On Experiments for Measuring Cognitive Weights for Software Control Structures

Volker Gruhn, Ralf Laue
University of Leipzig, Germany, Chair of Applied Telematics / e-Business
∗ {gruhn,laue}@ebus.informatik.uni-leipzig.de

Abstract

Shao and Wang have proposed a cognitive complexity measure[8] as a metric that can be used for estimating the comprehension effort for understanding software written in imperative programming languages. The key idea of their approach is to assign a cognitive weight to basic software control structures. The more difficult a control structure is to understand, the greater is its cognitive weight. In this paper, we discuss the experiments that have been used for calibrating the cognitive weights and show how they can be improved.

Keywords: cognitive complexity, cognitive weights, complexity metrics

1 Introduction

Since the late 1960’s, several complexity metrics have been proposed to measure the difficulty of understanding a piece of software. Very soon, it has been found that measuring the complexity of a program takes much more than just counting its lines of code. Curtis, Sheppard and Milliman wrote in [1]: “When the code grows beyond a subroutine or module, its complexity to the programmer is better assessed by measuring constructs other than the number of lines of code.” The reason for this is that a program is understood by a programmer in small pieces, not as a whole.

Shao and Wang have proposed a cognitive complexity measure[8] as a metric for measuring the cognitive complexity of software. The key idea of their approach is to assign a cognitive weight to basic software control structures. The more difficult a control structure is to understand, the greater is its cognitive weight. The easiest control structure - a number of operations is executed in a sequence without any branching - is defined to have the cognitive weight 1.

More advanced control structures (like parallel execution) have greater cognitive weights. This idea to assign complexity measures to control structures was already used by McQuaid [5] who defined so called “inherent complexities” for Ada95 statements.

For assigning realistic cognitive weights to the software control structures, psychological experiments must be conducted. In the following chapters, we compare the cognitive weights proposed by [8], [5] and [10] and discuss the setup of the experiments used in [10].

2 Cognitive Weights in [5] and [8]

Table 1 shows the cognitive weights proposed in [8] and [10] and the “inherent complexities” from [5]. One observation is that the three sources do not agree about the relations between the complexities of the control structures. The question arises how the authors found the numbers in Tab. 1.

[8] says that the cognitive weights in this paper have been defined “based on empirical studies in cognitive informatics”. However, the layout of the experiments (if there were any) has never been published. From a scientific point of view, this means nothing else than that these empirical studies have to be regarded as being non-existent.

[5] stresses that the inherent complexity weights used in their paper have been defined “as a starting point” as a subjective measurement. No experiments have been carried out.

3 Cognitive Weights in [10]

The first paper describing an experiment for calibrating cognitive weights for control structures is [10]. In this section, we will describe the setup of the experiment and discuss its validity.

Undergraduate and graduate software engineering students were given ten Java code fragments, each of them represented one control structure. After reading the code,
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Sequence</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>if-then-[else]</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>case</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Iteration</td>
<td>for-do</td>
<td>3</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>repeat-until</td>
<td>3</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>while-do</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Embedded Component</td>
<td>Call of a user-defined function</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Recursion</td>
<td>3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Concurrency</td>
<td>Parallel Execution</td>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interrupt</td>
<td>4</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Cognitive Weights

the students have had to answer questions about the value of a variable resulting from the execution of the code. The time needed to read the program and give the answer has been recorded. Assuming the cognitive weight of the control structure “sequence” being 1 for each student, the cognitive weights of the other control structures have been calculated as the quotient “time needed for the control structure X” / “time needed for the control structure sequence”.

Now, we have a closer look at the Java code fragments that represented the control structures.

The repeat-loop-construct was represented by this code:

```java
int Test5 (int A=1, B=2) {
    int i=0;
    do {
        B := B+i
    } while (i++ =2);
    return B-A
}
```

The parallel-construct was represented by this Java code:

```java
// The parallel class A (simulated)
pub int Test9A (int A=1, B=2) {
pub Boolean Test9A_completed;
pub int C;
C := A+B;
Test9A_completed := true;
Return C
}
// The parallel class B (simulated)
pub int Test9B (int A=3, B=2) {
pub Boolean Test9B_completed;
pub int D;
D:= A-B;
Test9B_completed := true;
Return D
}
//The convergence class
// Assume both parallel classes Test9A and Test 9B
// have already been completed
pub int Test9(int C,D) {
    if Test9A_completed & Test9B_completed
        return (C*D) *2
    else return 0
}
```

The first observation is that these code fragments in fact are not syntactically correct Java programs. Errors include a wrong assignment operator := (which does not exist in Java), wrong keywords (pub instead of public), wrong capitalisation (Boolean) to name just a few. This problem already invalidates the value of the experiment, because there could be understanding problems that did result from the wrong syntax instead of the difficulty to understand the execution of the program.

Another observation is that the code fragments clearly differ in length. To give an example, the code fragment representing the repeat-loop had 7 lines of code while the code fragment representing the interrupt construct had 21 lines of code (not counting comment lines). Taking this in account, it is not very surprising to read that the result of the experiment was that the “repeat”-fragment was assigned a cognitive weight of 7 while the “interrupt”-fragment was assigned a cognitive weight of 22. The time difference measured in the experiment can easily be explained by the fact that it takes three times longer reading 21 lines of code than reading 7 lines of code. For this reason, we argue that the experiment did not reveal anything about the real difficulties to understand the control structures.

Before discussing some special incidents related to the “parallel execution” control structure, we want to highlight a third flaw in the design of the experiment: For each control structure, the time was measured that was needed by the students to study the code and to answer a question about the value of a variable. From the text in [10], there is no information about the correctness of the given answers. If we take into account that it is rather unlikely that there was no wrong answer at all, it seems as no difference has been made between the time needed to give a wrong answer and the time needed to give a right one. However, this is obviously a very important difference in an experiment that is related to code comprehension[6].
4 Understanding Parallel Computing

The code fragment that has been used for testing the understanding of parallel execution has been quoted above. Instead of using the class `Thread` that is included in the language Java for dealing with parallel execution, the code example relies on comment lines in natural language for expressing the fact that two methods are executed in parallel. However, unfortunately, this example code does in no way express the difficulties that can arise from parallel execution and that can make it hard to understand parallel computing: The code for “class A” is completely independent from the code for “class B”, i.e. the both code fragments do not have to share any resources apart from reading from the “global” variables A and B. For this reason, it does not make a difference whether the method for class A and the method for class B are executed in a sequence or in parallel.

In fact, there are three types of possible parallel execution which are differently difficult to understand:

1. The two parallel threads do not write to shared variables and do not have to wait for each other. The example code in [10] deals with this type of parallel execution.

2. At least one of the parallel threads modifies a shared variable that is read by the other thread or both threads need to have access to a shared resource (like a database connection) that can be blocked and makes synchronisation between both threads necessary.

3. Not just one, but at least two shared variables or shared resources are involved. This means that synchronisation between both threads is necessary, with the danger of running into deadlocks.

The first type is easier to understand than the other ones, and the second type is still easier to comprehend than the third type.

For this reason, we think that using only a code example for the easiest of these cases cannot give an appropriate result on the difficulty for understanding parallel executions.

5 Additional Control Structures

If we want to design good tests on understandability of parallel executions, we see from the above that synchronisation between two threads should be taken into account. However, a mechanism for synchronisation between two threads or processes (in Java: synchronised blocks or Locks) is nothing else than an additional control structure that is not yet covered by [8] and its underlying theories[9].

When analysing the cognitive complexity of code that includes synchronisation, we have to add a new control structure with a name like “synchronisation” or “lock” to the list of basic control structures in Tab. 1.

On this occasion, we mention that Tab. 1 also does not yet include two other important control structures that can be found in various modern programming languages:

1. Exceptions

2. Internal exits from loops (for example expressed by the `break`-statement in C or Java). Note that some languages like Pascal do not allow such exits. However, there are good reasons for using this control structure under some circumstances[7].

6 Requirements for Experimental Settings

From the points stated in the previous sections, we come to the conclusion that there are not yet any reliable experimental results that allow us to define the cognitive weights for software control structures.

For finding cognitive weights that can be used in a cognitive metric in an industrial environment, the following factors should be considered in future experiments:

1. Care should be taken that only the effort needed to understand the control structures is measured. In particular, this means that the length of code to read or the number and naming of variables should not differ between the code examples.

2. As the cognitive complexity measure is language-independent, experiments should be made using more than one programming language.

3. Experiments performed with students might be affected by their lack of experience. For this reason it would be desirable to repeat such experiments with experienced software developers.

4. It is widely agreed that replication is necessary for this kind of experiments. One conclusion from this requirement is that the experimental setup must be described carefully.

5. The established standards for empirical research in the field of software engineering in general[4, 3] and program comprehension in particular[6] should be followed.

\(^{1}\)McQuaid[5] has assigned an “inherent complexity” of 4 to `select` and `accept`-statements that are used for synchronisation in Ada95. We have omitted this in Tab. 1, because there are no corresponding complexities defined in the other discussed papers.
7 Conclusions

Past experiments that aim to find reasonable values for the cognitive weights of software control structures are not yet satisfying. For this reason, experiments that avoid the problems highlighted in this paper will be necessary. We believe that cognitive weights for basic control structures can be a useful measure for the difficulty to understand software. Moreover, we think that the idea of cognitive weights is even more important for assessing the difficulty to understand graphical models that depict the control-flow of software or business processes[2]. However, before this idea can be used in an industrial context, there is still a lot of work to do for calibrating the cognitive weights for the control structures. Furthermore, we need experiments that deal with the question whether the cognitive complexity measure is a better indicator for the difficulty to comprehend a piece of software than simpler measures like “Lines of Code”. As a next step, it could be an interesting research question to expand the research on cognitive weights from imperative programming languages to other paradigms like logic programming or aspect-oriented programming.

References