Queue-based Cost Evaluation of Mental Simulation Process in Program Comprehension

Masahide Nakamura, Akito Monden, Tomoaki Itoh*,
Ken-ichi Matsumoto, Yuichiro Kanzaki, Hirotsugu Satoh

Graduate School of Information Science, Nara Institute of Science and Technology, JAPAN
{masa-n, akito-m, matumoto, yuichi-k, hirots-s}@is.aist-nara.ac.jp

**Matsushita Electric Industrial Co.,Ltd., JAPAN*
itoh.tomoaki@jp.panasonic.com

Abstract

This paper presents a method to estimate the cost of mental (hand) simulation of programs. In mental simulation, human short-term memory is extensively used to recall and memorize values of variables. When the simulation reaches a variable reference, the simulation can be performed easily if the value is still remembered. However, if not, we have to backtrack the simulation until the value is obtained, which is time-consuming.

Taking the above observation into consideration, we first present a model, called virtual mental simulation model (VMSM), which exploits a queue representing short-term memory. The VMSM takes one of the abstract processes `recall` *or* `backtrack`*, depending on whether the variable is currently stored in the queue or not. Then, applying cost functions to the VMSM, we derive four dynamic metrics reflecting the cost of mental simulation.*

In our empirical study, the proposed VMSM metrics reveal that the backtrack process for non-constant variables gives a significant impact on the cost of mental simulation. Since the proposed method can be fully automated, it can provide a practical means to estimate the cost of mental simulation, which can be also used as a program comprehension measure.

1 Introduction

Mental simulation (also called *hand simulation*) of a program is a quite primary but effective activity to understand how the program works [11]. In mental simulation, a person (programmer, maintainer or hacker, etc...) executes the program in mind instead of computers. To make our discussion clearer, we first give a definition adopted in this paper.

> *Mental simulation of a given program p with input I is a human activity such that: based on the source code of p and I, the person simulates execution of p in his/her mind as accurately as computers do.*

Mental simulation is widely used in various situations. It is typically used to locate faults in debugging, or to understand the existing program before adding a new feature onto it [1]. Program hacking is also based on mental simulation.

The goal of this paper is to propose a method that can be used to estimate the *cost* of mental simulation. The cost of mental simulation reflects important aspects of the program. If the cost is low, it is easy to maintain the program. On the other hand, if extremely expensive, it would be hard to analyze the program, which is a good characteristic from a viewpoint of *program protection* [6][13].

Mental simulation can be counted as a means for *program comprehension*. There has been a number of hypotheses, methodologies and empirical reports about program comprehension measure (some are discussed in Section 5). However, the notion of comprehension itself is so generic and qualitative. Therefore, most comprehension measures tend to be hypothetic or domain-specific, and often require expensive parameters tuning or input factors that are hard to be quantified. Also, due to dynamic nature of mental simulation, we found it difficult to directly apply the conventional *program complexity metrics* [7] to the cost estimation.

In this paper, by narrowing our scope in the cost of mental simulation only, we try to develop a more generic and easy-to-use method that *partially* characterizes an aspect of program comprehension.

Proceedings of the Ninth International Software Metrics Symposium (METRICS'03)
1530-1435/03 $17.00 © 2003 **IEEE**

A key factor of mental simulation lies in human *short-term memory*. When simulating a statement with some variables, we extensively use short-term memory to recall and memorize values of the variables. However, since human short-term memory is quite volatile, we cannot always recall the current values successfully.

Let us consider this from a viewpoint of the cost of mental simulation. When the simulation reaches a reference of a variable, if the current value of the variable is still remembered, then the cost is low since the value is recalled fast and easily. While, if the value is forgotten, we have to *backtrack* the simulation until the value is obtained, which is generally time-consuming and thus would yield an expensive cost.

Based upon the above idea, we develop a virtual model, called *virtual mental simulation model* (VMSM, in short). The VMSM exploits a queue modeling short-term memory, to which variables and their associated values are dynamically inserted. When a reference of a variable occurs, the VMSM executes an abstract process `recall` or `backtrack`, depending on whether the variable is in the queue or not. Then, by assigning cost functions to the VMSM, we propose four dynamic metrics that characterize the cost of mental simulation.

We have evaluated the proposed method through an empirical study. As a result, the proposed VMSM metrics reveal that the backtrack process for non-constant variables significantly influences the cost of mental simulation. The proposed method can be fully automated, and it does not require expensive parameters tuning. Therefore, it provides an easy and practical means to estimate the cost of mental simulation, which characterizes a part of efforts for program comprehension.

The rest of this paper is organized as follows: In Section 2, we conduct a preliminary experiment to address the problem. Section 3 presents the VMSM and new metrics for the cost estimation. In Section 4, we evaluate the proposed method through an experiment. We review the related work in Section 5. Finally, Section 6 concludes the paper with discussion and future work.

2 Preliminary experiment

2.1 Example programs

Firstly, we chose a simple Java program A, which finds and outputs the maximum element in a given 3-dimensional integral array. Borrowing an obfuscation technique [13], we have then prepared two different versions A1 and A2 of the program A, as shown in Figure 1 (see the last page). Note that A1 and A2 yield the same execution result since both satisfy the same specification.

Our concern here is to measure how much cost (effort) is needed to perform mental simulation for each of A1 and A2. In order to measure the cost taken purely for mental simulation, A1 and A2 are prepared based on the following careful considerations.

- The same set of Java instructions is used in both A1 and A2.
- The cumulative number of statements actually executed is almost the same in both A1 and A2.
- Neither A1 nor A2 has comment lines. Also, identifiers do not have special meanings. These are to avoid influences of *meaningfulness* [5].
- A1 and A2 contain several (partitioned) loops with relatively small number of iterations. This is to exclude the effect of *loop induction* by causal reasoning [1].

2.2 Setting and Instruction

8 subjects participated in the preliminary experiment. All of them had been learning the Java programming language, and had experienced coding of mid-scale programs. Either program A1 or A2 was assigned randomly to each subject, and a printed source code was given. No additional information about the programs, such as specification and usage, was provided.

The instruction of the experiment is described as follows. In the experiment, the subjects conduct a paper-based execution of the given program. They are allowed to take notes on the sheet, but without computer assistance. In order to examine the cost for *accurate* mental simulation, the subjects are required to describe a *program state* (i.e., values of all variables in the program) at every time a variable `p` is updated (`p` stores a return value of `func`. See Figure 1).

Once each subject finishes the simulation, we check the sequence of the program states. If the sequence is correct, the simulation is completed. Otherwise, we say "incorrect", and point out only the last program state that is correct. Then, the subject repeats the simulation until he/she reaches the correct answer. Although we do not set deadline for the simulation task, the subjects are requested to conduct the simulation as fast as possible.

We measured the time spent for each subject to complete the simulation. Also, we counted the number of failures of the simulation.

2.3 Observation

Table 1 summarizes the result. According to the setting of the experiment, it can be said that the measured time reflects the cost for accurate mental simulation. The result shows that the mean time spent to simulate A2 is three times as much as that of A1. Roughly speaking, reading program

Table 1. Results of preliminary experiment

Program	Mean time (sec)	Ave. # of failures
A1	1213	1
A2	3542	3

A2 would be three times harder than reading A1. Note that only different portions between A1 and A2 are bodies of method `func`.

Much works have been done over decades to measure program complexity and comprehension. However, as far as we know, there is no metric that specifically focuses the cost of mental simulation only (see Section 5). Table 2 shows the well-known metrics [7] for method `func` in A1 and A2 [1].

In Table 2, the values for A1 are larger than (or almost equal to) those for A2. This fact implies that A1 is harder to be understood than A2, which is completely against our expectation. Thus, it is difficult to directly apply these metrics to cost evaluation for mental simulation, since they are unable to justify our result.

Empirical evaluation in [13] shows that the obfuscation applied to getting A2 is generally more difficult than the one applied to A1. However, there is no quantitative consequence for this. So, we need to develop alternative metrics.

3 Queue-based model for mental simulation

3.1 Human memory and cognitive activities

When performing mental simulation, we extensively utilize human memory, specifically, *short-term memory*. For example, let us consider the following statement:

```
a = b + 1;
```

To simulate the above statement, we must perform the following steps:

Step1: Recall the value of `b`.

Step2: Add 1 to the value of `b`.

Step3: Memorize the sum as a new value of `a`.

Performance of the above simulation heavily depends on whether we remember the value of `b` or not (at Step1). If the value of `b` is still *cached* in short-term memory, the simulation is easily performed. However, if not, we have to *backtrack* the simulation, until the value of `b` is obtained. In this case, the cost will become much more expensive. Note that we do not need to recall the value of `a` in the simulation, since `a` is updated to a new value regardless of its previous value.

More generally, the following cognitive activities are essentially involved in mental simulation.

CA1: When a *reference* of a variable `x` is reached, we try to recall the value of `x`. If the value is not in short-term memory, a *backtrack* of the simulation occurs to get the value. The value of `x` is stored (or refreshed) in short-term memory.

CA2: When an *assignment* to a variable `y` is reached, the calculated right-hand value is stored in short-term memory as a new value of `y`.

CA3: Values of variables in short-term memory are vanished in course of the simulation, due to the memory capacity or time passing.

In the above activities, we explicitly distinguish the assignment from the reference. We say that an assignment to a variable occurs only when the variable appears in the left-hand of an assignment statement. On the other hand, a reference occurs only when the variable appears in an expression, an index of array, or a parameter (not in a declaration). We assume that every appearance of a variable is exactly one of either reference or assignment.

We consider that a certain *model* [12] involving these cognitive activities is inevitable for the cost estimation of mental simulation. Of course, there are other factors that might influence human memory in mental simulation, such as programming style [15] (naming conventions, comments, indentations, spacing, etc.) and tool supports [14]. However, these contains plenty of human factors, which are quite difficult to be quantified. To keep our model as simple and easy-to-use as possible, we take only CA1-CA3 into consideration.

3.2 Queue representing short-term memory

In order to build a model involving CA1-CA3, we first exploit a *queue* which simplifies human short-term memory. The proposed queue, called *mental simulation queue* (MS-queue, in short), holds a set of variables and their associated values that are temporally stored in the short-term memory at a certain instant during mental simulation. The definition of the MS-queue is described below.

An *MS-element* $e = (v, val(v))$ is defined as a pair of a variable v and its associated value $val(v)$, or an empty element $e = \epsilon$. Then, an *MS-queue* q is an FIFO queue storing ordered MS-elements as its contents. A *length* of q, denoted by $len(q)$, is the number of non-empty MS-elements currently stored in q. A *maximum length*, denoted by L, is a capacity of q such that q can store at most L elements, simultaneously. The first (or last) element of q is referred as

[1] We did not include OO metrics in Table 2, since method `func` does not use object-oriented features of Java, specifically.

Table 2. Conventional metrics for `func` in programs A1 and A2

Program	LOC	Max Nest	# of Statements	# of Variables	# of Operators	McCabe CYCL	Halstead Length	Halstead Volume	Knots	Ave. Live Variables	Ave. Span Size
A1	73	7	47	8	67	22	368	1730	6	6.894	3.656
A2	40	5	29	9	56	15	290	1394	4	6.379	2.925

$head(q)$ (or $tail(q)$, respectively). Next, we define *operations* to the MS-queue.

dequeue(q): Remove $head(q)$. Then, the next element becomes $head(q)$.

enqueue(q,e): Insert an element e as the tail of q when $len(q) < L$. If $len(q) = L$, execute $dequeue(q)$ first, then insert e. Finally, e becomes $last(q)$.

is_queue(q,e): Return *true* if an element e exists in q. Otherwise, return *false*.

refresh(q,e): If $is\_queue(q, e)$ is true, delete e from the queue. Then, execute $enqueue(q, e)$.

We regard an MS-queue q as a simple but intuitive model of human short-term memory. Each MS-element e represents a *chunk*, which is the information unit stored in short-term memory. $len(q)$ represents *the number of chunks* currently memorized, and L represents a *capacity* of short-term memory. The operation $dequeue(q)$ models to forget the oldest chunk. $enqueue(q, e)$ corresponds to that the information e is memorized as the newest chunk. $is\_queue(q, e)$ returns a *state* whether e is remembered or not. $refresh(q, e)$ simulates a situation that a chunk e that is still remembered is refreshed.

3.3 Execution trace for mental simulation

In addition to modeling short-term memory itself, we need to know how information is incoming to short-term memory. In mental simulation, we try to execute the program as exactly as the computer does. Hence, it is reasonable to use the *program trace* [2] to characterize the information flow. For this purpose, we introduce a specific trace, called *AR-trace*, which focuses assignments and references of variables.

For each appearance of a variable v in a given program, an *AR-action* is defined as a triplet $(v, val(v), type(v))$, where $val(v)$ is a (current) value of v, and $type(v)$ is either reference or assignment. For a program p and a given input I, an *AR-trace* is a sequence of AR-actions occurring in accordance with execution of p with respect to I.

Figure 2 shows an example of (a) a program (fragment) p and (b) the corresponding AR-trace. Traversing p from the beginning to the end derives the AR-trace consisting of the ordered AR-actions. Note that the input I is not especially needed in this example, and that every AR-trace is uniquely determined for given p and I [2].

```
i = 1 ;
j = 2 ;
A[1] = i + 4 ;

j = A[i] - j ;
```

(a)

```
i      1   assignment
j      2   assignment
i      1   reference
A[1]   5   assignment
i      1   reference
A[1]   5   reference
j      2   reference
j      3   assignment
```

(b)

Figure 2. Example of AR trace

It is not very difficult to obtain the AR-trace automatically, from given program p and input I. Our idea is to embed a `print` statement as a *monitoring code* [2] immediately after each appearance of a variable, which is performed by a simple analysis of stack operations at the assembler code level. The monitoring code outputs an AR-action at run-time when execution reaches there. Thus, executing the modified p with respect to I outputs an AR-trace.

3.4 Virtual mental simulation model (VMSM)

Using an MS-queue q and an AR-trace ρ for given program p and input I, we *imitate* the process of mental simulation for p and I. Figure 3 shows the proposed *virtual mental simulation model* (VMSM). In the figure, let i be a variable storing an integer, and let $|\rho|$ be a length of ρ.

The proposed VMSM takes an AR-action one-by-one from the given AR-trace ρ. Depending on the type of the AR-action, one of sub-routines **Reference** or **Assignment** is executed.

The sub-routine **Reference** models the cognitive activity CA1 (See Section 3.1). It contains two *abstract procedures*: `recall(e,i)` and `backtrack(v,i)`. According to CA1, if the referred variable is still memorized (i.e., $is\_queue(q, e)$ is true), the value of the variable is recalled through a chunk e (denoted by `recall(e,i)`). Then, the memory for e is refreshed via $refresh(q, e)$. While, if the

[2] This is because our target here is *sequential programs*

Step1: Initialize q to be empty, and $i = 1$.

Step2: If $i > |\rho|$ go to Step 5.

Step3: For i-th AR-action $(v, val(v), type(v))$ of ρ, if $type(v)$ is reference, then go to **Reference**. If $type(v)$ is assignment, then go to **Assignment**.

Step4: $i = i + 1$. Go to Step2.

Step5: End mental simulation.

Reference: Let $e = (v, val(v))$. If $is\_queue(q, e)$ is:

- **true:** Execute `recall(e,i)`. Then, $refresh(q, e)$, and return.
- **false:** Execute `backtrack(v,i)`. Then, $enqueue(q, e)$, and return.

Assignment: Let $e = (v, val(v))$. Execute `calc_righthand(v,i)`. Then, $enqueue(q, e)$, and return.

Figure 3. Virtual mental simulation model (VMSM)

variable is forgotten, a backtrack of simulation to obtain the variable's value occurs (i.e., `backtrack(v,i)`). After getting the value, the information is newly memorized via $enqueue(q, e)$.

The sub-routine **Assignment** corresponds to the cognitive activity CA2. As seen in the example in Section 3.1, the assignment does not require to recall the value. Instead, we have to calculate right-hand value of the assignment statement. The values of the all right-hand variables must have been obtained through the previous reference AR-actions. Hence, we purely apply operators to the operands, which is abstracted by `calc_righthand(v,i)`. The calculated value is newly memorized via $enqueue(q, e)$.

The details of the abstract procedures (`recall`, `backtrack` and `calc_righthand`) are not specifically given here. In order to achieve our goal, it is sufficient to have a cost calculation method for each of them, which will be discussed in the next subsection.

Note that, as the virtual simulation proceeds, the older MS-elements are dequeued due to the limited capacity L of q. This reflects the cognitive activity CA3.

Figure 4 shows how the VMSM ($L = 2$) works for the AR-trace in Figure 2(b). In the figure, a box represents an MS-element, a pair of parallel lines depicts an MS-queue, and an arrow depicts a transition caused by an AR- action.

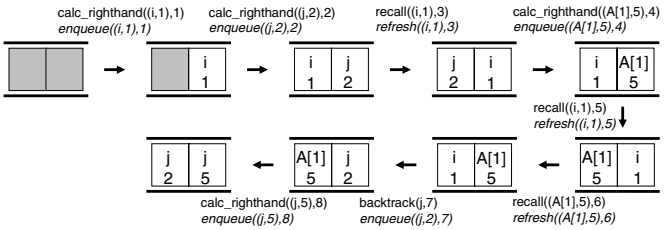

Figure 4. Example of VMSM execution

We assume that an MS-element is enqueued from right and dequeued to left.

3.5 Cost functions for VMSM

In order to calculate the cost for mental simulation, we assign a weighted cost function to each abstract procedure in the VMSM. Note that there are three abstract procedures `recall`, `backtrack` and `calc_righthand`. We consider that these are the dominant factors that influence the simulation cost.

Cost for `recall`

`recall(e,i)` involves a cognitive activity to recall the value of a variable memorized in short-term memory. Since the information is still remembered as a chunk e, the value can be obtained relatively easily and fast. So, the cost taken for this is inexpensive (compared with the cost for backtrack).

We suppose that the same amount cost is taken for each `recall(e,i)`, regardless of the position of e in the MS-queue q. This comes from our definition of the MS-element, stating that all chunks (MS-elements) have a isomorphic structure $(v, val(v))$ [3].

Thus, for each execution of `recall(e,i)`, a constant value is accumulated as the cost. For simplicity, we define a dynamic metric **RCL** as the number of `recall(e,i)` executed through a VMSM run.

Cost for `backtrack`

`backtrack(v,i)` contains a backtrack of simulation from i-th AR-action in order to obtain the current value of forgotten variable v. It can be considered that the cost heavily depends on whether the variable is a *constant variable* or not.

The constant variable is a variable to which a value is assigned only once (initialized) during entire execution. Since the initialized value never changes, we just *jump back* to

[3] There is no hierarchy or dependency between chunks in our model.

the initialization point to obtain the value in the backtrack. The cost for this process is independent of the simulation history. So, we suppose that a constant value is accumulated as the cost in this case. For simplicity, we define a metric **BT_CONST** as the number of `backtrack(v,i)` with constant variable v executed through a VMSM run.

If v is an ordinary (non-constant) variable, how would the backtrack be performed, and how should the cost be calculated? Here we define three kinds of *backtrack criteria*.

Constant: Backtrack the simulation to a certain (fixed) point, regardless of the simulation history. A constant value is accumulated as the cost for each execution of `backtrack(v,i)`.

Latest Reference: Backtrack the simulation to the most recent reference of v. The cost increases along with the *distance*, from the current appearance of v to the previous appearance in which v is referred.

Latest Assignment: Backtrack the simulation to the most recent assignment of v. The cost increases along with the distance to the previous assignment to v.

We characterize the above distance as an *interval* between two AR-actions on the AR-trace ρ. Let $a_i = (v, c_i, reference)$ be i-th AR-action currently processed. For i, we define *the latest referred index* $LR(i)$ and *latest assigned index* $LA(i)$ as follows. $LR(i)$ is an integer $j(< i)$ such that j-th AR-action $a_j = (v, c_j, reference)$ exists and that there is no $(v, c_k, reference)$ between a_i and a_j. Similarly, $LA(i)$ is an integer $j(< i)$ such that j-th AR-action $a_j = (v, c_j, assignment)$ exists and that there is no $(v, c_k, assignment)$ between a_i and a_j. Then, the distance to the latest reference (or assignment) can be defined by $i - LR(i)$ (or $i - LA(i)$, respectively).

For example, let us consider 5th AR-action `(i, 1, reference)` in Figure 2(b). Since the latest reference of `i` occurs in 3rd AR-action, $LR(5) = 3$. So, Latest Reference criterion says that we have to backtrack 2 $(= 5 - LR(5))$ steps to obtain the value of `i`. Similarly, since the latest assignment of `i` occurs in 1st AR-action, $LA(5) = 1$. Latest Assignment criterion forces to backtrack 4 $(= 5 - LA(5))$ steps.

After all, the cost for each `backtrack(v,i)` where v is a non-constant variable is defined as follows:

$$\begin{array}{lll} 1 & \cdots & \text{(if Constant is applied)} \\ i - LR(i) & \cdots & \text{(if Latest Reference is applied)} \\ i - LA(i) & \cdots & \text{(if Latest Assignment is applied)} \end{array}$$

For a VMSM run with a (given) criterion, we define a metric **BT_VAR** as the accumulated cost for `backtrack(v,i)` where v is a non-constant variable.

Cost for `calc_righthand(v,i)`

As mentioned in Section 3.4, `calc_righthand(v,i)` is to obtain the right-hand value of the assignment statement `v` = E, where `v` is a variable currently processed in i-th AR-action, and E is the right-hand expression. References of all variables in E must have been processed in the previous AR-actions. Hence, `calc_righthand(v,i)` is devoted purely to applying operators to the operands. So, for each `calc_righthand(v,i)`, where `v` = E, we define the cost by the number of operators in E + 1 (for the assignment operator itself).

For a VMSM run, we define a metric **ASSIGN** by the accumulated cost for `calc_righthand(v,i)`.

3.6 Calculating dynamic metrics with VMSM

Now, we present a procedure of cost calculation using the VMSM with the cost functions.

VMSM cost calculation procedure

Input: a program p, input I, maximum queue length L, and a backtrack criterion c.

Output: Four dynamic metrics

- **ASSIGN:** The accumulated cost for `calc_right hand` for assignment statements.
- **RCL:** The total number of `recall` executed.
- **BT_CONST:** The total number of `backtrack` executed with constant variables.
- **BT_VAR:** The accumulated cost for `backtrack` with non-constant variables, according to c.

Procedure: Obtain AR-trace ρ from p and I. Then, create an MS-queue q with length L. For q and ρ, run the VMSM. For each execution of `recall`, `backtrack` or `calc_righthand`, calculate and accumulate the cost, according to c and the corresponding cost function presented in Section 3.5.

4 Experimental evaluation

4.1 Preparation

In this experiment, we had 12 subjects. In addition to program A, two new programs B and C were introduced which are almost the same as A in size: **Program B** calculates sum of only positive (or negative) elements stored in an integral array. **Program C** counts the number of character 'X's in a char-type array. For program B (or C), we made two versions B1, B2 (or C1, C2, respectively), using the same technique as we had exploited for program A. The

Table 3. Results of experiment ($L = 3$)

Prgs	VMSM metrics							Empirical measures	
	ASSIGN	RCL	BT_CONS	BT_VAR			LEN_TR	Mean Time (Sec.)	Ave. # of Failures
				Constant	Latest Refer	Latest Assign			
A1	71	46	47	67	622	1609	231	1213	1
A2	72	47	36	81	1053	2631	236	3542	3
B1	93	95	55	64	788	1870	307	1294	0
B2	92	87	36	88	1338	2990	303	2955	3.5
C1	72	59	41	25	328	638	197	896	0.25
C2	71	52	22	48	652	1207	193	1689	1.25

six (= 3 prg. $\times 2$ ver.) programs were assigned to the subjects so that each subject had two different programs with different versions. As a result, each program was simulated by four subjects. We adopt basically the same setting and instruction as the ones in the preliminary experiment, presented in Section 2.

We have implemented two programs: `addtracer` is to embed the monitoring code to a given Java code for derivation of AR-trace (329 LOC, C). While, `vmsm-cost.pl` is a script (361 LOC, Perl) that computes the VMSM metrics for given AR-trace with L and c.

4.2 Setting VMSM parameters

To run the VMSM, we need to decide input parameters L and c.

The maximum length L models the capacity of short-term memory (See Section 3.2). The capacity of short-term memory has been believed to be 7 ± 2 chunks, traditionally. However, in an interview to the subjects, they said at most only 2 or 3 variables at a time could be memorized during mental simulation. Also, recent research [8] states that it is less than 7 ± 2, and is around 4 along with other non-capacity-limited sources. Moreover, the *running-memory aspect* of mental simulation decreases the capacity as well. Taking these facts into account, we consider that the reasonable value of L is around 3.

The backtrack criterion c is an important factor for calculation of the metric BT_VAR. Though we measured BT_VAR with respect to all the criteria, we consider that **Latest Assignment** agrees best with the reality of the experiment. As mentioned, the subjects were instructed to write down a program state every time a (given) variable is updated (i.e., a new value is assigned). Hence, in the backtrack, the subjects could refer to the list of program states, to get the latest updated values.

4.3 Result

For each of the six programs, we got two empirical measures: elapsed time and the number of failures. Also, we obtained the VMSM metrics as well as the length of AR-trace (LEN_TR). Table 3 summarizes the result.

First, we make inter-version comparison in the same program (category). Let us take a look at the result for program A. As we have seen in Section 2, the empirical measures show that simulating A2 costs much more than simulating A1. This fact is well explained by the metric BT_VAR, no matter which criterion is applied. Intuitively speaking, simulating A2 requires much more effort to *salvage* forgotten values of (non-constant) variables through simulation backtrack.

One of the main reasons why BT_VAR(A2) is greater than BT_VAR(A1) is that: `N`, `M` and `L` are constant variables in A1, while they are not in A2 (See Figure 1). This implies the fact that: when the current values of these variables are forgotten in the simulation, we can obtain the values through backtrack much more easily in A1, since the values never change during entire execution of A1. This is also characterized by BT_CONST.

Similar tendencies have been seen also for programs B and C. We did not get any interesting observation for ASSIGN, RCL and LEN_TR in this comparative evaluation. We conduct further investigation on BT_VAR in the next subsection.

4.4 BT_VAR as cost metric for mental simulation

It is, in general, difficult to reason simulation costs for different independent programs by a single metric. For example, there is no single metric by which we can completely explain why simulating C2 took more time (thus, more cost) than simulating B1 in Table 3.

Interestingly however, the metic BT_VAR had a quite high correlation to empirical cost measures in the experiment. Figure 5 shows BT_VAR (Latest Assignment, $L = 3$) and mean time taken for mental simulation of each of the six programs. In the figure, a solid bar represents a value of BT_VAR for each program, while a plot with a thin line depicts the mean time. The correlation factors between mean time and BT_VAR ($L = 3$) with respect to Constant, Latest Reference, or Latest Assignment are 0.768, 0.854, or 0.854,

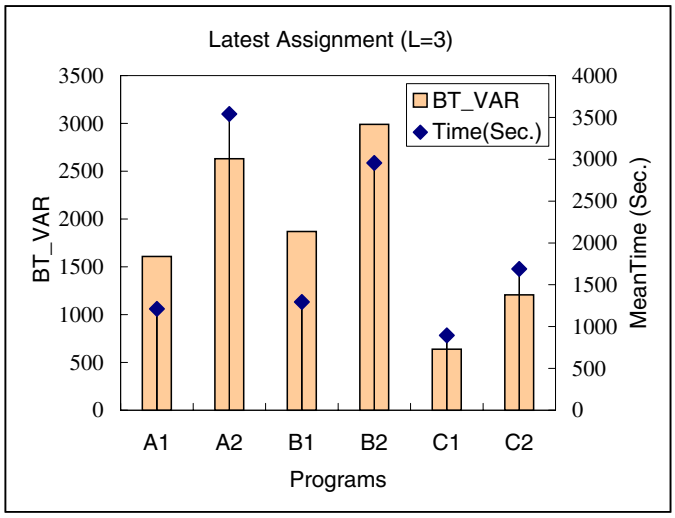

Figure 5. BT_VAR and mean time

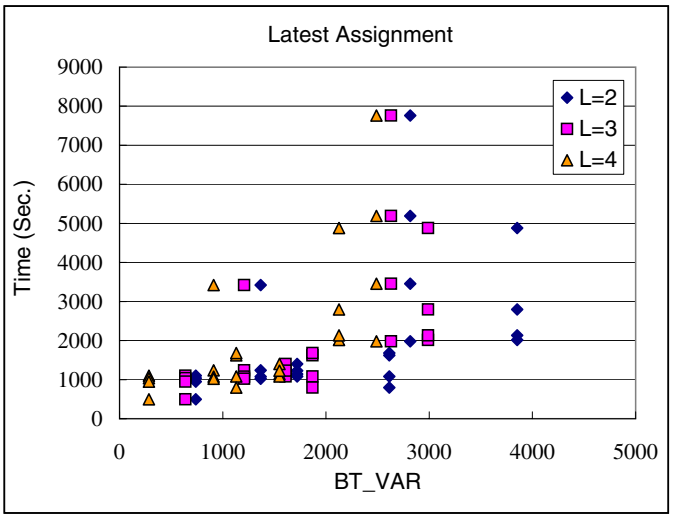

Figure 6. Measured time and BT_VAR

respectively.

Even when varying $L = 2, 3, 4$, BT_VAR kept high correlation with measured time. Figure 6 shows scattered plots, where the horizontal axis takes BT_VAR, and the horizontal axis draws the time taken for each subject. It can be seen that BT_VAR increases as L decreases. This is because: the smaller the capacity of short-term memory is, the more frequently we have to backtrack the simulation. Although BT_VAR varies with L, we can see a sufficient correlation between BT_VAR and the time for each fixed L.

After all, among the VMSM metrics, BT_VAR is shown to be quite essential to capture the simulation cost. Therefore, it can be utilized as a powerful metric for cost measurement of mental simulation.

5 Related works

The well-known complexity metrics focusing the variable usage are *live variables* and *span* [7]. These metrics measure only *static* aspects of variables without evaluating branch and loop conditions. By the dynamic nature of mental simulation, these metrics do not have strong relevance to the cost of mental simulation, as seen in Table 2.

Davis [9] presented a complexity metric with information chunks. Also, Cant et al. [4] proposed a cognitive complexity model (CCM) based on chunks. These metrics assume that a set of code blocks corresponding to basic chunks is available for a given program, which does not always hold for general programs. Burnstein et al. [3] presented a tool to identify the candidate basic chunks based on heuristics. In these methods, the complexity is accumulated in a *static* basis on the code blocks and their control/data relations. They do not especially concern dynamic aspects of the program as the proposed method does. Hence, we consider that these metrics are relevant to *causal reasoning*[1] (rather than mental simulation) by which the subject acquires a high level interpretation for the code blocks. Thus, they are complementary approaches to measure different comprehension aspects from ours.

In [11], mental simulation was shown to be the most reliable measure for program comprehension for novice programmers. However, the measure was based on the scores of questionnaires, but not derived by metrics.

6 Discussion and concluding remarks

In this paper, we have presented a new method to measure the cost of mental simulation, with a virtual model VMSM and new dynamic metrics. The key of the cost calculation with the VMSM is to accumulate different costs, depending on whether current values of variables are memorized in short-term memory modeled by a queue, or not. Through empirical evaluation, the VMSM metric BT_VAR revealed that the simulation backtrack is a dominant factor for the cost of mental simulation.

Since mental simulation is a fundamental technique to understand a program, the VMSM metrics (especially BT_VAR) have a wide range of application, such as maintenance cost estimation, testing cost estimation and temper proofing. The proposed method can be *fully automated* if input of the target program is given. This is a great advantage compared to the other related methods.

A limitation is that the proposed method cannot be applied to programs under construction. Since the VMSM re-

quires an execution trace of the target program, the program must be compilable and executable. Moreover, we do not count the *tracing cost* [4] for functions and control flows (i.e., the cost of locating the next code to be executed). To keep our model simple, we do not also consider any hierarchical chunk constructed by reasoning [3], nor meaningfulness of identifiers [5] in mental simulation.

Finally, we summarize our future work. First, we will conduct further empirical studies to investigate impacts of the VMSM metrics ASSIGN, RCL, BT_CONST on the total simulation cost.

Another interesting topic is to develop new VMSM metrics for other than cost measurement. For example, BT_VAR tends to increase according to LEN_TR (length of AR-trace, see Table 3). This is reasonable from a viewpoint of cost, since longer simulation costs more expensive. However, from a viewpoint of *difficulty*, $\frac{BT\_VAR}{LEN\_TR}$ would give a more reasonable indicator.

Also, it is important to clarify how mental simulation works in the whole program comprehension process. We need to investigate relationships between the proposed method and other method related to reasoning. This is quite challenging and our long-term goal.

Acknowledgment

The authors wish to thank Dr. Takao Yamaguchi and Yuji Sato in Matsushita Electric Industrial Co.,Ltd., for the fruitful discussion. This work is partly supported by a Grant-in-Aid for COE (Center Of Excellence) Research of the Ministry of Education, Science, Sports and Culture, Japan.

References

[1] Bisant, D. B. and Groninger, L., "Cognitive Processes in Software Fault Detection: A Review and Synthesis", *International Journal of Human-Computer Interaction*, 5(2):189-206, 1993.

[2] Ball, T. and Larus, J. R., "Optimally Profiling and Tracing Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, pp. 1319-1360, July 1994.

[3] Burnstein, I., Roberson, K., Saner, F., Mizra, A. and Tubaishat, A., "A Role for Chunking and Fuzzy Reasoning in a Program Comprehension and Debuggin Tool", *Proc. of the 9th Int'l Conf. on Tools with Artificial Intelligence (ICTAI'97)*, pp.102-109, 1997.

[4] Cant, S., Jeffery, D.R. and Henderson-Sellers, B., "A conceptual model of cognitive complexity of elements of the programming process", *Information Software Technology*, Vol. 37, No. 7, 351-362, 1995.

[5] Chaudhary, B. D. and Sahasrabuddhe, H. V., "Meaningfulness as a Factor of Program Complexity", *Proceedings of the ACM 1980 annual conference*, pp.457-466, 1980.

[6] Collberg, C. and Thomborson, C., "Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection", *IEEE Transactions on Software Engineering*, Vol. 28, No.8, pp. 735-746, 2002.

[7] Conte, S. D., Dunsmore, H. E. and Shen V. Y., "Software Engineering Metrics and Models,", *The Benjamin/Cummings*, 1986.

[8] Cowan, N., "The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity", *Behavioral and Brain Sciences*, Vol.24, pp.87-185, 2001.

[9] Davis, J. S., "Chunks: A Basis for Complexity Measurement", *Information Processing & Management*, Vol. 20, No.1-2, pp.119-127, 1984.

[10] Douce, C., "Long Term Comprehension of Software Systems : A Methodology for Study", *13th Workshop of the Psychology of Programming Interest Group, Bournemouth UK*, pp.147-159, April 2001.

[11] Dunsmore, A. and Roper, M., "A Comparative Evaluation of Program Comprehension Measures", *The Journal of Systems and Software*, Vol.52, Issue 3, pp.121-129, June, 2000.

[12] Fenton, N., "Software Measurement: A Necessary Scientific Basis", *IEEE Trans. of Software Engineering*, Vol.20, No.3, pp.199-206, 1994.

[13] Monden, A., Takada, Y. and Torii, K., "Methods for Scrambling Programs Containing Loops", *Trans. of the Institute of Electronics, Information and Communication Engineers*, Vol. J80-D-I, No. 7, pp. 644-652, July 1997 (in Japanese).

[14] Storey, M.-A. D., Fracchia, F. D. and Muller H. A., "Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization", *Proc. of the 5 th IEEE International Workshop on Program Comprehension*, pp17-28, May, 1997.

[15] ——, "Java Programming Style Guidelines, Version 3.0", *Geotechnical Software Services*, http://geosoft.no/javastyle.html, Jan. 2002.

```
import java.io.*;
public class a1 {
    static int func(int A[][][], int N, int M, int L) {
        int i, j, k;
        int p;
        p = A[0][0][0];
        for(i = 0;i < N;i++) {
            for(j = 0;j < M;j++) {
                k = 0;
                if(k < L) {
                    for(;;) {
                        if(A[i][j][k] < p) {
                            p = A[i][j][k];
                        }
                        k++;
                        if(k >= L) {
                            break;
                        }
                    }
                }
                j++;
                if(j >= M) {
                    break;
                }
                for(k = 0;k < L;k++) {
                    if(A[i][j][k] < p) {
                        p = A[i][j][k];
                    }
                }
            }
            i++;
            if(i >= N) {
                break;
            }
            j = 0;
            if(j < M) {
                for(;;) {
                    for(k = 0;k < L;) {
                        if(A[i][j][k] >= p) {
                            k++;
                        }
                        else {
                            p = A[i][j][k];
                            k++;
                        }
                    }
                    j++;
                    if(j >= M) {
                        break;
                    }
                    k = 0;
                    if(k < L) {
                        if(A[i][j][k] < p) {
                            p = A[i][j][k];
                        }
                        for(;;) {
                            k++;
                            if(k >= L) {
                                break;
                            }
                            if(A[i][j][k] < p) {
                                p = A[i][j][k];
                            }
                        }
                    }
                    j++;
                    if(j >= M) {
                        break;
                    }
                }
            }
        }
        return p;
    }
    public static void main(String args[]) {
        int A[][][];
        int N, M, L;
        A = new int[3][3][3];
        A[0][0][0] = 97;
        A[0][0][1] = 48;
        A[0][1][0] = 52;
        A[0][1][1] = 71;
        A[1][0][0] = 17;
        A[1][0][1] = 64;
        A[1][1][0] = 11;
        A[1][1][1] = 32;
        A[2][0][0] = 20;
        A[2][0][1] = 22;
        A[2][1][0] = 48;
        A[2][1][1] = 86;
        N = 3;
        M = 2;
        L = 2;
        System.out.println("Result: "
                    + func(A, N, M, L));
    }
}
```

(a) Program A1

```
import java.io.*;
public class a2 {
    static int func(int A[][][], int N, int M, int L) {
        int i, j, k, l;
        int p;
        p = A[0][0][0];
        l = M;
        for(i = 0; i < N; i++){
            for(j = 0; j < M; j++){
                M--;
                for(k = 0; k < L; k++){
                    if(A[i][j][k] < p){
                        p = A[i][j][k];
                    }
                }
                if(j >= M) break;
                for(k = 0; k < L; k++){
                    if(A[i][M][k] < p){
                        p = A[i][M][k];
                    }
                }
            }
            N--;
            if(i >= N) break;
            for(; M < l; M++){
                for(k = 0; k < L; k++){
                    if(A[N][M][k] < p) {
                        p = A[N][M][k];
                    }
                }
                if(j <= 0) break;
                j--;
                for(k = 0; k < L; k++){
                    if(A[N][j][k]<p) {
                        p = A[N][j][k];
                    }
                }
            }
        }
        return p;
    }
    public static void main(String args[]) {
        int A[][][];
        int N, M, L;
        A = new int[3][3][3];
        A[0][0][0] = 97;
        A[0][0][1] = 48;
        A[0][1][0] = 52;
        A[0][1][1] = 71;
        A[1][0][0] = 17;
        A[1][0][1] = 64;
        A[1][1][0] = 11;
        A[1][1][1] = 32;
        A[2][0][0] = 20;
        A[2][0][1] = 22;
        A[2][1][0] = 48;
        A[2][1][1] = 86;
        N = 3;
        M = 2;
        L = 2;
        System.out.println("Result: "
                    + func(A, N, M, L));
    }
}
```

(b) Program A2

Figure 1. Example programs