Using Formal Logic to Define the Grammar for Memory Transfer Language (MTL) on the mould of Register Transfer Language (RTL) and High Level Languages

Leonard J. Mselle

School of Informatics
The University of Dodoma
P.O.Box 490 Dodoma, Tanzania UR
mselel@yahoo.com

Keywords: POP-III.D. Visualization, Memory Transfer Language, Register Transfer Language, Formalization.

Abstract

This paper revisits visualization as a technique to enhance programming comprehension. It points out that animation, being a machine-driven visualization, is inadequate. Memory Transfer Language (MTL), as a visualization technique which is absolutely programmer-driven is demonstrated and discussed. It is shown that MTL can be plugged into current materials for teaching programming. Register Transfer Language (RTL) combined with high level languages are used as bedrocks on which MTL is formalized.

1. Introduction

Constructing and even understanding computer programs have proved to be a highly daunting task for most learners (Dehnadi 2006, Rajala et al. 2008). There are various techniques which have been suggested to aid learning and teaching programming. Visualization, generally defined as presenting the execution of program or algorithm with graphical components, is one of such techniques (Ben-Ari 2001, Naps et al. 2003).

Various researchers have reported on the positive impact of visualization in teaching programming to novices (Kuitinnen et al. 2008, Rajala et al. 2008, Ziegla and Crews 1999, Hundhausen and Brown 2007).

Program visualization is a research area that studies ways of visually assisting learners in understanding behavior of programs. Program visualization can be either dynamic or static. Dynamic visualization usually shows how the execution of programs progresses by highlighting parts of the code under execution and by visualizing changes in variable status. An example of a dynamic program visualization tool is Jeliot3 (Moreno et al. 2004) and MTL (Mselle, 2011c). Static visualization tools visualize program structures and relations between program objects. An example of a popular static program visualization tool is BlueJ (Kölling et al. 2003).

However, Hundhausen et al (2002) report that visualization is not widely popular among programming instructors. They confirm that one of the main reasons is because the teachers responsible for the courses refuse to use new methods in teaching. In addition, Hundhausen et al (2002) found that the sole use of visualization systems doesn't necessarily improve the learning results. They argue that it is more important to engage the learners in the subject using visualization system as an aid. Mselle (2010a, 2011a, 2011b) states that, the reason visualization is yet to be used in teaching programming is because visualization technique has yet to become an integral part of the way programming materials are presented. He contends that most programming books are written as programming-language manuals (without visual dose) and consequently programming syllabuses together with programming notes suffer from the same defect since they are mostly framed in the same fashion.

All popular visualization tools are machine-driven and or language specific. This property, apart from reinforcing the authority of the machine in learning to program, it denies the user the ability to visualize the program outside machine environment. A programmer-driven visualizer such as MTL has capabilities to enable the programmer to visualize code behavior outside machine environment. MTL is used to present programming materials for any high level language (Mselle 2011a, 2011b, 2011c). Any programmer-driven visualizer, will reinforce the sense of "I am working the machine" on the part of the programmer, hence strengthening the programmers' authority (Du Boulay 1986).

MTL, as new programmer-driven visualizer, is however, not yet formalized. By formalizing MTL, learners, instructors and writers of programming materials will be able to verify the correct application of this language.

2. Objective

The objective of this paper is to demonstrate the use of MTL as a program visualization tool which is capable of visually describing most of introductory programming aspects without relying on the machine. It is demonstrated that MTL is a visual language that can be embedded in the current teaching materials. Formal logic and the framework of Register Transfer Language (RTL) together with high level languages are employed as a mould on which MTL is formalized. In the end, it is pointed out that formalization of visualization tools (like MTL) could encourage the use of visualization techniques in teaching programming.

3. A brief revision of compilation process and the need for visual tools

Generally, compilation or interpretation of a program in high level language is carried out in the machine by a recognizer. The output of this process is the object code (if there are no errors) or an error report if there are syntactical errors. This process is depicted in Figure 1.

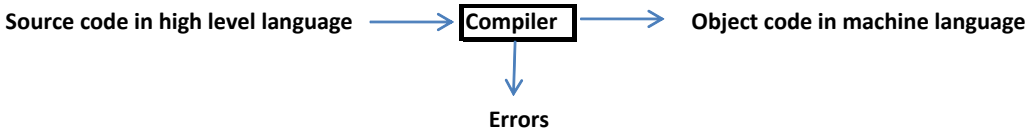

Figure 1 - Compilation/Interpretation process

Compilation framework assumes that the novice knows why the compiler is reporting the mistakes. However, this is hardly the case. Most novices can't reason as to why a compiler is rejecting their code statements. Novices are unsure of how to proceed even when the compiler has reported and indicated lines of errors. Knowledge and confidence of novice programmers in high level programming languages is extremely limited, and most of mistakes they commit are not accidental. Novices commit mistakes which represent their understanding of the syntax and semantics of a high level language in question. When a compiler rejects their notion, novices have no alternative but to call for help or give up or tinker (Du Boulay 1986, Samurcay 1986, Perkins et al. 1986). As a means to teach programming, visual tools have been reported to aid novices to trace and correct errors once these have been reported by the compiler (Rajala et al. 2008). Despite its momentum, visualization has not been adopted in mainstream teaching of programming (Naps et al. 2003).

4. Proposal for MTL as a visualization tool

Since visualization has shown promising results in enhancing comprehension, (Ben Ari 2001, Kuitinnen et al. 2008, Rajala et al. 2008, Mselle 2010a, 2011c) the situation now calls for effort to accelerate its use in teaching programming (Hundhausen and Brown 2007) .

In a survey performed by the ITiCSE 2002 Working Group on 'Improving the Educational Impact of Algorithm Visualization", the main reasons why educators do not use visualization materials in their lectures is reduced to two aspects: the *time* required to do so and the *lack of integration* with existing teaching materials (Naps et al. 2003).

Since that report, several approaches have addressed the *time* aspect, for example by providing tools or generators for quickly producing content that fits the educator's or learner's expectations, and allow the user to specify the input values (R¨oßling and Ackermann, 2006). However, the integration of visualization into the learning materials still needs to be addressed (R¨oßling et al. 2008).

Proposal for formalization of MTL is based on the assumptions that;

- there can be a language/device that a novice programmer could use to visually interpret the source code from any high level programming language, such that this interpretation is in fact a visual version of the machine semantics;
- this language/device can accept all statements of any high level programming language as its inputs and produce output exactly in machine semantics but in a form legible and visible to the novice programmer;
- this language/device could be used by the novice programmer to find and fix program bugs, and verify the program logic;
- such language/device can be integrated with the existing materials which the novice could use to understand programming, debug and avoid common misconceptions;
- such language/device could be used along with high level programming languages to present the current learning programming materials for novices in both soft and hard format.

Figure 2 is the schematic representation of such a language/device.

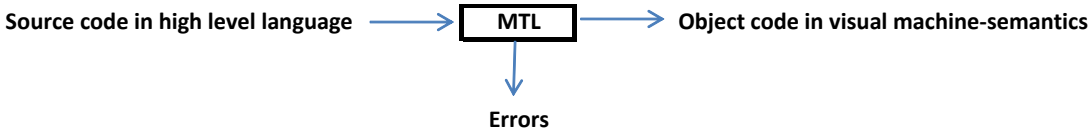

Figure 2- Program interpretation by MTL

MTL is proposed to be a programmer-driven language used to visually interpret the code. MTL is supposed to accept the syntax of high level programming language as its input. The output is a product which carries the visual semantics of the object code in a syntax that is understood by the programmer. The novice can rely on MTL to interpret each line of code to detect errors, just as the machine uses the compiler to interpret the code and generate error report. In addition, MTL is proposed to be useful for verifying the logic of the code. Formalizing MTL is intended to demonstrate its linguistic character and introduce a mechanism upon which this language can be embedded in teaching materials. In addition formalization will provide a framework for correct use of MTL and its systemic study for further application.

4.1 Demonstration of MTL as employed in algorithm visualization

Basic programming elements include; *variable declaration*, *data feeding*, *functions*, *arrays*, flow of control (*selection*, *sequence*, *loops* and *recursion*), *file handling* and *pointers* (Rajala et al. 2008). These are mostly the main issues which novice programmers are supposed to grasp. Mselle's works (2010b, 2011a, 2011b) have demonstrated that all these aspects can be visualized through MTL. The discussion in this paper to demonstrate the use of MTL shall be limited to; *variable declaration*, *data feeding*, *data operation*, *loops*, *functions*, *arrays* and *pointers*. Interpretation of codes by MTL shall be demonstrated

using five short programs (Program 1, Program 2, Program 3, Program 4 and Program 5) as demonstrated in figures 3-7.

Consider the code as represented by Program 1 in Figure 3. MTL is used to visualize variable declaration, data feeding (assignment) and data operation (addition).

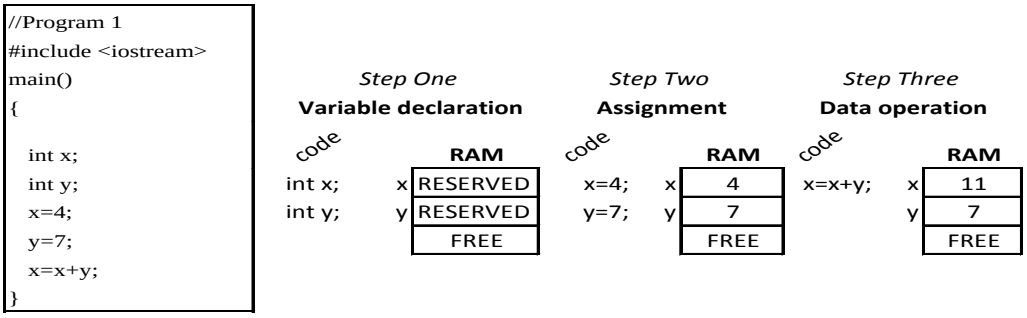

Figure3- Code interpretation (variable declaration, assignment and data operation) using MTL

Visualization of flow of control (*while loop*) by MTL is demonstrated in Figure 4 which is an interpretation of Program 2 (insert).

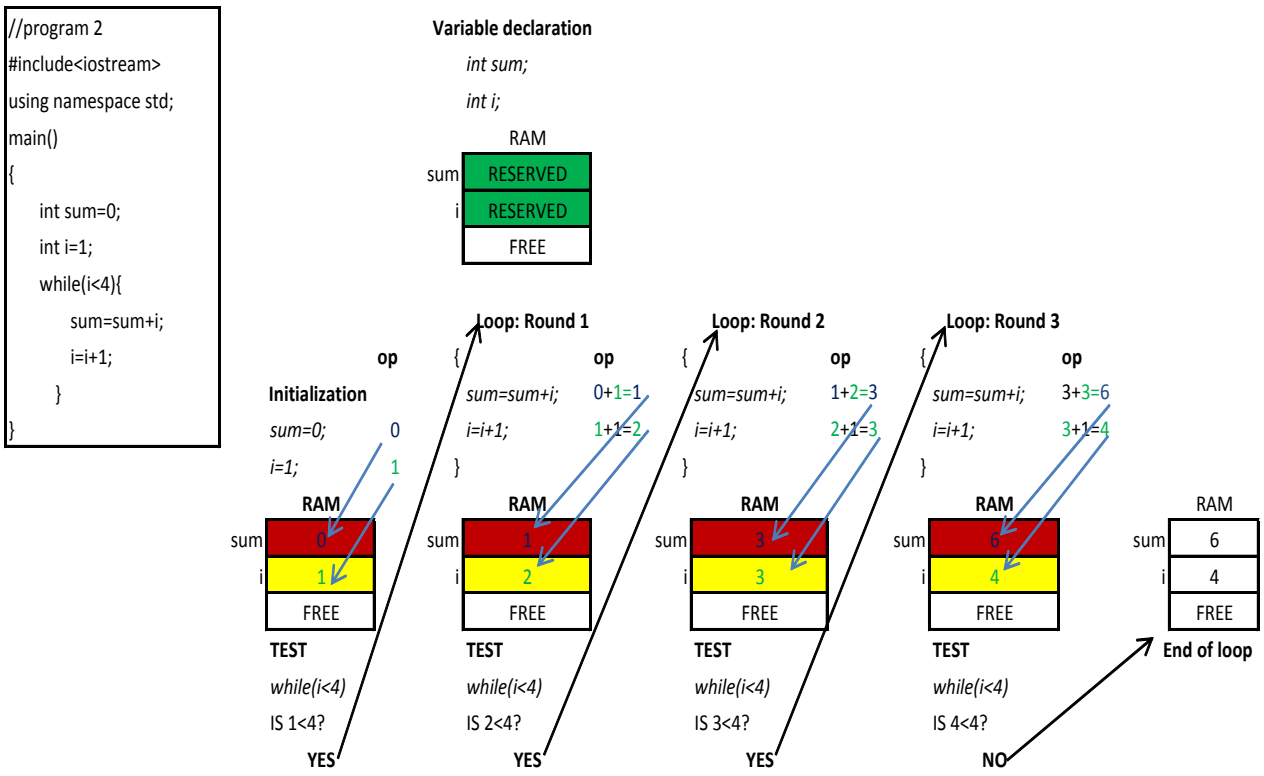

Figure4- Code interpretation (while loop) using MTL

Function declaration, function call, and parameter passing can be visualized as depicted in Figure 5, where Program 3 (insert) is interpreted using MTL.

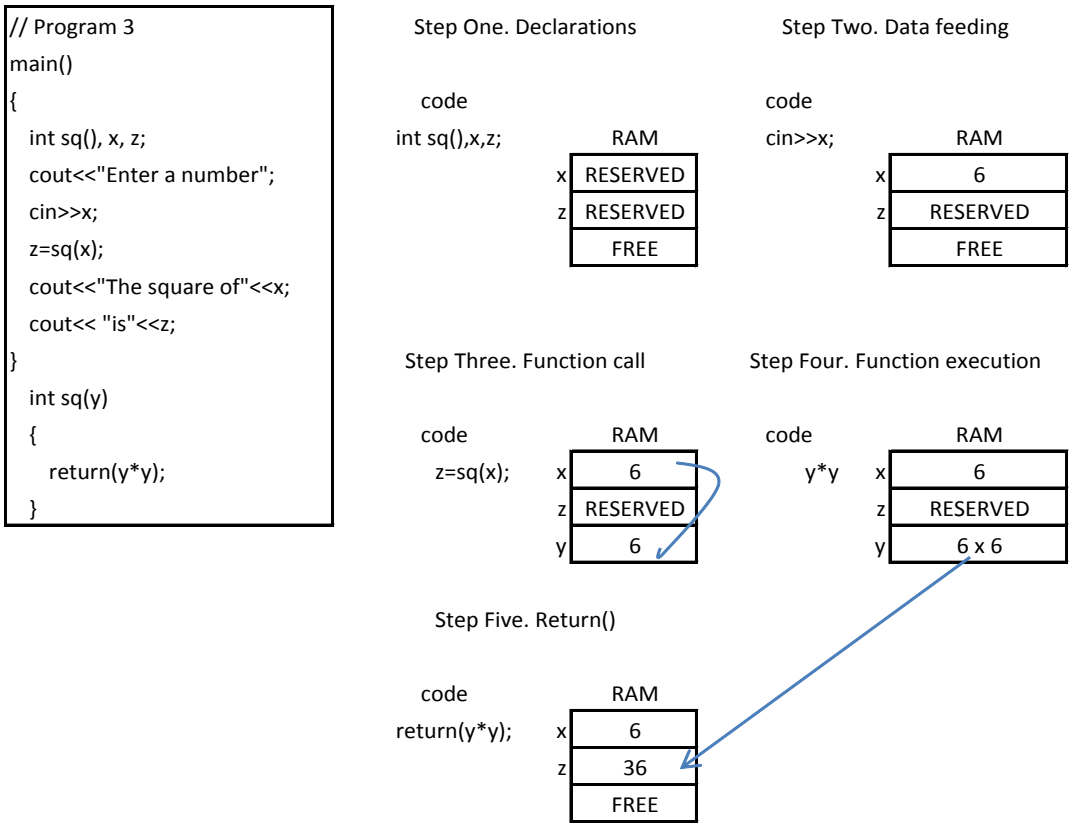

Figure 5- Code interpretation (functions) declaration, execution, call and return using MTL

In the same way as demonstrated in Figures 3, 4, and 5, MTL can be used to visualize the concept of arrays as demonstrated in Figure 6 in which Program 4 (insert) is visualized.

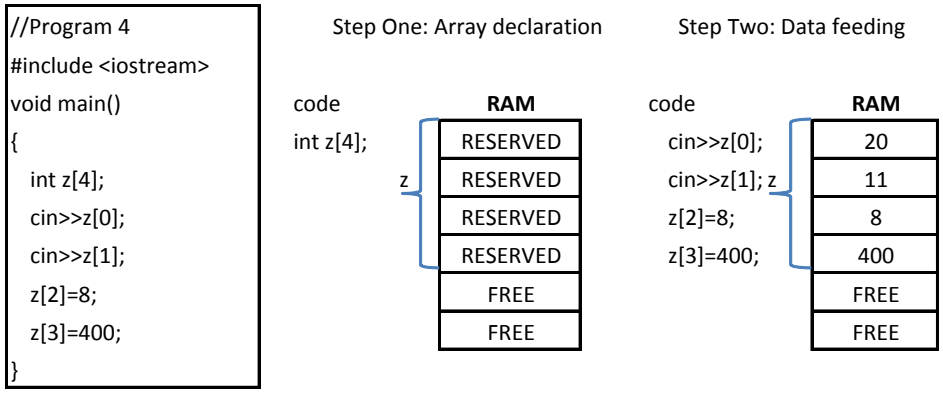

Figure 6- Code interpretation (array declaration and data inputting in an array) using MTL

Pointers, as argued by Dehnadi (2006), constitute the most difficult part in introductory programming. However, the concept of pointers as demonstrated in Figure 7 is simplified under similar MTL models.

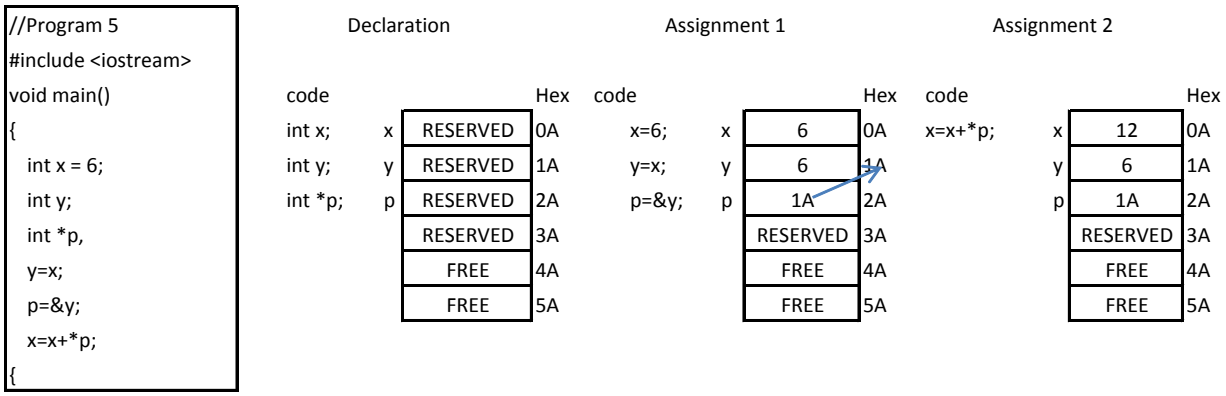

Figure 7- Code interpretation (pointers; concept, declaration, manipulation) using MTL

The illustrations provided by Figures 3-7, show the relationship between what takes place in the *RAM* and the effect of *high level language statement*. This is the foundation for MTL syntax and semantics.

Other basic programming issues such as *if* statement, *case* statement and *file handling* can equally be demonstrated using MTL, as evidenced in the work by (Mselle 2010b, 2011a, 2011b). Since these are the basic features of elementary programming, it is valid to confirm that MTL can be used to teach the entire curriculum pertaining to elementary programming in any high level language. MTL describes all basic programming issues using RAM status only. As demonstrated in this sub-section, MTL contextualizes programming by using RAM, represented by simple symbols (rectangles) which do not require any special knowledge to understand or employ. MTL consistently relay on one single aspect of computer (RAM) to interpret all basic programming aspects. This characteristic, as it will be demonstrated, gives room for MTL to be formalized as a language similar to RTL.

5. Using formal logic to define the MTL grammar

Chomsky (1957) states: "By a language we shall mean a set of sentences (finite or infinite), each of finite length, all constructed from a finite alphabet of symbols. Formalization of a language is a framework that is used to guide for correct use of such language (grammar). In order to formalize MTL, general characteristics of language as provided by (Chomsky 1957) are revisited.

5.1 Using formal logic to derive the MTL grammar

"By a grammar of the language *L* we mean a device of some sort that produces all of the strings that are sentences of *L* and only these" (Chomsky 1957).

Formally a grammar and its language are defined by the syntax and the semantics. Still relying on Chomsky (1957), the syntax of a grammar is defined thus:

- A finite set N of non-terminal symbols, none of which appear in strings formed from G (1)
- A finite set Σ of terminal symbols such that $\Sigma \cap N = \emptyset$ (2)
- A finite set *P* of *production rules*, each rule of the form
  $(\Sigma \cup N)^* N(\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$ (3)
  Meaning that, each production rule maps from one string of symbols to another, where the first string (the "head") contains an arbitrary number of symbols provided at least one of them is a non-terminal. In the case that the second string (the "body") consists solely of the empty string i.e., that it contains no symbols at all it is denoted with ε.
- A distinguished symbol *S*, called start symbol, such that $S \in N$ (4)
  It is concluded that a phrase grammar G is formally defined as a quadruple (Σ, N, S, P) (5)

Similarly, the semantics of a grammar is defined thus:

- Given a grammar G = (Σ, N, S, P), the binary relation $\Rightarrow_G$ on strings in (Σ ∪ N)* is defined by:
  $\alpha \Rightarrow G\beta$ iff $\exists\, u, v, p, q \in (\Sigma \cup N)^* : \alpha = upv \wedge p \rightarrow q \in P \wedge \beta = uqv$ (6)

5.2 Proposed formalism for MTL on the mould of RTL

MTL is proposed to be formalized on the mould of RTL. This proposal is based on the fact that;

- RTL is an established formalism. MTL is therefore proposed to bear all characteristics of RTL except that MTL is a device for interpreting high level languages whilst RTL is a device for interpreting statements of the machine language;
- RTL is a language used to describe information flow in computer memory (RAM). MTL is used to describe the information flow in RAM. With these similarities, it is possible to borrow some abstract and concrete terms from RTL and adopt them to MTL as it will be demonstrated in Table 1 and Table 2.

Any language grammar is made of abstract terms and concrete terms. According to Preparata (1985) the RTL abstract terms are as depicted in Table 1.

	RTL Term	**Example/Description**
1	*microsequence*	*1.IR<-PC 2.R1<-R2*
2	*concurrent step*	*2.R1<-R2*
3	*microstep*	*R1<-R2*
4	*assignment*	*R1<-R2*
5	*conditional assignment*	*if R1=R2 ++R2*
6	*go to*	*go to (machine)*
7	*register/RAM*	*R1*
8	*condition*	*R1>R2*
9	*op*	*R1+R2*
10	*function*	*++R1*
11	*label*	*1*
12	*constant*	*R2<-2*

Table 1- The RTL terms (source: Preparata 1985)

MTL is proposed to act as a device whose statements are transfers of information between memory to memory (Mselle 2011c). Concrete terms will be composed of RAM status as a function of execution of a statement in high level language. RAM status shall constitute sentences of MTL.

Since both RTL and MTL rely on RAM to describe flow of information, and since RAM and Registers are similar in their nature and function, modification of RTL productions (P_R) -which are based on registers and RAM- to evolve MTL productions (P_M) which are based on RAM will retain a similar logic. The entire RTL set of terms with their corresponding derivations is borrowed, with modifications to evolve a set of MTL terms and their corresponding derivations as reflected in Table 2.

	RTL Term	Example/Description	MTL Counterpart	Example (in C++)/Description
1	*microsequence*	*1.IR<-PC 2.R1<-R2*	*macrosequence*	*1. int x; 2. x=7*
2	*concurrent step*	*2.R1<-R2*	*concurrent step*	1. *int sum; (single statement with label)*
3	*microstep*	*R1<-R2*	*macrostep*	*int x; (single statement without label)*
4	*assignment*	*R1<-R2*	*assignment*	*int x; x=7;*
5	*conditional assignment*	*If R1=R2 ++R2*	*conditional assignment*	*if (x>y) ++y;*
6	*go to*	*go to (machine)*	*go to 1*	*go to 1;*
7	*register/RAM*	*R1*	*RAM status (variable)*	*concrete memory cell (see Figures 3-7)*
8	*condition*	*R1>R2*	*condition*	*an event whose occurrence can be tested i.e. x>y;*
9	*op*	*R1+R2*	*op*	*operation i.e. x+y;*
10	*function*	*++R1*	*function*	*operation on a single operand i.e. --x;*
11	*label*	*1*	*label*	*1. (an integer marking the position of an instruction)*
12	*constant*	*R2<-2*	*constant*	*#def c 300000*

Table 2- Modification of RTL terms and productions to produce MTL terms and productions

5.3 Combination of formal logic and RTL framework to establish formal definition for MTL

- From 1, set $MTL_N=\{N_M\}$, is declared to be the set of *all statements* making up the code (7)
- From 2, set $MTL_\Sigma=\{\Sigma_M\}$, is declared to be the set of *all RAM diagrams* corresponding to code statements;
- A *sentential form* is a member of $(\Sigma_M \cup N_M)^*$ that can be derived in a finite number of steps from the start symbol $S_M \in N_M$; (8)
- From Table 2, derivation rules P_M are adopted from $P_R \in$ RTL for MTL (9)

Using Aristotelian Syllogism (Russell 1945) three necessary tautologies are deduced:

1. Since any string $\mathcal{wm} \in MTL$ is generated from *high level languages* and since all high level languages are formal then $\{N_M\}$ is formal (10)
2. Since $\{\Sigma_M\}$, are adopted from RTL and since RTL is formal then $\{\Sigma_M\}$ is formal (11)
3. Since derivations P_M are derived from $P_R \in$ RTL which is formal, them P_M is formal (12)

From 10, 11, 12, grammar $G_M = (\Sigma_M, N_M, S_M, P_M)$, for MTL with corresponding semantics;

$\alpha \Longrightarrow G\beta$ iff $\exists\ u, v, p, q \in (\Sigma_M \cup N_M)^* : \alpha = upv \wedge p \rightarrow q \in P_M \wedge \beta = uqv$, is declared FORMAL (13)

From 13 it is concluded that MTL is a formal system just like RTL or any other language except that MTL is a device to describe flow of information in machine memory (RAM).

6. Conclusions and recommendations

MTL is a device that can be used to visualize most aspects taught in elementary programming. MTL can be used along with current books, class notes and laboratory examples to visualize basic programming concepts ranging from *variable declaration*, *data feeding*, *variable initialization*, *data processing*, *flow of control*, *functions*, *arrays* and *outputting*. Unlike other visualization tools, MTL can be plugged and played in books and notes and laboratory examples as an effortless programmer-driven visualizer which gives complete authority to the programmer.

It has been demonstrated that MTL can be defined by a grammar using Chomsky's (1957) definition. It can be pointed out that with the MTL grammar, the novice and programming teachers can verify the correctness of their interpretation of codes to avoid tinkering.

Like other visualization tools, MTL has yet to be infused in mainstream teaching of programming. Until this is achieved, it is not possible to fairly determine its effectiveness and impact in teaching programming. This study is the first attempt to formalize MTL. Further work to demonstrate MTL

formalism by using graph grammars is recommended. Class experiments to measure the effectiveness of MTL in virtual learning environment, children schools and huge classes are recommended.

7. References

Ben-Ari, M. (2001) Program visualization in theory and practice. Informatik/Informatique, 2, 8-11.

Chomsky, N. (1957) Three models for the description of language. Department of Modern and Research Laboratory of Electronics. Massachusetts Institute of Technology, Cambridge, Massachusetts.

Dehnadi, S. (2006) Testing programming aptitude. In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 18.

Du Boulay, B. 1986. Some difficulties of learning to program. Journal of Educational Computing Research, 2(4), 459-472.

Hundhausen, C. D., and Brown, J. L. (2007) What you see is what you code: A 'live' algorithm development and visualization environment for novice learners. Journal of Visual Languages and Computing, 18(1), 22-47.

Hundhausen, C. D., Douglas, S. A. and Stasko, J. D. (2002) A meta-study of algorithm visualization effectiveness. Journal of Visual Languages and Computing, 13, 259-290.

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003) The BlueJ system and its pedagogy. Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, 13(4).

Kuittinen, M., Tikansalo, T. and Sajaniemi, J. (2008) A Study of the development of students' visualizations of program state during an elementary Object-oriented Programming course. *ACM Journal of Educational Resources in Computing*, 7(4).

Msele, L. (2010a) Enhancing comprehension by using Random Access Memory (RAM) diagrams in teaching programming: class experiment. In *Proceedings of the 22nd Annual Psychology of Programming Interest Group* (Universidad Carlos III de Madrid, Leganés, Spain, September 19-22, 2010). Joseph Lawrence and Rachel Bellamy, editors.

Mselle, L. (2010b) *C++ for novice programmers*. LAP LAMBERT Academic Publishing, Berlin.

Mselle, L. (2011a) *Java for novice programmers*. LAP LAMBERT Academic Publishing, Berlin.

Mselle, L. (2011b) *C for novice programmers*. LAP LAMBERT Academic Publishing, Berlin.

Mselle, L. (2011c). The Impact of *Memory Transfer Language* (MTL) on reducing misconceptions in teaching programming. ITCSE 2011, TU, Darmstadt, Germany, June, 27-29.

Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. (2004) Visualizing programs with Jeliot 3. Proceedings of the Working Conference on Advanced Visual Interfaces, Gallipoli, Italy, 373-376.

Naps, T., R¨oßling, G., Almstrum, V., Dann, W. Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Vel´azquez-Iturbide, A. (2003). Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2), 131–152.

Perkins, D.N, Hancock, C., Hobbs, R. Martin, F. and Simmons, R. (1986) Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-55.

Preparata, F. (1985) *Introduction to computer engineering*. Prentice Hall, New York.

Rajala¸ T., Laakso, M., Kaila, E. and Salakoski, T. (2008) Effectiveness of program visualization: A case study with the ViLLe Tool. Journal of Information Technology Education: Innovations in Practice.

R¨oßling G. and Ackermann, T. (2006) A Framework for generating AV content on-thefly. In Guido R¨oßling, editor, Proceedings of the Fourth Program Visualization Workshop, Florence, Italy, 106–111.

R¨oßling, G. and Vellaramkalayil, T. (2008) First Steps Towards a visualization-based computer science hypertextbook as a Moodle module. CS Department, TU Darmstadt, Hochschulstr, 10 64289 Darmstadt, Germany.

Russell, B. (1945) *The History of western philosophy*, George Allen & Unwin, London.

Samurcay, R. (1985) The Concept of variable in programming: its meaning and use in problem-solving by novice programmers. Education Studies in Mathematics, 16(2), 143-161.

Ziegla, U. and Crews, T. (1999) An Integrated program development tool for teaching and learning how to program. In *Proceedings of the 30th. SIGCSE Symposium*, March 1999, 276-280.