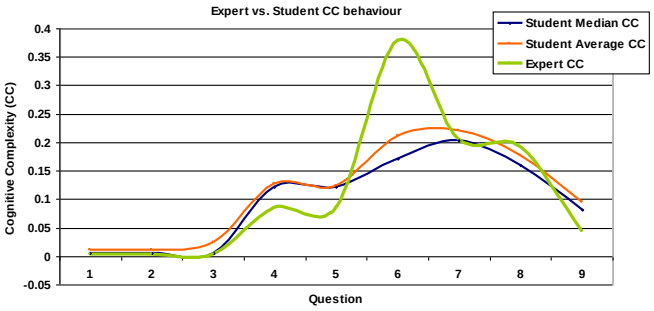

Figure 5. Student Average, Median CC vs. Expert CC

It can be seen from this graph that both the average and median CC graphs for students follow the same behavior in between themselves and the expert CC map too, except for question 6 which in fact was the nested looping question. In Figure 6 we plot the difference between the student CC values and expert CC values to highlight the students with CC graph behavior closest to the experts. In this way we can identify students who have near ideal graphs and compare their scores for the individual question to validate the results if necessary.

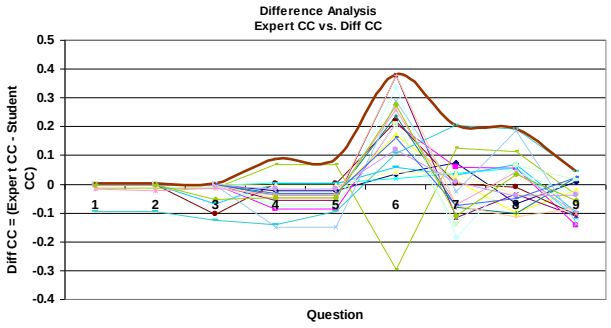

Figure 6. Difference analysis

It is seen that for question 1-3 there is almost little or no difference in student and expert CC graph behavior for most students. However after these questions, when higher level concepts start being questioned, the difference between the CC starts beginning to appear. Except for some outliers the difference remains uniform till question 6. After this however, the erratic behavior of the student graphs start. It will be very interesting to investigate the exact nature of causes for this happening.

In Figure 8 graph we first determine the correlation between the CC values for students and the CC values of the expert and then plot against the behavior complexity of students. Not surprisingly the two graphs follow very closely in behavior indicating the validity of this method of measuring comprehension abilities.

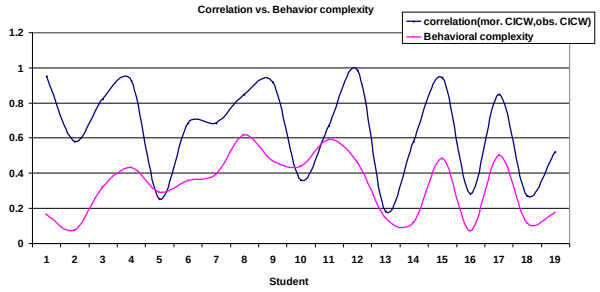

Figure 7. Correlation versus Behavior complexity

VI. CONCLUSION AND FUTURE WORK

In this paper we present a novel methodology to evaluate the bottom up technique for teaching programming concepts. The technique is based on theory of constructivism from educational psychology. Concepts are taught in an order of increasing complexity so that complex concepts can be learnt with the prior knowledge of simpler concepts. We test this technique of teaching in a classroom scenario where students are asked to answer specially designed questions and the solutions produced are compared with ideal solutions to make operational conclusions. On the basis of the parameter values we reason about the possible concepts which the student might have comprehended in the process of producing the solutions to the questions.

It is observed that although students learn and apply simpler concepts, they do not apply more complex concepts as often. When presented with a task which needs application of a complex concept, students in fact put to use their knowledge of simpler concepts rather than complex concepts. This implies that although the bottom up technique is employed by educators in teaching, students don't employ or are not able to employ the bottom up technique of constructing concepts in learning.

REFERENCES

[1] Basili, V.R.: Qualitative software complexity models: A summary. In: Tutorial on Models and Methods for Software Management and Engineering, IEEE Computer Society Press, Los Alamitos (1980).

[2] Cant S.N.; Jeffery D.R.; Henderson-Sellers B. "A conceptual model of cognitive complexity of elements of the programming process". Information and Software Technology, Volume 37, Number 7, 1995 , pp. 351-362(12).

[3] Downs, R.M. and Stea, D. *Cognitive Maps and Spatial Behavior: Process and products.* In Image and Environment. Downs, R.M. and Stea, D. (Eds.) Chicago: Aldine (1973:8-26)

[4] E. Soloway. *Learning to program = learning to construct mechanisms and explanations*. Communications of the ACM, 29(9):850--858, 1986.

[5] Ehrlich, K and Soloway, E., 'An empirical investigation of the tacit plan knowledge in programming' in Thomas, J C and Schneider, M L (eds) *Human factors in computer systems* Ablex Publishing (1984) pp 113-133.

[6] Javed Khan and Manas S. Hardas, "Hierarchical Course Knowledge Representation Using Course Ontologies", Proceedings of the 3rd Indian International Conference on Artificial Intelligence, IICAI 2007, Pune, India, December 17-19 2007, pp. 1684-1698.

[7] Kurland D. Midian, Pea Roy D., Children's Mental Models of Recursive Logo Programs. Technical Report No. 10. Bank Street College of Education, New York, NY, Center for Children and Technology, 13p.

[8] Kurland, D. Midian, Pea, Roy D., Clement, C., Mawby, R., A Study of the Development of Programming Ability and Thinking Skills in High School Students. Journal of Educational Computing Research, v2 n4 p429-58 1986.

[9] Kushwaha, D. S. and Misra, A. K. 2006. A modified cognitive information complexity measure of software. *SIGSOFT Softw. Eng. Notes* 31, 1 (Jan. 2006), 1-4. DOI= http://doi.acm.org/10.1145/1108768.1108776

[10] Kushwaha, D. S. and Misra, A. K. 2006. Improved cognitive information complexity measure: a metric that establishes program comprehension effort. *SIGSOFT Softw. Eng. Notes* 31, 5 (Sep. 2006), 1-7. DOI= http://doi.acm.org/10.1145/1163514.1163533

[11] Parker, J. R. and Becker, K. 2003. Measuring effectiveness of constructivist and behaviorist assignments in CS102. *SIGCSE Bull.* 35, 3 (Sep. 2003), 40-44. DOI= http://doi.acm.org/10.1145/961290.961526

[12] Pea, R. D. and Kurland, D. M. 1987. On the cognitive effects of learning computer programming. In *Mirrors of Minds: Patterns of Experience in Educational Computing*, R. D. Pea and K. Sheingold, Eds. Ablex Publishing Corp., Norwood, NJ, 147-177.

[13] Pea, R. D., Kurland, D. M., and Hawkins, J. 1987. Logo and development of thinking skills. In *Mirrors of Minds: Patterns of Experience in Educational Computing*, R. D. Pea and K. Sheingold, Eds. Ablex Publishing Corp., Norwood, NJ, 178-197.

[14] Piaget, J. (1937/1954). The construction of reality in the child. New York: Basic Books.

[15] Rauterberg, M.: *A Method of Quantitative Measurement of Cognitive Complexity*. In: Van der Veer, G., Tauber, M, e.a.: Human-Computer Interaction: Tasks and Organisation. CUD-Publ., Roma, 1992. pp. 295-307

[16] Roy D. Pea. Logo Programming and Problem Solving. [Technical Report No. 12.]. Bank Street Coll. of Education, New York, NY. Center for Children and Technology, 1983.

[17] Shneiderman, B and Mayer, R., 'Syntactic/semantic interactions in programmer behavior: a model and experimental results', *International Journal of Computer and Information Sciences*, Vol 8 No 3 (1979) pp 219-238.

[18] Wang, Y. (2002), On cognitive Informatics, Keynote Lecture, Proceedings of the 1st IEEE Conference on Cognitive Informatics (ICCI'02), Calgary, Canada, IEEE CS Press, August.

[19] Wang, Y., and Shao, J., Measurement of the Cognitive Functional Complexity of Software. IEEE International Conference on Cognitive Informatics, 18-20 Aug 2003, pp 67-74.