Cognitive Complexity of Software and its Measurement

Yingxu Wang, *PhD, Prof., P.Eng, F.WIF, SMIEEE, MACM*

International Center for Cognitive Informatics (ICfCI)
Dept. of Electrical and Computer Engineering
University of Calgary
2500 University Drive, NW, Calgary, Alberta, Canada T2N 1N4
Tel: (403) 220 6141, Fax: (403) 282 6855
Email: *yingxu@ucalgary.ca*

Abstract

The estimation and measurement of functional complexity of software are an age-long problem in software engineering. The cognitive complexity of software presented in this paper is a new measurement for cross-platform analysis of complexities, sizes, and comprehension effort of software specifications and implementations in the phases of design, implementation, and maintenance in software engineering. This paper reveals that the cognitive complexity of software is a product of its architectural and operational complexities on the basis of deductive semantics and the abstract system theory. Ten fundamental basic control structures (BCS's) are elicited from software architectural and behavioral specifications and descriptions. The cognitive weights of those BCS's are derived and calibrated via a series of psychological experiments. Based on this work, the cognitive complexity of software systems can be rigorously and accurately measured and analyzed. Comparative case studies demonstrate that the cognitive complexity is highly distinguishable in software functional complexity and size measurement in software engineering.

Keywords: *Cognitive informatics, software engineering, BCS, cognitive weights, cognitive complexity, psychological experiments, calibrated weights, cognitive functional size.*

1. Introduction

One of the central problems in software engineering is the inherited complexity. The estimation and measurement of functional complexity of software systems are an age-long problem in software engineering. The cognitive complexity provides a new approach to understand and measure the functional complexity of software and the effort in software design and comprehension.

The symbolic complexity of software is conventionally adopted as a measure in term of lines of code (LOC) [5, 11], but the functional complexity of software is too abstract to be measured or even estimated by LOC. Because numerous attributes of software systems are highly dependent on the understanding and measurability of software cognitive complexity, it has to be formally treated and empirically studied based on cognitive informatics [12] and theoretical software engineering methodologies [9].

In addition to the measure of LOC, the concepts of *function point* [1] and MaCabe's cyclomatic complexity [6] are proposed for measuring software functional complexity. However, the former did not define clearly what the physical meaning of a unit function point is; while the latter only considered the internal loop architectures of software systems without taking account of the throughput of the system in terms of data objects, and many other important internal architectures such as sequences, branches, and embedded constructs are excluded [2, 3, 5].

The *cognitive complexity* of software systems presented in this paper is a measure for the functional complexity in both software design and comprehension, which consist of the architectural and operational complexities. The new approach perceives software functional complexity is a measure of cognitive and psychological complexity of software as a human intelligent artifact, which takes into account of both internal structures of software and the I/O data objects under processing [4, 10, 13].

In this paper, a generic mathematical model of programs is created and the cognitive weights of fundamental software structures are identified in Section 2. A series of psychological experiments are described in Section 3 to determine the weights of relative cognitive complexity for the basic control structures (BCS's) of software. The cognitive complexity of software systems is formally defined in Section 4 with comparative case studies in software engineering applications presented in Section 5.

Proc. 5th IEEE Int. Conf. on Cognitive Informatics (ICCI'06)
Y.Y. Yao, Z.Z. Shi, Y. Wang, and W. Kinsner (Eds.)
1-4244-0475-4/06/$20.00 ©2006 IEEE

2. Mathematical Model of Programs and Cognitive Weights of Software Structures

This section creates a generic mathematical model of programs. Based on the abstract program model, the roles of embedded process relations, particularly the BCS's, are discussed, and their relative cognitive weights are formally defined.

2.1 The Mathematical Model of Programs

Despite the rich depository of empirical knowledge on programming and software engineering, the theoretical model of programs is still unknown. This subsection presents an embedded relational model (ERM) for describing the nature of programs and software systems [17].

Definition 1. A *process* P is a composed listing and a logical combination of n meta statements p_i and p_j, $1 \le i < n$, $1 < j \le m = n+1$, according to certain composing relations r_{ij}, i.e.:

$$P = \mathop{R}_{i=1}^{n-1}(p_i\ r_{ij}\ p_j), j = i+1 \\ = (...(((p_1)\ r_{12}\ p_2)\ r_{23}\ p_3)\ ...\ r_{n-1,n}\ p_n) \quad (1)$$

where the big-R notation [14, 16] is adopted that describes the nature of processes as the building blocks of programs.

Definition 2. A *program* $\mathfrak{P}$ is a composition of a finite set of m processes according to the time-, event-, and interrupt-based process dispatching rules, i.e.:

$$\mathfrak{P} = \mathop{R}_{k=1}^{m}(@e_k \hookrightarrow P_k) \quad (2)$$

The above definitions indicate that a program is an *embedded relational algebraic* entity. A *statement* p in a program is an instantiation of a meta instruction of a programming language that executes a basic unit of coherent function and leads to a predictable behavior.

Theorem 1. The *embedded relational model (ERM)* states that a software system or a program $\mathfrak{P}$ is a set of complex embedded relational processes, in which all previous processes of a given process form the context of the current process, i.e.:

$$\mathfrak{P} = \mathop{R}_{k=1}^{m}(@e_k \hookrightarrow P_k) \\ = \mathop{R}_{k=1}^{m}[@e_k \hookrightarrow \mathop{R}_{i=1}^{n-1}(p_i(k)\ r_{ij}(k)\ p_j(k))], j = i+1 \quad (3)$$

The ERM model provides a unified mathematical treatment of programs, which reveals that a program is a finite and nonempty set of embedded binary relations between a current statement and all previous ones that formed the semantic context or environment of computing.

Corollary 1. A program $\mathfrak{P}$ is not context-free rather than context-sensitive, where the context-sensitive characteristics are carried by the set of data objects or variables in the semantic environment [8].

Theorem 2. The sum of the cognitive weights of all r_{ij}, $w(r_{ij})$, in the ERM model determines the operational complexity of a software system C_{op}, i.e.:

$$C_{op} = \sum_{i=1}^{n-1} w(r_{ij}), j = i+1 \quad (4)$$

In the ERM model, a meta process is a statement in a programming language that severs as a common and basic building block for a program, and a process relation is a algebraic composing rule between processes.

Definition 3. A *meta process* is the most basic and elementary processes in computing that cannot be broken up further.

Definition 4. The set of *meta processes* $\boldsymbol{P}$ encompasses 17 fundamental primitive operations in computing as follows:

$$\boldsymbol{P} = \{:=, \blacklozenge, \Rightarrow, \Leftarrow, \Lleftarrow, \gg, \ll, |\gg, |\ll, @, \triangleq, \\ \uparrow, \downarrow, !, \oslash, \boxtimes, \S\} \quad (5)$$

Definition 5. A *process relation* is a composing rule for constructing complex processes by using the meta processes.

Definition 6. The *process relations* $\boldsymbol{R}$ of RTPA are a set of 17 composing operations and rules to built larger architectural components and complex system behaviors using the meta processes of RTPA, i.e.:

$$\boldsymbol{R} = \{\rightarrow, \curvearrowright, |, |...|, R^*, R^+, R^i, \circlearrowleft, \rightarrowtail, \|, \oiint, |||, \\ »\!, \leftrightsquigarrow, \hookrightarrow_t, \hookrightarrow_e, \hookrightarrow_i\} \quad (6)$$

The definitions, syntaxes, and formal semantics of each of these meta processes and process relations may be referred to real-time process algebra (RTPA) [8, 9, 14].

A *complex process* can be derived from the meta-processes by the set of algebraic process relations. Therefore, a complex process may operate on both operands and processes.

2.2 The Role of BCS's in Software Cognitive Complexity

Definition 7. *Basic Control Structures* (*BCS's*) are a set of essential flow control mechanisms that are used for modeling logical architectures of software.

The most commonly identified BCS's in computing are known as the *sequential, branch, iterations, procedure call, recursion, parallel,* and *interrupt* structures [8, 10, 14]. BCS's are a subset of the most frequently used process relations $\boldsymbol{R}$, i.e.:

$$BCS = \{\rightarrow, |, |\ldots|, R^*, R^+, R^i, \circlearrowleft, \rightarrowtail, \| (\oint\!\!\oint), \lightning\} \subset \boldsymbol{R} \quad (7)$$

The ten BCS's as formally modeled in RTPA [14] are shown in Table 1. These BCS's provide essential compositional rules for programming. Based on them, complex computing functions and processes can be composed.

The formal semantics known as the *deductive semantics* of each of the above meta processes and process relations may be referred to [8]. Almost all computing system or human behaviors can be described and implemented by a series of complex processes based on the combinations of meta processes and their relations [8, 9, 14].

Since BCS's are the most highly reusable constructus in programming, according to Definition 4 and Theorem 2, the cognitive weights of the ten fundamental BCS's play a crucial role in determining the cognitive complexity of software.

2.3 The Cognitive Weights of Basic Software Structures

Definition 8. The *cognitive weight* of a software system is the extent of relative difficulty or effort spent in time for comprehending the function and semantics of the given program.

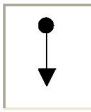
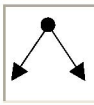
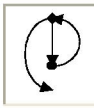
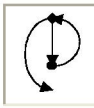
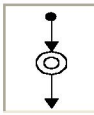
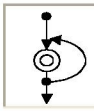
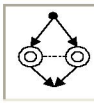
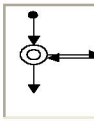
It is almost impossible to measure the cognitive weights of software at statement level since their variety and language dependency. However, it is found that it is feasible if the focus is put on the BCS's of software systems [10, 11], because there are only ten BCS's in programming no matter what kind of programming language is used. Detailed definitions of the ten BCS's and their RTPA syntax may be referred to [8, 14].

Definition 9. The *relative cognitive weight of a BCS*, $w_{BCS}(i)$, $1 \le i \le 10$, is the relative time or effort spent on comprehending the function and semantics of a BCS against that of the sequential BCS, i.e.:

$$w_{BCS}(i) = \frac{t_{BCS}(i)}{t_{BCS}(1)}, \quad 1 \le i \le 10 \quad (8)$$

where $t_{BCS}(1)$ is the relative time spent on the sequential BCS.

Table 1. BCS's and their Mathematical Models

Category	BCS	Structural model	Definition
Sequence	Sequence (SEQ)		P → Q
Branch	If-then-[else] (ITE)		(?exp**BL** = **T**) → P | (?~) → Q
	Case (CASE)		? exp**RT** = 0 → P_0 | 1 → P_1 | … | n-1 → P_{n-1} | else → ∅
Iteration	While-do (R^*)		$R^{F}_{\exp BL=T}$ (P)
	Repeat-until (R^+)		P → $R^{F}_{\exp BL=T}$ (P)
	For-do (R^i)		$R^{n}_{i=1}$ (P(i))
Embedded component	Procedure call (PC)		P ↳ Q
	Recursion ($R^{\circlearrowleft}$)		P ↻ P
Concurrence	Parallel (PAR)		P ‖ Q
	Interrupt (INT)		P ↯ Q

Definition 10. The *unit of cognitive weight of BCS's*, CWU, is the relative time spent on the *sequential* BCS, i.e.:

$$w_{BCS}(1) = \frac{t_{BCS}(1)}{t_{BCS}(1)} = 1 \quad [\text{CWU}] \quad (9)$$

3. Determining Cognitive Weights of BCS's by Psychological Experiments

The ten categories of BCS's described above are profound architectural attributes of software systems. Therefore, their relative cognitive weights is needed to be determined in an objective approach. A set of cognitive psychological experiments for testing the relative cognitive weight of a BCS is designed [7, 10], based on the axiom that the relative time spent on comprehending the function and semantics of a BCS is proportional to the relative cognitive weight of effort for the given BCS.

Although, absolutely, different persons may comprehend the set of ten BCS's in different speeds according to their experience and skill in a given programming language, the relative effort or the relative weight spent on each type of the BCS's are statistically stable assuming the relative weight of the *sequential* BCS is one according to Eqs. 8 and 9.

Definition 11. The generic psychological experimental method for establishing a benchmark of the cognitive weights of the ten BCS's can be conducted in the following steps:

a) Record the start time T_1 in mm:ss.
b) Read the given program Test_1:

```
int  Test_1 (int A=1, B=2) {
     return A + B
   }
```

c) Answer: C = ?
d) Record the end time T_2 in mm:ss.
e) Calculate the cognitive weights of the sequential construct, BCS_1, according to Eq. 8.

Applying the generic experimental method as given in Definition 11, the following cognitive psychological experiments can be carried out on the ten BCS's [7, 10].

3.1 The Sequential Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the sequential BCS is designed as shown in Experiment 1.

Experiment 1. The relative cognitive weight of the *sequential BCS* (*SEQ*), $w_{BCS}(1)$, is determined using the following program in Java:

```
int Test1 (int A=1, B=2) {
    return A + B;
}                                   (10)
```

The result will be used to calibrate the relative time and effort spent on the base unit of cognitive weight by a given individual.

3.2 The Branch Structure

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *if-then*-[*else*] BCS is designed as shown in Experiment 2.

Experiment 2. The relative cognitive weight of the *branch BCS* (*ITE*), $w_{BCS}(2)$, is determined using the following program in Java:

```
int Test2 (int A=2, B=3) {
        if A >= B
           return A - B
           else return B - A
}                                   (11)
```

3.3 The Switch Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *case-do* BCS is designed as shown in Experiment 3.

Experiment 3. The relative cognitive weight of the *switch* (*Case*) *BCS*, $w_{BCS}(3)$, is determined using the following program in Java:

```
int Test3 (int A=2, B=3) {
   switch (A) {
              case 1:  return B +1;
                       break;
              case 2:  return B +2;
                       break;
              …
              default: return B +10;
                       break;
            }
}                                   (12)
```

3.4 The For-Loop Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *for-do* BCS is designed as shown in Experiment 4.

Experiment 4. The relative cognitive weight of the *for-loop BCS* (R^i), $w_{BCS}(4)$, is determined using the following program in Java:

```
int Test4 (int A=2, B=3) {
    for (int i=0; i<3; i++) {
       A := A + i
        }
        return A + B
```

```
}                                                    (13)
```

3.5 The Repeat-Loop Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *repeat-until* BCS is designed as shown in Experiment 5.

Experiment 5. The relative cognitive weight of the *repeat-loop BCS* (R^+), $w_{BCS}(5)$, is determined using the following program in Java:

```
int Test5 (int A=1, B=2) {
    int i = 0;
    do {
        B := B + i
       } while (i++ = 2);
    return B - A
}                                                    (14)
```

3.6 The While-Loop Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *while-do* BCS is designed as shown in Experiment 6.

Experiment 6. The relative cognitive weight of the *while-loop BCS* (R^*), $w_{BCS}(6)$, is determined using the following program in Java:

```
int Test6 (int A=3, B=2) {
    int i = 0;
    while (i++ < 3) {
          B := B + i
          }
    return B - A
}                                                    (15)
```

3.7 The Function Call Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *function-call* BCS is designed as shown in Experiment 7.

Experiment 7. The relative cognitive weight of the *function call BCS* (*FC*), $w_{BCS}(7)$, is determined using the following program in Java:

```
Int Test0 (int A, B) {
    return A + B;
}

int Test7 (int A=2, B=3) {
    Test1 (A, B);
    return Test0;
}                                                    (16)
```

Note that only user defined functions/methods are considered in the determination of $w_{BCS}(7)$ on function calls. In other words, system functions are treated as meta statements.

3.8 The Recursion Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *recursion* BCS is designed as shown in Experiment 8.

Experiment 8. The relative cognitive weight of the *recursion BCS* (*REC*), $w_{BCS}(8)$, is determined using the following program in Java:

```
int x = 1;
…

int Test8 (int x) {
        int y = x+2;
        while (y <= 5) {
            Test8 (y);
        }
        return y;
}                                                    (17)
```

3.9 The Parallel Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *parallel* BCS is designed as shown in Experiment 9.

Experiment 9. The relative cognitive weight of the *parallel BCS* (*PAR*), $w_{BCS}(9)$, is determined using the following program in 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
}                                                    (18)

// 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
}                                                    (19)

// The convergence class
// Assume both parallel classes Test9A and Test9B
// have already been completed
pub int Test9 (int C, D) {
```

```
    if Test9A_completed && Test9B_completed
       return (C+D) * 2
       else return 0
}                                                    (20)
```

3.10 The Interrupt Construct

According to Definition 11, the cognitive psychological experiment on the cognitive weights of the *interrupt* BCS is designed as shown in Experiment 10.

Experiment 10. The relative cognitive weight of the *interrupt BCS* (*INT*), $w_{BCS}(10)$, is determined using the following program in Java:

```
// The interrupt capture class (simulated)
void InterruptCapture {
      …
      pub Boolean interruptEvent;
      interrupEvent := false;
      if interruptCaptured
         interruptEvent := true;
      …
}                                                    (21)

// The main class
int Test10 (int A=2, B=3) {
    pub boolean interruptEvent;
    pub int C;
    int i = 0;
    while (not interruptEvent) {
          i ++;
          if interruptEvent      // when i = 100
          interruptService (i, A);
          }
    return B + C
}                                                    (22)

// The interrupt service (simulated)
pub Int InterruptService (int x, y) {
     interruptEvent := false;
     C := (x+y) * 2;
}                                                    (23)
```

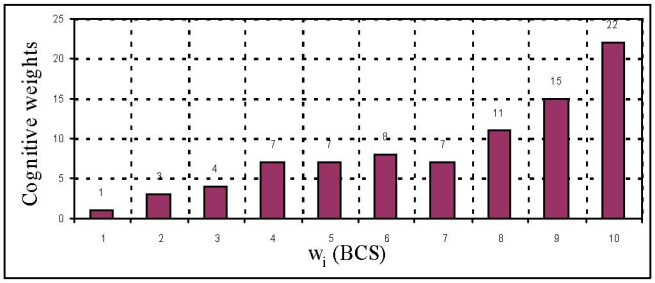
3.11 The Calibrated Cognitive Weights of BCS's

The cognitive psychological experiments as designed in the previous subsections have been carried out in undergraduate and graduate classes in software engineering. Based on 126 experiment results, the equivalent cognitive weights of the ten fundamental BCS's are statistically calibrated as summarized in Table 2 [10].

The calibrated cognitive weights for the ten fundamental BCS's are illustrated in Fig. 1, where the relative cognitive weight of the sequential structures is assumed one, i.e. $w_1 = 1$.

Table 2. Calibrated Cognitive Weights of BCS's

BCS	RTPA Notation	Description	Calibrated cognitive weight
1	$\rightarrow$	Sequence	1
2	$\mid$	Branch	3
3	$\mid\ldots\mid$	Switch	4
4	R^i	For-loop	7
5	R^*	Repeat-loop	7
6	R^*	While-loop	8
7	$\rightarrowtail$	Function call	7
8	$\circlearrowleft$	Recursion	11
9	$\parallel$ or $\oint\!\!\oint$	Parallel	15
10	$\lightning$	Interrupt	22



Note: 1 – sequence, 2 – branch, 3 – switch, 4 – for-loop, 5 – repeat-loop, 6 – while-loop, 7 – functional call, 8 – recursion, 9 – parallel, 10 - interrupt

Figure 1. The relative cognitive weights W of BCS's of software systems

4. Cognitive Complexities of Software Systems

As reviewed in [9, 11], the complexities of software system can be classified as the *symbolic, relational, architectural, operational, and functional* complexities. This section develops the mathematical model and measurement methodology of cognitive complexity of software systems on the basis of software semantic theories and semantic structures [8, 11]. A main notion is that the cognitive complexity of software can be treated as a product of its operational and structural complexities.

4.1 The Operational Complexity of Software Systems

Definition 12. The *operational complexity* of a software system *S*, $C_{op}(S)$, is determined by the sum of the cognitive weights of all relational operations $w(BCS)$ in each components of the system, i.e.:

$$C_{op}(S) = \sum_{k=1}^{n_C} C_{op}(C_k) = \sum_{k=1}^{n_C} \sum_{i=1}^{\#(C_s(C_k))} w(k,i) \quad [\mathrm{F}] \tag{24}$$

where $w(k, BCS)$ is given in Table 2.

Definition 13. The *unit of operational complexity* of software systems is a single sequential operation (*SeqOP*) in term of a single function F, i.e.:

$$C_{op}(S) = 1\ [\mathrm{F}] \Leftrightarrow \#(\mathrm{SeqOP}(S)) = 1 \tag{25}$$

With the cognitive weight of sequential process relation defined as a unit of operational function of software systems, complex process relations can be analyzed. The equivalent numbers of function units of the ten fundamental computing operations have been calibrated in Table 2.

It is noteworthy that for a fully sequential program where only w(sequence) = 1[F] is involved, the operational complexity is degenerated to the symbolic complexity known as the lines of code (*LOC*). In other words, the symbolic complexity, *LOC*, is a simplified measure of the operational complexity when the relational weights among all statements are roughly assumed as the same, i.e. $w_{ij} \equiv 1$.

4.2 The Architectural Complexity of Software Systems

The architectural complexity of software systems is proportional to its global and local data objects such as inputs, outputs, data structures, and internal variables.

Definition 14. The *architectural complexity* of a software system S, $C_a(S)$, is determined by the number of data objects at system and component levels, i.e.:

$$C_a(S) = \mathrm{OBJ}(S)) = \sum_{k=1}^{n_{CLM}} \mathrm{OBJ}(CLM_k) + \sum_{k=1}^{n_C} \mathrm{OBJ}(C_k) \quad [\mathrm{O}] \tag{26}$$

where *OBJ* is a function that counts the number of data objects in a given CLM (number of global variables) or components (number of local variables).

Definition 15. The *unit of architectural complexity* of software systems is a single data object, modeled either globally or locally, called an object O, i.e.:

$$C_a(S) = 1\ [O] \Leftrightarrow \#(\mathrm{OBJ}(S)) = 1 \tag{27}$$

For a high throughput system in which a large number of similar inputs and/or outputs are operated, the equivalent architectural complexity is treated as a constant *three* rather than infinitive [9]. This is determined on the basis of cognitive informatics where the inductive inference effort of a large or infinitive series of similar patterns is equivalent to three, typically the first and last items plus an arbitrary one in the middle. For instance, the equivalent number of the data object in the set {X[1]**N**, X[2]**N**, …, X[n]**N**} is counted as three rather than n.

Example 1. The architectural complexity of the *MaxFinder* algorithm as shown in Fig. 5 can be determined as follows:

$$\begin{aligned} C_a(MaxFider) &= \mathrm{OBJ}(MaxFider) \\ &= \#(\text{inputs}) + \#(\text{outputs}) + \#(\text{local variables}) \\ &= 3+1+3 \\ &= 7 \quad [\mathrm{O}] \end{aligned}$$

Example 2. The CLM SysClock**ST** given below encompasses 7 objects, therefore its architectural complexity is: C_a(SysClock**ST**) = 7 [O].

SysClock**ST** ≙ SysClock**S** ::
(<§t : **N** | 0 ≤ §t**N** ≤ 1M>,
<CurrentTime : **hh:mm:ss:ms** | 00:00:00:000 ≤ CurrentTime **hh:mm:ss:ms** ≤ 23:59:59:999>,
<Timer : **ss** | 0 ≤ Timer**ss** ≤ 3600>,
<MainClockPort : **B** | MainClockPort**B** = FFF0**H** >,
<ClockInterval : **N** | TimeInterval**N** = 1**ms**>,
<InterruptCounter : **N** | 0 ≤ InterruptCounter**N** ≤ 999>
)

4.3 The Cognitive Functional Complexity of Software Systems

It is empirically observed that the cognitive complexity of a software system is not only determined by its operational complexity, but also determined by its architectural complexity [11]. That is, software cognitive complexity is proportional to both its operational and architectural complexities. This leads to the formal description of the cognitive complexity of software systems.

Definition 16. The *semantic function* of a program $\wp$, $f(\wp)$, is a finite set of values V determined by a Cartesian product on a finite set of variables S and a finite set of executing steps T, i.e.:

$$f(\wp) = f: T \times S \to V = \begin{pmatrix} & \mathbf{s_1} & \mathbf{s_2} & \cdots & \mathbf{s_m} \\ \mathbf{t_0} & \perp & \perp & \cdots & \perp \\ \mathbf{t_1} & v_{11} & v_{12} & & v_{1m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{t_n} & v_{n1} & v_{n1} & \cdots & v_{nm} \end{pmatrix} \tag{28}$$

where $T = \{t_0, t_1, \ldots, t_n\}$, $S = \{s_1, s_2, \ldots, s_m\}$, and V is a set of values $v(t_i, s_j)$, $0 \le i \le n$, and $1 \le j \le m$.

The semantic space of a program can be illustrated by a two dimensional plane as shown in Fig. 2.

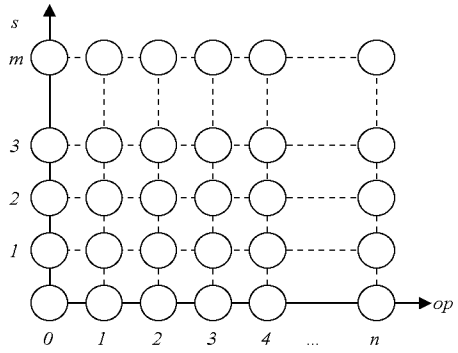


Figure 2. The semantic space of software systems

Observing Fig. 2 and Eq. 28, it can be seen that the complexity of a software system, or its semantic space, is determined not only by the number of operations, but also by the number of data objects under operation.

Theorem 3. The *cognitive complexity* $C_c(S)$ of a software system S is a product of the operational complexity $C_{op}(S)$ and the architectural complexity $C_a(S)$, i.e.:

$$\begin{aligned} C_c(S) &= C_{op}(S) \bullet C_a(S) \\ &= \{\sum_{k=1}^{n_C} \sum_{i=1}^{\#(C_s(C_k))} w(k,i)\} \bullet \\ &\quad \{\sum_{k=1}^{n_{CLM}} \mathrm{OBJ}(CLM_k) + \sum_{k=1}^{n_C} \mathrm{OBJ}(C_k)\} \quad [\mathrm{FO}] \end{aligned} \tag{29}$$

Based on Theorem 3, the following corollary can be derived.

Corollary 2. The cognitive complexity of a software system is proportional to both its the operational and structural complexities. That is, the more the architectural data objects and the higher the operational complicity onto these objects, the larger the functional complexity of the system.

Definition 17. The *unit of cognitive complexity* of a software system is a single sequential operation onto a single data object called a function-object *FO*, i.e.:

$$\begin{aligned} C_f &= C_{op} \bullet C_a \\ &= 1[\mathrm{F}] \bullet 1[\mathrm{O}] \\ &= 1 \quad [\mathrm{FO}] \end{aligned} \tag{30}$$

According to Definition 17, the physical meaning of software cognitive complexity is how many function-object [FO] are equivalent for a given software system. The cognitive complexity as a measure enables the accurate measurement of software functional sizes and design efforts in software engineering practice. It will be demonstrate in the next section that the cognitive complexity is the most distinguishable and accurate measure for the inherent functional complexity of software systems.

5. Comparative Case Studies on the Cognitive Complexity of Software Systems

This section comparatively analyzes the cognitive complexities of four sample software components. For self containment, all the RTPA specifications of the four cases are presented in Examples 3 through 6 below.

Example 3. The formal specification of the algorithm of In-Between Sum (IBS), *IBS_Algorithm***ST**, is specified in two different approaches as shown in Figs. 3 and 4, respectively.

```
IBS_AlgorithmST ({I:: AN, BN}; {O:: ⓈIBSResultBL, IBSumN}) ≜
{ // Specification (a)
    MaxN := 65535
    → ( ? (0 < AN < maxN) ∧ (0 < BN < maxN) ∧ (AN < BN)
            → IBSumN := ((BN - 1) * BN) / 2) - (AN * (AN + 1) / 2)
            → ⓈIBSResultBL := T
        | ? ~
            → ⓈIBSResultBL := F
            →! (@'AN and/or BN out of range, or AN ≥ BN')
        )
}
```

Figure 3. Formal description of the IBS algorithm (a)

```
IBS_AlgorithmST ({I:: AN, BN}; {O:: ⓈIBSResultBL, IBSumN}) ≜
{ // Specification (b)
    MaxN := 65535
    → ( ? (0 < AN < maxN) ∧ (0 < BN < maxN) ∧ (AN < BN)
→ IBSumN := 0
            → IBSumN := R_{iN=AN+1}^{BN-1} (IBSumN + iN)
            → ⓈIBSResultBL := T
        | ? ~
            → ⓈIBSResultBL := F
            →! (@'AN and/or BN out of range, or AN ≥ BN')
        )
}
```

Figure 4. Formal description of the IBS algorithm (b)

Example 4. An algorithm, *MaxFinder***ST**, is formally described in RTPA as shown in Fig. 5. Its function is to find the maximum number *max***N** from a set of *n* inputted integers {X[1]**N**, X[2]**N**, …, X[n]**N**}.

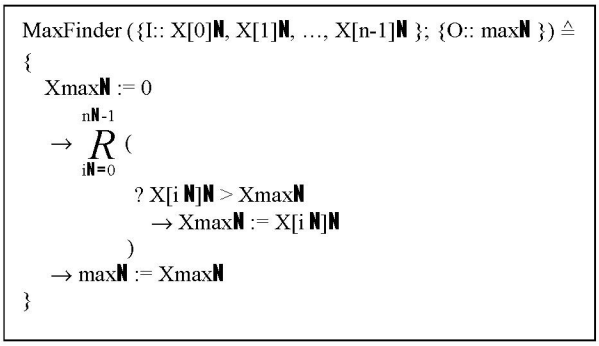


Figure 5. Formal description of the *MaxFinder* algorithm

Example 5. The *self-index sort* (*SIS*) algorithm, *SIS***ST**, can be formally described by RTPA as shown in Fig. 6. Detailed expalinations of the algorithm may be referred to [15].

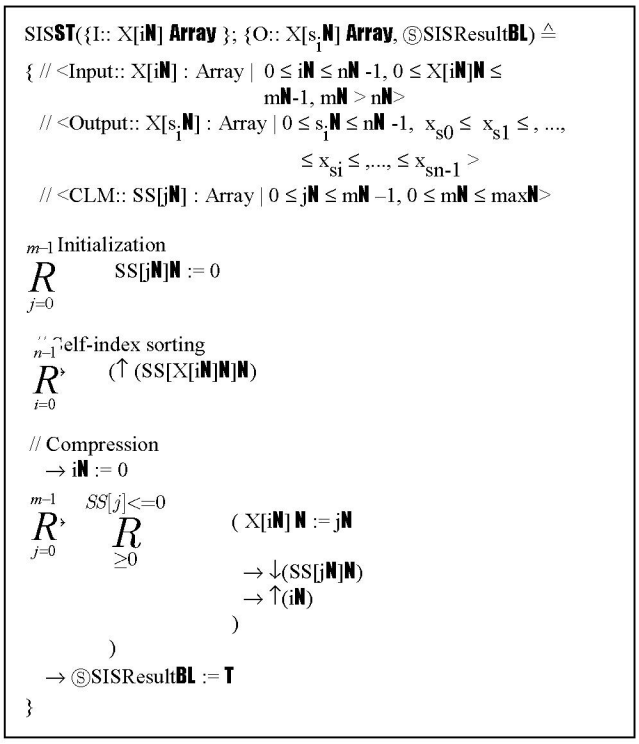


Figure 6. Formal description of the *SIS* algorithm

According to Eqs. 24, 26, and 29, the cognitive complexity of these four sample programs can be systematically analyzed as summarized in Table 3. In the table, the analyses results based on existing complexity measures, such as *time, cyclomatic,* and *symbolic* (LOC) complexities, are also contrasted.

Observe Table 3 it can be seen that the first three traditional measurements cannot actually reflect the real complexity of software systems in software design, representation, cognition, comprehension, and maintenance.

- Although four example systems are with similar symbolic complexities, their operational and functional complexities are greatly different. This indicates that the symbolic complexity cannot be used to represent the operational or functional complexity of software systems.

- The symbolic complexity (LOC) does not represent the throughput or the input size of problems.

- The time complexity does not work well for a system there is no loops and dominate operations, because in theory that all statements in linear structures are treated as zero in this measure no matter how long they are. In addition, time complexity can not distinguish the real complexities of systems with the same asymptotic function, such as in Case 2 (IBS (b)) and Case 3 (Maxfinder).

- The cognitive complexity is an ideal measure of software functional complexities and sizes, because it represents the real semantic complexity by integrating both the operational and architectural complexities in a coherent measure. For example, the difference between IBS(a) and IBS(b) can be successfully captured by the cognitive complexity. However, the symbolic and cyclomatic complexities cannot identify the functional differences very well.

Table 3. Measurement of Software System Complexities

System	Time complexity (C_t [OP])	Cyclomatic complexity (C_m [-])	Symbolic complexity (C_s [LOC])	Cognitive complexity		
				Operational complexity (C_{op} [F])	Architectural complexity (C_a [O])	Cognitive complexity (C_c [FO])
IBS (a)	ε	1	7	13	5	65
IBS (b)	O(n)	2	8	34	5	170
MaxFinder	O(n)	2	5	115	7	805
SIS_Sort	O(m+n)	5	8	163	11	1,793

6. Conclusions

This paper has presented a new approach to model and manipulate the cognitive complexity of software. This paper has revealed that the cognitive complexity of software is a product of its architectural and operational complexities on the basis of the formal deductive semantics and abstract system theories. The former has been modeled as the number of objects in the architecture of a program; and the latter has been described as the number of equivalent functions in the program. By introducing a generic program model as a set of embedded relational processes, ten fundamental basic control structures (BCS's) have been elicited and their cognitive weights have been quantitatively derived and calibrated via a series of psychological experiments. Based on this work, the cognitive complexity of software systems has been formally and accurately measured and analyzed. Robustness of the cognitive complexity measure has been analyzed with comparative case studies.

Acknowledgement

The author would like to acknowledge the Natural Science and Engineering Council of Canada (NSERC) for its support to this work.

References

[1] Albrecht, A.J. and J.E. Gaffney (1983), Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, *IEEE Transactions on Software Engineering*, Vol.9, No.6, pp.639-648.

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

[3] Halstead, M.H. (1977), *Elements of Software Science,* Elsevier North – Holland, New York.

[4] Hoare, C.A.R., I.J. Hayes, J. He, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin (1987), Laws of Programming, *Communications of he ACM*, Vol.30, No.8, August, pp. 672-686.

[5] Kearney, J.K., R.L. Sedlmeyer, W.B. Thompson, M.A. Gary and M.A. Adler (1986), *Software Complexity Measurement*, ACM Press, New York, Vol.28, pp. 1044-1050.

[6] McCabe, T.H. (1976), A Complexity Measure, *IEEE Trans. Software Eng.* SE-2(6), pp.308-320.

[7] Osgood, C. (1953), *Method and Theory in Experimental Psychology*, Oxford Univ. Press, UK.

[8] Wang, Y. (2006), On the Informatics Laws and Deductive Semantics of Software, *IEEE Transactions on Systems, Man, and Cybernetics (C),* Vol. 36, No.2, March, pp. 161-171.

[9] Wang, Y. (2006), *Software Engineering Foundations: A Transdisciplinary and Rigorous Perspective,* CRC Book Series in Software Engineering, Vol. 2, CRC Press, USA.

[10] Wang, Y. (2005), Keynote Speech: Psychological Experiments on the Cognitive Complexities of Fundamental Control Structures of Software Systems, *Proc. 4th IEEE International Conference on Cognitive Informatics* (ICCI'05), IEEE CS Press, Irvin, California, USA, August, pp. 4-5.

[11] Wang, Y. and J. Shao (2003), Measurement of the Cognitive Functional Complexity of Software, The 2nd IEEE International Conference on Cognitive Informatics (ICCI'03), IEEE CS Press, London, UK, August, pp.67-74.

[12] Wang, Y. (2003), On Cognitive Informatics, *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy,* Vol.4, No.2, pp.151-167.

[13] Wang, Y. (2003), Using Process Algebra to Describe Human and Software Behaviors, *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy,* Vol.4, No.2, pp.199-213.

[14] Wang, Y. (2002), The Real-Time Process Algebra (RTPA), *Annals of Software Engineering: An International Journal,* Vol. 14, USA, Oct., pp.235-274.

[15] Wang, Y. (1996), A New Sorting Algorithm: Self-Indexed Sort, *ACM SIGPLAN*, Vol. 31, No.3, March, USA, pp. 28-36.

[16] Wang, Y. (2006), On the Big-R Notation for Describing Iterative and Recursive Behaviors, *Proc. 5th IEEE International Conference on Cognitive Informatics (ICCI'06),* IEEE CS Press, Beijing, China, July.

[17] Wang, Y. (2006), A Unified Mathematical Model of Programs, *Proc. 2006 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'06),* Ottawa, Canada, May, pp. 2346-2349.