GUTSE WORKING PAPER PJ102:

PREDICTING THE EFFORT OF PROGRAM LANGUAGE COMPREHENSION - THE CASE OF HLL VS. ASSEMBLY -

Pontus Johnson, Mathias Ekstedt
Department of Industrial Information and Control Systems
Royal Institute of Technology (KTH)
SE-100 44 Stockholm, Sweden
{pj101, mek101}@ ics.kth.se

Abstract. One important aspect of the quality of programming languages is the effort required by a programmer to understand code written in the language. A historical case where this issue was at the forefront was in the debate between the proponents of high-level languages (HLL) and Assembly languages, where the main argument for HLLs were that they were easier for people to understand.

Being one out of a series of articles arguing for a unified theory for software engineering, this article proposes the use of a specific theoretical model from the discipline of cognitive psychology as a tool for predicting language comprehension effort. Describing human problem solving faculties, the ACT-R model [Anderson and Lebiere 1998] predicts that the effort of understanding a program written in C is only 36,5% of the effort of understanding a comparable program written in Assembly.

In order to validate the theory, an experiment was performed where a number of engineering students were exposed to tasks of program comprehension. This empirical assessment demonstrated that the effort of understanding a program written in C is 32,5% of the effort of understanding a comparable program written in Assembly. Comparing the results of the theoretical predictions and the empirical assessments of program comprehension effort, we find that the theoretical model performs surprisingly well. The prediction error for the execution of an Assembly program was 5,1% while the error for C was 6,8%. The prediction error for the ratio between the two program languages amounted to 12,6%.

Keywords. ACT-R, Programming Language, C, Assembly, HLL, Unified Theory of Software Engineering, Program Comprehension.

1 INTRODUCTION

1.1 Assembly versus HLL

A couple of decades ago, a heated debate raged over the pros and cons of high-level languages on the one hand and Assembly languages on the other. Proponents of high-level languages claimed that these were much better suited to the problem-solving involved in program development while the advocates of Assembly maintained that high-level languages were less efficient than Assembly. Although high-level languages have won the day in the eyes of most beholders, there are still those who defend the relevance of Assembly programming.

1.2 Dimensions of Program Language Quality

In the debate of programming language quality several arguments are normally put forth. Figure 1 outlines different lines of argumentation. Firstly, there are arguments concerning the suitability of programming languages with respect to the machines that in the end will execute the programs. These arguments are mostly concerned with issues such as the expressiveness of programming languages, i.e. what applications written in the languages can do, machine efficiency, i.e. how rapidly they can do them, and how easily they can be manipulated by other applications, such as compilers.

Secondly, there are arguments concerning the suitability of programming languages with respect to the human mind. These arguments normally focus on issues such as program comprehension, i.e. how difficult it is for a programmer to understand a given program, program language learnability, i.e. how difficult it is for a programmer to learn the language, and programmer efficiency, i.e. with what effort a programmer can accomplish a given task with the programming language.

Thirdly, there are arguments concerning the suitability of programming languages with respect to software developing organizations. Among these arguments, one states that certain languages provide code modularization mechanisms that facilitate division of labor by reducing the required amount of communication between programmers. Related arguments concern the extent to which the interfaces of such modules support distributed development.

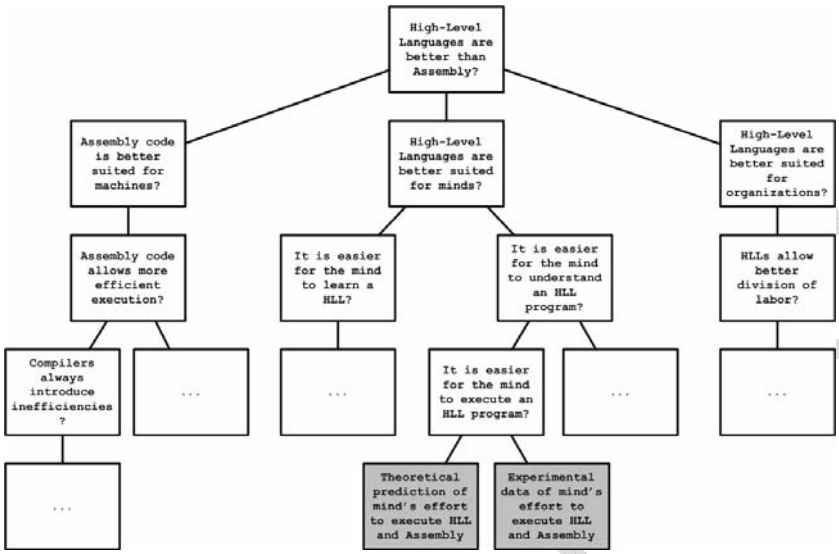

Figure 1. Outline of the argumentation for and against HLLs.

1.3 Purpose

In this article, we report on the shaded boxes in Figure 1, i.e. arguments concerning the effort required by the human mind to understand programming languages.

The article presents theoretical predictions of the effort of program execution with respect to two programming languages; C (a version developed by [B Knudsen Data, 2005]) and an Assembly programming language (a version developed by [Microchip Technologies, 2002] [Microchip Technologies, 2002]). The predictions are experimentally validated.

1.4 A Unified Theory of Software Engineering

This article is one in a series of publications arguing for the need of a unified theory of software engineering. Looking at other academic disciplines, the history of successful science is written in terms of its unified theories. In 1687, Sir Isaac Newton proposed a unified theory of mechanics in order to explain and predict all earthly and heavenly motion [Newton, 1687]. In 1803, John Dalton revived a unified theory of matter, the atomic theory, explaining the

nature of all matter [Dalton, 1808]. In 1839, Matthias Schleiden and Theodor Schwann developed the theory of the cell, explaining the fundamental structure of living organisms [Schwann, 1839]. In 1859, Charles Darwin proposed a unified theory of evolution by natural selection in order to explain all variability of the living [Darwin 1859]. In 1869, Dmitri Ivanovich Mendeleev presented the periodic system, a unified theory explaining the properties of all the chemical elements [Mendeleev 1869]. In 1990, Allen Newell proposed a unified theory of cognition in order to explain and predict all human problem-solving [Newell 1990].

Software engineering, however, only uses a large set of micro-theories that lack a common ground. Thus we are hampered by the shortage of the powerful conceptual vehicle of scientific thought that is found in a unified theory. Arguments for the benefits and viability of a unified theory of software engineering are presented in [Johnson and Ekstedt, 2005b]. A proposal for a unified theory of software engineering is detailed in [Johnson and Ekstedt, 2005a]. In this paper, one part of this proposed unified theory is corroborated by an empirical validation of its predictive capacity.

1.5 Outline

Section two outlines the experiment measuring language comprehension in terms of cognitive effort of executing computer programs. Section three delves into the details of the theory that is used for predicting the execution effort. Furthermore, the section presents the actual predictions. In section four the experimental setup and results are described. A detailed analysis and comparison of the theoretical predictions and experimental results are found in section five. Finally, the paper is concluded in section six.

2 PROGRAM LANGUAGE COMPREHENSION MEASURED BY EXECUTION EFFORT

Program comprehension is an established interest area within software engineering [Brooks 1983] [Soloway and Ehrlich, 1984] [Robson et al., 1991] [Paul et al., 1991]. One of the pioneers in the discipline, Schneiderman, views program comprehension as consisting of three levels [Schneiderman, 1980]: low-level comprehension of the function of each line of code, mid-level comprehension of the nature of the algorithms and data, and high-level comprehension of overall program function. Much of the work in the discipline has been conducted on Schneiderman's higher levels, where the overall function

and purpose of the program is focus. In this article, however, only the lowest level of comprehension is addressed. With respect to this level of program comprehension, a reasonable measure of the difficulty for the human mind to understand a program is the time it takes for the mind to execute that program. The mind thus acts as if it where a processor, interpreting the program instruction by instruction.

In order to determine the difference in ease of understanding between two different programming languages, we can employ the above measure, comparing the effort of executing a program in each language. Of course, the two programs in the two languages must be comparable. In this paper, we consider a program in Assembly versus a program in C, both performing a functionally identical task, namely to sort a list of integers according to the "bubble sort" algorithm, cf. Figure 2. The complete C program is presented in Figure 3, and the Assembly program is presented in Figure 4.

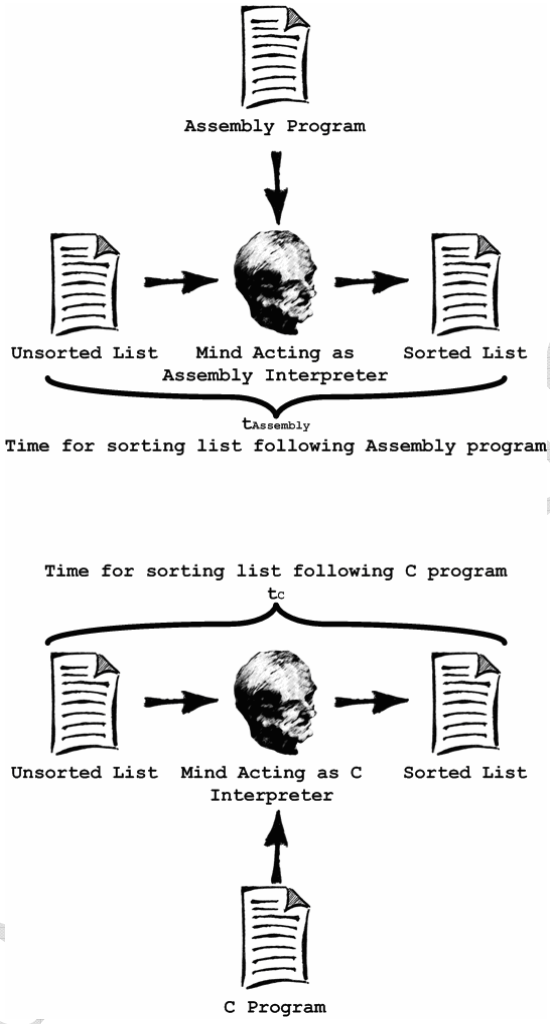

Figure 2. The execution time for the ACT-R mind interpreting Assembly, $t_{Assembly}$ is compared to the execution time for the mind interpreting C, t_C.

```
void main() {
  int i[9];
  int x, y, h, l;

  i[0]=9;
  i[1]=8;
  i[2]=7;
  i[3]=6;
  i[4]=5;
  i[5]=4;
  i[6]=3;
  i[7]=2;
  i[8]=1;
  for(x = 0; x < 3; x++) {
    for(y = 0; y < 2; y++) {
      h = i[y+1];
      l = i[y];
      if(l > h) {
        i[y] = h;
        i[y+1] = l;
      }
   }
  }
}
```

Figure 3. Bubble Sort program in C.

```
MOVLW .9
MOVWF i
MOVLW .8
MOVWF i+1
MOVLW .7
MOVWF i+2
MOVLW .6
MOVWF i+3
MOVLW .5
MOVWF i+4
MOVLW .4
MOVWF i+5
MOVLW .3
MOVWF i+6
MOVLW .2
MOVWF i+7
MOVLW .1
MOVWF i+8
CLRF  x
MOVLW .3
SUBWF x,W
BTFSC 0x03,Carry
GOTO  m007
CLRF  y
MOVLW .2
SUBWF y,W
BTFSC 0x03,Carry
GOTO  m006
MOVLW .i+1
ADDWF y,W
MOVWF FSR
MOVF  INDF,W
MOVWF h
MOVLW .i
ADDWF y,W
MOVWF FSR
MOVF  INDF,W
MOVWF l
SUBWF h,W
BTFSC 0x03,Carry
GOTO  m005
MOVF  h,W
MOVWF INDF
MOVLW .1
ADDWF y,W
MOVWF FSR
MOVF  l,W
MOVWF INDF
INCF  y,1
GOTO  m003
INCF  x,1
GOTO  m001
SLEEP
```

Figure 4. Bubble Sort program in Assembly.

3 THEORETICAL PREDICTION OF EXECUTION EFFORT

3.1 The ACT-R Model of the Mind

As described in [Johnson and Ekstedt, 2005a], the unified theory of software engineering proposed by the authors incorporates a model of the human mind, ACT-R, developed by Anderson et al. at the Carnegie Mellon University [Anderson, 1983] [Anderson and Lebiere, 1998]. In this model, it is possible to describe the concrete steps performed by a mind when executing an Assembly instruction or a C statement.

The ACT-R model of the mind is not unlike the von Neumann computer architecture. Recall that the von Neumann computer features a central

processing unit (an executor), a memory for both programs and data, and an input and an output system.

Comparing the ACT-R model of the mind to the von Neumann computer, we find many similarities: there is an *executor*, and there is an *input* and *output system* (Figure 5). There is also *memory* in both the mind and the machine. On a high level, we might make an analogy between that part of the mind called the *procedural memory system*, and the machine's program memory. Both store instructions that can be executed by the executor. An analogy may also be made between the part of the mind's memory called the *declarative memory system* and the machine's data memory. Both these systems store knowledge that can be evaluated and manipulated by the program instructions or production rules.

The analogy, however, ends there. A main feature of the mind is that it is *goal-driven*; it has goals and these goals influence its behaviour. The mind will choose what instruction to execute depending on its current goal. The executed instruction will in turn modify the goal of the mind. Since the goal has been modified, the subsequent instruction will be different from the previous one. So the execution of the mind may be represented as a constant transformation of goals. In this process of goal transformations, there are side effects that may exhibit themselves as, for instance, external behaviour (like speech or movement).

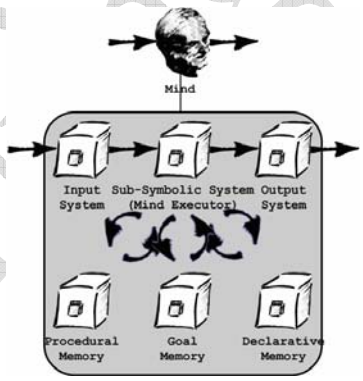

Figure 5. The ACT-R model of the mind.

The compositional units of the declarative memory are called *chunks*. Chunks store knowledge, like the name of a person, the fact that three plus four equals seven, etc. The goals are represented as residing in their own memory system, the *goal memory system*, which contains a special kind of chunks; *goal chunks*. A mind may have several goals, but only one can be active at any one time. It is

often convenient to view goals as stored in a stack, where new goals can be pushed and attained goals popped.

The compositional units of the procedural memory are called *production rules*. These are the analogues to the instructions of a von Neumann machine. A production rule has the following general form:

```
IF goal_condition (+ chunk_retrieval)
THEN goal_transformation (+ action).
```

The procedural memory system contains a large amount of different production rules in no particular order. What production rule is selected for execution depends on the left-hand side of the equation, and the result of the production rule execution is given by the right-hand side. The parentheses mean that "chunk retrieval" and "action" are optional. The drivers of the mind are thus a sequence of goal transformations; the goal (and optionally some other chunk) determines whether the production rule is executed, and the result of the execution is a transformation of the goal (and optionally some action).

The memory of the mind, be it procedural, declarative or goal memory, differs from the memory of the von Neumann machine in not only its structure, but also in its persistence; people have a hard time learning and remembering things.

In addition to the memory systems and the executor, the mind also features an input and an output system. The input system continually presents chunks that may or may not be considered by the executor. These chunks represent the external stimuli that the person is experiencing. They might, for instance, be a sequence of words as the person reads a text. Chunks are on a fairly high level of abstraction, where many interpretations are performed directly by the input system (as, e.g., the interpretation of certain patterns of black ink on a paper as a word in English).

The output system is manipulated by the executor in line with the production rules that are being executed. Also the output system provides a high-level interface to the executor (so that one action could be the writing of a word on a paper).

Summarizing, according to the ACT-R, the mind has an executor, an input and an output system, and three different memory systems. The procedural memory contains executable production rules, the declarative memory contains chunks, and the goal memory contains the goals that guide the mind's behaviour.

3.2 Theoretical Setup

This section presents the manner in which the ACT-R model was employed in order to predict the effort of program comprehension.

In order to make the prediction as precise as possible, our instantiation of the model is closely matched to the experimental setup employed to validate the prediction. We assume that the person executing the statement is positioned in front of a computer with two windows open. In one window, the program text is presented while in the other window, the states of the relevant variables are located (cf. the experimental setup in FiguresFigure 9 and Figure 10). The person can manipulate the states of the variables by pressing various keys on the keyboard.

ACT-R allows for several levels of detail in the workings of the mind. In this prediction, we will use the least detailed level, which is called the symbolic level. On this level, a fairly simple set of production rules and the input and output systems are employed. The so called sub-symbolic level, treating issues such as learning and forgetting, conflicting production rules, is thereby ignored.

Figure 6 presents the production rules required to execute the various Assembly instructions. As an example, let us consider MOVF x,W. The instruction states that the value of the variable x should be copied to the working register W. In this example, the mind first selects the working register (by manipulating a development environment user interface), then reads the current value of the variable x (this value is presented in the user interface), and finally writes that value to the previously selected working register (by pressing the appropriate key on the keyboard). This instruction is thus coded in three ACT-R production rules. These productions in turn specify the reading of three chunks ("MOVF", "x" and "3") and the performing of two motor functions (selecting the working register and writing "3"). In Figure 7, all the C statements are expressed in terms of production rules required for their execution.

In relation to Schneiderman's program comprehension levels, it can be noted that in order to predict the understanding on the higher levels, a different set of production rules, chunks, and goal chunks have to be specified in ACT-R.

```
MOVLW .3
IF the goal is to execute the next instruction AND "MOVLW" is read
THEN set goal to read parameter AND select the Working Register
IF the goal is to read the parameter AND ".3" is read
THEN set goal to execute the next instruction AND write "3"

INCF x,1
IF the goal is to execute the next instruction AND "INCF" is read
THEN set goal to read parameter
IF the goal is to read the parameter AND "x" is read
THEN set goal to read the value of "x" AND select the "x" variable
IF the goal is to read the value of "x" AND "3" is read
THEN set goal to retrieve 3+1
IF the goal is to retrieve 3+1 AND "4" is retrieved
THEN set goal to execute the next instruction AND write "4"

CLRF x
IF the goal is to execute the next instruction AND "INCF" is read
THEN set goal to read parameter
IF the goal is to read the parameter AND "x" is read
THEN set the goal to clear the parameter AND select the "x" variable
IF the goal is to clear the parameter
THEN set goal to execute the next instruction AND write "0"

MOVWF h
IF the goal is to execute the next instruction AND "MOVWF" is read
THEN set goal to read parameter
IF the goal is to read the parameter AND "h" is read
THEN set the goal to read the Working Register AND write "h"
IF the goal is to read the Working Register AND "3" is read
THEN set goal to execute the next instruction AND write "3"

MOVWF INDF
IF the goal is to execute the next instruction AND "MOVWF" is read
THEN set goal to read parameter
IF the goal is to read the parameter AND "INDF" is read
THEN set the goal to read the File Select Register AND write "i"
IF the goal is to read the File Select Register AND "2" is read
THEN set the goal to read the Working Register AND write "2"
IF the goal is to read the Working Register AND "4" is read
THEN set goal to execute the next instruction AND write "4"

MOVF x,W
IF the goal is to execute an instruction AND "MOVF" is read
THEN set goal to read parameter AND select the Working Register
IF the goal is to read the parameter AND "x" is read
THEN set goal to read the value of "x"
IF the goal is to read the value of "x" AND "3" is read
THEN set goal to execute the next instruction AND write "3"
```

Figure 6. Production rule set for executing Assembly instructions (cont.).

```
MOVF INDF,W
IF the goal is to execute the next instruction AND “MOVF” is read
THEN set goal to read parameter AND select the Working Register
IF the goal is to read the parameter AND “INDF” is read
THEN set goal to read the File Select Register
IF the goal is to read the File Select Register AND “3” is read
THEN set goal to read the value of i[3]
IF the goal is to read the value of i[3] AND “6” is read
THEN set goal to execute the next instruction AND write “6”

ADDWF y,W
IF the goal is to execute the next instruction AND “ADDWF” is read
THEN set goal to read parameter AND select the Working Register
IF the goal is to read the parameter AND “y” is read
THEN set goal to read the value of “y”
IF the goal is to read the value of “y” AND “2” is read
THEN set goal to read the Working Register
IF the goal is to read the Working Register AND “1” is read
THEN set goal to retrieve “2+1”
IF the goal is to retrieve “2+1” AND “3” is retrieved
THEN set the goal to execute the next instruction AND write “3”

SUBWF x,W
IF the goal is to execute the next instruction AND “SUBWF” is read
THEN set goal to read parameter AND select the Working Register
IF the goal is to read the parameter AND “x” is read
THEN set goal to read the value of “x”
IF the goal is to read the value of “x” AND “2” is read
THEN set goal to read the Working Register
IF the goal is to read the Working Register AND “1” is read
THEN set goal to retrieve “2-1”
       IF the goal is to retrieve “2-1” AND “1” is retrieved
       THEN set the goal to select the Carry Bit AND write “1”
       IF the goal is to select the Carry Bit
       THEN set the goal to set the Carry Bit AND select the Carry Bit
       IF the goal is to set the Carry Bit
       THEN set the goal to execute the next instruction AND write “1”

BTFSC 0x03,Carry
IF the goal is to execute the next instruction AND “BTFSC” is read
THEN set goal to read the Carry Bit
       IF the goal is to read the Carry Bit AND “1” is read
       THEN set the goal to execute the next instruction
       IF the goal is to read the Carry Bit AND “0” is read
       THEN set the goal to execute the next-next instruction

GOTO m002
IF the goal is to execute the next instruction AND “GOTO” is read
THEN set goal to read parameter
IF the goal is to read the parameter AND “m002” is read
THEN set the goal to execute the instruction at m002
```

Figure 6. (Cont.) Production rule set for executing Assembly instructions.

```
i[3]=4
IF the goal is to execute the next statement AND “i[..]=” is read
THEN set goal to read the index AND write “i”
IF the goal is to read the index AND “3” is read
THEN set goal to read the constant AND write “3”
IF the goal is to read the constant AND “4” is read
THEN set goal to execute the next instruction AND write “4”

for(x=0;x<4;x=x+1)
IF the goal is to execute the next statement AND “for(…)” is read
THEN set goal to read the variable AND
IF the goal is to read the variable AND “x” is read
THEN set goal to read the value of “x” AND write “x”
       IF the goal is to read the value of “x” AND first pass
       THEN set the goal to check the condition AND write “0”
       IF the goal is to read the value of “x” AND other pass
            AND “2” is read
       THEN set goal to retrieve “2+1”
IF the goal is to retrieve “2+1” AND “3” is retrieved
THEN set the goal to check the condition AND write “3”
       IF the goal is to check the condition AND “<4” is read
       THEN set goal to execute the first clause statement
       IF the goal is to check the condition AND “<3” is read
       THEN set goal to execute the first statement after the clause

h=i[y]
IF the goal is to execute the next statement AND “h=” is read
THEN set goal to read the array AND write “h”
IF the goal is to read the array AND “i” is read
THEN set goal to read the index
IF the goal is to read the index AND “y” is read
THEN set goal to read the vale of “y”
IF the goal is to read the value of “y” AND “3” is read
THEN set goal to read the value of “i[3]”
IF the goal is to read the value of “i[3]” AND “4” is read
THEN set goal to execute the next instruction AND write “4”

h=i[y+1]
IF the goal is to execute the next statement AND “h=” is read
THEN set goal to read the array AND write “h”
IF the goal is to read the array AND “i” is read
THEN set goal to read the index
IF the goal is to read the index AND “..+1” is read
THEN set the goal to read the indexvariable
IF the goal is to read the indexvariable AND “y” is read
THEN set goal to read the value of “y”
IF the goal is to read the value of “y” AND “3” is read
THEN set the goal to retrieve “3+1”
IF the goal is to retrieve “3+1” AND “4” is read
THEN set goal to read the value of “i[4]”
IF the goal is to read the value of “i[4]” AND “2” is read
THEN set goal to execute the next instruction AND write “2”
```

Figure 7. Production rule set for executing C statements (cont.).

```
i[y]=l
IF the goal is to execute the next statement AND "i[..]=" is read
THEN set goal to read the index AND write "i"
IF the goal is to read the index AND "y" is read
THEN set goal to read the value of "y"
IF the goal is to read the value of "y" AND "3" is read
THEN set the goal to read the variable AND write "3"
IF the goal is to read the variable AND "l" is read
THEN set the goal to read the value of "l"
IF the goal is to read the value of "l" AND "4" is read
THEN set goal to execute the next instruction AND write "4"

i[y+1]=l
IF the goal is to execute the next statement AND "i[..]=" is read
THEN set goal to read the index AND write "i"
IF the goal is to read the index AND "..+1" is read
THEN set the goal to read the indexvariable
IF the goal is to read the indexvariable AND "y" is read
THEN set goal to read the value of "y"
IF the goal is to read the index AND "y" is read
THEN set goal to read the value of "y"
IF the goal is to read the value of "y" AND "3" is read
THEN set the goal to read the variable AND write "3"
IF the goal is to read the variable AND "l" is read
THEN set the goal to read the value of "l"
IF the goal is to read the value of "l" AND "4" is read
THEN set goal to execute the next instruction AND write "4"

if(l>h)
IF the goal is to execute the next statement AND "if(..)" is read
THEN set goal to read the expression
IF the goal is to read the expression AND "..>.." is read
THEN set the goal to read the first variable
IF the goal is to read the first variable and "l" is read
THEN set the goal to read to value of "l"
IF the goal is to read the value of "l" AND "3" is read
THEN set the goal to read the second variable
IF the goal is to read the second variable and "h" is read
THEN set the goal to read to value of "h"
IF the goal is to read the value of "h" AND "2" is read
THEN set the goal to retrieve "3>2"
       IF the goal is to retrieve "3>2" AND "true" is read
       THEN set the goal to execute the first clause statement
       IF the goal is to retrieve "3>4" AND "false" is read
       THEN set goal to execute the first statement after the clause
```

Figure 7. (Cont.) Production rule set for executing C statements.

In order to use the above production rules for predicting the effort of executing Assembly and C code, their latencies must be specified. The time to fire an ordinary production is normally set to 50 milliseconds [Anderson and Lebiere, 1998]. The latencies associated with reading and writing depend to a large extent on the particular user interface (the layout of text and symbols on the computer screen, and the mouse and keyboard configurations) and the normal procedure

is consequently to fit it to empirical data. In this experiment the time required for reading a parameter is estimated to 0,7 seconds while the time for writing is set to 0,8 seconds.

3.3 Theoretical Results

The prediction of execution effort for some Assembly instructions and C statements are presented in Figure 8. As an example, the CLRF instruction requires three productions, two read operations and two write operations, which amounts to an effort of 3*0,05s+2*0,7s+2*0,8s=3,15s.

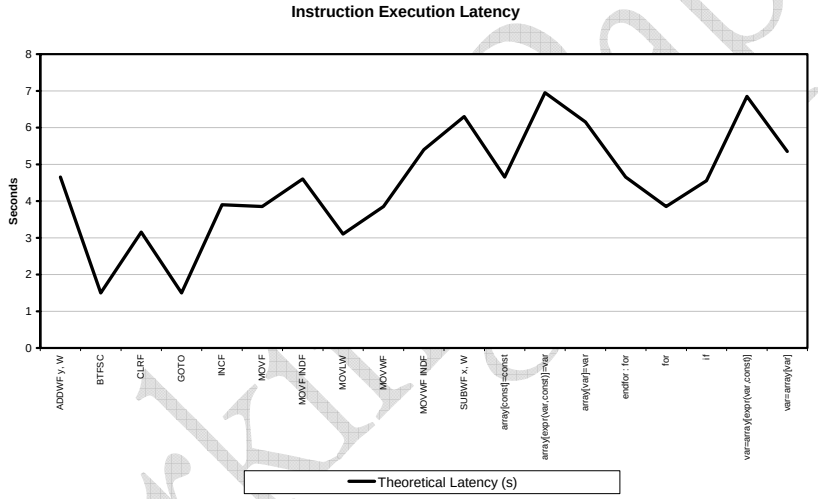

Figure 8. Predicted latencies for various Assembly instructions and C statements.

4 EXPERIMENTAL ASSESSMENT OF EXECUTION EFFORT

A theory may be corroborated by comparison to empirical observations. Here we do so with respect to the predictions presented in the previous section

4.1 Experimental Setup

The empirical observations were gathered during an experiment performed on a group of five engineering students. The students were subjected to the task of executing the same Assembly and C programs as presented above. The subjects performed the experiment in a software application where the appropriate latencies were recorded.

The graphical user interface of the software application is presented in Figure 9 and Figure 10. There are two windows, where one presents the program text and the other presents the state of the program variables. The subjects were instructed to manipulate the variables in the state window according to the text in the program window. For example, the MOVLW .3 instruction requires that the working register, variable *W*, is set to *3*. This is accomplished by the subject by first pressing the *W* key on the keyboard and subsequently pressing the key *3*.

The latencies for all key presses were recorded in the application. After the completion of the experiment, the key presses were mapped to the executed program instructions as detailed in Figures 6 and 7. The latencies for the complete execution of various instructions could then be aggregated. For instance, for the above MOVLW .3 instruction, the total latency was calculated as the time between the last key press of the previous statement and the last key press of the MOVLW .3 instruction, namely the key *3*.

Some program instructions do, however, not end with a key press, thus leaving their latencies undetermined. For instance, the C statement if(h>l) does not require any key presses at all, since no variable changes value at the execution of the statement. In order to obtain latencies for these statements, they were calculated from groups of statements that did contain key presses.

The prediction of the previous section assumes a person who is trained in executing these languages efficiently. The production rules are optimized so that no unnecessary tasks are performed. In order to obtain such data from the empirical assessment, the subjects were first trained in the code execution task. They spent approximately one hour practicing before conducting the actual test.

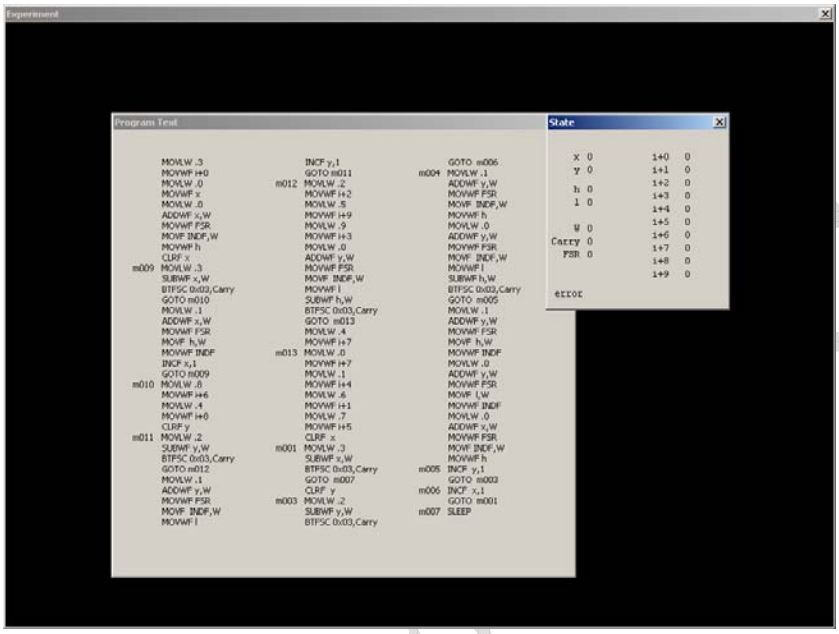

Figure 9. The graphical user interface for the Assembly experiment.

One undesirable possibility was that the subjects would deduce the ulterior purpose of the program and thus would be able to employ more efficient production rules to manipulate the state variables. For example, if the subjects understood that the program sorted a list according to some algorithm, they would not need to read every instruction carefully, but rather manipulate the state variables according to the algorithm. In order to avoid this, the programs employed were nonsense programs with no apparent rationality in their design.

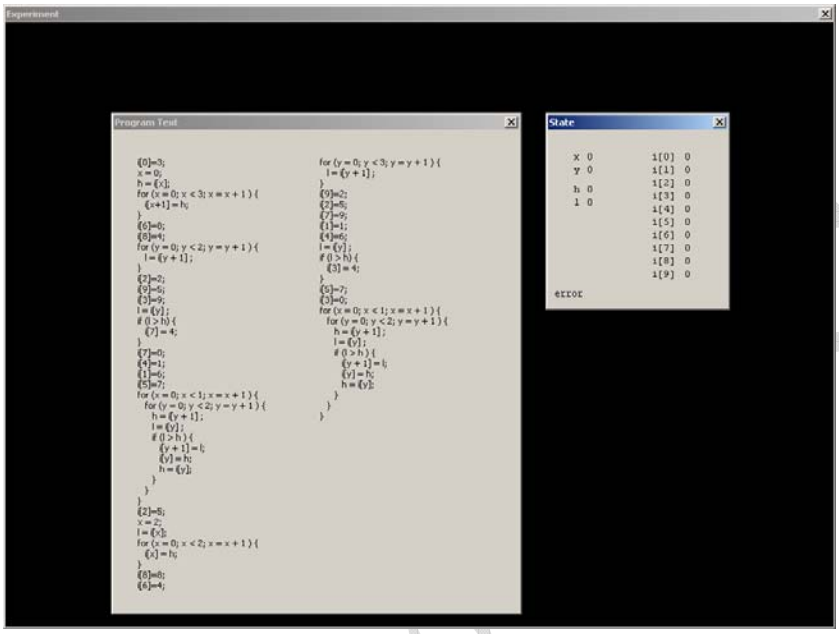

Figure 10. The graphical user interface for the C experiment.

Occasionally, the subjects became tired, confused or reengaged in learning activity, thus producing longer latencies than otherwise. In order to avoid taking such experimental data into account, the lowest 25% of the data set of a given individual was chosen (Figure 11) for further analysis.

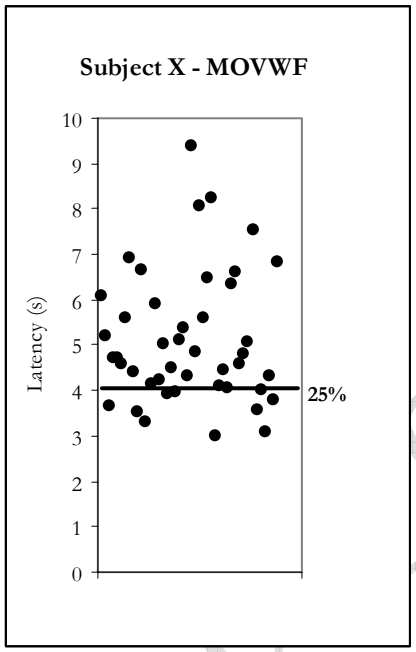

Figure 11. The diagram presents latency data from one experiment subject executing the Assembly instruction MOVWF. The line represents the lowest 25% latency number extracted for further analysis.

It also happened that the subjects committed errors. This was detected in two ways. Firstly, if the subject himself noted that an instruction had been incorrectly interpreted, a special key (Escape) could be pressed to record this event. Secondly, in the treatment of the execution traces, key presses that did not correspond to the program were detected. All latency data surrounding such errors was disqualified for further analysis, since it would be unrepresentative.

4.2 Empirical Results

The aggregated results are reported in Figure 12. The black line represents the experimentally assessed execution times, while the grey line denotes the theoretical prediction from Section 3.3.

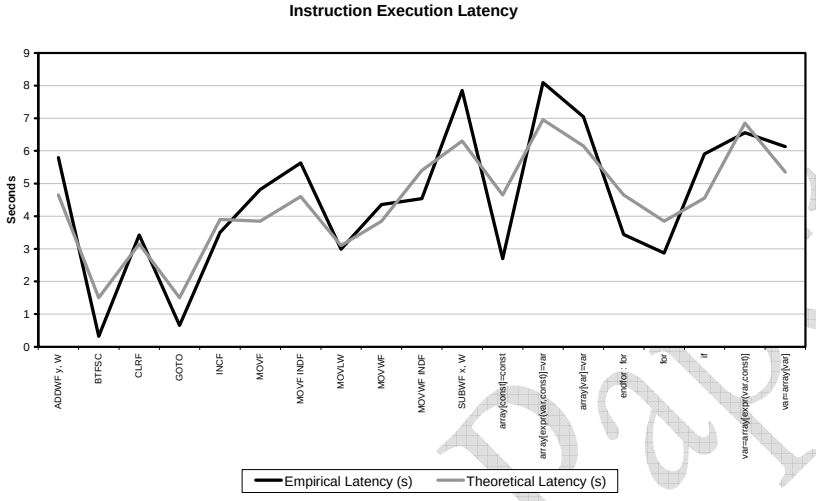

Figure 12. The black line represents experimentally measured latencies for various Assembly instructions and C statements. The gray line represents the theoretical predictions from Figure 8.

5 ANALYSIS

The above results may be further analyzed. In order to account for the fact that an Assembly program performing a certain task is typically longer than a C program performing the same task (counting lines of code), we calculate the total time required for performing a certain task.

In Section 2, Figure 3 and Figure 4 present two programs, in C and Assembly respectively, that functionally perform the same task, namely that of sorting an array of numbers according to the Bubble Sort algorithm.

By executing the two programs, we obtain execution traces. These traces provide the instruction distribution of Table 1 and Table 2. According to these tables, the same list sorting activity requires the execution of 46 statements if performed in C, and 179 statements if performed in Assembly.

Instruction	Population in Bubble Sort	Empirical Latency (s)	Theoretical Latency (s)
ADDWF y, W	15	5,790	4,65
BTFSC	19	0,320	1,5
CLRF	4	3,425	3,15
GOTO	16	0,654	1,5
INCF	9	3,503	3,9
MOVF	6	4,815	3,85
MOVF INDF	12	5,630	4,6
MOVLW	37	2,989	3,1
MOVWF	36	4,356	3,85
MOVWF INDF	6	4,539	5,4
SUBWF x, W	19	7,851	6,3

Table 1. The second column presents the distribution of Assembly instructions in trace of Bubble Sort program execution. The third and fourth columns present the experimentally and theoretically derivedlatencies respectively.

Statement	Population in Bubble Sort	Empirical Latency (s)	Theoretical Latency (s)
array[const]=const	9	2,699	4,65
array[expr(var,const)]=var	3	8,090	6,95
array[var]=var	3	7,046	6,15
endfor : for	9	3,441	4,65
for	4	2,870	3,85
if	6	5,903	4,55
var=array[expr(var,const)]	6	6,555	6,85
var=array[var]	6	6,131	5,35

Table 2. The second column presents the distribution of C statements in trace of Bubble Sort program execution. The third and fourth columns present the experimentally and theoretically derivedlatencies respectively.

Considering the accuracy of the theoretical prediction of the effort required for a human mind to execute the Bubble Sort algorithm in C and Assembly, we find the following. With respect to Assembly, the theoretical model predicted an execution effort of 10 minutes and 54 seconds. The experiment assessed the effort to 11 minutes and 29 seconds, resulting in a prediction error of 5,1%. With respect to C, the theoretical model predicted an execution effort of 3 minutes and 59 seconds, while the experiment assessed the effort to 3 minutes and 44 seconds. This resulted in a prediction error of 6,8%. Finally, with respect

to the relative execution effort between C and Assembly, the theoretical model predicted that the Bubble Sort program written in the C programming language would require 36,5% of the effort of that written in Assembly. The empirical assessment resulted in a effort ratio of 32,5%. The prediction error thus landed on 12,6%.

6 CONCLUSIONS

This article is part of a series of articles arguing for a unified theory of software engineering [Johnson and Ekstedt, 2005a] [Johnson and Ekstedt, 2005b]. A model of the human mind, ACT-R [Anderson and Lebiere 1998] has been employed in order to predict the effort required for a person to understand a problem in a given programming language. Programs written in the languages Assembly and C were used as examples. The theoretical model predicted that the effort associated with understanding a C program was 36,5% of the effort of understanding the corresponding Assembly program.

In order to validate the theoretical model, a group of engineering students were subjected to an experiment designed to determine their factual effort of understanding with respect to the two programming languages. The experiment assessed the effort ratio to 32,5%. The prediction error thus became 12,6%.

The results demonstrate the viability of the approach. It is reasonable to believe that the theoretical model of the human mind may be used in the assessment of a great many issues in software engineering. In addition to the issue of programming language quality, also the quality of other software engineering artifacts could be assessed, such as application program interfaces, software design languages, and formal specifications.

7 REFERENCES

Anderson, J. R., (1983), *The Architecture of Cognition*, Harvard University Press.

Anderson, J. R. and C. Lebiere, (1998(, *The atomic components of thought*. Lawrence Erlbaum Associates Publishers.

B Knudsen Data, (2005), CC5X C Compiler for the PICmicro Devices, Version 3.2, User's Manual, (Available at http://www.bknd.com/cc5x/).

Brooks R., (1983), *Towards a Theory of the Comprehension of Computer Programs*, International Journal of Man-Machine Studies, 18:543-554.

Dalton, J., (1808), *A New System of Chemical Philosophy*.

Darwin, C., (1859), *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*.

Johnson, P. and M. Ekstedt, (2005a), *The Grand Unified Theory of Software Engineering*, Preprint, Royal Institute of Technology, Sweden.

Johnson, P. and M. Ekstedt, (2005b), *Towards a Unified Theory of Software Engineering*, Royal Institute of Technology, Sweden.

Mendeleev, D., (1869), *Principles of Chemistry*.

Microchip Technologies, (2002), PIC16C5X Data Sheet EPROM/ROM-Based 8-bit CMOS Microcontroller Series, (Available at http://www.microchip.com).

Microchip Technologies, (2005), MPLAB® IDE User's Guide, (Available at http://www.microchip.com).

Newell, A., (1990), *Unified Theories of Cognition*, Harvard University Press. 1990.

Newton, I., (1678), *Philosophiae Naturalis **Principia** Mathematica*.

Paul S. et al., (1991), "Theories and techniques of program understanding", *Proceedings of the conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press.

Soloway E., and K. Ehrlich, (1984) *Empirical Studies of Programming Knowledge*, IEEE Transactions on Software Engineering, SE-10(5):595-609, September.

Schneiderman B., (1980), *Software Psychology, Human Factors in Computer and Information Systems*, Winthrop Publishers.

Schwann, T., (1839), *Mikroskopische Untersuchungen uber die Ubereinstimmung in der Structur und dent Wachsthum der Thiere und Pflanzen.*